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*

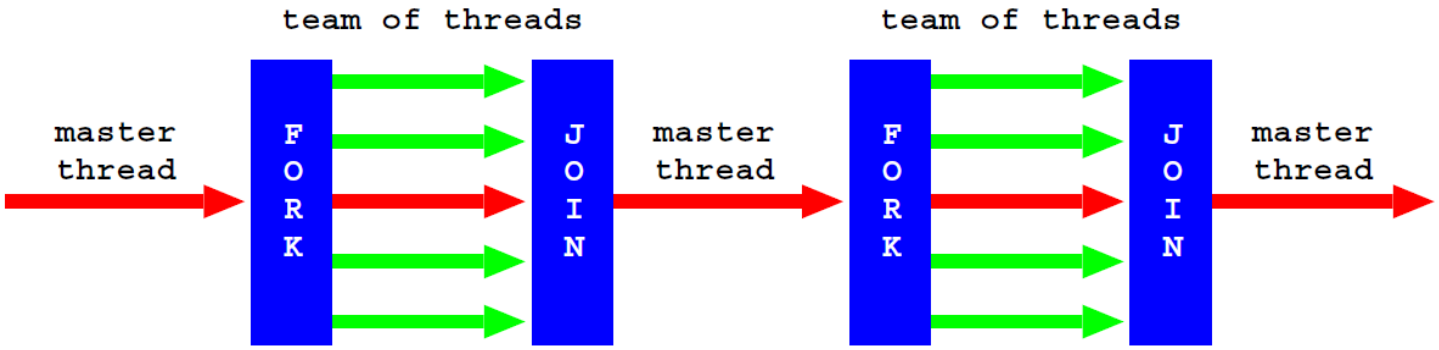
Acknowledgement: This lecture is based on "A 'Hands-on' Introduction to OpenMP", Tim Mattson, Intel Corp., timothy.g.mattson@intel.com

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 fork-join execution

- Program initiates with a single (master) thread
- Executes sequentially until parallel region 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 Mutual Exclusion

- OpenMP has constructs to enable mutual exclusion among threads
- 3 different ways:
  - **`omp critical`**
    - Defines critical region
    - Can be used for any code
  - **`omp atomic`**
    - Mutual exclusion for atomic variable updates
    - Only for simple memory updates
      - x *binop*= *expr*; x++; ++x; x--; --x
  - Explicit use of locks

OpenMP Mutual Exclusion

- **omp critical**

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

OpenMP Mutual Exclusion

- **omp atomic**

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

OpenMP Mutual Exclusion

- Locks

```
int dot_product(int* u, int* v, int n) {
  int r = 0;
  omp_lock_t lock;
  omp_init_lock(&lock); // initialize lock
  #pragma omp parallel
  {
    int private_r = 0;
    #pragma omp for
    for (int i = 0; i < n; i++)
      private_r += u[i] * v[i];
    omp_set_lock(&lock); // acquire lock
    r += private_r;
    omp_unset_lock(&lock); // release lock
  }
  omp_destroy_lock(&lock); // destroy lock
  return r;
}
```

OpenMP other synchronization

- **`omp master`**
  - Executed only be master thread
  - Ignored by others; no synchronization
- **`omp single`**
  - Executed by a single thread
  - Implicit barrier for all threads
    - Except **`nowait`** clause is used
- **`omp barrier`**
  - Explicit barrier

OpenMP other synchronization

- **`omp master`**
  - Executed only be master thread
  - Ignored by others; no synchronization
- **`omp single`**
  - Executed by a single thread
  - Implicit barrier for all threads
    - Except **`nowait`** clause is used
- **`omp barrier`**
  - Explicit barrier

OpenMP **`schedule`** clause

- **`schedule(static[,chunk])`**
  - Each thread has fixed number of iterations
- **`schedule(dynamic[,chunk])`**
  - Each thread grabs “chunk” iterations off a queue until all iterations have been handled
- **`schedule(guided[,chunk])`**
  - Threads dynamically grab blocks of iterations
  - The size of the block starts large and shrinks down to size “chunk” as the calculation proceeds.
- **`schedule(runtime)`**
  - Schedule and chunk size taken from the OMP_SCHEDULE environment variable (or the runtime library).
- **`schedule(auto)`**
  - Schedule is left up to the runtime to choose (does not have to be any of the above).

`nowait ordered lastprivate` clauses

- **`nowait`**
  - threads do not need to synchronize at the end of cycle
- **`ordered`**
  - Allows #pragma ordered blocks that must execute in the cycle iteration order
- **`lastprivate(list)`**
  - Variables in list are private
  - Their last value is transported to after the cycle

`nowait ordered lastprivate` clauses

```
#pragma omp for nowait
for (int i = 0; i < n; i++) a[i] = x*i;
#pragma omp for
for (int i = 0; i < n; i++) b[i] = i*i;
#pragma omp for
for (int i = 0; i < n; i++) c[i] = a[i] + b[i];
#pragma omp for ordered lastprivate(x)
for (int i = 1; i < n; i++)
#pragma omp ordered
{
  c[i] += c[i-1]; x = c[i];
  printf("c[%d] = %d\n", c[i]);
}
```

- threads may go to 2nd cycle even before all threads terminated 1st cycle (**`nowait`**)
- In the last cycle, each thread only executes after previous iterations are concluded (**`ordered`**)
- **`x`** gets the value of c[n-1] (**`lastprivate`**)

OpenMP `omp sections` directive

```
#pragma omp sections [clause, ...]
{
  #pragma omp section
  { ... }
  ...
  #pragma omp section
  { ... }
}
```

- The **`omp sections`** directive defines a set of code sections that can perform concurrently, allowing functional parallelism between sections.
- Each section is identified by an **`omp section`** directive and is performed by just one thread.
- All threads, including those that are not involved in any section, synchronize at the end of the **`omp sections`** region, unless the **`nowait`** clause is specified.

OpenMP `omp sections` directive

```
x = f1();
a = f2(x);
b = f3(x,1);
c = f4(x,a);
d = f5(a,b);
e = f6(d);
```

- Assuming that **`f1`** to **`f6`** can execute concurrently correct, how to parallelize the code?
- Dependencies allow for the following processing order:
  1. **`x = f1()`** for just one thread.
  2. **`a = f2(x)`** and **`b = f3(x, 1)`** for 2 threads in parallel.
  3. **`c = f4(x, a)`** and **`d = f5(a, b)`** for 2 threads in parallel.
  4. **`e = f6(d)`** for only one thread.

OpenMP `omp sections` directive

```
#pragma omp parallel
{
  #pragma omp single
  x = f1();
  #pragma omp sections
  {
    #pragma omp section
    a = f2(x);
    #pragma omp section
    b = f3(x,1);
  }
  #pragma omp sections
  {
    #pragma omp section
    c = f4(x, a);
    #pragma omp section
    d = f5(a,b);
  }
  #pragma omp single
  e = f6(d);
}
```

OpenMP `task`

- A task has
  - Code to execute
  - Data environment (it owns its data)
  - An assigned thread that executes the code and used the data
- Two activities: packaging and execution
  - Each encountering thread packages a new instance of a task (code and data)
  - Some thread in the team executes the task at some later time

OpenMP `task`

- Tasks have been fully integrated into OpenMP
- Key concept: OpenMP has always had tasks, we just never called them that.
  - Thread encountering parallel construct packages up a set of implicit tasks, one per thread.
  - Team of threads is created.
  - Each thread in team is assigned to one of the tasks (and tied to it).
  - Barrier holds original master thread until all implicit tasks are finished.
- We have simply added a way to create a task explicitly for the team to execute.
- Every part of an OpenMP program is part of one task or another!

OpenMP **`task`** example

- **`omp task`** used to process elements of a linked list:

```
#pragma omp parallel
{
  #pragma omp single private(p)
  {
     p = listhead ;
     while (p) {
       #pragma omp task
       process (p)
       p = next(p) ;
     }
  }
}
```