Introduction to OpenMP using Fortran

by Youjun Hu

Institute of Plasma Physics, Chinese Academy of Sciences
Email: yjhu@ipp.cas.cn

1 OpenMP "Hello world"

The following is an OpenMP hello-world program written in Fotran:

```
program hello_openmpi
   implicit none
!$omp parallel
   print*, "Hello world"
!$omp end parallel
end program
```

The lines beginning with `!$omp` are OpenMP directives which are parsed only by Fotran compilers with OpenMP enabled. Fortran compilers with OpenMP disabled consider these lines as comments. Save this file as `hello.f90` and then compile it with the following command line:

```
gfortran -fopenmp hello.f90 -o a.out
```

where we use the `gfortran` compiler and enable OpenMP with `-fopenmp` option.

The OpenMP directive pair `!$omp parallel`/`!$omp end parallel` mark a parallel region in the source code. New threads are created when the master thread enters this parallel region. Then the new threads together with the master thread concurrently execute the codes within the parallel region.

Run the above executable `./a.out`, we get something like:

```
Hello world
Hello world
Hello world
Hello world
```

The number of "Hello world" we get depends on the default number of threads on our operating system. This number can be changed by setting the OpenMP environment variable `OMP_NUM_THREADS` via the following command line (for bash shell):

```
export OMP_NUM_THREADS=8
```

which sets the number to 8. After this, running `a.out` will print 8 lines of "Hello world".

2 OpenMP directive for threads creation

The OpenMP directive pair `!$omp parallel`/`!$omp end parallel` mark a parallel region in the source code. New threads are created when the master thread enters this parallel region. In other words, the directive-pair `!$omp parallel`/`!$omp end parallel` manages the creation of new threads.

Within the parallel region, if no further OpenMp directives are used, then all threads do exactly the same thing, which is usually not what we want. OpenMP provides further directive-pairs that can be used within the parallel region to further control how the work are shared among threads. The next section discusses these OpenMP directives for working-sharing.

3 OpenMP directives for work-sharing

The following is an example:

```
!$omp parallel
!$OMP DO
do i=1,1000
  a(i)=2*i
enddo
!$OMP END DO
!$omp end parallel
```

Here we use the directive pair `!$omp do`/`!$omp end do` to distribute the do-loop over the different threads. In this case, each thread computes only part of the loop. For example, if 10 threads are in use, then in general each thread computes 100 of the loops: thread 0 computes from 1 to 100, thread 1 from 101 to 200 and so on.

If there is only one do-loop that needs to be distributed over threads in the parallel region, the directive pair `!$omp parallel`/`!$omp end parallel`, which manages the creation of new threads, can be combined with the directive pair `!$omp do`/`!$omp end do`, which manages work-sharing among threads. The combined directive-pair is `!$omp parallel do`/`!$omp end parallel do`. Using this, the above codes can be simplified as

```
!$omp parallel do
         do i = 2, 100
             a(i) = 2*i
         end do
!$omp end parallel do
```

Consider the case where the value of a scalar variable is modified and then used within a do-loop such as:

```
!$omp parallel do
         do i = 2, 100
             b=i**2
             a(i) = 2*i+b
         end do
!$omp end parallel do
```

In this case, different threads share the variable `b`, A thread modify the value of `b` and then use it, but between this, another thread may also modify the value of `b`. Then the first thread would get a value that is not desired. To prevent this from happening, OpenMP provides a way to inform every thread to create a local copy of `b`, i.e., declare `b` as private. The syntax is as follows:

```
!$omp parallel do private (b)
         do i = 2, 100
             b=i**2
             a(i) = 2*i+b
         end do
!$omp end parallel do
```

In this case, every thread has its only local variable `b` and thus prevents other threads from modifying its value.

4 Threads communication

Unlike the explicit message passing between different processes in MPI programming, OpenMP threads communicate implicitly by writing and reading shared variables.

5 Definition of threads

A thread is a basic unit of CPU utilization. In other words, a thread is the smallest sequence of instructions that can be managed independently by the operating system on a single CPU. At a single time-slice, only one thread can be running on a single CPU. Therefore, systems with a single-core processor generally implement multithreading by time slicing: the processor switches between different software threads. This context switching generally happens very often and rapidly enough that users perceive the threads or tasks as running in parallel.

A modern computer or a node of a supercomputer usually has a multi-core processor, which is equipped with more than one independent central processing units (CPUs) and thus can run multiple instructions on separate CPUs at the same time, increasing overall speed for programs amenable to parallel computing. In this context, CPUs are usually called cores.

There is a hardware technology called hyper-threading, which enables one physical CPU to handle two threads more efficiently than using the traditional time-slicing. In this case, one physical CPU can be considered as two logical CPUs.

6 Using OpenMP in practice

OpenMP programming is done to take advantage of a *multi-core* processor. Thus, to get a good speedup, we would typically let our number of threads be equal to the number of cores. However, there is nothing to prevent us from creating more threads: the operating system will use *time slicing* to let them all be executed. We just don't get a speedup beyond the number of actually available cores. On some modern processors there are *hardware threads*, meaning that a core can actually let more than one thread be executed, with some speedup over the single thread. This technology is called hyper-threading. To use such a processor efficiently we would let the number of OpenMP threads be $2\times$ or $4\times$ the number of physical cores, depending on the hardware.

7 Definition of process

A process is a program in execution. A thread is usually a component of a process. Multiple threads can exist within one process, executing concurrently and sharing resources such as memory, while different processes do not share these resources.

Compared with processes, threads are efficient due to their cheap creation and context switching.

8 Difference between OpenMP and MPI

OpenMP deals with threads, which share memory, while MPI deals with processes, which do not share memory. Another difference between OpenMP and MPI is that parallelism in OpenMP is dynamically activated by a thread spawning a team of threads. Furthermore, the number of threads used can differ between parallel regions, and threads can create threads recursively. This is known as as *dynamic mode* . By contrast, in an MPI program the number of running processes is (mostly) constant throughout the run, and determined by factors external to the program.

When developing these notes, I read the following materials: [1][2][3][4].

Bibliography

[1] Victor Eijkhout. Parallel Programming in MPI and OpenMP. http://pages.tacc.utexas.edu/~eijkhout/pcse/html/, 2016. [Online; accessed 24-Feb-2018].

[2] Miguel Hermanns. Parallel Program in Fortran 95 Using OpenMP. http://www.openmp.org/wp-content/uploads/F95_OpenMPv1_v2.pdf, 2002. [Online; accessed 24-Feb-2018].

[3] Https://en.wikipedia.org. Multi-core processor. https://en.wikipedia.org/wiki/Multi-core_processor, 2018. [Online; accessed 24-Feb-2018].

[4] Https://en.wikipedia.org. Thread(computing). https://en.wikipedia.org/wiki/Thread_(computing), 2018. [Online; accessed 24-Feb-2018].