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:

```
1  program parallel
2
3      !$ use OMP_LIB
4      implicit none
5      integer, parameter :: n=300
6      real, dimension(n) :: a
7      integer:: i, i_min, i_max, rank, nb_tasks
8
9      !$OMP PARALLEL PRIVATE(rank,nb_tasks,i_min,i_max)
10     rank=OMP_GET_THREAD_NUM(); nb_tasks= OMP_GET_NUM_THREADS(); i_min=n ;↩
           i_max=0
11
12     !$OMP DO
13     do i = 1, n
14         a(i) = 92290. + real(i) ; i_min=min(i_min,i) ; i_max=max(i_max,i)
15     end do
16     !$OMP END DO
17
18     print *,"Rank : ",rank,"; i_min :",i_min,"; i_max :",i_max
19     !$OMP END PARALLEL
20
21 end program parallel
```

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

Worksharing: parallel loop

Basic example using Python:

```
1  if __name__ == "__main__":
2      import numpy as np
3      from pyccel.stdlib.internal.openmp import omp_get_thread_num, ↩
           omp_get_num_threads
4
5      n = 300
6      a = np.zeros(n)
7
8      #$omp parallel private(rank, nb_tasks, i_min, i_max)
9      rank = omp_get_thread_num(); nb_tasks=omp_get_num_threads(); i_min=n; i_max=0
10     #$omp for
11     for i in range(n):
12         a[i] = 92290. + i
13         i_min = min(i_min, i)
14         i_max = max(i_min, i)
15
16
17     print("Rank", rank, "i_min", i_min, "i_max", i_max)
18     #$omp end parallel
```

```
1 pyccel --language=c E15_omp_for.py --openmp
2 export OMP_NUM_THREADS=3; ./E15_omp_for
3
4 Rank 0 i_min 0.0000000000000000 i_max 99.000000000000000
5 Rank 1 i_min 100.00000000000000 i_max 199.00000000000000
6 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: +, -, ×
  - 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.

```
1  program parallel
2      implicit none
3      integer, parameter::n=5
4      integer:: i, s=0, p=1, r=1
5      !$OMP PARALLEL
6      !$OMP DO REDUCTION(+:s) REDUCTION(*:p,r)
7      do i = 1, n
8          s = s + 1
9          p = p * 2
10         r = r * 3
11     end do
12
13     !$OMP END PARALLEL
14
15     print *,"s =",s, "; p =",p, "; r =",r
16 end program parallel
```

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

Worksharing

A reduction operation

```
1  if __name__ == '__main__':
2
3      n = 5;  s = 0
4      p = 1;  r = 1
5
6      #$ omp parallel
7      #$ omp for reduction(+:s) reduction(*:p, r)
8      for i in range(n):
9          s = s + 1
10         p = p * 2
11         r = r * 3
12
13     #$ omp end parallel
14
15     print("s =", s, ",p =", p, ",r =:", r)
```

```
1 pyccel --openmp E16_reduction.py
2 export OMP_NUM_THREADS=3; ./E16_reduction
3 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.

```
1  program parallel
2      !$ use OMP_LIB
3
4      implicit none
5      integer, parameter :: n1=100, n2=1000000
6      real, dimension(:,:)::A(n1,n2)
7      integer :: i, j, k
8      real ::itime, ftime
9
10     itime = omp_get_wtime();
11     !$OMP PARALLEL
12     !$OMP DO COLLAPSE(2)
13     do i=1,n1
14         do j=1,n2
15             A(i,j)=exp(cos(A(i,j)))
16         enddo
17     enddo
18     !$OMP END DO
19     !$OMP END PARALLEL
20     ftime = omp_get_wtime();
21
22     print '("Time = ",f6.3," seconds.")',↩
           ftime-itime
23 end program parallel
```

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

Worksharing

Fusion of loop nests

```
1  def ft_collapse():
2      import numpy as np
3      from math import cos, exp
4
5      n1 = 100
6      n2 = 1000000
7      a = np.empty((n1, n2), float)
8      #$ omp parallel
9      #$ omp for collapse(2)
10     for i in range(n1):
11         for j in range(n2):
12             a[i,j] = exp(cos(a[i, j]))
13     #$ omp end parallel
14
15 if __name__ == '__main__':
16     from pyccel.epyccel import epyccel
17
18     f1 = epyccel(ft_collapse, accelerators='openmp')
19     import timeit
20
21     t1 = timeit.default_timer()
22     f1()
23     print("Time =", timeit.default_timer() - t1, "second")
```

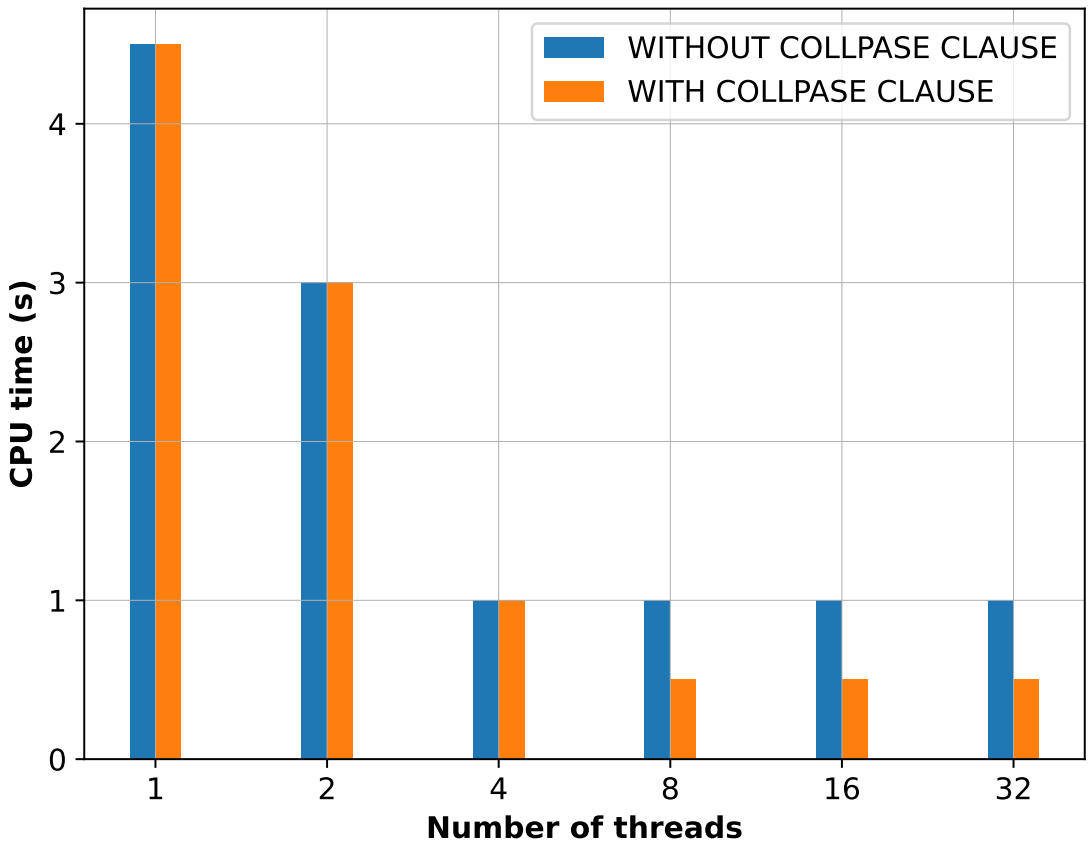
- To run this example, use:

```
1 export OMP_NUM_THREADS=3
2 $ 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.

```
1  program parallel
2      !$ use OMP_LIB
3      implicit none
4      integer, parameter :: n=9
5      integer :: i, rank
6      real :: temp
7      !$OMP PARALLEL PRIVATE(rank)
8      !$OMP DO LASTPRIVATE(temp)
9
10     do i = 1, n
11         temp = real(i)
12     end do
13
14     !$OMP END DO
15     rank=OMP_GET_THREAD_NUM()
16
17     print *,"Rank:",rank,";temp=",temp
18     !$OMP END PARALLEL
19
20 end program parallel
```

```
1 gfortran -fopenmp example17.f90
2 export OMP_NUM_THREADS=3; ./a.out
3
4 Rank: 0 ;temp= 9.00000000
5 Rank: 1 ;temp= 9.00000000
6 Rank: 2 ;temp= 9.00000000
```

Worksharing

Parallel loop: Additional clauses

```
1  if __name__ == '__main__':
2      from pyccel.stdlib.internal.openmp import omp_get_num_threads
3      n = 9
4
5      #$omp parallel private(rank)
6      #$omp parallel for lastprivate(temp)
7      for i in range(1, n+1):
8          temp = float(i)
9
10     rank=omp_get_num_threads()
11
12     print("Rank:", rank, 'temp=', temp)
13     #$omp end parallel
```

```
1 pyccel --openmp E17_ompfor.py
2 $ export OMP_NUM_THREADS=3; ./E17_ompfor
3 Rank: 3 temp= 9.0000000000000000
4 Rank: 3 temp= 9.0000000000000000
5 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.

```
1  program parallel
2
3      implicit none
4      integer, parameter :: n=9
5      integer :: i
6      real :: temp
7
8      ! $OMP PARALLEL DO LASTPRIVATE(temp)
9      do i = 1, n
10         temp = real(i)
11     end do
12     ! $OMP END PARALLEL DO
13
14 end program parallel
```

```
1  if __name__ == '__main__':
2      n = 9
3
4      #$ omp parallel for lastprivate(temp)
5      for i in range(1, n+1):
6          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

```
1  program parallel
2
3   implicit none
4   integer , parameter:: n=513, m=4097
5   real , dimension(m,n):: a, b
6   real , dimension(m):: coord_x, coord_y
7   real:: pas_x, pas_y
8   integer:: i
9
10  !$OMP PARALLEL
11   !$OMP SECTIONS
12    !$OMP SECTION
13    call lecture_champ_initial_x(a)
14    !$OMP SECTION
15    call lecture_champ_initial_y(b)
16    !$OMP SECTION
17
18    pas_x = 1./real(m-1); pas_y = 2./real(n-1)
19
20    coord_x(:) = (/ (real(i-1)*pas_x,i=1,m) /)
21    coord_y(:) = (/ (real(i-1)*pas_y,i=1,n) /)
22
23   !$OMP END SECTIONS NOWAIT
24  !$OMP END PARALLEL
25
26 end program parallel
```

```
1  subroutine lecture_champ_initial_x(x)
2   implicit none
3   integer , parameter :: n=513, m=4097
4   real , dimension(m,n) :: x
5
6   call random_number(x)
7  end subroutine lecture_champ_initial_x
8
9  subroutine lecture_champ_initial_y(y)
10  implicit none
11  integer , parameter :: n=513, m=4097
12  real , dimension(m,n) :: y
13
14  call random_number(y)
15 end subroutine lecture_champ_initial_y
```

Worksharing

Construction SECTIONS

```
1  def lecture_champ_initial_x(x:'float[:,:]', m:int, n:int):
2      from numpy.random import rand
3      x[:,:] = rand((m, n))
4  def lecture_champ_initial_y(y:'float[:,:]', m:int, n:int):
5      from numpy.random import rand
6      y[:,:] = rand((m, n))
7  if __name__ == '__main__':
8      import numpy as np
9      n = 513; m = 4097
10     a = np.empty((m,n)); b = np.empty((m,n))
11     coord_x = np.empty(m); coord_y = np.empty(m)
12     #$ omp parallel
13     #$ omp sections nowait
14
15     #$ omp section
16     lecture_champ_initial_x(a, m, n)
17     #$ omp end section
18
19     #$ omp section
20     lecture_champ_initial_y(b, m, n)
21     #$ omp end section
22
23     #$ omp section
24     pas_x = 1. / (m - 1.);  pas_y = 2. / (n - 1.)
25     for i in range(m):
26         coord_x[i] = pas_x * i
27     for i in range(n):
28         coord_y[i] = pas_y * i
29     #$ omp end section
30     #$ omp end sections
31     #$ 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.

```
1  program parallel
2   !$ use OMP_LIB
3   implicit none
4   integer :: rank
5   real :: a
6
7   !$OMP PARALLEL DEFAULT(PRIVATE)
8   a = 92290.
9   !$OMP SINGLE
10  a = -92290.
11  !$OMP END SINGLE
12  rank=OMP_GET_THREAD_NUM()
13  print *,"Rank :",rank,"; A vaut :",a
14
15  !$OMP END P ARALLEL
16 end program parallel
```

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

Worksharing

The SINGLE construct

```
1  if __name__ == '__main__':
2      from pyccel.stdlib.internal.openmp import omp_get_thread_num
3
4      #$ omp parallel default(private)
5      a = 92290
6      #$ omp single
7      a = -92290
8      #$ omp end single
9
10     rank = omp_get_thread_num()
11
12     print("Rank =", rank, "A = ", a)
13     #$ omp end parallel
```

```
1 $ export OMP_NUM_THREADS=3; ./E19_single
2 Rank = 2 A = 92290
3 Rank = 1 A = 92290
4 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

```
1  program parallel
2   !$ use OMP_LIB
3   implicit none
4   integer :: rank
5   real :: a
6
7   !$OMP PARALLEL DEFAULT(PRIVATE)
8   a = 92290.
9   !$OMP SINGLE
10  a = -92290.
11  !$OMP END SINGLE COPYPRIVATE(a)
12  rank=OMP_GET_THREAD_NUM()
13  print *,"Rank :",rank,"; A vaut :",a
14
15  !$OMP END P ARALLEL
16 end program parallel
```

```
1 $ export OMP_NUM_THREADS=3; ./a.out
2 Rank : 1 ; A vaut : -92290.0000
3 Rank : 0 ; A vaut : -92290.0000
4 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).

```
1  program parallel
2   !$ use OMP_LIB
3   implicit none
4   integer :: rank
5   real :: a
6
7   !$OMP PARALLEL DEFAULT(PRIVATE)
8   a = 92290.
9   !$OMP MASTER
10  a = -92290.
11  !$OMP END MASTER
12  rank=OMP_GET_THREAD_NUM()
13  print *,"Rank :",rank,"; A vaut :",a
14
15  !$OMP END P ARALLEL
16 end program parallel
```

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

Worksharing

The MASTER construct

```
1  if __name__ == '__main__':
2      from pyccel.stdlib.internal.openmp import omp_get_thread_num
3
4      #$ omp parallel default(private)
5      a = 92290
6      #$ omp master
7      a = -92290
8      #$ omp end master
9
10     rank = omp_get_thread_num()
11
12     print("Rank =", rank, "A = ", a)
13     #$ omp end parallel
```

```
1  $ export OMP_NUM_THREADS=3; ./E21_master
2  Rank = 0 A = -92290
3  Rank = 1 A = 92290
4  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

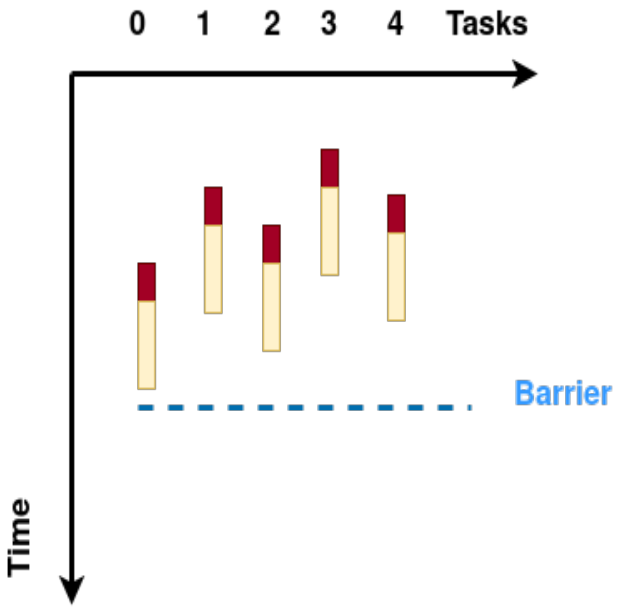
Introduction

- 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).




```
1  program parallel
2   implicit none
3   real,allocatable,dimension(:) :: a,b
4   integer:: n, i
5   n = 5
6   !$OMP PARALLEL
7   !$OMP SINGLE
8
9   allocate(a(n),b(n))
10  !$OMP END SINGLE
11  !$OMP MASTER
12  read(9) a(1:n)
13  !$OMP END MASTER
14  !$OMP BARRIER
15  !$OMP DO
16  do i = 1, n
17    b(i) = 2.*a(i)
18  end do
19  !$OMP SINGLE
20  deallocate(a)
21  !$OMP END SINGLE NOWAIT
22 !$OMP END PARALLEL
23  print *, "B = ", b(1:n)
24 end program parallel
```

Synchronization

Barrier

```
1  if __name__ == '__main__':
2      import numpy as np
3
4      n = 5
5      #$ omp parallel
6      #$ omp single
7      a = np.empty(n); b = np.empty(n)
8      #$ omp end single
9
10     #$ omp master
11     for i in range(n):
12         a[i] = i + 1
13     #$ omp end master
14
15     #$ omp barrier
16
17     #$ omp for
18     for i in range(n):
19         b[i] = 2 * a[i]
20
21     #$ omp single nowait
22     del(a)
23     #$ omp end single
24
25     #$ omp end parallel
26     print(``B ='', b[:])
```

```
1 $ export OMP_NUM_THREADS=3; ./E22_barrier
2 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.

```
1  program parallel
2   !$ use OMP_LIB
3   implicit none
4   integer :: cmpt, rank
5   cmpt = 92290
6   !$OMP PARALLEL PRIVATE(rank)
7   rank=OMP_GET_THREAD_NUM()
8   !$OMP ATOMIC
9   cmpt = cmpt + 1
10  print *,"Rank :",rank, "; cmpt = ",↩
        cmpt
11  !$OMP END PARALLEL
12  print *,"In total, cmpt = ", cmpt
13 end program parallel
```

```
1 export OMP_NUM_THREADS=3; ./a.out
2 Rank : 1 ; cmpt = 92293
3 Rank : 0 ; cmpt = 92293
4 Rank : 2 ; cmpt = 92293
5 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

```
1  program parallel
2   implicit none
3   integer :: s, p
4   s=0
5   p=1
6   !$OMP PARALLEL
7   !$OMP CRITICAL
8   s = s + 1
9   !$OMP END CRITICAL
10  !$OMP CRITICAL
11  p = p * 2
12  !$OMP END CRITICAL
13  !$OMP CRITICAL
14  s = s + 1
15  !$OMP END CRITICAL
16  !$OMP END PARALLEL
17  print *, "s= ",s, "  ; p= ",p
18 end program parallel
```

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

Synchronization

Critical regions

```
1
2  if __name__ == '__main__':
3      from pyccel.stdlib.internal.openmp import omp_get_thread_num
4      s = 0
5      p = 1
6      #$ omp parallel
7      #$ omp critical
8      s = s + 1
9      #$ omp end critical
10
11     #$ omp critical
12     p = p * 2
13     #$ omp end critical
14
15     #$ omp critical
16     s = s + 1
17     #$ omp end critical
18
19     #$ omp end parallel
20     print("s =", s, "p = ", p)
```

```
1 $ export OMP_NUM_THREADS=3; ./E24_critical
2 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												