OpenMP programming : 2020/2021

Jan Gmys

1 About this tutorial

This is an explanation. **Read it** before moving on to the ...

⋆ ... exercice

If you have compilation/execution problems :

- If you get any error message(s) : **Read** and try to understand them.
- This tutorial has been carefully tested and hopefully most errors have been removed : please tell me if you think you've found one !

2 Compiling and executing OpenMP programs

OpenMP is a shared memory API providing an interface to write software that can use multiple cores of a computer. This is a well-established standard (v5.0 currently) which is provided by default in nearly all compilers in nearly all operating systems for C, C++ and Fortran. It consists of a set of *compiler directives*, *library routines* and *environment variables* that influence run-time behavior.

⋆ For the complete OpenMP 5.0 Specifications, Examples and Summary Cards visit :
`https://www.openmp.org/specifications/`

Don't panic ! There's no need to read the whole documentation. In this tutorial you will learn about the basic OpenMP functionalities that will allow you to write your first parallel programs for shared-memory multi-core systems. If you need more detailed information and advanced features, this is the place to go.

2.1 First OpenMP program :

⋆ Open a file called `hello_world.f90` and copy in the following code :

```
 PROGRAM hello_world
  IMPLICIT NONE

  !$omp parallel
  PRINT*,'hello'
  !$omp end parallel
END PROGRAM hello_world
```

Listing 1 – hello_world.f90

A block preceded by a `!$omp parallel` *directive* is called a parallel OpenMP region. It is executed by a newly created team of threads. This is an instance of the SPMD model : all threads execute the same segment of code. Note that lines starting with `!` indicate a commentary in Fortran, so they will be ignored if you compile this code without specifying the `-fopenmp` option to the compiler.

- ⋆ Compile this code with the following command :

```
$ gfortran -fopenmp hello_world.f90 -o hello_world
```

  and run the executable `hello_world`.
- ⋆ In the command line type `lscpu` to find out how many cores/threads your computer has.
- ⋆ Execute the `hello_world` program with a varying number of threads. This can be done by setting the `OMP_NUM_THREADS` *environment variable*. For example, this

```
$ OMP_NUM_THREADS=16 ./hello_world
```

  will execute `hello_world` with 16 OpenMP threads, while

```
$ export OMP_NUM_THREADS=5
```

  sets the environment variable to 5 for *all* subsequently executed commands (in that shell).
- ⋆ Compile and run the program again without the `-fopenmp` flag.

2.2 Threads information

We can get some information about the OpenMP threads by using *library routines* defined in the OpenMP standard, mainly the number of threads and an unique number for each thread.

- ⋆ Extend the previous `hello_world.f90` as follows (copy to a new file `hello_threads.f90`)

```
 PROGRAM hello_threads
  USE omp_lib
  IMPLICIT NONE

  INTEGER :: thread_id,threads_num
  PRINT*,'Only one thread here...'

  !$omp parallel private(thread_id,threads_num)
  thread_id = omp_get_thread_num()
  threads_num = omp_get_num_threads()
  PRINT*,'hello! my thread id is',thread_id,' out of ',threads_num
  !$omp end parallel

  PRINT*,'back to one thread'
END PROGRAM hello_threads
```

Listing 2 – hello_threads.f90

Note that the two variables `thread_id,threads_num` are declared with the `private` *data-sharing clause* so each thread has his own memory copy of these variables. By default, all the variables are shared in memory and can be written by all the threads. Note also the declaration `USE omp_lib`, which is necessary for using OpenMP functions.

- ⋆ Compile and execute. Vary the number of threads and observe the output.
- ⋆ How does the output change if you remove the `private` clause ?
- ⋆ Use the `default(none)` clause (instead of `private(...)`) and re-compile. Notice the compiler messages.

3 Parallel loop

Loops are the main way for parallelizing in OpenMP but the following rules need to be followed :

- The `!$omp do` **work-sharing directive** specifies that iterations of the loop (the one following the directive!) are distributed among the existing threads in a parallel region.
  - NB : all iterations should be independent of each other!
- The following restrictions apply to loops associated with a `!$omp do` directive :
  - `DO WHILE` or infinite loops are not parallelizable with OpenMP
  - loops should not contain statements that terminate the loop prematurely (`EXIT`, `GOTO`)

  (however, recent versions of OpenMP allow premature loop cancellation using `!$omp cancel` directives)
- If a parallel region contains only one parallel loop the `!$omp parallel` and `!$omp do` directives can be merged into a single equivalent `!$omp parallel do`.
- By default, the loop iteration variables are made private.
- The parallelized loop is (only!) the one which comes immediately after the `!$omp do` directive

3.1 A simple loop

- ⋆ Open a file called `loops.f90` and copy in the following code :

```
PROGRAM loops
 USE omp_lib
 IMPLICIT NONE

 INTEGER,PARAMETER :: n=1002
 INTEGER :: i,num_loops=0

 !$omp parallel
 !$omp do
 DO i=1,n
    num_loops = num_loops + 1
 ENDDO
 !$omp end do
```

```
  PRINT*,'thread-id ',omp_get_thread_num(),' : number of loops : ',num_loops
  !$omp end parallel
  PRINT*,'---------------------'
  PRINT*,'total number of loops',num_loops

END PROGRAM loops
```

Listing 3 – loops.f90 (WRONG!)

⋆ Compile the program and run the executable several times with 4 threads. Is the final value of `num_loops` coherent ? Before moving on to the next step, you should understand why the value is (in general) different from 1002.

⋆ Add the `reduction(+:num_loops)` clause to the `!$omp parallel` directive in order to obtain the sum of `num_loops` at the end of the parallel region.

3.2 Reduction : π computation

A reduction is an associative operation applied to a shared variable. The operation can be :

- *arithmetic :* $+$, $-$, x
- *logical : .AND., .OR., .EQV., .NEQV.*
- *an intrinsic function : MAX, MIN, IAND,IOR, IEOR*

Each thread computes a partial result independently of the others. They are then synchronized in order to update the final result.

⋆ Create a serial program `pi_serial.f90` that computes the value of π with the following formula. Increasing the number of iterations `nsteps` will improve the precision.

$$\pi/4 = \arctan(1) = \sum_{k=0}^{k=\text{nsteps}} \frac{(-1)^k}{2k+1}$$

(Optional !) You can use the following piece of code to read `nsteps` as an argument from the command line.

```
 CHARACTER(len=32) :: arg

CALL getarg(1, arg)
read(arg, *)nsteps
```

Listing 4 – read arguments

⋆ Copy your serial program in a file named `pi_omp.c` and parallelize the main loop with a reduction.

⋆ Use the function `system_clock` to measure the elapsed time taken to compute π.

```
 INTEGER(kind=8) :: t1, t2, clock_rate
CALL SYSTEM_CLOCK(t1)
 !pi computation here...
CALL SYSTEM_CLOCK(t2, clock_rate)
PRINT*, 'time :', REAL(t2-t1)/REAL(clock_rate)
```

Listing 5 – timing with system_clock

⋆ Increase the number of iterations until the serial execution takes about 10 seconds and compare the time when increasing the number of threads. For example, you can execute your code like this :

```
for i in 1 2 4 6 8;do echo ${i} `OMP_NUM_THREADS=${i} ./pi_omp`;done
```

4 Load balancing : the schedule instruction

- Work in loop iterations can be imbalanced
- The distribution mode of the iterations can be specified with the `schedule` clause :

  !$omp do schedule(type[,chunk])

  where `type` is one of the following and `[,chunk]` an optional positive integer
  - `static[,K]` : divide loop iterations into chunks of size K, statically assigned to threads
  - `dynamic[,K]` : divide loop iter into chunks size K. When a thread finishes one chunk, it is dynamically assigned another. Default is K=1.
  - `guided[,K]` : similar to dynamic, with decreasing chunk size ($\geq$ K)
  - `runtime` : scheduling policy defined by environment variable `OMP_SCHEDULE`
- The choice of the scheduling policy allows a better control of the load-balancing between threads.

4.1 Prime numbers count

The following algorithm computes the number of primes contained in the interval $[2, n]$ (in a very naive way). The `j` loop number of iterations depends on `i` so the amount of work will be different on each `i` loop.

```
PROGRAM main
  USE omp_lib
  IMPLICIT NONE

  INTEGER :: i,j,n=200000,prime=0,total=0
  DOUBLE PRECISION :: tstart
```

```
  !$omp parallel default(none)
  tstart=omp_get_wtime()
  !$omp do reduction(+ :total)
  DO i=2,n
     prime=1
     DO j=2,i-1
        IF(MOD(i,j)==0)THEN
           prime=0
           EXIT
        ENDIF
     ENDDO
     total = total + prime
  ENDDO
  !$omp end do
  PRINT*,'thread ',omp_get_thread_num(),' time :',(omp_get_wtime()-tstart)
  !$omp end parallel

  PRINT*,'nb primes ',total
END PROGRAM main
```

Listing 6 – primes_omp.f90

- ⋆ The program won't compile because of the `default(none)` clause. Add appropriate `shared` and `private` data sharing clauses.
- ⋆ The program uses the `omp_get_wtime()` function to measure the time *each thread* spends in the `i` loop. Does this work as desired ? Add the `nowait` clause to the `omp end do` directive and observe how this changes the (per thread) timing.
- ⋆ Add a `schedule` clause to the `omp do` directive. Run the code with an increasing number of cores and different scheduling policies (try at least `static` and `dynamic` with chunk sizes 1, 10, 1000, 100000, 200000). How to explain the differences ?

5 Parallel performance : Matrix multiplication

> When writing parallel software, you need to check the efficiency of the parallelization. To do this, you need to time your application with an increasing number of processors and check the efficiency.

- ⋆ Write a serial code (`matmul_serial.f90`) that computes the product of two matrices $C = A \times B$. (To keep things simple, you can consider square matrices.)
- ⋆ Copy your serial code into another file (`matmul_omp.f90`) and use OpenMP to parallelize it. Check that the results are identical to the serial code (checking for a 3x3 matrix will be enough).
- ⋆ Write the execution time to a file and plot the speedup (read explanations below) for varying matrix size (for example $n = 100, 500, 1000, 2000$ and 1, 2, 4, 8,... OpenMP threads).

A few explanations/comments :

The speedup S_p is defined by the following formula : $S_p = \frac{T_1}{T_p}$, where

- p is the number of processors
- T_1 is the execution time of the sequential algorithm
- T_p is the execution time of the parallel algorithm with p processors

Linear speedup or ideal speedup is obtained when $S_p = p$. When running an algorithm with linear speedup, doubling the number of processors doubles the speed [1]. As this is ideal, it is considered very good scalability.

OpenMP code use more stack memory than serial code, if you encounter a segmentation fault, try to increase the limit with the two commands `ulimit -s 100000` and `export OMP_STACKSIZE=100m` (here we increase the stack limit to 100Mb, increase to a higher value if necessary)

⋆ Plot also the efficiency :

$$E_p = \frac{S_p}{p} = \frac{T_1}{pT_p}$$

It is a value, typically between zero and one, estimating how well-utilized the processors are in solving the problem

⋆ In the matrix multiplication, you have three nested loops. Try to vary the order of the loop and run the code, with one thread. Is a loop order better than the others ? Why ?

⋆ When you have found the worst and the best order, rerun your test of the parallelized code and compare with the previous results - Does the best loop order also provide the best scalability ?

6 Solution of linear system : Iterative method

The idea of this section is to see that the best algorithm for a serial code is not necessary the best for a parallel code.

6.1 Jacobi

We want to solve a Poisson equation $\Delta u = f$ discretized by finite difference. In order to have an easy-to-compute exact solution u_{ex} we may use $f(x, y) = \sin(\pi x)\cos(\pi y)$. The linear system $Ax = b$ resulting from the discretization will be solved with the Jacobi iterative method :

$$u_{i,j}^{k+1} = \frac{1}{4}(u_{i+1,j}^k + u_{i-1,j}^k + u_{i,j+1}^k + u_{i,j-1}^k - \Delta x \Delta y f_{i,j}) \quad i = 2, nx-1, j = 2, ny-1$$

⋆ You'll be provided a serial version `jacobi.f90` of this algorithm...

⋆ Parallelize this code with OpenMP. Check that you obtain identical results when you use more than one thread (e.g. by plotting the solution...)

⋆ Plot the performance of your parallelization.

1. for fixed problem size : this is called *strong scaling*

6.2 Gauss-Seidel

Gauss-Seidel iterative method is an improvement over the jacobi method (with faster convergence) where we use already computed points in the algorithm (check the $k+1$ on the rhs of the algorithm) :

$$u_{i,j}^{k+1} = \frac{1}{4}(u_{i+1,j}^{k} + u_{i-1,j}^{k+1} + u_{i,j+1}^{k} + u_{i,j-1}^{k+1} - \Delta x \Delta y f_{i,j}) \quad i = 2, nx-1, j = 2, ny-1$$

- ⋆ Create a new function `seidel` and implement this algorithm.
- ⋆ Try to parallelize the Gauss-Seidel algorithm and explain why it is not (directly) possible.

6.3 Red-Black Gauss-Seidel

Fortunately, there are other strategies that converge faster than Jacobi and are more parallelizable than Gauss-Seidel, for example, Red-Black Gauss-Seidel :

$$u_{i,j}^{k+1} = \frac{1}{4}(u_{i+1,j}^{k} + u_{i-1,j}^{k} + u_{i,j+1}^{k} + u_{i,j-1}^{k} - \Delta x \Delta y f_{i,j}) \quad \text{when} \quad i+j \quad \text{is even, followed by}$$

$$u_{i,j}^{k+1} = \frac{1}{4}(u_{i+1,j}^{k+1} + u_{i-1,j}^{k+1} + u_{i,j+1}^{k+1} + u_{i,j-1}^{k+1} - \Delta x \Delta y f_{i,j}) \quad \text{when} \quad i+j \quad \text{is odd}$$

Another advantage of this method is that it requires only a single array u of size $nx \times ny$ instead of 2

- ⋆ Implement and parallelize this algorithm and plot the performance.