SAPIENZA
UNIVERSITÀ DI ROMA

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

Programmazione di Sistemi Embedded e Multicore

Autore:
Simone Lidonnici

16 dicembre 2024

Indice

1 Programmazione parallela **1**
- 1.1 Perché usare la programmazione parallela . . . 1
- 1.2 Scrivere programmi paralleli . . . 1
  - 1.2.1 Tipi di parallelismo . . . 2
  - 1.2.2 Come scrivere programmi paralleli . . . 2
- 1.3 Tipi di sistemi paralleli . . . 3
  - 1.3.1 Concorrenti vs Paralleli vs Distribuiti . . . 3
- 1.4 Architettura di Von Neumann . . . 4

2 MPI **5**
- 2.1 Comunicatori . . . 6
- 2.2 Messaggi . . . 6
  - 2.2.1 Modalità di comunicazione Point-to-Point . . . 7
  - 2.2.2 Comunicazione non bloccante . . . 8
- 2.3 Comunicazioni collettive . . . 10
  - 2.3.1 Comunicazioni collettive su array . . . 11
  - 2.3.2 Comunicazioni collettive su matrici . . . 12
- 2.4 Datatype custom . . . 14
- 2.5 Thread in MPI . . . 15
  - 2.5.1 Threading level in MPI . . . 15

3 Architettura multicore e ottimizzazioni **16**
- 3.1 Come progettare un programma parallelo . . . 16
  - 3.1.1 Pattern di tipo GPLS . . . 17
  - 3.1.2 Pattern di tipo GSLP . . . 17
- 3.2 Calcolo delle performance . . . 18
  - 3.2.1 Statistiche sui tempi . . . 18
  - 3.2.2 Strong e Weak scaling . . . 19
- 3.3 Sistemi a memoria distribuita . . . 20
  - 3.3.1 Organizzazione della memoria . . . 20
- 3.4 Cache . . . 20
  - 3.4.1 Consistenza nelle cache . . . 20
  - 3.4.2 Cache multicore . . . 21
- 3.5 Tool utili per ottimizzare . . . 22

4 Pthreads **23**
- 4.1 Threads . . . 23
  - 4.1.1 Creare un thread . . . 23
  - 4.1.2 Pinnare un thread . . . 25
- 4.2 Sezioni critiche di codice . . . 25
  - 4.2.1 Busy-Waiting . . . 26
  - 4.2.2 Mutex . . . 26
  - 4.2.3 Problemi con la gestione di sezioni critiche . . . 27
  - 4.2.4 Semafori . . . 27

4.2.5 Barriere . . . 28
4.2.6 Variabili condizionali . . . 29
4.2.7 Read Write Lock . . . 30
4.2.8 Thread-safety . . . 31

5 OpenMP **32**
5.1 Pragma . . . 32
5.1.1 Compilatori non supportanti OpenMP . . . 33
5.2 Scope delle variabili . . . 33
5.2.1 Clausola di riduzione . . . 34
5.2.2 Modificare lo scope di una variabile . . . 34
5.3 Cicli for parallelizzabili . . . 35
5.3.1 Loop annidati . . . 37
5.3.2 Schedulare i loop . . . 37
5.4 Dipendenza tra dati . . . 38
5.4.1 Rimozione delle dipendenze RAW . . . 39
5.4.2 Rimozione delle dipendenze WAR . . . 43
5.4.3 Rimozione delle dipendenze WAW . . . 43
5.5 Clausole di sincronizzazione . . . 44

6 CUDA **45**
6.1 Architettura di una GPU . . . 45
6.1.1 Differenze tra CPU e GPU . . . 45
6.1.2 Disposizione dei core . . . 45
6.2 Programmazione su GPU . . . 45
6.3 Schedulazione dei therad in CUDA . . . 46
6.3.1 Modello di esecuzione di CUDA . . . 46
6.3.2 Scrivere un programma con CUDA . . . 47
6.3.3 Scheduling dei thread . . . 48
6.4 Gestione della memoria con CUDA . . . 48
6.4.1 Tipi di memoria . . . 50
6.5 Stima delle performance . . . 51

1

Programmazione parallela

1.1 Perché usare la programmazione parallela

Dal 1986 al 2003 le performance dei microprocessori aumentavano di circa il 50% l'anno, dal 2003 questo aumento è diminuito fino ad arrivare al 4% l'anno. Questa diminuzione è causata dal fatto che l'aumento delle performance dipende dalla densità dei transistor, che diminuendo di grandezza generano più calore e questo li fa diventare inaffidabili. Per questo conviene avere più processori nello stesso circuito rispetto ad averne uno singolo più potente.

1.2 Scrivere programmi paralleli

Per utilizzare al meglio i diversi processori, bisogna volutamente scrivere il programma in modo da usare il parallelismo, in alcuni casi è possibile convertire un programma sequenziale in uno parallelo ma solitamente bisogna scrivere un nuovo algoritmo.
Esempio:
Computare n valori e sommarli tra loro.
Soluzione sequenziale:

```
sum=0;
for(i=0;i<n;i++){
    x=ComputeValue(...);
    sum+=x;
}
```

Soluzione parallela:
Avendo p core ognuno eseguirà $\frac{n}{p}$ somme

```
sum=0;
start=...; // definiti in base al core
end=...;   // che esegue il programma
for(i=start;i<end;i++){
    x=ComputeValue(...);
    sum+=x;
}
```

Dopo che ogni core avrà finito la sua somma, per ottenere la somma totale si può designare un core come **master core**, a cui tutti gli altri core invieranno la propria somma e che eseguirà la somma totale.

Questa soluzione però non è ottimale perché il master core esegue una ricezione e una somma per ogni altro core ($p-1$ volte), mentre gli altri sono fermi. Per migliorare l'efficienza si sommano a coppie i risultati di ogni core:

- il core 0 somma il risultato con il core 1, il core 2 con il core 3 ecc...
- si ripete con solo i core 0, 2 ecc...
- si continua creando uno schema ad albero binario

Questo secondo metodo è molto più efficiente perché il numero di ricezioni e somme che esegue il core che ottiene il risultato finale sono $\log_2(p)$.

1.2.1 Tipi di parallelismo

Ci sono due principali tipi di parallelismo:

- **Task parallelism**: Diversi task vengono divisi tra i vari core e ogni core esegue operazioni diverse su tutti i dati (il parallelismo temporale nei circuiti è un tipo di task parallelism)
- **Data parallelism:** Vengono divisi i dati tra i core e ogni core esegue operazioni simili su porzioni di dati diverse (il parallelismo spaziale nei circuiti è un tipo di task parallelism)

Esempio:
Bisogna correggere 300 esami ognuno con 15 domande e ci sono 3 professori disponibili.
In questo caso:

- Data parallelism:
  - Ogni assistente corregge 100 esami
- Task parallelism:
  - Il primo professore corregge le domande 1-5 di tutti gli esami
  - Il secondo professore corregge le domande 6-10 di tutti gli esami
  - Il terzo professore corregge le domande 11-15 di tutti gli esami

1.2.2 Come scrivere programmi paralleli

Se i core possono lavorare in modo indipendente scrivere un programma parallelo è molto simile a scriverne uno sequenziale, in pratica però i core devono comunicare, per diversi motivi:

- **Comunicazione:** un core deve inviare dei dati ad un altro
- **Bilanciamento del lavoro:** bisogna dividere il lavoro in modo equo per far si che un core non sia sovraccaricato rispetto agli altri, sennò tutti i core aspetterebbero il più lento
- **Sincronizzazione:** ogni core lavora a velocità diversa e bisogna controllare che uno non vada troppo avanti rispetto agli altri

Per creare programmi paralleli useremo 4 diverse estensioni di C:

1. **Message-Passing Interface (MPI)**: Libreria
2. **Posix Threads (Pthreads)**: Libreria
3. **OpenMP**: Libreria e Compilatore
4. **CUDA**: Libreria e Compilatore

1.3 Tipi di sistemi paralleli

I sistemi paralleli possono essere divisi in base alla divisione della memoria:

- **Memoria Condivisa:** Tutti i core hanno accesso alla memoria del computer e si coordinano modificando locazioni di memoria condivisa
- **Memoria Distribuita:** Ogni core possiede una propria memoria e per coordinarsi devono mandarsi dei messaggi

Possono essere anche divisi in base al numero di **Control Unit**:

- **Multiple-Instruction Multiple-Data (MIMD):** Ogni core ha la sua CU e può lavorare indipendentemente dagli altri
- **Single-Instruction Multiple-Data (SIMD):** I core condividono una sola CU e devono eseguire tutti le stesse operazioni o stare fermi

Le libreria nominate precedentemente si collocano in una tabella:

	SIMD	MIMD
Memoria Condivisa	CUDA	Pthreads OpenMP CUDA
Memoria Distribuita		MPI

1.3.1 Concorrenti vs Paralleli vs Distribuiti

I sistemi possono essere:

- **Concorrenti:** più task possono essere eseguite in ogni momento, possono essere anche sequenziali
- **Paralleli:** più task cooperano per risolvere un problema comune, i core sono condividono la memoria o sono connessi tramite un network veloce
- **Distribuiti:** un programma che coopera con altri programmi per risolvere un problema comune, i core sono connessi in modo più lento

I sistemi paralleli e distribuiti sono anch'essi concorrenti.

1.4 Architettura di Von Neumann

Per poter scrivere codice efficiente bisogna conoscere l'architettura su cui si sta eseguendo il codice ed ottimizzarlo per essa.
L'architettura di Von Neumann è composta da:

- **Memoria principale:** Insieme di locazioni, ognuna con un indirizzo e del contenuto (dati o istruzioni)
- **CPU/Processore/Core:** Control Unit, che decide le istruzioni da eseguire, e **datapath**, che eseguono le istruzioni. Lo stato di un programma in esecuzione viene salvato nei registri, un registro molto importante è il **Program Counter (PC)** dove viene salvato l'indirizzo della prossima istruzione da eseguire
- **Interconnessioni:** Usate per trasferire dati tra CPU e memoria, tradizionalmente con un bus

Una macchina di Von Neumann esegue un istruzione alla volta, operando su piccole porzioni di dati contenuti nei registri. La CPU può leggere (fetch) dati dalla memoria o scriverci (store), la separazione tra CPU e memoria è conoscuta come **Bottleneck di Von Neumann**, cioè le interconnessioni determinano la velocità con cui i dati vengono trasferiti.

2
MPI

La programmazione parallela tramite **MPI** utilizza un approccio **Single-Program Multiple-Data (SPMD)** in cui si compila un solo programma eseguito da più processi e tramite degli if-else si specifica quale processo deve eseguire quale parte di codice. I processi non hanno memoria condivisa quindi comunicano tramite **messaggi**.
Per poter utilizzare MPI bisogna importare l'header `mpi.h` che permetterà di utilizzare le funzioni MPI, per esempio:

- `MPI_Init`: esegue il setup necessario

```
int MPI_Init(
    int* argc_p,    // puntatore al numero di argomenti
    char*** argv_p  // puntatore agli argomenti in input
);
```

- `MPI_Finalize`: termina la parte multiprocesso del programma e dealloca la memoria utilizzata

```
int MPI_Finalize(void);
```

Per compilare il programma si utilizza:

```
mpicc -g -Wall file.c -o exe
// g usato per debbuging e Wall per avere i warning
```

Per eseguirlo invece:

```
mpiexec --oversubscribe -n 4 ./exe
// n indica il numero di processi creati
/* oversubscribe permette di creare n processi
anche su una macchina con meno di n core*/
```

2.1 Comunicatori

Un **comunicatore** è un insieme di processi che può scambiarsi messaggi, `MPI_Init` crea un comunicatore ganerico che comprende tutti i processi chiamato `MPI_COMM_WORLD`. All'interno di un comunicatore ogni processo ha un suo **rank** che lo identifica, che va da 0 a n-1 per un comunicatore con n processi. Alcune funzioni utili che riguardano i comunicatori sono:

- `MPI_Comm_size`: restituisce il numero di processi nel comunicatore

```
int MPI_Comm_size(
    MPI_Comm comm,    // comunicatore in input
    int* comm_sz_p    // variabile di output
);
```

- `MPI_Comm_rank`: restituisce il rank del processo chiamante all'interno del comunicatore

```
int MPI_Comm_rank(
    MPI_Comm comm,    // comunicatore in input
    int* my_rank_p    // variabile di output
);
```

L'ordine con cui vengono eseguiti i processi è casuale, quindi il processo con rank 1 potrebbe terminare prima del processo con rank 0.

2.2 Messaggi

Per poter comunicare tra loro i processi devono scambiarsi dei messaggi tramite due funzioni, `MPI_Send` e `MPI_Recv`. I messaggi devono essere **nonovertaking**, cioè se un mittente manda due messaggi ad un destinatario, l'ordine deve essere mantenuto.

- `MPI_Send`:

```
int MPI_Send(
    void* msg_buf_p,  // puntatore ai dati da inviare
    int msg_size,     // numero di elementi da inviare
    MPI_Datatype msg_type,  // tipo di dati da inviare
    int dest,  // rank del destinatario nel comunicatore
    int tag,   // tag opzionale
    MPI_Comm comm   // comunicatore
);
/* msg_size non va scritto in byte,
ma indica proprio il numero di elementi */
```

- `MPI_Recv`:

```
int MPI_Recv(
    void* msg_buf_p,  // variabile in output
    int msg_size,     // numero di elementi da ricevere
    MPI_Datatype msg_type,  // tipo di dati da ricevere
    int source,  // rank del mittente nel comunicatore
    int tag,   // tag opzionale
    MPI_Comm comm,  // comunicatore
    MPI_Status* status_p   // status dell'operazione
);
/* msg_size non va scritto in byte,
ma indica proprio il numero di elementi */
```

Per quanto riguarda i datatype, di base esiste un datatype per ogni tipo di c, però possono esserne creati di nuovi nel caso si voglia inviare strutture o tipi particolari. I tag invece servono per differenziare dei messaggi mandati alla stessa destinazione, per esempio se abbiamo due tipologie di messaggi con scopi diversi.
Un messaggio inviato dal processo `q` al processo `r` verrà ricevuto correttamente se:

- il comunicatore di `q.MPI_Send` e `r.MPI_Recv` è lo stesso
- la destinazione di `q.MPI_Send` è r e il mittente di `r.MPI_Recv` è q (si può omettere il mittente in `r.MPI_Recv` inserendo come parametro `MPI_ANY_SOURCE`)
- il datatype di `q.MPI_Send` e `r.MPI_Recv` è lo stesso
- il tag di `q.MPI_Send` e `r.MPI_Recv` è lo stesso (si può omettere in `r.MPI_Recv` inserendo come parametro `MPI_ANY_TAG`)
- il numero di elementi inviati da `q.MPI_Send` è minore di quelli ricevuti da `r.MPI_Recv` (si può omettere la grandezza in `r.MPI_Recv`)

Il parametro `MPI_Status` da informazioni sull'operazione di ricezione del messaggio se non si è specificato il mittente, il tag oppure la grandezza del messaggio. In quest'ultimo caso è possibile usare la funzione `MPI_Get_count` con input lo status e il datatype ricevuto per ottenere la quantità di dati che si è ricevuta dal messaggio.

2.2.1 Modalità di comunicazione Point-to-Point

`MPI_Send` usa la modalità di comunicazione **standard**, cioè che decide autonomamente in base alla grandezza del messaggio se bloccare la chiamata, aspettando che ci sia qualche processo pronto abbia ricevuto il messaggio, oppure ritornare prima di ricevere la conferma di ricezione, copiando in un buffer temporaneo il messaggio. Il primo metodo viene usato se il messaggio è molto grande e non si può allocare un buffer di quelle dimensione, questo fa diventare `MPI_Send` **localmente bloccante**.

Ci sono altre tre modalità possibili (tra parentesi il nome della funzione che usa questa modalità):

- **Buffered** (`MPI_Bsend`): in questa modalità l'operazione è sempre localmente bloccante e ritornerà non appena il messaggo verrà copiato sul buffer, che deve essere fornito dall'utente.
- **Sincrona** (`MPI_Ssend`): in questa modalità l'operazione è globalmente bloccante e ritorna solo dopo che il messaggio è stato ricevuto dal destinatario. Permette al mittente di sapere a che punto è il ricevente.
- **Ready** (`MPI_Rsend`): in questa modalità l'invio ha esito positivo solo se il ricevente è pronto a ricevere sennò fallisce ritornando un errore. Permette di ridurre il numero di operazioni di handshaking.

Se ci trovassimo in una situazione in cui due processi devono sia inviare che ricevere dei dati, se entrambi chiamassero prima la `MPI_Send` e poi la `MPI_Recv` ci sarebbe il rischio di deadlock (risolvibile anche con le send non blccanti spiegate al paragrafo successivo). Per questo esiste una funzione che permette sia di ricevere che di mandare dati contemporaneamente, senza il rischio di deadlock, chiamata `MPI_Sendrecv`:

```
int MPI_Sendrecv(
    void* send_buf_p,
    int send_buf_size,
    MPI_Datatype send_buf_type,
    int dest,
    int send_tag,
    void* recv_buf_p,
    int recv_buf_size,
    MPI_Datatype recv_buf_type,
    int source,
    int recv_tag,
    MPI_Comm communicator,
    MPI_Status* status_p
);
```

2.2.2 Comunicazione non bloccante

Le comunicazioni bloccanti sono considerate poco performanti perché il mittente potrebbe bloccarsi, le comunicazioni non bloccanti invece permettono di massimizzare la concorrente e processare altro mentre si inviano o ricevono dati. Il difetto è che per sapere se un'operazione è stata completata o no bisogna chiederlo esplicitatamente, nel mittente per sapere se possiamo riutilizzare il buffer del messaggio, nel ricevente per sapere se possiamo iniziare a processare il messaggio ricevuto. Le comunicazioni non bloccanti posso essere accoppiate con qualunque delle modalità di comunicazione: `MPI_ISend`, `MPI_Ibsend`, `MPI_Issend` e `MPI_Irsend`.

Le comunicazioni non bloccanti esistono sia con `MPI_Isend` che con `MPI_Irecv`:

- `MPI_Isend`:

```
int MPI_Send(
    void* msg_buf_p,  // puntatore ai dati da inviare
    int msg_size,     // numero di elementi da inviare
    MPI_Datatype msg_type,  // tipo di dati da inviare
    int dest,  // rank del destinatario nel comunicatore
    int tag,   // tag opzionale
    MPI_Comm comm,   // comunicatore
    MPI_Request *req  // output
);
```

- `MPI_Irecv`:

```
int MPI_Recv(
    void* msg_buf_p,  // variabile in output
    int msg_size,     // numero di elementi da ricevere
    MPI_Datatype msg_type,  // tipo di dati da ricevere
    int source,  // rank del mittente nel comunicatore
    int tag,   // tag opzionale
    MPI_Comm comm,  // comunicatore
    MPI_Request* *req   // output
);
```

Viene aggiunto nella `MPI_Isend`, e sostituito a `MPI_Status` nella `MPI_Irecv`, un parametro `MPI_Request` che serve per controllare se l'operazione è finita dopo la chiamata della funzione. Questo controllo si può fare con diverse funzioni:

- `MPI_Wait`: aspetta fino a quando la comunicazione non è terminata

```
int MPI_Wait(
    MPI_Request *req  // request della comunicazione
    MPI_Status   // variabile in output
)
```

- `MPI_Test`: controlla se la comunicazione è terminata e ritorna una flag con valore 0 o 1

```
int MPI_Wait(
    MPI_Request *req  // request della comunicazione
    int *flag    // flag che indica il completamento
    MPI_Status   // variabile in output
)
```

Esistono anche altre funzioni che prendono in input una lista di `MPI_Request` come:

- WaitAll
- TestAll
- WaitAny
- TestAny

2.3 Comunicazioni collettive

Le comunicazioni collettive sono comunicazioni che permettono a tutti i processi all'interno di un comunicatore di comunicare insieme, ciò permette di risparmiare tempo e deadlock vari. Bisogna seguire delle regole quando si usano le comunicazioni collettive:

- Tutti i processi devono chiamare la stessa funzione collettiva
- Il datatype deve essere uguale in tutti i processi e anche il processo di destinazione
- Le comunicazioni collettive non hanno tag quindi vengono eseguite in ordine in base a come vengono chiamate

Una funzione collettiva è `MPI_Reduce`, che aggrega i dati secondo un'operazione specificata di tipo `MPI_Op`. Sono già implementate le operazioni basilari come somma, or, xor ecc... ma possono anche esserne create di nuove.

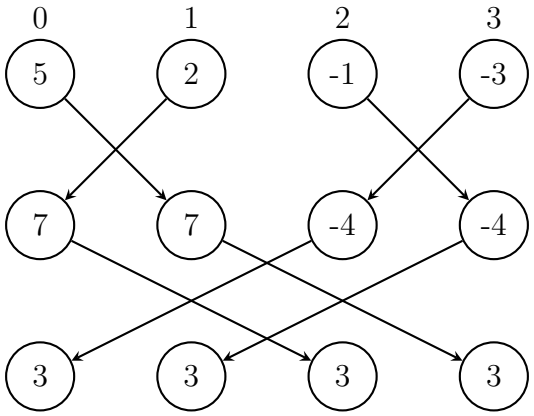
```
int MPI_Reduce(
    void* input_data_p  // dati in input
    void* output_data_p  // dati in output
    int count  // numero di dati in input
    MPI_Datatype  // tipo dei dati in input
    int dest_process // rank del processo con il risultato finale
    MPI_Comm  // comunicatore
)
```

Il puntatore al buffer dove verranno messi i dati di ouput va specificato in ogni processo (può essere `NULL`) anche quelli che non avranno il risultato finale.
Un'altra funzione collettiva è `MPI_Bcast`, che permette a un singolo processo di inviare dei dati a tutti gli altri nel comunicatore.

```
int MPI_Bcast(
    void* data_p  // dati in input o otput
    int count  // numero di dati in input
    MPI_Datatype  // tipo dei dati in input
    int source_process // rank del processo che invia i dati
    MPI_Comm  // comunicatore
)
```

Esiste anche una funzione che unisce `MPI_Reduce` e `MPI_Bcast` chiamata `MPI_Allreduce`, che aggrega i dati secondo un'operazione e poi manda il risultato a tutti i processi. Se fosse implementata effettivamente come una Reduce e poi una Bcast sarebbe molto inefficiente, quindi viene implementata con un pattern a farfalla.

Ad esempio una somma verrebbe effettuata con il pattern:



2.3.1 Comunicazioni collettive su array

Per un vettore di un processo in modo equo tra tutti i processi di un comunicatore, possiamo usare la funzione `MPI_Scatter` che permette di decidere quanti elementi mandare ad ognuno degli altri processi:

```
int MPI_Scatter(
    void* send_buf_p, // utile solo nel processo che invia
    int send_count,
    MPI_Datatype send_type,
    void* recv_buf_p,
    int recv_count,
    MPI_Datatype recv_type,
    int src_proc,   // rank del processo che invia
    MPI_Comm communicator
);
/* send_count e recv_count indicano il numero
di elementi da inviare e ricevere per ogni processo,
non il numero totale */
```

Gli elementi vengono mandati in ordine di rank, cioè se invio ad ogni processo 3 elementi il rank 0 avrà gli elementi 0-2, il rank 2 gli elementi 3-5 ecc..., e il processo che invia avrà comunque gli elementi che gli spettano nel buffer di ricezione. Ad esempio se chiamo la funzione:

```
MPI_Scatter(buff,3,MPI_INT,dest,3,MPI_INT,0,MPI_COMM_WORLD);
```

e il processo con rank 0 ha nel buffer:

Rank 0	0	1	2	3	4	5	6	7	8

dopo la chiamata della funzione i buffer di destinazione dei processi avranno:

Rank 0	0	1	2
Rank 1	3	4	5
Rank 2	6	7	8

Si può sostituire nel processo che invia il vettore il buffer di destinazione con `MPI_IN_PLACE` per far mettere la parte di vettore del processo chiamante nello stesso buffer che contiene il vettore completo, sovrascrivendolo.
Per unire poi i sottovettori posseduti da ogni processo in un unico vettore, possiamo usare la funzione `MPI_Gather`:

```
int MPI_Gather(
    void* send_buf_p,
    int send_count,
    MPI_Datatype send_type,
    void* recv_buf_p, // utile solo nel processo che riceve
    int recv_count,
    MPI_Datatype recv_type,
    int dest_proc,   // rank del processo che riceve
    MPI_Comm communicator
);
/* send_count e recv_count indicano il numero
di elementi da inviare e ricevere per ogni processo,
non il numero totale */
```

Anche in questo caso i sottovettori vengono uniti in ordine in base al rank, cioè i primi elementi saranno del rank 0 i secondi del rank 1 ecc.... Esiste anche una funzione che permette di fare `MPI_Gather` e `MPI_Bcast` contemporaneamente, chiamata `MPI_Allgather` (implementata in un modo più efficente rispetto a una gather e poi una broadcast).

2.3.2 Comunicazioni collettive su matrici

Una matrice $n \times m$ allocata in modo statico viene memorizzata in memoria come un vettore con nm elementi ordinati per righe, quindi per eseguire delle comunicazioni collettive (ma anche send e recv) basta specificare un numero di elementi di $n \cdot m$. Per allocarle dinamicamente possiamo fare in due modi:

- Lista di puntatori a righe:

```
int** m;
m = (int**) malloc(sizeof(int*)*n_righe);
for(int i=0;i<n_righe,i++){
    a[i] = (int*) malloc(sizeof(int)*n_col);
}
```

- Array monodimensionale:

```
int* m = (int*) malloc(sizeof(int)*n_col*n_righe);
```

Il primo metodo ci permette di accedere ad un elemento nella riga i e nella colonna j tramite `m[i][j]`, ma non ci permette di inviarle tramite una sola comunicazione, ma bisogna eseguirne una per ogni riga.
Con il secondo metodo per accedere ad un elemento nella riga i e nella colonna j dobbiamo scrivere `m[i*n_col+j]`, però ci permette di inviare la matrice con una sola comunicazione di $n \cdot m$ elementi, risparmiando tempo.

Altre funzioni che possono essere utili per vettori e matrici sono:

- **Reduce-Scatter**: esegue l'operazione specificata sui vettori dei processi e poi divide il vettore risultante tra i processi
- **Alltoall**: permette di dividere i vettori dei processi combinandoli, cioè ogni processo avrà all'inizio del suo vettore risultante una parte del vettore del processo con rank 0 (molto utile per fare la trasposizione di matrici). Per esempio se i processi hanno i vettori:

Rank 0	0	1	2
Rank 1	3	4	5
Rank 2	6	7	8

  dopo la chiamata della funzione i buffer di destinazione dei processi avranno:

Rank 0	0	3	6
Rank 1	1	4	7
Rank 2	2	5	8

2.4 Datatype custom

I datatype custom permettono di rappresentare un insieme di dati in memoria, sapendo sia il tipo degli elementi sia la loro posizione relativa in memoria. In questo modo sia la funzione che invia i dati che la funzione che li riceve sanno come posizionarli correttamente in memoria, sennò potrebbe succedere che in base a diverse versioni di sistema o di compilatore vengano immagazzinati in modo diverso. In pratica un datatype custom è un insieme di datatype con il loro offset relativo (in base al primo elemento).
La funzione per creare un nuovo Datatype è `MPI_Type_create_struct`:

```
int MPI_Type_create_struct(
    int count, // numero di elementi
    int array_lengths[], // lunghezza degli elementi
    MPI_Aint array_offset,  // offset relativo degli elementi
    MPI_Datatype array_types[], // tipo degli elementi
    MPI_Datatype* new_type_p  // output del nuovo tipo
);
```

Prima di usarla bisogna però chiamare la funzione `MPI_Type_commit`.
Esempio:
Data la struct:

```
typedef struct T{
    float a;
    float b;
    int n;
};
```

con indirizzi delle variabili:

Variabile	Indirizzo
a	24d
b	40d
n	48d

possiamo definire un Datatype custom per questa struttura con la funzione:

```
MPI_Type_create_struct(
    count = 3,
    array_length = [1,1,1],
    array_offset = [0,16,24],
    array_types = [MPI_DOUBLE,MPI_DOUBLE,MPI_INT],
    new_type_p
);
```

Solitamente l'indirizzo di una variabile `a` è uguale al suo puntatore (`&a`), ma in alcuni casi potrebbero differire, quindi per sicurezza conviene usare la funzione `MPI_Get_address` per ottenere l'indirizzo di una determinata variabile dal suo puntatore. Quando si è finito di usare il nuovo datatype definito si può liberare la memoria con `MPI_Type_free`.

2.5 Thread in MPI

In MPI è possibile utilizzare i thread, per farlo all'inizio del programma bisogna chiamare invece di `MPI_Init` la funzione `MPI_Init_thread`, che ha due parametri aggiuntivi, uno in input che indica il **threading level** che si richiede al programma e uno in output che ritorna il threading level supportato (se il livello specificato in input è supportato saranno uguali, ma alcune implementazioni di MPI potrebbero non supportare tutti i livelli).
Per capire a pieno la sezione sottostante, leggere prima la sezione sulle funzioni thread-safe nel capitolo Pthread.

2.5.1 Threading level in MPI

MPI supporta diversi threading level:

- `MPI_THREAD_SINGLE`: i processi non possono usare thread, rendendo la funzione equivalente a `MPI_Init`
- `MPI_THREAD_FUNNELED`: i processi possono creare più thread a solo il thread principale di ogni processo può chiamare funzioni MPI. Utile nel caso di parallelismo fork-join.
- `MPI_THREAD_SERIALIZED`: i processi possono creare più thread, ma solo uno alla volta può chiamare una funzione MPI. Il processo si assicura che le chiamate MPI vengano eseguite in maniera thread-safe.
- `MPI_THREAD_MULTIPLE`: i processi possono creare più thread e qualsiasi thread può chiamare una funzione MPI in ogni momento. La libreria MPI si assicura che l'accesso sia sicuro tra tutti i thread, ma questo rende le operazioni MPI meno efficienti.

3

Architettura multicore e ottimizzazioni

3.1 Come progettare un programma parallelo

Per progettare un programma parallelo si utilizza la metodologia di Foster, che consiste in 4 fasi:

1. **Partizionamento:** Identificare delle task che possono essere eseguite in parallelo (che non hanno dipendenze da altre task)
2. **Comunicazione:** Daterminare quali dati devono scambiarsi i diversi task
3. **Agglomerazione o aggregazione:** Raggruppare i singoli task in task più grandi, per giustificare il costo della creazione di un nuovo processo
4. **Mapping:** Assegnare i task composti ai vari processi per minimizzare le comunicazioni necessarie

Esempio:
Se si hanno in input dei numeri decimali e si vuole creare un istrogramma per sapere quanti numeri sono contenuti in ogni intervallo intero di tipo `[i,i+1]`, il modo migliore di parallelizzare il programma con n processi, sarebbe dividere i dati in input in p parti e creare un istogramma locale in ogni processo, sommandoli tutti alla fine.

Possiamo dividere i pattern di programmazione parallela in due grandi categorie:

- **Globalmente Parallela, Localmente Sequenziale (GPLS):** il programma può svolgere diverse task in modo concorrente, con ogni task eseguito in maniera sequenziale. Alcuni pattern in questa categoria sono:
  - Single Program, Multiple Data (SPMD)
  - Multiple Program, Multiple Data (MPMD)
  - Master Worker
  - Map reduce
- **Globalmente Sequenziale, Localmente Parallela (GSLP):** il programma viene eseguito come un programma sequenziale, con alcune parti eseguite in parallelo. Alcuni pattern in questa categoria sono:
  - Fork/Join
  - Loop parallelism

3.1.1 Pattern di tipo GPLS

- **Single Program, Multiple Data (SPMD):** I programmi SPMD tengono tutta la logica in un singolo programma, un tipico esempio di come questo tipo di programmi funziona:

  1. Inizializzazione
  2. Si ottiene un ID unico: numerati da 0 che identificano i thread o processi usati. Alcuni sistemi, tipo CUDA, usano vettori come identificatori
  3. Esecuzione: ogni processo eseguire parti di codice diversi in base al suo ID
  4. Terminazione: si libera lo spazio e si salvano i risultati

- **Multiple Program, Multiple Data (MPMD):** nei casi in cui la memoria richiesta per tutti i processi sia troppa oppure si utilizzano multiple piattaforme diverse, in cui SPMD fallirebbe, si utilizza MPMD. Ha la stessa esecuzione di SPMD ma consiste nell'avere differenti programmi in base alle diverse piattaforme.

- **Master Worker:** i processi vengono divisi in due tipi: master e workers, il master deve:

  – Dare i dati per far lavorare i workers
  – Ottenere i risultati della computazione dai workers
  – Eseguire le operazioni di I/O, come accedere ad un file
  – Interagire con l'utente

  Questo tipo di pattern è utile per bilanciare il lavoro, perché non ci sono scambi di dati dai workers, ma il master potrebbe causare bottleneck (risolvibile creando una gerarchia di master)

- **Map reduce:** variazione del pattern Master Worker, usato dal motore di ricerca di Google, in cui il master coordina tutta l'operazione, i workers eseguono due tipi di task:

  – Map: applicare una funzione ai dati, dividendoli in set di risultati parziali
  – Reduce: ottenere i risultati parziali e ne crearne uno finale

3.1.2 Pattern di tipo GSLP

- **Fork/Join:** all'inizio dell'esecuzione c'è un singolo processo o thread padre, che creerà figli in modo dinamico durante l'esecuzione o userà un pool di thread statico che eseguiranno le task. I figli devono tutti aver finito per far continuare l'esecuzione al padre. Viene usato da OpenMP/Pthread.

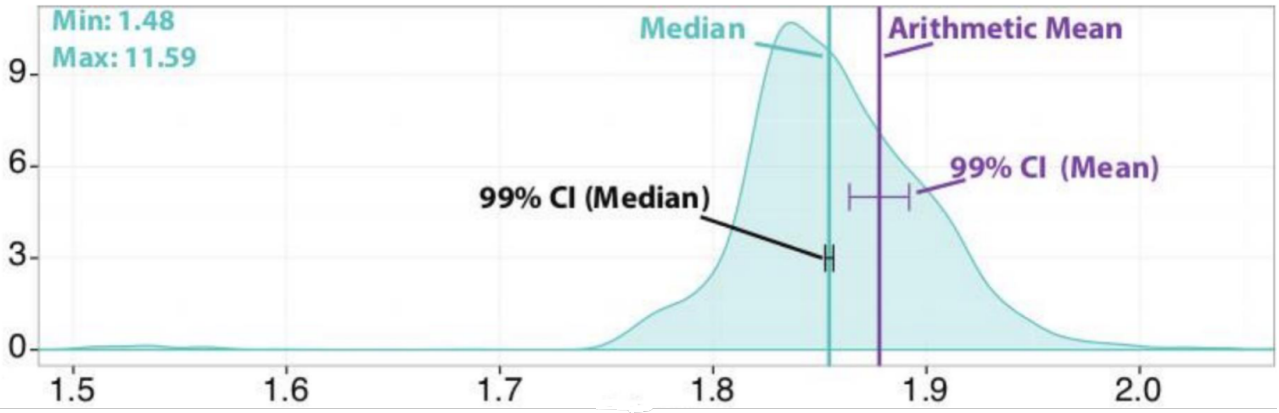
- **Loop parallelism:** usato per trasformare codice sequenziale in codice multiprocesso, si esegue rompendo i cicli loop in sottocicli indipendenti. I loop però devono avere una particolare forma per supportare questo pattern.

3.2 Calcolo delle performance

Per sapere quanto è efficente un programma parallelo bisogna innanztutto sapere quanto tempo ci mette ad essere eseguito, per fare ciò esiste una funzione `MPI_Wtime` che ritorna il tempo attuale (in base al tempo di accensione della macchina o altre variabili interne). Per sapere il tempo di esecuzione di un programma chiamiamo `MPI_Wtime` all'inizio e alla fine dell'esecuzione e facciamo la differenza. Essendo però un programma parallelo ogni processo calcolerà il proprio tempo, bisogna quindi prendere il maggiore, tramite una `MPI_Allreduce`.
I vari processi però potrebbero iniziare in momenti diversi, per assicurarsi che i processi inizino insieme esiste una funzione chiamata `MPI_Barrier`, che una volta chiamata da un processo, non permette di andare avanti finché tutti i processi non hanno chiamato la funzione. Questo non assicura che i processi usciranno dalla funzione contemporaneamente, ma solo che il primo processo uscirà quando ttti ci saranno entrati (è comunque una stima abbastanza buona).

3.2.1 Statistiche sui tempi

Una sola misurazione non è abbastanza per sapere quanto è efficiente il programma, è meglio misurare più volte e riportare un grafico della distribuzione dei tempi.
Esempio:



Speedup

Dato il tempo del programma seriale $T_s(n)$ e il tempo del programma parallelo con p processi $T_p(n,p)$, definiamo lo **speedup** come:

$$S(n,p) = \frac{T_s(n)}{T_p(n,p)}$$

Il valore ideale sarebbe $S(n,p) = p$, in questo caso si dice che il programma ha **speedup lineare**.

Da notare che $T_s(n)! = T_p(n,1)$, e solitamente $T_s(n) \leq T_p(n,1)$. Questa seconda tempistica $T_p(n,1)$ si usa per calcolare la scalabilità, cioè:

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

Efficienza

L'**efficienza** ci dice quanto il programma parallelo spreca risorse rispetto al programma sequenziale:

$$E(n,p) = \frac{T_s(n)}{T_p(n,p) \cdot p}$$

Il valore ideale sarebbe $E(n,p) = 1$, cioè con p processi il programma va p volte più veloce.

3.2.2 Strong e Weak scaling

Ci sono due ipi di scaling che un programma può avere:

- **Strong scaling:** quando fissata una dimensione del problema, aumentando il numero di processi il programma ha un'efficienza vicina a 1.
- **Weakscaling:** quando aumentando la dimensione del problema in modo parallelo rispetto al numero di processi il programma ha un'efficienza vicina a 1.

Legge di Amdahl

La legge di Amdahl ci permette di stimare quale sarà lo speedup massimo del nostro programma, in base alla quantità di programma che possiamo parallelizzare, data α la percentuale di programma che si può parallelizzare:

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

Da questo possiamo calcolare lo speedup:

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

Ponendo l'equazione con p tendente a infinito:

$$\lim_{p\to+\infty} S(n,p) = \frac{1}{1-\alpha}$$

Questa legge da un massimo speedup solo nel caso di strong scaling.

Legge di Gustafson

La legge di Gustafson ci permette di stimare lo speedup massimo del nostro programma considerando il weak scaling, cioè all'aumentare dei processi aumenterà anche la percentuale α di parte parallelizzabile:

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

Questo viene anche chiamato **speedup scalato**.

3.3 Sistemi a memoria distribuita

I sistemi a **memoria distribuita** sono formati diversi nodi (detti anche server o blade), interconnessi da una rete. Ogni nodo è formato da una CPU multicore, una o più GPU e delle interfaccie di rete (NIC) che connettono il nodo alla rete. Avendo ogni nodo la propria memoria, per parallelizzare l'intera serie di nodi conviene usare **MPI**, invece per parallelizzare i core all'interno della CPU di ogni noda conviene usare un approccio con memoria condivisa come **Pthreads** o **OpenMP**.

3.3.1 Organizzazione della memoria

Alcuni sistemi, detti **NUMA (Non-Uniform Memory Access)**, possiedono più di una memoria, ognuna collegata ad un gruppo di chip. Ovviamente accedere alla propria memoria "locale" è più veloce rispetto ad accedere ad una memoria "remota", quindi per migliorare ancora di più l'efficienza del programma si potrebbe specificare in quale memoria allocare i dati (di base le malloc non permettono di decidere dove allocare i dati), tramite la libreria `numa.h`. In verità si potrebbe ancore ottimizzare perchè alcuni core sono fisicamente più vicini alla NIC e quindi più veloci ad eseguire operazioni tra diversi nodi.

3.4 Cache

Una **cache** è un insieme di locazioni di memoria che vengono accedute più velocemente rispetto alla memoria principale della macchina. La cache è formata da una tecnoligia più potente ma più costosa (SRAM) rispetto alla DRAM e solitamente si trova sullo stesso chip (o in generale fisicamente vicino) alla CPU con cui lavora. Le cache per decidere cosa salvarsi nella propria memoria, assumono la località delle richieste, cioè richiedere una locazione di memoria presuppone che ne venga acceduta una vicina (**località spaziale**) e in un tempo ristretto (**località temporale**). Nelle cache i dati vengono quindi salvati in righe e quando un qualsiasi elemento della riga viene richiesto dalla memoria principale viene caricata in cache l'intera riga, questo perchè è più veloce trasferire 64 byte in una volta solare rispetto a trasferire 64 volte un solo byte. Le cache sono divise in livelli, ognuna più grande della preedente ma più lenta, quando viene richiesto un valore la CPU controllera prima la cache L1, poi la L2 e così via. Se il valore richiesto non si trova in nessuna cache allora viene richiesto dalla memmoria principale.

3.4.1 Consistenza nelle cache

Se la CPU modifica un valore nella cache, quello diventerà inconsistente rispeto a quello in memoria principale, per assicurarsi di salvare la modifica anche in memoria principale le cache utilizzano due approcci:

- **Write-through**: la cache ogni volta che un valore viene modificato al suo interno lo modifica anche nella memoria principale
- **Write-back**: la cache marca il valore modificato come "sporco" e quando la linea verrà rimpiazza in cache il valore "sporco" viene trascritto in memoria

3.4.2 Cache multicore

Nelle cache di sistemi multicore ogni core possiede una cache L1 dove salvare i dati. Questa cosa rischia di causare inconsistente nei casi in cui 2 core hanno entrambi in cache lo stesso valore e uno due lo modifica.
Esempio:

Core 1	Core 2
y0=x	y1=3*x
x=7	Istruzione non comprendente x
Istruzione non comprendente x	z1=4*x

In questo caso, in cui le variabili con un numero indicano il core che le ha privatamente e x è condivisa, il valore di x salvato nella cache del core 2 sarebbe ancora quello precedente all'assegnamento `x=7` del core 1.
Per risolvere questo problema di coerenza tra diverse cache si usano diversi metodi:

- **Snooping**: I core hanno un bus condiviso in cui ogni segnale inviato viene visto da tutti. Ogni volta che un core modifica un valore lo comunica sul bus, permettendo agli altri core di marcare la propria copia di x come invalida. Questa procedura è molto costosa, infatti non si usa quasi più, soprattutto su sistemi multicore con tanti core.
- **Directory based**: Viene usata una struttura chiamata directory che contiene lo status di ogni linea di cache (una bitmap o lista di tutti i core che hanno una copia di quella linea). Quando una variabile di una linea viene modificata, viene consultata la directory e la linea contenente la variabile viene marcata come invalida in tutte le altre cache.

Questo secondo metodo può causare dei problemi di **false sharing**, per cui anche se due core devono accedere a variabili diverse sulla stessa linea, ognuno invaliderà l'intera linea obbligando anche l'altro a richiedere i dati dalla memoria. Questo aumenta molto il tempo di esecuzione di un programma. Per risolvere questo problema si potrebbe forzare le variabili usate da core diversi ad essere su linee di cache differenti, facendo **padding**, cioè aggiungendo un numero di byte vuoti (in una struct per esempio) pari alla lunghezza di una linea tra le variabili usate da due core diversi. Il compilatore potrebbe comunque cambiare l'ordine dei campi in una struct, per evitare questo si può scrivere una cosa tipo:

```
struct alignTo64ByteCacheLine {
    int _onCacheLine1 __attribute__((aligned(64))),
    int _onCacheLine2 __attribute__((aligned(64)))
}
```

Per ottenere la lughezza della cache L1 si può usare la funzione:

```
sysconf(_SC_LEVEL1_DCACHE_LINESIZE);
```

3.5 Tool utili per ottimizzare

Per controllare l'efficienza del nostro codice si possono usare diversi tool e funzioni da linea di comando per ottenere diverse informazioni utili:

- `perf stat`: Permette controllare la quantità di cache-miss, cicli a vuoto ecc....
  Si può anche specificare cosa controllare tramite la flag `-e`
- `gprof`: Permette di controllare quanto tempo viene passato dal programma in ogni funzione per sapere quale ottimizzare
- `gdb`: Per eseguire debugging e vedere lo stack delle chiamate quando il programma si è fermato per un errore
- `valgrind`: Da info utili sull'esecuzione

Inoltre per misurare in modo più corretto le performance bisognerebbe:

- Disattivare il turbo boost
- Disattivare il frequency scaling

4

Pthreads

4.1 Threads

I processi sono un istanza di un programma, ognuno con la propria memoria, i threads sono analoghi ad un processo "meno pesante" (light-weight) che costa meno da creare e hanno memoria condivisa tra loro. Ovviamente i thread hanno comunque stack diversi perché potrebbero eseguire parti di codice differenti. La libreria **POSIX Threads**, abbreviata con Pthreads è una libreria standard per i sistemi operativi basati su UNIX e va linkata con il programma in C che si vuole eseguire con diversi threads importando l'header `pthread.h`. Un programma che utilizza Pthread si compila con `gcc` ma aggiungendo alla fine `-lpthread` per specificare la libreria di Pthread, poi si runna come qualsiasi programma C.

4.1.1 Creare un thread

Per creare un thread si usa la funzione `pthread_create`:

```
int pthread_create(
    pthread_t* thread_p,  // puntatore al thread creato
    const pthread_attr_t* attr_p, // attributi del thread
    void* (*start_routine) (void*), // funzione da eseguire
    void* arg_p  // argomenti della funzione
);
/* gli attributi possono anche essere settati a NULL per
creare un thread senza attributi particolari /*
```

Il tipo `pthread_t` è l'oggetto che rappresenta il thread e contiene tutte le informazioni per identificare il thread associato. La funzione che deve eseguire il thread indica proprio un **puntatore a funzione**, cioè un puntatore che indica la locazione di memoria in cui inizia la determinata funzione. Un puntatore a funzione può essere creato e utilizzato:

```
void* func(int a){
    // testo funzione
}

void main(){
    void(*func_ptr)(int)=func;
    *func_ptr(10); // come chiamare la funzione dal puntatore
}
```

Nel nostro caso la funzione che passiamo in input quando creiamo il thread deve ritornare un `void*` e avere come argomento un `void*`, un `void*` è un puntatore che permette di essere castato a qualsiasi altro tipo di puntatore. Oltre la funzione anche gli argomenti che passiamo in input a `pthread_create` devono essere di tipo `void*`, quindi se volessimo passare più di un argomento dovremmo creare una struct e passare il puntatore alla struct (castato a `void*`) a `pthread_create` per poi ricastarlo come puntatore alla struct dentro la funzione eseguita dal thread.

Quando si vuole aspettare che un thread finisca per mandare avanti l'esecuzione del codice si può usare la funzione bloccante `pthread_join`:

```
int pthread_join(pthread_t thread, void** valur_ptr)
```

La funzione attende la fine del determinato thread passato in input (se il thread ha già terminato l'esecuzione bisogna comunque chiamare la funzione per distruggerlo) e ritorna il valore della funzione eseguita del thread.
I thread, come i processi in MPI, hanno un ID univoco che può essere ottenuto tramite `pthread_self` che ritorna l'ID del thread chiamante e si possono confrontare due thread tramite `pthread_equal` che ci dice se l'ID dei due thread è uguale (sono lo stesso thread).
Esempio:
Se voglio eseguire una moltiplicazione di un vettore di n elementi per una matrice di $n \times m$ elementi con t threads, ogni thread processerà $\frac{n}{t}$ righe. In particolare il q-esimo thread processerà le righe dalla $q\frac{n}{t}$ alla $(q+1)\frac{n}{t}$. La funzione che eseguirà ogni thread sarà:

```
/*
Valori comuni a tutti i thread
n = righe della matrice e del vettore
m = colonne della matrice
x = vettore in input
A = matrice in input
y = vettore risultato
*/

void* Pth_mat_vec(void* rank){
    long my_rank = (long) rank;
    int i,j;
    int local_m = m/thread_count;
    int first_row = my_rank*local_m;
    int last_row = (my_rank+1)*local_m - 1;

    for(i=first_row;i<=last_row;i++){
        y[i] = 0.0;
        for(j=0;j<m;j++){
            y[i] += A[i][j]*x[j];
        }
    }
    return NULL;
}
```

4.1.2 Pinnare un thread

In alcuni casi è necessario specificare su quale core deve essere eseguito uno specifico thread. Possiamo farlo con il seguente codice:

```
// Non includere niente prima di queste 3 righe
#define _GNU_SOURCE
#include <pthread.h>
#include <sched.h>

void* thread_func(void* thread_args){
...
cpu_set_t cpuset;  // inizializza il cpuset
pthread_t thread = pthread_self();

CPU_ZERO(&cpuset);
CPU_SET(3, &cpuset);

s = pthread_setaffinity_np(thread, sizeof(cpuset), &cpuset);
...
}
```

Le due funzioni eseguite sul `cpuset` permettono prima di azzerare i core su cui può essere eseguito questo thread e poi specificare di eseguirlo sul core 3. La funzione `pthread_setaffinity_np` infine specifica che il thread che deve essere eseguito solo sui core specificati nel `cpuset` sia esattamente questo.

4.2 Sezioni critiche di codice

Una **sezione critica** di codice è una parte del codice in cui più processi o thread accedono agli stessi dati. Queste sezioni potrebbero causare problemi perché due processi potebbero modificare contemporaneamente lo stesso valore, l'uno ignorando le modifiche dell'altro. Ad esempio se abbiamo due thread che devono eseguire l'operazione:

```
y = Compute(my_rank);
x = x+y;
```

potremmo avere un'esecuzione:

T_1	T_2
Start	
Chiama Compute()	Start
Assegna y=1	Chiama Compute()
Salva x=0 e y=1 nei registri	Assegna y=2
Somma 0+1	Salva x=0 e y=2 nei registri
Scrive 1 nella locazione x	Somma 0+2
	Scrive 2 nella locazione x

Il secondo thread modifica quindi il valore di x senza considerare il cambiamento del primo thread.

4.2.1 Busy-Waiting

Una delle soluzioni possibili ai problemi dovuti alle sezioni critiche è il **busy-waiting (attesa attiva)**, in cui un thread controlla a ripetizione una condizione e fino a quando non si verifica non esegue una parte critica di codice. Per l'esempio di prima si potrebbe aggiungere una flag che indica il thread che deve eseguire la parte critica:

```
// flag = 0 di default

y = Compute(my_rank);
while(flag != my_rank);
x = x+y;
flag++;
```

Questa soluzione ha comunque dei problemi, il primo è che il thread sta comunque consumando energia mentre aspetta e il secondo è che in alcuni casi il compilatore potrebbe modificare l'ordine di alcune istruzioni se non sono dipendenti l'una dall'altra (non può sapere che il while serve per fare del busy-waiting), quindi potrebbe cambiare il codice precedente eseguendo:

```
// flag = 0 di default

y = Compute(my_rank);
x = x+y;
while(flag != my_rank);
flag++;
```

Ciò causerebbe lo stesso problema di prima, rendendo inutile il busy-waiting.

4.2.2 Mutex

I **mutex**, che sta per mutua esclusione (**mut**ual **ex**clusion), sono un tipo di variabile che permette l'accesso ad una sezione critica ad un solo thread alla volta. Pthread include un tipo standard per i mutex, cioè `pthread_mutex_t`.
Per creare un mutex bisogna usare la funzione `pthread_mutex_init`:

```
int pthread_mutex_init(
    pthread_mutex_t* mutex,  // mutex in output
    const pthread_mutexattr_t* attr_p
);
// gli attrubuti possono essere settati a NULL
```

Si può anche creare un mutex standard con la linea di codice:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Quando invece si è finito di utilizzare un mutex, per liberare lo spazio bisogna chiamare la funzione `pthread_mutex_destroy`.
Quando un thread deve ottenere l'accesso ad una sezione critica tramite il mutex deve chiamare `pthread_mutex_lock` e successivamente quando avrà finito la sezione critica dovrà chiamare `pthread_mutex_unlock`. Di base il lock è bloccante, quindi se viene chiamato da un thread e il mutex non è libero il thread aspetterà fino a quando non si libererà il mutex. Esiste anche

una versione non bloccante del lock chiamata `pthread_mutex_trylock`, che ottiene il lock se il mutex è libero, ma se non lo è non si blocca.
Per l'esempio di prima, il giusto codice usando i mutex è:

```
y = Compute(my_rank);
pthread_mutex_t mutex;
pthread_mutex_init(mutex,NULL);
pthread_mutex_lock(&mutex);
x = x+y;
pthread_mutex_unlock(&mutex);
```

La scelta del thread a cui dare il lock di un mutex è casuale o gestito dal sistema. A differenza del busy-waiting però il thread che sta aspettando un lock viene disattivato.

4.2.3 Problemi con la gestione di sezioni critiche

Starvation

La **starvation** è una condizione che accade quando un thread o un processo è fermo per un indeterminato periodo di tempo, anche se potrebbe ancora continuare l'esecuzione. Solitamente accade per colpa di una questione di priorità per cui il determinato thread non può accedere alle risorse che gli servono.

Nel caso dei mutex, se quando un thread viene disattivato viene messo in una coda FIFO, nessun processo potrà andare in starvation.

Deadlock

Un **deadlock** è uno stallo che si verifica quando ogni thread di un insieme di thread sta aspettando che un altro thread rilasci un lock, in maniera ciclica. Questo causa un blocco di tutti i thread dell'insieme.

Un esempio di deadlock potrebbe causarsi se due thread eseguono due lock sugli stessi mutex in ordine inverso:

```
// Thread 1
phread_mutex_lock(&mutex1);
phread_muex_lock(&mutex2);

// Thread 2
phread_mutex_lock(&mutex2);
phread_mutex_lock(&mutex1);
```

4.2.4 Semafori

Se ho bisogno di poter controllare l'ordine con cui i thread eseguono la sezione critica di codice, posso usare i **semafori**. I semafori hanno un valore iniziale che indica quanti thread o processi possono accedere contemporaneamente alla sezione critica e viene decrementato di 1 ogni volta che un thread entra nella sezione critica e incrementato di 1 ogni volta che un thread ci esce. I

semafori vengono implementati da una libreria di POSIX, usabile includendo `semaphor.h`. Le funzioni base per poter utilizzare un semaforo, che ha tipo di variabile `sem_t`, sono:

- `sem_init`: Inizializza il semaforo potendo scegliere se condividero anche tra processi e potendo settare un valore iniziale
- `sem_destroy`: Elimina il semaforo dopo il suo utilizzo
- `sem_wait`: Blocca il thread se il semaforo è 0. Se è maggiore, esegue la sezione critica e decrementa il semaforo
- `sem_post`: Incrementa il valore del semaforo
- `sem_getvalue`: Ritorna il valore attuale del semaforo

4.2.5 Barriere

Una **barriera** è una parte di codice (implementabile in diversi modi) che permette di assicurarsi che tutti i thread siano allo stesso punto dell'esecuzione. Nessun thread può continuare la sua esecuzione oltre la barriera finché tutti i thread non sono arrivati a quel punto. Le barriere possono essere usate per controllare il tempo di un programma o per debugging. Le barriere sono facilmente implementabili tramite busy-waiting e mutex con un counter condiviso protetto da un mutex. Il codice di una barriera implementata con busy-waiting e mutex è:

```
// variabili globali
int counter=0;
int thread_count;
phread_mutex_t barrier;
...

void* thread_work(...){
...
// Barriera
pthread_mutex_lock(&barrier);
counter++;
pthread_mutex_unlock(&barrier);
while(counter<thread_count);
...
}
```

Oltre a busy-waiting e mutex una barriera è implementabile con i semafori:

```
// variabili globali
int counter=0;
int thread_count;
sem_t count_sem; // valore iniziale 1
sem_t barrier;   // valore iniziale 0
...

void* thread_work(...){
...
```

```
// Barriera
sem_wait(&count_sem);
if(counter==thread_count-1){
    counter=0;
    sem_post(&count_sem);
    for(j=0;j<thread_count-1;j++){
        sem_post(&barrier);
    }
    else{
        counter++;
        sem_post(&count_sem);
        sem_post(&barrier);
    }
}
...
}
```

Il problema con queste due implementazioni è che non è possibile riutilizzare la barriera, perché bisognerebbe portare il counter a 0, ma se lo facesse un solo thread potrebbe farlo troppo velocemente non permettendo agli altri di controllare le condizione per uscire dalla barriera.

4.2.6 Variabili condizionali

Le **variabili condizionali** permettono di sospendere l'esecuzione di un thread finché una determinata condizione non avviene. Quando questa condizione avviene un altro thread può mandare un segnale al thread sospeso per svegliarlo. Se un thread aveva un lock quando viene sospeso lo rilascia e quando viene svegliato lo riottiene. Una variabile di condizione è sempre associata ad un mutex.
Pthread ha un tipo standard per le variabili condizionali, cioè `pthread_cond_t` e vengono utilizzate tramite diverse funzioni:

- `pthread_cond_init`: Inizializza la variabile condizionale potendo settare diversi attributi (possono essere `NULL`).
  Si può anche creare una variabile condizionale standard con la linea di codice:

  ```
  pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
  ```

- `pthread_cond_wait`: Sospende il thread fino a quando non verrà svegliato da un altro thread
- `pthread_cond_signal`: Sveglia uno dei thread che è stato sospeso
- `pthread_cond_broadcast`: Sveglia tutti i thread che sono stati sospesi
- `pthread_cond_destroy`: Elimina la variabile condizionale dopo il suo utilizzo

A differenza dei semafori la funzione wait è sempre bloccante, mentre le funzioni signal e broadcast nel caso in cui non ci sia nessun thread sospeso vengono ignorate.
Il codice per creare una barriera con le variabili condizionali è:

```
// variabili globali
int counter=0;
int thread_count;
pthread_mutex_t mutex;
pthread_cond_t cond_var;
...

void* thread_work(...){
...
// Barriera
pthread_mutex_lock(&mutex);
counter++;
if(counter==thread_count){
    counter=0;
    pthread_cond_broadcast(&cond_var);
}
else{
    while(pthread_cond_wait(&cond_var,&mutex)!=0);
}
pthread_mutex_unlock(&mutex);
...
}
```

Si utilizza un while al posto di un singolo `pthread_cond_wait` per evitare i casi in cui un thread viene svegliato anche se la condizione non si è verificata. La funzione wait in questi casi ritorna un valore diverso da 0.

4.2.7 Read Write Lock

Se dobbiamo accedere ad una grande struttura condivisa, come una **linked list**, potremmo controllare l'accesso dei thread tramite un mutex per l'intera lista, causando però la serializzazione del codice. Un'altra soluzione sarebbe usare un mutex ad ogni nodo, per non bloccare l'intera lista per ogni operazione, questo però diminuisce ancora di più l'efficienza per la grande quantità di operazioni di lock e unlock eseguite. Per risolvere problemi come questo si possono usare i **read write lock**, che sono simili ai mutex ma con due diverse operazioni di lock, una per poter leggere e una per poter scrivere. Questo tipo di lock permette di avere infiniti read lock simultanei su una stessa risorsa ma solo un write lock. Pthread implementa i read write lock con un oggetto `pthread_rwlock_t` e con le funzioni:

- `pthread_rwlock_init`: Inizializza il lock, potendo settare degli attributi (possono essere `NULL`). Un possibile attributo permette di decidere se dare la priorità alla scrittura o alla lettura quando ci sono due thread in attesa, tramite `pthread_rwlockattr_setkind_np`. Si può anche creare un raed write lock standard con la linea di codice:

  ```
  pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;
  ```

- `pthread_rwlock_destroy`: Distrugge il lock quando si è smesso di utilizzarlo
- `pthread_rwlock_rd`: Richiede un lock in lettura

- `pthread_rwlock_wr`: Richiede un lock in scrittura

Questo tipo di lock è più efficiente tanto quanto sono in più le operazioni di lettura rispetto a quelle di scrittura.

4.2.8 Thread-safety

Funzione thread-safe

Una funzione viene detta **thread-safe** se può essere eseguita da multipli thread contemporaneamente senza avere nessun problema.

Un esempio di una funzione non thread-safe è `strtok` che divide una stringa in input secondo dei caratteri specifici e ritorna la prima sottostringa trovate. Il puntatore alla stringa viene cachato per permettere di chiamare la funzione successivamente con stringa in input `NULL`. Questo causa però problemi se usiamo multipli thread perchè il puntatore in cache è condiviso.

Funzioni ri-entranti

Una funzione viene detta **ri-entrante** se ci assicura che se il processo viene stoppato durante la funzione, quando ricomincerà le cache non saranno state cambiate da altri thread o processi. Solitamente le funzioni ri-entranti sono anche thread-safe, ma non è sicuro.

Solitamente in C esistono delle versioni thread-safe delle funzioni, con nome che aggiunge "`_r`" alla fine, ad esempio `strtok_r`.

5 OpenMP

La programmazione secondo il paradigma **OpenMP** segue un approccio **GSLP (Globally Sequential, Locally Parallel)** a memoria condivisa. Viene utilizzata importando l'header `omp.h` e permette di convertire un codice sequenziale in parallelo in modo "esponenziale". Per compilare un programma che utilizza OpenMP si può usare `gcc` aggiungendo la flag `-fopenmp`.

5.1 Pragma

La conversione da codice sequenziale a parallelo avviene tramite dei **pragma**, cioè delle direttive che vengono date del preprocessore al compilatore nel caso in cui quel determinato pragma sia supportato (se non è supportato viene considerato come un commento e saltato). Un pragma viene definita con:

```
#pragma
```

Per esempio per far iniziare una sezione di codice parallelo basta aggiungere `#pragma omp parallel` e racchiudere il codice tra parentesi graffe:

```
#pragma omp parallel
{
    // questo codice viene eseguito in parallelo
}
```

Dopo un blocco di codice parallelo, prima di ricominciare l'esecuzione sequenziale tutti i thread devono aver finito, come se ci fosse una barriera implicita. Si può decidere il numero di thread che verranno usati in diversi modi (ordinati per priorità inversa):

- Universalmente: settando il valore tramite bash con il comando `echo OMP_NUM_THREAD=n`. Questo è il valore che verrà utilizzato da qualsiasi processo fatto partire nella shell in cui si è eseguito il comando.
- Proramma: tramite la funzione `omp_set_num_threads`. Questo è il valore che verrà utilizzato nel programma.
- Pragma: inserendo nel pragma la clausola `num_threads(n)`. Questo è il valore che verrà utilizzato in quel singolo blocco di codice parallelo.

Un determinato thread può conoscere il proprio id tramite la funzione `omp_get_thread_num` e il numero di thread attivi in una sezione di codice tramite la funzione `omp_get_num_threads`. In OpenMP si una determinata terminologia:

- **Team**: l'insieme di thread che esegue una parte di codice parallela

- **Master**: il thread iniziale dell'esecuzione
- **Padre**: il thread che incontrando la direttiva di eseguire una parte di codice parallela, crea altri thread formando un team. Solitamente è uguale al master.
- **Figlio**: il thread creato da un'altro thread

Ai pragma possono essere aggiunte diverse **clausole** che permettono di modificare alcune direttive nella sezione che si sta per eseguire. Si può, per esempio, definire una sezione critica all'interno di una sezione parallela con `critical`, nel caso invece si voglia aggiornare un valore e quindi non si vuole tutta l'intera istruzione venga eseguita da un solo thread ma solo l'incremento si può usare `atomic`.

5.1.1 Compilatori non supportanti OpenMP

Nel caso in cui si voglia poter far girare il codice anche su compilatori che non supportano OpenMP (facendolo girare in modo sequanziale), si possono usare delle istruzioni per cambiare il codice in base a se OpenMP è supportato. Per esempio al posto di:

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

si dovrebbe scrivere:

```
#ifdef _OPENMP
#include <omp.h>
#endif
```

Nel caso invece di una funzione che permette a un thread di ottenere il suo id e sapere quanti thread sono attivi in quella sezione di codice del tipo:

```
int my_rank=omp_get_thread_num();
int thread_count=omp_get_um_threads();
```

si dovrebbe scrivere:

```
#ifdef _OPENMP
int my_rank=omp_get_thread_num();
int thread_count=omp_get_um_threads();
#else
int my_rank=0;
int thread_count=1;
#endif
```

5.2 Scope delle variabili

In un programma seriale lo **scope** di una variabile indica in quale parti del programma quella variabile può essere vista e utilizzata. Nel caso di OpenMP si intende quali thread possono utilizzarla. Di defaut una variabile dichiarata fuori da una sezione di codice parallelo viene considerata condivisa da tutti i threads, mentre quelle dichiarate dentro sono private al singolo thread e scompaiono dopo la fine della sezione parallela.

5.2.1 Clausola di riduzione

Se si vuole modificare una variabile condivisa all'interno di una sezione parallela, ma senza che l'intera operazione, si può la clausola `reduction`, che permette di specificare un'operazione binaria e una variabile ed eseguire quell'operazione in modo da non avere problemi di concorrenza:

```
result = 0 // variabile condivisa
#pragma omp parallel \
reduction(+: result)
result+=local();

// lo \ permette di andare a capo in un pragma
```

In questo modo sono sicuro che l'aggiornamento della variabile `result` venga eseuito da un thread alla volta, ma senza che anche l'operazione `local()` faccia lo stesso. Per eseguire la riduzione vengono inizializzate delle variabili private al valore neutro pe r l'operazione specificata.

5.2.2 Modificare lo scope di una variabile

Si può specificare lo scope di una o più variabili in modi diversi tramite delle clausole:

- `default(none)`: obbliga a specificare lo scope di ogni variabile inizializzata fuori dalla sezione che viene usata all'interno
- `shared(x)`: la variabile `x` è condivisa da tutti i thread, comportamento di base, da usare solo insieme a `default(none)`
- `private(x)`: viene creata una copia della variabile `x` privata in ogni thread, non inizializzata
- `firstprivate(x)`: la variabile `x` è privata in ogni thread, ma con valore iniziato come la variabile esterna
- `lastprivate(x)`: funzione come `private(x)` ma il thread che esegue l'ultima iterazione della sezione copia il suo valore nella variabile esterna
- `threadprivate(x)`: viene creata una copia della variabile `x` privata in ogni thred, ma persistente per tutto il programma, inoltre un determinato thread n avrà esattamente la stessa variabile che aveva nella sezione precedente
- `copyin(x)`: usata insieme a `threadprivate(x)` permette di inizializzare la variabile privata di ogni thread con valore uguale a quella esterna
- `copyprivate(x)`: deve essere usata insieme a `single` per permettere a un singolo thread di settare una determinata variabile privata di tutti i thread a un valore specifico

5.3 Cicli for parallelizzabili

OpenMP permette di parallelizzare i cicli for semplicemente scrivendo `#pragma omp parallel for` prima di esso e il programa creerà dei thread e dividerà le iterazioni tra di loro. Per essere parallelizzato in questo modo un ciclo for deve avere un numero di iterazioni calcolabile durante la compilazione, quindi deve essere della forma:

$$\texttt{for}\left(\texttt{index=start}\ ;\ \begin{matrix}\texttt{index<end}\\ \texttt{index>end}\\ \texttt{index<=end}\\ \texttt{index>=end}\end{matrix}\ ;\ \begin{matrix}\texttt{index++}\\ \texttt{++index}\\ \texttt{index--}\\ \texttt{--index}\\ \texttt{index+=incr}\\ \texttt{index-=incr}\end{matrix}\right)$$

Inoltre la variabile `index` deve essere un intero o un puntatore e tutte le variabili coinvolte (`index, start, end, incr`) non devono essere modificate durante l'esecuzione del loop.
Esempi:

1. Break: non può essere parallelizzato

```
for(i=0;i<n;i++){
    ...
    if(...){
        break;
    }
}
```

2. Return: non può essere parallelizzato

```
for(i=0;i<n;i++){
    ...
    if(...){
        return 1;
    }
}
```

3. Exit: può essere parallelizzato

```
for(i=0;i<n;i++){
    ...
    if(...){
        exit();
    }
}
```

In un caso con dei for in sequenza (non annidati), se si scrivesse il pragma prima di tutti e due i for i thread del primo ciclo verrebbero distrutti e ne verrebbero creati altri per il secondo, sprecando tempo. Per renderlo più efficiente possiamo scrivere `#pragma mop parallel` all'inizio e poi `#pragma omp for` prima dei cicli for. In questo modo i thread verrebbero creati all'inizio e non distrutti tra i due cicli.

Esempio:
Nel caso dell'Odd-Even Sort che ha codice:

```
for(phase=0;phase<n;phase++){
    if(phase\%2==0){
        for(i=1;i<n;i+=2){
            if(a[i-1]>a[i]){
                tmp=a[i-1];
                a[i-1]=a[i];
                a[i]=tmp;
            }
        }
    }
    else{
        for(i=1;i<n-1;i+=2){
            if(a[i]>a[i+1]){
                tmp=a[i+1];
                a[i+1]=a[i];
                a[i]=tmp;
            }
        }
    }
}
```

Il modo migliore per parallelizzarlo con OpenMP, considerando che il primo for non è parallelizzabile è:

```
#pragma omp parallel num_thread(thread_count)\
default(none) shared(a,n) private(i, tmp, phase)
for(phase=0;phase<n;phase++){
    if(phase\%2==0){
        #pragma omp for
        for(i=1;i<n;i+=2){
            if(a[i-1]>a[i]){
                tmp=a[i-1];
                a[i-1]=a[i];
                a[i]=tmp;
            }
        }
    }
    else{
        #pragma omp for
        for(i=1;i<n-1;i+=2){
            if(a[i]>a[i+1]){
                tmp=a[i+1];
                a[i+1]=a[i];
                a[i]=tmp;
            }
        }
```

```
    }
}
```

5.3.1 Loop annidati

Nel caso di loop annidati solitamente basta parallelizzare il ciclo più esterno. Se però il numero di iterazioni del ciclo esterno siano meno del numero di core o quelle del ciclo interno siano poche conviene compressare i due cicli in uno solo.
Esempio:

```
for(int i=0;i<3;++i){
    for(int j=0;j<6;++j){
        c(i,j);
    }
}
```

In questo caso conviene compressare i due cicli in uno solo:

```
for(int ij=0;ij<3*6;++ij){
    c(ij/6,ij\%6);
}
```

OpenMP offre anche una clausola `collapse(n)` che compressa i primi n cicli annidati:

```
#pragma omp parallel for collapse(2)
for(int i=0;i<3;++i){
    for(int j=0;j<6;++j){
        c(i,j);
    }
}
```

5.3.2 Schedulare i loop

Quando si usa `#pragma omp for` per parallelizzare un loop le iterezioni vengono divise tra i vari thread in modo equo partendo dal thread 0, cioè se abbiamo un loop con n iterazioni e t thread, il primo thread eseguirà le iterazioni $[0, \frac{n}{t}]$, il secondo thread le iterazioni $[\frac{n}{t}+1, \frac{2n}{t}]$, e così via. Questa assegnazione delle iterazioni va bene nei casi in cui le iterazioni hanno tutte lo stesso costo, ma in alcuni casi non è così. Lo scheduling delle iterazioni può essere fatto in due modi:

- **Default**: i thread si dividono le iterazioni normalmente, quella effettuata se non si specifica nulla nel pragma
- **Ciclico**: i thread si dividono le iterazioni in modo ciclico, con blocchi di grandezza specificata, per effettuarle si deve specificare nel pragma la clausola `schedule(type, chunksize)`

Lo scheduling ciclico può essere specificato di diversi tipi:

- **static**: le iterazioni vengono assegnate ai thread ciclicamente in blocchi di grandezza `chunksize`, per esempio con un blocco di grandezza n specificato, il primo thread eseguirà

le prime n iterazioni, il secondo le seconde n e così via. Finiti i thread si ricomincia dal primo in modo ciclico. Nel caso sia omesso `chunksize` viene settato a 1.

- **dynamic**: le iterazioni vengono assegnate ai thread i blocchi di grandezza `chunksize`, ma solo quando un thread ha finito le proprie ne richiede altre (così che se un thread finisca le proprie non deve aspettare gli altri). Bisogna stare attenti alla grandezza di `chunksize` in questo caso perché un valore troppo piccolo farebbe si che il tempo per richiedere le iterazioni da parte di un thread sia maggiore del tempo per eseguirle, allo stesso modo un valore troppo grande farebbe perdere tutti i vantaggi dello scheduling ciclico. Nel caso sia omesso `chunksize` viene settato a 1.
- **guided**: le iterazioni vengono assegnate ad ogni thread in blocchi pari ad $\frac{i}{t}$ in cui i è il numero di iterazioni ancora da assegnare e t il numero di thread. Questo tipo di scheduling da per scontato che le iterazioni abbiano costo crescente andando avanti con l'indice. In questo caso `chunksize` non viene usato per specificare la grandezza dei blocchi di iterazioni, ma la minima grandezza che i blocchi possono avere. Anche qui nel caso sia omesso `chunksize` viene settato a 1.
- **auto**: il tipo di scheduling viene deciso autonomamente dal sistema o dal compilatore
- **runtime**: il tipo di scheduling viene deciso durante l'esecuzione tramite una variabile di ambiente `OMP_SCHEDULE` oppure con la funzione `omp_set_schedule`

Esiste inolte una clausola `ordered` che se inserita nel pragma che definisce il for permette di esplicitare all'interno del for un sezione con `#pragma omp ordered` che verrà eseguita come se il ciclo fosse in ordine.

5.4 Dipendenza tra dati

Se in un ciclo for si hanno delle dipendenze tra dati, come nel caso della sequenza di Fibonacci, cercando di parallelizzare il ciclo for non avremmo nessun errore nel codice, ma i numeri in output saranno sbagliati perché non è detto che l'iterazione i sia fatta dopo la $i-1$ e $i-2$. Dato un loop nella forma:

```
for(...){
    S1 // operazione su una locazione x
    ...
    S2 // operazione su una locazione x
}
```

Ci possono essere 4 differenti tipi di dipendenza tra S1 e S2 in base a se stanno leggendo o scrivendo `x`. I problemi ci sono se una dipendenza attraversa le iterazioni del ciclo, cioè la locazione x che serve all'iterazione i viene acceduta anche nell'iterazione $i+1$ o viceversa:

1. **Flow dependence**: Read after Write (RAW)

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

2. **Anti-flow dependence**: Write after Read (WAR)

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

3. **Output dependence**: Write after Write (WAW)

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

4. **Input dependence**: Read after Read (RAR), non è veramente una dipendenza

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

5.4.1 Rimozione delle dipendenze RAW

Per rimuovere un tipo di dipendenza RAW, ci sono 6 metodi possibili:

1. **Reduction, Induction variables**
2. **Loop skewing**
3. **Partial parallelizzation**
4. **Refactoring**
5. **Fissioning**
6. **Cambio di algoritmo**

In particolare i metodi consistono in:

1. **Reduction, Induction variables**: eliminare le variabili con incrementi costanti e eseguire riduzioni sulle variabili condivise
   Esempio:

```
double v=start;
double sum=0;
for(int i=0;i<N;i++){
    sum=sum+f(v);   //S1
    v=v+step;       //S2
}
```

   In questo caso ci sono 3 tipi di dipendenze RAW:

   - RAW(S1→S1) causata dalla variabile `sum`
   - RAW(S2→S2) causata dalla variabile `v`
   - RAW(S2→S1) causata dalla variabile `v`

   Essendo che `v` aumenta di un numero fisso ogni iterazione, possiamo direttamente calcolarla senza avere il valore dell'iterazione precedente, questo elimina le seconda e terza dipendenza:

```
double v;
double sum=0;
for(int i=0;i<N;i++){
    v=start+i*step;
    sum=sum+f(v);
}
```

Per eliminare la terza dipendenza si può usare una riduzione sulla variabile `sum`:

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

2. **Loop skewing**: modificare le operazioni del loop, per far si che ogni iterazione usi variabili calcolate dall'iterazione stessa
Esempio:

```
for(int i=1;i<N;i++){
    y[i]=f(x[i-1]);    //S1
    x[i]=x[i]+c[i];    //S2
}
```

In questo caso c'è una dipendenza RAW:

- RAW(S2→S1) causata dalla variabile `x[i]`

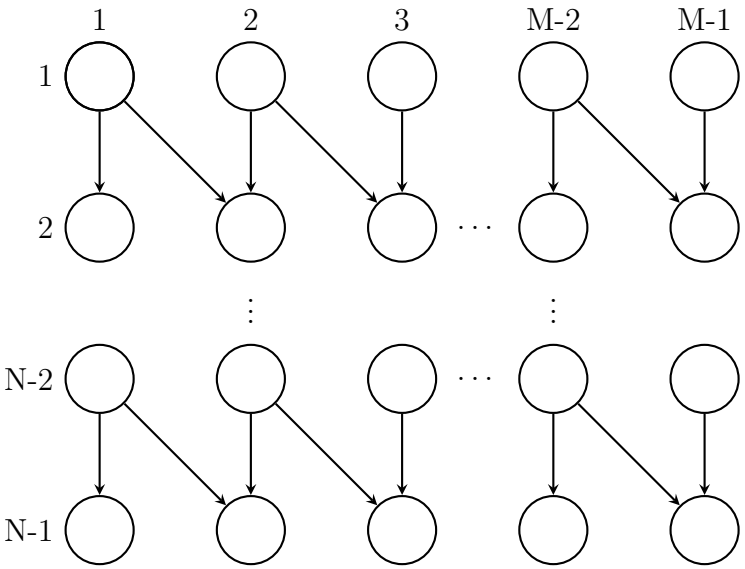
Se "srotoliamo" il ciclo ci accorgiamo che le operazioni su `x[i]` e `y[i+1]` sono sempre consecutive e non hanno bisogno di altre variabili calcolate in altre iterazioni, quindi possiamo scrivere il ciclo con queste due operazioni (devo rimuovere l'assegnazione di `y[1]` e `x[N-1]` dal ciclo):

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

3. **Partial parallelizzation**: tramite un grafo delle dipendenze traviamo un gruppo di nodi che non ha dipendenze tra loro e li parallelizzo
Esempio:

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

Disegnamo il grafo delle dipendenze:



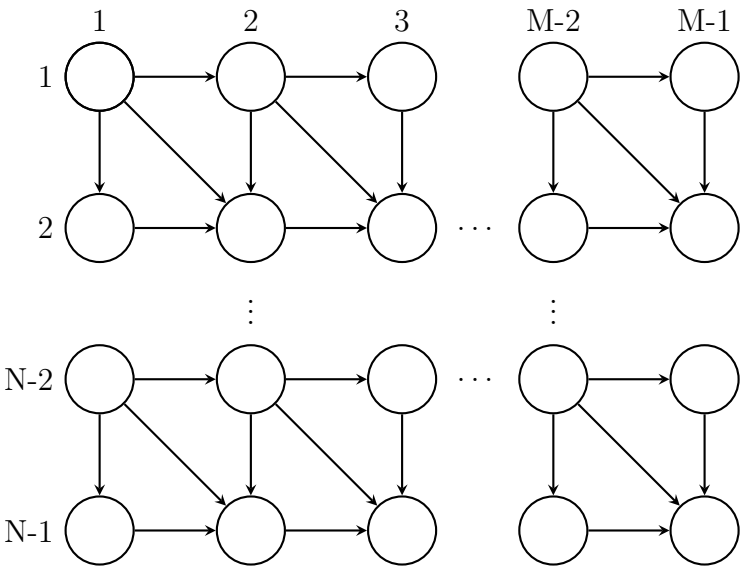
In questo grafo possiamo vedere che in ogni riga nessuna iterazione ha bisogno di una variabile calcolata in un'iterazione della stessa riga, quindi si possono parallelizzare le righe (il ciclo interno):

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

4. **Refactoring**: si usa lo stesso principio della partial parallelizzation, ma il ciclo deve essere completamente cambiato
Esempio:

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

Disegnamo il grafo delle dipendenze:



In questo grafo possiamo trovare delle ”onde” diagonali (partono da i,j e arrivano a j,i) in cui nessuna iterazione ha bisogno di una variabile calcolata in un'iterazione della stessa onda, quindi si possono parallelizzare, però bisogna cambiare completamente il ciclo:

```
for(wave=0;wave<n_wave;wave++){
    diag=F(wave);    // numero di iterazioni nell'onda
    #pragma omp parallel for
    for(k=0;k<diag;k++){
        int i=get_i(diag,k); // ottiene il giusto indice i
        int j=get_j(diag,k); // ottiene il giusto indice j
        data[i][j]=data[i-1][j]+data[i][j-1]+data[i-1][j-1];
    }
}
```

5. **Fissioning**: si divide un loop in diversi loop, alcuni parallelizzabili e altri no
Esempio:

```
s=b[0]
for(int i=1;i<N;i++){
    a[i]=a[i]+a[i-1];
    s=s+b[i];
}
```

In questo caso `a` non può essere parallelizzato, ma visto che `s` non richiede `a` possiamo dividere il ciclo in due, uno sequenziale che aggiorna `a` e uno parallelo con una riduzione che aggiorna `s`:

```
// parte sequanziale
for(int i=1;i<N;i++){
    a[i]=a[i]+a[i-1];
}

// parte parallela
s=b[0]
```

```
#pragma omp parallel for reduction(+: s)
for(int i=1;i<N;i++){
    s=s+b[i];
}
```

6. **Cambio di algoritmo**: si cambia l'algoritmo per far si che sia parallelizzabile

5.4.2 Rimozione delle dipendenze WAR

Per questo tipo di dipendenze il valore della variabile quando viene letta è noto, quindi posso farne una copia e rimuovere la dipendenza.
Esempio:

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

In questo caso posso fare una copia di `a` all'inizio e usare quella parallelizzando il ciclo:

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

#pragma omp parallel for
for(int i=0;i<N-1;i++){
    a[i]=A[i]+c;
}
```

5.4.3 Rimozione delle dipendenze WAW

Questo tipo di dipendenze possono facilmente essere risolte perché basta sapere il valore finale della variabile che causa la dipendenza senza modificarla ogni volta.
Esempio:

```
for(int i=o;i<N;i++){
    y[i]=a*x[i]+c;
    d=fabs(y[i]);  //fabs calcola il valore assoluto dei float
}
```

In questo caso la dipedenza è sulla variabile `d`, che non siamo sicuri che alla fine abbia il valore che gli viene assegnato nella n-esima iterazione. Per rimuovere la dipendenza basta spostare l'assegnazione di `d` fuori dal ciclo:

```
for(int i=o;i<N;i++){
    y[i]=a*x[i]+c;
}
d=fabs(y[i]);
```

5.5 Clausole di sincronizzazione

Quando si vogliono creare delle sezioni critiche con OpenMP, oltre alle clausole già citate `critical` e `atomic`, ci sono diverse altre clausole utilizzabili:

- **master**: la sezione verrà eseguita solo dal thread master, mentre gli altri thread andranno avanti. Questo ci assicura che il thread che eseguirà tutte le sezioni critiche con `master` sia sempre lo stesso
- **single**: la sezione verrà eseguita soo da un thread, mentre gli altri lo aspetteranno. Il thread che esegue la sezione non è specificabile a priori.
- **Barrier**: blocca l'esecuzione fino a quando tutti i thread non raggiungono la sezione

Se invece vogliamo far eseguire a thread diversi parti di codice diverse possiamo usare la clausola `sections` all'inizio di una sezione parallela e poi specificare all'interno `# pragma omp section` per definire una specifica sezione di codice da far eseguire ad un solo thread (le sezioni devono essere lo stesso numero dei thread sennò alcuni rimarranno fermi ad espettare). Alla fine di tutte le sezioni c'è una barriera implicita.

6
CUDA

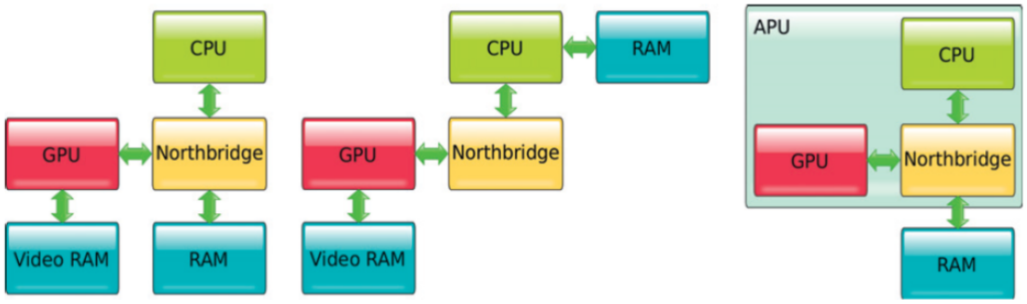
6.1 Architettura di una GPU

6.1.1 Differenze tra CPU e GPU

A differnza delle CPU in cui la maggior parte dello spazio è occupato dalla cache e dalla Control Unit (CU), nelle **GPU (Grafic Processing Unit)** la maggior parte dello spazio è occupato dai core, che sono molti di più ma molto più piccoli. Questo permette di usare una grande quantità di thread contemporaneamente. Un'altra differenza è il fatto che le GPU non facciano nessun tipo di "branch prediction" (calcolare in anticipo se c'è un if e calcolare le prossime istruzioni di conseguenza) e neanche un'esecuzione fuori ordine. Per queste differenze è consigliato eseguire dei programmi sequenziali sulla CPU mentre dei programmi paralleli sulla GPU.

6.1.2 Disposizione dei core

Nelle GPU i core, chiamati **streaming processor (SP)**, vengono raggruppati in **streaming multiprocessor (SM)**, cioè gruppo che condividono la stessa CU e istruzioni. Due o più di questi SM vengono raggruppati in dei "building block" che condividono una cache e sono tutti collegati ad una memori principale ad alta frequenza. In un sistema la GPU e la CPU possono essere disposte in 3 diversi modi:



I primi due casi rappresentano GPU esterne mentre il terzo una GPU integrata, come quella dei portatili.

6.2 Programmazione su GPU

Quando si programma del codice da far eseguire su una GPU ci sono una serie di problematiche da considerare, la prima è il fatto che la memoria host e la memoria della GPU sono divise, quindi è richiesto un trasferimento esplicito dei dati tra le due. Altri problemi possono derivare dallo diverso arrotondamento di numeri a virgola mobile o altri casi limite.
Ci sono molti modi di programmare una GPU, tra i più usati ci sono:

- **CUDA (Computer Unified Device Architecture)**: fornisce due serie di API (una ad alto ed una a basso livello) ed è disponibile su tutti i sistemi operativi. Funziona però solo su GPU Nvidia.
- **HIP**: la controparte di CUDA per AMD.
- **OpenCL (Open Computer Language)**: standard per scrivere programmi che possono essere eseguiti su diverse piattaforme tra cui GPU, CPU e ltri processori. Funziona sia su Nvidia che su AMD, infatti è la piattaforma base di svilupo per le GPU AMD e ha un modello molto simile a CUDA.
- **OpenACC**: specifiche per un API che permette di usare delle direttive del compilatore per mappare la computazione sulle GPU o su chip multicore.

6.3 Schedulazione dei therad in CUDA

CUDA permette di avere un modello di programmazione sulle GPU Nvidia, ha bisogno di hardware addizionale per interagire con l'utente e fornire della computazione. Permette di gestire in modo esplicito la memoria della GPU, che viene vista come un dispositivo che:

- Fa da co-processore alla CPU
- Ha la propria DRAM (memoria globale secondo il gergo id CUDA)
- Esegue diversi thread in parallelo con un costo di context switching molto basso

La struttura di un programma che utilizza CUDA è:

1. Allocare memoria sulla GPU
2. Trasferire i dati dall'host alla GPU
3. Eseguire un kernel CUDA (programma che viene eseguito sulla GPU)
4. Copiare i risultati sulla memoria dell'host

6.3.1 Modello di esecuzione di CUDA

L'esecuzione dei vari thread è organizzata in blocchi (da 1,2 o 3 dimensioni), posizionati su una griglia (anch'essa a 1,2 o 3 dimensioni) che permette di creare strutture finoa 6 dimensioni. Ogni thread conosce la sua posizione e questo permette di mappare 1:1 i punti dello spazio su cui si sta lavorando. Ogni GPU Nvidia ha una sua **capability** che determina la generazione della GPU e quali funzionalità può svolgere. Viene rappresentata tramite un numero di versione, chiamato "SM version".
Per ottenere la posizione di un thread nel blocco e successivamente nell'intera griglia si possono usare 4 variabili:

- `blockDim`: contiene la dimensione (x, y, z) del blocco
- `gridDim`: contiene la dimensione (x, y, z) della griglia
- `threadIdx`: contiene la posizione (x, y, z) del thread all'interno del blocco

- `blockIdx`: contiene la posizione (x, y, z) del blocco all'interno della griglia

Se vogliamo calcolare l'id globale del thread in una struttura a 6 dimensioni possiamo usare la formula:

```
int myID = (blockIdx.z * gridDim.x * gridDim.x  +
            blockIdx.y * gridDim.x + blockIdx.x)*
           (blockDim.x * blockDim.y * blockDim.z) +
            threadIdx.z * blockDim.x *blockDim.y +
            threadIdx.y * blockDim.x + threadIdx.x
```

6.3.2 Scrivere un programma con CUDA

Per scrivere un programma con CUDA bisogna definire una funzione che verrà eseguita da tutti i thread (kernel) e bisogna anche specificare come sono disposti i thread nei blocchi e nella griglia. Questo si può fare tramite un tipo di variabile `dim3`, che rappresenta una tupla di 3 interi (x, y, z). Un esempio di hello world in CUDA:

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

__global__ void hello(){
    printf("Hello␣world");
}

int main(){
    dim3 grid(2,3,4);  // griglia 2x3x4
    dim3 block(2,3,5); // blocchi formati da 2x3x5 thread
    hello<<<grid,block>>>();
    cudaDeviceSynchronize();
    return 1;
}
```

I file che contengono codice CUDA vengono nominati con l'estenzione `.cu`.
Per compilare il programma bisogna eseguire il seguente comando specificando la capability della GPU:

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

Nell'esempio la funzione `cudaDeviceSynchronize` serve a specificare al compilatore che deve aspettare la fine dell'esecuzione del codice sulla GPU, che di default va in modo asincrono rispetto al programma in c.
Ogni funzione CUDA ha delle specifiche di visibilità, che indicano da chi può essere chiamata quella determinata funzione:

- **__global__**: può essere chiamata dall'host o dalla GPU ed essere eseguita sul dispositivo/GPU. Dalla capability 3.5 anche il dispositivo può definire funzioni con __global__.
- **__device__**: può essere chiamata solo da un kernel e viene eseguita sulla GPU.
- **__host__**: può essere chiamata solo dall'host.

6.3.3 Scheduling dei thread

Ogni thread viene eseguito su uno streaming processor (core), i core nella stessa SM condividono la CU quindi devono eseguire lo stesso kernel. Diverse SM possono eseguire diversi kernel e ogni SM può eseguire più di un blocco, ma ogni blocco deve essere eseguito da una sola SM. I thread vengono divisi all'interno della SM tramite dgli scheduler in ulteriori gruppi chiamati **warp** attraverso il loro ID all'interno del blocco. Tutti i thread all'interno del warp devono in ogni istante eseguire la stessa istruzione (se si fanno branch i due branch vengono eseguiti come fossero in sequenziale).
In una GPU il content switching viene eseguito molto velocemente, questo perché una GPU ha molti registri che permettono di tenere salvati oltre che i dati del thread attualmente in eseuzione anche quelli dei thread inattivi. In questo modo ogni volta che un thread va in attesa ne ciene subito schedulato un altro da far eseguire.
Esempi:

1. Se abbiamo una GPU che permette di avere:

   - 8 blocchi per SM
   - 1024 thread per SM
   - 512 thread per blocco

   Potremmo definire dei blocchi composti da $8 \times 8 = 64$ threads, ma ciò permetterebbe di avere massimo $64 \cdot 8 = 512$ threads in una SM sprecando metà dei thread possibili.
   Un'altra opzione potrebbe essere di definire blocchi da $32 \times 32 = 1024$ threads, ma ciò va oltre il limite di 512 thread per blocco.
   La miglior opzione è quella di creare dei blocchi da $16 \times 16 = 256$ threads, che ci permetterebbe di sfruttare al massimo la SM con 4 blocchi.

2. Se abbiamo una GPU che permette di avere:

   - 8 blocchi per SM
   - 1536 thread per SM
   - 1024 thread per blocco

   Potremmo definire dei blocchi composti da $8 \times 8 = 64$ threads, ma ciò permetterebbe di avere massimo $64 \cdot 8 = 512$ threads in una SM sprecando più di metà dei thread possibili.
   Un'altra opzione potrebbe essere di definire blocchi da $32 \times 32 = 1024$ threads, ma visto il limite di 1536 thread potremmo usare un solo blocco.
   La miglior opzione è quella di creare dei blocchi da $16 \times 16 = 256$ threads, che ci permetterebbe di sfruttare al massimo la SM con 6 blocchi.

In generale quando si creano i blocchi e la griglia bisogna cercare di massimizzare il numero di thread in una SM e creare blocchi che sono un multiplo della grandezza di un warp (di solito 32).

6.4 Gestione della memoria con CUDA

Per far eseguire delle operazioni su dei dati alla GPU bisogna prima copiare i dati dalla memoria dell'host a quella della GPU, questo si può fare con la funzione:

```
cudaError_t cudaMalloc(void** devPtr, size_t size)
```

che alloca una quantità di byte pari a `size` e ritorna il puntatore in `devPtr` (come una malloc). Per poi copiare i dati dalla memoria dell'host a quella della GPU si può usare la funzione:

```
cudaError_t cudaMemcpy(void* dst, const void* src, size_t count, cudaMemc
```

Questa funzione decide chi siano la destinazione e la partenza attraverso il parametro `cudaMemcpyKind`:

1. `cudaMemcpyHostToHost` = 0
2. `cudaMemcpyHostToDevice` = 1
3. `cudaMemcpyDeviceToHost` = 2
4. `cudaMemcpyDeviceToDevice` = 3 (per configurazioni multi-GPU)
5. `cudaMemcpyDefault` = 4

Esempio:
Se vogliamo eseguire una somma tra due vettori, dovremo prima allocare due array sulla memoria della GPU, poi copiarci i due vettori dentro, eseguire la computazione e poi copiare il risultato di nuovo sull'host. Il codice eseguito dall'host sarà:

```
void vecAdd(float* A, float* B, float* C, int n){
    int size = n * sizeof(float);
    float *d_A, *d_B, *d_c;

    cudaMalloc((void**) &d_A, size);
    cudaMemcpy(d_a, A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void**) &d_B, size);
    cudaMemcpy(d_b, B, size, cudaMemcpyHostToDevice);
    cudaMalloc((void**) &d_C, size);

    vecAddKernel<<<ceil(n/256.0), 256>>> (d_A,d_B,d_C,n);

    cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);

    // libera la memoria della GPU
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
```

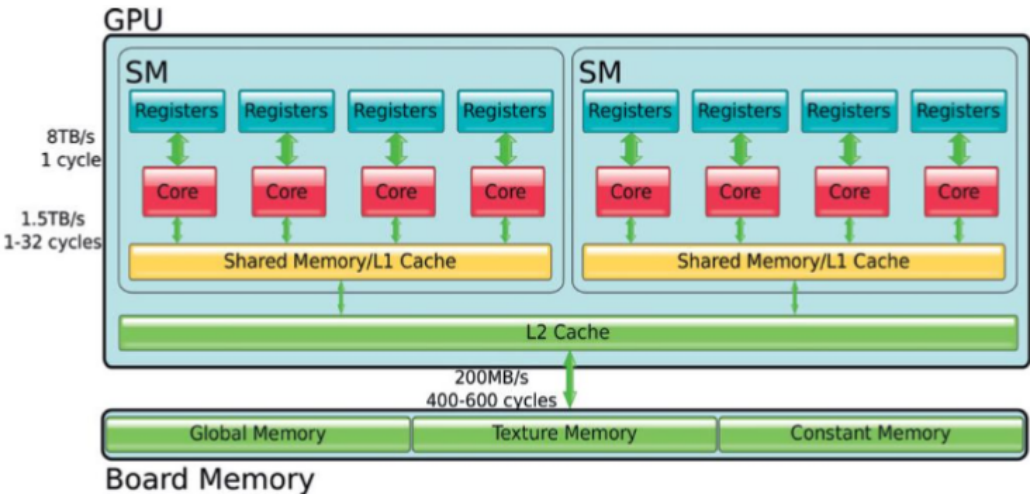
Il kernel eseguito dalla GPU sarà semplicemente:

```
__global__ vecAddKernel(float* A, float* B, float* C, int n){
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if(i<n){  // serve nel caso ci siamo piu thread di n
        C[i]=A[i]+B[i];
```

```
    }
}
```

6.4.1 Tipi di memoria

A livello visivo una GPU è composta in questo modo:



Ci sono diversi tipi di memoria, alcuni sul chip ed altri non:

- **Registri**: contengono variabili locali e vengono divisi tra tutti i thread sul core. La capabilità di una GPU definisce il numero massimo si registri usabili per thread, se questo numero viene superato alcuni dati vengono salvati sulla memoria globale (molto più lenta). Nvidia definisce l'**occupancy** come:

$$\text{occupancy} = \frac{\text{n° thread per SM}}{\text{max thread per SM}}$$

- **Memoria condivisa**: memoria sul chip che contiene dati usati frequentemente, è come una cache L1 gestita dall'utente. Viene condivisa da tutti thread su una SM e può essere usata per salvare dati usati frequentemente e per condividere sati tra i core.
- **Cache L1/L2**
- **Memoria globale**: parte principale della memoria fuori dal chip, ha una grande capacità ma è molto lenta.
- **Texture e surface memory**: dati maneggiati da hardware specializzato che permette di implementare funzioni veloci di filtering.
- **Memoria delle costanti**: contiene solo costanti. Può essere cachata e permette di fare broadcast di un valore a tutti thread di un warp. Se si vuole definire una costante bisogna dichiararla con `__constant__` e poi usare una versione alternativa della Memcopy:

```
cudaMemcpyToSymbol(variable_name, &host, sizeof(type), cudaMemcpyHost
```

I tipi delle variabili in CUDA definiscono dove vengono salvate e chi può vederle e utilizzarle:

Dichiarazione della variabile	Memoria	Scope	Lifetime
Variabili automatiche esclusi array	Registri	Thread	Kernel
Array automatici	Globale	Thread	Kernel
`__device__ __shared__`	Condivisa	Blocco	Kernel
`__device__`	Globale	Grglia	Applicazione
`__device__ __constant__`	Costante	Griglia	Applicazione

Esempio:
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.

```
__global__ void stancil_1d(int *in, int *out){
    __shared__ int temp[BLOCK_SIZE + 2*3];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + 3;

    temp[lindex] = in[gindex];
    if(threadId.x<3){
        temp[lindex - 3] = in[gindex -3];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    int result = 0;
    for(int offset = -3; offset <= 3; offset++){
        result+=temp[lindex + offset];
    }
    out[gindex]=result;
}
```

Con questo codice potrebbe comunque esserci un errore nel caso un warp all'iterno 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();
```

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 operandi al secondo. Se eseguiamo una sola operazione con ogni operando, possiamo 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.