Distributed Computing and Introduction to High Performance Computing

Imad Kissami[1]
[1]Mohammed VI Polytechnic University, Benguerir, Morocco

Introduction

Introduction

- OpenMP is a parallel programming model which initially only targeted shared memory architectures. Today, it also targets accelerators, integrated systems and real-time systems.
- The calculation tasks can access a common memory space. This limits data redundancy and simplifies information exchanges between tasks.
- In practice, parallelization is based on the use of light-weight processes (threads). We are speaking, therefore, of a multithreaded program.

Introduction

History

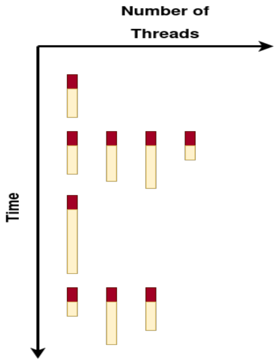
- Multithreaded parallelization has existed for a long time at certain manufacturers (e.g. CRAY, NEC, IBM, $\cdots$) but each one had its own set of directives.
- The resurgence of shared memory multiprocessors made it compelling to define a standard.
- The standardization attempt of the PCF (Parallel Computing Forum) was never adopted by the official standardization authorities.
- On the 28th October 1997, a large majority of industry researchers and manufacturers adopted OpenMP (Open Multi-Processing) as an "industrial standard".
- Today, the OpenMP specifications belong to the ARB (Architecture Review Board), the only organization responsible for its development.

- The **OpenMP 2** version was finalized in November 2000. Most importantly, it provided parallelization extensions to certain Fortran 95 constructions.
- The **OpenMP 3** version of May 2008 primarily introduced the concept of tasks.
- The version **OpenMP 4** of July 2013 followed by version 4.5 of November 2015 brought numerous innovations, notably accelerator support, dependencies between tasks, SIMD (vectorization) programming and management of thread placement.
- The version **OpenMP 5** of November 2018 followed by version 5.1 of November 2020 focused mainly on improving accelerator support. It also brought improvements for task programming, handling of non-uniform memory and support for the latest versions of C (11), C++ (17) and Fortran (2008) languages.

Terminology and definitions

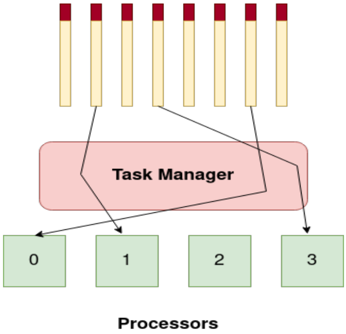
- *Thread*: An execution entity with a local memory (*stack*).
- *Team*: A set of one or several threads which participate in the execution of a parallel region.
- *Task*: An instance of executable code and its associated data. These are generated by the PARALLEL or TASK constructs.
- *Shared variable*: A variable for which the name provides access to the same block of storage shared by the tasks inside a parallel region.
- *Private variable*: A variable for which the name provides access to a different block of storage for each task inside a parallel region.
- *Host device*: Hardware (usually an SMP node) on which OpenMP begins its execution.
- *Target device*: Hardware (accelerator card such as GPU or Xeon Phi) on which a portion of code and the associated data can be transferred and then executed.

- When it begins, an OpenMP program is sequential. It has only one process, the master thread with rank 0, which executes the initial implicit task.
- OpenMP allows defining parallel regions which are code portions destined to be executed in parallel.
- At the entry of a **parallel region**, new threads and new implicit tasks are created. Each thread executes its implicit task concurrently with the others in order to share the work.
- An OpenMP program consists of an alternation between sequential regions and parallel regions.



General concepts: Threads (light-weight processes)

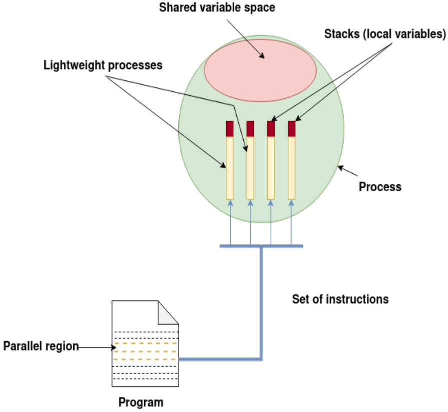
- Each thread executes its own sequence of instructions corresponding to its task.
- The operating system chooses the execution order of the processes (light-weight or not). It assigns them to the available computing units (processor cores).
- There is no guarantee of the overall order in which the parallel program instructions will be executed.



Introduction

General concepts: Threads (light-weight processes)

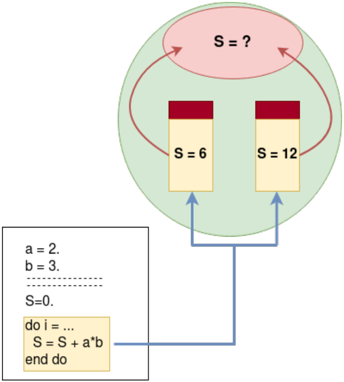
- Tasks of the same program share the memory space of the initial task (shared memory) but also dispose of a local memory space: the stack.
- Therefore, it is possible to define the shared variables (stored in the shared memory) or the private variables (stored in the stack of each one of the tasks).



Introduction

General concepts: Threads (light-weight processes)

- In shared memory, it is sometimes necessary to introduce synchronization between concurrent tasks.
- Synchronization ensures that 2 threads do not modify the value of the same shared variable in a random order (reduction operations).



General concepts: Threads (light-weight processes)

- OpenMP facilitates the writing of parallel algorithms in shared memory by proposing mechanisms to:
  - Share the work between tasks. For example, it is possible to distribute the iterations of a loop between the tasks. Then, when the loop acts on arrays, it can easily distribute the data processing between the threads.
  - Share or privatize the variables.
  - Synchronize the threads.
- Starting with the 3.0 version, OpenMP has also allowed expressing parallelism in the form of a group of explicit tasks to be performed. OpenMP 4.0 allows offloading a part of the work to an accelerator.

Introduction

OpenMP versus MPI

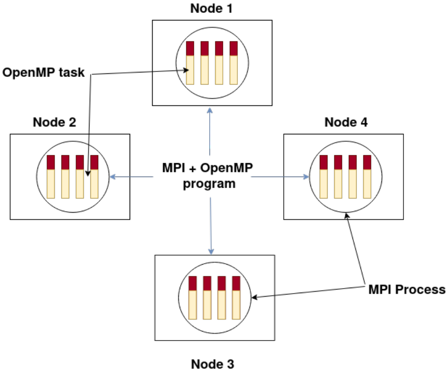
These two programming models are adapted to two different parallel architectures:

- MPI is a distributed memory programming model: Communication between the processes is explicit and the user is responsible for its management.
- OpenMP is a shared memory programming model: Each thread has a global scope of the memory.

Introduction

OpenMP versus MPI

- On a cluster of independent shared memory multiprocessor machines (compute nodes), the implementation of parallelization at two levels (MPI and OpenMP) in the same program can be a major advantage for the parallel performance or the memory footprint of the code.



Principles

programming interface: Format of a directive

- An OpenMP directive has the following general form :

```
sentinelle directive [clause [ clause] ...]
```

- It is a comment line which is ignored by the compiler if the option that allows the interpretation of OpenMP directives is not specified.
- The sentinel is a character string whose value depends on the language used.
- There is an OMP_LIB Fortran 95 module and an ‘omp.h‘ C/C++ include file which define the prototype of all the OpenMP functions. It is mandatory to include them in any OpenMP program unit which uses these functions.

Principles

programming interface: Format of a directive

- For Fortran, in free format

```
!$ use OMP_LIB
...
! $OMP PARALLEL PRIVATE(a,b) &
! $OMP FIRSTPRIVATE(c,d,e)
...
! $OMP END PARALLEL ! This is a comment
```

- For Fortran, in fixed format:

```
  !$ use OMP_LIB
  ...
C$OMP PARALLEL PRIVATE(a,b)
C$OMP1 FIRSTPRIVATE(c,d,e)
    ...
C$OMP END PARALLEL
```

- For C and C++:

```
#ifdef _OPENMP
#include <omp.h>
#endif
...
#pragma omp parallel private(a,b) firstprivate(c,d,e)
{ ... }
```

Principles

Compilation

Compilation options for activating the interpretation of OpenMP directives by some compilers are as follows:

- The GNU compiler: -fopenmp

```
gfortran -fopenmp prog.f90   # Fortran compiler
```

- The Intel compiler: -fopenmp or -qopenmp

```
ifort -fopenmp prog.f90      # Fortran compiler
```

- The PGI/NVIDIA compiler : -mp

```
pgfortran/nvfortran -mp prog.f90     # Fortran compiler
```

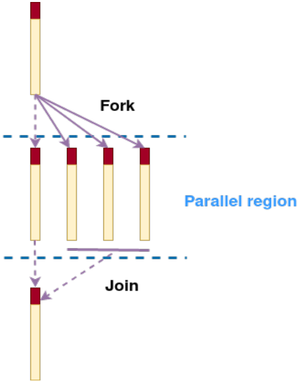
- Execution example:

```
export OMP_NUM_THREADS=3 # Number of desired threads
./a.out          # Execution
```

Principles

Parallel construct

- An OpenMP program is an alternation of sequential and parallel regions ("fork and join" model)
- At the entry of a parallel region, the master thread (rank 0) creates/activates (forks) the "child" processes (light-weight processes or threads) and an equal number of implicit tasks. Each child thread executes its implicit task, then disappears or hibernates at the end of the parallel region (joins).
- There is an implicit synchronization barrier at the end of the parallel region.



Principles

Compilation

- Within the same parallel region, each thread executes a separate implicit task but the tasks are composed of the same duplicated code.
- The data-sharing attribute (DSA) of the variables are shared, by default.
- There is an implicit synchronization barrier at the end of the parallel region.

```fortran
program parallel
  !$ use OMP_LIB

  implicit none
  real:: a
  logical :: p
  a = 92290; p=.false.

  !$OMP PARALLEL
  !$ p = OMP_IN_PARALLEL()
  print *,"A = ",a
  !$OMP END PARALLEL
  print*,"Parallel ?:", p

end program parallel
```

```
ifort -fopenmp example2.f90
export OMP_NUM_THREADS =3
a.out
```

```
A = 92290.0000
A = 92290.0000
A = 92290.0000
Parallel ?: T
```

Principles

Compilation: Python implementation

```python
if __name__ == "__main__":
    from pyccel.stdlib.internal.openmp import omp_in_parallel
    a = 92290
    p = True

#$omp parallel
    p = omp_in_parallel()
    print("A =", a)
#$omp end parallel

print("Parallel ?:", p)
```

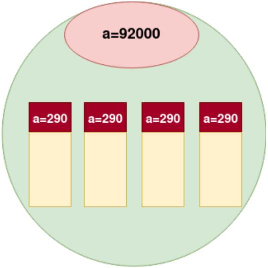
```
pyccel --language=c example2.py --openmp
export OMP_NUM_THREADS=3
./example2
```

```
A = 92290
A = 92290
A = 92290
Parallel ?: True
```

Principles

Data-sharing attribute of variables: Private variables

- The PRIVATE clause allows changing the DSA of a variable to private.
- If a variable has a private DSA, it is allocated in the stack of each task.
- The private variables are not initialized on entry to the parallel region.




```
program parallel
  !$ use OMP_LIB

  implicit none
  real :: a
  logical :: p
  integer :: rank

  a = 92000

  !$OMP PARALLEL private(rank, a)
  !$ rank = OMP_GET_THREAD_NUM()
  a = a + 290
  print *,"Rank : ", rank, &
       "; A = ",a
  !$OMP END PARALLEL
  print*,"Out of region, A = ",a

end program parallel
```

```
Rank : 0 ; A = 290.000000
Rank : 2 ; A = 290.000000
Rank : 1 ; A = 290.000000
Out of region, A = 92000.0000
```

Principles

Private variables: Python implementation

```python
if __name__ == "__main__":
    from pyccel.stdlib.internal.openmp import omp_get_thread_num
    a = 92000

#$ omp parallel private(rank, a)
    rank = omp_get_thread_num()
    a = a + 290
    print("Rank :", rank, "a = ", a)
#$ omp end parallel

print("Out of the region, A will be", a)
```

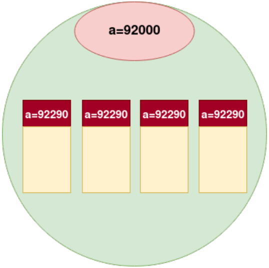
```bash
pyccel --language=c example3.py --openmp
export OMP_NUM_THREADS=3
./example3
```

```
Rank : 0 a = 290
Rank : 2 a = 290
Rank : 1 a = 290
Out of the region, A will be 92000
```

Principles

Data-sharing attribute of variables: Private variables

- With the FIRSTPRIVATE clause, however, it is possible to force the initialization of a private variable to the last value it had before entry to the parallel region.
- After exiting the parallel region, the private variables are lost.




```
program parallel
  !$ use OMP_LIB

  implicit none
  real :: a
  a = 92000.

  !$OMP PARALLEL FIRSTPRIVATE(a)
  a = a + 290
  print *,"A vaut : ",a
  !$OMP END PARALLEL
  print*,"Out of region, A = ",a

end program parallel
```

```
A = 92290.0000
A = 92290.0000
A = 92290.0000
Out of region, A = 92000.0000
```

Principles

Private variables: Python implementation

```python
if __name__ == "__main__":
    from pyccel.stdlib.internal.openmp import omp_in_parallel
    a = 92000

#$omp parallel firstprivate(a)
    a = a + 290
    print("A =", a)
#$omp end parallel

    print("Out of the region, a =", a)
```

```
pyccel --language=c example4.py --openmp
export OMP_NUM_THREADS=3
./example4
```

```
A = 92290
A = 92290
A = 92290
Out of the region, a = 92000
```

Principles

Data-sharing attribute of variables: The DEFAULT clause

- The variables are shared by default but to avoid errors, it is recommended to define the DSA of each variable explicitly.
- Using the DEFAULT(NONE) clause requires the programmer to specify the status of each variable.
- In Fortran, it is also possible to change the implicit DSA of variables by using the DEFAULT(PRIVATE) clause.

```fortran
program parallel
  !$ use OMP_LIB

  implicit none
  logical :: p
  p=.false.

  !$OMP PARALLEL DEFAULT(NONE) &
  !$OMP SHARED(p)
  !$ p = OMP_IN_PARALLEL ()
  !$OMP END PARALLEL

  print*,"Parallel ?:", p

end program parallel
```

```
Parallel ?: T
```

Principles

Private variables: Python implementation

```python
if __name__ == "__main__":
    from pyccel.stdlib.internal.openmp import omp_in_parallel

    p = False
#$omp parallel default(none) shared(p)
    p = omp_in_parallel()
#$omp end parallel

print("Parallel ?:", p)
```

```
pyccel --language=c example5.py --openmp
export OMP_NUM_THREADS=3
./example5
```

```
Parallel ?: True
```

Data-sharing attribute of variables: Dynamic allocation

The dynamic memory allocation/deallocation operation can be done inside the parallel region.

- If the operation concerns a private variable, this local variable will be created/destroyed on each task.
- If the operation concerns a shared variable, it would be more prudent if only one thread (for example, the master thread) does this operation. Because of the data locality, it is recommended to initialize the variables inside the parallel region ("first touch").

Principles

Data-sharing attribute of variables

```fortran
program parallel
!$ use OMP_LIB
implicit none
integer::n,start,end,rank,nb_tasks,i
real, allocatable, dimension(:)::a
n=1024
allocate(a(n))
!$OMP PARALLEL DEFAULT(NONE) PRIVATE(start,end,nb_tasks,rank,i) &
!$OMP SHARED(a,n) IF(n .gt. 512)

  nb_tasks= OMP_GET_NUM_THREADS () ; rank=OMP_GET_THREAD_NUM ()
  start=1+(rank*n)/nb_tasks
  end=((rank+1)*n)/nb_tasks

  do i = start, end
    a(i) = 92290. + real(i)
  end do
  print *,"Rank : ",rank,"; A(",start,"),...,A(",end,") : ",a(start),",...,",a(end)

  !$OMP END PARALLEL
  deallocate(a)
end program parallel
```

```
Rank : 0 ; A( 1 ),...,A( 341 ) : 92291.0000 ,..., 92631.0000
Rank : 1 ; A( 342 ),...,A( 682 ) : 92632.0000 ,..., 92972.0000
Rank : 2 ; A( 683 ),...,A( 1024 ) : 92973.0000 ,..., 93314.0000
```

Principles

Data-sharing attribute of variables: IF is not supported on Pyccel

```python
if __name__ == "__main__":
    from pyccel.stdlib.internal.openmp import omp_get_thread_num, omp_get_num_threads
    import numpy as np
    n = 1024; a = np.empty(n)
    #$omp parallel default(none) private(start, end, nb_tasks, i, rank) shared(a, n)
    nb_tasks = omp_get_num_threads()
    rank=omp_get_thread_num()
    start = int(1 + (rank * n) / nb_tasks)
    end = int(((rank + 1) * n) / nb_tasks)
    for i in range(start, end+1):
        a[i] = 92290 + float(i)
    print("Rank :", rank, ", A[", start, "],..., A[", end ,"]:", a[start], a[end], "...")
    #$omp end parallel
```

```bash
pyccel --language=c example6.py --openmp
export OMP_NUM_THREADS=3
./example6
```

```
Rank : 2 , A[ 683 ],..., A[ 1024 ]: 92973.000000000000 93314.000000000000 ...
Rank : 0 , A[ 1 ],..., A[ 341 ]: 92291.000000000000 92631.000000000000 ...
Rank : 1 , A[ 342 ],..., A[ 682 ]: 92632.000000000000 92972.000000000000 ...
```

Principles

Extent of a parallel region

- The extent of an OpenMP construct represents its scope in the program.
- The influence (or scope) of a parallel region includes the code lexically contained in this region (the static extent) as well as the code of the called routines. The union of these two represents the "dynamic extent".

```
program parallel
  implicit none
  !$OMP PARALLEL
  call sub()
  !$OMP END PARALLEL
end program parallel

subroutine sub()
  !$ use OMP_LIB
  implicit none
  logical :: p
  !$ p = OMP_IN_PARALLEL ()
  !$ print *,"Parallel ?:", p
end subroutine sub
```

```
Parallel ?: T
Parallel ?: T
Parallel ?: T
```

Principles

Extent of a parallel region

```python
from pyccel.stdlib.internal.openmp import omp_in_parallel

def sub():
    p = omp_in_parallel()
    print("Parallel ?:", p)

if __name__ == "__main__":
    #$ omp parallel
    sub()
    #$ omp end parallel
```

```bash
pyccel --language=c example7.py --openmp
export OMP_NUM_THREADS=3
./example7
```

```
Parallel ?: True
Parallel ?: True
Parallel ?: True
```

Principles

Extent of a parallel region

- In a routine called in a parallel region, the local variables and automatic arrays are implicitly private for each task. (They are defined in the stack.)
- In C/C++, the variables declared inside a parallel region are private.

```
program parallel
  implicit none
  !$OMP PARALLEL DEFAULT(SHARED)
  call sub()
  !$OMP END PARALLEL
end program parallel

subroutine sub()
  !$ use OMP_LIB
  implicit none
  integer :: a
  a = 92290
  a = a + OMP_GET_THREAD_NUM ()
  print *,"A = ",a
end subroutine sub
```

```
A = 92290
A = 92292
A = 92291
```

Principles

Extent of a parallel region

```python
from pyccel.stdlib.internal.openmp import omp_get_thread_num

def sub():
    a = 92290
    a += omp_get_thread_num()
    print("A will be :", a)

if __name__ == "__main__":
    #$ omp parallel default(shared)
    sub()
    #$ omp end parallel
```

```bash
pyccel --language=c example8.py --openmp
export OMP_NUM_THREADS=3
./example8
```

```
A will be : 92290
A will be : 92291
A will be : 92292
```

Principles

Transmission by arguments

- In a subroutine or function, all the variables transmitted by argument (*dummy parameters*) inherit the DSA defined in the lexical (static) extent of the region.

```
program parallel
  implicit none
  integer :: a, b
  a = 92000
  !$OMP PARALLEL SHARED(a) PRIVATE(b)
  call sub(a, b)
  print *,"B = ",b
  !$OMP END PARALLEL
end program parallel

subroutine sub(x, y)
  !$ use OMP_LIB
  implicit none
  integer :: x, y
  y = x + OMP_GET_THREAD_NUM ()
end subroutine sub
```

```
B = 92000
B = 92002
B = 92001
```

Principles

Extent of a parallel region

```python
from pyccel.stdlib.internal.openmp import omp_get_thread_num


def sub(x:int, y:int):
    y = x + omp_get_thread_num()
    return y

if __name__ == "__main__":
    b = 0
    a = 92000
    #$ omp parallel shared(a) private(b)
    b = sub(a, b)
    print("B will be:", b)
    #$ omp end parallel
```

```
pyccel --language=c example9.py --openmp
export OMP_NUM_THREADS=3
./example9
```

```
B will be: 92000
B will be: 92001
B will be: 92002
```

Principles

Static variables

- A static variable is accessible during the entire lifespan of a program.
  - In Fortran, this is the case with variables appearing in COMMON, in a MODULE, declared SAVE, or initialized in the declaration (instruction DATA or symbol =).
  - In C/C++, these are variables declared with the keyword *static*.
- In an OpenMP parallel region, a **static** variable is shared by default.

```
module var_stat
  real::c
end module var_stat
```

```
gfortran -fopenmp var_stat.f90 example9.f90
```

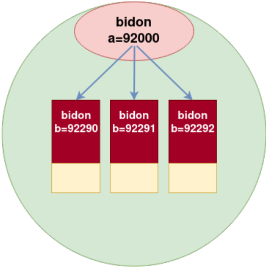
```
program parallel
  use var_stat
  implicit none
  real :: a
  common /bidon/a
  !$OMP PARALLEL
  call sub()
  !$OMP END PARALLEL
end program parallel

subroutine sub()
  use var_stat
  use OMP_LIB
  implicit none
  real::a, b=10.
  integer :: rang
  common /bidon/a
  rang = OMP_GET_THREAD_NUM ()
  a=rang; b=rang; c=rang
  !$OMP BARRIER
  print *,"A, B and C : ",a,b,c
end subroutine sub
```

```
A, B and C : 2.00000000 2.00000000 2.00000000
A, B and C : 2.00000000 2.00000000 2.00000000
A, B and C : 2.00000000 2.00000000 2.00000000
```

Principles

Static variables

- The THREADPRIVATE directive allows privatizing a static instance (for the threads and not the tasks) and makes this persistent from one parallel region to another.
- If the COPYIN clause is specified, the initial value of the static instance is transmitted to all the threads.




```fortran
program parallel
  !$ use OMP_LIB
  implicit none
  integer :: a
  common/bidon/a
  !$OMP THREADPRIVATE(/bidon/)
  a = 92000
  !$OMP PARALLEL COPYIN(/bidon/)
  a = a + OMP_GET_THREAD_NUM ()
  call sub()
  !$OMP END PARALLEL
  print*,"Out of region , A = ",a
end program parallel

subroutine sub()
  implicit none
  integer :: a, b
  common/bidon/a
  !$OMP THREADPRIVATE(/bidon/)
  b = a + 290
  print *,"B = ",b
end subroutine sub
```

```
B = 92290
B = 92292
B = 92291
Out of region, A = 92000
```

Principles

Complementary information

- A parallel region construct accepts two other clauses:
  - REDUCTION : with implicit synchronization between the threads.
  - NUM_THREADS : Allows specifying the number of desired threads at the entry of a parallel region in the same way as the OMP_SET_NUM_THREADS routine would do this.
- The number of concurrent threads can vary, if desired, from one parallel region to another.

```fortran
program parallel
  implicit none
  !$OMP PARALLEL NUM_THREADS(2)
  print *,"Good Morning !"
  !$OMP END PARALLEL
  !$OMP PARALLEL NUM_THREADS(3)
  print *,"Hello !"
  !$OMP END PARALLEL
end program parallel
```

```
Good Morning !
Good Morning !
Hello !
Hello !
Hello !
```

Principles

Complementary information

```python
from pyccel.stdlib.internal.openmp import omp_get_thread_num

if __name__ == "__main__":
    #$ omp parallel num_threads(2)
    print("Hello !")
    #$ omp end parallel
    #$ omp parallel num_threads(3)
    print("Ahoy !")
    #$ omp end parallel
```

```
pyccel --language=c example10.py --openmp
./example10
```

```
Hello !
Hello !
Ahoy !
Ahoy !
Ahoy !
```

Principles

Complementary information

- It is possible to nest parallel regions but this will have no effect if it isn't activated by a call to the OMP_SET_NESTED routine or by setting the OMP_NESTED environment variable at true.

```
gfortran -fopenmp prog.f90
export OMP_NESTED=true
./a.out
```

```fortran
program parallel
  !$ use OMP_LIB
  implicit none
  integer::rank
  !$OMP PARALLEL NUM_THREADS(3) &
  !$OMP PRIVATE(rank)
  rank=OMP_GET_THREAD_NUM ()
  print *,"My rank in region 1 :",rank
  !$OMP PARALLEL NUM_THREADS(2) &
  !$OMP PRIVATE(rank)
  rank=OMP_GET_THREAD_NUM ()
  print *," My rank in region 2 :",rank
  !$OMP END PARALLEL
  !$OMP END PARALLEL
end program parallel
```

```
My rank in region 1 : 0
My rank in region 1 : 2
  My rank in region 2 : 0
  My rank in region 2 : 1
My rank in region 1 : 1
  My rank in region 2 : 0
  My rank in region 2 : 1
  My rank in region 2 : 0
  My rank in region 2 : 1
```

Principles

Complementary information

```python
from pyccel.stdlib.internal.openmp import omp_get_thread_num

if __name__ == "__main__":
    #$ omp parallel num_threads(3) private(rank)
    rank = omp_get_thread_num()
    print("My rank in region 1:", rank)
    #$ omp parallel num_threads(2) private(rank)
    rank = omp_get_thread_num()
    print(" My rank in region 2:", rank)
    #$ omp end parallel
    #$ omp end parallel
```

```bash
pyccel --language=c example11.py --openmp
export OMP_NESTED=true
./example11
```

```
My rank in region 1: 0
My rank in region 1: 1
  My rank in region 2: 0
  My rank in region 2: 1
My rank in region 1: 2
  My rank in region 2: 0
  My rank in region 2: 1
  My rank in region 2: 0
  My rank in region 2: 1
```