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

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 = 0;

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, ser

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$_SUCCESS) printf("C'statounerroreconilprogramma"); MPI_Abort(MPI_COMMWORLD, r);$

printf("Hello world");

// Per terminare l'esecuzione di tutti i threads si usa MPI$_F inalize MPI_F inalize(); return 0;$

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 stat
`MPI_SUCCESS`: è il segnale con cui è possibile comparare l'output di `MPI_Init` per controllare se MPI è stato inizializzato co
`MPI_Abort(MPI_COMM_WORLD, <mpi_boot_result>)`: **abortisce** l'esecuzione di MPI, ad esempio nel caso in cui l'inizializza
`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 coman

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

Il compilatore ha molte flags che possono essere usate, così da personalizzare il processo di compilazione. Nel secondo c

Processo Un **processo** è un'**istanza di computazione** di un programma che può essere in esecuzione, in stato di atte

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

Sui sistemi UNIX, un processo si crea tramite la chiamata di sistema `fork()`. Facendo così, il processo padre viene dup

[language = C, numbers = none] int pthread_create(pthread_t * thread, const pthread_attr_t * attr, void * (*start_routine)

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$_rank = omp_get_thread_num()$;
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 av
Operazioni di riduzione
Anche OpenMP permette di avere operazioni di riduzione, che vengono specificate nelle pragma che specificano quali b
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

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.
Ciclo `for` parallelo
Uno dei motivi del perché OpenMP è molto utilizzato è anche grazie alla sua implementazione del `for` parallelo. Sempl

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

Questo è richiesto perché prima di parallelizzare il ciclo `for`, OpenMP vuole sapere il numero di iterazioni che il ciclo `f`
OpenMPNestedFor.c [language = C] pragma omp parallel for for (int i = 0; i < 3; ++i) for (int j = 0; j < 6; ++j) c(
In questo caso, se il numero di threads disponibili fosse 4, allora avremmo il thread 3 che non eseguirebbe nulla. Come
OpenMPNestedForCollapsed.c [language = C] pragma omp parallel for collapse(2) for (int i = 0; i < 3; ++i) for (int j
Un'operazione che OpenMP vieta è l'usare due pragma di parallelizzazione per i cicli `for`. Ad esempio:
OpenMPDisabledNestedFor.c [language = C] pragma omp parallel for for (int i = 0; i < 3; ++i) pragma omp parallel
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 d
OpenMPFibonacci.c [language = 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 corr
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 esecuzioni, quindi quando un 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 s