UNIVERSITÀ DI PARMA

DIPARTIMENTO DI SCIENZE MATEMATICHE, FISICHE ED INFORMATICHE
Corso di Laurea in Informatica

Programmazione a memoria condivisa con openMP

Programmazione parallela e HPC - a.a. 2023/2024

Roberto Alfieri

Programmazione Parallela e HPC: sommario

PARTE 1 - INTRODUZIONE

PARTE 2 – SISTEMI PER IL CALCOLO AD ALTE PRESTAZIONI

PARTE 3 – PERFORMANCE DELL'HARDWARE

PARTE 4 – PROGETTAZIONE DI PROGRAMMI PARALLELI

PARTE 5 – PROGRAMMAZIONE A MEMORIA CONDIVISA CON OPENMP

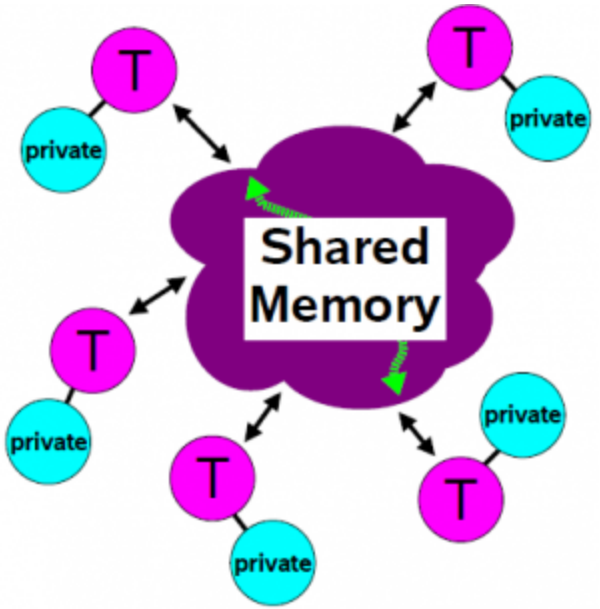
PARTE 6 – PROGRAMMAZIONE A MEMORIA DISTRIBUITA CON MPI

PARTE 7 – PROGRAMMAZIONE GPU CON CUDA

openMP

openMP (Open specifications for Multi Processing) è un modello di programmazione parallela a memoria condivisa.
E' gestito dal consorzio no profit *OpenMP Architecture Review Board* (or OpenMP ARB) assieme ad un gruppo di produttori di Hardware e di sviluppatori di software.
Sito ufficiale: https://www.openmp.org/

Aggiunge costrutti di programmazione multithreading per C, C++ e Fortran.
Unico codice seriale/parallelo: utilizzabile anche dove non c'è supporto per openMP



Tutorial e link esterni: LLNL - LearnMicrosoft

Versioni openMP

Versioni rilevanti di openMP:

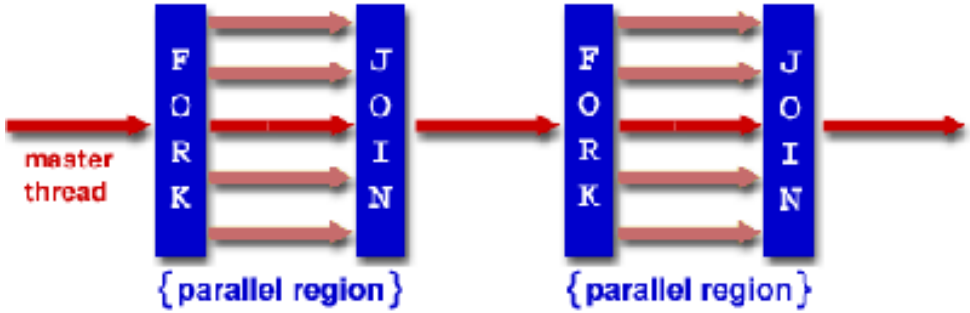
Versione	Rilascio	Caratteristiche
openMP 3.1	07/2011	
openMP 4.0	07/2013	Supporto per thread affinity, direttiva SIMD Direttiva target per acceleratori (GPU?)
openMP 4.5	11/2015	
openMP 5.0	11/2018	
openMP 5.2	11/2021	Versione corrente
openMP 6.0	??/2024	

openMP: modello di esecuzione

L'esecuzione inizia con un singolo thread (master thread)

openMP interviene quando serve il parallelismo e utilizza il modello fork/join grazie a specifiche direttive che attivano la regione parallela (fork) e la disattivano al termine dell'esecuzione parallela quando tutti i thread hanno completato il loro compito (Join all)

```
//  master thread
#pragma omp parallel
{
// parallel region
}
//  master thread
```



openMP fornisce al programmatore **direttive, funzioni e variabili d'ambiente** per la gestione delle regioni parallele, il timing e il sincronismo.

openMP: funzioni

L’uso delle funzioni rende il codice dipendente da openMP.

Le principali funzioni sono:

FUNZIONE	
omp_set_num_threads()	Imposta il numero di thread nelle aree parallele future, a meno che non venga sottoposto a override da una clausola num_threads .
omp_get_num_threads()	Restituisce il numero di thread nell'area parallela.
omp_get_max_threads()	Restituisce un numero intero uguale o maggiore del numero di thread che sarebbero disponibili se un'area parallela senza num_threads fosse definita in quel punto del codice.
omp_get_thread_num()	Restituisce il numero di thread del thread in esecuzione all'interno del team di thread.
omp_get_wtime()	Restituisce un valore in secondi del tempo trascorso da un certo punto.
omp_set_lock()	Blocca l'esecuzione del thread fino a quando non è disponibile un blocco.
omp_set_unlock()	Rilascia un blocco.
omp_test_lock()	Tenta di impostare un blocco ma non blocca l'esecuzione del thread.

Dettagli: https://learn.microsoft.com/it-it/cpp/parallel/openmp/reference/openmp-functions

openMP: numero di threads e identificazione

Esistono 3 modi per impostare il numero di thread.

1) La variabile d'ambiente OMP_NUM_THREADS può contenere un intero che viene utilizzato se non sono presenti altre indicazioni.
Esempio: **OMP_NUM_TREADS=4 a.out**
2) All'interno del programma possiamo chiamare la funzione omp_set_num_threads()
Esempio: **omp_set_num_threads(4);**
3) Il numero di thread può essere impostato per ogni regione parallela con la clausola num_threads (num). Questo metodo è prioritario. Esempio:

```
#pragma opm parallel num_threads(8)
{ .. }
```

In un team di thread ogni thread è identificato da un valore numerico da 0 a N-1.
Un thread può conoscere il proprio identificatore con la funzione omp_get_thread_num()
Esempio: **t=omp_get_thread_num();**
Il master thread ha valore 0.
Se la funzione viene chiamata al di fuori di una regione parallela restituisce 0.

openMP: Le variabili

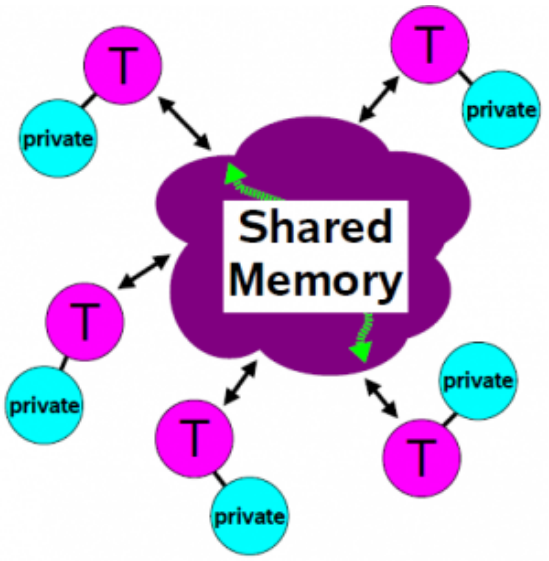
La variabili di un thread in un programma openMP possono essere shared o private.

Variabili shared: La variabili dichiarate esternamente alla regione parallela sono per default shared. Ogni thread può leggere e modificare le variabili shared.

Variabili private: Sono localizzate nello stack del thread e non sono inizializzate
Le variabili dichiarate all'interno di una regione parallela sono private

Quando viene definita una nuova regione parallela è possibile modificare lo scopo delle variabili con le clausole **shared (var list) e private (var list)** Esempio:

```
int share1, share2;
# pragma opm parallel private (share2) shared (priv2)
{
int priv1, priv2;
..
}
```



openMP: direttiva parallel e clausole

La direttive gestiscono il comportamento delle regioni parallele.
Il comportamento di un direttiva può essere modificato da opportune clausole.

```
#pragma omp <direttiva> [<elenco clausole>]
{
// regione governata dalla direttiva
}
```

Per creare una regione parallela si usa la direttiva **parallel**. Esempi di causole di questa direttiva sono shared() private() num_threads() e reduction() . La clausola reduction() determina l'operazione di riduzione di una variabile. Esempio somma:

```
int t;
#pragma opm parallel reduction (+:t)
{
t=omp_get_thread_num();
}
```

All'uscita della regione parallela il thread master riceve e somma i valori t da tutti gli altri thread.

openMP: altre direttive

La direttive posso essere nested; la direttiva parallel è la più esterna.

Altri direttive importanti sono:

Direttiva	
for	Fa in modo che il lavoro eseguito in un for ciclo all'interno di un'area parallela venga diviso tra i thread.
section	Identifica le sezioni di codice da dividere tra tutti i thread.
single	Consente di specificare che una sezione di codice deve essere eseguita su un singolo thread, non necessariamente il thread principale
master	Specifica che solo il thread principale deve eseguire una sezione del programma.
critical	Specifica che il codice viene eseguito solo su un thread alla volta.
barrier	Sincronizza tutti i thread in un team; tutti i thread si sospendono sulla barriera, fino a quando tutti i thread non eseguono la barriera.

https://learn.microsoft.com/it-it/cpp/parallel/openmp/reference/openmp-directives?view=msvc-170

openMP: direttiva for

Fa in modo che il lavoro eseguito in un for ciclo all'interno di un'area parallela venga diviso tra i thread. E' la direttiva che implementa la decomposizione di dominio: tutti i thread svolgono le stesse operazioni su dati differenti.

```
#pragma omp parallel
#pragma omp for [clauses]
    for_statement
```

Le direttive **parallel** e **for** possono essere accorpate: **#pragma omp parallel for [clauses]**

La clausola **schedule** consente di scegliere il **chunk** (numero cicli eseguiti consecutivamente da un thread) e il tipo di schedulazione che può essere **static** (i chunk ruotano staticamente tra i thread) o **dynamic** (il primo thread libero si prende carico del prossimo chunk). Esempio:

```
 #pragma omp parallel for schedule (dynamic,5)
for( )   { … }
```

Per default lo schduling è static con un chunk pari a numero_iter / numero_thread

https://learn.microsoft.com/it-it/cpp/parallel/openmp/reference/openmp-directives?view=msvc-170#for-openmp

openMP: direttiva sections

Identifica le sezioni di codice da dividere tra tutti i thread.

E' la direttiva che implementa la **decomposizione funzionale.**

Le diverse sezioni sono identificate da #pragma omp section

Esempio:

```
#pragma omp sections
{
  #pragma omp section
  printf("Section1 executed by thread %d\n", omp_get_thread_num());

  #pragma omp section
  printf("Section2 executed by thread %d\n", omp_get_thread_num());
}
```

https://learn.microsoft.com/it-it/cpp/parallel/openmp/reference/openmp-directives?view=msvc-170#sections-openmp

openMP: direttive single e master

La direttiva SINGLE consente di specificare che una sezione di codice deve essere eseguita su un singolo thread, non necessariamente il thread principale.
Questa direttiva ha una **barriera implicita** all'uscita: i thread che non entrano attendono comunque all'uscita.
Per specificare che una sezione di codice deve essere eseguita solo nel thread principale, usare invece la direttiva MASTER.

Esempio:

```
#pragma omp parallel
{
  #pragma omp single
  printf ("Solo il primo thread che arriva entra, gli altri non entrano ma attendono all'uscita");
  // barriera implicita

  #pragma omp master
  printf ("Solo il thread master entra, gli altri  proseguono senza attendere");
}
```

https://learn.microsoft.com/it-it/cpp/parallel/openmp/reference/openmp-directives?view=msvc-170#single

openMP: direttive critical e atomic

Critical specifica che il codice viene eseguito solo su un thread alla volta.

Esempio:

```
#pragma omp parallel
{
  #pragma omp critical
  printf("tutti I thread entrano, ma un per volta");
}
```

Atomic specifica che una posizione di memoria che verrà aggiornata in modo atomico.

```
int count=0;
#pragma omp parallel
{
  #pragma omp atomic
  count++;
}
```

https://learn.microsoft.com/it-it/cpp/parallel/openmp/reference/openmp-directives?view=msvc-170#critical

openMP: direttiva barrier

Sincronizza tutti i thread in un team; tutti i thread si sospendono sulla barriera, fino a quando tutti i thread non eseguono la barriera.

Esempio:

```
#pragma omp parallel
{
  printf (“la barriera si sblocca quando tutti I thread l’hanno raggiunta”);
 #pragma omp barrier
}
```

https://learn.microsoft.com/it-it/cpp/parallel/openmp/reference/openmp-directives?view=msvc-170#barrier

openMP: thread affinity

Un thread in esecuzione su un determinato core potrebbe essere spostato run-time su un core diverso dalla schedule del sistema operativo, invalidando i dati nella cache.
La CPU affinity consente di legare un processo o thread ad un determinato core fisico.
openMP dalla ver.4 supporta la thread affinity tramite 2 variabili d'ambiente

- **OMP_PLACES** definisce il livelli di affinity che può essere *socket, core o thread*:
  - socket : i thread si possono muovere all'interno del socket
  - core : i thread si posso muovere l'interno degli hyper-thread di un core
  - thread: thread legati ad uno specifico hyper-thread
- **OMP_PROC_BIND** consente di impostare il modello di affinity:
  - False : affinity disabilitata
  - True: affinity abilitata
  - Close : thread posizionati tra core vicini (i core sono numerati progressivamente)
  - Spread: thread posizionati tra core distanti

Esempio:

OMP_PLACES=socket OMP_PROC_BIND=close ./a.out

Si può visualizzare lo stato dell'affinity con la funzione omp_get_proc_bind().

https://theartofhpc.com/pcse/omp-affinity.html