

PROGRAMMAZIONE DI SISTEMI EMBEDDED E MULTICORE

Leonardo Biason

November 13, 2024

CHAPTER 1

Programmazione Parallela

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 **GPUs** (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.

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.

CHAPTER 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
1 #include <stdio.h>
2
3 int main() {
4     printf("Hello world");
5     return 0;
6 }
```

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
1 #include <stdio.h>
2 #include <mpi.h>
3
4 int main(void) {
5     // Per usare MPI, serve usare una funzione chiamata MPI_INIT;
6     int r = MPI_Init(NULL, NULL);
7
8     if(r != MPI_SUCCESS) {
9         printf("C'è stato un errore con il programma");
10        MPI_Abort(MPI_COMM_WORLD, r);
11    }
12
13    printf("Hello world");
14 }
```

```
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**.

Rank

DEFINITION

Il **rank** di un processo appartenente a un programma di MPI è un **indice incrementale**, nell'intervallo $[0, 1, 2, \dots, p)$, 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
12    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

```
MPIRankAndSize.c
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 ≥ 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?

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:

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

Speed-up

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

Scalabilità

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

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.

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:

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

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

Legge di Gustafson

DEFINITION

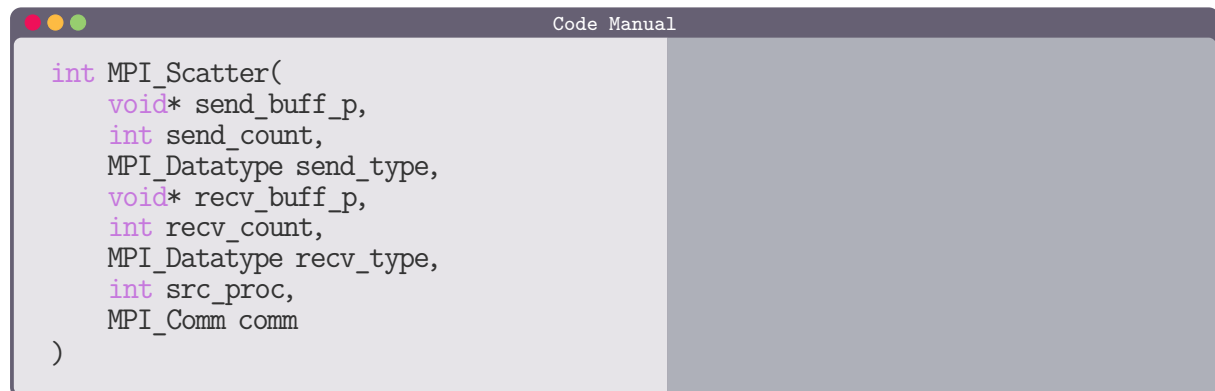
La **legge di Gustafson** definisce ciò che si chiama **speed-up scalabile**, e che prende in assunzione che, se si **aumenta** il **numero di processi** p per una costante α , allora anche la **dimensione del problema** n **aumenta** di α .

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

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:



```
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 derivati.

Tipi Derivati

DEFINITION

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()
```

I

CHAPTER 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:

Processo

DEFINITION

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

Thread

DEFINITION

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, similamente a come è stato fatto per MPI, un programma che calcoli il valore 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 istruzioni.

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.

Mutex

DEFINITION

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 Confini

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 operazioni 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.

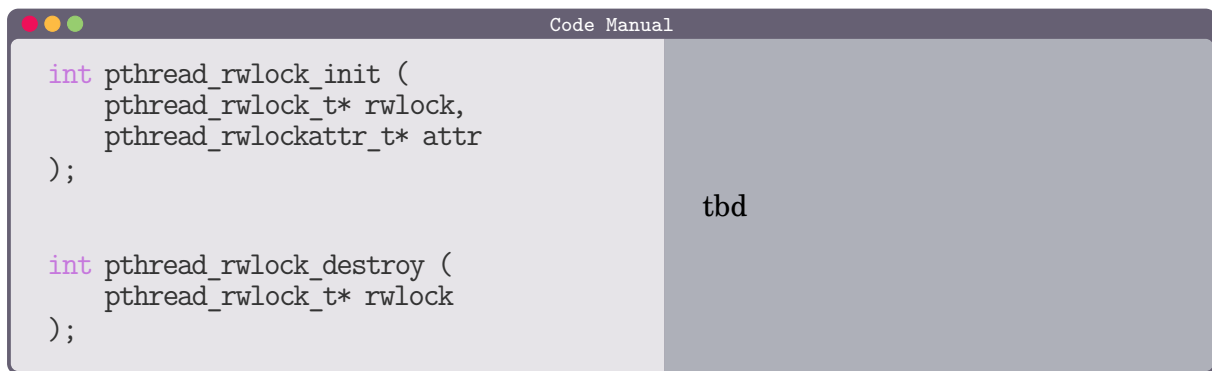
L'idea ottimale sarebbe quella di usare un lock solo in fase di scrittura, e non in fase di lettura. Questo tipo di lock esiste e si chiama **Read-Write Lock**.

Read-Write Lock

DEFINITION

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:

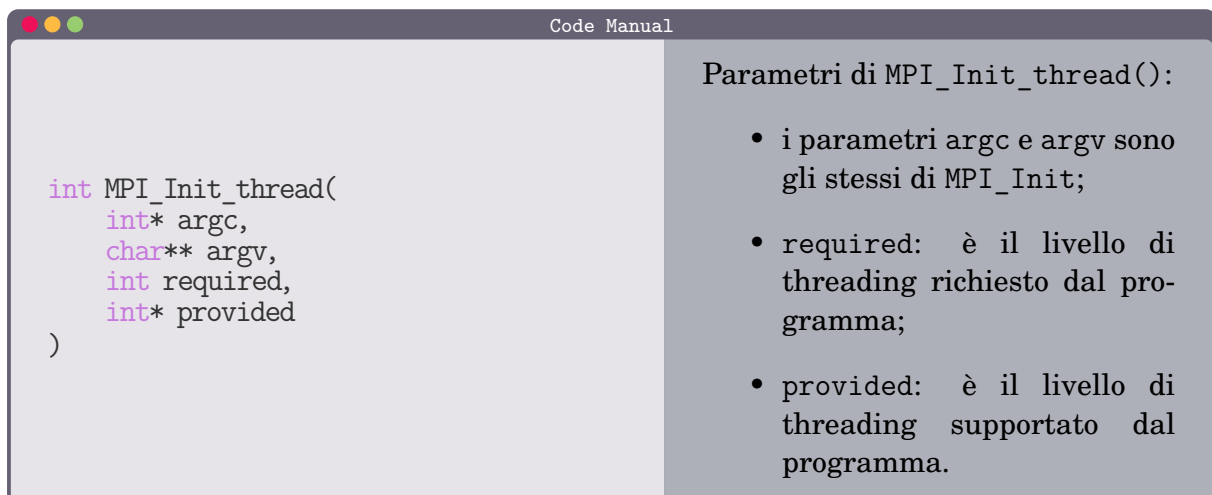


3.5 Thread Safety

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:



I possibili livelli di threading

CHAPTER 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?

Cache

DEFINITION

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 transistor, 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 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:

Pragma

DEFINITION

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

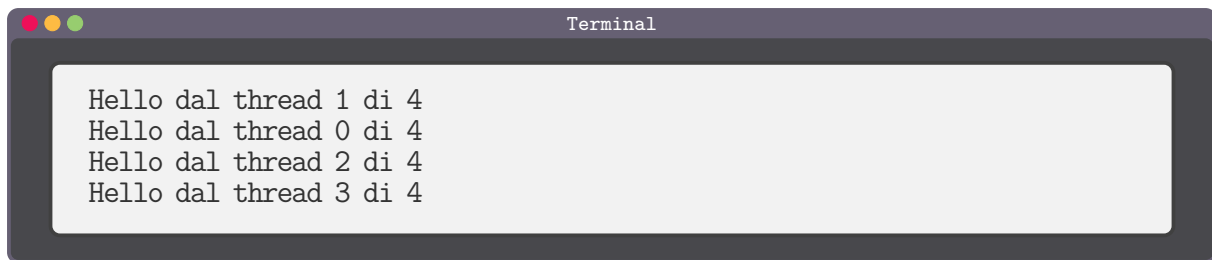
Le pragma possono usare delle **clausole**, ovverosia delle **direttive** che **modificano le pragma** e la loro esecuzione

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
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <omp.h>
4
5 void Hello(void) {
6     int my_rank = omp_get_thread_num();
7     int thread_count = omp_get_num_threads();
8
9     printf("Hello dal thread %d su %d\n", my_rank, thread_count);
10 }
11
12 int main(int argc, char** argv) {
13     int thread_count = strtol(argv[1], NULL, 10);
14
15     #pragma omp parallel num_threads(thread_count)
16     Hello();
17
18     return 0;
19 }
```

Possiamo compilare il codice tramite il seguente comando:

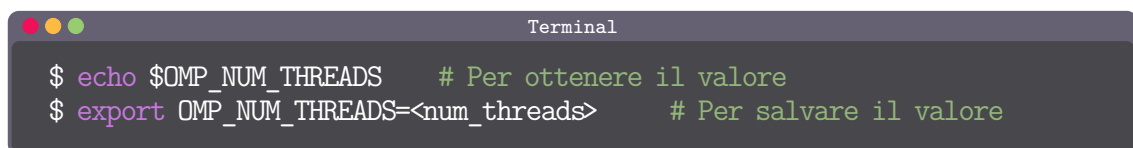
```
Terminal
$ gcc -g -Wall -fopenmp -o hello_world HelloWorld.c
$ ./hello_world 4
```



```
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 che 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 (environmental 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
1 // Inclusione delle librerie...
2 #ifndef _OPEN_MP
3 #include <omp.h>
4 #endif
5
6 void parallelFunc() {
7     // Nella funzione che verrà eseguita in parallelo
8
9     #ifdef _OPEN_MP
10         int my_rank = omp_get_thread_num();
11         int thread_count = omp_get_num_threads();
12     #else
13         int my_rank = 0;
14         int thread_count = 1;
15     #endif
16
17     // Altro codice...
18 }
19
20 int main() {
21     // Codice del main...
22
23     return 0;
24 }
```

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 avessimo un'istruzione di C tipo `y += 1;`, allora le istruzioni che Assembly farebbe 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
1 int y = 0; // Variabile pubblica per tutti i threads
2 #pragma omp critical
3 {
4     int my_rank = omp_get_thread_num(); // Variabile privata del thread
5     y += my_rank;
6 }
```

```

Atomic.c
1 // Tramite la pragma atomic sarebbe così:
2 int y = 0;
3
4 #pragma omp parallel
5 {
6     int my_rank = omp_get_thread_num();
7
8     #pragma omp atomic
9     y += my_rank;
10 }

```

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
1 int result = 0;
2 #pragma omp parallel
3 {
4     #pragma omp critical
5     result += local_sum(a, b, n);
6 }

```

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 clausola `reduction(<operatore>: <variabile>)`.

```

Reductions.c
1 int result = 0;
2 #pragma omp parallel reduction(+: result)
3 result += local_sum(a, b, n);

```

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