Advanced parallel Fortran

A. Shterenlikht

Mech Eng Dept, The University of Bristol
Bristol BS8 1TR, UK, Email: mexas@bristol.ac.uk

coarrays.sf.net

12th November 2017
```fortran
integer :: i1, i2(10,20,-3:8)
real, allocatable :: r(:,:)
complex(kind=8) :: z
character(:), allocatable :: c
type t
    logical, allocatable :: l(:,,:,:)
end type t
type(t) :: tvar
    c = "kuku"
    allocate( tvar%l(3,3,3), source=.true. )
    write (*,*), len(c), size(tvar%l)
end

$ ifort z.f90
$ ./a.out
  4     27
```
```
integer :: i1 [*], i2(10,20,−3:8) [2,2,*]
real, allocatable :: r(:,,:) [:]
complex(kind=8) :: z [-5:*]
character(:,), allocatable :: c [:]
type t
    logical, allocatable :: l(:,,:) (:)
end type t

type(t) :: tvar [*]
c = "kuku"
allocate( tvar%l(3,3,3), source=.true. )
write (*,*) len(c), size(tvar%l)
```

- Vars with [ ] are coarray variables.
- Most variables can be made into coarrays. Among exceptions are variables of types C_PTR, C_FUNPTR and TEAM_TYPE (Fortran 2015) from the intrinsic module ISO_C_BINDING.
Fortran 2008 runtime

$ ifort -coarray z.f90
$ setenv FOR_COARRAY_NUM_IMAGES 7
$ ./a.out

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>27</td>
<td></td>
</tr>
</tbody>
</table>

- Concurrent asynchronous execution of multiple identical copies of the executable (images).
- Number of images can be set at compile or run time.
- Different options in Cray, Intel, GCC/OpenCoarrays compilers.
Course outline

- coarray syntax and usage, remote operations
- images, execution segments, execution control, synchronisation
- DO CONCURRENT construct
- allocatable coarrays
- termination
- dealing with failures
- collectives, atomics, critical sections, locks
- (briefly) upcoming Fortran 2018 standard - teams, events, further facilities for dealing with image failures.
Fortran coarrays

- Native Fortran means for SPMD (single program multiple data) parallel programming
- Over 20 years of experience, mainly on Cray
- Fortran standard since Fortran 2008 [1], Many more features added in Fortran 2015 [2, 3]
- Supported on Cray, Intel, GCC/OpenCoarrays
iso_fortran_env is the intrinsic module, introduced in Fortran 2003, and expanded in Fortran 2008. Named constants: input_unit, output_unit, error_unit.

All I/O units, except input_unit, are private to an image.

The runtime environment typically merges output_unit and error_unit streams from all images into a single stream.

input_unit is preconnected only on image 1.
Coarray syntax

```
integer :: i[*] ! scalar integer coarray
            ! with a single codimension
integer, codimension(*) :: i ! equiv. to above
integer i
codimension :: i[*] ! equiv. to above

! complex array coarray of corank 3
! lower    upper
! cobound  cobound
! upper    |    |    |
! bound    |    |    |
! lower    |    |    |
! bound    |    |    |
! bound    |    |    |
!
complex :: c(7,0:13) [-3:2,5,]*
! subscripts    cosubscripts
```
Coarray basic rules

- Any image has read/write access to all coarray variables on all images.
- It makes no sense to declare coarray parameters.
- The last upper cobound is always an *, meaning that it is only determined at run time.
- corank is the number of cosubscripts.
- Each cosubscript runs from its lower cobound to its upper cobound.
- New intrinsics are introduced to return these values: lcobound and ucobound

Remember: lcobound, ucobound.
Cosubscripts 1

- An image can be identified by its image number, from 1 to `num_images`, or by its `cosubscripts set`.
- `this_image` with no argument returns image number of the invoking image.
- `this_image` with a coarray as an argument returns the set of cosubscripts corresponding to the coarray on the invoking image.
- `image_index` is the inverse of `this_image`. Given a valid set of cosubscripts as input, `image_index` returns the index of the invoking image.
- There can be cosubscript sets which do not map to a valid image index. For such `invalid` cosubscript sets `image_index` returns 0.

Remember: `num_images`, `this_image`, `image_index`.
$ cat z.f90

character(len=10) :: i[-2:2,1:*], p[*], z[2,*]
if (this_image().eq.num_images()) then
  write (*,*), this_image()
  write (*,*), this_image(i), this_image(p), &
  this_image(z)
  write (*,*), lcobound(i), lcobound(p), &
  lcobound(z)
  write (*,*), ucobound(i), ucobound(p), &
  ucobound(z)
  write (*,*), image_index(i, ucobound(i))
  write (*,*), image_index(p, ucobound(p))
  write (*,*), image_index(z, ucobound(z))
end if
end
Cosubscripts - logical arrangement of images

$ ifort -coarray -coarray -num-images=10 z.f90
$ ./a.out

```
10
2  2  10  2  5
-2  1  1  1  1
  2 10  2  5
10
10
```

```
i
-2
-1
0
1
2
1
2
3
4
5
z
```

i, p and z are all scalar coarrays, but with different logical arrangement across images
Cosubscripts - logical arrangement of images

```
$ ifort -coarray -warn -coarray -num-images=17 z.f90
$ ./a.out

17
−1
−2
2
0
17
0

4  17  1   9
1  1   1   1
4  17  2   9
```

```
1  2  3  4
−2
−1
0
1
2

1  2  3  4
1  2  3  4
1  2  3  4
1  2  3  4
1  2  3  4
1  2  3  4
1  2  3  4
1  2  3  4
```
Corank 3 for 3D models

\[ \text{integer} :: i(n,n,n) [3,2,*] \]

on 18 images results in a logical arrangement of images as \((3 \times 2 \times 3)\)

\[ \text{integer} :: i(n,n,n) [4,4,*] \]

on 64 images results in a logical arrangement of images as \((4 \times 4 \times 4)\)
Remote calls

```plaintext
integer :: i[*], j
real :: r(3,8) [4,*]
i[5] = i  ; remote write – this image
; copies its i to i on
; image 5.

r(:, :) = r(:, :) [3,3]  ; remote read – this image
; copies the whole array r
; from image with cosubscript
; set (3,3) to its own
; array r.

i = j  ; both i and j taken from
; the invoking image
```

Syntax without [] always refers to a variable on the invoking image.
A Coarray program consists of one or more execution segments.

The segments are separated by image control statements.

If there are no image control statements in a program, then this program has a single execution segment.

sync all is a global barrier, similar to MPI_Barrier.
SYNC ALL image control statement

- If it is used on any image, then every image must execute this statement.
- On reaching this statement each image waits for each other.
- Its effect is in ordering the execution segments on all images. All statements on all images before \texttt{sync all} must complete before any image starts executing statements after \texttt{sync all}.
Coarray segment rules 1

▶ From [1]:

*If a variable is defined on an image in a segment, it shall not be referenced, defined or become undefined in a segment on another image unless the segments are ordered.*

▶ A (simple) standard conforming coarray program should not deadlock or suffer from races.

▶ All coarray programs implicitly synchronise at start and at termination.
More flexible means of image control.

`sync images` takes a list of image indices with which it must synchronise:

```fortran
if ( this_image() .eq. 3 ) sync images( (/ 2, 4, 5 /) )
```

There must be corresponding `sync images` statements on the images referenced by `sync images` statement on image 3, e.g.:

```fortran
if ( this_image() .eq. 2 ) sync images( 3 )
if ( this_image() .eq. 4 ) sync images( 3 )
if ( this_image() .eq. 5 ) sync images( 3 )
```

Asterisk, *, is an allowed input. The meaning is that an image must synchronise with all other images:

```fortran
if ( this_image() .eq. 1 ) sync images( * )
if ( this_image() .ne. 1 ) sync images( 1 )
```

In this example all images must synchronise with image 1, but not with each other, as would have been the case with `sync all`. 
When there are multiple sync images statements with identical sets of image indices, the standard sets the rules which determine which sync images statements correspond. From [1]:

*Executions of SYNC IMAGES statements on images M and T correspond if the number of times image M has executed a SYNC IMAGES statement with T in its image set is the same as the number of times image T has executed a SYNC IMAGES statement with M in its image set. The segments that executed before the SYNC IMAGES statement on either image precede the segments that execute after the corresponding SYNC IMAGES statement on the other image.*
SYNC IMAGES - swapping a value between 2 images

$ cat swap.f90
integer :: img, nimgs, i[*], tmp ! implicit sync all
    img = this_image()
    nimgs = num_images()
    i = img          ! i is ready to use
    if ( img .eq. 1 ) then
        sync images( nimgs ) ! sync: 1 <-> last (1)
        tmp = i[ nimgs ]
        sync images( nimgs ) ! sync: 1 <-> last (2)
        i = tmp
    end if
    if ( img .eq. nimgs ) then
        sync images( 1 )  ! sync: last <-> 1 (1)
        tmp = i[ 1 ]
        sync images( 1 )  ! sync: last <-> 1 (2)
        i = tmp
    end if
    write (*,*) img, i
end ! all other images wait here

- How many execution segments are there on each image?
- Which sync images statements correspond?
Swapping a value between 2 images - image 1

```f90
$ cat swap.f90

integer :: img, nimgs, i[*], tmp ! implicit sync all

img = this_image()

nimgs = num_images()

i = img ! i is ready to use

sync images(nimgs) ! sync: 1 <-> last (1)

tmp = i[nimgs]

sync images(nimgs) ! sync: 1 <-> last (2)

i = tmp

write(*,*), img, i

end ! all other images wait here
```
Swapping a value between 2 images - last image

```fortran
$ cat swap.f90

integer :: img, nimgs, i[*], tmp ! implicit sync all
img = this_image()
nimgs = num_images()
  i = img
  ! i is ready to use
  sync images(1) ! sync: last <-> 1 (1)
  tmp = i[1]
  sync images(1) ! sync: last <-> 1 (2)
  i = tmp
write (*.*) img, i
end ! all other images wait here
```
Swapping a value between 2 images - other images

```
$ cat swap.f90

integer :: img, nimgs, i[*], tmp ! implicit sync all
img = this_image()
nimgs = num_images()
i = img                              ! i is ready to use
write (*,*) img, i
end                                        ! all other images wait here
```

```
$ ifort -coarray swap.f90
$ setenv FOR_COARRAY_NUM_IMAGES 5
$ ./a.out

  3    3
  1    5
  2    2
  4    4
  5    1
```
New Fortran 2008 construct: **DO CONCURRENT**

- For when the order of loop iterations is of no importance. The idea is that such loops can be optimised by compiler.

```fortran
integer :: i, a1(100)=0, a2(100)=1

do concurrent ( i=1, 100 )
    a1(i) = i  ! valid, independent

    a2(i) = sum( a2(1:i) ) ! invalid,
    ! order is important
end do
```

- The exact list of restrictions on what can appear inside a **do concurrent** loop is long. These restrictions severely limit its usefulness.

- Potentially a portable parallelisation tool, there might or might not be a performance gain, depending on the implementation.
Implementation and performance

- The standard deliberately (and wisely) says nothing on this.
- A variety of parallel technologies can be used - MPI, OpenMP, SHMEM, GASNet, ARMCI, DMAPP, etc. As always, performance depends on a multitude of factors.
- The standard *expects*, but does not require, that coarrays are implemented in a way that each image knows the address of all coarrays in memories of all images, i.e. using *symmetric memory*:
Example: calculation of $\pi$ using the Gregory - Leibniz series

$$\pi = 4 \sum_{n=1}^{\infty} \frac{(-1)^{n-1}}{2n - 1}$$

- Each image sums the terms beginning with its image number and with a stride equal to the number of images. Then image 1 sums the contributions from all images.
Calculation of $\pi$ - comparing coarrays, MPI, OpenMP and DO CONCURRENT

The partial $\pi$ loop, and the total $\pi$ calculation.

1. Coarrays

```fortran
!
do i = this_image(), limit, num_images()
  pi = pi + (-1)**(i+1) / real( 2*i-1, kind=rk )
!end do
sync all ! global barrier
if (img .eq. 1) then
  do i = 2, nimgs
    pi = pi + pi[i]
  end do
  pi = pi * 4.0rk
!end if
```

2. MPI

```fortran
!
do i = rank+1, limit, nprocs
  pi = pi + (-1)**(i+1) / real( 2*i-1, kind=rk )
!end do
call MPI_REDUCE( pi, picalc, 1, MPI_DOUBLE_PRECISION, &
                 MPI_SUM, 0, MPI_COMM_WORLD, ierr )
!
picalc = picalc * 4.0rk
```
Calculation of $\pi$ - comparing coarrays, MPI, OpenMP and DO CONCURRENT

3. DO CONCURRENT

```plaintext
loops = limit / dc_limit
do j = 1, loops
    shift = (j-1)*dc_limit
    do concurrent (i = 1:dc_limit)
        pi(i) = (-1)**(shift+i+1) / real(2*(shift+i)-1, kind=rk)
    end do
    pi_calc = pi_calc + sum(pi)
end do
pi_calc = pi_calc * 4.0_rk
```

4. OpenMP

```plaintext
!$OMP PARALLEL DO DEFAULT(NONE) PRIVATE(i) REDUCTION(+:pi)
do i = 1, limit
    pi = pi + (-1)**(i+1) / real( 2*i-1, kind=rk )
end do
!$OMP END PARALLEL DO
pi = pi * 4.0_rk
```
Calculation of $\pi$ - comparing coarrays, MPI, OpenMP and DO CONCURRENT

Coarray implementation is closest to MPI. Coarray collectives, e.g. CO_SUM, are available already in Cray and GCC/OpenCoarrays compilers.

<table>
<thead>
<tr>
<th></th>
<th>Fortran standard</th>
<th>shared memory</th>
<th>distributed memory</th>
<th>ease of use</th>
<th>flexibility</th>
<th>performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coarrays</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>easy</td>
<td>high</td>
<td>high</td>
</tr>
<tr>
<td>Do concurrent</td>
<td>yes</td>
<td>possibly</td>
<td>possibly</td>
<td>easy</td>
<td>poor</td>
<td>uncertain</td>
</tr>
<tr>
<td>OpenMP</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>easy</td>
<td>limited</td>
<td>medium</td>
</tr>
<tr>
<td>MPI</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>hard</td>
<td>high</td>
<td>high</td>
</tr>
</tbody>
</table>

Table: A highly subjective comparison of Fortran coarrays, MPI, OpenMP and Fortran 2008 do concurrent.
Allocatable coarrays

- The last upper codimension must be an asterisk on allocation, to allow for the number of images to be determined at runtime:

```fortran
! real allocatable array coarray
real, allocatable :: r(:) [:]
! complex allocatable scalar coarray
complex, allocatable :: c[:]
! integer allocatable array coarray
! with 3 codimensions
integer, allocatable :: i(:, :, :, :) [:, :, :, :]
```

```fortran
allocate( r(100) [*], source=0.0 )
allocate( c[*], source=cmplx(0.0,0.0) )
allocate( i(3,3,3,3) [7,8,*], stat=errstat )
```

- Coarrays must be allocated with the same bounds and cobounds on all images (symmetric memory!).

```fortran
allocate( r(10*this_image()) [*] ) ! NOT VALID
allocate( c[7*this_image(), 8,*] ) ! NOT VALID
```
Allocatable coarrays

- Allocation and deallocation of coarrays involve implicit image synchronisation.
- All images must allocate and deallocate allocatable coarrays together.
- All allocated coarrays are automatically deallocated at program termination.
- Allocatable coarrays can be passed as arguments to procedures.
- If a coarray is allocated in a procedure, the dummy argument must be declared with `intent(inout)`.
- The bounds and cobounds of the actual argument must match those of the dummy argument.

```fortran
subroutine coal(i, b, cob)
  integer, allocatable, intent(inout) :: i(:, :, :)
  integer, intent(in) :: b, cob
  if (.not. allocated(i)) allocate( i(b) [cob, *] )
end subroutine coal
```
Coarrays of derived types with allocatable components

```f90
$ cat pointer.f90

type t
  integer, allocatable :: i(:)
end type

type(t) :: value[*]
  integer :: img
  img = this_image() 
allocate( value%i(img), source=img ) ! no sync here
sync all ! separate execution segments
if ( img .eq. num_images() ) value%i(1)=value[1]%i(1)
write (*,*), "img", img, value%i
end

$ ifort -coarray -warn all -o pointer.x pointer.f90
$ setenv FOR_COARRAY_NUM_IMAGES 3
$ ./pointer.x

img  1  1
img  2  2
img  3  1  3
```

Termination

- **normal** and **error** termination.
- Normal termination on one image allows other images to finish their work. `stop` and `end program` initiate normal termination.
- New intrinsic **error stop** initiates error termination. The purpose of error termination is to terminate all images as soon as possible.
- Example of a normal termination:

```fortran
integer :: img
img = this_image()
if (img .eq. 1) stop "img1: avoiding div by zero"
write (*,*) "img:" , img , "val:" , 1.0 / (img - 1 )
end
```

```
$ ifort -coarray -coarray -num_images=3 z.f90
$ ./a.out
  img: 2 val: 1.000000
  img1: avoiding div by zero
  img: 3 val: 0.5000000
```
Error termination

```fortran
integer :: img
img = this_image()
if (img .eq. 1) & error stop "img1: avoiding div by zero"
write (*,*) "img:" , img , "val:" , 1.0 / (img - 1 )
end
```

$ ifort -coarray -coarray -num_images=3 z.f90
$ ./a.out
application called
MPI_Abort(comm=0x84000000 , 3) — process 0

- Use it for truly catastrophic conditions, when saving partial data or continuing makes no sense.
Dealing with (soft/easy) failures

- Use `stat=` specifier in `sync all` or `sync images` to detect whether any image has initiated normal termination.

- If at the point of an image control statement some image has already initiated normal termination, then the integer variable given to `stat=` will be defined with the constant `stat_stopped_image` from the intrinsic module `iso_fortran_env`.

- The images that are still executing might decide to take a certain action with this knowledge:

```fortran
use, intrinsic :: iso_fortran_env
integer :: errstat=0
   ! all images do work
sync all( stat=errstat )
if ( errstat .eq. stat_stopped_image ) then
   ! save my data and exit
end if
! otherwise continue normally
```
Fortran 2018 collectives (Cray, GCC/OpenCoarrays)

CO_MAX (A [, RESULT_IMAGE, STAT, ERRMSG])
CO_MIN (A [, RESULT_IMAGE, STAT, ERRMSG])
CO_SUM (A [, RESULT_IMAGE, STAT, ERRMSG])
CO_BROADCAST (A, SOURCE_IMAGE [, STAT, ERRMSG])
CO_REDUCE (A, OPERATOR [, RESULT_IMAGE, STAT, ERRMSG])

▶ "A" does not need to be a coarray variable!
▶ "A" is overwritten on all images, or, if result_image is given, then only on that image:

$ cat z.f90
integer :: img, z
img = this_image()
call co_sum(img, 1)
write (*,*) img
end

$ caf z.f90
$ cafrun -np 5 a.out
15
2
3
4
5
Atomics

- Fortran 2008: \texttt{atomic\_define}, \texttt{atomic\_ref}
- Fortran 2018 added: \texttt{atomic\_add}, \texttt{atomic\_and}, \texttt{atomic\_cas}, \texttt{atomic\_fetch\_add}, \texttt{atomic\_fetch\_and}, \texttt{atomic\_fetch\_or}, \texttt{atomic\_fetch\_xor}, \texttt{atomic\_or}, \texttt{atomic\_xor}.
- Must define and reference atomic variables \textit{only} through atomic routines!

```fortran
use, intrinsic :: iso_fortran_env, only : atomic_int_kind
integer(atomic_int_kind) :: x[*]=0, z=0
call atomic_add( x[1], 1 )
if (this_image() .eq. num_images()) then
  do
    call atomic_ref( z, x[1] )
    if ( z .eq. num_images() ) exit
  end do
end if
end
```
**Atomics and SYNC MEMORY**

- **sync memory** adds a segment boundary *with no* synchronisation.

```fortran
use, intrinsic :: iso_fortran_env, only : atomic_int_kind
integer(atomic_int_kind) :: x[*], z=0

call atomic_define(x, 0)

sync memory              ! segment boundary 1
call atomic_add(x[1], 1)
if (this_image() .eq. num_images()) then
  do
    call atomic_ref(z, x[1])
    if (z .eq. num_images()) exit
  end do
end if

sync memory              ! segment boundary 2
if (this_image() .eq. num_images()) write(*,*) z
end
```

$ caf z.f90
$ cafrun -np 6 a.out
  6
$ cafrun -np 17 a.out
  17
A **critical / end critical** construct limits execution of a block to one image at a time:

```
critical
end critical
```

The order of execution of the critical section by images is unpredictable.

Critical is a serial operation - bad for performance.
Locks

- A **lock**/ **unlock** construct. Locks are coarray variables of derived type **lock_type**
- Use locks to avoid races on shared resources, e.g. global variables.

```fortran
use, intrinsic :: iso_fortran_env, &
  only : lock_type

type(lock_type) :: l[*]

integer :: i[*] = 0

lock( l[ num_images()] ) ! segment boundary
  i[1] = i[1] + 1
unlock( l[ num_images()] ) ! segment boundary

sync all

if ( this_image().eq. 1) write(*,*) i
end
```

```
$ caf z.f90
$ cafrun -np 16 a.out
  16
```
Further Fortran 2018 extensions

- **Teams** of images. Can create a team of a subset of all images, do some work in the team, synchronise with just the team, etc. FORM TEAM, END TEAM, CHANGE TEAM, SYNC TEAM.

- **Events.** A more flexible way to synchronise. Can post events (EVENT POST), wait for a specified number of events to occur (EVENT WAIT), and query the event counter (EVENT_QUERY).

- **Facilities for detecting image failures** (and possibly dealing with them). New status value STAT_FAILED_IMAGE which typically means hardware or system software failure. New intrinsic functions: IMAGE_STATUS, STOPPED_IMAGES, FAILED_IMAGES.

- See Bill Long’s article [3] for more details.
Books with coarray examples

- R. J. Hanson, T. Hopkins, Numerical Computing with Modern Fortran, SIAM, 2013
Coarray resources

- comp.lang.fortran usenet group - many participants, few experts, any topic - coarrays, OOP, libraries, etc.
- WG5 Fortran standards ➤ WG5 page
- COMP-FORTRAN-90 mailing list ➤ web interface
- Compiler forums and mailing lists, e.g. GCC, Intel...
- ACM SIGPLAN Fortran Forum journal ➤ home page
- Ian Chivers, Jane Sleightholme, Compiler Support for the Fortran 2003 and 2008 Standards Revision 21, ACM SIGPLAN Fortran Forum 36 issue 1, APR-2017, p.21-42:

<table>
<thead>
<tr>
<th>Fortran 2008 Features</th>
<th>Absoft</th>
<th>Cray</th>
<th>g95</th>
<th>gfortran</th>
<th>HP</th>
<th>IBM</th>
<th>Intel</th>
<th>NAG</th>
<th>Oracle</th>
<th>Path scale</th>
<th>PGI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler version number</td>
<td>14</td>
<td>8.4.0</td>
<td>5.2</td>
<td>15.1.3</td>
<td>17.0</td>
<td>6.1</td>
<td>8.7, 32</td>
<td>6.0</td>
<td>16.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Submodules</td>
<td>N</td>
<td>Y</td>
<td>N, 201</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Coarrays</td>
<td>N</td>
<td>Y</td>
<td>P</td>
<td>Y, 301</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Regularly updated ➤ online
