Distributed Computing and Introduction to High Performance Computing

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

Worksharing:

Introduction

- The creation of a parallel region and the use of some OpenMP directives/functions should be sufficient to parallelize a part of the code. However, in this case, it is the responsibility of the programmer to distribute the work and to manage the data inside a parallel region.
- Fortunately, there are directives which facilitate this distribution (**DO, WORKSHARE, SECTIONS**)
- Furthermore, it is possible for some portions of code located in a parallel region to be executed by only one thread (**SINGLE, MASTER**).
- Synchronization between threads will be addressed in the following chapter.

Worksharing:

Parallel loop

- A loop is parallel if all of its iterations are independent of each other.
- This is a parallelism by distribution of loop iterations.
- The parallelized loop is the one which immediately follows the **DO** directive.
- The "infinite" and do while loops are not parallelisable with this directive but they can be parallelized via the explicit tasks.
- The distribution mode of the iterations can be specified with the **SCHEDULE** clause (**Won't be done in this course**).
- Being able to choose the distribution mode allows better control of load-balancing between the threads.
- The loop indices are private integer variables by default, so it is not necessary to specify their DSA.
- A global synchronization is done, by default, at the end of an **END DO** construct unless the **NOWAIT** clause was specified.
- It is possible to introduce as many **DO** constructs as desired (one after another) in a parallel region.

Worksharing: parallel loop

Basic example:

```fortran
program parallel

    !$ use OMP_LIB
    implicit none
    integer, parameter :: n=300
    real, dimension(n) :: a
    integer :: i, i_min, i_max, rank, nb_tasks

    !$OMP PARALLEL PRIVATE(rank,nb_tasks,i_min,i_max)
    rank=OMP_GET_THREAD_NUM(); nb_tasks= OMP_GET_NUM_THREADS(); i_min=n ;↩
          i_max=0

    !$OMP DO
    do i = 1, n
        a(i) = 92290. + real(i) ; i_min=min(i_min,i) ; i_max=max(i_max,i)
    end do
    !$OMP END DO

    print *,"Rank : ",rank,"; i_min :",i_min,"; i_max :",i_max
    !$OMP END PARALLEL

end program parallel
```

```
gfortran -fopenmp example13.f90
export OMP_NUM_THREADS=3; ./a.out
Rank : 0 ; i_min : 1 ; i_max : 100
Rank : 1 ; i_min : 101 ; i_max : 200
Rank : 2 ; i_min : 201 ; i_max : 300
```

Worksharing: parallel loop

Basic example using Python:

```python
if __name__ == "__main__":
    import numpy as np
    from pyccel.stdlib.internal.openmp import omp_get_thread_num, ↩
         omp_get_num_threads

    n = 300
    a = np.zeros(n)

    #$omp parallel private(rank, nb_tasks, i_min, i_max)
    rank = omp_get_thread_num(); nb_tasks=omp_get_num_threads(); i_min=n; i_max=0
    #$omp for
    for i in range(n):
        a[i] = 92290. + i
        i_min = min(i_min, i)
        i_max = max(i_min, i)


    print("Rank", rank, "i_min", i_min, "i_max", i_max)
    #$omp end parallel
```

```
pyccel --language=c E15_omp_for.py --openmp
export OMP_NUM_THREADS=3; ./E15_omp_for

Rank 0 i_min 0.0000000000000000 i_max 99.000000000000000
Rank 1 i_min 100.00000000000000 i_max 199.00000000000000
Rank 2 i_min 200.00000000000000 i_max 299.00000000000000
```

Worksharing

A reduction operation

- A reduction is an associative operation applied to a shared variable.
- The operation can be:
  - Arithmetic: $+$, -, $\times$
  - Logical: .AND., .OR., .EQV., .NEQV.
  - An intrinsic function: MAX, MIN, IAND, IOR, IEOR
- Each thread calculates a partial result independently from the others, followed by synchronizing with each other to obtain the final result.

```fortran
program parallel
    implicit none
    integer, parameter::n=5
    integer:: i, s=0, p=1, r=1
    !$OMP PARALLEL
    !$OMP DO REDUCTION(+:s) REDUCTION(*:p,r)
    do i = 1, n
        s = s + 1
        p = p * 2
        r = r * 3
    end do

    !$OMP END PARALLEL

    print *,"s =",s, "; p =",p, "; r =",r
end program parallel
```

```
gfortran -fopenmp example15.f90
export OMP_NUM_THREADS=3; ./a.out
s = 5 ; p = 32 ; r = 243
```

Worksharing

A reduction operation

```python
if __name__ == '__main__':

    n = 5;  s = 0
    p = 1;  r = 1

    #$ omp parallel
    #$ omp for reduction(+:s) reduction(*:p, r)
    for i in range(n):
        s = s + 1
        p = p * 2
        r = r * 3

    #$ omp end parallel

    print("s =", s, ",p =", p, ",r =:", r)
```

```
pyccel --openmp E16_reduction.py
export OMP_NUM_THREADS=3; ./E16_reduction
s = 5 ,p = 32 ,r =: 243
```

Worksharing

Fusion of loop nests

- When loops are perfectly nested and without dependencies, it can be beneficial to fuse them to obtain a unique loop with a larger iteration space.
- In this way, the granularity of each thread's work is increased and this can sometimes significantly improve the performance.
- The **COLLAPSE(n)** clause allows fusing the n nested loops which immediately follow the directive. The new iteration space is then shared by the threads according to the chosen distribution mode.

```fortran
program parallel
    !$ use OMP_LIB

    implicit none
    integer, parameter :: n1=100, n2=1000000
    real, dimension(:,:)::A(n1,n2)
    integer :: i, j, k
    real ::itime, ftime

    itime = omp_get_wtime();
    !$OMP PARALLEL
    !$OMP DO COLLAPSE(2)
    do i=1,n1
        do j=1,n2
            A(i,j)=exp(cos(A(i,j)))
        enddo
    enddo
    !$OMP END DO
    !$OMP END PARALLEL
    ftime = omp_get_wtime();

    print '("Time = ",f6.3," seconds.")',↩
        ftime-itime
end program parallel
```

```
gfortran -fopenmp example16.f90
export OMP_NUM_THREADS=3; ./a.out
Time = 3.500 seconds.
```

Worksharing

Fusion of loop nests

```python
def ft_collapse():
    import numpy as np
    from math import cos, exp

    n1 = 100
    n2 = 1000000
    a = np.empty((n1, n2), float)
    #$ omp parallel
    #$ omp for collapse(2)
    for i in range(n1):
        for j in range(n2):
            a[i,j] = exp(cos(a[i, j]))
    #$ omp end parallel

if __name__ == '__main__':
    from pyccel.epyccel import epyccel

    f1 = epyccel(ft_collapse, accelerators='openmp')
    import timeit

    t1 = timeit.default_timer()
    f1()
    print("Time =", timeit.default_timer() - t1, "second")
```

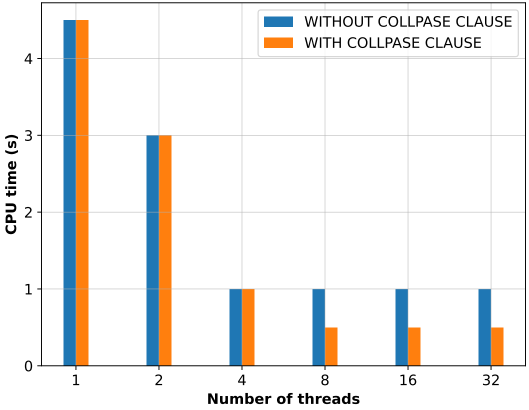
- To run this example, use:

```
export OMP_NUM_THREADS=3
$ python E17_collapse.py
```

Worksharing

Parallel loop

- Execution of the preceding program with and without the COLLAPSE clause.
- Evolution of the execution elapsed time (in s.) function of the number of threads, varying from 1 to 32.



Worksharing

Parallel loop: Additional clauses

- The other clauses accepted in the DO directive:
  - PRIVATE : To declare the private DSA of a variable.
  - FIRSTPRIVATE : To privatise a shared variable throughout the DO construct and assign it the last value it had before entering this region.
  - LASTPRIVATE : To privatise a shared variable throughout the DO construct. This allows conserving, at the exit of this construct, the value calculated by the thread executing the last iteration of the loop.

```fortran
program parallel
    !$ use OMP_LIB
    implicit none
    integer, parameter :: n=9
    integer :: i, rank
    real :: temp
    !$OMP PARALLEL PRIVATE(rank)
    !$OMP DO LASTPRIVATE(temp)

    do i = 1, n
        temp = real(i)
    end do

    !$OMP END DO
    rank=OMP_GET_THREAD_NUM()

    print *,"Rank:",rank,";temp=",temp
    !$OMP END PARALLEL

end program parallel
```

```
gfortran -fopenmp example17.f90
export OMP_NUM_THREADS=3; ./a.out

Rank: 0 ;temp= 9.00000000
Rank: 1 ;temp= 9.00000000
Rank: 2 ;temp= 9.00000000
```

Worksharing

Parallel loop: Additional clauses

```python
if __name__ == '__main__':
    from pyccel.stdlib.internal.openmp import omp_get_num_threads
    n = 9

    #$omp parallel private(rank)
    #$omp parallel for lastprivate(temp)
    for i in range(1, n+1):
        temp = float(i)

    rank=omp_get_num_threads()

    print("Rank:", rank, 'temp=', temp)
    #$omp end parallel
```

```
pyccel --openmp E17_ompfor.py
$ export OMP_NUM_THREADS=3; ./E17_ompfor
Rank: 3 temp= 9.0000000000000000
Rank: 3 temp= 9.0000000000000000
Rank: 3 temp= 9.0000000000000000
```

Worksharing

Parallel loop

- The PARALLEL DO directive is a combination of the PARALLEL and DO directives with the union of their respective clauses.
- The END PARALLEL DO termination directive includes a global synchronization barrier and cannot accept the NOWAIT clause.

```fortran
program parallel

    implicit none
    integer, parameter :: n=9
    integer :: i
    real :: temp

    !$OMP PARALLEL DO LASTPRIVATE(temp)
    do i = 1, n
        temp = real(i)
    end do
    !$OMP END PARALLEL DO

end program parallel
```

```python
if __name__ == '__main__':
    n = 9

    #$ omp parallel for lastprivate(temp)
    for i in range(1, n+1):
        temp = float(i)
```

Worksharing:

Parallel sections

- A section is a portion of code executed by one and only one thread.
- Several code portions can be defined by the user by using the SECTION directive within a SECTIONS construct.
- Synchronization between threads will be addressed in the following chapter.
- The goal is to be able to distribute the execution of several independent code portions to different threads.
- The NOWAIT clause is accepted at the end of the construct (END SECTIONS) to remove the implicit synchronization barrier.

Worksharing

Construction SECTIONS

```fortran
program parallel

 implicit none
 integer, parameter:: n=513, m=4097
 real, dimension(m,n):: a, b
 real, dimension(m):: coord_x, coord_y
 real:: pas_x, pas_y
 integer:: i

 !$OMP PARALLEL
  !$OMP SECTIONS
   !$OMP SECTION
   call lecture_champ_initial_x(a)
   !$OMP SECTION
   call lecture_champ_initial_y(b)
   !$OMP SECTION

   pas_x = 1./real(m-1); pas_y = 2./real(n-1)

   coord_x(:) = (/ (real(i-1)*pas_x,i=1,m) /)
   coord_y(:) = (/ (real(i-1)*pas_y,i=1,n) /)

  !$OMP END SECTIONS NOWAIT
 !$OMP END PARALLEL

end program parallel
```

```fortran
subroutine lecture_champ_initial_x(x)
 implicit none
 integer, parameter :: n=513, m=4097
 real, dimension(m,n) :: x

 call random_number(x)
end subroutine lecture_champ_initial_x

subroutine lecture_champ_initial_y(y)
 implicit none
 integer, parameter :: n=513, m=4097
 real, dimension(m,n) :: y

 call random_number(y)
end subroutine lecture_champ_initial_y
```

Worksharing

Construction SECTIONS

```python
def lecture_champ_initial_x(x:'float[:,:]', m:int, n:int):
    from numpy.random import rand
    x[:,:] = rand((m, n))
def lecture_champ_initial_y(y:'float[:,:]', m:int, n:int):
    from numpy.random import rand
    y[:,:] = rand((m, n))
if __name__ == '__main__':
    import numpy as np
    n = 513; m = 4097
    a = np.empty((m,n)); b = np.empty((m,n))
    coord_x = np.empty(m); coord_y = np.empty(m)
    #$ omp parallel
    #$ omp sections nowait

    #$ omp section
    lecture_champ_initial_x(a, m, n)
    #$ omp end section

    #$ omp section
    lecture_champ_initial_y(b, m, n)
    #$ omp end section

    #$ omp section
    pas_x = 1. / (m - 1.);  pas_y = 2. / (n - 1.)
    for i in range(m):
        coord_x[i] = pas_x * i
    for i in range(n):
        coord_y[i] = pas_y * i
    #$ omp end section
    #$ omp end sections
    #$ omp end parallel
```

Worksharing:

Complementary information

- All the SECTION directives must appear in the lexical extent of the SECTIONS construct.
- The clauses accepted in the SECTIONS construct are those we already know:
  - PRIVATE
  - FIRSTPRIVATE
  - LASTPRIVATE
  - REDUCTION
- The PARALLEL SECTIONS directive is a fusion of the PARALLEL and SECTIONS directives, unifying their respective clauses.
- The END PARALLEL SECTIONS termination directive includes a global synchronization barrier and cannot admit the NOWAIT clause .

Worksharing:

Exclusive execution

- It may occur that we want to exclude all the threads except one to execute certain code portions included in a parallel region.
- To do this, **OpenMP** offers two directives: SINGLE and MASTER .
- Although the desired goal is the same, the behaviour induced by these two constructs is fundamentally different.

Worksharing

The SINGLE construct

- The SINGLE construct allows executing a portion of code by only one thread without being able to indicate which one.
- In general, it's the thread which arrives first on the SINGLE construct but this is not specified in the standard.
- All the threads which are not executing in the SINGLE region wait at the end of the construct END SINGLE until the thread executing has terminated, unless a NOWAIT clause was specified.

```
program parallel
 !$ use OMP_LIB
 implicit none
 integer :: rank
 real :: a

 !$OMP PARALLEL DEFAULT(PRIVATE)
 a = 92290.
 !$OMP SINGLE
 a = -92290.
 !$OMP END SINGLE
 rank=OMP_GET_THREAD_NUM()
 print *,"Rank :",rank,"; A vaut :",a

 !$OMP END P ARALLEL
end program parallel
```

```
$ export OMP_NUM_THREADS=3; ./a.out
Rank : 1 ; A vaut : 92290.0000
Rank : 0 ; A vaut : 92290.0000
Rank : 2 ; A vaut : -92290.0000
```

Worksharing

The SINGLE construct

```python
if __name__ == '__main__':
    from pyccel.stdlib.internal.openmp import omp_get_thread_num

    #$ omp parallel default(private)
    a = 92290
    #$ omp single
    a = -92290
    #$ omp end single

    rank = omp_get_thread_num()

    print("Rank =", rank, "A = ", a)
    #$ omp end parallel
```

```
$ export OMP_NUM_THREADS=3; ./E19_single
Rank = 2 A = 92290
Rank = 1 A = 92290
Rank = 0 A = -92290
```

Worksharing

exclusive execution

- A supplementary clause accepted only by the END SINGLE termination directive is the COPYPRIVATE clause.
- It allows the thread which is charged with executing the SINGLE region, to broadcast the value of a list of private variables to other threads before exiting this region.
- The other clauses accepted by the SINGLE directive are PRIVATE and FIRSTPRIVATE .
- Not yet supported in pyccel

```
program parallel
 !$ use OMP_LIB
 implicit none
 integer :: rank
 real :: a

 !$OMP PARALLEL DEFAULT(PRIVATE)
 a = 92290.
 !$OMP SINGLE
 a = -92290.
 !$OMP END SINGLE COPYPRIVATE(a)
 rank=OMP_GET_THREAD_NUM()
 print *,"Rank :",rank,"; A vaut :",a

 !$OMP END P ARALLEL
end program parallel
```

```
$ export OMP_NUM_THREADS=3; ./a.out
Rank : 1 ; A vaut : -92290.0000
Rank : 0 ; A vaut : -92290.0000
Rank : 2 ; A vaut : -92290.0000
```

Worksharing

The MASTER construct

- The MASTER construct allows the execution of a portion of code by the master thread only.
- This construct does not accept any clauses.
- No synchronization barrier exists, neither at the beginning (MASTER) nor at the termination (END MASTER).

```
program parallel
 !$ use OMP_LIB
 implicit none
 integer :: rank
 real :: a

 !$OMP PARALLEL DEFAULT(PRIVATE)
 a = 92290.
 !$OMP MASTER
 a = -92290.
 !$OMP END MASTER
 rank=OMP_GET_THREAD_NUM()
 print *,"Rank :",rank,"; A vaut :",a

 !$OMP END P ARALLEL
end program parallel
```

```
$ export OMP_NUM_THREADS=3; ./a.out
Rank : 1 ; A vaut : 92290.0000
Rank : 0 ; A vaut : -92290.0000
Rank : 2 ; A vaut : 92290.0000
```

Worksharing

The MASTER construct

```python
if __name__ == '__main__':
    from pyccel.stdlib.internal.openmp import omp_get_thread_num

    #$ omp parallel default(private)
    a = 92290
    #$ omp master
    a = -92290
    #$ omp end master

    rank = omp_get_thread_num()

    print("Rank =", rank, "A = ", a)
    #$ omp end parallel
```

```
$ export OMP_NUM_THREADS=3; ./E21_master
Rank = 0 A = −92290
Rank = 1 A = 92290
Rank = 2 A = 92290
```

Synchronization

Introduction

Synchronization becomes necessary in the following situations:

1. To ensure that all the concurrent threads have reached the same instruction point in the program (global barrier).
2. To order the execution of all the concurrent threads when they need to execute the same code portion affecting one or more shared variables whose memory coherence (in read or write) must be guaranteed (mutual exclusion).
3. To synchronize at least two concurrent threads among all the others (lock mechanism)

Synchronization

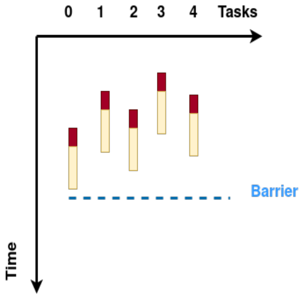
Introduction

- As we have already indicated, the absence of a NOWAIT clause means that a global Synchronization barrier is implicitly applied at the end of the **OpenMP** construct. However, it is possible to explicitly impose a global Synchronization barrier by using the BARRIER directive
- The mutual exclusion mechanism (one task at a time) is found, for example, in the reduction operations (REDUCTION clause) or in the ordered execution of a loop (DO ORDERED directive). This mechanism is also implemented in the ATOMIC and CRITICAL directives.
- Finer synchronizations can be done either by the implementation of lock mechanisms (requiring a call to **OpenMP** library subroutines) or by using the FLUSH directive (won't be done in this course).

Synchronization

Barrier

- The BARRIER directive synchronizes all the concurrent threads within a parallel region.
- Each thread waits until all the other threads reach this synchronization point in order to continue, together, the execution of the program.
- No synchronization barrier exists, neither at the beginning (MASTER) nor at the termination (END MASTER).



```fortran
program parallel
 implicit none
 real,allocatable,dimension(:) :: a,b
 integer:: n, i
 n = 5
 !$OMP PARALLEL
 !$OMP SINGLE

 allocate(a(n),b(n))
 !$OMP END SINGLE
 !$OMP MASTER
 read(9) a(1:n)
 !$OMP END MASTER
 !$OMP BARRIER
 !$OMP DO
 do i = 1, n
  b(i) = 2.*a(i)
 end do
 !$OMP SINGLE
 deallocate(a)
 !$OMP END SINGLE NOWAIT
!$OMP END PARALLEL
 print *, "B = ", b(1:n)
end program parallel
```

Synchronization

Barrier

```
if __name__ == '__main__':
    import numpy as np

    n = 5
    #$ omp parallel
    #$ omp single
    a = np.empty(n); b = np.empty(n)
    #$ omp end single

    #$ omp master
    for i in range(n):
        a[i] = i + 1
    #$ omp end master

    #$ omp barrier

    #$ omp for
    for i in range(n):
        b[i] = 2 * a[i]

    #$ omp single nowait
    del(a)
    #$ omp end single

    #$ omp end parallel
    print(``B ='', b[:])
```

```
$ export OMP_NUM_THREADS=3; ./E22_barrier
B = [2.000000000000 4.000000000000 6.000000000000 8.000000000000 10.000000000000]
```

Synchronization

Atomic update

- The ATOMIC directive ensures that a shared variable is read and modified in memory by only one thread at a time.
- Its effect is limited to the instruction immediately following the directive.

```
program parallel
 !$ use OMP_LIB
 implicit none
 integer :: cmpt, rank
 cmpt = 92290
 !$OMP PARALLEL PRIVATE(rank)
 rank=OMP_GET_THREAD_NUM()
 !$OMP ATOMIC
 cmpt = cmpt + 1
 print *,"Rank :",rank, "; cmpt = ",↩
      cmpt
 !$OMP END PARALLEL
 print *,"In total, cmpt = ", cmpt
end program parallel
```

```
export OMP_NUM_THREADS=3; ./a.out
Rank : 1 ; cmpt = 92293
Rank : 0 ; cmpt = 92293
Rank : 2 ; cmpt = 92293
In total, cmpt = 92293
```

Synchronization

Atomic update

```python
if __name__ == '__main__':
    from pyccel.stdlib.internal.openmp import omp_get_thread_num

    cmpt = 92290
    #$ omp parallel private(rank)
    rank = omp_get_thread_num()
    #$ omp atomic
    cmpt = cmpt + 1
    print("Rank =", rank, "cmpt =", cmpt)
    #$ omp end parallel

print("in total, cmpt =", cmpt)
```

```
$ export OMP_NUM_THREADS=3; ./E23_atomic
Rank = 0 cmpt = 92293
Rank = 2 cmpt = 92293
Rank = 1 cmpt = 92293
in total, cmpt = 92293
```

Synchronization

Atomic update

- The instruction in question must have one of the following forms:
  - $x = x(op)exp$
  - $x = exp(op)x$
  - $x = f(x, exp)$
  - $x = f(exp, x)$
- (op) represents one of the following operations: $+, -, , /, .AND., .OR., .EQV., .NEQV..$
- f represents one of the following intrinsic functions: $MAX, MIN, IAND, IOR, IEOR$.
- exp is any arithmetic expression independent of x.

Synchronization

Critical regions

- A critical region can be seen as a generalization of the ATOMIC directive although the underlying mechanisms are distinct.
- All the threads execute this region in a non-deterministic order but only one at a time.
- A critical region is delimited by the CRITICAL / END CRITICAL directives.
- Its extent is dynamic.
- For performance reasons, it is not recommended to emulate an atomic instruction by a critical region.
- An optional name can be given to a critical region.
- All critical regions which are not explicitly named are considered as having the same non-specified name.
- If several critical regions have the same name, the mutual exclusion mechanism considers them as being one and the same critical region.

Synchronization

Critical regions

```fortran
program parallel
 implicit none
 integer :: s, p
 s=0
 p=1
 !$OMP PARALLEL
 !$OMP CRITICAL
 s = s + 1
 !$OMP END CRITICAL
 !$OMP CRITICAL (RC1)
 p = p * 2
 !$OMP END CRITICAL (RC1)
 !$OMP CRITICAL
 s = s + 1
 !$OMP END CRITICAL
 !$OMP END PARALLEL
 print *, "s= ",s, " ; p= ",p
end program parallel
```

```
export OMP_NUM_THREADS=3; ./a.out
s= 6 ; p= 8
```

Synchronization

Critical regions

```python

if __name__ == '__main__':
    from pyccel.stdlib.internal.openmp import omp_get_thread_num
    s = 0
    p = 1
    #$ omp parallel
    #$ omp critical
    s = s + 1
    #$ omp end critical

    #$ omp critical
    p = p * 2
    #$ omp end critical

    #$ omp critical
    s = s + 1
    #$ omp end critical

    #$ omp end parallel
    print("s =", s, "p = ", p)
```

```
$ export OMP_NUM_THREADS=3; ./E24_critical
s = 6 p = 8
```

Worksharing

Summary

- The table of supported clauses for each workshare

	default	shared	private	firstprivate	lastprivate	copyprivate	if	reduction	schedule	ordered	copyin	nowait
Parallel	✓	✓	✓	✓			✓				✓	
do			✓	✓	✓			✓	✓	✓		✓
sections			✓	✓	✓			✓				✓
single			✓	✓		✓						✓
master												
critical												