Programmazione di Sistemi Embedded e Multicore

Leonardo Biason

3 dicembre 2024

About these notes

Those notes were made during my three years of university at Sapienza, and **do not** replace any professor, they can be an help though when having to remember some particular details. If you are considering of using *only* these notes to study, then **don't do it**. Buy a book, borrow one from a library, whatever you prefer: these notes won't be enough.

License

The decision of licensing this work was taken since these notes come from **university classes**, which are protected, in turn, by the **Italian Copyright Law** and the **University's Policy** (thus Sapienza Policy). By licensing these works I'm **not claiming as mine** the materials that are used, but rather the creative input and the work of assembling everything into one file.

All the materials used will be listed here below, as well as the names of the professors (and their contact emails) that held the courses.

The notes are freely readable and can be shared, but **can't be modified**. If you find an error, then feel free to contact me via the socials listed in my website. If you want to share them, remember to **credit me** and remember to **not** obscure the **footer** of these notes.

Bibliography & References

[1] Peter Pacheco, Matthew Malensek. (2021). *An Introduction to Parallel Programming (Second Edition)*. Morgan Kaufmann

[2] David B. Kirk, Wen-mei W. Hwu. (2017). *Programming Massively Parallel Processors (Third Edition)*. Morgan Kaufmann

[3] Gerassimos Barlas. (2022). *Multicore and GPU Programming (Second Edition)*. Morgan Kaufmann

The "*Programmazione di Sistemi Embedded e Multicore*" course was taught in the Winter semester in 2024 by prof. Daniele De Sensi (desensi@di.uniroma1.it)

I hope that this introductory chapter was helpful. Please reach out to me if you ever feel like. You can find my contacts on my website. Good luck!

Leonardo Biason

→ leonardo@biason.org

Indice

Chapter 1 ▸ Programmazione Parallela & C — Page 1

1.1 Da Sequenziale in Parallelo 1
1.1.1 Concorrenza, parallelismo e distribuzione 3
1.1.2 Review dell'architettura di Von Neumann 3
1.2 Il linguaggio C 4
1.2.1 Variabili, `printf` e `scanf` 5
1.2.2 Tipi primitivi, promozione e casting 7
1.2.3 Logica e cicli 7
1.2.4 Array, stringhe e matrici 11
1.2.5 Puntatori & gestione della memoria 12

Chapter 2 ▸ Message Passing Interface (MPI) — Page 14

2.1 Invio e Ricezione di Dati 17
2.2 Misurazione delle Prestazioni 19
2.2.1 Speed-Up ed Efficienza 20
2.2.2 Scalabilità Forte e Scalabilità Debole 21
2.2.3 Leggi di Amdhal e Gustafson 21
2.3 Esempi con MPI 22
2.4 Tipi Derivati 22

Chapter 3 ▸ PThreads — Page 25

3.1 Concorrenza 27
3.1.1 Busy Waiting 27
3.1.2 Mutex 27
3.2 Semafori 27
3.3 Barriere e Variabili di Confizioni 28
3.4 Read-Write Locks 28
3.5 Thread Safety 29
3.5.1 Thread Safety su MPI 29

Chapter 4 ▸ Architetture Multicore & OpenMP — Page 30

4.1 Organizzazione delle cache 30
4.1.1 Coerenza delle cache 31
4.1.2 Organizzazione della memoria 31

4.2 OpenMP 32
4.2.1 Operazioni di riduzione 35
4.2.2 Ciclo `for` parallelo 36
4.2.3 Dipendenze dei dati 37
4.2.4 Scheduling dei loop 37

CHAPTER 5 ▸ PROGRAMMAZIONE DI GPU E CUDA PAGE 39
5.1 CUDA 39
5.1.1 Scrivere un programma di CUDA 40

Capitolo 1

Programmazione Parallela & C

Ad oggi, molte tasks e molti programmi necessitano di grandi capacità di calcolo, soprattutto nel campo delle AI, e costruire centri di elaborazione sempre più grandi può non sempre costituire una soluzione. Certo, negli ultimi anni abbiamo assistito a una grande evoluzione dei microprocessori e delle loro potenze, ma non è abbastanza avere hardware sempre più potente, serve saperlo impiegare bene.

La società Nvidia, produttrice di **GPU**s (Graphical Processing Units) è riuscita, in questi anni, a produrre schede grafiche contenenti sempre più milioni di transistors, rendendo una GPU un oggetto estremamente complicato ma anche estremamente potente e versatile. Per la legge di Moore, il **numero di transistors** che si trovano all'interno di un circuito **raddoppia** sempre ogni anno, tuttavia ci sono dei limiti fisici che non possono essere valicati. Inoltre, avere più transistors all'interno di un circuito non è sempre positivo: più transistors equivale sì a una prestazione più alta, ma equivale anche a quantità maggiori di calore generato, e grandi quantità di calore rischiano di creare incosistenza nel circuito.

Come detto in precedenza, avere oggetti così potenti non è abbastanza, serve saper usare questi ultimi nel modo migliore possibile, ed è proprio questo l'obiettivo di questo corso: cercare di **programmare** nel modo **quanto più efficiente possibile** dei circuiti che possano eseguire codice in parallelo.

Per farlo, non possiamo usare linguaggi troppo ad alto livello come Python o Java: benché siano molto famosi e semplici da scrivere, non garantiscono un buon controllo sull'hardware. Una buona scelta sono i linguaggi a basso livello come **C**, **C++** o **Rust**, che offrono un ottimo compromesso tra facilità di scrittura del codice ed efficienza (altrimenti dovremmo scrivere il tutto in Assembly, il che risulterebbe forse troppo tedioso).

1.1 Da Sequenziale in Parallelo

Immaginiamo di voler fare un programma seriale in C che conti tutti i valori in un intervallo specifico $\{1, 2, \ldots, n\}$. Un modo semplice per farlo sarebbe sommare, attraverso un `for` loop, tutti gli elementi da 1 a n:

Sum.c

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

Tuttavia con valori di n molto larghi possiamo ottenere tempi enormi (immaginiamo con 10^{16}, o numeri di questa magnitudine). Per renderlo parallelo, immaginiamo di avere p cores (dove $p \ll n$), dove ogni core esegue il seguente codice:

SumForEachCore.c

```
// Variabili private di ogni core
sum_core_i = 0;
first_i = ...;
last_i = ...;

for (value_i = first_i; value_i < last_i; value_i++) {
    x_i = compute_next_value(i);
    sum_core_i += x_i;
}
```

Generalmente, si hanno 2 modi per scrivere del codice parallelo:

DEFINITION

Task e Data Parallelism

Definiamo come **task parallelism** quando una task principale viene spezzata in k sotto-tasks e distribuita ai vari p cores, i quali portano avanti le loro tasks autonomamente.

Definiamo invece con **data parallelism** quando i dati vengono divisi in k sotto-gruppi, che vengono affidati poi ai diversi p cores, che eseguono tutti quanti la stessa operazione.

Normalmente, se ogni core potesse eseguire codice indipendentemente, non sarebbe diverso scrivere codice seriale dallo scrivere codice parallelo. Tuttavia, come possiamo far sì che i core completino completare una task comune? Non hanno modo di passarsi dei dati in questo modo. Di base, ci deve essere una **comunicazione** tra i core, principalmente per i seguenti motivi:

- **Comunicazione**: se ad esempio i core devono condividere dei dati;
- **Divisione del carico**: far sì che i cores possano lavorare assieme, così da bilanciare il carico di lavoro;
- **Sincronizzazione**: siccome ogni core può avere tempi di esecuzione diversi, è importante che si sincronizzino a tratti, così da non creare concorrenza.

Come si possono scrivere programmi paralleli? Ci sono 4 librerie che verranno viste:

- **Message-Passing Interface** $(\text{MPI})_L$;
- **Posix Threads** $(\text{PThreads})_L$;
- **OpenMP**$_{L+C}$;
- **CUDA**$_{L+C}$

Dove X_L indica che la voce è una libreria e X_{L+C} indica che la voce è sia una libreria che un compiler.

DEFINITION

Classificazione di sistemi paralleli

I sistemi paralleli sono classificabili in multiple categorie. Una prima categorizzazione è la seguente:

- a **memoria condivisa**: i cores hanno accesso alla stessa porzione di memoria, e si sincronizzano per quanto riguarda la posizione dei dati in memoria;
- a **memoria distribuita**: ogni core ha una sua memoria privata, e per scambiare dati con gli altri core c'è bisogno che un core mandi un messaggio agli altri cores.

Un'altra possibile modalità di categorizzazione è la seguente:

- **Multiple Instruction Multiple Data (MIMD)**: ogni core ha una sua unità di controllo, può eseguire istruzioni diverse ed è indipendente dagli altri cores;
- **Single Instruction Multiple Data (SIMD)**: i cores hanno una stessa unità di controllo (quindi o eseguono tutti la stessa istruzione o sono tutti in idle). Un esempio di sistema SIMD è la GPU.

	Shared Memory	**Distributed Memory**
SIMD	CUDA_{L+C}	
MIMD	$\text{Posix Threads (PThreads)}_{L}$ OpenMP_{L+C} CUDA_{L+C}	$\text{Message-Passing Interface (MPI)}_{L}$

1.1.1 Concorrenza, parallelismo e distribuzione

Non c'è un'unica definizione, ma generalmente noi possiamo dire che:

- un sistema in concorrenza permette di eseguire tasks diverse ad un dato momento t di esecuzione del sistema (ad esempio, il sistema operativo);
- un sistema in parallelo permette di eseguire multiple tasks (anche diverse) per risolvere un problema comune;
- un sistema distribuito permette di distribuire le tasks su diverse entità, che però potrebbero (come non potrebbero) collaborare.

Generalmente, la differenza tra un sistema distribuito e un sistema parallelo si vede nella collaborazione del sistema, tuttavia il confine è molto labile.

1.1.2 Review dell'architettura di Von Neumann

L'architettura di Von Neumann è caratterizzata dalla presenza dei seguenti elementi:

- **Memoria**: insieme di posizioni (locations), ognuna con un indirizzo (address). In una posizione ci possono essere o dati o istruzioni;

- **CPU** o **processore**: un'unità di controllo che decide quali istruzioni eseguire e una sezione di **datapath** che esegue le istruzioni. In una CPU, lo stato delle variabili è salvato in dei registri, ovverosia un tipo di memoria estremamente veloce (utilizza flip-flops); un registro importante è il **program counter** (**PC**), che salva l'indirizzo della prossima istruzione da eseguire;
- **Interconnect**: è il mezzo attraverso il quale la CPU scambia dati con la memoria. Generalmente è un bus, ma può essere anche di un altro tipo (a volte più complesso del semplice bus).

Le architetture di Von Neumann hanno un **bottleneck**, dato dal necessario spostamento dei dati: accedere a un dato in una cache di tipo L_1 (una cache molto veloce situata vicino ai core della CPU) potrà forse costare $0.5ns$, tuttavia portare un dato dalla memoria ai registri costa $100ns$. La maggior parte del tempo speso in questo tipo di architetture, viene speso nel suddetto spostamento di dati.

Un modo per mitigare questi problemi è usare le **cache**, che permettono di salvare temporaneamente dei dati. Più è recente un dato, più è probabile trovarlo nella cache. Programmare una cache è molto complicato, ma è possibile e se ben fatto permette di ridurre drasticamente i tempi di accesso in memoria.

1.2 Il linguaggio C

Scritto nel 1972 come linguaggio di programmazione che sostituisse Assembly, C è un linguaggio a basso livello che permette di avere un controllo granulare delle risorse usate, e che viene usato spesso per disegnare e creare software che deve essere veloce e performante. In C, ogni istruzione è scritta nel formato `istruzione;`, dove il `;` separa un'istruzione da quella successiva. Come molti linguaggi, anche in C è possibile usare commenti (attraverso i simboli `//` per il commento a linea singola e `/* ... */` per il commento a multiple linee).

Per poter includere alcune funzioni di input e output, bisogna includere un file header. Il file responsabile per l'I/O è `stdio.h`, e va incluso con il comando `#include <stdio.h>`. L'inclusione di librerie esterne avviene in un momento di pre-processing, dove anche avviene ad esempio la rimozione dei commenti dai file di codice. Tale sessione di pre-processing viene fatta da un'entità chiamata **pre-processore**.

Ogni programma di C **deve** avere una funzione `main`, ma può anche avere più funzioni. L'importante è che ci sia una funzione `main`, poiché è da essa che parte l'esecuzione del codice. La funzione `main` peraltro **deve** finire con l'istruzione `return 0;`, poiché essa determina che il programma è stato eseguito con successo. Segue un esempio di programma in C:

HelloWorld.c

```c
#include <stdio.h>

int main () {
    printf("First C program!");
    return 0;
}
```

La funzione `printf()` fa parte della libreria `stdio`, che è stata inclusa in testa al file.

1.2.1 Variabili, `printf` e `scanf`

In C si possono definire delle variabili, e assegnargli un valore, grazie alla seguente sintassi: `<tipo variabile> <nome variabile> = <valore variabile>;`. Seguono degli esempi:

Variables.c

```c
int a = 0;          // Dichiarazione di un intero (int)
float b = 4.65;     // Dichiarazione di un decimale a virgola mobile (float)
char c = 'c';       // Dichiarazione di un carattere (char)

int x, y;           // Dichiarazione di due interi senza aver
                    // assegnato un valore
x = 4;              // Assegnamento di un valore alla variabile x
y = 5;              // Ugualmente alla riga precedente, y riceve un valore
```

Nell'ultima riga, abbiamo solo definito due variabili, ma non gli abbiamo assegnato alcun valore. Questa cosa è possibile in C, e torna utile quando dobbiamo definire una variabile ma non sappiamo ancora quale valore preciso dovrà essere inserito.

C permette anche di eseguire operazioni attraverso certi operatori. Segue una lista di tutte le operazioni disponibili:

Operatore	Operazione	Sintassi
+	Somma	`a + b`
-	Differenza	`a - b`
*	Moltiplicazione	`a * b`
/	Divisione (intera)	`a / b`
%	Modulo (resto della divisione)	`a % b`
( )	Parentesi	`(a + b) * c`
++	Incremento (di 1)	`a++`
--	Decremento (di 1)	`a--`

La funzione `printf` accetta molteplici argomenti, e permette anche di stampare il valore delle variabili a schermo:

Printf.c

```c
// Uso di printf con molteplici argomenti
printf("String 1");
printf("String 1", "String 2");
printf("String 1", ..., "String n");

// Stampa di variabili
int x = 4;
printf("Value of x: %d", x);
```

Si può usare il carattere % seguito da una lettera per determinare il placeholder per la variabile che vuole essere stampata. Segue una lista dei possibili placeholders:

Placeholder	Tipo variabile
%d	Interi (int)
%f	Decimali a virgola mobile (float)
%c	Caratteri (char)
%s	Stringhe
%u	Interi senza segno
%ld	Interi lunghi (32 bits)
%lu	Interi lunghi senza segno (32 bits)
%lld	Interi grandi (64 bits)
%llu	Interi grandi senza segno (64 bits)
%x	Interi esadecimali con lettere minuscole
%X	Interi esadecimali con lettere maiuscole
%o	Interi ottali
%e	Notazione scientifica con lettere minuscole
%E	Notazione scientifica con lettere maiuscole
%g	Rappresentazione ristretta di %f e %e
%G	Rappresentazione ristretta di %f e %E
%%	Stampa di un simbolo %

La funzione scanf permette di ricevere dall'utente in input dei valori. La sintassi della funzione è scanf("<descrizione>", <lista indirizzi in memoria>). Nella descrizione devono figurare i placeholder con i corretti tipi di dati che vogliamo richiedere. Un esempio è:

PrintfWithPointers.c

```c
int x;

// Il simbolo & ritorna l'indirizzo in memoria della variabile x,
// che specifica dove vogliamo salvare i dati raccolti con scanf
int* address = &x;

// Chiamando scanf, chiediamo all'utente di inserire un valore intero
scanf("Inserisci un intero: %d", address)

```

Nella sezione sui puntatori, verrà spiegato meglio il concetto di indirizzo di memoria.

Come in tutti gli altri linguaggi di programmazione, C segue uno specifico ordine di esecuzione: tutte le istruzioni vengono eseguite in ordine sequenziale, una dopo l'altra. Supponiamo di avere queste righe di codice:

ExecutionOrder.c

```c
int x = 40;
int y = 60;
y = y + x;
x = x - (y / 2);

```

Il risultato finale di x sarà -10, poiché prima gli viene assegnato 40, e poi gli viene sottratto il valore di (y + x) / 2.

1.2.2 Tipi primitivi, promozione e casting

Un computer non può memorizzare numeri infinitamente grandi o piccoli, in quanto si ha una precisione e una memoria limitata. Questo comporta che ci sono vari tipi di numeri, che possono occupare più o meno spazio in memoria rispetto ad altri. Questi tipi di variabili sono detti **tipi primitivi**, poiché sono definiti da C stesso. Una lista di tipi primitivi è la seguente:

Tipo primitivo	Spazio in memoria occupato
`char`	8 bits
`short`	16 bits
`int`	32 bits
`long`	32 bits
`long long`	64 bits
`float`	32 bits
`double`	64 bits
`long double`	128 bits

Esistono anche delle versioni `unsigned` dei primi 5 tipi, che, come suggerisce il nome, sono privi di segno, e possono essere dunque solo positivi.

Quando si esegue un'operazione tra due variabili di tipo diverso, C effettua una conversione della variabile con tipo più piccolo (a livello di bits) a una variabile di tipo più grande. Questa conversione è chiamata **promozione**.

Nel caso si volesse cambiare in corso d'opera il tipo di una variabile, si può effettuare il **casting**, ovverosia il cambio di tipo. Per esempio, supponiamo di avere due interi e volerne fare la divisione. Un modo per sfruttare il casting sarebbe il seguente:

CastingAndPromotion.c

```c
int x, y;
x = 10;
y = 3;
z = (float) x / y;
```

1.2.3 Logica e cicli

Quando si vuole eseguire un'azione piuttosto che un altra, in base ad una certa condizione, si possono usare vari controlli condizionali. Ci sono due tipi di controlli condizionali: l'`if-else if-else` e lo `switch`. L'`if-else if-else` è strutturato in questo modo:

IfElse.c

```c
if (<condizione>) {
    // Codice...
} else if (<condizione>) {
    // Codice...
} else {
    // Codice...
}
```

Lo `switch` invece è strutturato nel seguente modo:

Switch.c

```
switch (<variabile>) {
    case <valore>:
        // Codice...
        break;
    case <valore>:
        // Codice...
        break;
    // Tanti case quanti sono i casi che vogliamo gestire...

    default:
        // Codice...
        break;
}
```

Sia l'`if` che lo `switch` si avvalgono delle operazioni logiche per funzionare. Tali operazioni sono basate sul fatto che una condizione a un certo punto sia o vera o falsa. Infatti, esistono i valori booleani: `true` e `false`.

Per poter creare espressioni logiche, possiamo usare i seguenti operatori:

Operatore	Nome	Sintassi
<	Minore	`a < b`
>	Maggiore	`a > b`
<=	Minore o uguale	`a <= b`
>=	Maggiore o uguale	`a >= b`
==	Uguale	`a == b`
!=	Diverso	`a != b`
&&	E (AND)	`a && b`
||	O (OR)	`a || b`
!	Non (NOT)	`!a`

Altri strumenti che usano la logica sono i cicli, che permettono di eseguire codice fino a che una certa condizione non si avvera / finché una certa condizione non diventa falsa. I tre cicli che ci sono in C sono il `for`, il `while` e il `do-while`.

Il `for` ripete un'operazione per un numero finito di volte, fino a che la condizione non diventa falsa. La cosa comoda del ciclo `for` è che permette di tener conto dell'iterazione in cui ci si trova.

ForLoop.c

```
for (inizializzazione ciclo; condizione ciclo; incremento ciclo) {
    // Codice...
}

// Ad esempio...
for (int i = 0; i < 10; i++) {
    // Codice...
}
```

Il ciclo `while` e il `do-while` sono simili concettualmente: entrambi eseguono il codice al loro interno fino a che la condizione iniziale non diventa falsa. La differenza fra i due

è che il `do-while` esegue almeno una volta il codice all'interno del blocco `do`, mentre il `while` non assicura che il codice al suo interno venga eseguito.

DoAndDoWhile.c

```
// Ciclo while
while (<condizione>) {
    // Codice...
}

// Ciclo do-while
do {
    // Codice...
} while (<condizione>)
```

Si possono usare anche due keywords importanti: `break` farà uscire il codice dal ciclo in cui si trova, in modo anticipato (ad esempio se una sotto-condizione viene raggiunta); `continue` farà saltare un'iterazione del ciclo.

EXERCISE 1.2.0

Si scriva un programma che legge in input due numeri interi.

1) Il programma deve stampare se il primo numero è maggiore, minore o uguale al secondo.

2) Il programma deve stampare la somma e il prodotto dei due numeri.

La soluzione sarebbe la seguente:

Exercise1_2_0.c

```
#include <stdio.h>

int main () {
    int a, b;
    printf("Inserisci il tuo primo numero a: ");
    scanf("%d", &a);
    printf("Inserisci in tuo secondo numero b: ");
    scanf("%d", &b);

    if (a < b) {
        printf("%d è maggiore di %d\n", a, b);
    } else if (a == b) {
        printf("%d è uguale a %d\n", a, b);
    } else {
        printf("%d è minore di %d\n", a, b);
    }

    int sum = a + b;
    int product = a * b;
    printf("La somma è uguale a %d\nIl prodotto è uguale a %d", sum, product);
    return 0;
}
```

EXERCISE

1.2.1

Si scriva un programma che legge in input un numero intero `a`.

1) Il programma stampa tutti i numeri da 0 ad `a` in ordine crescente.
2) Il programma stampa tutti i numeri da 0 ad `a` in ordine decrescente.

Per svolgere l'esercizio si utilizzino cicli `for`

La soluzione sarebbe la seguente:

Exercise1_2_1.c

```c
#include <stdio.h>

int main () {
    int a;
    printf("Inserisci il tuo numero a: ");
    scanf("%d", &a);

    for (int i = 0; i <= a; i++) {
        printf("%d\n", i);
    }

    for (int i = a; i <= 0; a--) {
        printf("%d\n", a);
    }

    return 0;
}
```

EXERCISE

1.2.2

Si scriva un programma che legge in input un numero intero `a`.

1) Il programma stampa tutti i numeri da 0 ad `a` in ordine crescente.
2) Il programma stampa tutti i numeri da 0 ad `a` in ordine decrescente.

Per svolgere l'esercizio si utilizzino cicli `while`

La soluzione sarebbe la seguente:

Exercise1_2_2.c

```c
#include <stdio.h>

int main () {
    int a, i;
    printf("Inserisci il tuo numero a: ");
    scanf("%d", &a);
    i = 0;

    while (i <= a) {
        printf("%d\n", i);
        i++;
    }

    i = a;
    while (i >= 0) {
        printf("%d\n", i);
        i--;
    }

    return 0;
}
```

1.2.4 Array, stringhe e matrici

Un array è una **collezione ordinata** di elementi dello **stesso tipo**, e ogni array richiede che venga specificata la grandezza quando l'array viene dichiarato. La sintassi per creare un array è

```
<tipo> <nome_array> [<grandezza>]
```

Ad esempio, un array di 3 interi viene specificato con `int my_array [3]`. Per stampare il contenuto di un array non possiamo passare direttamente la variabile dell'array a `printf()`, ma bisogna usare un `for` loop per iterare attraverso l'intero array. Se passassimo un array alla funzione `printf()` non potremmo predirre direttamente il risultato di tale funzione.

Si possono anche creare **array multidimensionali**, per poter rappresentare tabelle o matrici. La sintassi per creare un array multidimensionale è la stessa di Python, dove la sintassi è

```
<tipo> <variabile> [<grandezza righe>][<grandezza colonne>]
```

In linguaggi più ad alti livelli, le stringhe sono considerate come un tipo a sé stante, ma in C questo non è il caso. Infatti, queste sono viste come **array di caratteri**. Ipotizziamo quindi di voler immagazzinare in una stringa di C la frase "*Hello world*", come lo faremmo in C? Semplicemente scrivendo

```
char string[12] = "Hello world";
```

Ma perché abbiamo dato una lunghezza di 12 caratteri e non 11? Perché al termine di ogni stringa c'è un carattere chiamato **terminatore**, identificato con `\0`. Dunque nella memoria avremmo i seguenti caratteri:

H	e	l	l	o		w	o	r	l	d	\0

Per eseguire operazioni particolari sulle stringhe, possiamo anche usare la libreria `string.h`. Alcune delle funzioni più utilizzate sono

- `strlen(<stringa>)`: per ottenere la lunghezza di una stringa
- `strcpy(<destinazione>, <source>)`: per copiare una stringa;
- `strcat(<destinazione>, <source>)`: per concatenare la stringa definita da `<source>` nella stringa definita da `<destinazione>`;
- `strcmp(<stringa_1>, <stringa_2>)`: per comparare due stringhe.

1.2.5 Puntatori & gestione della memoria

In C, è possibile accedere direttamente all'indirizzo di memoria dove è salvata una variabile. Per farlo, bisogna usare la sintassi

```
<tipo variabile>* <nome variabile> = &<nome variabile>
```

Ci sono due operatori che possiamo usare assieme ai puntatori, ovverosia `*` e `&`:

- l'operatore `*` restituisce il contenuto della cella di memoria indicata dal puntatore;
- l'operatore `&` restituisce il puntatore di una variabile.

La memoria può anche essere gestita liberamente dal programmatore in C. Tuttavia va fatta una distinzione tra due parti della memoria diverse, ovverosia lo stack e l'heap.

DEFINITION

Stack & Heap

La memoria di un computer è divisa in 2 tipi:

- `STACK`: è una parte **statica** della memoria, che viene usata per l'esecuzione delle funzioni;
- `HEAP`: è una parte **dinamica** della memoria, che permette di allocare oggetti ed elementi di dimensione variabile. In C, va allocata manualmente;

L'allocazione di memoria viene fatta tramite l'uso della libreria `stdlib.h`, che comprende principalmente 4 funzioni:

- `void* malloc(int size)`: permette di allocare un blocco di memoria contiguo, il cui spazio totale è determinato da `size`;
- `void* realloc(void* ptr, int size)`: realloca a memoria localizzata dal puntatore `ptr` a una dimensione determinata da `size`. Questa funzione può essere usata quando si necessita di riallocare in memoria una sezione di spazio contiguo più grande dello spazio originariamente richiesto. Non sempre è possibile estendere un blocco di memoria: potrebbe essere necessario che il sistema operativo riallochi tutto il blocco di memoria in un secondo spazio. Infatti, se questo dovesse essere il caso, verrà ritornato un puntatore nuovo;

- `void* calloc(int numElements, int size)`: permette di riservare una porzione di memoria di dimensione uguale a `numElements * size`, dove `size` è la dimensione dell'elemento singolo che vuole essere allocato;
- `void* free(void* ptr)`: libera la memoria, indicata dal puntatore `ptr`, riservata precedentemente con una delle precedenti funzioni.

È molto importante ricordarsi di liberare memoria quanto più possibile, proprio perché C non è dotato di un garbage collector, ed è compito del programmatore ricordarsi di liberare la memoria quando necessario.

Se serve sapere quanti bytes serve allocare per certi oggetti o elementi, potremmo voler usare la funzione `sizeof()`. Tale funzione restituisce, in bytes, la dimensione dell'elemento. Questa funzione è comoda perché non sempre possiamo sapere con precisione il numero di bytes necessari per allocare qualcosa.

CAPITOLO 2

Message Passing Interface (MPI)

MPI (acronimo di **M**essage **P**assing **I**nterface) è una libreria usata per **programmare sistemi a memoria distribuita**, e delle varie librerie menzionate nell'introduzione è l'unica pensata per sistemi a memoria distribuita. Questo vuol dire che la memoria e il core usato per ogni thread o processo sono **unici**. Tale core e memoria possono essere collegati attraverso vari metodi: un bus, la rete, etc...

MPI fa uso del paradigma **Single Program Multiple Data** (**SPMD**), quindi ci sarà un **unico programma** che verrà compilato e poi eseguito da vari processi o threads. Per determinare cosa ogni processo o thread deve fare, si usa semplicemente un'istruzione di **branching**, come l'`if-else` o lo `switch`.

Siccome la memoria non è condivisa tra i vari processi, l'unico modo per passarsi dei dati è attraverso l'invio di **messaggi** (da qui il nome della libreria). Per esempio, abbiamo visto come fare un semplice "Hello world" in C in modo sequenziale, ma possiamo anche renderlo parallelo tramite MPI. Ad esempio:

HelloWorld.c

```c
#include <stdio.h>

int main() {
    printf("Hello world");
    return 0;
}
```

Per rendere questo Hello World un programma parallelo tramite MPI, serve includere la libreria `mpi.h` e usare alcune funzioni della libreria. Vediamo intanto come potremmo scrivere il programma:

ParallelHelloWorld.c

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

int main(void) {
    // Per usare MPI, serve usare una funzione chiamata MPI_INIT;
    int r = MPI_Init(NULL, NULL);

    if(r != MPI_SUCCESS) {
        printf("C'è stato un errore con il programma");
        MPI_Abort(MPI_COMM_WORLD, r);
    }

    printf("Hello world");

```

ParallelHelloWorld.c

```
15      // Per terminare l'esecuzione di tutti i threads si usa MPI_Finalize
16      MPI_Finalize();
17      return 0;
18  }
```

Nel precedente codice sono state usate alcune funzioni e alcuni valori di MPI, che possiamo notare grazie al prefix "`MPI_`", **comune a tutte le definizioni**, siano esse di funzioni, variabili o costanti, **della libreria**:

- `MPI_Init()`: **inizializza** un programma su più processi o threads, e restituisce come output un `int`, che identifica se è stato possibile inizializzare con successo la libreria di MPI o meno (ovverosia restituisce 0 se la libreria è stata inizializzata con successo, un altro numero altrimenti);
- `MPI_SUCCESS`: è il segnale con cui è possibile comparare l'output di `MPI_Init` per controllare se MPI è stato inizializzato correttamente o meno;
- `MPI_Abort(MPI_COMM_WORLD, <mpi_boot_result>)`: **abortisce** l'esecuzione di MPI, ad esempio nel caso in cui l'inizializzazione non sia stata eseguita con successo;
- `MPI_Finalize()`: **interrompe** l'esecuzione di MPI a fine programma.

Per compilare ed eseguire un programma con MPI si usa `mpicc`, che è un wrapper del compilatore `gcc` di C. Un comando che viene usato per compilare un programma che usa MPI può essere il seguente:

Terminal

```
$ mpicc <file>.c -o <output>
$ mpicc -g -Wall <file>.c -o <output>    # Fa stampare i warning in console
```

Il compilatore ha molte flags che possono essere usate, così da personalizzare il processo di compilazione. Nel secondo comando si può notare l'uso di due flags che possono risultare comode in fase di debug:

- `-Wall`: fa stampare in console **tutti i warnings** del compilatore;
- `-g`: fa stampare in console varie **informazioni di debug**.

Questo è per quanto riguarda la compilazione, ma per eseguire il programma invece? Dovremo usare `mpirun`, attraverso il seguente comando:

Terminal

```
$ mpirun -n <numero_core_fisici> <programma>
$ mpirun --oversubscribe -n <numero_core> <programma>
        # Permette di usare più core di quelli fisici
```

Normalmente MPI esegue il codice solo sui core fisici di una CPU, tuttavia è possibile far sì che questa limitazione non venga considerata. La flag `--oversubscribe` permette di lanciare il programma su n processi, dove $n \geq$ numero di core fisici.

In un programma complesso, è spesso utile sapere quale core esegue quale parte di programma, magari anche per assegnare dei compiti diversi ad ogni core. In questi casi, possiamo differenziare i processi in base al loro **rank**.

DEFINITION

Rank

Il **rank** di un processo appartenente a un programma di MPI è un **indice incrementale**, nell'intervallo [0, 1, 2, …, p), che viene assegnato ad ogni processo.

Prima abbiamo usato, nella funzione `MPI_Abort()`, un parametro di MPI: `MPI_COMM_WORLD`. Questo è un **comunicatore**, ovverosia un canale di comunicazione con altri processi / thread (può anche essere usato per tutti i processi). `MPI_COMM_WORLD` è il comunicatore di default in MPI, a cui tutti i processi sono collegati. Ogni comunicatore ha vari parametri associati, tra cui anche la sua **grandezza** (ovverosia quanti processi sono collegati al comunicatore). Ci sono due funzioni importanti relativi al comunicatore, che permettono di ottenere sia la grandezza del comunicatore che il rank del processo che chiama la funzione:

Code Manual

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

Questa funzione ritorna il numero di processi che usano il comunicatore specificato dal parametro `comm` (di tipo `MPI_Comm`).

Parametri di `MPI_Comm_size()`:

- `comm`: il comunicatore di cui vogliamo sapere la dimensione;
- `comm_size_p`: il puntatore di dove verrà salvata la dimensione del comunicatore.

Code Manual

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

Questa funzione ritorna il rank relativo al comunicatore, ovverosia l'indice del processo che ha eseguito la chiamata di questa funzione.

Parametri di `MPI_Comm_rank()`:

- `comm`: il comunicatore rispetto cui si vuole conoscere il proprio rank;
- `my_rank_p`: il puntatore di dove verrà salvato il rank del processo nel comunicatore.

Possiamo provare ad eseguire il seguente codice, che dati p processi fa stampare ad ognuno di loro il proprio rank e la dimensione del comunicatore:

MPIRankAndSize.c

```
1  #include <stdio.h>
2  #include <mpi.h>
3
4  int main(void) {
5      int res = MPI_Init(NULL, NULL);
6      int comm_sz, my_rank;
7
8      // Determiniamo la dimensione del comunicatore
9      MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
10
11     // Determiniamo il rank del processo che esegue
```

MPIRankAndSize.c

```
12      MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
13
14      if(res != MPI_SUCCESS) {
15          printf("C'è stato un errore con il programma");
16          MPI_Abort(MPI_COMM_WORLD, res);
17      }
18
19      printf("'Hello world' dal core %d di %d\n", my_rank, comm_sz);
20
21      MPI_Finalize();
22      return 0;
23  }
24
```

Chiamando il precedente programma con il seguente comando di MPI, noteremo un effetto interessante, che magari potremmo non aspettarci:

Terminal

```
$ mpirun --oversubscribe -n 4 02-mpi-ranks-comm

// Prima esecuzione
'Hello world' dal core 1 di 4
'Hello world' dal core 0 di 4
'Hello world' dal core 2 di 4
'Hello world' dal core 3 di 4

// Seconda esecuzione
'Hello world' dal core 3 di 4
'Hello world' dal core 0 di 4
'Hello world' dal core 1 di 4
'Hello world' dal core 2 di 4
```

Notiamo che l'ordine dei processi è sempre diverso, sia nella prima esecuzione che nella seconda. Ma perché accade? Siccome i processi vengono eseguiti ognuno su un core diverso, soprattutto se $p = |\text{core}|$, allora in certi momenti alcuni dei core verranno usati dal sistema operativo, tramite cambi di contesto, per effettuare altre operazioni, magari del kernel. Questo può causare dei ritardi di esecuzione, e quindi il motivo del perché ogni volta l'ordine è diverso.

2.1 Invio e Ricezione di Dati

Per mandare qualcosa da un processo all'altro, MPI usa i **comunicatori**. Ci sono molti modi per far sì che i processi si scambino messaggi attraverso un comunicatore, e molte funzioni sono costruite sulla base di due funzioni fondanti di MPI, ovverosia le funzioni `MPI_Send()` e `MPI_Recv()`. Iniziamo vedendo la sintassi di `MPI_Send()`:

Code Manual

```
int MPI_Send(
    void* msg_buf_p,
    int msg_size,
    MPI_Datatype msg_type,
    int dest,
    int tag,
    MPI_Comm communicator,
)
```

Parametri di `MPI_Send()`:

- `msg_buf_p`: puntatore del buffer da cui vengono inviati i dati;
- `msg_size`: grandezza **in elementi** del messaggio;
- `msg_type`: tipo dei dati che vengono inviati;
- `dest`: rank del processo che riceve i dati;
- `communicator`: Comunicatore usato per lo scambio dei dati

Per mandare dei dati però è necessario che ci sia, all'interno del comunicatore, almeno un processo / thread che sia pronto a ricevere i dati. Per far sì che un processo si dichiari pronto a ricevere si usa la funzione `MPI_Recv()`, che ha la seguente sintassi:

Code Manual

```
int MPI_Recv(
    void* msg_buf_p,
    int buf_size,
    MPI_Datatype buf_type,
    int source,
    int tag,
    MPI_Comm communicator,
    MPI_Status* status_p
)
```

Parametri di `MPI_Send()`:

- `msg_buf_p`: puntatore del buffer in cui verranno salvati i dati inviati;
- `msg_size`: grandezza **in elementi** del messaggio;
- `msg_type`: tipo dei dati che vengono inviati;
- `source`: rank del processo che invia i dati;
- `communicator`: Comunicatore usato per lo scambio dei dati.

In entrambe le funzioni c'è un parametro che non abbiamo spiegato, ovverosia `tag`. Tale parametro permette di distinguere i messaggi mandati su un comunicatore attraverso un tag (in questo caso, un intero), e far sì che i vari processi / threads eseguano parte di codice diverso in base al tag.

Ma possiamo esser certi che, quando inviamo un messaggio, esso sia stato ricevuto correttamente? In genere non possiamo avere la certezza che sia stato ricevuto correttamente usando solo una chiamata di `MPI_Send()` e `MPI_Recv()`, ma per certo il messaggio è stato inviato e ricevuto se le seguenti condizioni sono vere allo stesso momento:

- se `recv_type = send_type`;
- se `recv_buf_sz` $\geq$ `send_buf_sz`.

Il destinatario può anche ricevere un messaggio **senza sapere** la **quantità di dati** nel messaggio, il **mittente** del messaggio (usando `MPI_ANY_SOURCE`) e il **tag** del messaggio (`MPI_ANY_TAG`).

Supponiamo che in un momento del codice non ci interessi sapere chi manda il messaggio, ma che in un secondo momento ci interessi saperlo. Per poter vedere in un momento successivo il mittente o altri dati del messaggio possiamo usare il parametro `status`, di

tipo `MPI_Status`, che permette di salvare in memoria lo status del messaggio (quindi vedremo che il parametro è scritto come `MPI_Status* &status`). I possibili valori di `status` sono i seguenti:

- `MPI_Source`: indica il mittente del messaggio;
- `MPI_Tag`: indica il tag del messaggio;
- `MPI_Error`: indica se ci sono stati errori nella comunicazione.

Un'altra funzione utile relativa ai messaggi è `MPI_Get_count()`, che indica quanti dati stanno venendo mandati .

2.2 Misurazione delle Prestazioni

Una volta che sviluppiamo un programma con MPI, è importante misurarne le prestazioni. Esistono vari modi per controllare l'efficienza di un programma, e qui ne vedremo alcuni.

Un modo molto semplice per capire l'efficienza di un programma consiste nel misurare il tempo di esecuzione, e possiamo farlo tramite la funzione `MPI_Wtime()`, la quale ritorna un timestamp che misura il tempo dal momento in cui il programma viene eseguito. Per poter ottenere il tempo di esecuzione, bisognerà sottrarre dal tempo a fine esecuzione il tempo di quanto la funzione viene chiamata.

In un programma parallelo, ogni processo calcola il suo tempo, ma può essere che alcuni processi impieghino meno tempo, soprattutto se ogni processo esegue operazioni diverse. Per questo motivo, l'efficienza si calcola considerando il tempo maggiore. Per ovviare a questo problema, tutti i processi possono mandare a un solo processo i loro tempi, e tramite una chiamata di `MPI_Reduce()` si può ritornare solo il tempo maggiore.
Tuttavia, un altro problema si presenta: **non tutti i processi possono iniziare allo stesso tempo**. E questo chiaramente è un problema, soprattutto se il processo che inizia in ritardo inizia parecchio in ritardo. Un modo per far sì che tutti i processi inizino assieme è tramite l'uso di una funzione chiamata `MPI_Barrier()`. Tale funzione è una **funzione collettiva**, che si propaga a tutti i processi, e che li blocca fino al momento in cui tutti non raggiungono tale barriera. Possiamo considerare questa funzione come una sorta di checkpoint, che **posta all'inizio del programma** fa sì che tutti i processi inizino ad eseguire il loro codice "assieme".

Tuttavia, se anche questa collettiva generasse ritardi venendo propagata, non avremmo più un inizio collettivo. Riuscire a sincronizzare tutti i processi assieme è complicato, e richiederebbe l'uso di clock interni per ogni core. Tuttavia, in casi in cui non sia fondamentale avere grande precisione, `MPI_Barrier()` **garantisce** una **buona approssimazione**.

Ma è abbastanza eseguire l'applicazione solo una volta per avere una buona misurazione? Generalmente no, e ci sono vari motivi: sappiamo che il sistema operativo, di tanto in tanto, può effettuare dei cambi di contesto, per cui uno o più core vengono usati per eseguire delle operazioni dell'OS. Questo chiaramente può aumentare i tempi per alcune

esecuzioni. In altri casi invece potrebbe essere che lo scheduler della memoria liberi la cache prima del necessario, etc...

Tali motivi di ritardo vengono chiamati **rumore** (o **noise**). In genere, per misurare l'efficienza di un programma, **conviene sempre basarsi su tutte le esecuzioni di un programma**, considerando media, mediana, il tipo di distribuzione, eventuali intervalli di confidenza, etc... Il tempo di esecuzione minimo, da solo, non indica molto, poiché per vari motivi di interferenza può non sempre (e in realtà non accade quasi mai) indicare la vera efficienza del programma.

2.2.1 Speed-Up ed Efficienza

Generalmente, potremmo aspettarci che se un processo viene eseguito da quanti più core possibili, allora potremmo ottenere tempi sempre più bassi. Benché in molti casi questo sia vero, lo speed-up dell'esecuzione parallela non è sempre esistente aumentando il numero di processi e di core. Infatti, consideriamo il seguente esempio:
Se notiamo la prima colonna, il tempo di esecuzione con 8 e 16 core è identico. Questo perché, oltre a un certo punto, anche dividendo le operazioni su più core, raggiungeremmo un tempo minimo di inizializzazione che non è ottimizzabile. Dunque, lo speed-up terminerebbe. Ma cosa intendiamo effettivamente con speed-up?

DEFINITION

Speed-up

Definiamo le seguenti variabili:

- $T_s(n)$: il **tempo di esecuzione** di un problema di dimensione n in **seriale**;
- $T_p(n, p)$: il **tempo di esecuzione** di un problema di dimensione n eseguito in **parallelo** su p core;

Lo **speed-up** di un'applicazione in parallelo su p core viene dunque definito come il rapporto tra il tempo di esecuzione in seriale e il tempo di esecuzione in parallelo con p core:

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

Se $S(n, p) = p$, allora lo speed-up viene definito come **speed-up lineare**.

C'è un dettaglio importante di questa precedente definizione: il tempo di esecuzione in sequenziale **non è uguale** al tempo di esecuzione in parallelo con $p = 1$, anzi, tendenzialmente $t_p(n, 1) \geq t_s(n)$. A causa di questo, possiamo definire due implementazioni diverse della precedente formula:

$$S(n, p) = \frac{T_s(n)}{T_p(n, p)} \qquad S(n, p) = \frac{T_p(n, 1)}{T_p(n, p)}$$

Speed-up **Scalabilità**

Grazie alla definizione di speed-up, possiamo definire anche il concetto di **efficienza**:

DEFINITION

Efficienza

L'**efficienza** di un programma viene calcolata come il rapporto tra lo speed-up di un programma e il numero di core su cui viene eseguito:

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

2.2.2 Scalabilità Forte e Scalabilità Debole

Esistono due tipi di scalabilità, in base ai risultati che si ottengono dalle misurazioni dell'efficienza di un programma: **scalabilità forte** e **scalabilità debole**.

- Per **scalabilità forte** si intende quando, dato un problema di grandezza n, se si incrementa il numero di processi p, allora l'efficienza rimane alta;
- Per **scalabilità debole** si intende quando, dato un problema di grandezza n e un numero di processi p, se incrementando ugualmente n e p, allora l'efficienza rimane alta.

`comm_size`	Ordine della matrice				
	1024	2048	4096	8192	16384

2.2.3 Leggi di Amdhal e Gustafson

Quando dobbiamo rendere un programma parallelo, sappiamo che possiamo parallelizzare solo alcune operazioni: ad esempio, leggere dal disco dei dati, richiedere dei dati in input all'utente o mandare dati attraverso un comunicatore di MPI. Avremo dunque una parte **sempre sequenziale** e una parte **parallelizzabile**. La frazione del programma sempre sequenziale viene indicata con α. C'è una legge, chiamata **legge di Amdhal**, che definisce lo speed-up possibile di un'applicazione.

DEFINITION

Legge di Amdhal

Per la **legge di Amdhal**, lo speed-up di un'applicazione, se resa parallela, è limitato dalla **frazione di codice sequenziale** α:

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

Lo speed-up calcolabile sarà dunque uguale a

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

Generalmente, se portassimo il numero di core p all'infinito, raggiungeremmo il seguente valore:

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

La legge di Amdhal tuttavia ha dei problemi: intanto non tiene conto della scalabilità debole (poiché la legge di Amdhal considera un n costante)

DEFINITION

Legge di Gustafson

La **legge di Gustafson** definisce ciò che si chiama **speed-up scalabile**, e che prende in assunzione che, se si **aumenta** il **numero di processi** p per una costante α, allora anche la **dimensione del problema** n **aumenta** di α.

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

2.3 Esempi con MPI

Un altro esempio di codice parallelizzabile è la **somma tra vettori**. Per questo esempio, abbiamo dei vettori composti da n elementi, e abbiamo p core disponibili per fare la somma delle componenti dei vettori. Intuitivamente, il metodo più semplice per parallelizzare tale esempio è assegnare a ogni core un numero $\frac{n}{p}$ di elementi da ogni vettore e calcolare dunque la somma tra le varie parti autonomamente.

MPI ha una funzione collettiva che ci permette di mandare ad ogni processo una parte di un elemento, ovverosia `MPI_Scatter()`. Tale funzione è definita come segue:

Code Manual

```
int MPI_Scatter(
    void* send_buff_p,
    int send_count,
    MPI_Datatype send_type,
    void* recv_buff_p,
    int recv_count,
    MPI_Datatype recv_type,
    int src_proc,
    MPI_Comm comm
)
```

2.4 Tipi Derivati

Sappiamo che in C è possibile creare `structs`, ovverosia composizioni di vari tipi. Tuttavia, nel caso in cui volessimo usare una `struct` custom, come possiamo dire ad MPI come funziona la nostra `struct`? Si possono definire dei **tipi di MPI derivati**. Ad esempio, nell'esercizio del trapezio, noi per inviare dei dati abbiamo impiegato 3 invii e 3 ricezioni. Possiamo fare di meglio utilizzando proprio i tipi derviati.

DEFINITION

Tipi Derivati

Un **tipo derivato** di MPI consiste in una **sequenza** di **tipi primitivi** di MPI, che indica anche lo spazio che ogni tipo primitivo occupa in memoria.

Possiamo creare nuovi tipi attraverso la funzione `MPI_Type_create_struct()`, che è definita come segue:

Code Manual

```
int MPI_Type_create_struct (
    int count,
    int array_of_blocklengths[],
    MPI_Aint array_of_displacements[],
    MPI_Datatype array_of_types[],
    MPI_Datatype* new_type_p
)
```

Nel seguente codice, vedremo come trasformare una `struct` di C in un tipo derivato:

CustomStruct.c

```
struct my_struct {
    double a;
    double b;
    int c;
}

// Trasformiamo my_struct in un tipo derivato
MPI_Aint a_addr, b_addr, c_addr;

MPI_Get_address(&a, %a_addr);
array_of_displacements[0] = 0;
MPI_Get_address(&b, %b_addr);
array_of_displacements[0] = b_addr - a_addr;
MPI_Get_address(&c, %c_addr);
array_of_displacements[0] = c_addr - b_addr;

// Facciamo il commit del tipo derivato
MPI_Type_commit()
```

Capitolo 3
PThreads

Con MPI, ci siamo sempre occupati di sistemi a memoria distribuita, dove i processi sono collegati da qualche mezzo (come un bus comune, o una connessione internet). Tuttavia come rappresentazione, questa è molto semplificata.

Ogni nodo (o server, o addirittura *blade*) contiene una memoria DRAM e una CPU multicore. In questa CPU, per ogni core sono presenti, assieme all'ALU, anche una cache L1 privata. La CPU ha anche assegnata una cache L2 comune. Assegnato al nodo, può esserci anche una, o più, GPUs (Graphics Processing Unit) e una, o più, NICs (Network Interface Card).

Come possiamo sfruttare un cluster di servers? Sappiamo che ogni server può comunicare attraverso MPI, ma come usarlo al meglio? Dovremmo scrivere un programma di MPI per ogni core? O per ogni nodo?

Di norma, si usa creare un processo MPI per ogni nodo, e poi internamente sfruttare librerie come PThreads e CUDA (per le GPU Nvidia) per gestire rispettivamente i core / threads delle CPU e i core della GPU. Il vantaggio di usare dei threads al posto di processi separati è che i **threads condividono la memoria DRAM** della CPU.

Ridefiniamo al volo cosa intendiamo con il termine **processo** e **thread**, così da poter distinguere la differenza tra un processo e un thread:

DEFINITION

Processo

Un **processo** è un'**istanza di computazione** di un programma che può essere in esecuzione, in stato di attesa o sospeso.

DEFINITION

Thread

Un **thread** è l'**istanza di computazione più piccola** e **indipendente** che può essere eseguita su un computer.

Sui sistemi UNIX, un processo si crea tramite la chiamata di sistema `fork()`. Facendo così, il processo padre viene duplicato e riassegnato al processo figlio. Quando viene creato il processo, viene copiato il codice del processo, la sua memoria, etc... Una volta copiata la memoria del processo, questa viene poi sovrascritta con il codice

Sempre sui processi UNIX, si usa la libreria PThreads, che sta per POSIX Threads. Questa libreria può essere utilizzata tramite C, includendola proprio come veniva fatto per MPI.

Con MPI dovevamo usare un wrapper specifico di `gcc`, ma con PThreads non è necessario: i threads infatti vengono creati automaticamente una volta lanciato il programma. La funzione per creare un thread è la seguente:

Code Manual

```
int pthread_create (
    pthread_t* thread_p,
    const pthread_attr_t* attr_p,
    void* (*start_routine) (void*),
    void* arg_p
)
```

Parametri di `pthread_create()`:

- `thread_p`: è un handle che rappresenta lo stato del thread. Tale handle è opaco, quindi non va modificato, e il puntatore va sempre allocato prima della chiamata della funzione. PThreads garantisce che tale handle contenga abbastanza informazioni per identificare il thread;
- `attr_p`: è un set di attributi, che non verranno coperti in queste note;
- `(*start_routine)`: è un puntatore a funzione, che definisce da dove il thread inizierà ad eseguire il codice;
- `arg_p`: è un puntatore ai parametri che la funzione `start_routine` richiede.

Un puntatore a funzione è semplicemente un indirizzo di memoria che indica il luogo in memoria dove è immagazzinata una funzione. Il nome di una funzione è utilizzabile per ottenere il puntatore della funzione stessa. Ad esempio:

FunctionPointer.c

```
1 void func(int a) {
2     printf("Hello world");
3 }
4
5 void main() {
6     // Riprendere dalle slides...
7 }
```

Le variabili globali sono visibili da tutti i threads, tuttavia è meglio evitarne l'uso a meno che non sia assolutamente necessario.

Per attendere la fine dell'esecuzione di un thread, si usa la funzione `pthread_join()`. Tale funzione richiede in input l'handle del thread, e aspetterà finché il thread non terminerà la sua esecuzione.

3.1 Concorrenza

Supponiamo di voler creare, similarmente a come è stato fatto per MPI, un programma che calcoli il vlaore di π tramite approssimazione. Proviamo dunque a scrivere una versione del codice con PThreads:

PthreadPi.c

```
1 // Put code...
```

Proviamo a vedere il valore delle varie approssimazioni in base al numero di threads t con cui eseguiamo il programma:

Threads	n			
	10^5	10^6	10^7	10^8
π	3,14159	3,141593	3,1415927	3,14159265
$t = 1$	3,14158	3,141592	3,1415926	3,14159264
$t = 2$	3,14158	3,141480	3,1413692	3,14164686

Come possiamo notare, più il numero di threads aumenta, più la stima si allontana dal valore reale. Questo però è controintuitivo, in quanto ci aspetteremmo che con più threads la stima dovrebbe essere più accurata. Quindi, come mai succede questo?

Il motivo sta nella riga di codice `sum += factor/(2 * i + 1)`, più nello specifico, nel fatto che la variabile `sum` è condivisa da tutti i threads. Quando si fa l'operazione `+=` non stiamo facendo solo una somma, ma stiamo eseguendo più istruzione.

3.1.1 Busy Waiting

3.1.2 Mutex

Un modo alternativo per far sì che l'accesso a una variabile sia controllato è usare una **mutex** (mutual exclusive). Le mutex esistono per ovviare al problema del busy waiting, cioè che la CPU continua a non eseguire nulla mentre controlla il valore di una variabile.

DEFINITION

Mutex

Una variabile **mutex** (mutually exclusive) è una variabile che **controlla l'accesso** a una sezione critica, e che permette a un solo thread alla volta di accedervi.

PThread implementa il concetto di mutex attraverso un tipo speciale: `pthread_mutex_t`. Le mutex sono implementate attraverso istruzioni di Assembly particolari, come ad esempio le `test/set`. Le istruzioni `test/set` sono istruzioni **atomiche** che riescono a controllare una variabile e impostarla a 1 quando essa è 0.

3.2 Semafori

temp

3.3 Barriere e Variabili di Confizioni

temp

3.4 Read-Write Locks

Quando si usa una struttura di dati condivisa, è importante controllare l'accesso a tale struttura. Consideriamo ad esempio una linked list **ordinata** senza doppi elementi (cioè una lista di elementi dove ogni elemento contiene un valore e un puntatore al prossimo elemento) dove le possibili operationi saranno `Member` (che controlla se un elemento è nella lista), `Insert` (che inserisce un elemento nella lista) e `Delete` (che rimuove un elemento).

Come possiamo far sì che questa lista sia utilizzabile in parallelo? Perché chiaramente scrivere o cancellare elementi da una lista necessita di una sezione critica. Finché più threads leggono solo la lista non è un problema, ma quando si fanno operazioni di scrittura allora si creano vari problemi.

La soluzione più semplice è di usare un lock sulla lista, cosicché se un thread deve usare la lista dovrà prima ottenere il relativo lock, per poi rilasciarlo una volta finito. Questo però serializza la lista, sia in lettura che in scrittura, e soprattutto se la maggior parte delle operazioni fosse di lettura (quindi con `Member`) allora perderemmo una chance di parallelizzare la lista. Nel caso in cui ad esempio si facessero più operazioni di scrittura che di lettura allora non sarebbe così tanto problematico. Chiaramente ci sarebbe un certo tempo necessario per acquisire il lock e per liberarlo, che non è negligibile.

Una seconda idea è quella di usare un lock per ogni nodo della lista, che nonostante sia più complesso da implementare, garantirebbe a tutti di poter usare la lista. Questo approccio però è parecchio lento, siccome ogni volta che bisogna accedere a un nodo serve acquisirne il relativo lock. Inoltre, a livello di memoria si va ad occupare parecchio spazio per immagazzinare ogni lock.

L'idea ottimale sarebbe quella di usare un lock solo in fase di scrittura, e non in fase di lettura. Questo tipo di lock esiste e si chiama **Read-Write Lock**.

DEFINITION

Read-Write Lock

Un **Read-Write Lock** è una **mutex** che permette di **differenziare** il **tipo di lock** in base all'operazione da fare. Pthreads implementa tale lock usando due funzioni: una per il **lock in lettura** e una per il **lock in scrittura**.

Come ogni lock, c'è una funzione per inizializzare il lock e una per distruggerla:

Code Manual

```
int pthread_rwlock_init (
    pthread_rwlock_t* rwlock,
    pthread_rwlockattr_t* attr
);


int pthread_rwlock_destroy (
    pthread_rwlock_t* rwlock
);
```

tbd

3.5 Thread Safety

3.5.1 Thread Safety su MPI

Abbiamo visto come MPI viene usato per fare programmi paralleli su memoria distribuita, ma se un programma di MPI viene chiamato su un solo sistema, ogni processo viene considerato come un thread. In tale caso, ci ritroveremmo in una situazione simile a PThreads, con dei problemi di race conditions. La domanda dunque sorge spontanea: le funzioni di MPI sono thread-safe? Ad esempio, potremmo usare, contemporaneamente, la funzione `MPI_Send()`?

La risposta è sì, ma non in tutti i casi. Se vogliamo usare dei threads su MPI, dobbiamo usare una funzione di inizializzazione diversa, chiamata `MPI_Init_thread`, che ha i seguenti parametri:

Code Manual

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

Parametri di `MPI_Init_thread()`:

- i parametri `argc` e `argv` sono gli stessi di `MPI_Init`;
- `required`: è il livello di threading richiesto dal programma;
- `provided`: è il livello di threading supportato dal programma.

I possibili livelli di threading

Capitolo 4
Architetture Multicore & OpenMP

Per spiegare un'architettura multicore ci vorrebbero parecchie pagine, per cui in questo capitolo ci concentreremo a vedere i punti salienti delle architetture multicore.

4.1 Organizzazione delle cache

Spesso si parla di cache quando si parla di processori, ma cosa intendiamo di preciso?

Definition

Cache

Una **cache** è una memoria, posizionata vicino ai core della CPU, che permette di accedere a dei dati a velocità molto più alte rispetto alla DRAM.

Le cache sono fatte con una tecnologia chiamata SRAM, che è molto più rapida della DRAM (la DRAM necessita di essere caricata e scaricata continuamente, risultando in tempi di richiesta molto più alti; la SRAM invece è composta da soli transistors, motivo della velocità della cache stessa). Le memorie in SRAM sono, tuttavia, molto costose, quindi di solito le cache in SRAM sono abbastanza piccole, se comparate alla memoria principale.

Il motivo del perché usiamo le cache è perché, quando si esegue un programma, c'è la possibilità che il programma, in un qualsiasi momento, richieda alcune istruzioni immagazzinate vicino all'istruzione che sta venendo eseguita. Parliamo dunque di **località temporale** e **località spaziale**.

I dati sono spostati verso la cache in blocchi (o linee), proprio per il concetto di località. Inoltre, è molto più rapido per una cache fare un'unico spostamento di n elementi piuttosto che fare n spostamenti di un elemento ciascuno.

All'interno di una CPU ci sono 3 tipi di cache, cioè la cache **Level 1** (o **L1**), **Level 2** (o **L2**) e **Level 3** (o **L3**). La cache L1 è la più rapida, ma anche la più piccola, mentre invece la cache L3 è la più lenta, ma anche la più spaziosa. In base al tipo di cache, un dato potrebbe essere immagazzinato sia in una cache (come ad esempio la L1) che in quelle sottostanti, ma generalmente dipende da come viene implementata.

Quando la CPU richiede un dato (diciamo il valore di una variabile `x`), allora quel dato verrà cercato nella cache L1; se non verrà trovato, la CPU cercherà poi il dato nella cache L2; se non venisse trovato cercherà poi nella cache L3 e infine, se non trovasse il dato in nessuna delle cache, lo cercherà nella memoria.

Ma come possiamo ottimizzare l'uso delle cache? Di norma, un programmatore non ha accesso alle cache, e non può specificare quale dato va dove. Quindi l'unico modo per ottimizzare un programma è tramite l'uso di varie flags di ottimizzazione di `gcc`.

4.1.1 Coerenza delle cache

Ma come funzionano le cache quando sono coinvolti più core in un programma? Supponiamo di avere un programma con due threads dove il thread 0 è assegnato al core 0, mentre il thread 1 è assegnato al core 1. Supponiamo inoltre che il core 0 abbia una variabile privata `y_0` e che il core 1 abbia invece le variabili private `y_1` e `z_1`; consideriamo inoltre una variabile condivisa `x = 2`. Consideriamo il seguente programma, che si diversifica in base al core:

Time	Core 0	Core 1
0	`y_0 = x;`	`y_1 = 3 * x;`
1	`x = 7;`	Istruzioni che non usano `x`
2	Istruzioni che non usano `x`	`z_1 = 4 * x;`

Ora, nella cache del core 0 abbiamo che `y_0 = 2` e `x = 7`, ma che cosa avremo nella cache del core 1? Siccome `x` viene modificato solo nella cache del core 0, questa modifica non arriva alla cache del core 1. Questo vuol dire che nella cache del core 1 avremo `y_1 = 6` e `z_1 = 8`, e non `y_1 = 6` e `z_1 = 28`.

Ci sono vari modi per risolvere questo problema, e uno di questi è lo **snooping cache coherence**. L'idea di base è che i core sono connessi tutti tramite un bus comune, che permette a tutti i core di vedere quali sono i dati che passano tramite tale bus. Ciò viene sfruttato dalle cache come canale di broadcast, per cui ogni volta che una cache viene aggiornata da un core, allora anche gli altri cores lo sapranno. Tuttavia, questo metodo non viene ormai più usato, e questo perché ormai, con processori con 64/128 cores, effettuare un broadcast è costoso.

Un altro metodo, più efficace, è il **directory based cache coherence**, per cui c'è una una struttura di dati, chiamata **directory**, che segna lo stato di ogni linea della cache, simile a una bitmap. Quando una variabile viene modificata, allora viene anche modificata la directory, e questa modifica viene propagata anche ai controllori di ogni cache, che invalideranno il contenuto nelle cache di ogni core dove la variabile non è aggiornata.

4.1.2 Organizzazione della memoria

Generalmente la memoria di un sistema può essere organizzata in due modi: può essere o **Uniform Memory Access** (**UMA**) o **Non-Uniform Memory Access** (**NUMA**). Per l'organizzazione UMA, i core condividono la stessa memoria, mentre per i NUMA, ogni gruppo di core ha una sua sezione di memoria. Con un sistema NUMA, bisogna fare attenzione a dove vengono allocati i dati, perché se dei dati che servono a un core vengono allocati nella parte di memoria nella quale il core non può accedere, allora si creano problemi di latenza: accedere a una memoria "locale" è infatti più economico che accedere a una memoria "remota". Tramite l'uso della libreria `numa.h` è possibile gestire la modalità di allocazione dei dati (in caso si può usare anche l'utilità da terminale `numactl`).

4.2 OpenMP

Un altro framework/API per creare applicazioni a memoria condivisa è **OpenMP**, che garantisce, rispetto a PThreads, un'interfaccia più ad alto livello. Il sistema su cui viene eseguito un programma di OpenMP è visto come una collezione di cores, dove ogni core ha accesso alla memoria principale.

Rispetto a PThreads, OpenMP cerca di eseguire un codice in maniera parallela partendo da un programma sequenziale, ed è molto più semplice da applicare rispetto al trasformare un programma con PThreads. Per poter usare OpenMP, serve usare delle **pragma** e un compilatore che supporti la libreria: questo perché ci sono alcune direttive che devono essere eseguite al momento di compilazione. Segue la definizione di una pragma:

DEFINITION

Pragma

Le **pragma** sono alcune **direttive** speciali che vengono **riconosciute** dal **preprocessore**. In `C`, una pragma ha la forma `#pragma`. Se un compilatore non supportasse le pragma di una libreria, allora le pragma verrebbero ignorate.

Le pragma possono usare delle **clausole**, ovverosia delle **direttive** che **modificano le pragma** e la loro esecuzione

La pragma più utilizzata di OpenMP è `#pragma omp parallel`, che dice al compilatore che il blocco di codice successivo va eseguito in parallelo. Per fare un esempio:

HelloWorld.c

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

void Hello(void) {
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();

    printf("Hello dal thread %d su %d\n", my_rank, thread_count);
}

int main(int argc, char** argv[]) {
    int thread_count = strtol(argv[1], NULL, 10);

    #pragma omp parallel num_threads(thread_count)
    Hello();

    return 0;
}
```

Possiamo compilare il codice tramite il seguente comando:

Terminal

```
$ gcc -g -Wall -fopenmp -o hello_world HelloWorld.c
$ ./hello_world 4
```

Terminal

```
Hello dal thread 1 di 4
Hello dal thread 0 di 4
Hello dal thread 2 di 4
Hello dal thread 3 di 4
```

Come possiamo impostare il numero di threads the eseguiranno il codice? Ci sono vari modi, che verranno ora elencati in ordine gerarchico (per primi quelli con gerarchia più bassa):

- In modo **universale**: attraverso l'impostazione della variabile d'ambiente (enviromental variable di UNIX) `OMP_NUM_THREADS`. Questa variabile può essere impostata attraverso il comando `export` di UNIX:

  Terminal

  ```
  $ echo $OMP_NUM_THREADS    # Per ottenere il valore
  $ export OMP_NUM_THREADS=<num_threads>     # Per salvare il valore
  ```

- A livello di **programma**: attraverso l'uso della funzione `omp_set_num_threads()` a inizio codice;
- A livello di **pragma**: usando la clausola `num_threads()` nella pragma.

Per sapere quanti threads stanno eseguendo una certa parte di codice, possiamo usare la funzione `omp_get_num_threads()` (se verrà eseguita in una porzione di codice sequenziale, ritornerà 1), mentre invece per avere il rank del thread bisognerà usare la funzione `omp_get_thread_num()`.

Una differenza sostanziale con PThreads è che OpenMP potrebbe, se lo ritiene necessario, creare dei threads fin dall'inizio dell'esecuzione del codice e lasciarli in `sleep` fino a che non servano, il che è sostanzialmente diverso da come funziona PThreads (può anche succedere, se una sezione di codice vada eseguita in sequenziale, che i threads vengano interrotti e messi in `sleep` fino al momento in cui servano nuovamente).

Come abbiamo detto, non tutti i compilatori di C supportano OpenMP, e se un compilatore non supportasse la libreria potremmo avere problemi se provassimo a chiamare delle funzioni della libreria stessa. Come possiamo risolvere questo problema? Possiamo usare delle pragma specifiche per definire dei corsi d'azione diversi in base al fatto che il compilatore supporti OpenMP o meno:

OpenMPCheck.c

```c
// Inclusione delle librerie...
#ifdef _OPEN_MP
#include <omp.h>
#endif

void parallelFunc() {
    // Nella funzione che verrà eseguita in parallelo

    #ifdef _OPEN_MP
        int my_rank = omp_get_thread_num();
        int thread_count = omp_get_num_threads();
    #else
        int my_rank = 0;
        int thread_count = 1;
    #endif

    // Altro codice...
}

int main() {
    // Codice del main...

    return 0;
}
```

Come possiamo garantire invece ciò che su PThreads avevamo con le mutex? Si può usare la pragma `#pragma omp critical`: tale pragma considererà il blocco sotto di essa come un blocco di codice che deve essere gestito da una mutex, e che quindi solo un thread alla volta potrà eseguire.

Possiamo anche usare un'altra pragma, che può essere usata solo in alcuni casi: la pragma `#pragma omp atomic`. Ci sono alcune operazioni che le CPU supportano che sono definite come **atomiche**: queste istruzioni permettono di **eseguire molteplici istruzioni** di Assembly **come una singola istruzione**. Ad esempio, se volessimo eseguire un'istruzione di C tipo `y += 1;`, allora le istruzioni che Assembly genererebbe sarebbero generalmente `LOAD`, `ADD` e `STORE`. Tuttavia, se l'addizione venisse supportata come operazione atomica dalla CPU, allora si potrebbe fare l'addizione, a livello Assembly, con una singola operazione. Un esempio segue:

Critical.c

```c
int y = 0;      // Variabile pubblica per tutti i threads
#pragma omp critical
{
    int my_rank = omp_get_thread_num();     // Variabile privata del thread
    y += my_rank;
}
```

Atomic.c

```c
// Tramite la pragma atomic sarebbe così:
int y = 0;

#pragma omp parallel
{
    int my_rank = omp_get_thread_num();

    #pragma omp atomic
    y += my_rank;
}
```

È importante far sì che l'istruzione che vogliamo eseguire atomicamente non richieda l'esecuzione di una funzione: se avessimo qualcosa tipo `y += sum_trapezoid()`, e se `sum_trapezoid()` richiedesse di fare vari calcoli che potrebbero comprendere all'interno sezioni critiche, allora la funzione verrebbe eseguita in parallelo da ogni thread senza controllo sulle sezioni critiche, e l'unica operazione atomica sarebbe la somma finale sulla variabile `y`.

4.2.1 Operazioni di riduzione

Anche OpenMP permette di avere operazioni di riduzione, che vengono specificate nelle pragma che specificano quali blocchi di codice vadano eseguiti in parallelo. Ad esempio, supponiamo di avere una funzione `local_sum(int a, int b, int n)`, e che dobbiamo far sommare in un'unica variabile `result` tutti i risultati di `local_sum()`. Potremmo fare qualcosa del genere:

Reductions.c

```c
int result = 0;
#pragma omp parallel
{
    #pragma omp critical
    result += local_sum(a, b, n);
}
```

Tuttavia, in questo modo tutti i threads eseguirebbero la funzione in sequenziale, e questo non sarebbe efficiente. Se ci salvassimo il risultato di `local_sum()` in una variabile privata, staremmo comunque facendo un'operazione di riduzione. Tuttavia, OpenMP ha un modo per definire delle operazioni di riduzione, tramite la seguente clausola:

```
reduction(<operatore>: <variabile_critica>)
```

Segue un esempio d'uso:

Reductions.c

```c
int result = 0;
#pragma omp parallel reduction(+: result)
result += local_sum(a, b, n);
```

È importante specificare la clausola `reduction`, altrimenti si avrebbero problemi di corsa critica.

4.2.2 Ciclo `for` parallelo

Uno dei motivi del perché OpenMP è molto utilizzato è anche grazie alla sua implementazione del `for` parallelo. Semplicemente, OpenMP esegue una `fork()` di un gruppo di threads, che eseguiranno il blocco del `for`, e poi divide le iterazioni tra i threads. L'idea è di usare la seguente pragma:

```
#pragma omp parallel for num_threads(<n_threads>) reduction(<op>: <var>)
```

L'uso del `for` parallelo è possibile solo in alcune situazioni, qui elencate:

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

Questo è richiesto perché prima di parallelizzare il ciclo `for`, OpenMP vuole sapere il numero di iterazioni che il ciclo `for` eseguirà. Inoltre, non si possono usare le istruzioni `break` e `return`, e non si può tantomeno manomettere l'index dell'iterazione del `for` loop. Si può tuttavia usare l'istruzione `exit`, che fa terminare l'esecuzione del programma.

Come ci comportiamo nel caso di cicli `for` annidati? In quel caso basterebbe mettere una pragma sopra il ciclo `for` più esterno, e OpenMP farebbe partire abbastanza thread per dare un numero equo di iterazioni a ogni thread, distribuendo le iterazioni in stile round robin. Questo è comodo se il numero di threads disponibili può dividere senza resto il numero di iterazioni del ciclo `for` più esterno. Tuttavia, se il numero non fosse divisibile, ci ritroveremmo un carico di lavoro sbilanciato fra i threads. Supponiamo di avere il seguente codice:

OpenMPNestedFor.c

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

In questo caso, se il numero di threads disponibili fosse 4, allora avremmo il thread 3 che non eseguirebbe nulla. Come possiamo fare? Un'idea potrebbe essere quella di mettere la pragma legata al ciclo interno, ma in quel caso creeremmo delle bolle tra i vari threads. Infatti, la pragma distribuirebbe 6 iterazioni tra 4 threads (per 3 volte), lasciando due thread con un'iterazione in più, e siccome prima di passare alla prossima iterazione del ciclo più esterno bisognerebbe aspettare che le iterazioni del ciclo più interno finiscano, avremmo dei problemi.

C'è la possibilità di unire i due cicli, tramite una direttiva ulteriore che viene aggiunta alla pragma. Questa direttiva è `collapse()`, e si usa nel seguente modo:

OpenMPNestedForCollapsed.c

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

Un'operazione che OpenMP vieta è l'usare due pragma di parallelizzazione per i cicli `for`. Ad esempio:

OpenMPDisabledNestedFor.c

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

4.2.3 Dipendenze dei dati

C'è una cosa da tenere bene a mente quando usiamo il ciclo `for` parallelo: ci potrebbero essere delle dipendenze tra i dati. Ad esempio, quando calcoliamo la sequenza di Fibonacci, noi abbiamo bisogno, per calcolare un numero i, del numero $i-1$ e $i-2$. Infatti, avremmo un codice del genere:

OpenMPFibonacci.c

```
fibo[0] = fibo[1] = 1;
for (int i = 2; i < n; i++) {
    fibo[i] = fibo[i - 1] + fibo[i - 2];
}
```

Noi non potremmo usare la pragma per parallelizzare il `for` loop perché non avremo la garanzia che il risultato sia corretto.

4.2.4 Scheduling dei loop

Una volta che decidiamo di voler parallelizzare un ciclo `for`, come possiamo distribuire le varie iterazioni ai vari nodi? Ci sono vari modi: il modo più classico è quello di distribuire le iterazioni **a blocchi** (quindi, se abbiamo n iterazioni e k nodi, ad ogni nodo assegnamo una quantità sequenziale di n/k iterazioni). Tuttavia, un altro modo che possiamo utlizzare è quello di distribuire le iterazioni **ciclicamente**, ovverosia alla round robin. In questo modo ogni thread esegue comunque n/k iterazioni, ma saranno tutte sparse in modo diverso. Questo torna comodo ad esempio quando le ultime iterazioni di un ciclo sono quelle più costose a livello di calcolo.

Per poter specificare lo scheduling dei loop, serve usare la clausola

```
schedule(type, chunksize)
```

all'interno della pragma. La clausola specifica, tramite il parametro `type`, di assegnare staticamente le iterazioni ai threads, e ne assegna `chunksize` alla volta ad ogni thread prima di passare al prossimo thread a cui assegnare.

La clausola ammette diversi valori di `type`:

- `static`: le iterazioni vengono assegnate ai threads prima dell'esecuzione delle iterazioni stesse;
- `dynamic` o `guided`: le iterazioni vengono assegnate ai threads durante l'esecuzione delle esecuzioni, quindi quando un thread si libera gli viene assegnata un'iterazione. Lo scheduling `guided` è simile al `dynamic`, ma riduce, ad ogni chunk assegnato, il numero di iterazioni assegnate ai threads. Possiamo specificare comunque un numero minimo di iterazioni da assegnare ad ogni thread, così da evitare overhead;
- `auto`: il tipo di scheduling viene assegnato al runtime dallo scheduler o dal compiler;
- `runtime`: il tipo di scheduling viene determinato al runtime.

Il valore di `chunksize` è importante perché permette di distribuire più granularmente le iterazioni a ogni thread, ed è sempre un valore intero positivo. Bisogna sempre considerare bene il valore di `chunksize`: se troppo piccolo, avremo un maggiore overhead per inizializzare il `for` parallelo, mentre se troppo grande, allora caricheremmo troppo il peso dei singoli thread.

Un altro modo di specificare lo scheduling di OpenMP è esportando una variabile d'ambiente chiamata `OMP_SCHEDULE`. Ad esempio, potremmo specificare la seguente:

Terminal

```
$ gcc -g -Wall -fopenmp -o hello_world HelloWorld.c
$ ./hello_world 4
```

```
Hello dal thread 1 di 4
Hello dal thread 0 di 4
Hello dal thread 2 di 4
Hello dal thread 3 di 4
```

CAPITOLO 5

Programmazione di GPU e CUDA

Una componente essenziale in tutti i computer è la CPU, che è nata essenzialmente per eseguire calcolo sequenziale nel più breve tempo possibile. Fin dalla loro creazione, l'idea era di ottimizzare quanto più possibile l'efficienza e la riduzione della latenza. Un computer necessita di una CPU per poter funzionare, ma non necessita strettamente invece di una GPU.

DEFINITION

Graphics Processing Unit

Una **Graphics Processing Unit** (**GPU**) è una componente ideata per condurre tasks di grafica, ideata per eseguire operazioni in parallelo

La struttura di una GPU è molto diversa da quella di una CPU: quest'ultima infatti è pensata per eseguire codice sequenziale, non parallelo. Una GPU è composta dalle seguenti componenti:

- una sezione ridotta di cache, che puntano a massimizzare il throughput della memoria;
- una sezione massiva di ALUs (più lenti di quelli nelle CPU), che però sono collegati da una pipeline pemsata apposta per svolgere compiti in parallelo;
- una memoria di tipo HBM (High Bandwidth Memory) chiamata VRAM (**Video RAM**).

Su questo corso ci concentreremo sulle architetture delle GPU CUDA, create da NVIDIA. In questo tipo di architettura ci sono vari cores, chiamati **streaming processors** (**SP**), che sono uniti in più blocchi, chiamati **streaming multiprocessors** (**SM**). Ogni streaming multiprocessor è controllato da una stessa unità di controllo: questo vuol dire che ogni streaming processor in uno stesso multiprocessor eseguirà la stessa istruzione di Assembly in un dato momento.

Ci sono alcuni caveat quando si programmano le GPU: la memoria della GPU e quella della CPU sono diverse, e quindi per poter usare un dato sulla GPU che risiede nella memoria della CPU, questo dato andrà prima caricato sulla memoria della GPU.

5.1 CUDA

Abbiamo spesso parlato di CUDA, ma di cosa parliamo quando usiamo questo termine? CUDA è un acronimo che sta per **Compute Unified Device Architecture**, che intende generalmente l'architettura delle schede GPU di NVIDIA. CUDA offre due API diverse, una più low level e una più high level per i 3 grandi sistemi operativi.

CUDA va oltre il semplice linguaggio di programmazione: è una **piattaforma di calcolo** a tutti gli effetti. Le piattaforme CUDA vengono considerate come **co-processori** della CPU, non come sostituzioni. Infatti, per funzionare, la GPU necessita della CPU.

Al giorno d'oggi ci sono due grandi competitors nel mercato delle GPU: NVIDIA e AMD. Al momento, NVIDIA è la società più quotata, ma anche AMD offre hardware performante. ADM offre un'API proprietara chiamata HIP, che nella maggior parte delle istruzioni è simile a CUDA. Ci sono anche altre due API open source, chiamate OpenCL e OpenACC.

CUDA organizza i suoi thread in una struttura a sei dimensioni: ogni thread è posizionato in un blocco che va da una a tre dimensioni, e ogni blocco è posizionato in una griglia che va anch'essa da una a tre dimensioni. Dunque ogni thread può sapere la sua posizione in questo spazio a sei dimensioni, chiamando opportune funzioni offerte dall'API di CUDA.

Ogni piattaforma CUDA ha una sua capacità di calcolo, che viene chiamata **CUDA compute capability**. La capacità viene rappresentata da un numero di versione chiamato **SM version**.

5.1.1 Scrivere un programma di CUDA

La struttura di un programma CUDA è la seguente:

1) **allocare la memoria** sulla piattaforma CUDA;
2) **trasferire i dati** dalla DRAM alla VRAM della piattaforma;
3) **eseguire un kernel CUDA**, ovverosia una funzione;
4) una volta terminato il kernel, serivrà **copiare i dati** processati dalla VRAM della piattaforma nella DRAM.

Per scrivere un programma di CUDA bisogna specificare una funzione, chiamata **kernel**, che andrà eseguita su tutti i threads. Chiaramente non è pensabile specificare, thread per thread, cosa ogni thread dovrà eseguire, qundi l'idea è quella di specificare come arrangiare i threads in blocchi e griglie. Di norma, la GPU ha già una sua struttura di blocchi e griglie, tuttavia noi abbiamo la possibilità di scegliere una nostra struttura che sia compatibile con l'architettura della nostra GPU.

Per eseguire una funzione su una GPU NVIDIA, servirà specificare appunto la dimensione dei blocchi e della griglia, e ciò si fa attraverso le seguenti linee di codice:

CUDAStartKernel.c

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