Coarray Fortran

A Partitioned Global Address Space Language

Author: Reinhold Bader

Fortran and Parallelism

- **Design target:**

  Permit Fortran programs to run in **SPMD** mode natively

  - SPMD: "single program multiple data"

- **Design considerations:**

  smallest changes required to convert Fortran
  into a robust and efficient parallel language

  - add only a few new rules to the language

- **Standardization effort:**
  - Baseline features in Fortran 2008
    (ISO/IEC 1539-1:2010, published in October 2010)
  - Further parallel features in Fortran 2018
    (ISO/IEC 1539-1:2018, published in November 2018)

Feature Overview

Fortran 2008
SPMD execution
Partitioned memory data model ("**coarrays**")
One-sided communication ("coindexing")
Synchronization against data races
Memory management for coarrays

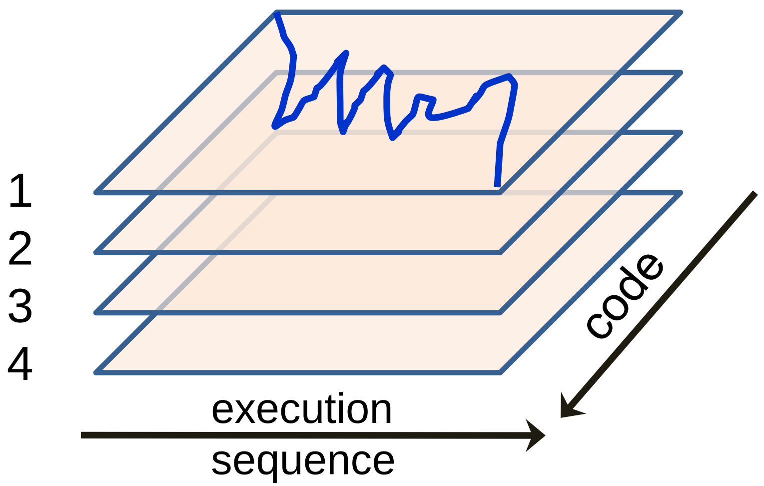
Fortran 2018
Collective subroutines
One-sided synchronization ("events")
Composable parallelism ("teams")
Atomic subroutines
Continue execution after image failure **(optional)**

Current compiler support

Compiler	Version	Extent of support
NAG (nagfor)	7.0	Shared Memory only, most F18 features
GCC (gfortran)	10.2	Distributed Memory (uses MPI), partial support
Cray (ftn)	10	Distributed Memory, all F18 features (Cray systems)
Intel (ifort)	2021	Distributed Memory (uses MPI), all F18 features

SPMD execution model

- **Concept of image:**



  - image count:
    between 1 and number of images

- **Replicate program a fixed number of times**
  - set number of replicates at **compile** time or at **execution** time (processor dependent method)
  - asynchronous execution – **loose** coupling (unless program-controlled synchronization occurs)
- **Separate set of entities on each image**
  - program-controlled exchange of data
  - usually necessitates synchronization

Simplest possible program

- **Uses intrinsic functions for image management**

```
PROGRAM hello
  IMPLICIT none
  WRITE(*, '(''Hello from image '',i0, '' of '',i0)') &
        this_image(), num_images()
END PROGRAM
```

file: `parallel_hello.f90`

- `num_images()`
  - The form without arguments returns the number of images (set by environment) – returns a default-kind integer
- `this_image()`
  - generic intrinsic. The form without arguments returns a number between `1` and `num_images()` – returns a default-kind integer

Compiling and running the program (NAG)

- **Compilation**

optional: compile-time image count

```
nagfor -f2018 -coarray [-num_images=2] -o parallel_hello.exe parallel_hello.f90
```

  - note: compiling with **`-coarray=single`** permits executing with a single image only

- **Execution**
  - with 4 images

overrides compile-time setting

```
export NAGFORTRAN_NUM_IMAGES=4

./parallel_hello.exe
```

- **Output**

```
Hello from image 2 of 4
Hello from image 1 of 4
Hello from image 3 of 4
Hello from image 4 of 4
```

non-repeatably unsorted output if multiple images are used

Compiling and running the program (Intel)

Compilation

optional: compile-time image count

```
ifort -coarray [-coarray-num_images=2] -o parallel_hello.exe parallel_hello.f90
```

- note: compiling with **`-coarray=single`**
  permits executing with a single image only
- compiling with **`-coarray=distributed`**
  permits execution with multiple nodes (config file required)

Execution

- with 4 images

overrides compile-time setting

```
export FOR_COARRAY_NUM_IMAGES=4

./parallel_hello.exe
```

Output

```
Hello from image 2 of 4
Hello from image 1 of 4
Hello from image 3 of 4
Hello from image 4 of 4
```

non-repeatably unsorted output
if multiple images are used

Compiling and running the program (GCC)

- **Compilation:** Use Opencoarrays wrapper for gfortran

```
caf -o parallel_hello.exe parallel_hello.f90
```

  - note: compiling with additional **`-fcoarray=single`** option limits execution to with a single image only
  - See also http://www.opencoarrays.org/

- **Output**

```
Hello from image 2 of 4
Hello from image 1 of 4
Hello from image 3 of 4
Hello from image 4 of 4
```

non-repeatably unsorted output
if multiple images are used

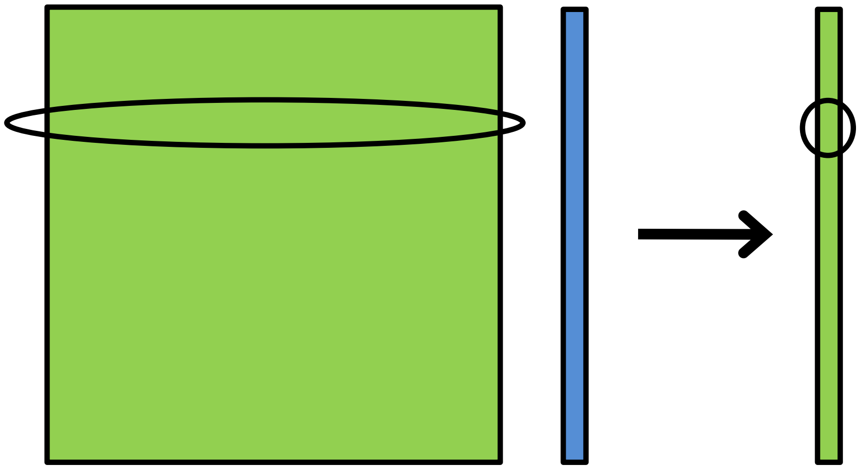
- **Execution**
  - with 4 images

```
cafrun -n 4 ./parallel_hello.exe
```

A more elaborate example: Matrix-Vector Multiplication

$$\sum_{j=1}^{n} M_{ij} \cdot v_j = b_i$$

- **Basic building block for many algorithms**



  - independent collection of scalar products

Serial Matrix-Vector code

```
INTEGER, PARAMETER :: N = …
INTEGER :: icol, irow
REAL :: Mat(N, N), V(N)
REAL :: B(N)   ! result

DO icol=1,N
  DO irow=1,N
    Mat(irow,icol) = matval(irow,icol)
  END DO
  V(icol) = vecval(icol)
END DO

CALL sgemv(‘n‘,N,N,1.0,Mat,N,V,1,0.0,B,1)

: ! Use result
```

Prepare input data

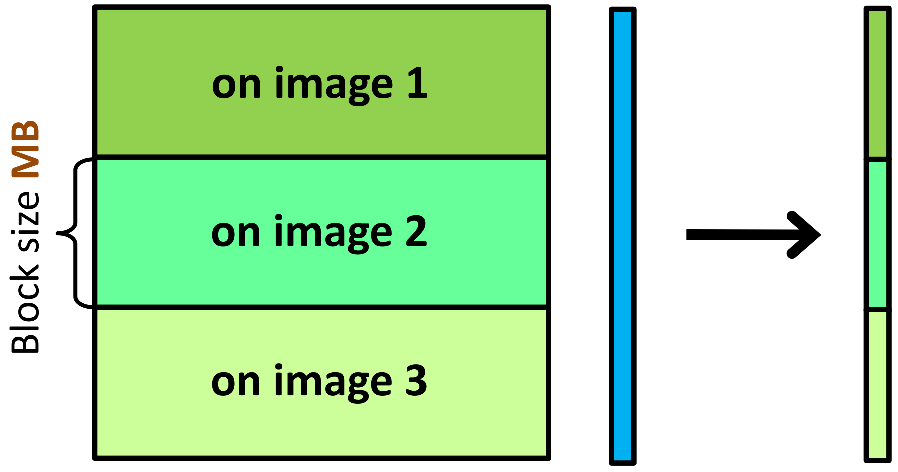
BLAS routine does calculation

- functions **`matval()`** and **`vecval()`** calculate matrix elements and input vectors

Nearly-trivial parallelism: Data decomposition

Block row distribution:

- calculate only a block of B on each image (but that completely)
- the shading indicates the assignment of data to images
- blue: data are replicated on all images



Further alternatives:

- cyclic, block-cyclic
- column, row and column

Memory requirement:

- $(n^2 + n)$ / <no. of images> + n words per image/thread
- load balanced (same compu-tational load on each task)

Assumption: MB == N / (no. of images)

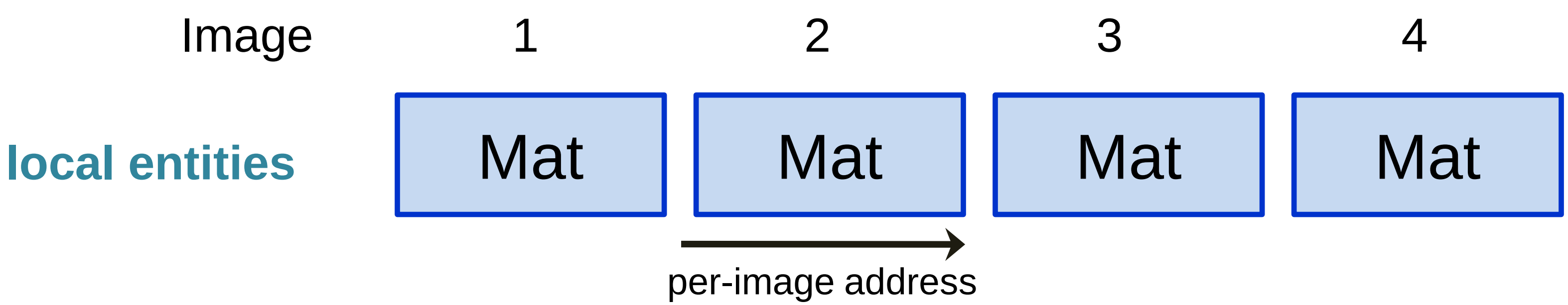
- dynamic allocation is more flexible
- if mod(N, no. of images) > 0, conditioning is required

Memory model part 1: Image-local entities

Modified declarations

```
REAL :: Mat(MB, N), V(N)
REAL :: B(MB)
```

Semantics for PGAS replicated execution



- each image has its **local** (or **private**) copy of any declared object

  „private": as in OpenMP, but here is the **default**

- private objects are only accessible to the image which „owns" them

  (extrapolated from conventional "serial" language semantics, and consistent with executing in serial mode i.e. only one image)

Work sharing of initialization and M*v processing

"Fragmented data" model

- need to calculate **global** row index from local iteration variable (or vice versa)

```
DO icol=1,N
 DO i=1,MB
   irow = (this_image() - 1) * MB + i
   Mat(i,icol) = matval(irow,icol)
 END DO
 V(icol) = vecval(icol)
END DO

CALL sgemv('n',MB,N,1.0,Mat,MB,V,1,0.0,B,1)
```

`i` is image-local index;
need to calculate global index **`irow`**

each image:
works on its own, private instances of Mat, V, B

- degenerates into serial version of code for 1 image

Work sharing: General mapping of data to images

Index transformation for an array dimension

- a one-to-one mapping between local and global indices

$$(i)[p] \leftrightarrow j$$

local array index

image index

global index

conceptual notation only

- local problem size on image **`p`**: **`nlocal{p}`**

```
REAL :: a(ndim, …)
p = this_image()
DO i=1, nlocal
   j = …        ! global index
   a(i,…) = … ! expression involving j
END DO
```

`ndim` large enough to hold **`nlocal{p}`** elements

may vary between images

$$j = \sum_{q=1}^{p-1} nlocal\{q\} + i$$

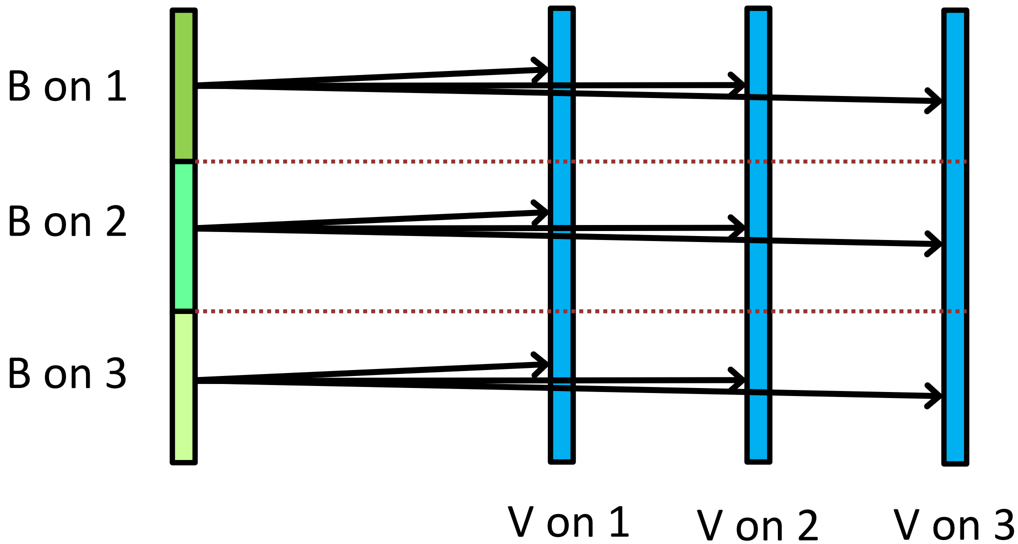
for a **blocked** distribution

- for a work-balanced problem: **`nlocal{p}`** typically the same on all images (some of the last images may have a slightly smaller value)

Inter-image data transfer
and
Synchronization

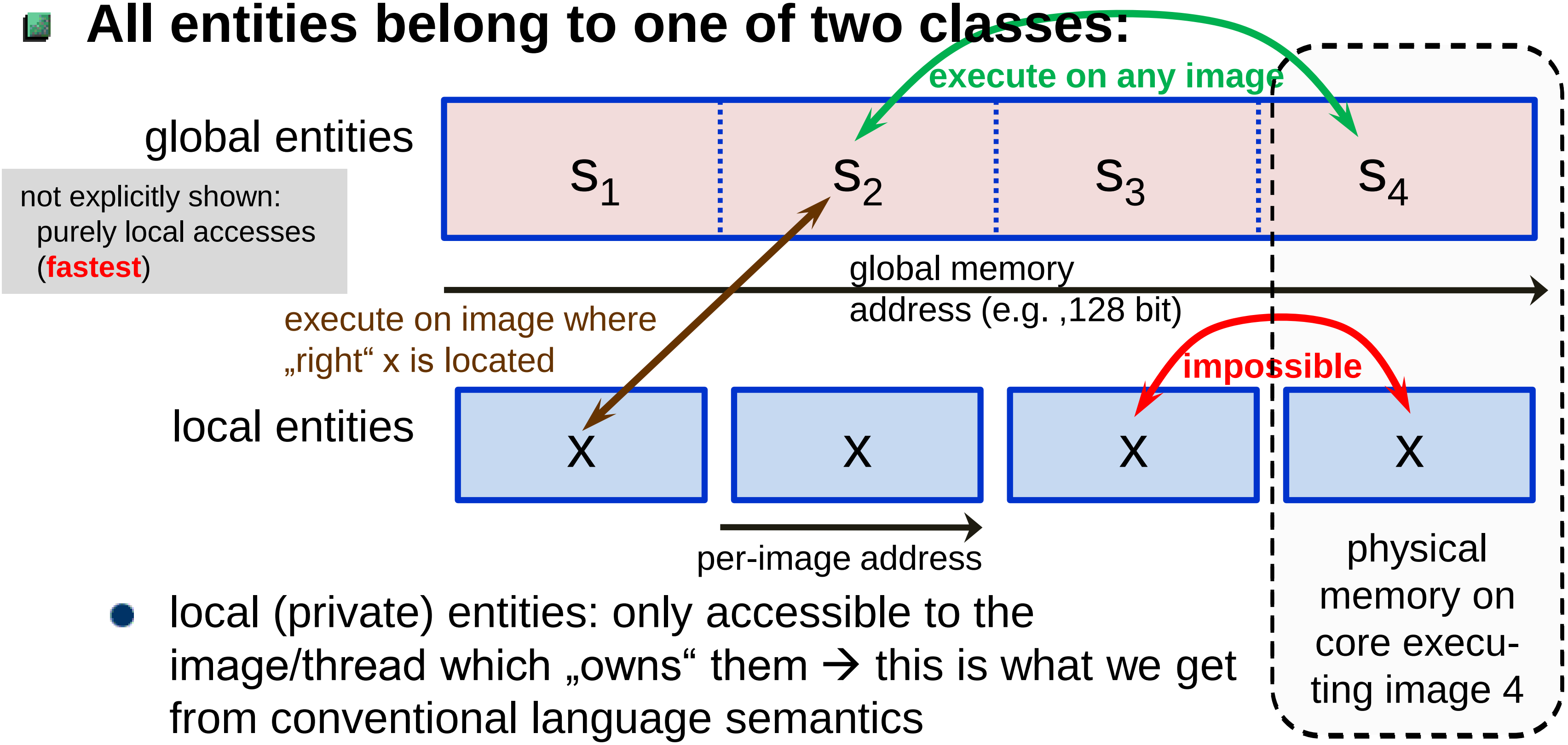
Illustrating the need for communication

- **Open issue from M * v example**
  - iterative solvers require **repeated** evaluation of matrix-vector product
  - but the result we received is distributed across the images
- **Therefore, a method is needed**
  - to **transfer** each B to the appropriate portion of V on all images



PGAS data and memory model

- **All entities belong to one of two classes:**



- local (private) entities: only accessible to the image/thread which „owns“ them → this is what we get from conventional language semantics
- global (**shared**) entities in partitioned global memory: objects declared on and physically assigned to one image/thread may be accessed by any other one
- allows implementation for distributed memory systems

the term „shared“:
→ slightly **different** semantics than in OpenMP

Declaration of coarrays / shared entities (simplest case)

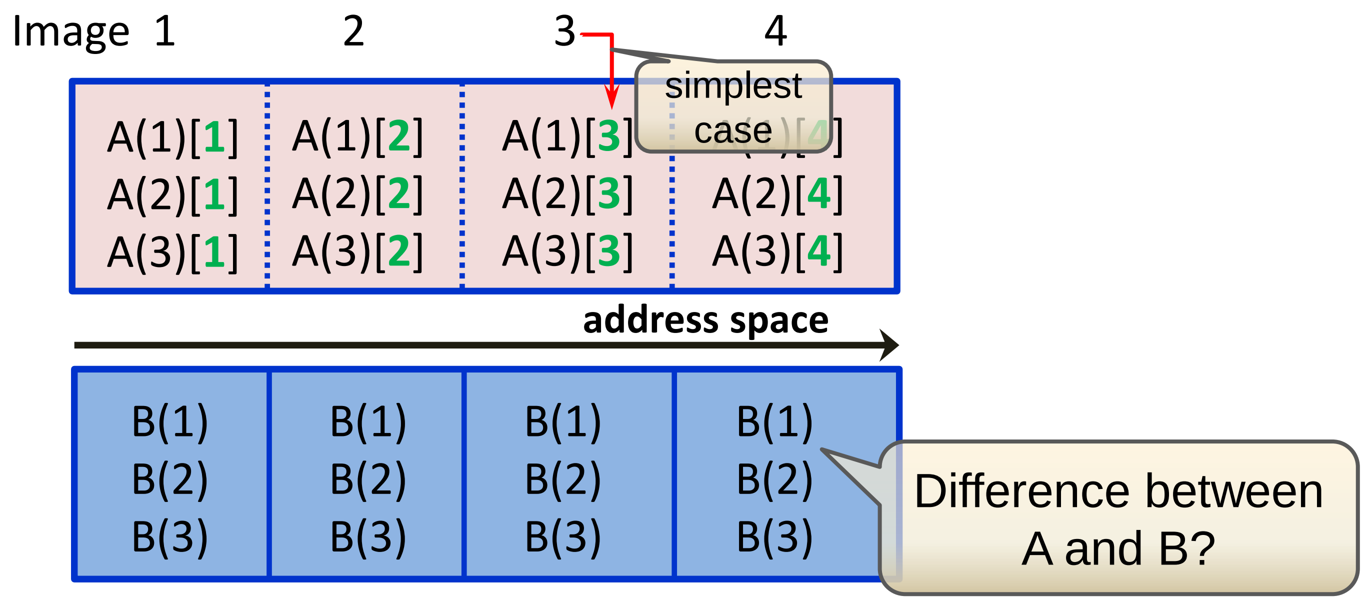
- **Coarray declaration**
  - **symmetric** objects

```
INTEGER :: b(3)
INTEGER :: a(3)[*]
```

equivalent alternative to declaration shown to the left

```
INTEGER, CODIMENSION[*] :: a(3)
```

- **Execute with 4 images**



  - one-to-one mapping of **coindex** to image index

Further declaration variants

- **a scalar coarray**

```
INTEGER, CODIMENSION[*] :: s
```

- **support for cartesian topology:**
  example for a corank 2 coarray with non-default lower cobound

```
REAL :: c(ndim, ndim)[0:pdim,*]
```

Details explained later

- **coarray of derived type**
  dispense with all of `MPI_TYPE_*` API

```
TYPE(body) :: asteroids(ndim)[*]
```

POD types trivially usable, non-POD types have usage restrictions

- **polymorphic coarray**
  example for a polymorphic coarray dummy argument

```
CLASS(body) :: asteroids(:)[*]
```

Inter-image communication: coindexed access

- **Get**

```
IF (this_image() == p) &
       b = a(:)[q]
```

sectioning is obligatory

a **coindexed** reference

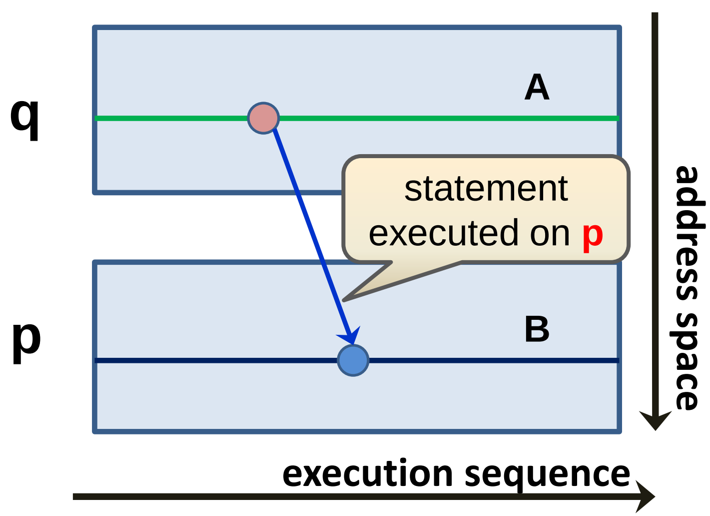
- **Put**

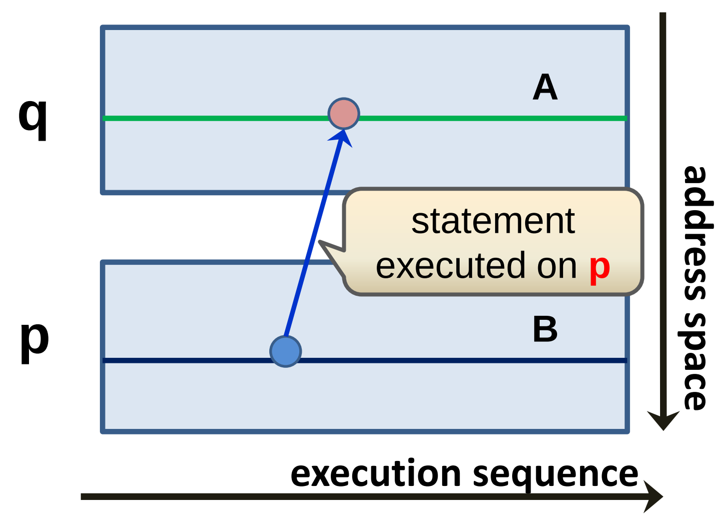
```
IF (this_image() == p) &
       a(:)[q] = b
```

a **coindexed** definition

assumption: **p** and **q** have the same value on all images, respectively

- **one-sided communication** between images **p** and **q**





Local accesses to coarrays

Design aim for non-coindexed accesses:

- should be optimizable as if they were local entities

```
INTEGER :: a(3)[*]
INTEGER :: i
a(:) = [ … ]
:
i = a(3) + …
:
CALL my_proc(a, …)
```

```
a(:)[this_image()] = [ … ]
```

same meaning, but likely slower execution speed

permitted: interface of **`my_proc`** declares dummy argument corresponding to **`a`** as **`real :: x(:)`** (not a coarray)

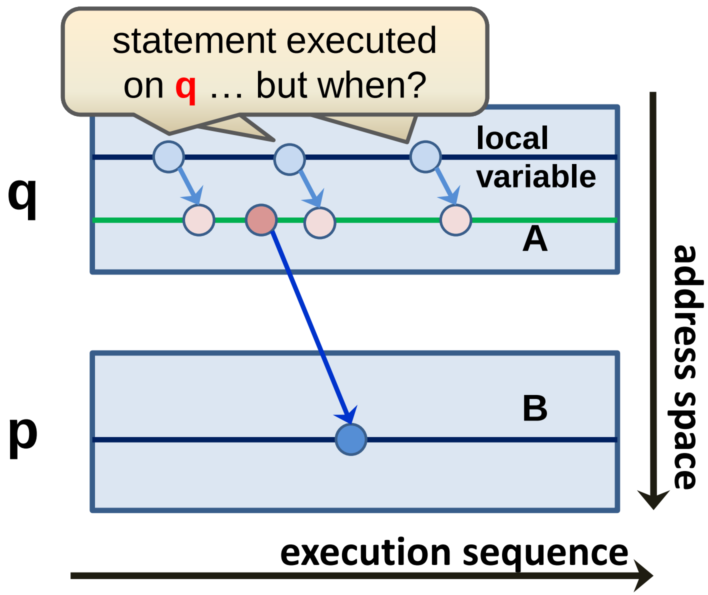
Explicit coindexing:

- indicates to programmer that communication is happening
- **distinguish:** coarray (a) ↔ coindexed entity (a[p])
- cosubscripts must be **scalars** of type integer

Synchronization requirements

Asynchronous execution

```
a = …
IF (this_image() == p) &
        b = a(:)[q]
```

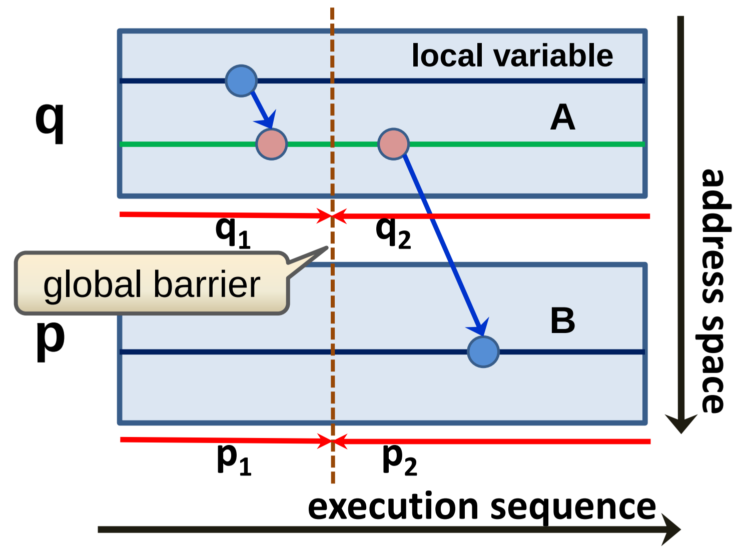


- causes race condition → **violates** language rules

Image control statement

```
a = …
SYNC ALL
IF (this_image() == p) &
        b = a(:)[q]
```

programmer's responsibility



- enforce segment ordering: **q_1 before p_2**, p_1 before q_2
- q_j and p_j are **unordered**

Semantics of SYNC ALL

- **All images synchronize:**
  - SYNC ALL provides a global barrier over **all** images
  - segments preceding the barrier on any image will be ordered before segments after the barrier on any other image → implies ordering of statement execution
- **If SYNC ALL is not executed by all images,**
  - the program will discontinue execution indefinitely (**deadlock**)
  - however, it is allowed to execute the synchronization via two different SYNC ALL statements
    (for example in two different subprograms)
- **For large image count or sparse communication patterns, exclusively using SYNC ALL may be too expensive**
  - limits scalability, depending on algorithm (load imbalance!)

  → we'll learn about alternatives later

General synchronization rules

- **Synchronization is required**
  - between segments on **any** two different images P, Q
  - which both access the **same entity** (may be local to P or Q or another image)

  (1) P writes and Q writes, or
  (2) P writes and Q reads, or
  (3) P reads and Q writes.

- **Status of dynamic entities**
  - replace „P writes“ by „P allocates“ or „P associates“
  - will be discussed later (additional constraints exist on who is allowed to allocate)
- **Synchronization is not required**
  - for concurrent reads
  - if entities are modified via atomic procedures (see later)

A special case where no synchronization is needed

- **Against compile-time initialized objects**
- **Example:**
  - a very inefficient method for calculating a sum

```
INTEGER :: count[*] = 1

IF (this_image() == 1) THEN
  DO i=2, num_images()
    count[i] = count[i] + count[i-1]
  END DO
  sum = count[num_images()]
END IF
```

no synchronization needed
because initialization
is done at compile time

no synchronization needed
because references and definitions
happen on the same image

- **Coindexing is not permitted in constant expressions that perform initialization** (e.g. DATA statements)

Image control statements needed for Get and Put patterns

p and **q** are assumed to have the same value on all images, respectively. Otherwise, more than one image pair communicates data.

Get

```
a = …
SYNC ALL
IF (this_image() == p) THEN
   b = a(:)[q]

   … = b
END IF
```

no sync required (no communication)

consume b on image **p**

Put

```
b = …

IF (this_image() == p) &
       a(:)[q] = b
: ! further statements
SYNC ALL
IF (this_image() == q) &
       … = a
```

consume a on image **q**

- might be asynchronously executed

Completing the M*v: Broadcast results to all images

Assumption: must update V on each image with values from B

Using "Get" implementation variant

- modified declaration

```
REAL :: Mat(MB, N), V(N)
REAL :: B(MB)
```

```
REAL :: Mat(MB, N), V(N)
REAL :: B(MB)[*]
```

only B needs to be accessible across images

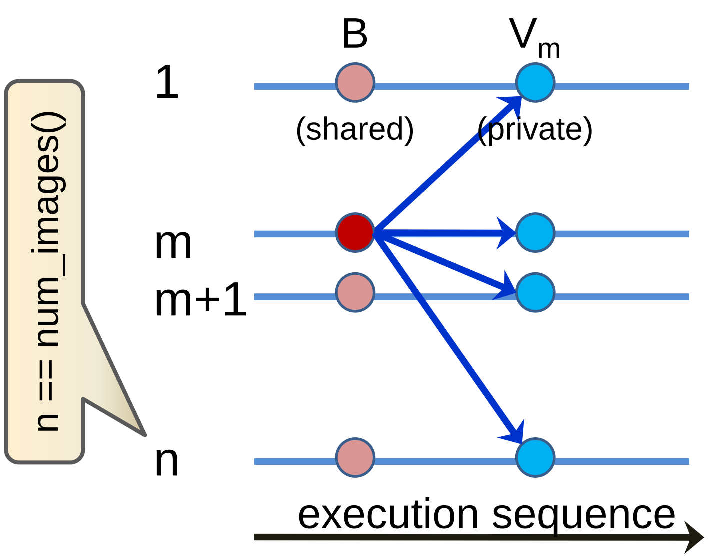
- first suggestion for communication code:

```
CALL sgemv(…)
SYNC ALL   ! assure remote B is available

DO m=1, num_images()
  V((m-1)*MB+1:m*MB) = B(:)[m]
END DO
: ! use V again
```

Formally, a correct solution ...
but what about **performance?**

Analyzing the communication pattern

In m-th loop iteration:



- effectively, a collectively executed scatter operation
- note that **each** image concurrently executes a communication statement

Slowest communication path

- might be a network link between two images, with bandwidth BW in units of GBytes/s
- subscription factor is **n**
- estimate for transfer duration of each loop iteration is

$$T = T_{lat} + \frac{MB * Size(real) * \boldsymbol{n}}{BW}$$

(latency T_{lat} included)

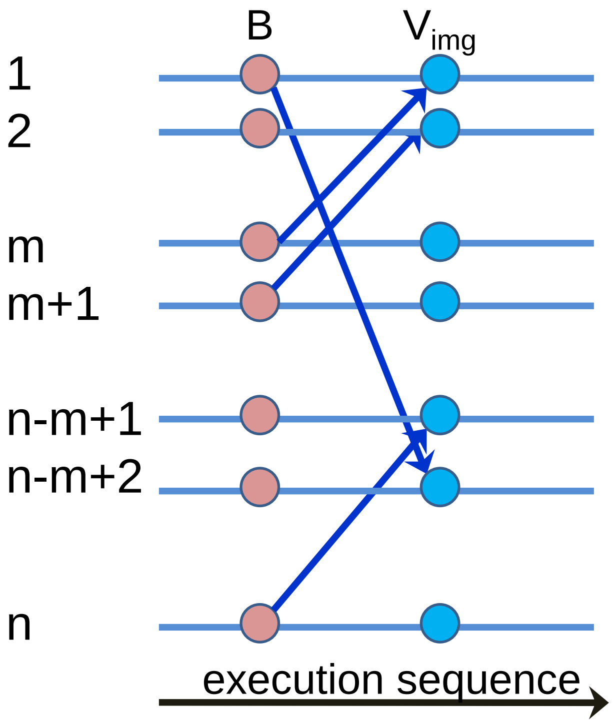
- this is **unfavourable** (an $\mathbf{n^2}$ effect when all loop iterations are accounted)

Improved communication pattern

- **Introduce a per-image shift of source image**
  - efficient pipelining of data transfer

```
CALL sgemv(…)
SYNC ALL   ! assure remote B
           ! is available
DO m=1, num_images()
  img = m + this_image() - 1
  IF (img > num_images()) &
      img = img - num_images()
  V((img-1)*MB+1:img*MB) = B(:)[img]
END DO
: ! use V again
```

In m-th loop iteration

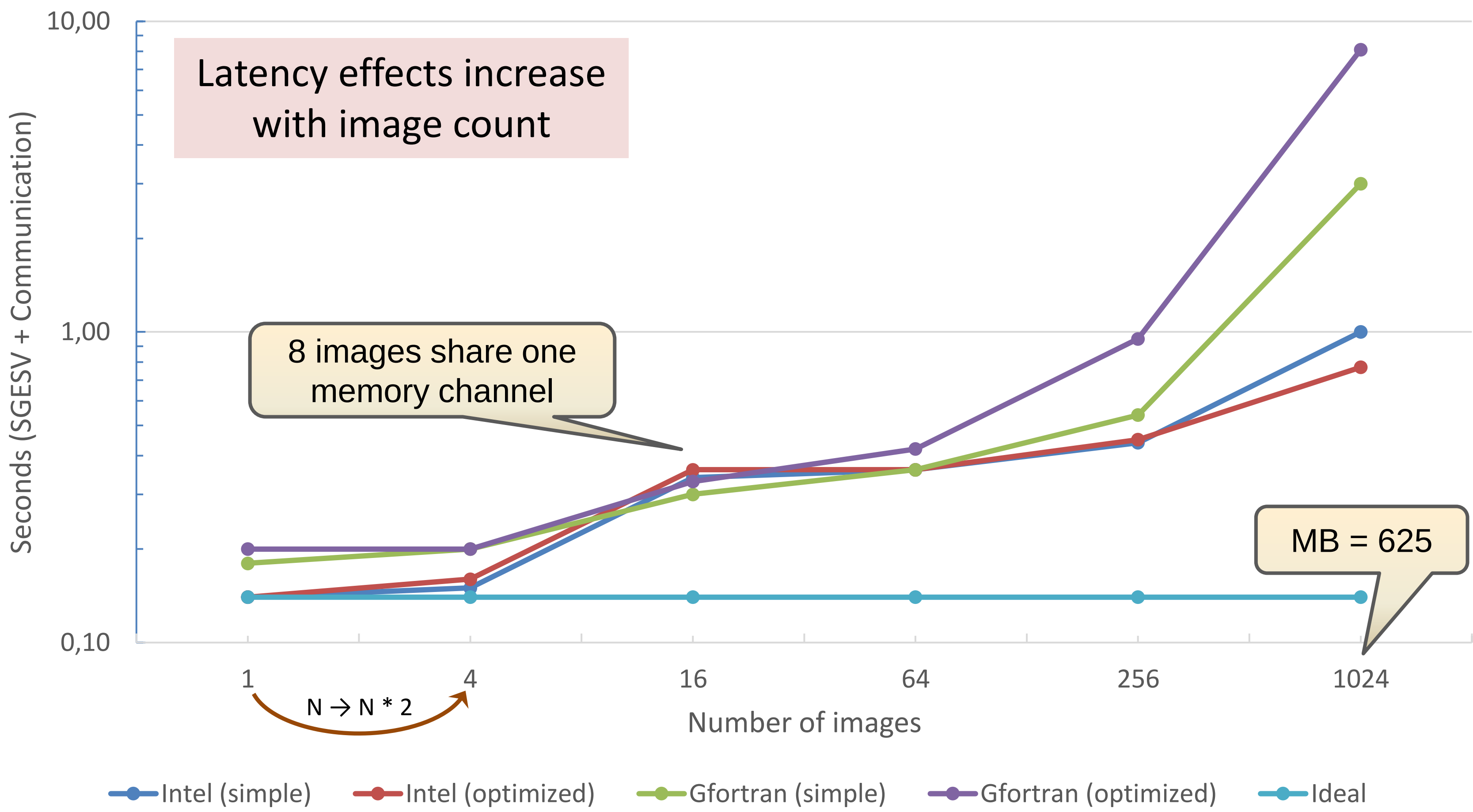


  - balanced use of network links:

$$T \le T_{lat} + \frac{MB * Size(real) * \mathbf{(images\ per\ node)}}{BW}$$

Weak scaling results: $N_{(1\ image)} = 20000$

on Sandy Bridge
with FDR 10



Collective Procedures

Added in F18

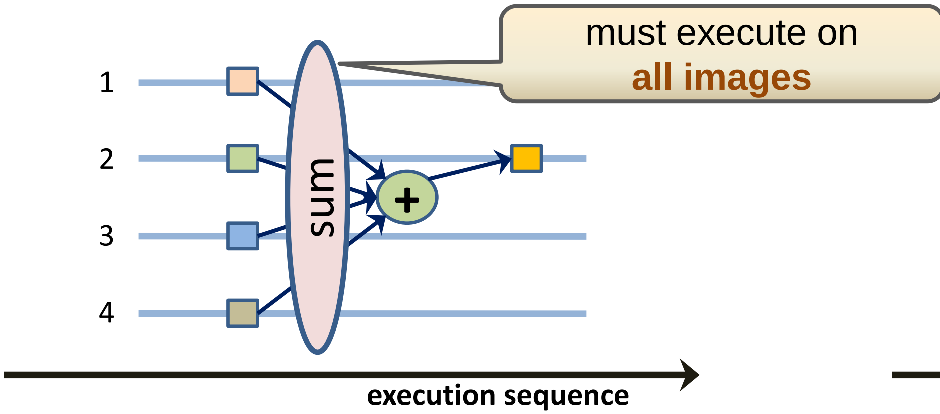
Motivation

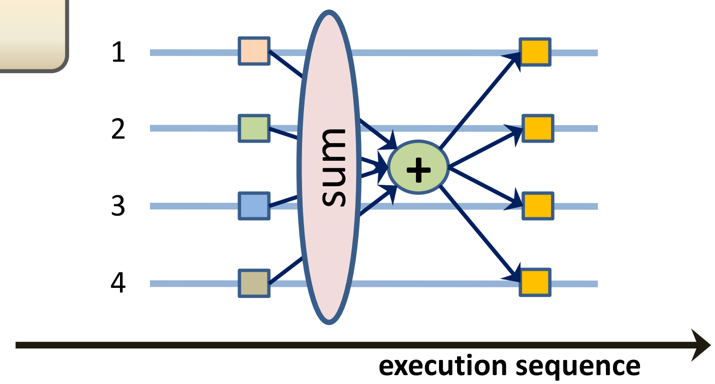
- **Common pattern in serial code:**
  - use of reduction intrinsics, for example:
    SUM for evaluation of global system properties

```
REAL :: mass(NDIM,NDIM), velocity(NDIM,NDIM)
REAL :: e_kin
:
e_kin = 0.5 * sum( mass * velocity**2 )
```

- **Coarray code:**
  - on each image, an image-dependent **partial sum** is evaluated
  - i. e. the intrinsic is not image-aware
- **Variables that need to have the same value across all images**
  - e.g. global problem sizes
  - values are initially often only known on one image

Reductions: CO_SUM, CO_MAX, CO_MIN






```
REAL :: a(2)
:
CALL co_sum(a, RESULT_IMAGE=2)
```

a becomes undefined on images ≠ 2

```
REAL :: a(2)
:
CALL co_sum(a)
```

a becomes defined on all images

- **Arguments:**
  - **a** may be a scalar or array of numeric type
  - **result_image** is an optional integer with value between 1 and num_images()
  - without **result_image**, the result is broadcast to **a** on all images, otherwise only to **a** on the specified image

Reductions with user-defined operations

Example: derived type

```
TYPE :: matrix
  : ! implementation detail
END TYPE
```

- might already have a specific used to overload addition

```
PURE FUNCTION matrix_plus(x, y) &
                        result(r)
  TYPE(matrix), intent(in) :: x, y
  TYPE(matrix) :: r
  : ! implementation detail
END FUNCTION
```

must be **mathematically associative**

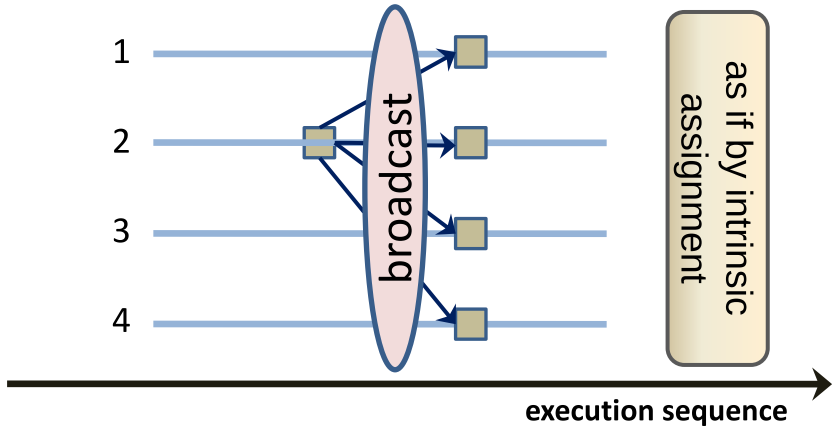
- PURE function with scalar, nonpolymorphic, nonallocatable, nonpointer, nonoptional arguments

CO_REDUCE:

```
TYPE(matrix) :: xm
:
CALL co_reduce(      A=xm, &
     OPERATOR=matrix_plus, &
     RESULT_IMAGE=2 )
```

- assignment to result is done as if it were intrinsic (finalizers might be invoked!)
- **`operator`** must be the same function on all images

Data redistribution with CO_BROADCAST




```
TYPE(matrix) :: xm
:
CALL co_broadcast(A=xm, SOURCE_IMAGE=2)
```

Arguments:

- **a** may be a scalar or array of any type. it must have the same type and shape on all images. It is overwritten with its value on **SOURCE_IMAGE** on all other images
- **SOURCE_IMAGE** is an integer with value between 1 and num_images()

Further comments on collective subroutines

- **All collectives are "in-place"**
  - programmer needs to copy data argument if original value is still needed
- **Data arguments need not be coarrays**
  - however if a coarray is supplied, it must be the same (ultimate) coarray on all images

For coarrays, all collectives could of course be implemented by the programmer. However it is expected that **collective subroutines will perform better**, apart from being more generic in semantics.

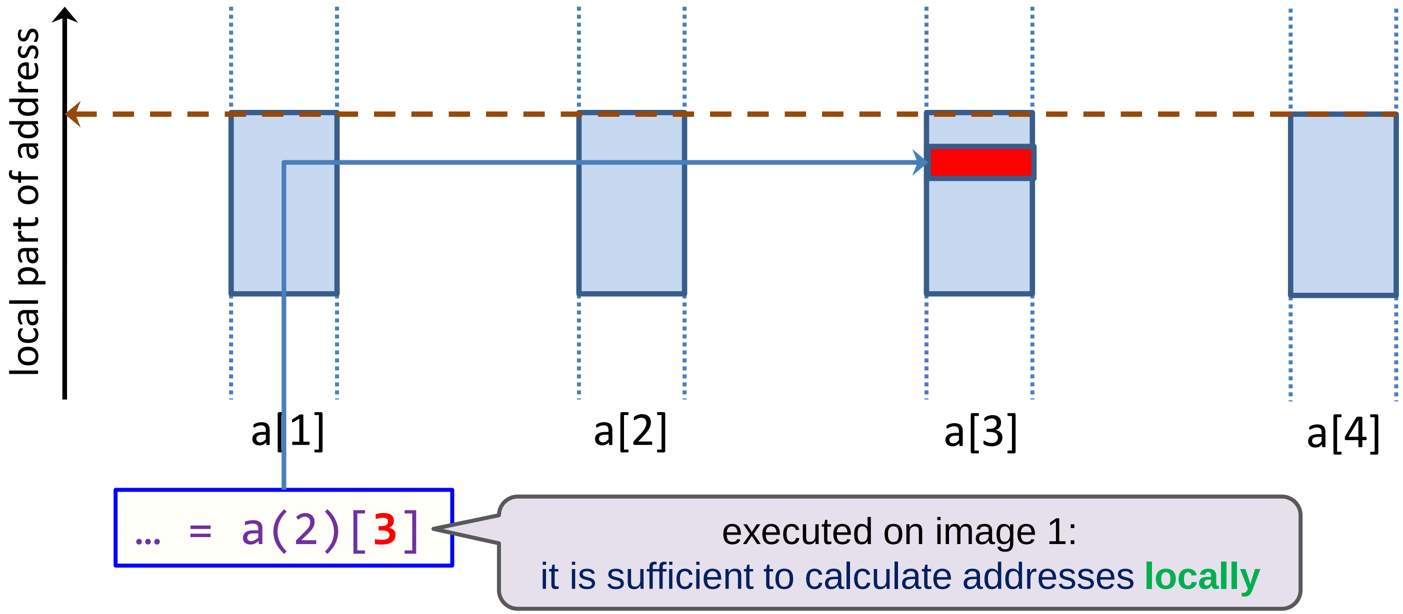
- **No segment ordering is implied by execution of a collective**
- **Collectives must be invoked by all images**
  - and from unordered segments, to avoid deadlocks

Coarrays and dynamic memory

Symmetric memory

For addressing efficiency, there is an advantage

- in using **symmetric** memory for coarrays (i.e. on each image, same local part of start address for a given object): no need to obtain a remote address for accessing remote elements



- carry this property over to dynamic memory: **symmetric heap**

Allocatable coarrays: Declaration

a coarray **cannot** have the POINTER attribute

Allocatable object

```
INTEGER, ALLOCATABLE :: id(:)[:]
TYPE(body), ALLOCATABLE :: pavement(:,:)[:,:]
```

intrinsic type

derived type, corank 2

both shape and coshape are **deferred**

Allocatable component

- part of type declaration

```
TYPE :: co_vector
  REAL, ALLOCATABLE :: v(:)[:]
END TYPE
```

component is an allocatable array

- objects of such a type must be **scalars**

```
TYPE(co_vector) :: a_co_vector
```

and are **not permitted** to have the ALLOCATABLE or POINTER attribute, or to themselves be coarrays

Allocatable coarrays: Executing the allocation

Symmetric and collective:

- the same ALLOCATE statement must be executed on **all images** in unordered segments

same bounds **and** cobounds
(as well as type and type parameters)
must be specified on all images

```
ALLOCATE (id(n)[0:*], pavement(n,10)[p,*], stat=my_stat)

ALLOCATE ( a_co_vector % v(m)[*] )
```

Semantics:

permits an implementation to make use of a symmetric heap

1. each image performs allocation of its **local** (equally large) portion of the coarray
2. if successful, all images **implicitly** synchronize against each other

subsequent references or definitions are
race-free against the allocation

Deallocation of allocatable coarrays

- **Symmetric and collective:**
  - the same DEALLOCATE statement must be executed on **all images** in unordered segments

```
DEALLOCATE ( id, pavement, a_co_vector % v )
```

  - for objects without the SAVE attribute, DEALLOCATE will be executed **implicitly** when the object's scope is left

- **Semantics:**

1. all images **implicitly** synchronize against each other
2. each image performs deallocation of its **local** portion of the coarray

> preceding references or definitions are
> race-free against the deallocation

Reallocation and moving an allocation

- **Auto-(re)allocation is not permitted for coarrays:** In

```
INTEGER, ALLOCATABLE :: id(:)[:]

id = some_other_array(:)
```

  - the LHS must already be allocated and the RHS must conform
  - this avoids potential asymmetry as well as implicit synchronization (or even deadlock)

- **The MOVE_ALLOC intrinsic** F18

  - if the FROM argument is a coarray, it must be executed on all images, and will imply synchronization of all images
  - TO must have the same corank as FROM

Coarrays as targets

- **Specification of a TARGET attribute is permitted ...**

```
INTEGER, TARGET :: id(id_dim)[*]
```

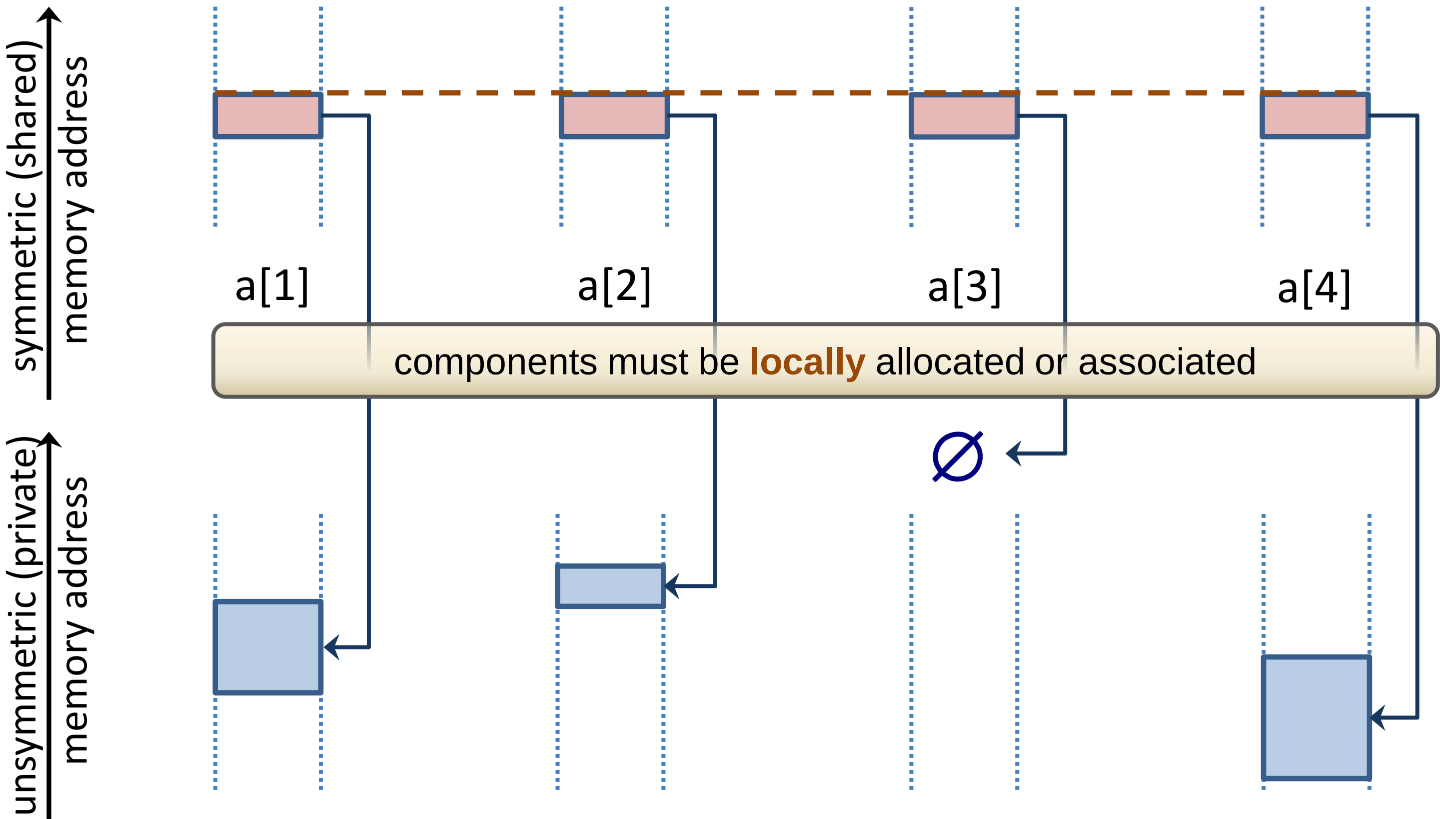
- **... but only local pointer association and referencing is possible**

```
INTEGER, POINTER :: id_ptr(:)
:
id_ptr => id(::2)        ! OK

… = id_ptr(:)[3]         ! Not permitted

id_ptr => id(:)[3]       ! Not permitted
```

Coarray objects of "container type"

Unsymmetric objects

- **Type definition contains dynamic components**
  - might have either the POINTER or the ALLOCATABLE attribute
- **A coarray object of such a type is permissible**



Example: expand an existing serial code

- **Idea:**
  - avoid modification of type and data design
  - implement necessary communication mechanism separately
- **Add a suitably constructed derived type, for example:**

```
TYPE :: communication_container
  REAL, POINTER :: data(:) => null()
  : ! possibly further components
END TYPE

TYPE(communication_container) :: subfield[*]
```

coarray of container type

Assumptions on existing code and parallel extension

- **Baseline algorithm works on** `REAL, ALLOCATABLE, TARGET :: field(:,:)`
- **In between, each image needs data from another image q:**
  - say, a row or a column from **`field`**

```
IF (use_row) THEN
  subfield % data => field(row, :)
ELSE
  subfield % data => field(:, column)
END IF
SYNC ALL
q = …
n(q) = size( subfield[q] % data, 1 )

CALL process(field, …)
SYNC ALL
local_data(:n(q)) = … + subfield[q] % data
```

`field` must be allocated

assure pointer association on q is ordered against references to q from another image

assure that updates to **`field`** on q are ordered against references to q from another image

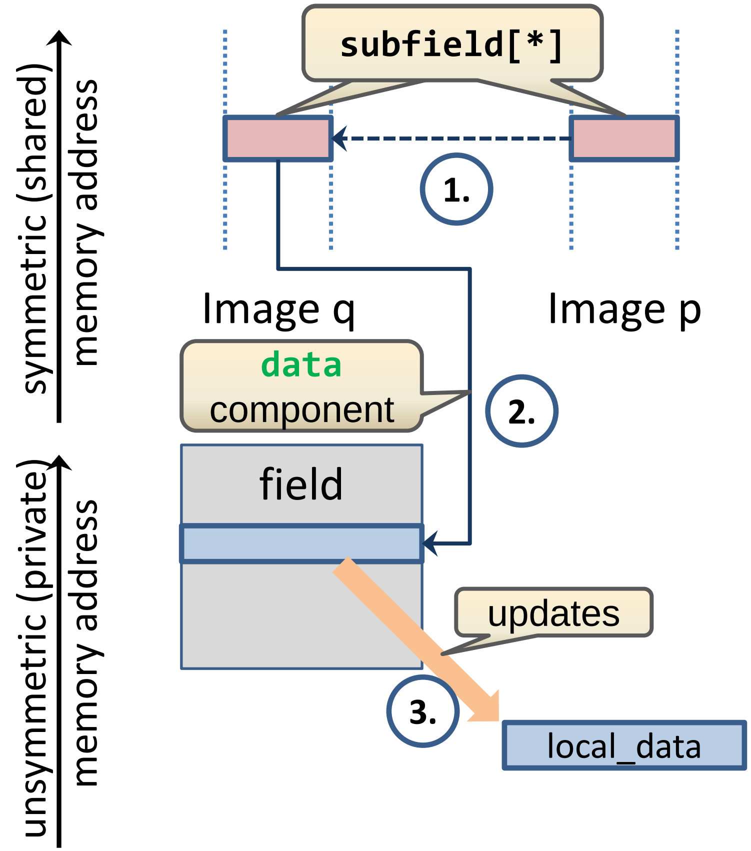
remote size must be locally known (array conformance)

look at this together with next slide

Accessing remote component data (here: „Get“)

reference to `subfield[q] % data` executed on image p

1. access remote object **`subfield[q]`** from image p
2. obtain location and size of **`data`** component
3. transfer data component to executing image



Performance impact:

- additional latency due to lookup step
- for pointers, non-contiguous access is likely to reduce performance

Some limitations on intrinsic assignment

POINTER components

- shallow copy semantics may conflict with locality requirement

on image q, **`subfield % data`** may become **undefined**

```
subfield[q] = communication_container(field(:,1))
```

```
TYPE :: polynomial
  PRIVATE
  REAL, ALLOCATABLE :: f(:)
END TYPE
```

Allocatable components

- copying of data is allowed, but **no (implied) remote** allocation

This is **not** permitted

```
TYPE(polynomial) :: ps[*]

ps[q] = polynomial( [2.0, 5.0 ] )

ps[q] % f = [2.0, 5.0 ]
```

if executed on an image other than q, **`ps % f`** must be allocated there with size 2

Coarray subobjects

- **A subobject of a coarray is also a coarray if**
  - it is not coindexed,
  - no vector subscript is involved in establishing it, and
  - no POINTER or allocatable component selection is involved in establishing it.

  Otherwise, it is not a coarray.
- **Relevance:**
  - when passing as an argument to a procedure with a coarray dummy
  - in an association block context

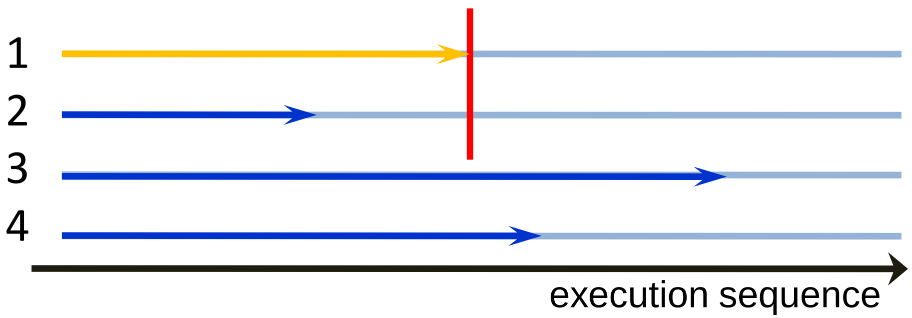
following now: Exercise session 6

Advanced Synchronization

Partial synchronization

Image subsets

- sometimes, it is sufficient to synchronize only a few images



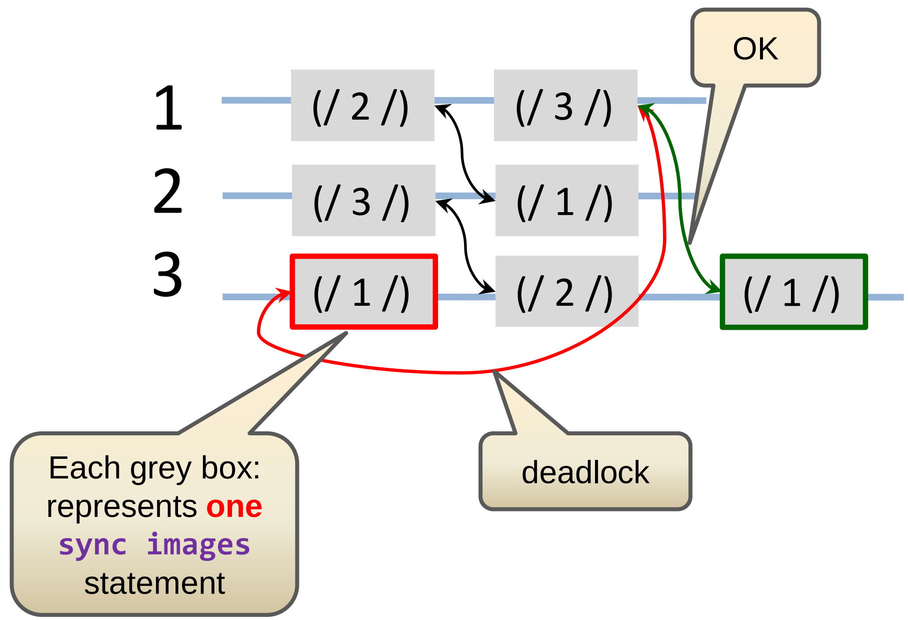
- synchronization statement:

```
IF (this_image() < 3) THEN
  SYNC IMAGES ( [ 1, 2 ] )
END IF
```

executing image is implicitly included in image set

More than 2 images:

- need not have same image set on each image
- but: eventually all image **pairs** must be resolved, else deadlock occurs



Example: Simple Master-Worker

Scenario:

- one image sets up data for computations
- others do computations

```
IF (this_image() == 1) THEN
  : ! send data
  SYNC IMAGES ( * )
ELSE
  SYNC IMAGES ( 1 )
  : ! use data
END IF
```

„all images“

images 2 etc. don‘t mind stragglers

- difference between **SYNC IMAGES (*)** and **SYNC ALL**: no need to execute from all images

Performance notes:

- sending of data by image 1

```
DO i=2, num_images()
  a(:)[i] = …
END DO
```

- "Put" mode

  an optimizing compiler might perform non-blocking transfers, and processing of data by other images might start up in a staggered sequence.

Partial synchronization: Best Practices

Localize complete set of partial synchronization statements

- **avoid** interleaved subroutine calls which do synchronization of their own

```
IF (this_image() == 1) SYNC IMAGES ( 2 )
CALL mysub(…)
:
IF (this_image() == 2) SYNC IMAGES ( 1 )
```

- a very bad idea if subprogram does the following

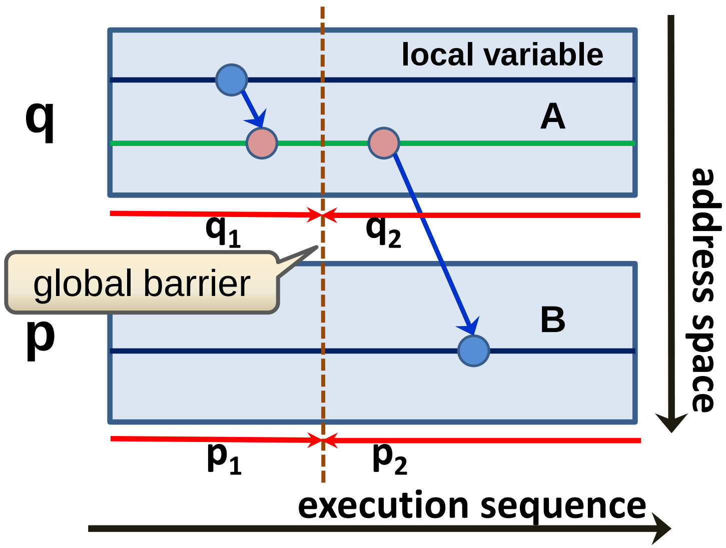
```
SUBROUTINE mysub(…)
  :
  IF (this_image() == 2) SYNC IMAGES ( 1 )
  :
END SUBROUTINE
```

`sync images` is not context-safe

- likely to produce wrong results even if no deadlock occurs

Weaknesses of previously treated synchronization constructs

- **Recall semantics of SYNC ALL**



- enforces segment ordering: q_1 before p_2, p_1 before q_2
- q_j and p_j are unordered
- applies for SYNC IMAGES as well

- **Symmetric synchronization is overkill**
  - the ordering of p_1 before q_2 is often not needed
  - image q therefore might continue without waiting
- **Therapy:**
  - F18 introduces a **lightweight, one-sided** synchronization mechanism – **Events**

```
USE, INTRINSIC :: iso_fortran_env

TYPE(event_type) :: ev[*]
```

special opaque derived type;
all its objects must be coarrays

One-sided synchronization with Events

- **Image q executes**

```
a = …
EVENT POST ( ev[p] )
```

  - and continues **without** blocking

- **Image p executes**

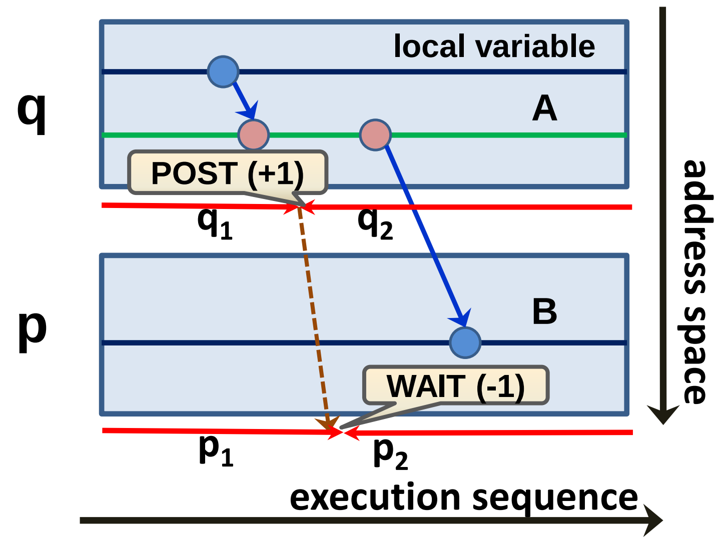
```
EVENT WAIT ( ev )
b = a(:)[q]
```

no coindex permitted on event argument here

  - the WAIT statement **blocks** until the POST has been received. Both are image control statements.

an event variable has an internal counter with default value zero; its updates are **exempt** from the segment ordering rules („atomic updates“)

- **One sided segment ordering**



  - **q_1 ordered before p_2**
  - no other ordering implied
  - no other images involved

The dangers of over-posting

- **Scenario:**
  - Image **p** executes

```
EVENT POST ( ev[q] )
```

  - Image **q** executes

```
EVENT WAIT ( ev )
```

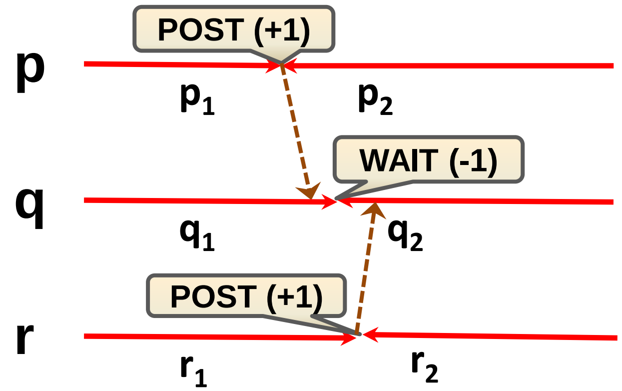
  - Image **r** executes

```
EVENT POST ( ev[q] )
```

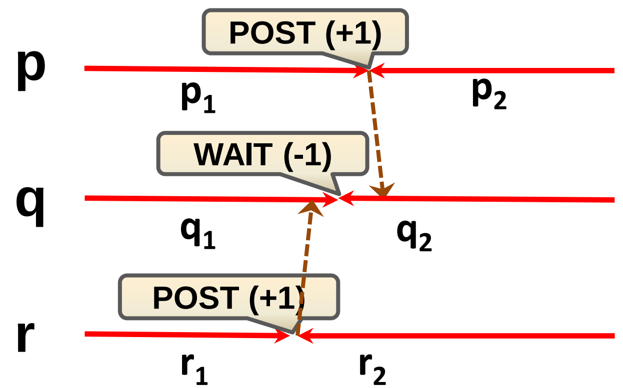
- **Question:**
  - what synchronization effect results?
- **Answer: 3 possible outcomes**
  - which one happens is **indeterminate**

Avoid over-posting from multiple images!

- **Case 1: p_1 ordered before q_2**



- **Case 2: r_1 ordered before q_2**

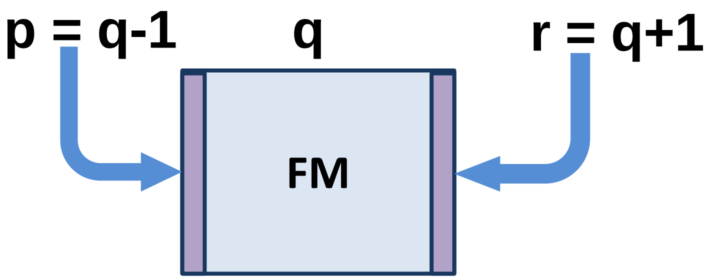


- **Case 3: ordering as given on next slide**

Multiple posting done correctly

Why multiple posting?

- **Example:** halo update



Correct execution:

- Image **p** executes

```
fm(:,1)[q] = …
EVENT POST ( ev[q] )
```

- Image **r** executes

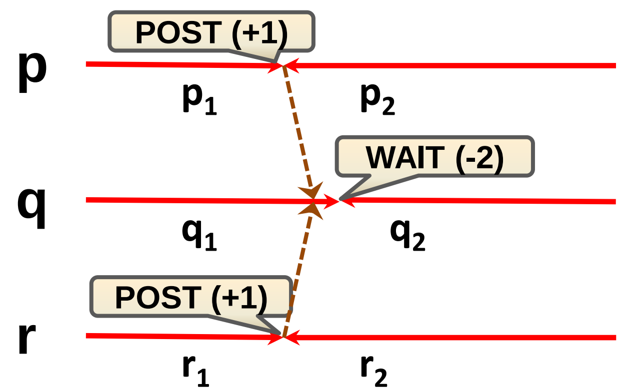
```
fm(:,n)[q] = …
EVENT POST ( ev[q] )
```

- Image **q** executes

```
EVENT WAIT ( ev, UNTIL_COUNT = 2 )
… = fm(:,:)
```

number of posts needed

p_1 and r_1 ordered before q_2



This case is enforced by using an UNTIL_COUNT

Mutual Exclusion (simplest case)

- **Critical region**
  - block of code only executed by one image at a time
  - order is indeterminate

```
CRITICAL
  : ! statements in region
END CRITICAL
```

  - can have a name, but this has no semantics associated with it

- **Subsequently executing images:**
  - segments corresponding to execution of the code block are ordered against one another
  - this does **not** apply to preceding or subsequent code blocks

  → may need additional synchronization to protect against race conditions

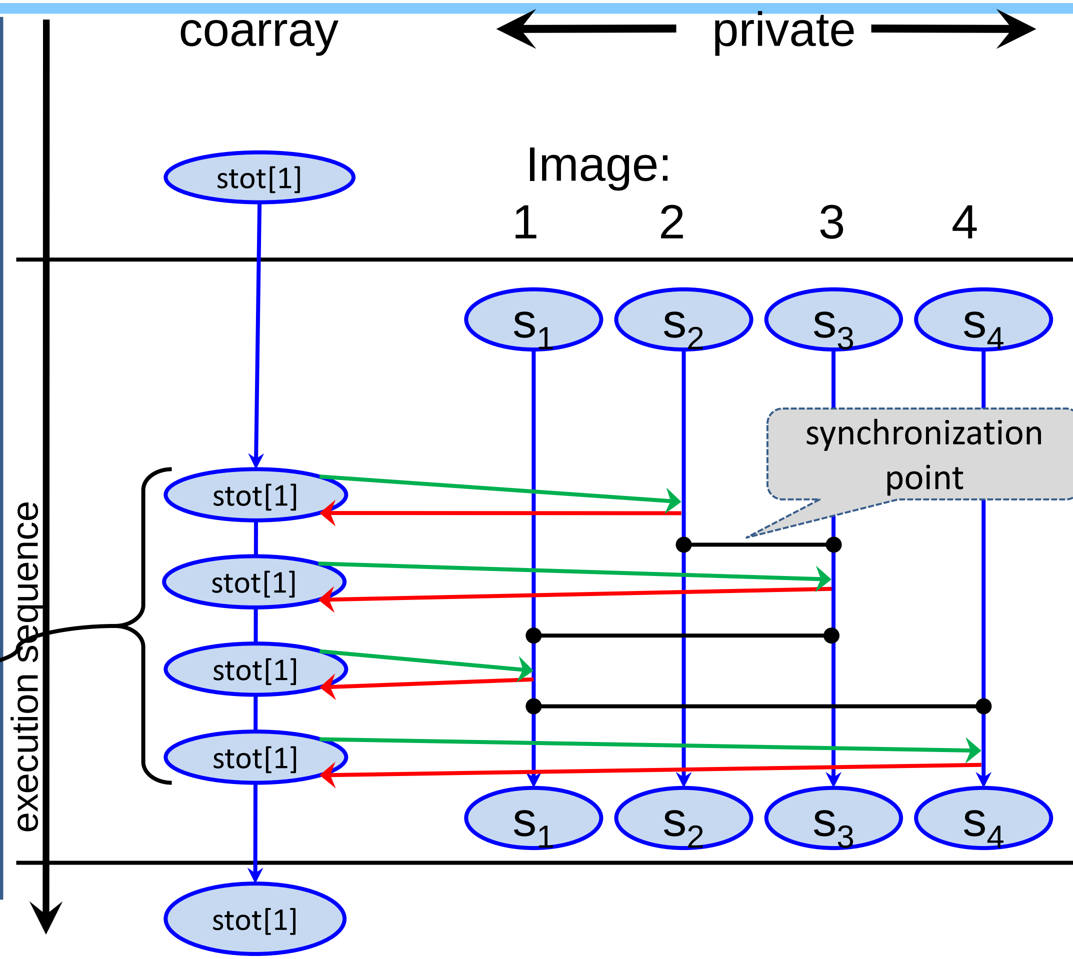
Example for mutual exclusion via a `critical` region

```
REAL :: s, stot[*]
REAL :: a(:)
INTEGER :: i
stot = 0.0
SYNC ALL
s = 0.0
DO i = 1, size(a)
  s = s + a(i)
END DO
CRITICAL
  stot[1] = stot[1] + s
END CRITICAL
SYNC ALL
… = stot[1]
```

avoid race of above assignment against first update

give all images the final value

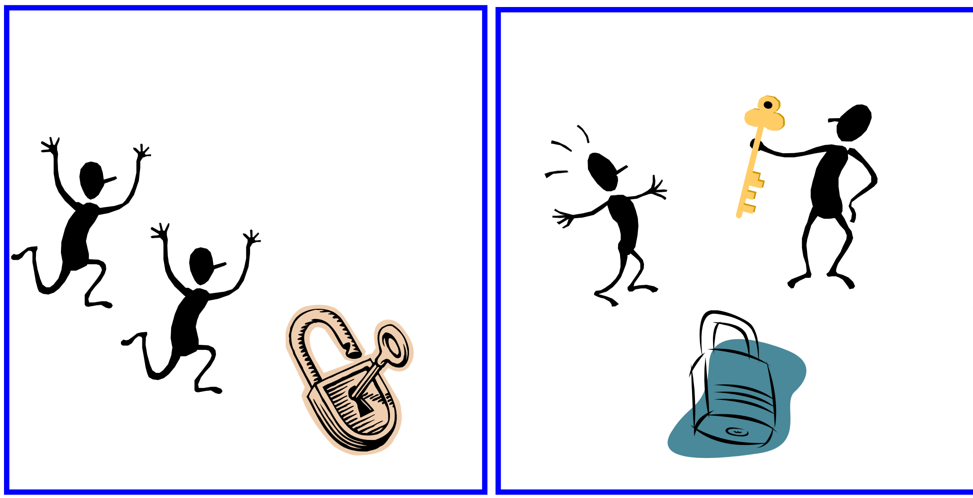
inefficient sum reduction

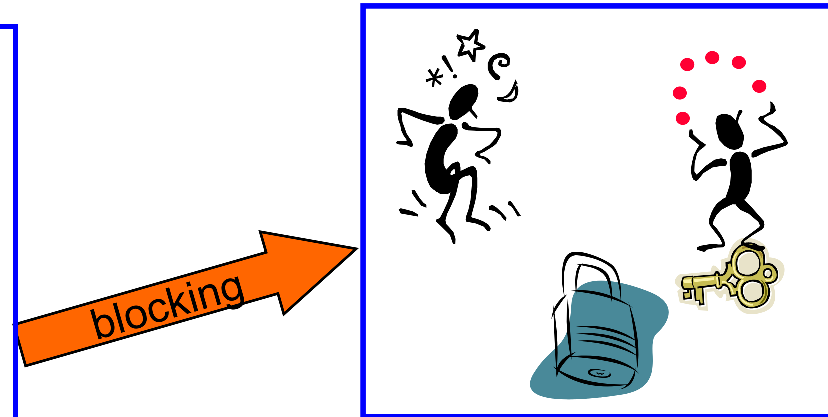


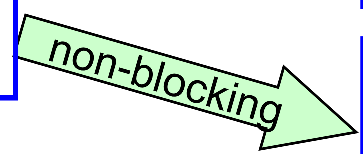
- **Only one image at a time can execute the critical region**
  - others must wait → code in region is **effectively serialized**

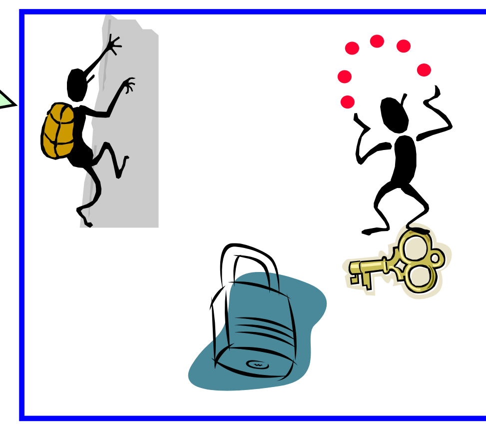
Mutual exclusion with locks

A coarray lock variable can be used to implement specifically designed synchronization mechanisms









- mutual exclusion bound to objects
  → more flexible than critical regions
- objects may require protection in multiple blocks
- different objects protected by different locks
  → improved scalability

Simplest example for lock usage

Image control statements

- LOCK and UNLOCK

```
USE, INTRINSIC :: iso_fortran_env

TYPE(lock_type) :: my_lock[*]

LOCK (my_lock[1])
: !   update shared data
: !   through coindexing
UNLOCK (my_lock[1])
```

Example works analogous to a CRITICAL region

default initialized to the "unlocked" state

blocks until the variable has the state "unlocked", then acquires the lock

must be invoked by the image that previously acquired the lock. Immediately continues after releasing the lock.

Lock variable:

- two states - unlocked or locked
- locked means: acquired by a specific image (until that image releases the lock again)

Notes:

- typically there exist as many locks as there are images, but only one is used

  Quiz: why image 1 in the example?

- segment ordering is one-way (like for events)

Non-blocking usage and performance considerations

```
USE, INTRINSIC :: iso_fortran_env

TYPE(lock_type) :: nb_lock[*]
LOGICAL :: got_it


DO
  LOCK (nb_lock[1], ACQUIRED_LOCK=got_it)
  IF (got_it) EXIT

  : ! do stuff unrelated to shared data
  : ! protected by the lock
END DO
: ! update protected shared data
UNLOCK (nb_lock[1])
```

instead of blocking, returns whether the lock was successfully acquired.

locks are an expensive synchronization mechanism

- **Best case timing for lock acquisition**

  $$T_{lock} = T_{lat} * \log_2 N$$

  where

  T_{lat} is the maximum latency in the system

  (a couple of µs → 10,000 cycles)

  N is the number of image groups for which T_{lat} applies.

Typical value for large programs: 100,000 cycles

(excludes outstanding data transfers)

The EVENT_QUERY intrinsic

F18

- **Permits to inquire the state of an event variable**

```
CALL event_query( event = ev, count = my_count )
```

- the event argument cannot be coindexed
- the current count of the event variable is returned
- the facility can be used to implement non-blocking execution on the WAIT side of event processing
- invocation has **no** synchronizing effect

Event and lock subobjects

- **Declare type components as events or locks**

```
TYPE :: queue
  TYPE(lock_type) :: lock
  TYPE(work_item) :: work
  TYPE(queue), POINTER :: &
              next => null()
END TYPE
```

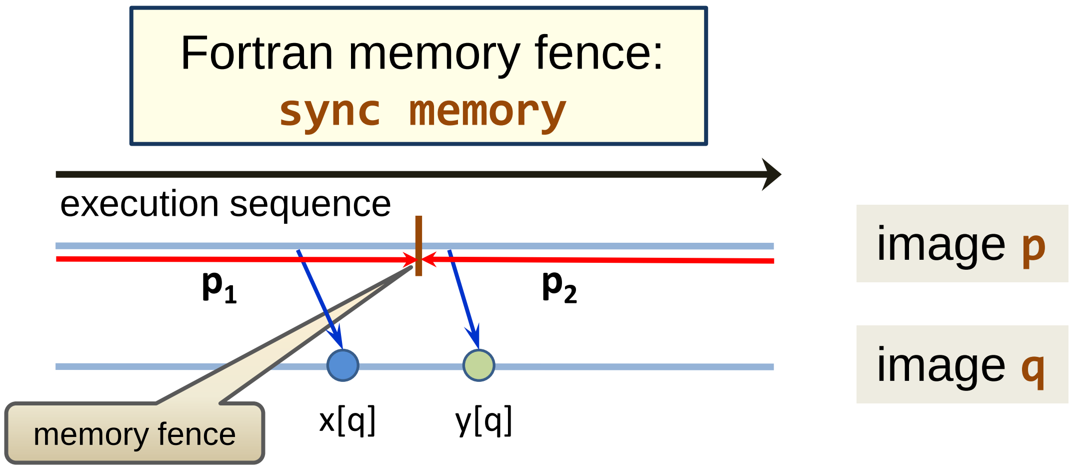
```
TYPE :: pipeline
  TYPE(event_type) :: start
  TYPE(work_item) :: work
END TYPE
```

- but then objects of that type are obliged to be coarrays:

```
TYPE(queue) :: my_queue[*]
TYPE(pipeline), ALLOCATABLE :: my_pipeline(:)[:]

TYPE(queue) :: incorrect_queue ! Not permitted
```

Memory fence

- **Target: support for user-defined synchronization**
- **Prerequisite: subdivide a segment into two segments**



Note:
A memory fence is implied by **many, but not all** of the image control statements

- **Assurance given by memory fence:**
  - operations on **`x[q]`** and **`y[q]`** via statements on **`p`**
  - action on **`x[q]`** precedes action on **`y[q]`** → code movement by compiler prohibited
  - **`p`** is subdivided into two segments
  - **but:** segment on **`q`** is unordered with respect to both segments on **`p`**

- **Exception to segment ordering rules is given for**
  - for **scalars** of **some intrinsic** datatypes

```
INTEGER(atomic_int_kind)
LOGICAL(atomic_logical_kind)
```

  - that are **only** modified via invocation of **atomic procedures**, for example those defined in the F08 standard:

```
atomic_define(atom, value)
```

atom[q] := value

```
atomic_ref(value, atom)
```

value := atom[q]

`atom` a coarray or coindexed variable

Envisioned purpose:
Permit the **experienced** programmer to implement customized synchronization mechanisms

- **Programming with race conditions:**
  - might be very fast (hardware atomics, asynchronous execution), but also is dangerous to use
  - high likelihood of producing unportable code

Example for usage and effect of atomic procedures

- **With**

```
INTEGER(atomic_int_kind) :: x[*] = 0, y[*] = 0
INTEGER :: tmp = 0
```

  - execute the following statements

on image **p**

```
CALL atomic_define( x, 1 )
SYNC MEMORY
CALL atomic_define( y, 1 )
```

optionally added

on image **q**

```
DO WHILE (tmp == 0)
  CALL atomic_ref( tmp, y[p] )
END DO
CALL atomic_ref( tmp, x[p] )
WRITE(*, *) tmp
```

- **Then the following applies:**
  - this is standard-conforming (with or without the SYNC MEMORY)
  - the result printed out may be **0 or 1** - there is **no ordering requirement** for visibility of atomic updates seen from unordered segments
  - This is even the case if the additional SYNC MEMORY statement is executed on image **p** as indicated

Programmer's responsibility

- **To evaluate all possible results of a set of atomic operations, the programmer must**
  - check **all possible interleavings** of atomic operations executed on unordered segments

    > The assumption that any issued atomic operation eventually completes is legitimate, though.

  - taking care that atomic references and definitions of **different entities** may also be unordered against each other
  - and that ordering may also **depend on the image** that observes values of variables involved in atomic operations.

Example for user-defined segment ordering (purely illustrative)

- **With**

```
INTEGER(atomic_int_kind) :: x[*] = 0
INTEGER :: tmp = 0
REAL :: a[*] = 0.0
```

  - execute the following statements

on image **p**

```
a = 2.5
SYNC MEMORY          ! end segment p1
CALL atomic_define( x, 1 )
```

on image **q**

```
DO WHILE (tmp == 0)
  call atomic_ref( tmp, x[p] )
END DO
SYNC MEMORY          ! start segment q2
WRITE(*, *) a[p]
```

- **Simple (!) state change of x:**
  - guarantees that SYNC MEMORY on p is executed before that on q
  - and therefore p_1 is ordered against q_2
  - and therefore the coindexed access to a[p] on q is conforming

> Only slightly less simple state changes can easily trip you up:
> just search for „ABA race condition“

Further atomic functions defined in F18

```
atomic_add( atom, value )
```

atom[q] := atom[q] + value (integer)

```
atomic_<and|or|xor>( atom, value )
```

atom[q] := atom[q] <op> value (logical)

```
atomic_fetch_<op>( atom, value, old )
```

incoming atom[q] assigned to OLD in addition to operation <op>, which may be any of the above

```
atomic_cas( atom, old, compare, new )
```

compare and swap (integer or logical):
old = atom[q]
if (atom[q] == compare) atom[q] = new

Example for how `atomic_add()` could be used

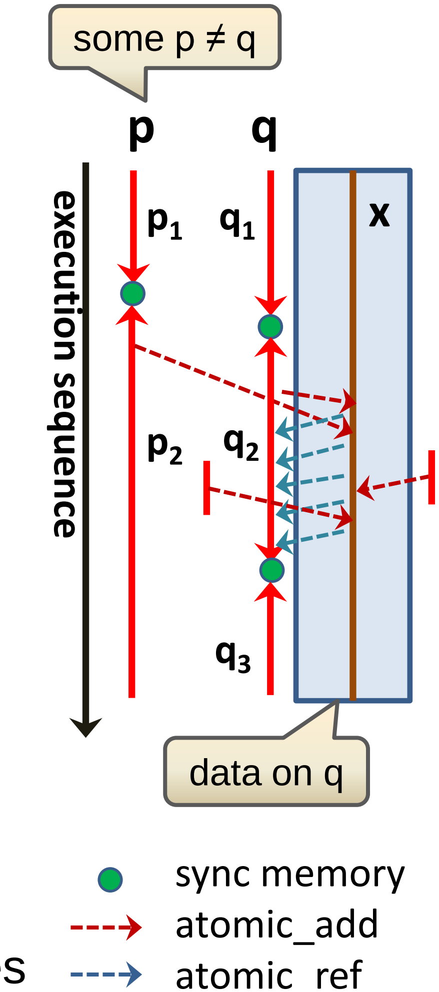
- **For synchronization involving all images**

(A)

```
INTEGER(atomic_int_kind) :: x[*] = 0, z
INTEGER :: q
q = … ! same value on each image
SYNC MEMORY
CALL atomic_add(x[q], 1)
IF (this_image() == q) THEN
  wait: DO
    CALL atomic_ref(z, x)
    IF (z == num_images()) EXIT wait
  END DO wait
  SYNC MEMORY
END IF
```

order of updates is indeterminate

guarantee exit once all images have executed **(A)**



- **Result:**
  - Segment q_3 is ordered against 1st segment of all images

Using coarrays together with object-oriented features

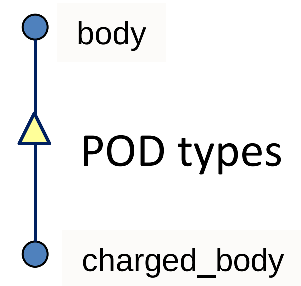
- Shaky ground due to implementation issues
- Limited semantics

Combining coarrays with object orientation

A coarray may be polymorphic

- example shows typed allocation

```
CLASS(body), ALLOCATABLE :: particles(:)[:]

ALLOCATE( charged_body :: particles(n)[*] )
```



Collective allocation and synchronization.
It must be **guaranteed** that the dynamic type is the same on each image.

- coindexing is not permitted for a polymorphic left hand side:

```
particles(:)[p] = …
```

Not permitted for intrinsic assignment

```
SELECT TYPE (particles)
TYPE IS (charged_body)
  particles(:)[p] = …
END SELECT
```

OK - **`particles`** are non-polymorphic here

note that it would need to be allocatable

- LHS coarray in intrinsic assignment cannot be polymorphic

Restrictions on association

- **Coindexing is not permitted:**

```
TYPE(body) :: asteroids(ndim)[*]
```

Not permitted

```
SELECT TYPE(particles[2])
   :
END SELECT
```

```
ASSOCIATE(p => asteroids[2])
   p = …
END ASSOCIATE
```

appears as local

- **But appearance of a coarray is OK**
  - we've already seen it for SELECT TYPE
  - here an example for coarray subobject association:

```
ASSOCIATE(p => asteroids%mass)
   p(:)[q] = …
END ASSOCIATE
```

p is a discontiguous real array coarray, because **asteroids%mass** is a coarray subobject.

Limitation on type extension

- **Applies for types with coarray components:**

```
TYPE, EXTENDS( co_m ) :: co_mv
  REAL, ALLOCATABLE :: v(:)[:]
END TYPE
```

  - is only permitted if the parent type already has a coarray component:

```
TYPE :: co_m
  REAL, ALLOCATABLE :: m(:,:)[:]
END TYPE
```

  - otherwise, existing code for **`co_m`** would stop working for the extension → violation of inheritance mechanism

Execution of type- and object-bound procedures

```fortran
TYPE :: body
  : ! data components
  PROCEDURE(p), POINTER :: print
contains
  PROCEDURE :: dp
END TYPE

SUBROUTINE dp(this, kick)
  CLASS(body), INTENT(INOUT) :: this
  REAL, INTENT(IN) :: kick(3)
  : ! give body a kick
END SUBROUTINE
```

object-bound procedure (pointer)

type-bound procedure (TBP)

```fortran
CALL particles(7) % dp(kick)
CALL particles(8) % print()

IF (this_image() == 1) THEN
  SELECT TYPE(particles)
  TYPE IS (charged_body)
    CALL particles(7)[2] % print()
    CALL particles(8)[2] % dp(kick)
  END SELECT
END IF
```

coindexed actual arguments to be discussed

- **Discussed:**
  - local vs. coindexed execution

- procedure pointer: remote alias is not locally known, no remote execution supported
- type-bound procedure is the same on all images
- polymorphism removed via SELECT TYPE (RTTI)

Restrictions for container types with polymorphic components

- **For explicit references to such components,**
  - coindexing is not permitted.
- **A cooperative circumlocution is required, for example:**

```
TYPE :: trajectory
  CLASS(body), ALLOCATABLE :: &
               particle(:)
  INTEGER :: nsize
END TYPE

TYPE(trajectory) :: mytr[*]
CLASS(body), ALLOCATABLE :: &
               auxiliary(:)[:]

ALLOCATE(charged_body :: &
      mytr%particle(n) )
mytr%nsize = n
: ! supply data
```

assuming the same dynamic type on all images

```
ALLOCATE( charged_body :: &
              auxiliary(nmax)[*] )
p = … ! target image
SELECT TYPE (auxiliary)
TYPE IS (charged_body)
  auxiliary(1:mytr[p]%nsize)[p] = &
                  mytr % particle
  : ! further code elided
END SELECT

SYNC IMAGES ([p,q])

: ! consume local portion
  ! of auxiliary(:)
```

assuming one-to-one mapping between source and target images

Comments on parallel library design

Coarrays as dummy arguments

explicit interface required

- **Library codes may need**
  - to communicate and synchronize argument data
  - declare these as coarrays

```
SUBROUTINE co_sub(n,w,x,y)
  INTEGER :: n
  REAL :: w(n)[*]        explicit shape
  REAL :: x(n,*)[*]      assumed size
  REAL :: y(:,:)[*]      assumed shape
  :
END SUBROUTINE
```

- **Restrictions** that **prevent** copy-in/out of coarray data:
  - if dummy is not assumed-shape, actual must be simply contiguous or have the **`CONTIGUOUS`** attribute
  - **`VALUE`** attribute prohibited for dummy argument

- **Invocation:**
  - actual argument **must** be a coarray if the dummy is

```
REAL :: a(ndim)[*], b(ndim,2)[*]
REAL, ALLOCATABLE :: c(:,:,:)[:]
ALLOCATE(c(10,20,30)[*])

CALL co_sub( ndim, a, b, c(1,:,:) )
```

  - argument **c**: for an assumed shape dummy, the actual may be discontiguous

Image-dependent coarray passing

Example Procedure:

here, dummy is a scalar coarray

```
SUBROUTINE add_pivot(x, img, y, n)
  INTEGER, INTENT(IN) :: img, n
  REAL, INTENT(IN) :: x[*]
  REAL, INTENT(INOUT) :: y(:)

  y(n) = y(n) + x[img]
END SUBROUTINE
```

Invocation

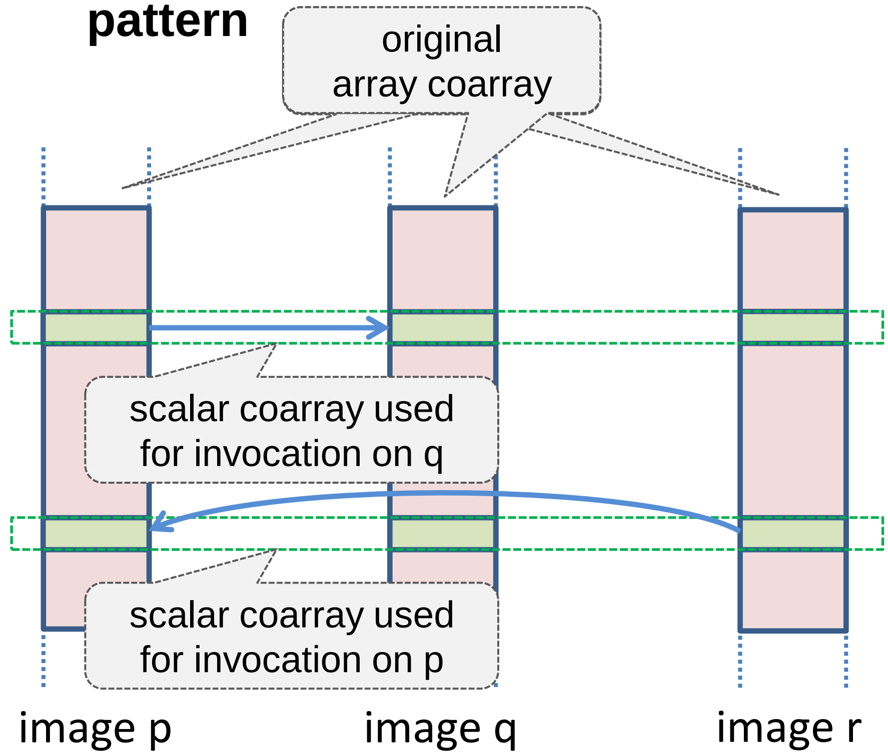
- with a different coarray (subobject) on each image

```
REAL :: x(ndim)[*]
INTEGER :: p, n
p = …; n = …
x(:) = …
SYNC ALL
CALL add_pivot( x(n), p, x, n )
```

p /= this_image(), n and **p** are different on each image

actual is a scalar coarray subobject

Illustrating the communication pattern



- all references and definitions are done „in-place“, on elements of the original array coarray
- not all images need to call the procedure

Limitations for execution inside PURE procedures

- **Coindexed definitions („Put“) are not permitted**
  - because this constitutes a side effect
  - coindexed references („Get“) are OK though
- **Image control statements are not permitted**
- **ELEMENTAL procedures**
  - are not permitted to have coarray dummy arguments

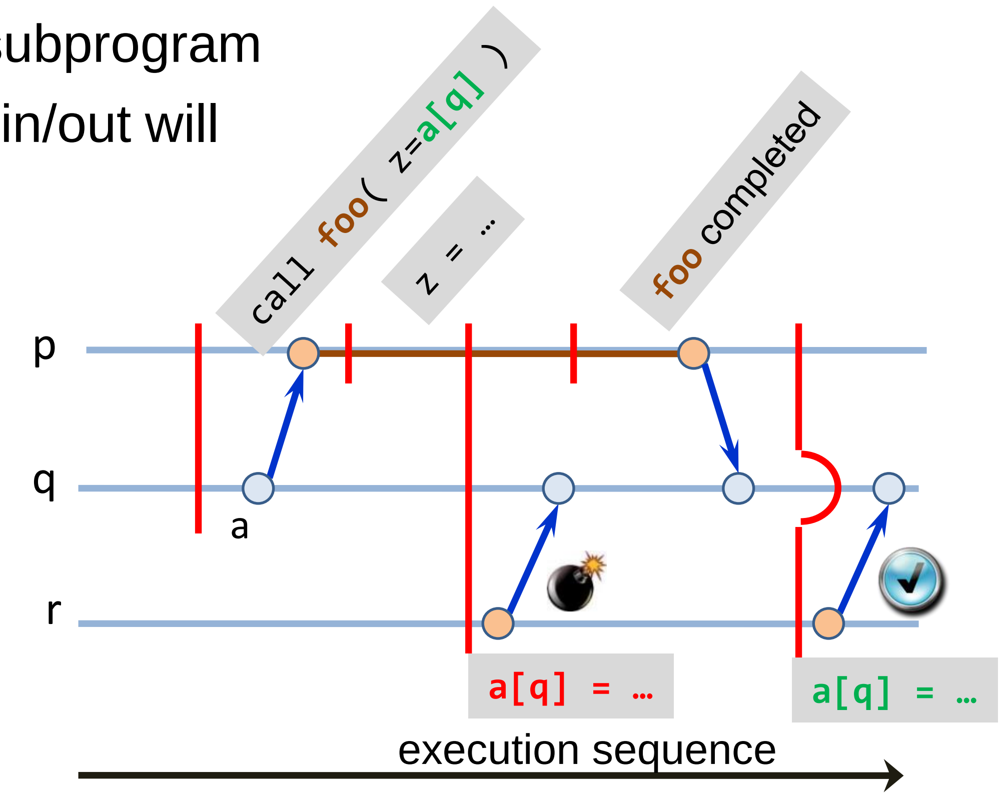
Procedure-local coarrays

- **Requirements:**
  - must have the SAVE or the ALLOCATABLE attribute or both
  - a function result cannot be declared a coarray
- **Consequence:**
  - automatic coarrays or coarray function results are not permitted
- **Rationale:**
  - not prohibiting this would imply a need for **implicit** synchronization of (and hence also invocation from) all images
  - Note that for an allocatable procedure-local coarray this is the case anyway, but the synchronization point is **explicitly** visible! If that coarray does not also have the SAVE attribute, it will be auto-deallocated at exit from the procedure if no explicit DEALLOCATE was previously issued.

Coindexed actual arguments

- **Assumptions:**
  - actual argument is a coindexed object (therefore not a coarray)
  - it is modified inside the subprogram
  - therefore, typically copy-in/out will be required

→ **an additional synchronization rule is needed**



- **Usually not a good idea**
  - performance issues
  - problematic or impermissible for container types (effective assignment!)

Factory procedures for coarrays

Allocatable dummy argument is a coarray:

```
SUBROUTINE read_coarray_data( simulation_field, file_name )
  REAL, ALLOCATABLE, INTENT(INOUT) :: simulation_field(:,:,:)[:]
  CHARACTER(LEN=*), INTENT(IN) :: file_name
  : ! determine size
  IF (allocated( simulation_field )) DEALLOCATE( simulation_field )
  ALLOCATE( simulation_field(n1, n2, n3)[0:*] )
  : ! read data
END SUBROUTINE read_coarray_data
```

- **intent(out)** is not permitted (would imply synchronization)
- actual argument: must be allocatable, with matching type, rank **and corank**
- procedure must be executed on **all images**, and with the **same** effective argument

Overloading the assignment

- **Use this as circumlocution in cases where intrinsic assignment is prohibited**
  - Example: polymorphic coarray

```
MODULE mod_body
  : ! type definition etc
  INTERFACE ASSIGNMENT (=)
    MODULE PROCEDURE assign_body
  END INTERFACE
CONTAINS
  SUBROUTINE assign_body(out, in)
    CLASS(body), INTENT(INOUT), ALLOCATABLE :: out(:)[:]
    CLASS(body), INTENT(IN) :: in(:)
    : ! assert that size of in is the same on all images
    ALLOCATE(out(size(in,1))[*], SOURCE = in)
  END SUBROUTINE
  :
END MODULE
```

```
USE mod_body
TYPE(charged_body) :: nuclei(ndim)
CLASS(charged_body), &
       ALLOCATABLE :: conuc(:)[:]

conuc = nuclei
```

could also be a coarray

RHS might also be a function call

- **Generic resolution of coarray vs. noncoarray specific is not possible** (syntax identical for calls with / without coarray)

Using type-bound procedures to implement communication

Example:

- handle data transfer for the container type

```
TYPE :: polynomial
  REAL, ALLOCATABLE :: f(:)
CONTAINS
  PROCEDURE :: get, put
END TYPE
```

- here we only look at **put**

remember that
s[p] = …
is not permitted for an
s of type polynomial

comm_success and **comm_fail**
are distinct integer constants

```
TYPE(polynomial) :: s[*]
INTEGER :: status[*]
```

Execution

- of **put** on image **p**

```
SYNC ALL
:
s = …
status[q] = s%put(q)
EVENT POST (ev[q])
```

- of code consuming **s** on image **q**

```
s = …   ! or  … = s
SYNC ALL
:
EVENT WAIT (ev)
IF ( status == comm_success ) THEN
  : ! reference local part of s
END IF
```

Implementation sketch

```
INTEGER FUNCTION put(this, img)
  CLASS(polynomial), INTENT(INOUT) :: this[*]
  INTEGER, INTENT(IN) :: img
  INTEGER :: rem_size, lb, ub
  IF ( .NOT. allocated( this[img]%f ) .AND. allocated( this%f ) ) THEN
     put = comm_fail
     RETURN
  END IF
  rem_size = size( this[img]%f,1 )
  IF ( rem_size >= size( this%f ) ) THEN
     lb = lbound( this[img]%f,1 ); ub = lb + size( this%f,1 ) - 1
     this[img]%f(lb:ub) = this%f
     this[img]%f(ub+1:) = 0.0
     put = comm_success
  ELSE
     put = comm_fail
  END IF
END FUNCTION
```

failure is will occur if component on target image

- is not allocated
- is allocated, but too small to hold data

- **For support of type extensions writing an overriding TBP is most appropriate**

Documenting the synchronization behaviour

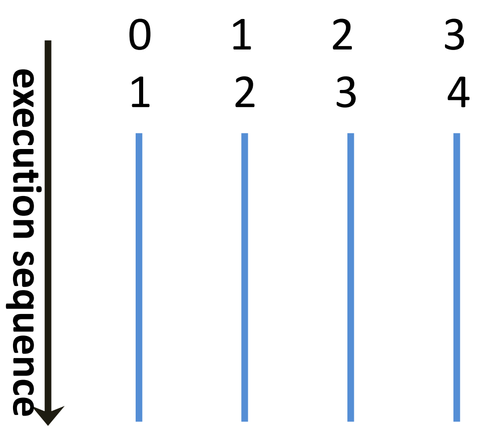
- **Synchronization performed by library code**
  - is part of its semantics and should be **documented**
- **In particular,**
  - whether (and which) additional synchronization is required by **the user** of a library,
  - and whether a procedure needs to be called from all images („collectively“) or can be called from image subsets
- **It may be a good idea**
  - to supply optional arguments that permit to change the default synchronization behaviour

following now: Exercise session 7

Interoperation with MPI

Basic execution model

- **Nothing is formally standardized**
- **Existing practice:**
  - each MPI task is identical with a coarray image



result of calling `MPI_Comm_rank()`

`this_image()`

no guarantee on ordering, though

obtained from call to `MPI_Initialized()`

```
PROGRAM with_mpi
  USE mpi_f08
  : ! further declarations, including coarrays
  IF (.not. initialized) CALL MPI_Init()
  : ! code with both MPI calls and
  : ! coarray communication / synchronization
  CALL MPI_Finalize()
END PROGRAM
```

implementation may either want this or not like this

Program design ideas

- **Do not rewrite an existing MPI code base**
- **Instead, extend it with coarray functionality**
  - to avoid deadlocks, keep MPI synchronizations separate from coarray synchronizations
  - avoid coindexed actual arguments in MPI calls
  - coarrays can be used in MPI calls (always considering segment ordering rules), but be careful with non-blocking MPI calls
  - it is probably a good idea to avoid using the same object in both MPI and coarray atomics
- **Knowledge of communication structure is required**
  - analysis with tracing tool may be needed

Technical details

- **Compilation**
  - use mpifort/mpif90 wrapper together with switch for coarray activation
  - not every MPI implementation might be usable:

    if the compiler uses MPI as implementation layer for coarrays, it is likely that you'll need to use at least a binary compatible MPI together with it

- **Execution**
  - at least for distributed-memory, it is likely that you will need to use mpiexec to start up
  - consult your vendor's or computing centre's documentation
  - facilities for pinning of MPI tasks are likely to be useful for coarray performance as well ☺

F18

Composing coarray programs: The concept of Teams

Weaknesses of flat coarray model (1)

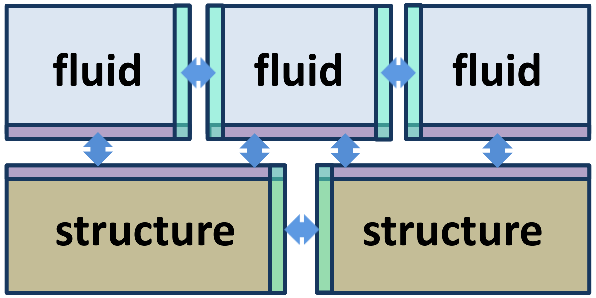
- **Development of parallel library code**

by independent programmer teams

typically each doing its own internal synchronization
maybe doing internal coarray allocation/deallocation

  - coarrays are symmetric → memory management not flexible enough
  - avoid deadlocks → obliged to do library call from **all** images
  - collectives, global barriers must be executed from **all** images
  - management of image subsets can become a headache

- **MPMD scenario: coupling of domain-specific simulation codes**
  - we‘ll look at a pseudo-application of this type to illustrate the new semantics

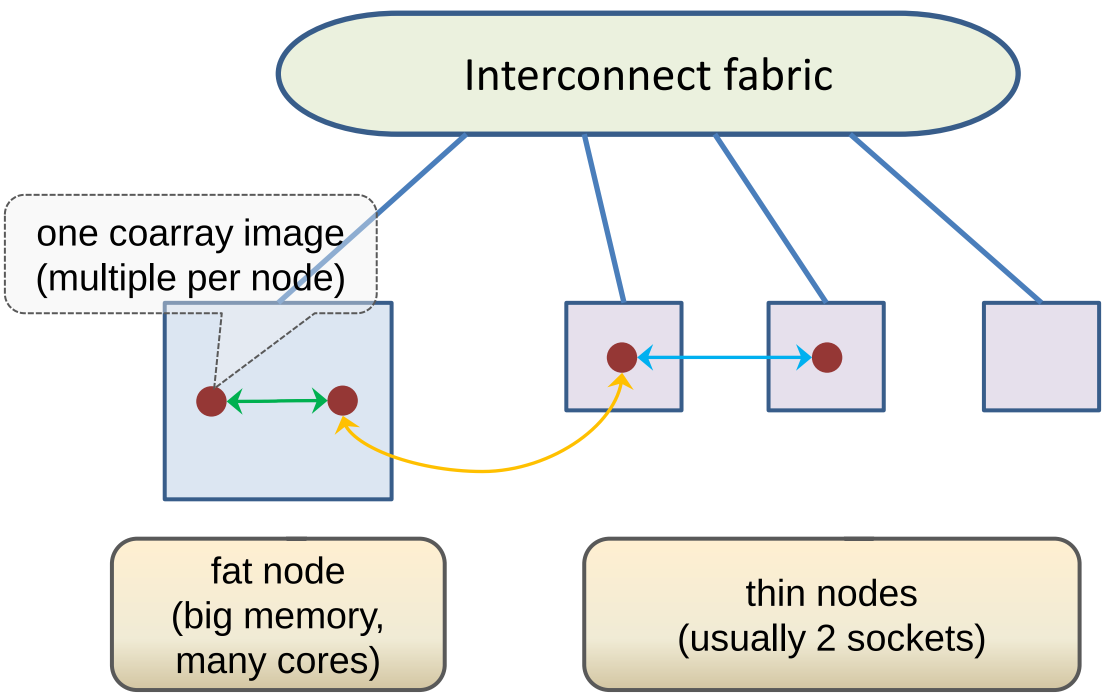


data distribution strategy:
workload balance and
memory requirements

Weaknesses of flat coarray model (2)

Matching execution to hardware

- future systems likely are non-homogeneous (memory, core count)
- A **unified hybrid** programming model is desired → want to fully exploit high internal bandwidth and fast synchronization of node architecture via independent image subsets



per image pair

Connection	typical comm BW (GB/s)	typical sync Latency (ns)
	5-50	50-250
	1-10	800-5000
	0.5-5	800-5000

- F18 **defines the concept of a team of images**

- **Teams provide additional syntax and semantics to**
  - subdivide set of images into **subsets** that can **independently** execute, allocate/deallocate coarrays, communicate, and synchronize;
  - repeated (i.e., recursive and/or nested) subsetting is also permitted.

- **Two essential steps:**
  1. define the subsets
  2. change the execution context to a particular subset (and back again)

„composable parallelism“

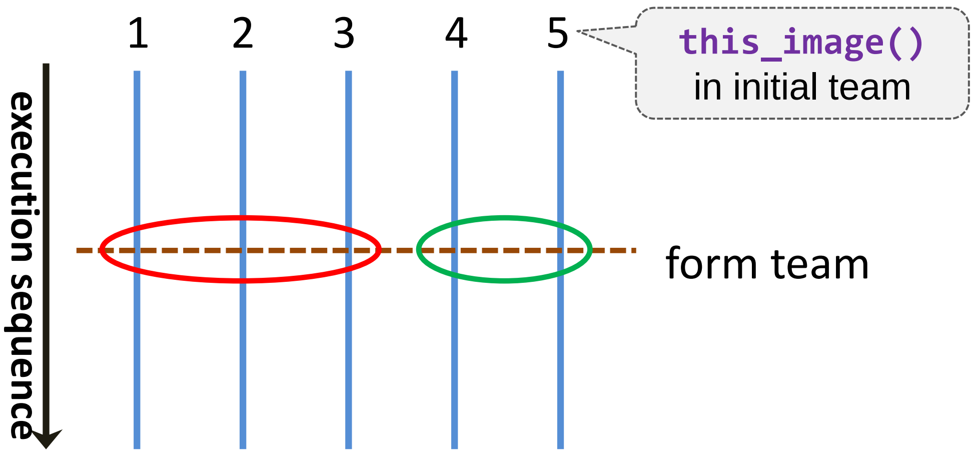
- **Breaking composability where necessary**
  - cross-team communication is also supported – as usual, with clear **visual indication** to the programmer

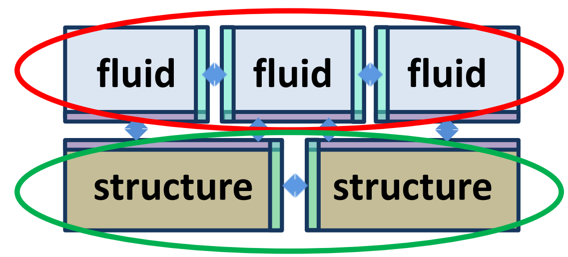
Setting up a team decomposition

FORM TEAM statement

here: the initial team

- must be executed on all images of the current team
- all images of that team are synchronized






```
FORM TEAM ( id, team [, NEW_IMAGE=…] )
```

integer supplies „color“

resulting team of opaque type **team_type**

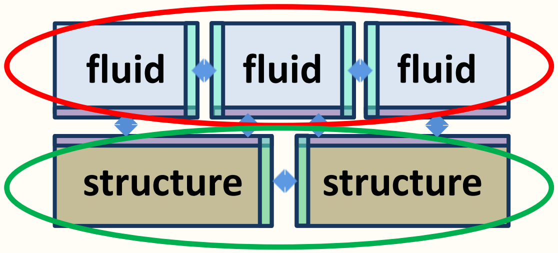
option for programmer-defined image indexing inside new teams

Example code

```
PROGRAM coupled_systems
  USE, INTRINSIC :: iso_fortran_env
  IMPLICIT NONE
  INTEGER, PARAMETER :: fluid = 1, structure = 2
  INTEGER :: nf, id
  TYPE(team_type) :: coupling_teams
  :
  nf = …
  IF ( this_image() <= nf ) THEN
    id = fluid
  ELSE
    id = structure
  END IF

  FORM TEAM ( id, coupling_teams )
  :
END PROGRAM
```

declares the type **team_type**

further declarations

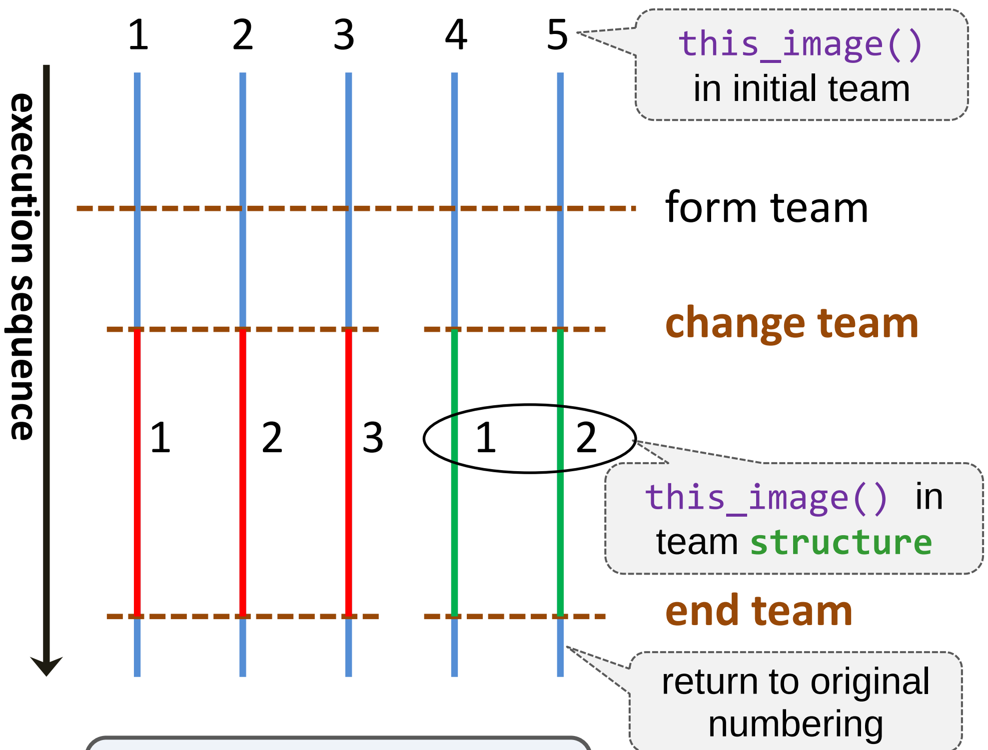


two teams are formed

further executable statements

- **FORM TEAM does not by itself split execution**
  - after the statement, regular execution continues on all images

Switching the execution context: The CHANGE TEAM block construct



fluid and **structure** are **sibling** teams

Properties:

- at beginning, changes **current team** to become the one the executing image belongs to

  sets up an **ancestor relationship** between previous and new team

- at end of block, reverts to execution as ancestor team
- team-wide synchronization of images of each team at beginning and end of each block
- **programmer is responsible** for setting up appropriate control flow inside the block

Inside the new context:

- wherever we had “all images”, now understand this as “all images of the current team”

Syntax for CHANGE TEAM

```
CHANGE TEAM( coupling_teams )
  SELECT CASE( team_number() )
  CASE( fluid )
    :
  CASE( structure )
    :
  END SELECT
END TEAM
```

only executed by members of **fluid**

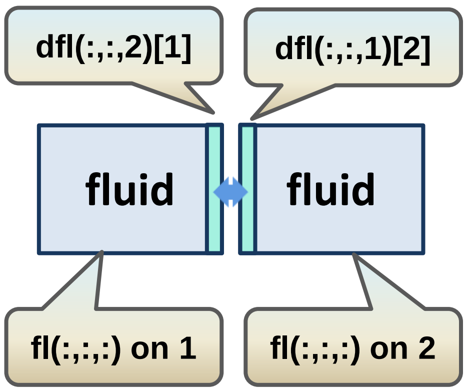
only executed by members of **structure**

- **Code fragment left**
  - indicates how control flow is implemented
  - information stored in **coupling_teams** determines which team the executing image belongs to

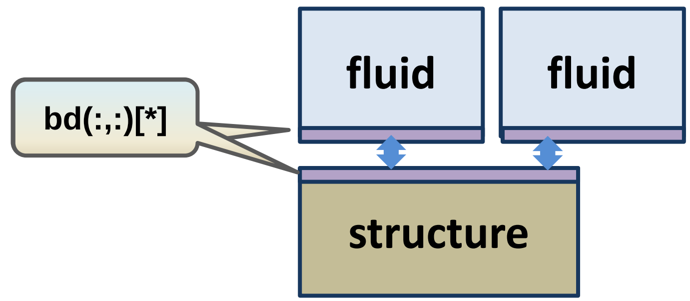
Supporting intrinsics (team identification)	
TEAM_NUMBER([TEAM]) variable of **type(team_type)**	returns an **integer** with the identifier ("color") of the specified (default: the current) team, -1 if the current team is the initial team.
GET_TEAM([LEVEL]) constants are defined in **iso_fortran_env**	returns the **type(team_type)** team value for the current team if LEVEL is not specified, or the team value corresponding to the integer constants INITIAL_TEAM, PARENT_TEAM, or CURRENT_TEAM.

Simulation data (representing physical observables)

- **Fluid:**
  - field **`fl`**, boundary **`dfl`**



- **Structure:**
  - field **`st`**, boundary **`dst`**
- **Interaction Fluid-Structure:**
  - boundary **bd**



  - applies for **both** teams!
- **Declarations:**

```
REAL, ALLOCATABLE :: fl(:,:,:), dfl(:,:,:)[:]
REAL, ALLOCATABLE :: st(:,:,:), dst(:,:,:)[:]
REAL, ALLOCATABLE :: bd(:,:)[:]
```

Processing of each subsystem

```
ALLOCATE( bd(…)[*] )
simulation : CHANGE TEAM ( coupling_teams )
  SELECT CASE( team_number() )
  CASE( fluid )
     ALLOCATE( fl(…), dfl(…)[*] )
     DO
       CALL process_fluid(fl, dfl, …)
       :
       IF ( completed ) EXIT simulation
     END DO
  CASE( structure )
     ALLOCATE( st(…), dst(…)[*] )
     DO
       CALL process_structure(st, dst, …)
       :
       IF ( completed ) EXIT simulation
     END DO
  END SELECT
 END TEAM simulation
```

data established in initial team

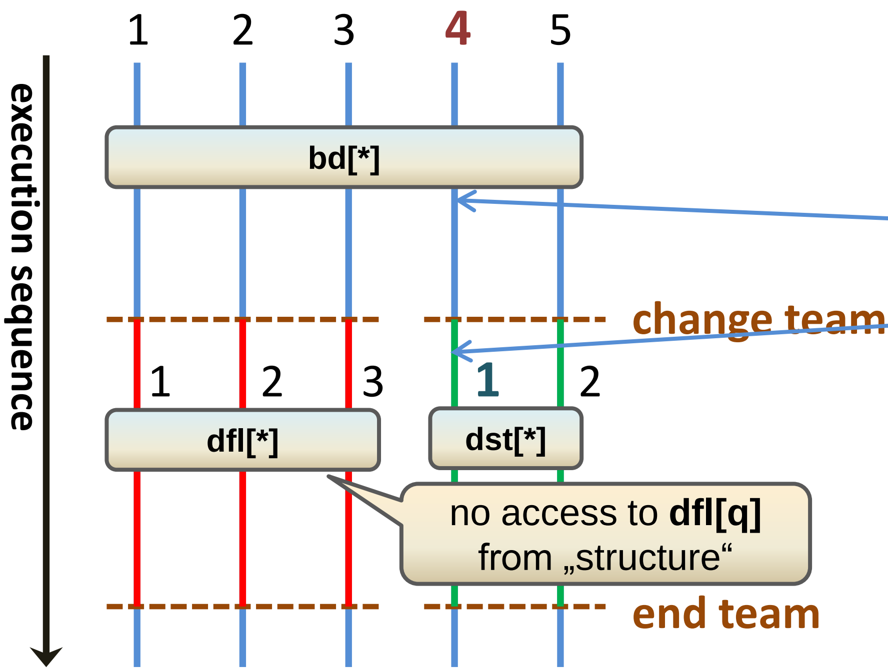
data only established in team fluid

fluid-structure interactions etc. (see later slide)

data only established in team structure

auto-deallocation of team-allocated coarrays is done here

Data distribution and access



- applies to team-specific coarrays as well as to pre-established coarrays
  → what is **bd[4]** in the initial team becomes **bd[1]** when the CHANGE TEAM starts executing
  → team-local coindexing preserves composability ☺

Interaction fluid ↔ structure

- need to communicate **across** team boundaries **without** leaving the team execution context (otherwise allocated data vanish …)

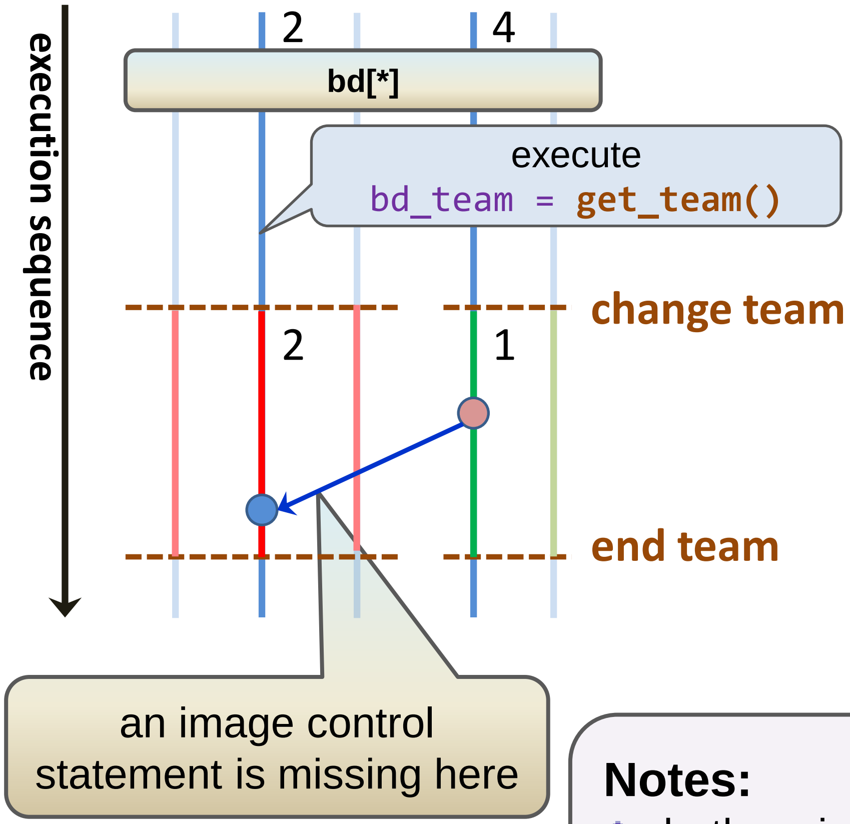
Special syntax required

- for cross-team accesses

Coindexing semantics:

- coindices are evaluated relative to the **new image index** (which is processor dependent unless NEW_IMAGE is specified in FORM TEAM)

Extending the image selector:
Cross-team coarray references and definitions



- **Example:**
  - statements below are executed on image 2 of the **fluid** team
  - sibling team syntax:

```
… = bd(:,:)[ 1, TEAM_NUMBER=structure ]
```

  - ancestor team syntax:

```
… = bd(:,:)[ 4, TEAM=bd_team ]
```

Notes:
- both variants yield the same result in this situation
- which to use depends on the image‘s knowledge of image indices and teams, and on the data assignment strategies.

Team synchronization and memory management

- **Synchronization of all images of a team**

```
SYNC TEAM ( my_team )
```

  - for example, synchronize all images of the parent team while executing in the descendant team context
  - contrast to **`SYNC ALL`**, which applies to the current team
- **Restrictions on coarray allocation and deallocation:**
  - coarrays cannot have „holes" → in the current team, it is not permitted to deallocate a coarray that has been allocated in an ancestor team
  - avoid appearance of overlapping coarrays → all coarrays allocated while a **`CHANGE TEAM`** block is executing are deallocated at the latest when the corresponding **`END TEAM`** statement is reached (even if they have the SAVE attribute)

Dealing with the fluid-structure interaction (including necessary synchronization)

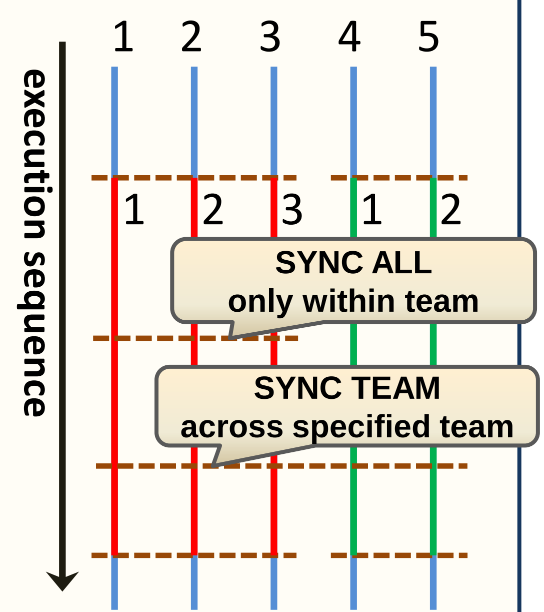
```
TYPE(team_type) :: bd_team
:
bd_team = get_team()
simulation : CHANGE TEAM (coupling_teams)
  select case( team_number() )
  CASE( fluid )
    ALLOCATE( … )
    DO
     :
     IF (interact) then
        DO i=1, nimg
          bd(…)[img(i),TEAM_NUMBER=structure] = fl(…)
        END DO
        SYNC TEAM (bd_team)
        CALL process_interaction(bd, fl, …)
      END IF
    END DO
  CASE( structure )
    :
  END SELECT
END TEAM simulation
```

team-local processing shown earlier

Array indexing and possibly needed interpolation are glossed over below

against update of local **bd** by `structure`

pushes data to `fluid` and also needs to execute `sync team`



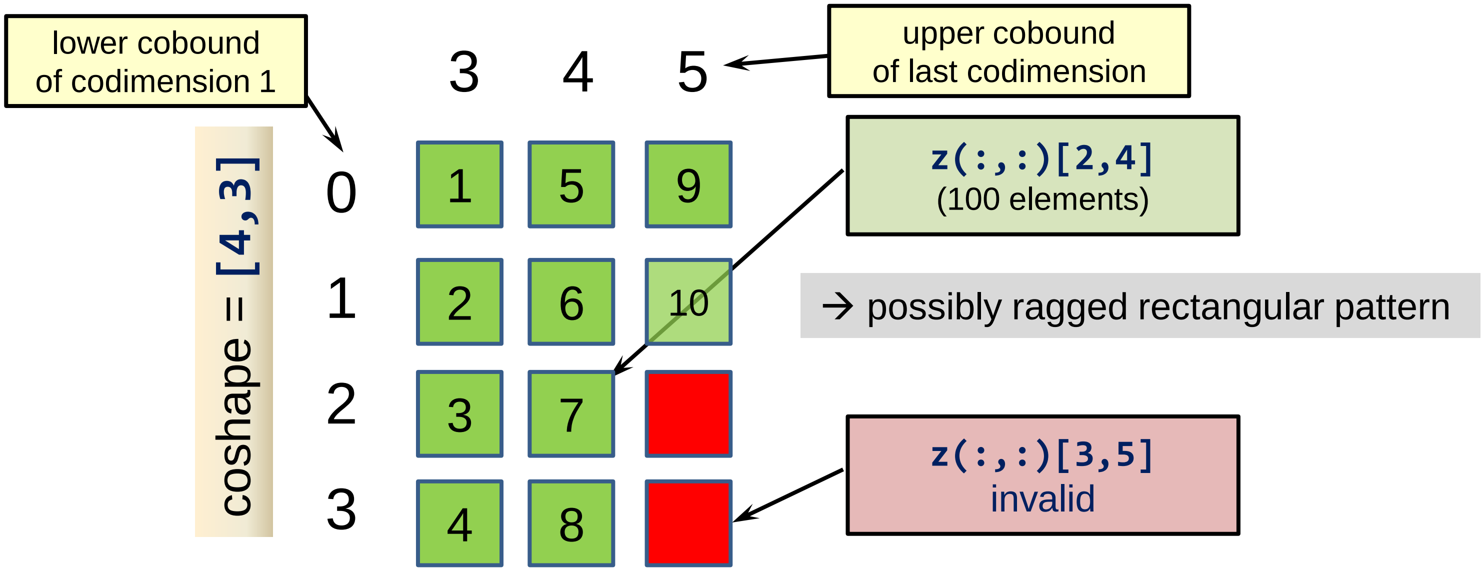
Non-default topologies
and
coindexing rules

Non-trivial coindex-to-image mappings

- **Corank of a coarray may be larger than one**
  - sum of rank and corank can be up to 15
- **Lower cobound for each codimension can be different from 1**
- **Example: corank 2**

```
REAL :: z(10,10)[0:3,3:*]
```

- **Mapping to image index for 10 executing images**



Supporting intrinsics (1)

Programmer's responsibility to specify valid coindices

this_image(coarray [,dim])	compute (local) coindices from object on an image, optionally only that for a specified codimension
image_index(coarray, sub)	compute (remote) image index from object and coindex value; zero for invalid coindex.

e.g., for later use in synchronization statements

Examples

```
cindx = this_image(z)
```
on image 7, returns `[2,4]`

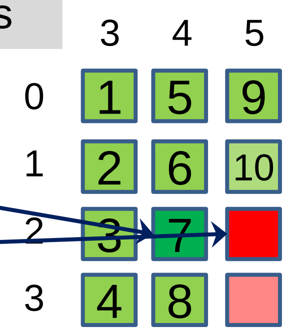
```
m1   = this_image(z,1)
```
on image 7, returns `2`

```
img = image_index(z,[2,4])
```
on **all** images, returns `7`

```
img = image_index(z, [2,5])
```
on **all** images, returns `0`

```
REAL :: z(10,10) [0:3,3:*]
INTEGER :: cindx(2), m1, img
```

10 images



Supporting intrinsics (2)

Cobounds and coshape

lcobound(coarray [,dim] [,kind])	compute lower cobound(s) of a coarray
ucobound(coarray [,dim] [,kind])	compute upper cobound(s) of a coarray
coshape(coarray [,dim] [,kind])	compute size(s) of the codimensions of a coarray (ucobound – lcobound + 1) F18

- additional `dim` argument: return scalar value for specified codimension
- additional `kind` argument: determine kind of result value

Examples

```
uc = ucobound(z)
```

on **all** images, returns `[3,5]`

```
lc = lcobound(z)
```

on **all** images, returns `[0,3]`

```
REAL :: z(10,10) [0:3,3:*]
INTEGER :: lc(2), uc(2)
```

10 images

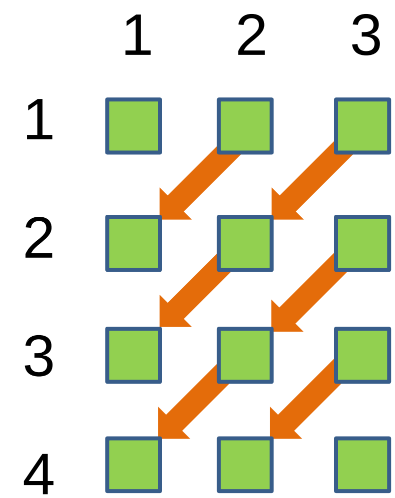
Usage scenario

- **Cartesian topology**
  - e.g. require data access to a neighbouring submatrix
  - usually want to **avoid** ragged pattern

```
REAL, ALLOCATABLE :: a(:,:)[:,:]
INTEGER :: n, p, q, ip, iq
: ! calculate symmetric n, p, q
ALLOCATE ( a(n,n) [p,*] )
ip = this_image(a, 1); iq = this_image(a, 2)

: ! do calculations on local part of a

SYNC ALL
IF ( ip > 1 .AND. iq < q ) &
  a(:,:) = ... + a(:,:)[ip-1, iq+1] * ...

:
```

assert that
p * q == number
of images

assure cobounds
are not violated



Procedures with a coarray dummy argument

Implementation

```
SUBROUTINE process_co_mat(a, p, q)
  INTEGER, INTENT(IN) :: p, q
  REAL, INTENT(INOUT) :: a(:,:)[p,*]
  :
  ip = this_image(a, 1)
  iq = this_image(a, 2)
  SYNC ALL
  IF ( ip > 1 .AND. iq < q ) &
    a(:,:) = ... + &
    a(:,:)[ip-1, iq+1] * ...
  :
END SUBROUTINE
```

Invocation

```
REAL, ALLOCATABLE :: a(:,:)[:,:]
INTEGER :: n, p, q
: ! calculate symmetric n, p, q
ALLOCATE ( a(n,n) [p,*] )

CALL process_co_mat(a, p, q)
```

Permissible but questionable invocations

Corank mismatch

- corank 1 actual vs. corank 2 dummy argument

```
REAL, ALLOCATABLE :: a(:,:)[:]
INTEGER :: n, p, q
: ! calculate symmetric n, p, q
ALLOCATE ( a(n,n) [*] )
: ! set up local a
CALL process_co_mat(a, p, q)
```

- remapping of coindices is done when procedure is called
- this will work OK if all communication is done using the same remapping
  (either explicitly by the programmer, or via consistently used interfaces)

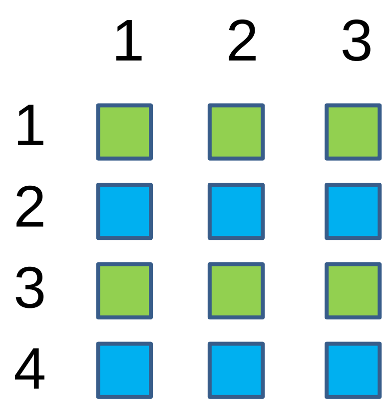
Image-dependent setup

```
REAL, ALLOCATABLE :: a(:,:)[:,:]
INTEGER :: n, p, q, p1
: ! calculate symmetric n, p, q
ALLOCATE ( a(n,n) [p,*] )

: ! give p1, q1 an
: ! image-dependent value

CALL process_co_mat(a, p1, q1)
```

- is permissible in principle because the mapping is done image-locally,
- but confusing for programmer,
- and likely to cause algorithmic trouble or illegal accesses inside called procedure

Generalized coindexing and teams

```
REAL, ALLOCATABLE :: a(:,:)[:,:]
INTEGER :: n, p, q, id
TYPE(team_type) :: my_teams
id = mod(this_image(),2) + 1
FORM TEAM(id, TEAM=my_teams, &
              NEW_IMAGE=...)
: ! calculate symmetric n, p, q
ALLOCATE ( a(n,n) [p,*] )
: ! set up local a

CHANGE TEAM ( my_teams )
  CALL process_co_mat(a, p/2, q)
END TEAM
```

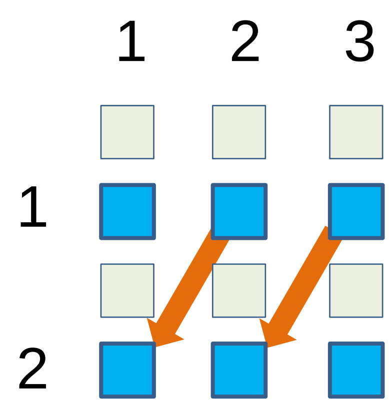


assuming **two** teams comprised of even/odd images **and** NEW_IMAGE retains ordering

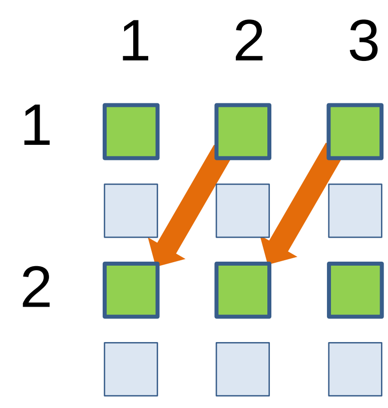
Inside the construct

- a is still a corank 2 coarray
- coindex mapping is to team-local image index, though

→ additional bookkeeping may be needed!



procedure call on team even



procedure call on team odd

Bookkeeping support (1)

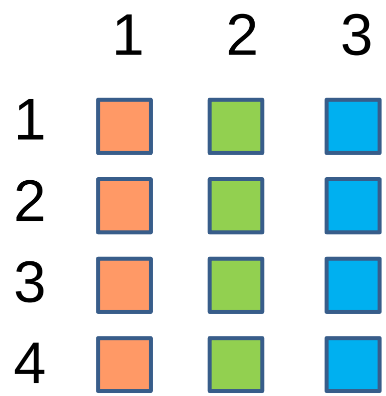
Associating coarray

- permits remapping on CHANGE TEAM opening statement

```
REAL, ALLOCATABLE :: a(:,:)[:,:]
INTEGER :: n, p, me(2)
TYPE(team_type) :: my_teams
: ! calculate symmetric n, p
ALLOCATE ( a(n,n) [p,*] )
me = this_image(a)
FORM TEAM(me(2), TEAM=my_teams, &
                 NEW_IMAGE=me(1))
: ! set up local a

CHANGE TEAM ( my_teams, &
              acol[*] => a )
  : ! work on acol
END TEAM
```

Example

- creates three teams



- each team corresponds to a „column“ part of the original coarray
- in each team, **`acol`** permits addressing this part of the coarray directly via its coindex

Bookkeeping support (2)

Extra `team` argument for some coarray intrinsics

```
USE, INTRINSIC :: iso_fortran_env, &
     ONLY : INITIAL_TEAM
REAL, ALLOCATABLE :: a(:,:)[:,:]
INTEGER :: n, p, me, local_ix(2), ix
TYPE(team_type) :: my_teams, initial
: ! calculate symmetric n, p
ALLOCATE ( a(n,n) [p,*] )
: ! set up my_teams

CHANGE TEAM ( my_teams )
  initial = get_team(INITIAL_TEAM)
  me = this_image(TEAM=initial)
  local_ix = this_image(a, TEAM=initial)
  ix = image_index(a, &
       [2,2], TEAM=initial)
  :
END TEAM
```

- permits inquiries for a team other than the current one
- `image_index()` can – instead of a `type(team_type)` argument – also take an integer `TEAM_NUMBER` argument to inquire on a sibling team
- for the cases with a coarray argument, the coarray must be established in the referenced team

Program Termination

Termination variants

error termination	by (implicit or explicit) execution of an ERROR STOP statement
normal termination	by execution of a STOP or END PROGRAM statement
F18 image failure	through a hardware or system software failure, or by execution of a FAIL IMAGE statement

highest

order of precedence

support for continued program execution upon image failure is an **optional** feature of the standard

Error termination

- **Initiation of error termination:**
  - by processor if an error condition is encountered on an image

    (e.g., I/O statement cannot be processed and is not handled by user code)
  - explicitly by executing an **ERROR STOP** statement in user code
- **Upon error termination by any image,**
  - the intent is that the implementation should terminate execution of **all** images as quickly as possible
- **Usual implications:**
  - all program state vanishes
  - files that were connected to opened I/O units for write access at the time error termination was initiated are likely to have an undefined state (corrupt or incomplete data)

Normal termination

- **Three steps:**
  1. image initiates termination
  2. synchronizes with all other images
     other images may still request data from terminating one
  3. image terminates execution
- **Step 2 guarantees that no image terminates before all have completed Step 1**
  - if all images execute normal termination from unordered segments, all is fine

    (for example, a stopping criterion might be propagated across all images prior to termination via a collectively executed STOP statement)

Normal termination from conceptually ordered segments

Semantics need to suppress deadlock

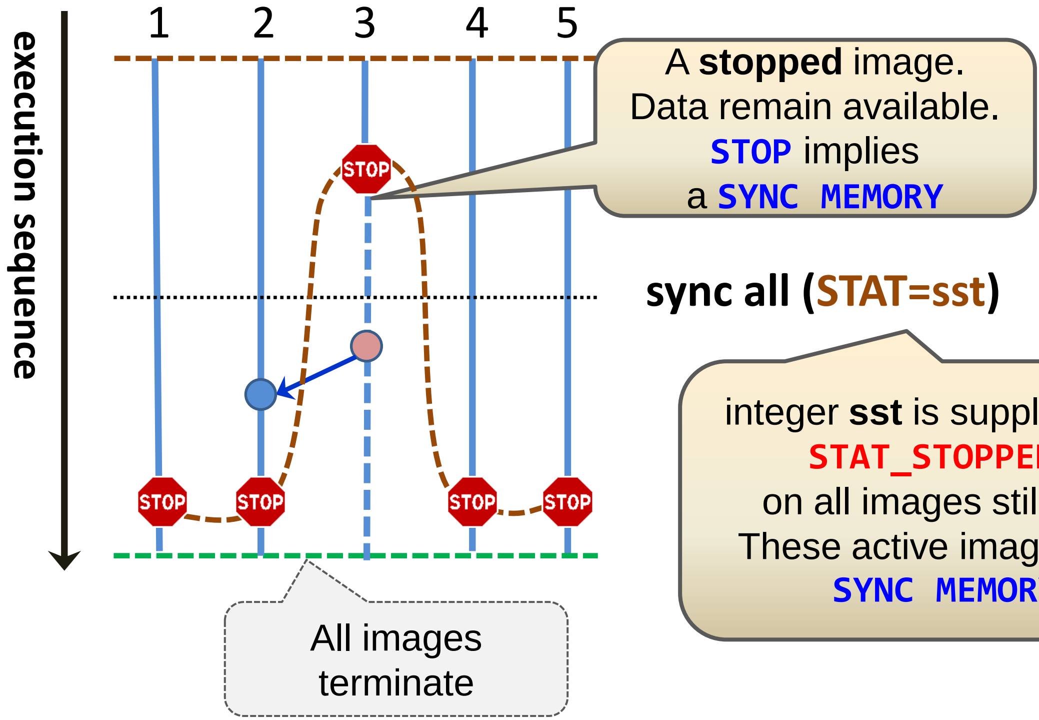
- obligation to add a STAT argument to all involved image control statements,
- else error termination is initiated

image 3

```
IF (…) STOP   ! normal termination
```

image 2

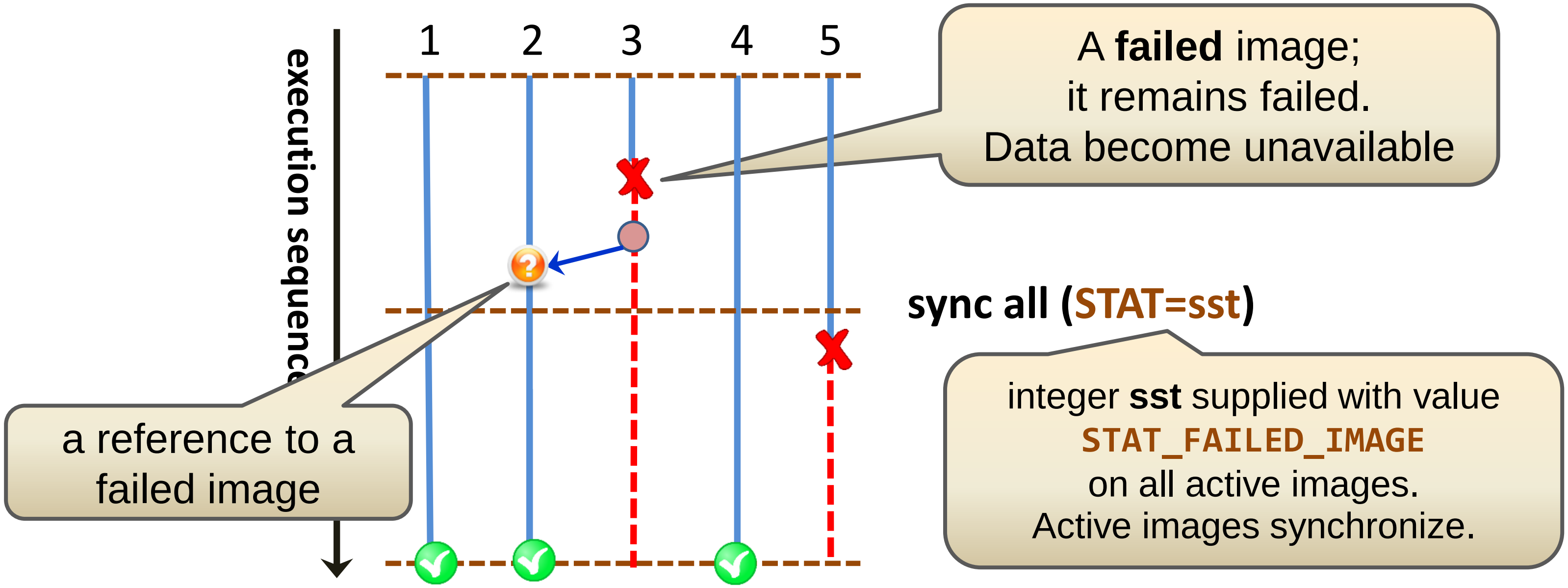
```
SYNC ALL (STAT=sst)
IF (sst==stat_stopped_image) THEN
  x = a(:)[3]      ! get and
  WRITE(…) x       ! save data
  STOP
END IF
```



integer **sst** is supplied with value STAT_STOPPED_IMAGE on all images still executing. These active images execute a SYNC MEMORY only.

Fail-safe execution (1): Behaviour after image failure

- **What happens in case an image fails?**
  - typical cause: hardware problem (DIMM, CPU, network link, …)
  - F08 (and all the rest of the HPC infrastructure): complete program terminates
- F18 : **optional support for continuing execution**
  - images that are not directly impacted by partial failure might continue
  - supported if the constant STAT_FAILED_IMAGE from ISO_FORTRAN_ENV is positive, unsupported if it is negative



Fail-safe execution (2): Programmer's Responsibilities

- **Synchronization: Without a STAT specifier on**
  - image control statements (including ALLOCATE and DEALLOCATE),
  - collective, MOVE_ALLOC, or atomic subroutine invocations,

  the program terminates if an image failure is determined to have occurred.

  With a STAT specifier, active images continue execution,
  - image control statements work as expected for these images,
  - collective and atomic subroutine results are undefined
- **Data handling and Control flow:**
  - programmer must deal with **loss of data** on failed image, and
  - with side effects triggered by references and definitions of variables on failed images
  - FAILED_IMAGES intrinsic: produces list of images **known** to have failed.

```
INTEGER, ALLOCATABLE :: fl(:)
:
SYNC ALL (STAT=sst)
fl = FAILED_IMAGES()
```

Returns indices of at least the images that have failed up to the „sync all"

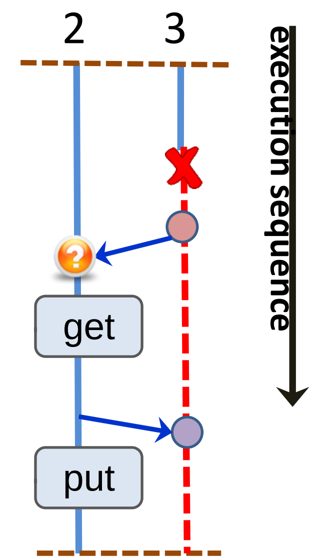
Referencing and defining objects

Reference to an object located on a failed image:

- Referencing image **continues** execution, but the object has a processor-dependent value
- example: statement executed on image 2

```
… = a(:)[3, STAT=sst]
```

optional **`stat`** argument permits identifying image 3 as failed



Definition of an object located on a failed image:

- Does not do anything, except setting a STAT argument if present
- example: statement executed on image 2

```
a(:)[3, STAT=sst] = …
```

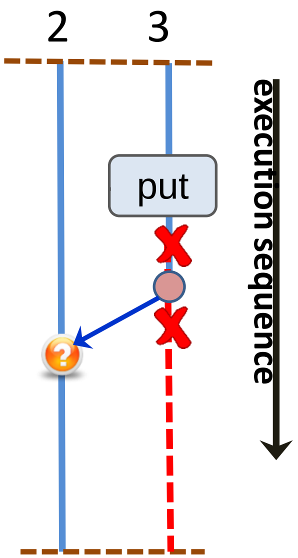
Defining objects (continued)

- **Definition of an object performed by a failed image:**
  - Objects that would become defined by the failed image during execution of the segment in which failure occurred become **undefined**.
  - example: statement executed on image 3

```
a(:)[2] = …
```

a[2] becomes undefined

„race condition“ of failure vs. put



- **Difficulty of diagnosis: images that reference a[2] in a subsequent segment need to**
  - know the communication pattern, and hence
  - identify image 3 as failed

avoid propagation of NaNs or incorrect values

FAIL IMAGE statement

- **A statement that causes an image executing it to fail**

- **Enables testing of code that should execute in a fail-safe manner**
  - execution might be conditioned on value returned by `random_number`

Fail-safe execution and teams

- **Algorithm may rely on a particular image-to-data mapping**
  - missing images cause this concept to fail
- **Possible solution:**
  - split image set into two subteams, `worker` (many) and `spare` (few)
  - only the `worker` team runs the simulation
  - if an image in `worker` fails, end team execution and generate a new `worker` team that uses an image from the original `spare` pool, assigning it the image index of the missing image.
  - this can be repeated until the `spare` pool is empty

Conclusions for PGAS programming in Fortran

Strengths

- easier to use than MPI
  - syntactic integration
  - one-sided semantics
- better control of memory locality than OpenMP
- implementation can optimize for latency
- independent of memory paradigm (coherency)
- integration into language standard
- no dependence on library idiosyncrasies

Weaknesses

- MPMD / hybrid will take some time to implement
- Irregular problems

  program-wide linked structures
- Without use of teams, assumes UMA → NUMA performance issues

  (all parallel models are impacted)
- Combination
  - with MPI
  - with OpenMP

  may hit implementation issues

following now: Exercise session 8

Thanks for your attention and good luck with your Fortran programming endeavours