Computação Paralela

Mest. Engenharia Computacional
Mest. Int. Engenharia Computacional

Ano letivo 2022/2023

Rui Costa, Nuno Lau

OpenMP

- **Open** *specifications for* **Multi Processing** *via collaborative work between interested parties from the hardware and software industry, government and academia*

Acknowledgement: This lecture is based on "A 'Hands-on' Introduction to OpenMP", Tim Mattson, Intel Corp., timothy.g.mattson@intel.com

OpenMP

- **Shared Memory Programming Model**
- Cooperation of several hardware and software companies (AMD, Intel, arm, Fujitsu, IBM, HP, NASA, NEC, NVIDIA, Siemens, SUSE, …)
- Parallel Programming API for multiprocessor / multicore architectures
- Languages: C/C++ or Fortran
- OS: Unix/Linux or Windows
- **OpenMP is a specification not an implementation!**

OpenMP objectives

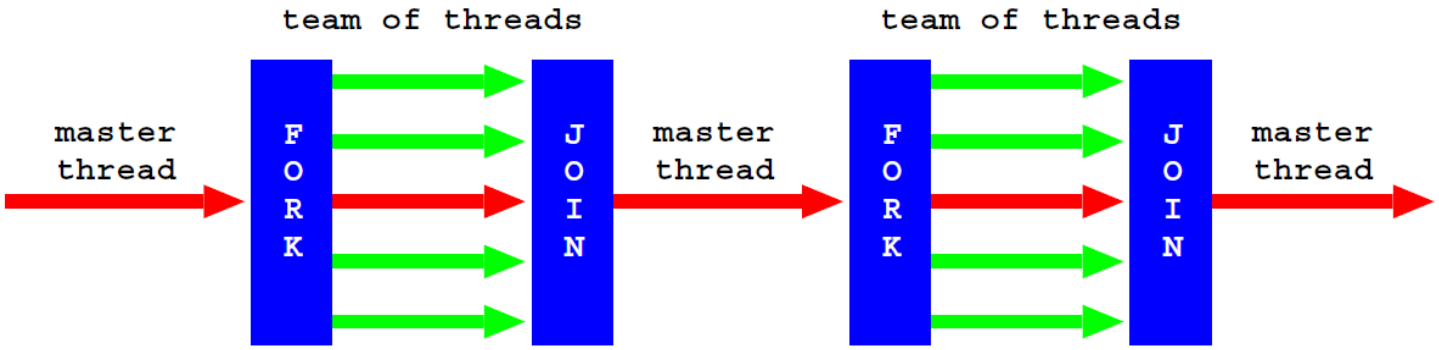
- Portable programming model for shared memory architectures
- Simple set of programming directives
- Enable incremental parallelism for sequential programs
- Efficient implementations for different problems

OpenMP Programming model

- Explicit parallelism
  - Programmer is responsible for annotating execution tasks and synchronization points
  - Embedded compiler directives
- Implicit multithreading
  - Process is a set of threads that communicate through shared variables
  - Creation and termination of threads is performed implicitly by the execution environment
    - Global address space is shared by all threads
    - Variables may be shared or private for each thread
    - Control, management and synchronization of variables involved in parallel tasks is transparent to the programmer

OpenMP fork-join execution

- Program initiates with a single (master) thread
- Executes sequentially until parallel region is defined by OpenMP constructor, and then:
  - Master thread forks “team of threads”
  - Parallel region code is executed concurrently by all threads (including master thread)
  - There is an implicit barrier at the end of parallel region
  - “team of threads” terminates and master thread continues sequentially



From CP@FCUP

OpenMP API

- API includes
  - Compiler directives: **`#pragma omp <directive>`**
  - Library of functions declared at **`omp.h`**
  - Environment variables (**`OMP_NUM_THREADS`**; …)
- OpenMP code
  - Must include **`omp.h`** header file

    ```
    #include <omp.h>
    ```
  - To compile with **`gcc`** use **`-fopenmp`** option

    ```
    gcc –fopenmp myprog.c …
    ```

omp parallel directive

`#pragma omp parallel [clause, …]`

- Defines parallel region
- Number of threads determined by:
  - Only master if **`if(expr)`** clause is used and **`expr`** is false
  - Number defined by **`num_threads(expr)`** clause
  - Number defined by last call to **`omp_set_num_threads(expr)`**
  - Number defined by environment variable **`OMP_NUM_THREADS`**
  - Implementation dependent

Hello.c

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

int main ()
{
    #pragma omp parallel
    {
    int ID = omp_get_thread_num();

    printf("Hello (%d) ", ID);
    printf("World (%d) \n", ID);
    }
}
```

OpenMP basic functions

- **`int omp_get_thread_num(void);`**
  - Returns thread **id** from **0** to **n-1**, where **n** is the number of threads
  - Master thread has id **0**
- **`int omp_get_num_threads(void);`**
  - Returns the number of active threads
- **`void omp_set_num_threads(int num_threads);`**
  - Specifies the maximum number of threads to be used in the next parallel regions
  - Can only be called from sequential code

Shared and private variables

```
#pragma omp parallel shared(list1) private(list2)
```

- Shared variables
  - Shared by all threads
  - Beware of race conditions!
- Private variables
  - Duplicated in each thread
  - Initial value is undefined
  - Value after parallel region is undefined
- Variables are shared by default
  - Can be changed by using **`default(mod)`** clause
    - **`mod`**: **`private`**, **`shared`**, **`none`**
- Variables declared inside parallel region are private

Shared and private variables

```
int n = ...;
int *v = (int *) malloc(sizeof(int) * n);
int i, j;
#pragma omp parallel num_threads(n) default(none) \
                     shared(v,n) private(i,j)
{
  int tid = omp_get_thread_num();
  j = 0;
  for (i = 0; i < n * 100000; i++) j += tid;
  v[tid] = j;
}
```

- private variables: **i**, **j** and **tid**
- shared variables: **v** and **n**
- Which is the final value of **v[tid]**?

Fistprivate variables

```
#pragma omp parallel firstprivate(list)
```

- **`firstprivate`** variables
  - Duplicated in each thread
  - Initial value is copied from value of variable with the same name
  - Can be used to improve efficiency
    - Latency of reading private variable may be lower than latency of reading shared variable by all threads

Example: Dot product

```
int dot_product(int* u, int* v, int n) {
    int r = 0;
    for (int i = 0; i < n; i++) {
        r += u[i] * v[i];
    }
    return r;
}
```

- Partition iterations / vector among threads
- Reduction of partial results

Example: Dot product

First version (incorrect!):

```
int dot_product(int* u, int* v, int n) {
  int r = 0;
  #pragma omp parallel
  {
    int rank = omp_get_thread_num();
    int my_n = n / omp_get_num_threads();
    for (int i = rank * my_n; i < (rank+1)*my_n; i++)
      r += u[i] * v[i];
  }
  return r;
}
```

- Manual partition
- Error prone

Example: Dot product

Better version:

```
int dot_product(int* u, int* v, int n) {
  int r = 0;
  #pragma omp parallel
  {
    #pragma omp for reduction(+:r)
    for (int i = 0; i < n; i++)
      r += u[i] * v[i];
  }
  return r;
}
```

- **`#pragma omp for`**
  - distributes iterations among threads
- Reduction is automatically performed by using **`reduction`** clause
- Scheduling may be controlled through **`schedule`** clause

Example: Dot product

Even better version:

```
int dot_product(int* u, int* v, int n) {
  int r = 0;
  #pragma omp parallel for reduction(+:r)
  for (int i = 0; i < n; i++) {
    r += u[i] * v[i];
  }
  return r;
}
```

- **`#pragma omp parallel for`**
  - creates parallel region and distributes iterations among threads

OpenMP Reduction

- OpenMP **`reduction`** clause:
  - **`reduction (op : list)`**
- Inside a parallel or a work-sharing construct:
  - A local copy of each list variable is made and initialized depending on the “op” (e.g. 0 for “+”).
  - Updates occur on the local copy.
  - Local copies are reduced into a single value and combined with the original global value.
- The variables in “list” must be shared in the enclosing parallel region.

```
double ave=0.0, A[MAX]; int i;
#pragma omp parallel for reduction (+:ave)
for (i=0;i< MAX; i++) {
   ave + = A[i];
}
ave = ave/MAX;
```

OpenMP Reduction

Operator	Initial value
+	0
*	1
-	0
min	Largest pos. number
max	Most neg. number

C/C++ only

Operator	Initial value
&	1
|	0
^	0
&&	1
||	0

OpenMP Reduction

```
int a = 0, b = 5, c = -2, d = 3;
#pragma omp parallel num_threads(3) \
            reduction(+:a,b) reduction(min:c,d)
{
   a = b = c = d = omp_get_thread_num();
}
```

- Final values a=3, b=8, c=-2, d=0