

HTML1E4C86

\*About these notes

Those notes were made during my three years of university at Sapienza, and **do not** replace any professor, they can be

\*License

The decision of licensing this work was taken since these notes come from **university classes**, which are protected, in

\*Bibliography & References

9

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

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

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

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

```
while (i <= a) { printf("i++;
```

```
i = a; while (i >= 0) { printf("i--;
```

```
int main() printf("Hello world"); return 0; Per rendere questo Hello World un programma parallelo tramite MPI, serve
```

```
ParallelHelloWorld.c [language = C] include <stdio.h> include <mpi.h>
```

```
int main(void) // Per usare MPI, serve usare una funzione chiamata MPI_Init; intr = 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");
```

```
// Per terminare l'esecuzione di tutti i threads si usa MPI_Finalize MPI_Finalize(); return 0;
```

Nel precedente codice sono state usate alcune funzioni e alcuni valori di MPI, che possiamo notare grazie al prefix "MPI".

**MPI\_Init():** **inizializza** un programma su più processi o threads, e restituisce come output un **int**, che identifica se è stato inizializzato correttamente. **MPI\_SUCCESS:** è il segnale con cui è possibile comparare l'output di **MPI\_Init** per controllare se MPI è stato inizializzato correttamente. **MPI\_Abort(MPI\_COMM\_WORLD, <mpi\_boot\_result>):** **abortisce** l'esecuzione di MPI, ad esempio nel caso in cui l'inizializzazione fallisce. **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 tipico è:

```
[style = notexterm] mpicc < file > .c -o < output > mpicc -g -Wall <file>.c -o <output> Fa stampare i warning in caso di errore.
```

Il compilatore ha molte flags che possono essere usate, così da personalizzare il processo di compilazione. Nel secondo capitolo si vedranno alcune di queste flags.

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

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.

language = C, numbers = none, int pthread\_create(pthread\_t \*thread, const pthread\_attr\_t \*attr, void \* (\*start\_routine)(void \*), void \*)

Per spiegare un'architettura multicore ci vorrebbero parecchie pagine, per cui in questo capitolo ci concentreremo a ved

Organizzazione delle cache

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

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

```
pragma omp parallel int my_rank = omp_get_threadnum();  
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 una funzione che fa operazioni di riduzione

Anche OpenMP permette di avere operazioni di riduzione, che vengono specificate nelle pragma che specificano quali blocchi di codice fanno operazioni di riduzione. Ad esempio:

```
Reductions.c [language = 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 fosse una funzione che fa operazioni di riduzione

Segue un esempio d'uso:

```
Reductions.c [language = 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.

Scheduling dei loop

Una volta che decidiamo di voler parallelizzare un ciclo **for**, come possiamo distribuire le varie iterazioni ai vari nodi?

**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 iterazioni, quindi quando un thread termina di eseguire una iterazione, il scheduler assegna una nuova iterazione a un altro thread;

**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 utile per controllare il carico di lavoro di ogni thread.