Parallel Computing
Introduction to OpenMP

Marco Grossi
mgrossi@ectstar.eu

Ricerca ECT*, FBK Trento

Secondo corso di formazione
"Calcolo Parallelo su Grid (CSN4cluster)"
September 26th-28th, 2011
Parma

Thread vs. process

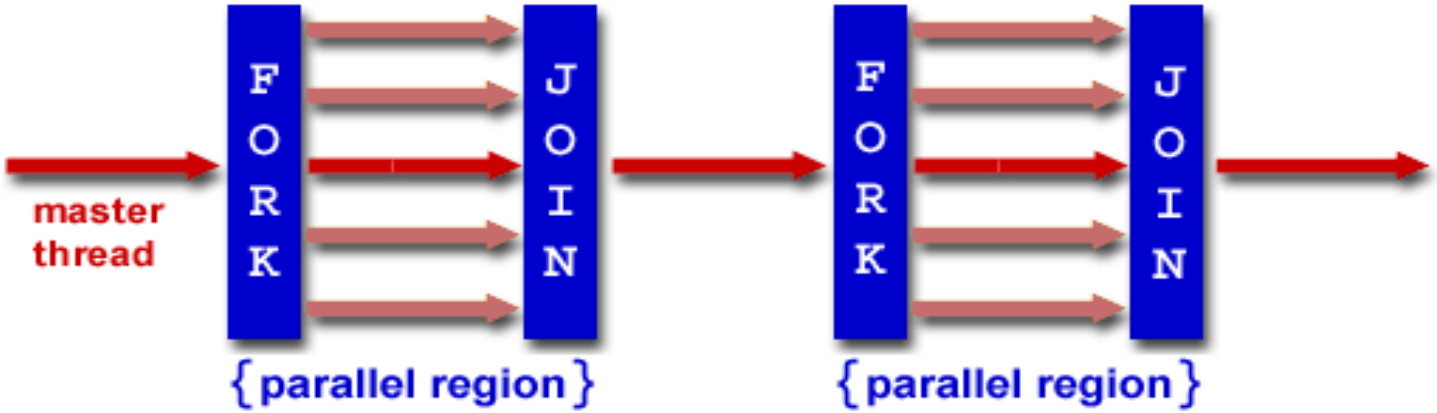
- Different processes have different memory space
- A process can create more independent execution flow, named threads, that share the same memory space, open files, …
- Message passing between threads it's faster than between processes
- Concurrency on shared data structure must be carefully designed and managed
- Pthread it's a POSIX standard for threads

Open Multi-Processing(OpenMP)

- Compiler directives and library calls for multi-threaded programming
  - Easy to create threaded C/C++ and Fortran codes
  - Explicit parallelization
    - Especially oriented for loop parallelization
  - Supports the data parallelism model for shared memory paradigm(but also task parallelism)
  - Offers incremental parallelism
  - Combines serial and parallel code in a single source

OpenMP – Programming model

- Fork-Join parallelism
  - When you start your OpenMP program exists only one thread: the master thread
  - The master thread create a set of worker thread that remain in a sleeping state until the program flow reach a parallel region
  - Inside the parallel region the workers(all or a subset) and master are executing in parallel
  - After the parallel region the worker set return to a sleeping state and the master continue to execute



OpenMP – Directives, routines and environment

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

int
main (int argc, char *argv[]) {

  int th_id, nthreads;

  #pragma omp parallel private(th_id)
  { /* begin of parallel region */
    th_id = omp_get_thread_num();
    printf("Hello World from thread %d\n", th_id);

    #pragma omp barrier

    if (th_id == 0) {
      nthreads = omp_get_num_threads();
      printf("There are %d threads\n", nthreads);
    }
  } /* end of parallel region; implicit barrier */

  exit(0);
}
```

```
gcc -W -Wall --pedantic -std=c99 -fopenmp hello.c -o hello_world
export OMP_NUM_THREADS=4
./hello_world
```

Environment

Directive

Routine

OpenMP - Terminology

- **Variables** can be:
  - **private**: each thread has it's own private copy
  - **shared**: unique copy available to all threads
- OpenMP team: **master + workers**
  - The master thread always has thread ID 0
- A **parallel region** is a block of code executed by all threads simultaneously
- A **work-sharing construct** divides the execution of the enclosed code region among the members of the team

OpenMPI – Work sharing: for

```
#pragma omp parallel for
{
  for (i=0; i < N; ++i) {
    c[i] = a[i] + b[i];
  }
}
```

As the same as

```
#pragma omp parallel
{
  #pragma omp for
  for (i=0; i < N; ++i) {
    c[i] = a[i] + b[i];
  }
}
```

The iteration variable **i** it's by default private

All other variables are considered shared

```
#pragma omp parallel for private(j)
{
  for (i=0; i < N; ++i) {

    for (j=0; j < N; ++j) {
      c[i] = a[i] + b[i];
    }

  }
}
```

This is the code of a single iteration of the external for

The iteration variable **j** must be declared private

OpenMP – For: reduction

```
#pragma omp parallel for reduction(+:result)
{
  for (i=0; i < N; ++i)
    result += a[i] + b[i];
}
```

The variable `result` it's automatically initialized to zero

Reduction operators (init value may vary):

`+ * - & | ^ && ||`

OpenMP – For: schedule types

```
#pragma omp for schedule(kind[,chunk_size])

    kind in {static, dymamic, guided, auto, runtime}
```

- `static`
  - Iterations are divided into chunks of size `chunk_size`, and the chunks are assigned to the threads in the team in a round-robin fashion in the order of the thread number.
- `dynamic`
  - Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be distributed.
- `guided`
  - Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be assigned. The chunk sizes start large and shrink to the indicated `chunk_size` as chunks are scheduled.

OpenMP – NUMA memory allocation

- We are in the NUMA-age, so we need a memory allocation as near as possible to the thread that have to access that area
- There is a simple trick
  - Request memory with `posix_memalign` or similar
  - With the same ”for cycle” and scheduler that we will use later for distribute load between thread we ”touch” the allocated area with the initial value
    - Only now the memory it's allocated and normally placed on the same memory node were the thread will execute
- Some compiler implement an extension that provide explicit directives for memory and cpu affinity
- A more complex procedure will use the `numalib` for memory affinity and other functions(e.g. `pthread_setaffinity_np`) for CPU affinity

OpenMP – Single and master directives

- Single
  - the associated structured block is executed by only one of the threads in the team (not necessarily the master thread)
- Master
  - specifies a structured block that is executed by the master thread of the team. There is no implied barrier either on entry to, or exit from, the master construct

OpenMP – Synchronization directives

- Critical
  - the enclosed code block will be executed by only one thread at a time, and not simultaneously executed by multiple threads. It is often used to protect shared data from race conditions.
- Atomic
  - the memory update (write, or read-modify-write) in the next instruction will be performed atomically. It does not make the entire statement atomic; only the memory update is atomic. A compiler might use special hardware instructions for better performance than when using critical.
- Ordered
  - the structured block is executed in the order in which iterations would be executed in a sequential loop

OpenMP – Synchronization directives

- Barrier
  - each thread waits until all of the other threads of a team have reached this point. A work-sharing construct has an implicit barrier synchronization at the end.
- Nowait
  - specifies that threads completing assigned work can proceed without waiting for all threads in the team to finish. In the absence of this clause, threads encounter a barrier synchronization at the end of the work sharing construct.

OpenMP - Example

```
#pragma omp parallel private(th_id)
{
  th_id = omp_get_thread_num();

  #pragma omp critical
  {
    /* Executed by all threads, but only one at a time */
    printf("Hello World from thread %d\n", th_id);
  }

  #pragma omp barrier

  #pragma omp master
  {
    /* Only executed by the master thread */
    nthreads = omp_get_num_threads();
    printf("There are %d threads\n", nthreads);
  }
    result += a[i] + b[i];
}
```

OpenMP – Section

- We have a set of structured block that could be executed in parallel
- OpenMP provide the `sections` and `section` directive
- Each single section will be executed by only one thread

```
#pragma omp parallel
{
  #pragma omp sections
  {
   #pragma omp section
    {X_calculation();}

   #pragma omp section
    {Y_calculation();}

   #pragma omp section
    {Z_calculation();}

  } /* end of sections; implicit barrier */

  […]
}
```

`sections`, plural

`section`, singular

Wait a minute

- Keep the parallel section as huge as possible, in orther to minimize the wake-up time of sleeping worker thread
- Before parallelize a computational code block, verify if it's safe to execute that code in parallel
  - In a code like that
    - `new[i, j] = update_operator(old[i,j])`
  - if `new` and `old` are different memory area, and the `update_operator` does not modify any other data structure → this may be ok
  - But if the code it's like that
    - `update_operator_inplace(array[i, j])`
  - there are a lot of warnings that arise

False sharing

- When you write to a memory location, all entry of that location in the various level of cache must be invalidated(cache coherency)
- The granularity of the invalidation it's a cache line, e.g. 64Byte
- If more than one thread share the same cache line(e.g. the first thread use the first 32Byte and the former the latter 32Byte), and at least one of that thread it's writing to that cache line, you will lead to the problem named ”false sharing”
  - In orther to avoid that you have to split the workload between thread with a granularity of at least cache line size
  - If you need to share some reduce variable, keep a private copy to each thead and execute the reduce at the end of the parallel execution

Hybrid programming: MPI + OpenMP

- Using OpenMP inside a computing node, and MPI between the nodes, we can:
  - reduce the communication costs inside the node
  - reduce the size of data that we exchage with the other nodes
  - reduce the cost of collective calls

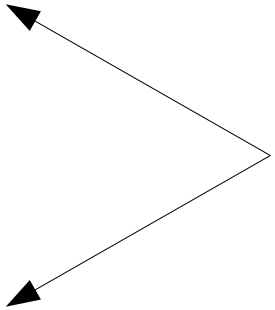
- In orther to avoid possible NUMA memory mis-allocation you can:
  - allocate an MPI process for each CPU socket
  - choose the number of OpenMP thread for each MPI process as the number of core available on the CPU socket

Hybrid programming: MPI + OpenMP

```
int MPI_Init_thread(int *argc, char ***argv,
                    int required, int *provided)

This call initializes MPI in the same way that a call to MPI_Init would.
In addition, it initializes the thread environment.
The argument required is used to specify the desired level of thread
support.

The possible values are listed in increasing order of thread support.

MPI_THREAD_SINGLE
  Only one thread will execute.

MPI_THREAD_FUNNELED
  The process may be multi-threaded, but the application must ensure
  that only the main thread makes MPI calls.

MPI_THREAD_SERIALIZED
  The process may be multi-threaded, and multiple threads may make
  MPI calls, but only one at a time.

MPI_THREAD_MULTIPLE
    Multiple threads may call MPI, with no restrictions.
```

Hybrid programming: MPI call throw master

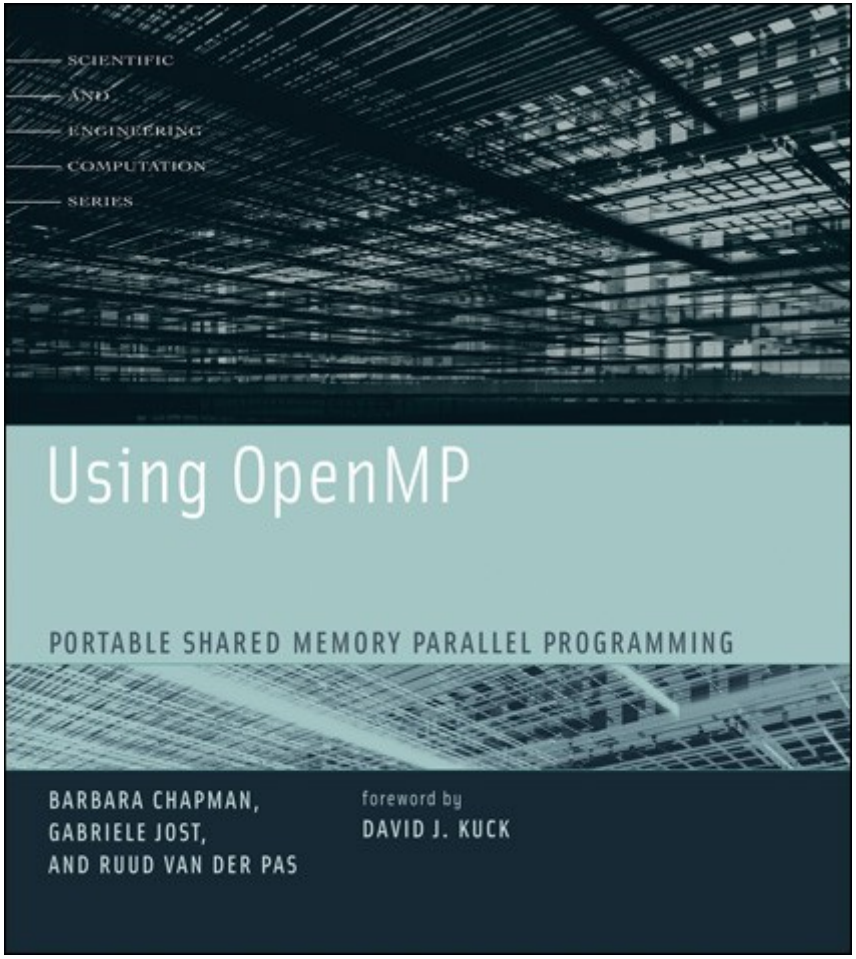
```
[…]

#pragma omp parallel
{
  do_parallel_calc();

  #pragma omp barrier
  #pragma omp master
  {
    MPI_xxx(...);
  }
  #pragma omp barrier

  […]
}

[…]
```



Barriers are mandatory here !!!

References

- https://computing.llnl.gov/tutorials/openMP/
- http://openmp.org/wp/openmp-specifications/



Your questions & hints

Thank you for your attention!
For any questions and hints
please send an email to

`mgrossi@ectstar.eu`