Introduction to the PGAS (Partitioned Global Address Space) Languages Coarray Fortran (CAF) and Unified Parallel C (UPC)

Dr. R. Bader (LRZ)
Dr. A. Block (LRZ)

March 2019

Part 1: Basic Concepts

Execution and Memory Model

Declaration and usage of shared entities

Simple synchronization

Applying PGAS to classical HPC languages

- **Design target for PGAS extensions:**

smallest changes required to convert Fortran and C
into robust and efficient parallel languages

- add only a few new rules to the languages
- provide mechanisms to allow

explicitly parallel execution: **SPMD style** programming model

data distribution: **partitioned memory** model

synchronization vs. race conditions

memory management for dynamic sharable entities

collectively executed procedures (data redistribution and reductions)

- some additional "specialist" features may not be universally supported

Standardization efforts

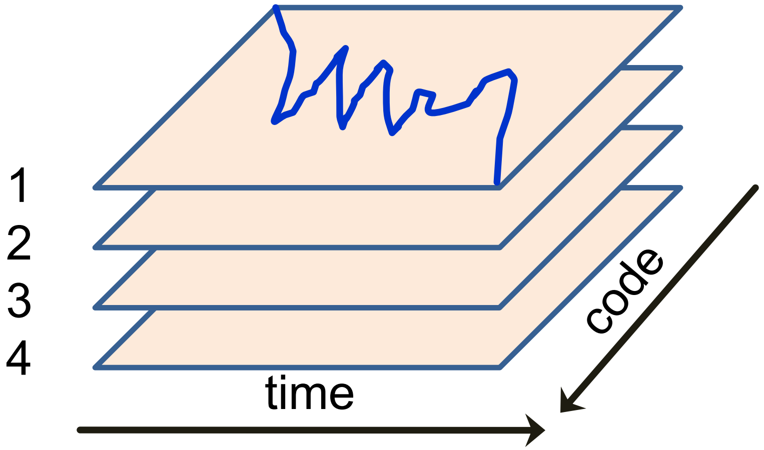
- **Baseline Coarrays**
  - Fortran 2008 standard
    (ISO/IEC 1539-1:2010, published in October 2010)
- **Additional parallel features in Fortran**
  - Fortran 2018 standard
    (ISO/IEC 1539-1:2018, published in November 2018)

current coarray compilers implement a subset of the additional features

- **UPC separate specification in three subdocuments**
  - language specification
  - **required** library specification
  - **optional** library specification
- **See „References“ slide near the end of the talk**

Execution model: UPC threads / CAF images

- **Going from single to multiple execution contexts**
  - CAF - **images**:



  - UPC uses zero-based counting
  - UPC uses the term **thread** where CAF has images

- **Replicate single program a fixed number of times**
  - set number of replicates at **compile** time or at **execution** time
  - asynchronous execution – **loose** coupling unless program-controlled synchronization occurs
- **Separate set of entities on each image/thread**
  - program-controlled exchange of data (imposed by algorithm)
  - synchronization may be needed

Execution model: Resource mappings

- **One-to-one:**
  - each image is executed by a single physical processor core
- **Many-to-one:**
  - some (or all) images are executed by multiple cores each (e.g., implementation could support OpenMP multi-threading within an image)
- **One-to-many:**
  - fewer cores are available to the program than images
  - scheduling issues
  - useful typically only for algorithms which do not require the bulk of CPU resources on one image
- **Many-to-many**
- **Note:**
  - startup mechanism and resource assignment method are implementation-dependent

Comparison with other parallelization methods

ratings: 1-low 2-moderate 3-good 4-excellent

	MPI	OpenMP	Coarrays	UPC
Portability	yes	yes	yes	yes
Interoperability (C/C++)	yes	yes	no	yes
Scalability	4	2	1-4	1-4
Performance	4	2	2-4	2-4
Ease of Use	1	4	2.5	3
Data parallelism	no	partial	partial	partial
Distributed memory	yes	no	yes	yes
Data model	fragmented	global	fragmented	global
Type system integrated	no	yes	yes	yes
Hybrid parallelism	yes	partial	(no)	(no)

PGAS languages' hardware needs:

good scalability for fine-grain parallelism in distributed memory systems will require use of special interconnect hardware features

„Hello world“ with PGAS

CAF – integer-valued intrinsics for image management

```
program hello
  implicit none
  write(*, '(''Hello from image '',i0, '' of '',i0)') &
        this_image(), num_images()
end program
```

between `1` and `num_images()`

non-repeatably unsorted output if multiple images/threads used

UPC

- uses integer expressions (macro functions) for the same purpose

```
#include <upc.h>
#include <stdlib.h>
#include <stdio.h>

int main (void) {
  printf(“Hello from thread %d of %d \n”, \
          MYTHREAD, THREADS);
  return 0;
}
```

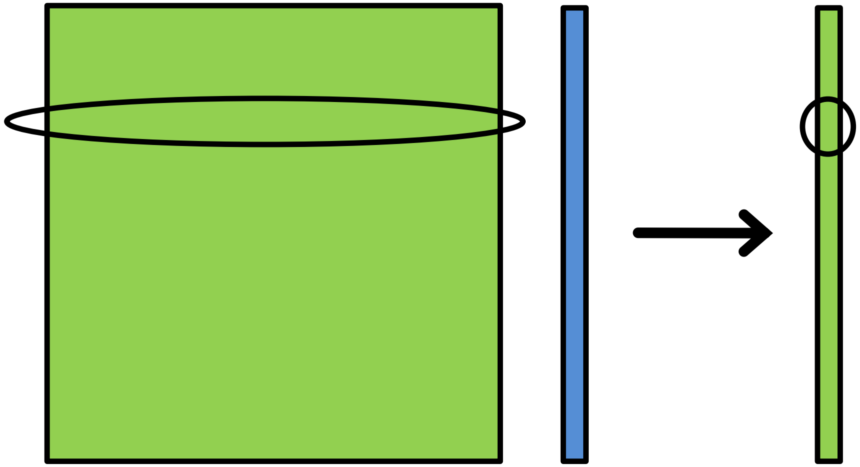
required for use of UPC macros and functions

between `0` and `THREADS - 1`

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

Fortran:

```
integer, parameter :: N = …
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)
```

BLAS routine

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

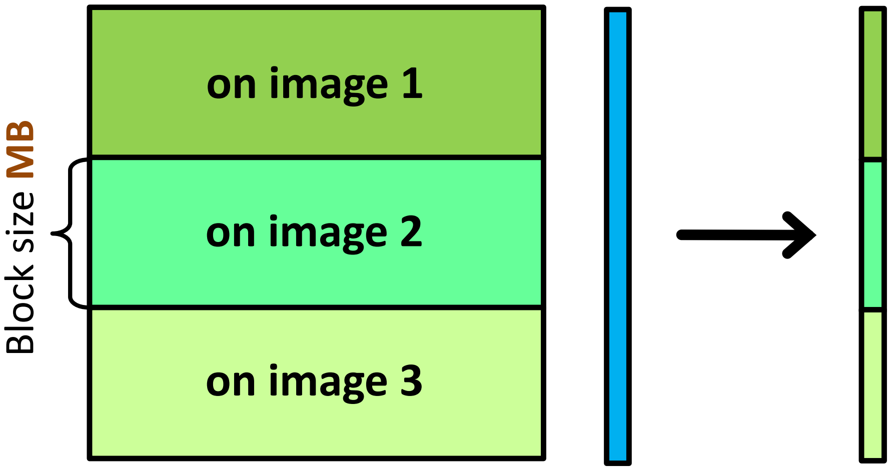
C:

```
float Mat[N][N], V[N];
float B[N];  // result

for (icol=0; icol<N; icol++) {
  for (irow=0;irow<N;irow++) {
    Mat[icol][irow] =
         matval(irow+1,icol+1);
  }
  V[icol] = vecval(icol+1);
}
cblas_sgemv(CblasColMajor,
     CblasNoTrans,N,N,1.0,
     (float *) Mat,N,V,1,0.0,B,1);
```

- C compared to Fortran: row-major mapping of indices to storage, zero based

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 computational 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

© 2010-19 LRZ

Memory model part 1: Image-local entities

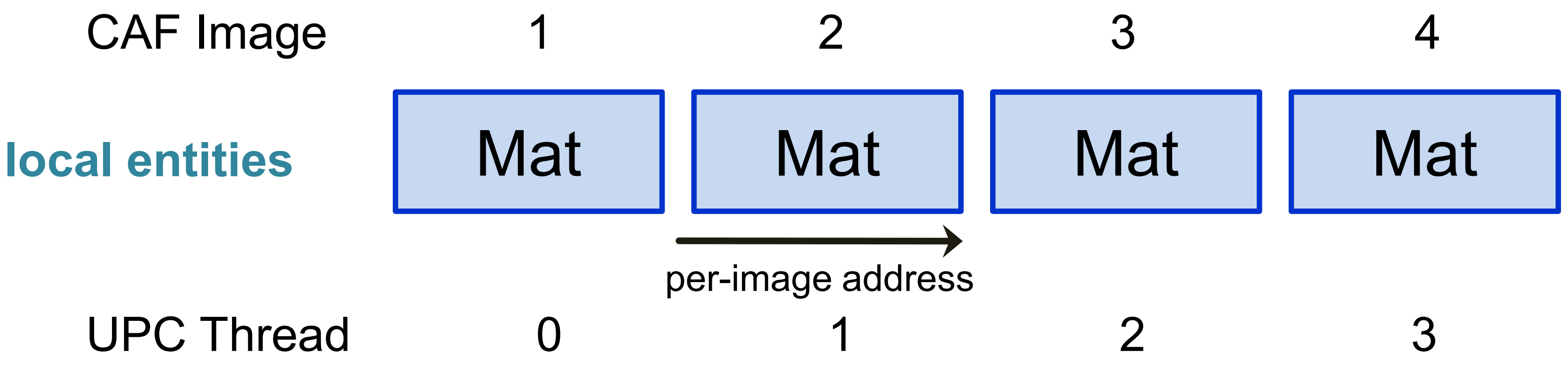
Modified declarations

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

```
float Mat[N][MB], V[N];
float B[MB];
```

Semantics for PGAS replicated execution

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



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

© 2010-19 LRZ

Work sharing the initialization and the 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 the initialization and the M*v processing

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

```
for (icol=0,icol<N,icol++) {
  for (i=0,i<MB,i++) {
    irow = MYTHREAD * MB + i;
    Mat[icol][i] = matval(irow+1,icol+1);
  }
  V[icol] = vecval(icol+1);
}
cblas_sgemv(CblasColMajor,
     CblasNoTrans,MB,N,1.0,
     (float *) 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
- **Fragmenting can be avoided in UPC** → discussed later

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}`**

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

```
real :: a(ndim, …)
p = this_image()
do i=1, nlocal
   j = …        ! global index
   a(i,…) = … ! expression involving j
end do
```

may vary between images

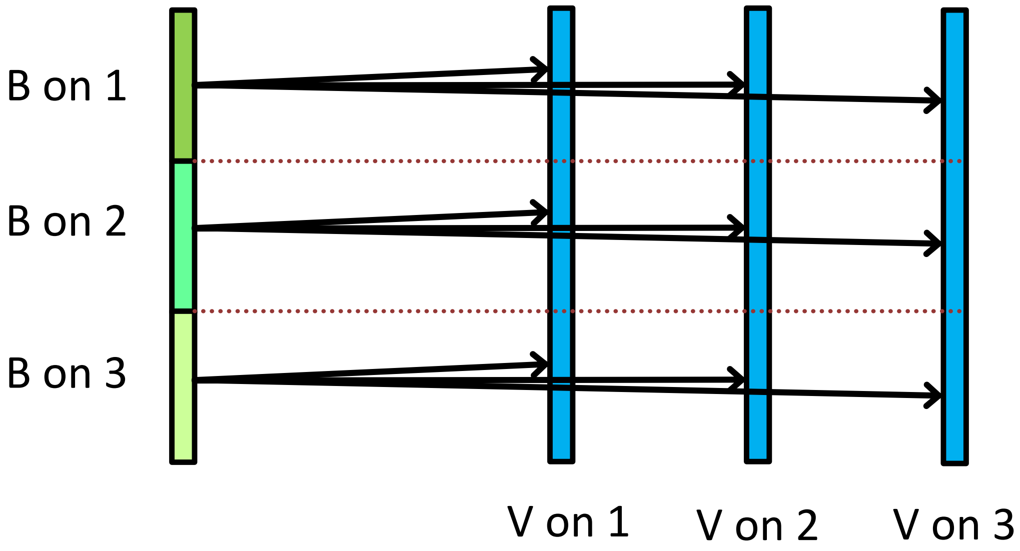
$$j = \sum_{q=1}^{p-1} nlocal\{q\} + i$$

for a **blocked** distribution

- for a work-balanced problem: **`nlocal{p}`** typically the same on all images, except the last one, which may have a smaller value

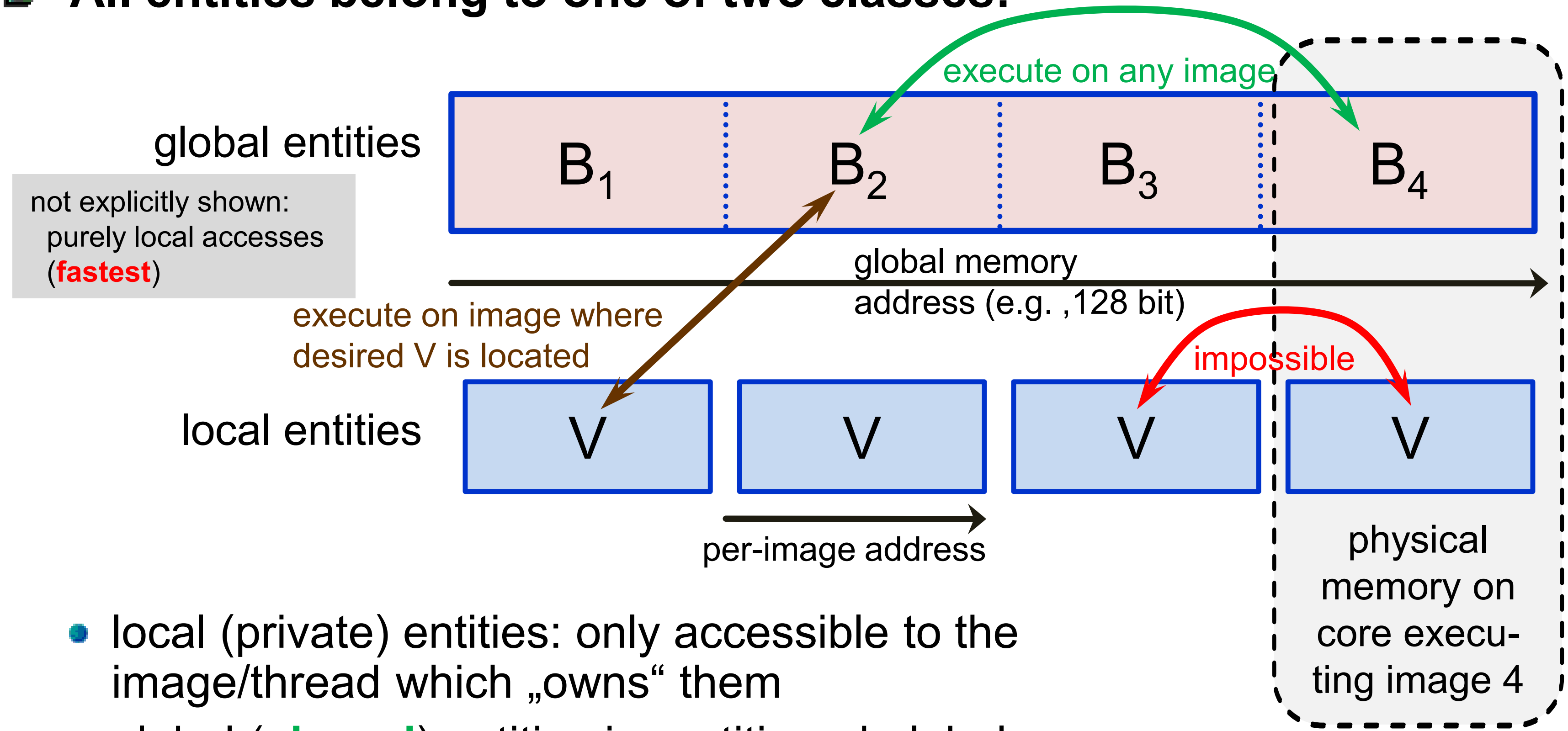
Ilustrating the need for communication

- **Open issue from „trivial“ 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 memory model

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



  - local (private) entities: only accessible to the image/thread which „owns“ them
  - 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“:
→ similar (but not exactly the same) as in OpenMP

Declaration of coarrays/shared entities

(simplest case)

CAF

- coarray requires explicit or implicit `codimension` attribute (square brackets)

support dynamic configuration

```
real, &
     codimension[*] :: B(MB)
```

- declare **local** number of elements per image
- star in square brackets: program can be agnostic about number of images to be used at compile time

UPC

- shared entity must be declared with the `shared` attribute

```
shared [1] float B[MB*NTMX];
```

- specify **aggregate** number of elements across all threads

MB = 3, NTMX = 3:
constants viz. macro constants

Data distribution of coarrays/shared entities
(simplest case)

CAF

- same distribution as for private objects
- coarray notation with **explicit** indication of location (coindex in square brackets)
- symmetry is enforced
  (asymmetric data must use derived types)

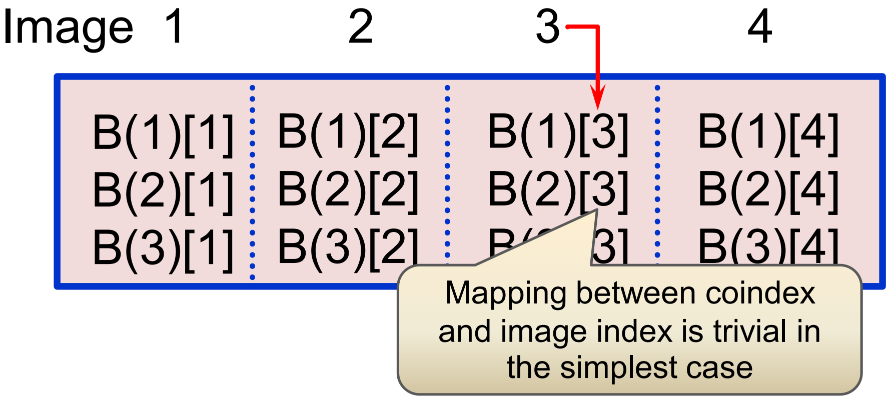


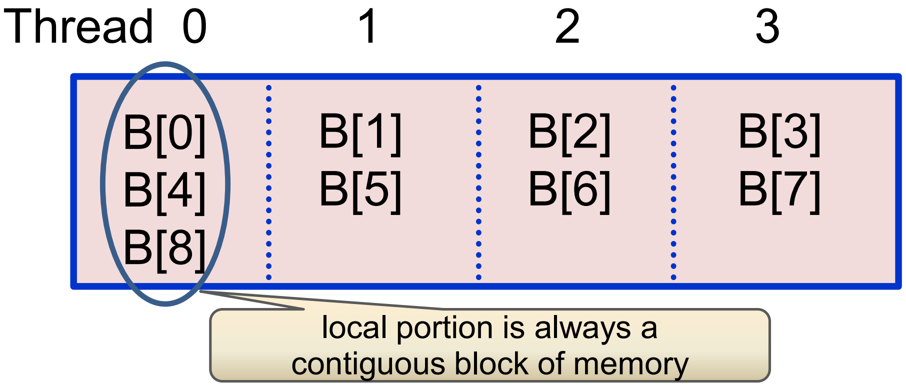
Image 1	2	3	4
B(1)[1]	B(1)[2]	B(1)[3]	B(1)[4]
B(2)[1]	B(2)[2]	B(2)[3]	B(2)[4]
B(3)[1]	B(3)[2]	B(3]	B(3)[4]

Mapping between coindex and image index is trivial in the simplest case

- more images → additional coindex value

UPC

- round-robin distribution
- implicit locality (various **blocking** strategies)
- potential asymmetry – threads in general may have uneven share of data



Thread 0	1	2	3
B[0]	B[1]	B[2]	B[3]
B[4]	B[5]	B[6]	B[7]
B[8]			

local portion is always a contiguous block of memory

- more threads → e.g., B[4] located on a different physical memory

© 2010-19 LRZ

Enforcing symmetry for UPC shared objects

(if you desire to make them as similar as possible to coarrays)

- **Two methods**
  - extra dimension indexes threads
  - THREADS macro in declaration
- **Method 1**

```
shared int A[3][THREADS];
```

Thread	0	1	2	3
	A[0][0] A[1][0] A[2][0]	A[0][1] A[1][1] A[2][1]	A[0][2] A[1][2] A[2][2]	A[0][3] A[1][3] A[2][3]

- **Method 2**
  - use a non-default block size
    (number of subsequent elements placed on any thread)

```
shared [3] int A[THREADS][3];
```

Thread	0	1	2	3
	A[0][0] A[0][1] A[0][2]	A[1][0] A[1][1] A[1][2]	A[2][0] A[2][1] A[2][2]	A[3][0] A[3][1] A[3][2]

- **Notes:**
  - THREADS macro may not be usable in certain declaration contexts (e.g., inside function body) if number of threads is determined at run time
  - implementation dependent block size limit can make use of method 2 problematic
  - programmers may prefer implicit distribution for simplicity of use (but then: **beware unintentioned** cross-thread accesses)

© 2010-19 LRZ

UPC shared data: variations on blocking

General syntax

- for a one-dimensional array

```
shared [block_size] type \
  var_name[total size];
```

- scalars and multi-dimensional arrays also possible

Values for `block_size`

- omitted → **default** value is 1
- integer constant (maximum value `UPC_MAX_BLOCK_SIZE`)
- `[*]` → one block on each thread, as large as possible, size depends on number of threads
- `[]` or `[0]` → all elements on one thread

Some examples:

```
shared [N] float C[N][N];
```

- complete matrix rows on each thread (≥1 per thread if at most N threads are used)

```
shared [*] float \
         B[THREADS][MB];
```

- in this example, storage sequence **matches with method 2** from previous slide
- static THREADS environment may be required (compile-time thread number determination)

CAF: Coarray declaration variants

```
integer :: a(3)[*]
```

implicit CODIMENSION attribute

is equivalent to

```
integer, codimension[*] :: a(3)
```

- **A scalar coarray:**

```
integer, codimension[*] :: s
```

- **An array coarray of rank 2 and corank 2**
  (details explained later)

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

Inter-image communication: coindexed access

CAF Pull (Get)

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

sectioning is obligatory

a **coindexed** reference

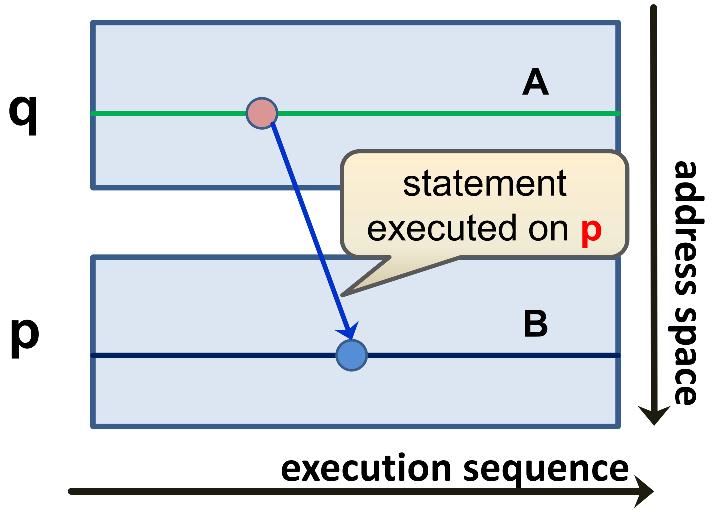
CAF Push (Put)

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

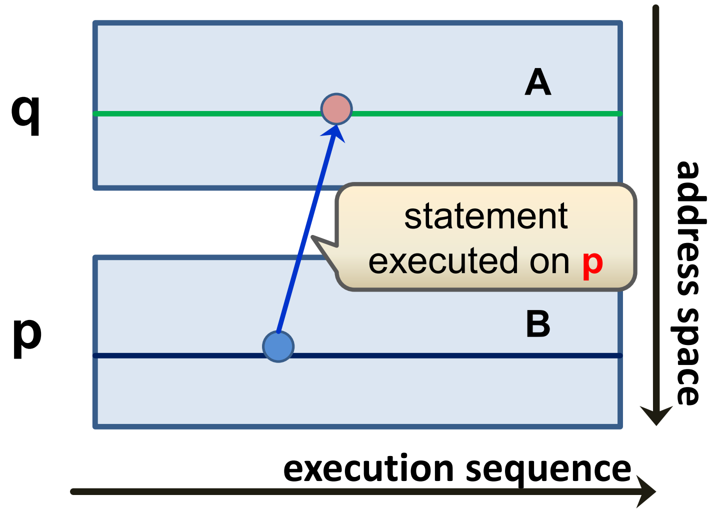
a **coindexed** definition

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

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



statement executed on **p**



statement executed on **p**

Inter-thread communication with UPC

- **Using symmetric declaration of shared object**

```
int b[MB];
shared [MB] int a[THREADS][MB];
```

- **UPC Pull**

```
if (MYTHREAD == p) {
  for (i=0; i<MB; i++) {
    b[i] = a[q][i];
  }
}
```

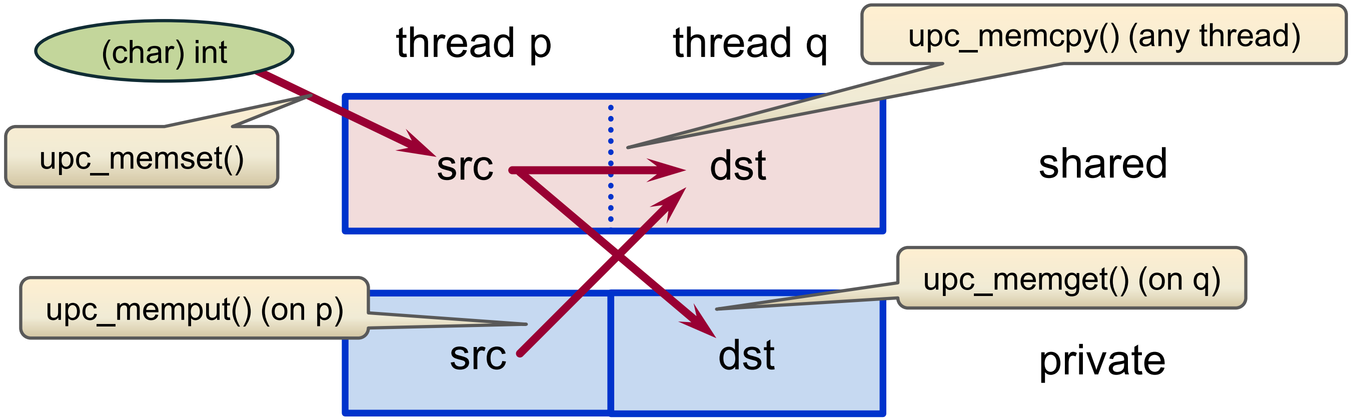
a[q][ ] is located on thread q

- **UPC Push**

```
if (MYTHREAD == p) {
  for (i=0; i<MB; i++) {
    a[q][i] = b[i];
  }
}
```

- **Note:**
  - lack of array support may cause this to be inefficient compared with Fortran → work around this with ...

UPC: One-sided memory block transfers



Available for efficiency

- operate in units of **bytes**
- use restricted pointer arguments
- more concise for structs, arrays

Restriction

- **contiguous** blocks of memory
- Berkeley UPC has extension for strided transfers

prototypes from `upc.h`

```
void upc_memcpy(shared void *dst,
    shared const void *src, size_t n);
void upc_memget(void *dst,
    shared const void *src, size_t n);
void upc_memput(shared void *dst,
    void *src, size_t n);
void upc_memset(shared void *dst,
    int c, size_t n);
```

Rewriting as block transfers ...

- **UPC Pull**

```
if (MYTHREAD == p) {
   upc_memget( &b[0],&a[q][0],MB*sizeof(int) );
}
```

MB elements starting at a[q][ ]
are located on thread q

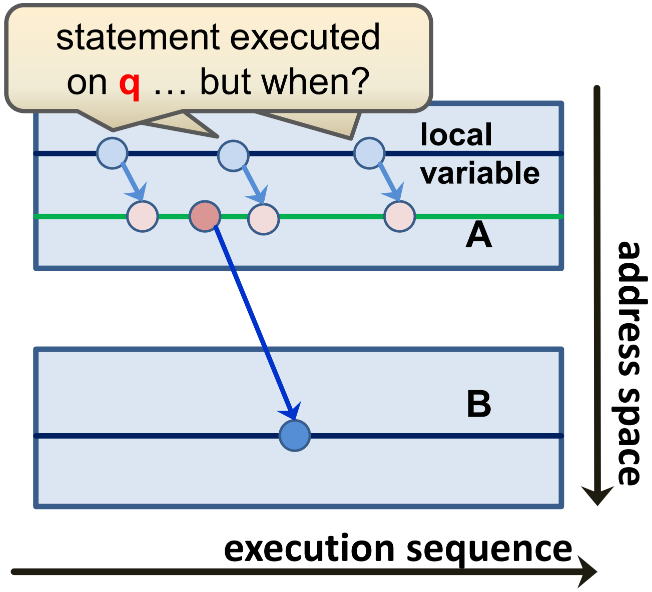
- **UPC Push**

```
if (MYTHREAD == p) {
   upc_memput( &a[q][0],&b[0],MB*sizeof(int) );
}
```

Synchronization requirements

Asynchronous execution

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

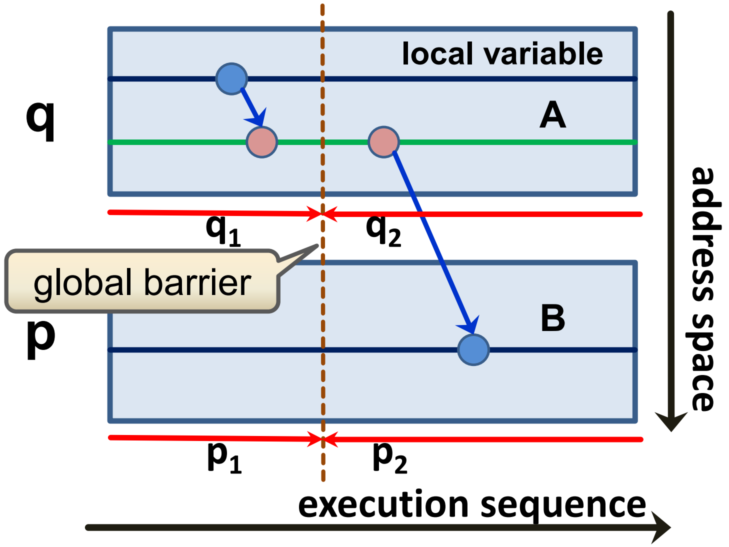


- causes race condition → **violates** language rules

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 global barrier

- **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)

In UPC, the spelling for the global barrier is **`upc_barrier;`**

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)

© 2010-19 LRZ

Image control for Get and Put patterns

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

UPC Pull (Get)

```
  a[MYTHREAD][i] = …;
upc_barrier;
if (MYTHREAD == p) {
  upc_memget( &b[0],&a[q][0],
            MB*sizeof(int) )



   … = b[i];

}
```

no sync required
(no communication)

consume b on
thread **p**

UPC Push (Put)

```
  b[i] = …;

if (MYTHREAD == p) {
  upc_memput( &a[q][0],&b[0],
            MB*sizeof(int) );
:  // further statements
upc_barrier;
if (MYTHREAD == q) {
      … = a[MYTHREAD][i];
}
```

consume a on
image **q**

Image control 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.

CAF Pull (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**

CAF Push (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

Local accesses to CAF coarrays

- **Design aim for non-coindexed accesses:**
  - should be optimizable as if they were local entities

Performance!

```
integer :: a(MB)[*]
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

Local accesses to UPC shared objects

- **Programmer is responsible for correct indexing**
  - symmetric object setup can help:

```
shared int A[MB][THREADS];
int B, i;

B = 0
for (i=0; i<MB; i++) {
  B += A[i][MYTHREAD];
}
```

  - non-symmetric shared objects require care to avoid unwanted communication
  - performance for current implementations will still be bad, because communication calls are still generated by the compiler

Tuning local accesses in UPC

- **Cast to a thread-local pointer to extract local portion of a shared object**

```
shared int A[MB][THREADS];
int B, i;
int *A_loc;

B = 0;
A_loc = (int *) A;
for (i=0; i<MB; i++) {
  B += A_loc[i];
}
```

on thread **p**,
`A_loc` selects
`A[0][p]`
`A[1][p]`
`A[2][p]`

  - non-symmetric shared objects require care to avoid misaddressing

- **Casting is also needed when calling functions that assume local memory**

```
  my_proc( (int *) A, … );
```

first formal parameter of my_proc is an int *

© 2010-19 LRZ

Integration of the type system
(„POD“ data: static type components)

CAF:

```
type :: body
  real :: mass
  real :: coor(3)
  real :: velocity(3)
end type
```

UPC:

```
typedef struct {
  float mass;
  float coor[3];
  float velocity[3];
} Body;
```

enforced storage order

declare and use entities of this type (symmetric variant):

Components of shared object are shared

```
type(body) :: asteroids(100)[*]
type(body) :: s
:
if (this_image() == p) &
    s = asteroids(5)[q]
```

```
shared [1] \
   Body asteroids[100][THREADS];
Body s;
:
if (MYTHREAD == p) {
  s = asteroids[4][q];
}
```

- compare this with effort needed

  to implement the same with MPI (dispense with **all** of `MPI_TYPE_*` API)
- what about dynamic type components? → later in this talk

Part 2: Dynamic Entities

Pointer classification
Allocation and deallocation
Distributed structures

Pointers and Pointees

Remember pointer semantics

- different between C and Fortran

Fortran

```
<type> [, dimension (:[,:,…])], pointer :: ptr

ptr => var     ! ptr is an alias for target var
```

no pointer arithmetic
type and rank matching
ALLOCATABLE vs. POINTER

C

```
<type> *ptr;

ptr = &var;    ! ptr holds address of var
```

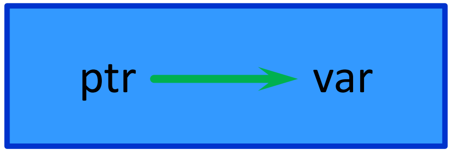
pointer arithmetic
rank irrelevant
pointer-to-pointer
pointer-to-void / recast

Joint Fortran and C feature:

- possibility to reference or define another entity via the pointer:

```
ptr = xy    ! defines target var
```

```
*ptr = xy; // defines pointee var
```


ptr → var

PGAS and pointers:

- more variants of pointer association because of different kinds of memory

© 2010-19 LRZ

Case 1: private pointers to private memory

- **CAF**

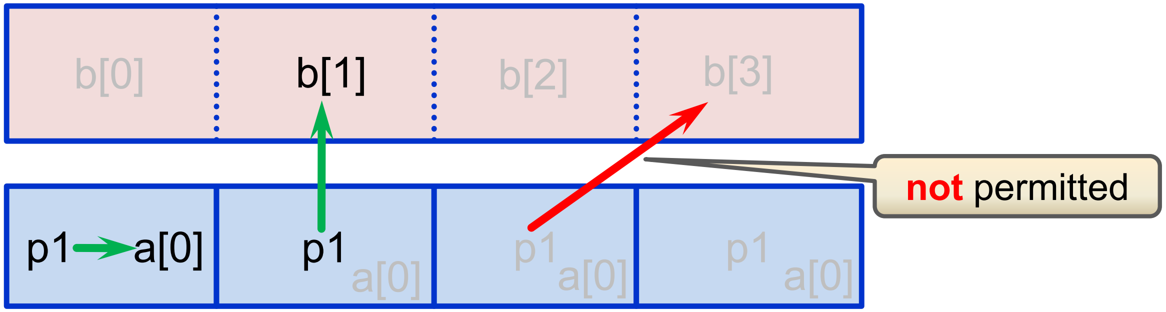
```
integer, pointer :: p1
integer, target :: a(0:n)
integer, target :: b[0:*]

if (this_image() == 1) then
  p1 => a(0)
elseif (this_image() == 2) then
  p1 => b
end if
```

pointer to **local** portion of scalar coarray

- **UPC**

```
int *p1;
int a[N];
shared int b[THREADS];

if (MYTHREAD == 0) {
   p1 = &a[0];
} elseif (MYTHREAD == 1) {
   p1 = (int *) b;
}
```

cast to local



not permitted

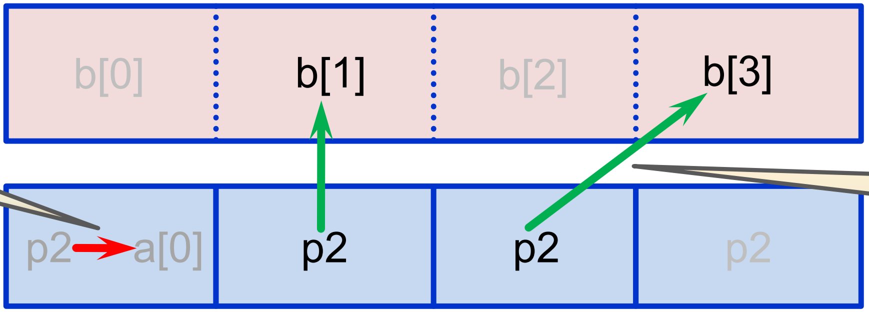
Case 2: private pointers to shared memory

CAF

- concept is not defined – a POINTER cannot be associated with more than the local portion of a coarray

UPC

```
shared int *p2;
shared int b[THREADS];

if (MYTHREAD == 1) {
   p2 = &b[1];
} elseif (MYTHREAD == 2) {
   p2 = &b[3];
}
if (p2) {
   // dereference local p2
}
```



not permitted

associated with remote object

Case 3: shared pointers to private memory

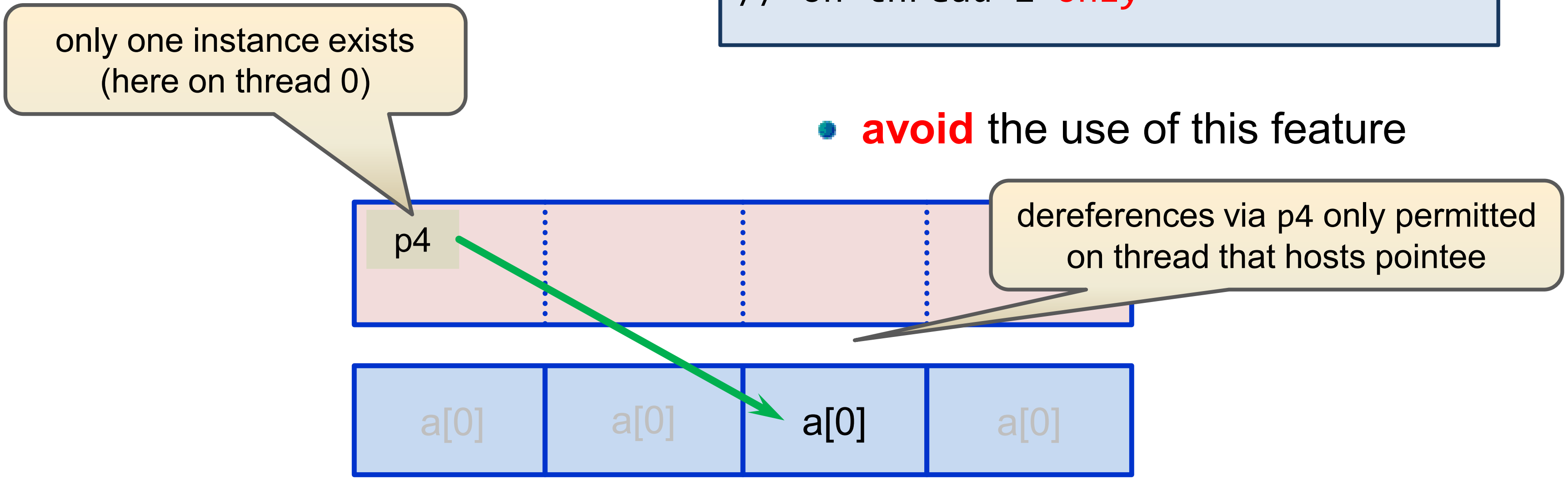
CAF

- concept is not defined – a coarray cannot have the POINTER attribute
  (However, dynamic type components provide more extended semantics that will be discussed soon)

UPC

```
int shared *p4;
int a[N];

if (MYTHREAD == 2) {
   p4 = &a[0];
}
// dereference the shared pointer
// on thread 2 only
```

- **avoid** the use of this feature



Case 4: shared pointers to shared memory

CAF

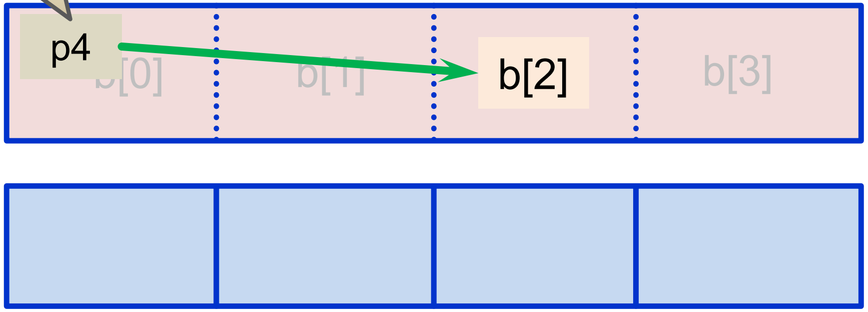
- concept is not defined – a coarray cannot have the POINTER attribute

UPC

```
shared int shared *p4;
shared int b[THREADS];

if (MYTHREAD == 2) {
   p4 = &b[2];
}
upc_barrier;
// dereference the shared pointer
// on any thread
```

likely expensive

only one instance exists (here on thread 0)



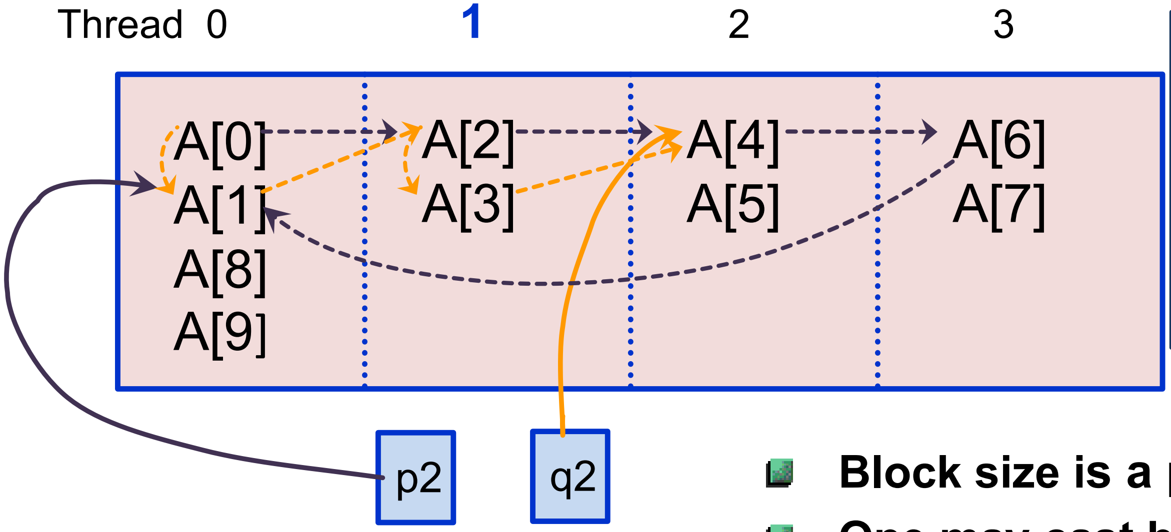
UPC: Shared Pointer blocking and casting

- **Assume 4 threads:**

```
shared [2] int A[10];
shared int *p2;
shared [2] int *q2;
```

block size different from A

block size same as for A



after pointer increment

```
if (MYTHREAD == 1) {
  p2 = (shared int *) &A[0];
  p2 += 4;
  q2 = &A[0];
  q2 += 4;
}
```

unintuitive sequence

natural sequence

- **Block size is a part of the variable‘s type**
- **One may cast between pointers with different block sizes**
  - pointer arithmetic follows blocking („phase“) of pointer (not pointee)!
  - cast changes the view but does not move any data
- **Consequences for libraries → see later**

Dynamic allocation and deallocation:

Remember serial semantics

- **Fortran:**
  - one of two attributes usable: POINTER or ALLOCATABLE
  - favour use of ALLOCATABLE for „simple“ objects (reason: no dangling pointers, no memory leaks)
  - ALLOCATE and DEALLOCATE statements
- **C:**
  - pointers can be used to point at a dynamically allocated object
  - avoid dangling pointers and memory leaks (programmer‘s responsibility)
  - library functions: malloc() and free()

Making the vector „v“ from the M*v example a dynamic entity:

```
real, allocatable :: V(:)
integer :: NV
NV = … ! determine size
allocate(V(NV))
:      ! use V
deallocate(V)
```

```
float *v;
int nv;
nv = … // determine size
v = (float *) \
     malloc(nv*sizeof(float));
:      // use v
free(v);
```

Dynamic entities: Shared memory area management

collective allocation facility which **synchronizes all** images/threads

CAF:

deferred shape and coshape

```
integer, &
  allocatable :: b(:)[:]
mb = …
allocate( b(mb)[*] )
```

- **symmetric** allocation required: same type, type parameters, bounds and cobounds on every image, in unordered segments
- referencing and defining is straightforward

```
deallocate( b )
```

- deallocation: on **all** images, synchronizes on entry

UPC:

```
shared [MB] int *b;
b = (shared [MB] int *) \
      upc_all_alloc( \
      THREADS,MB*sizeof(int) );
```

number of blocks

bytes per block

- layout equivalent to coarray on the left (but MB is compile time constant)
- arguments of type `size_t`
- deallocation via

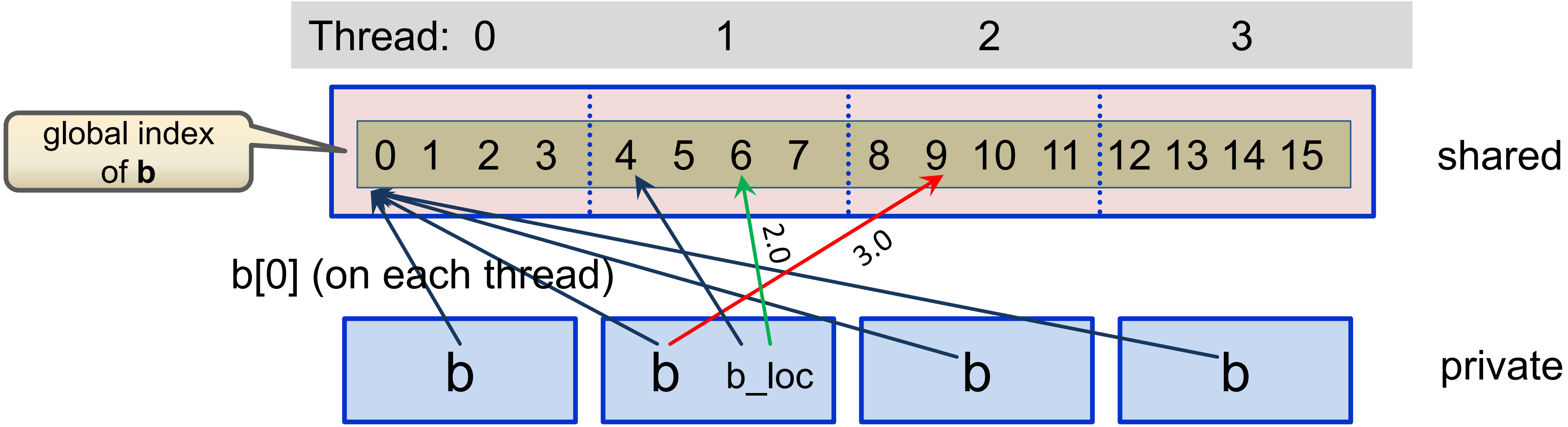
```
upc_barrier;
if (MYTHREAD==0) upc_free( b );
```

is **not** collective (must be performed only on one thread)

UPC 1.3 provides `upc_all_free()`

Referencing or defining the allocated UPC pointer

- **After invocation of upc_all_alloc(), on each thread**
  - a private copy of the pointer „b“ exists (→ can use independently),
  - which points at the same start address of a set of blocks **distributed** in the shared memory space
- **Assuming MB==4 and using 4 threads, we have**



```
int *b_loc = (int *) b;
if (MYTHREAD==1) {
   b[9]=3.0;
   b_loc[2] = 2.0;
}
```

cross-thread and local definitions – see correspondingly color-coded arrows above and note the **`b_loc`** reindexing!

CAF: More on allocatable coarrays

- **Allocation and deallocation**
  - collectively operate on local portions of object
- **Allocatable components**
  - 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

  - allocation:

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

m has same value on all images

CAF: 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

Further Notes

Disallowed in Fortran:

- coarrays with POINTER attribute

```
integer, pointer :: p(:)[:]
```

- asymmetric allocation

```
! b declared earlier
allocate(b(this_image())[*])
```

```
allocate(b(mb) &
         [this_image():*])
```

- coarray allocation on image subset

```
if (this_image() < 2) &
         allocate(b(mb)[*])
```

UPC casting:

- inconsistency of block sizes in declaration and cast may cause problems

Inflexibility of symmetric data

- in CAF, may need to overallocate
  - load balance (one straggler)
- in UPC, may need to use block cyclic arrangements:
  - specify more blocks than threads (run time setting!)
  - beware load balancing (lose symmetry)
- further support for non-symmetric data → soon

© 2010-19 LRZ

Asymmetric (non-collective) allocation in UPC (1)

- **Per-thread pointer to a distributed set of shared blocks**

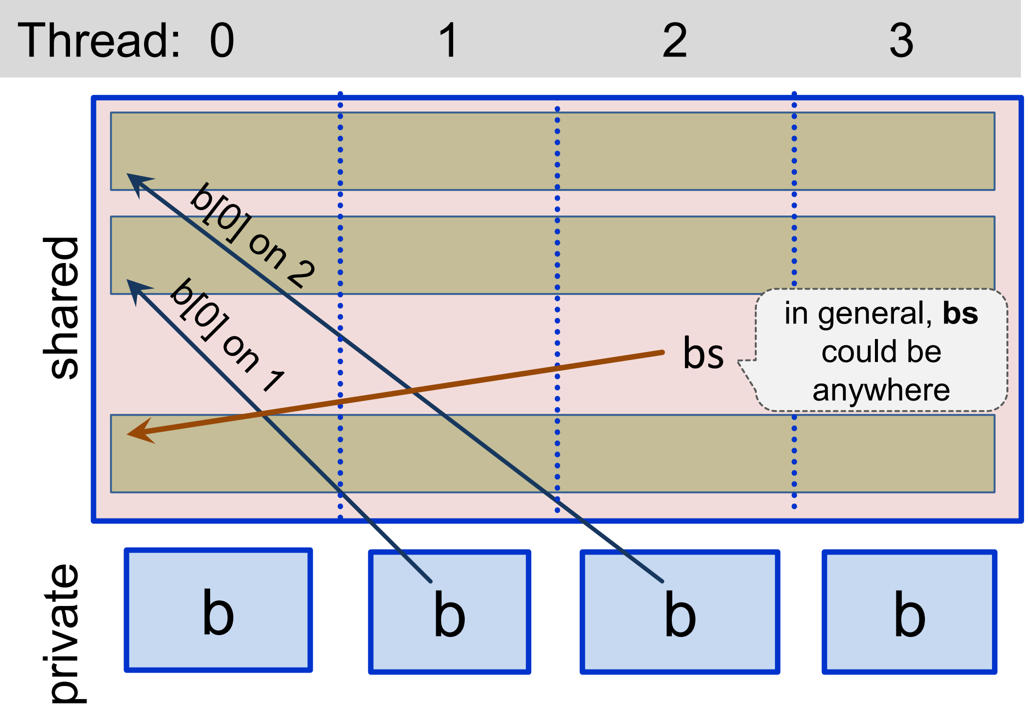
```
shared void * upc_global_alloc( size_t nblocks, size_t nbytes )
```

```
shared [MB] int *b;
shared [MB] int *shared bs;

if (MYTHREAD==1 || MYTHREAD==2) {
  b = (shared [MB] int *) \
      upc_global_alloc( \
      THREADS,MB*sizeof(int));
}
```

memory only accessible from allocating thread

```
if (MYTHREAD==3) {
  bs = (shared [MB] int *) \
      upc_global_alloc( \
      THREADS,MB*sizeof(int));
}
```

for a shared pointer to shared, only one thread may execute the allocation.



in general, **bs** could be anywhere

Asymmetric (non-collective) allocation in UPC (2)

- **Per-thread pointer to a shared block with affinity to allocating thread**

```
shared [] void * upc_alloc( size_t nbytes )
```

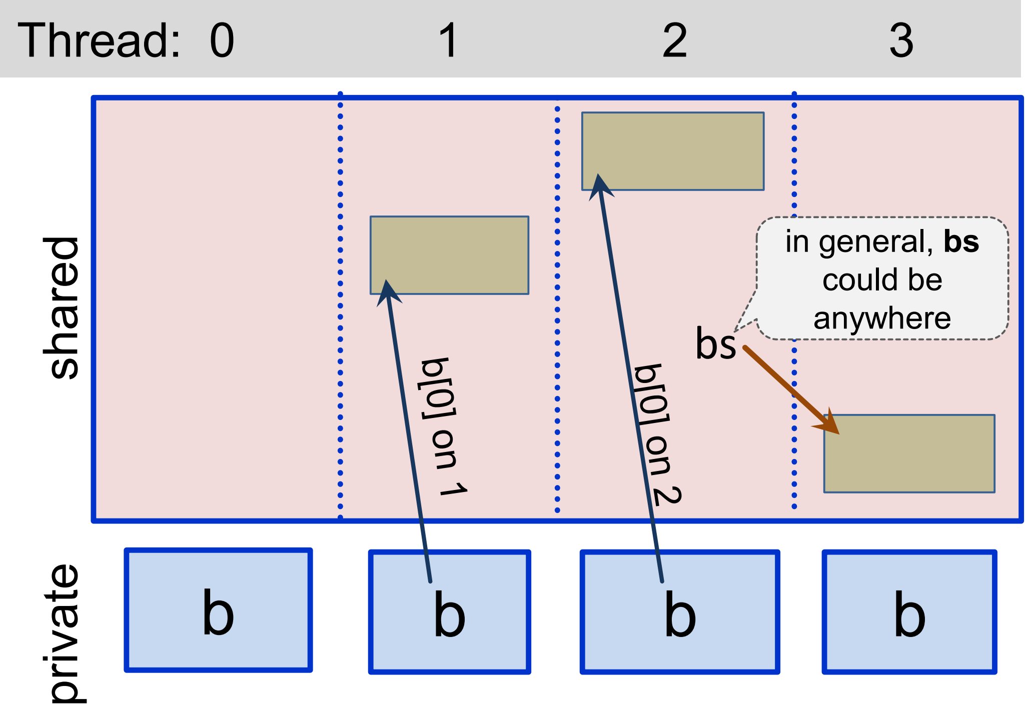
```
shared [] int *b;
shared [] int *shared bs;

if (MYTHREAD==1 || MYTHREAD==2) {
  b = (shared [] int *) \
      upc_alloc(MB*sizeof(int));
}
```

memory only accessible from allocating thread

```
if (MYTHREAD==3) {
  bs = (shared [] int *) \
       upc_alloc(MB*sizeof(int));
}
```

for a shared pointer to shared, only one thread may execute the allocation.

→ must **avoid** non-zero blocking factor



Distributed structures

- **Fortran „container types“**

```
type :: container
  real, pointer :: data(:) => null()
  : ! possibly further components
end type

type(container) :: a[*]
```

scalar coarray of container type

- with either POINTER or ALLOCATABLE components
- don‘t care which for this purpose

- **UPC shared component structure**

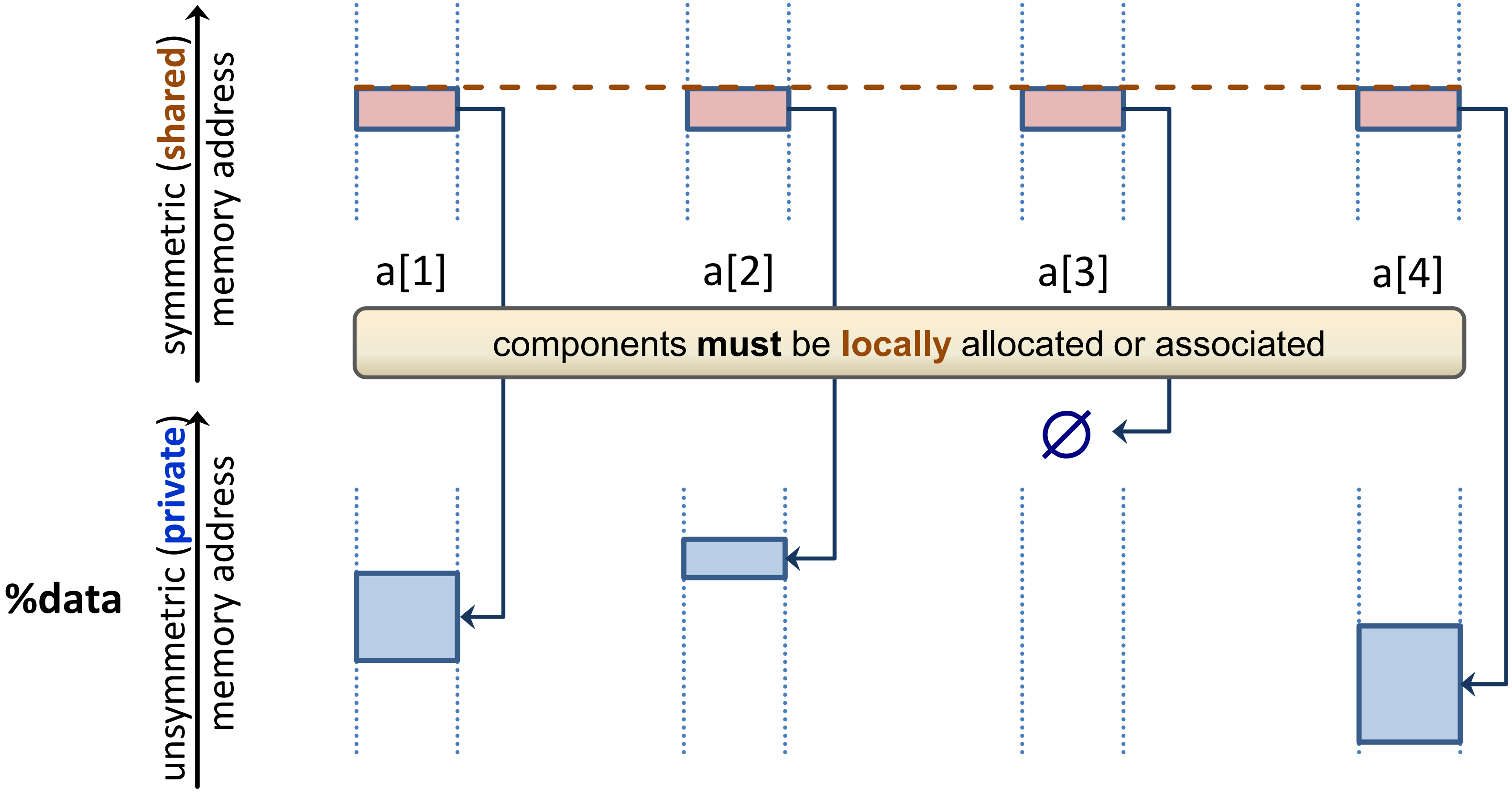
```
typedef struct {
  shared [] float *data;
  : // etc
} Container;

shared [1] Container a[THREADS];
```

shared object

- requires a pointer-to-shared component to enable cross-thread access to `.data`

CAF: unsymmetric objects

- **Illustrating the data layout**



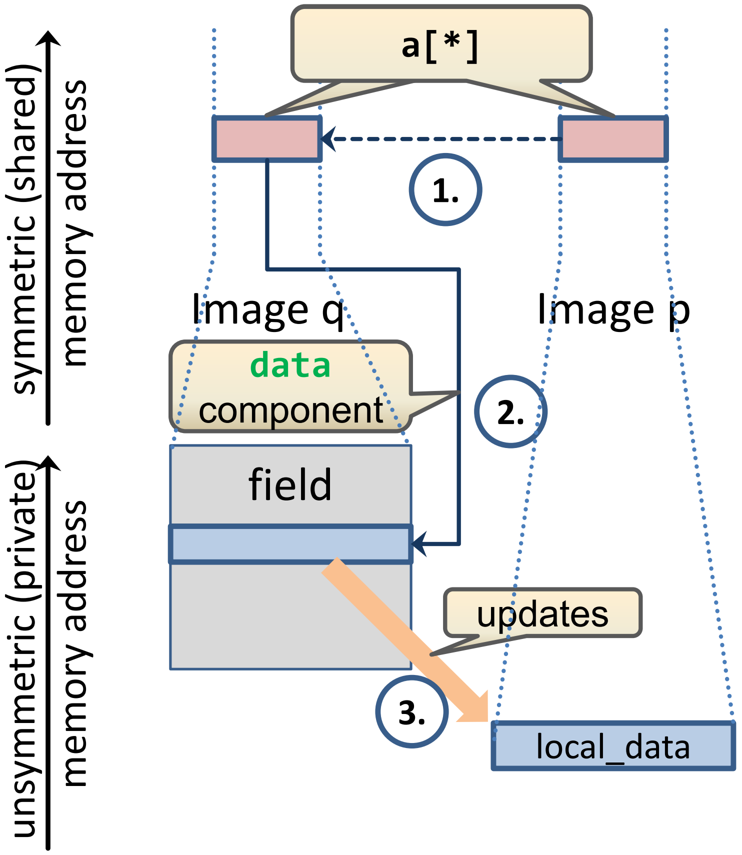
CAF: Accessing remote component data

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

1. access remote object **`a[q]`** from image p
2. obtain location of **`data`** component
3. transfer data component (or a subobject of it) to the executing image

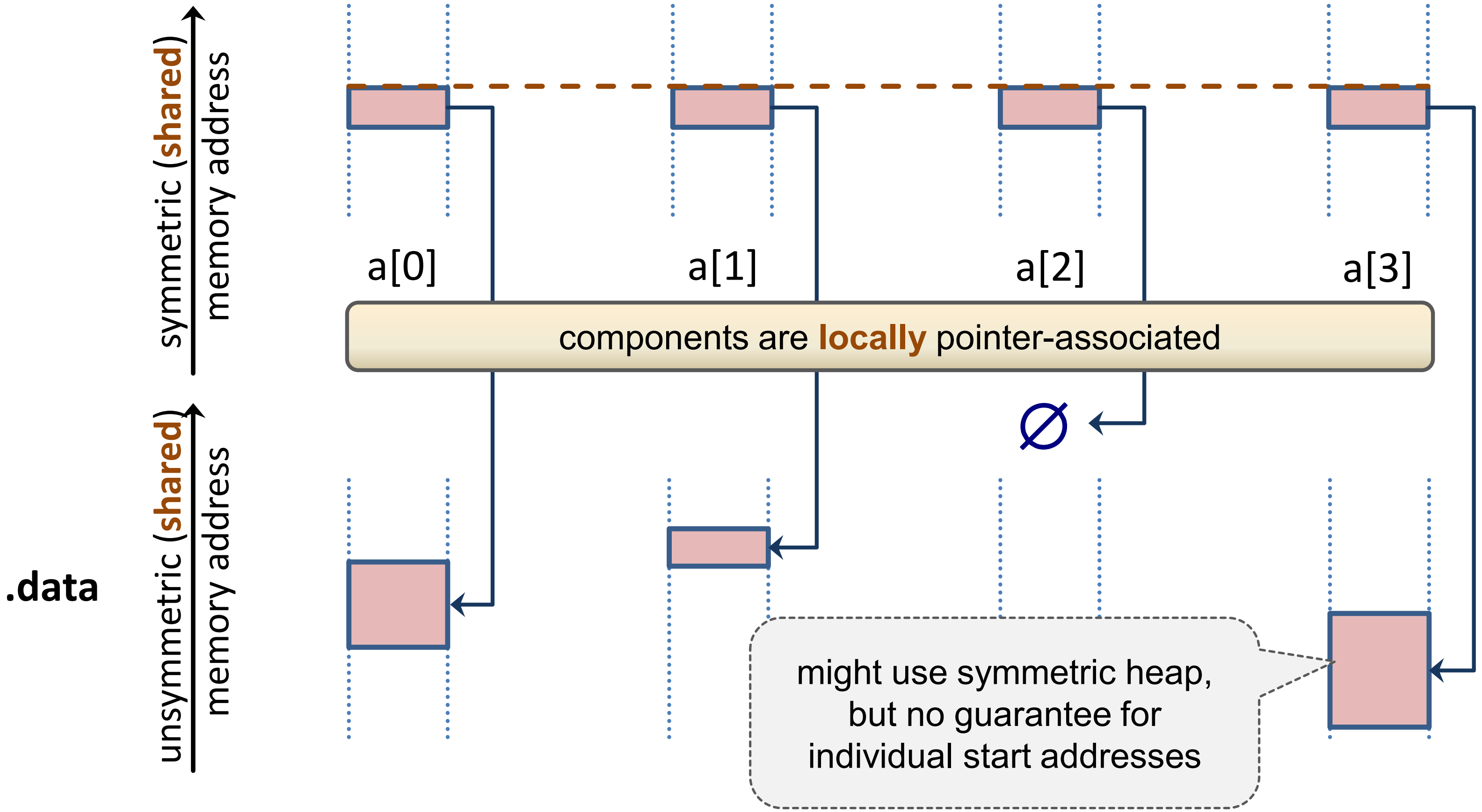
Performance impact:

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



UPC: unsymmetric objects

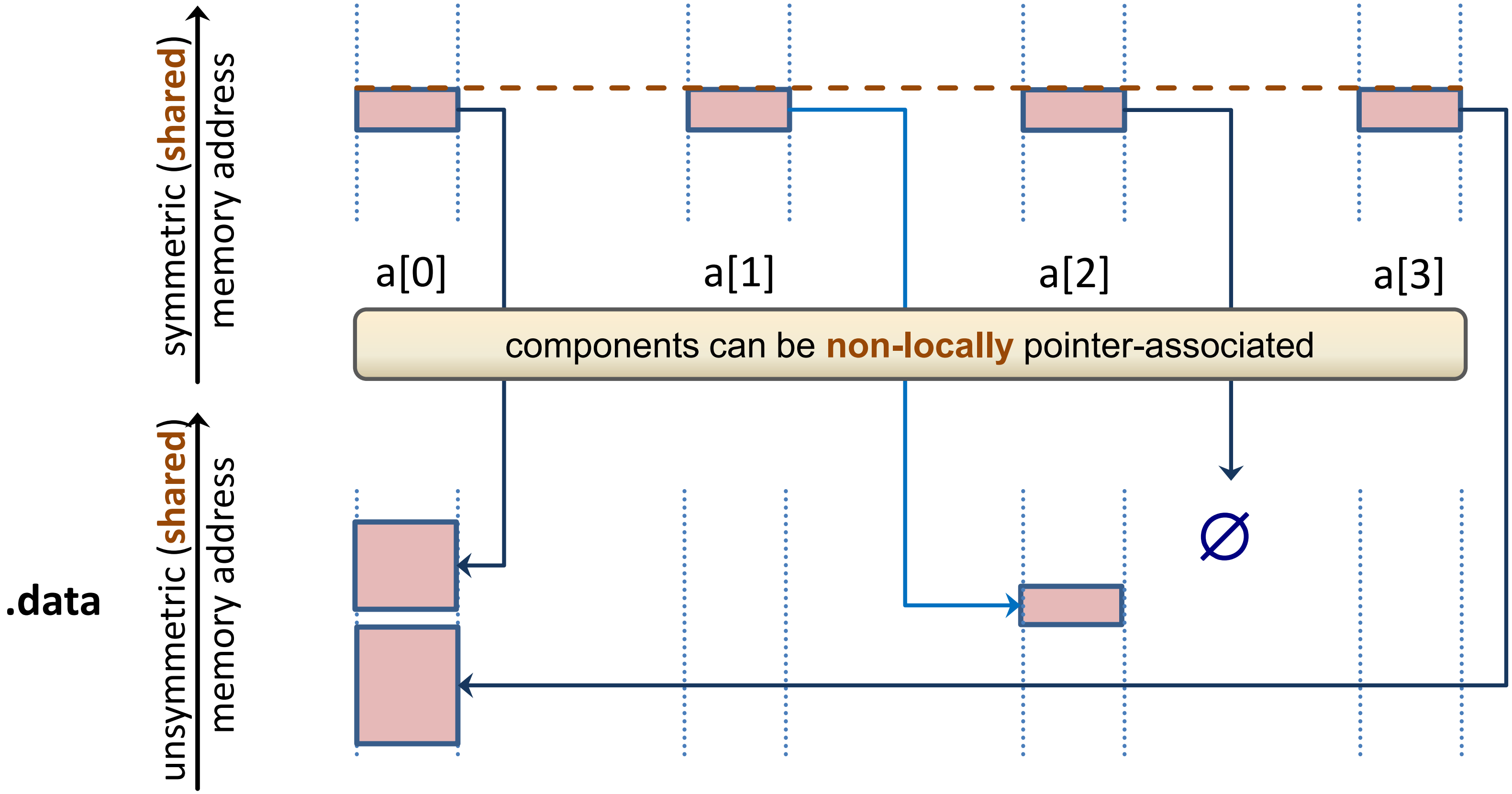
- **Variant 1 for data layout – locality consistent with parent object**



- programmer establishes the locality convention

UPC: unsymmetric objects

- **Variant 2 for data layout – arbitrary locality**



- for example, execute the following on thread 2

```
a[1].data = upc_alloc(n*sizeof(float));
```

Setup – Local processing – Data exchange

CAF

```
real, allocatable, &
      target :: field(:,:)
allocate(field(NR, NC))
: ! determine column
a % data => field(:, column)
```

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

```
sync all
q = … ! some other image
n = size( a[q] % data, 1 )
call process(field, …)
```

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

```
sync all
localdata(:n) = … + a[q] % data
```

UPC (using variant 1)

```
shared [] float *field;
field = upc_alloc(
           NR*NC*sizeof(float) );
: ! determine column
a[MYTHREAD].data =
          &field[NR*column];

upc_barrier;
q = … ! some other thread
n = … ! size of remote column
process( (float *) field, …);

upc_barrier;
upc_memget(aux, a[q].data,
           n * sizeof(float) );
```

then, update **`localdata`** using local buffer **`aux`**

Note that NR and NC might vary between images

© 2010-19 LRZ

CAF: Some limitations on intrinsic assignment

POINTER components

- shallow copy semantics may conflict with locality requirement

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

```
a[q] = container(field(:,1))
```

```
type :: polynomial
  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 corresponding coarray dummy
  - in an association block context

Part 3a: Data layout and processing

CAF corank-image mapping

UPC locality intrinsics

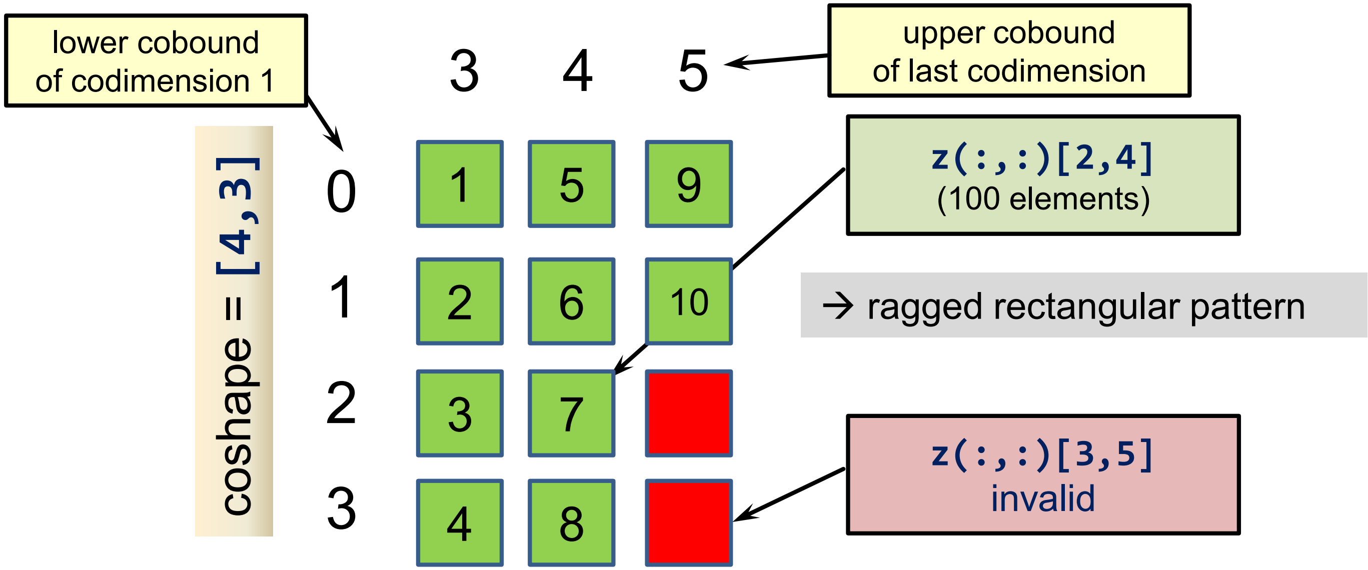
UPC global view and upc_forall

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**



lower cobound of codimension 1

upper cobound of last codimension

coshape = [4,3]

	3	4	5
0	1	5	9
1	2	6	10
2	3	7	🟥
3	4	8	🟥

`z(:,:)[2,4]`
(100 elements)

→ ragged rectangular pattern

`z(:,:)[3,5]`
invalid

Supporting intrinsics

- **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 (local) coindex; zero for invalid coindex.

e.g., for later use in synchronization statements

- **Examples for "z" declared previously**

```
cindx = this_image( z )
```

on image 7, returns `[2,4]`

```
m1   = this_image( z, 1 )
```

on image 7, returns `2`

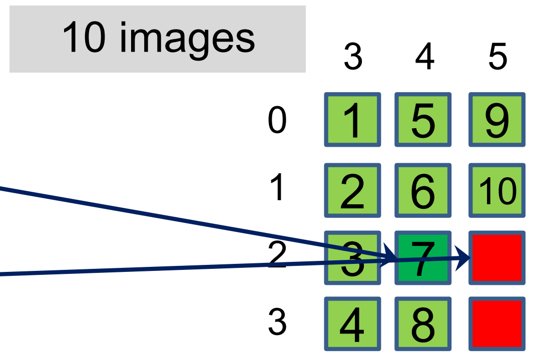
```
img = image_index( z, [2,4] )
```

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
```



UPC: Locality and phase

- **May require assurance about where a subobject is located**
  - e.g., to avoid cross-thread accesses

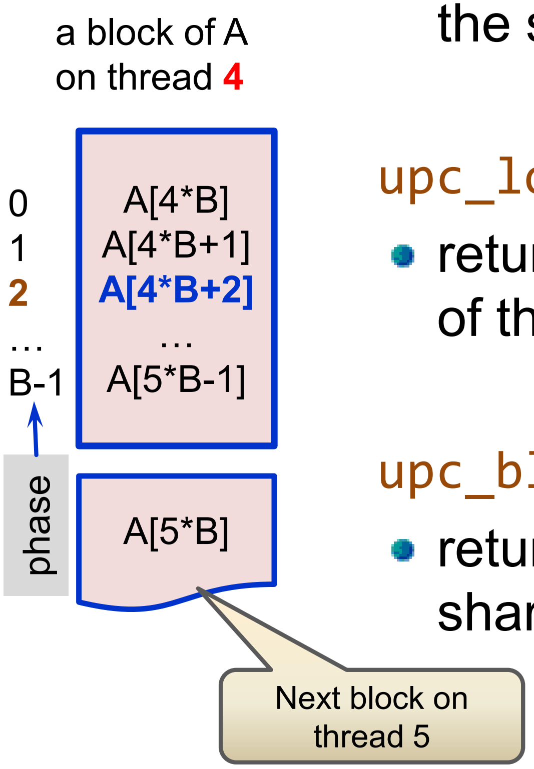
assuming B > 2

```
shared [B] int A[N];
size_t thr, pos;

thr = \
  upc_threadof(&A[4*B+2]);
```

on **any** thread, returns **4**

```
pos = \
  upc_phaseof(&A[4*B+2]);
```

on **any** thread, returns **2**



- **Further intrinsics:**

`upc_elemsizeof(object)`

- returns size of an element of the shared object in bytes

`upc_localsizeof(object)`

- returns size of the local part of the shared object in bytes

`upc_blocksizeof(object)`

- returns blocking factor of the shared object

UPC: Processing global data

- **Fragmented data**
  - requires code restructuring (e.g. for loop processing)
- **UPC supports global data**
  - locality to a thread is implicit
- **Global loop processing:**
  - **`upc_forall`** integrates data affinity to threads with loop construct
  - must be collectively executed by all threads
  - fourth argument is an **affinity expression** that controls which subset is executed

- **Example: matrix initialization**

distribute columns round-robin

```
shared [N] float Mat[N][N];

upc_forall (icol=0; icol<N;
                    icol++; icol) {
  for (irow=0;irow<N;irow++) {
    Mat[icol][irow] =
           matval(irow+1,icol+1);
  }
}
```

global indexing retained

  - MYTHREAD only executes that subset of iterations with **`icol%THREADS == MYTHREAD`**
  - effect: all assignments are thread-local

Affinity expressions in upc_forall

Type of affinity expression	Iterations of loop executed on MYTHREAD
integer i	with `i%THREADS == MYTHREAD`
shared pointer *x	with `upc_threadof(x) == MYTHREAD`
"continue" or empty	all iterations. In this case, collective execution is not required

- **In the example,** using

```
shared [N] float Mat[N][N];
upc_forall (icol=0; icol<N; icol++; &Mat[icol][0]) { … }
```

  would have the equivalent effect
- **Note:**
  - multiple shared entities with incommensurate block sizes inside code block might perforce lead to non-local accesses / communication

Part 3b:
Collective Procedures

Note:

In Fortran, collectives were added by TS18508

Currently, they are not yet generally supported

Motivation

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

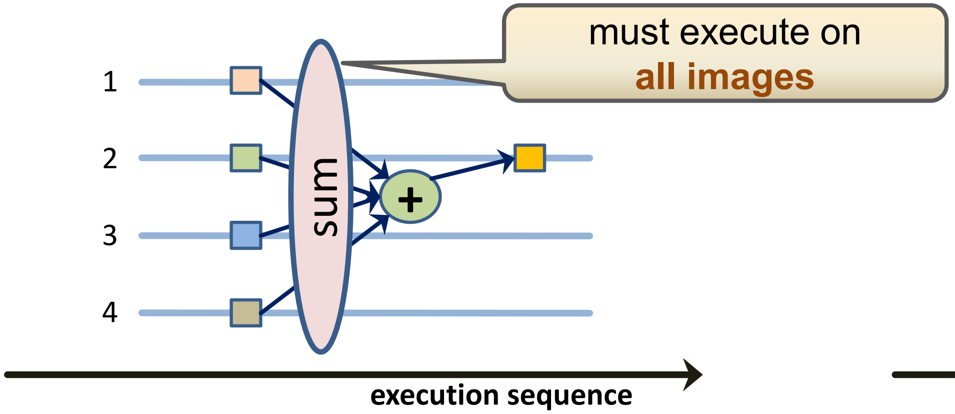
```
real :: mass(ndim,ndim), velocity(ndim,ndim)
real :: e_kin
:
e_kin = 0.5 * sum( mass * velocity**2 )
```

In C, you don't even have those ... so need to roll your own.

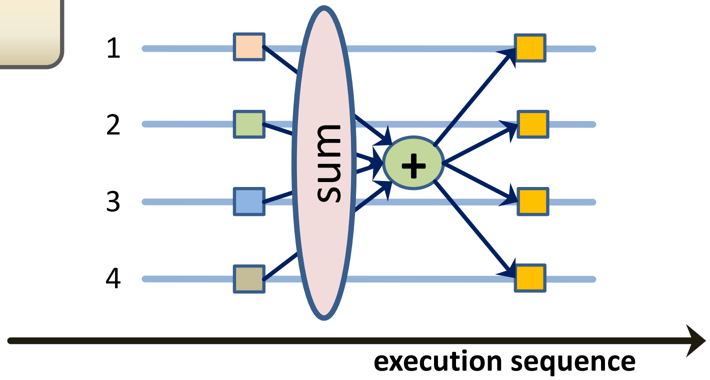
- **Coarray / UPC code:**
  - on each image, an image-dependent **partial sum** is evaluated
  - i. e. the intrinsic is not image-aware
- **Variables that need to have the same value across all images**
  - e.g. global problem sizes
  - values are initially often only known on one image

Classification

- **Collectives that perform a computation**

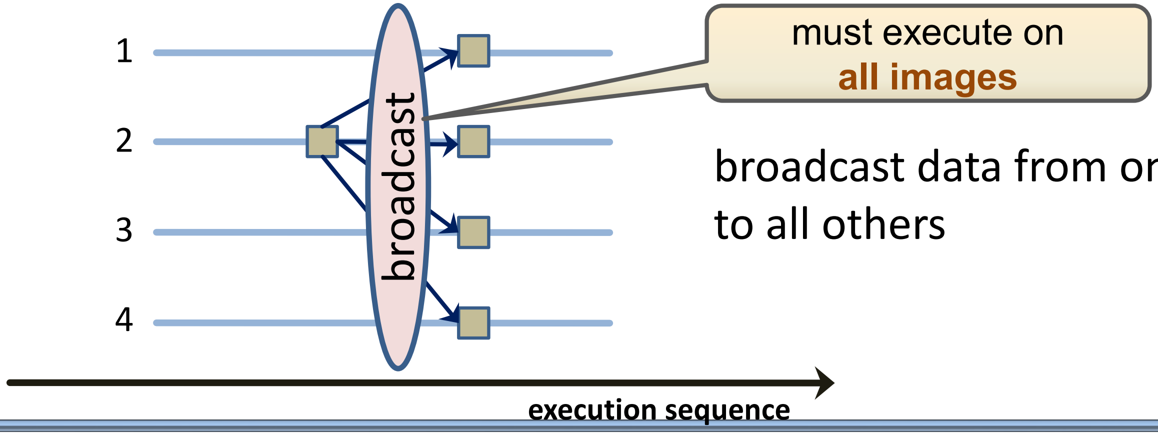


reduction with result on one image



reduction with result on all images

- **Collectives that re-localize data**



broadcast data from one image
to all others

General properties

Both CAF and UPC

- Collectives must be invoked by all images, and from unordered segments, to avoid deadlocks

CAF

- Data arguments need **not** be coarrays – however if a coarray is supplied, it must be the same (ultimate) coarray on all images
- No segment ordering is implied by execution of a collective – valid result data on exit
- All collectives are "in-place" – programmer needs to copy data argument if original value is still needed

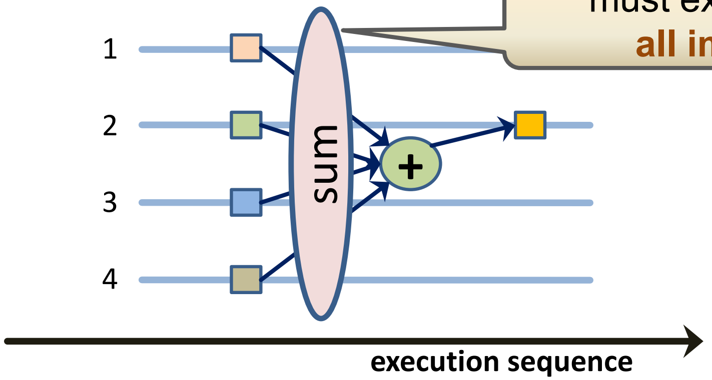
UPC

`#include <upc_collective.h>`

- Data arguments are **always** shared entities
- Programmer must specify whether synchronization is performed
- Separate „source“ and „destination“ arguments, which are not allowed to be aliased (undefined behaviour)

Collectives could of course be implemented by the programmer. However it is expected that the supplied ones **will perform better**, apart from being more generic in semantics.

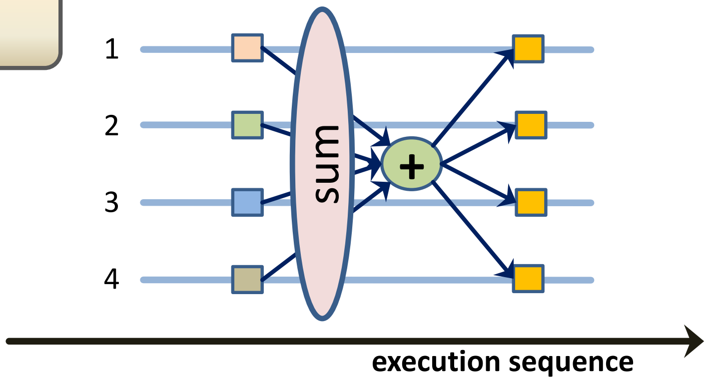
CAF Reductions: CO_SUM, CO_MAX, CO_MIN



must execute on
all images

```
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

CAF 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

UPC Reduction prototype

```
void upc_all_reduce<<T>>(
  shared void *restrict dst,
  shared const void *restrict src,
  upc_op_t op,
  size_t nelems,
  size_t blk_size,
  <<TYPE>>(*func)(<<TYPE>>, <<TYPE>>),
  upc_flag_t flags);
```

destination and source, respectively

number of elements of specified type

source pointer block size.
Permits to respect phase:

`shared [blk_size] TYPE *src[nelems]`

- replace `<<T>>` by type specifier (C, UC, etc., see next slide)
- function argument will be `NULL` unless user-defined
- reduction function is specified through `op`
- synchronization is specified through `flags`

UPC Reductions:
Supported types and operations

Reduction types

- encoded as part of the function name → 11 variants per function

T	TYPE	T	TYPE
C/UC	signed char/ unsigned char	L/UL	signed long/ unsigned long
S/US	signed short/ unsigned short	F/D/LD	float/double/long double
I/UI	signed int/unsigned int		

- note that only intrinsic types are supported

Operations:

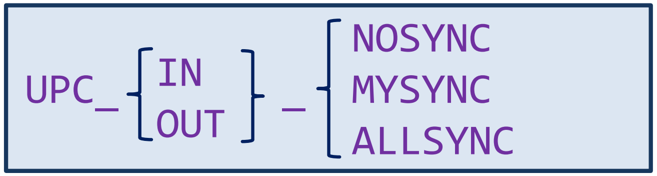
Numeric	Logical	User-defined function
UPC_ADD	UPC_AND	UPC_FUNC
UPC_MULT	UPC_OR	UPC_NONCOMM_FUNC
UPC_MAX	UPC_XOR	
UPC_MIN	UPC_LOGAND	
	UPC_LOGOR	

- are constants of type `upc_op_t`

UPC collectives: specifying synchronization

Synchronization mode

- constants of type `upc_flag_t` in `upc_collectives.h`



UPC_ {IN, OUT} _ {NOSYNC, MYSYNC, ALLSYNC}

IN/OUT

- refers to whether the specified synchronization applies at the **entry** to or **exit** from the call

Relaxing synchronization

- programmer‘s responsibility to assure that no race conditions occur
- typically used for multiple reductions on disjoint variables

Synchronization semantics

- NOSYNC – threads do not synchronize at entry or exit
- MYSYNC – start processing of data only if owning threads have entered the call / exit function call only if all local read/writes are complete
- ALLSYNC – synchronize all threads at entry / exit

Combining modes

- `UPC_IN_NOSYNC | UPC_OUT_MYSYNC`
- `UPC_IN_NOSYNC` same as `UPC_IN_NOSYNC | UPC_OUT_ALLSYNC`
- `0` same as `UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC`

UPC Example use

```
shared double cc[THREADS];
shared double res[THREADS];
shared [0] double cc_0[THREADS];
shared [0] double res_0;

int main () { // initializations omitted
```

reduction that includes a broadcast to multiple result variables `res`

```
  upc_all_reduceD(&res,cc,UPC_ADD,THREADS,1,NULL,UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC);
  printf("Reduce variant 1 - Thread %i: %12.4f\n", MYTHREAD, (double) *res);
```

reduction to a localized result variable `res_0`

```
  upc_all_reduceD(&res_0,cc,UPC_ADD,THREADS,1,NULL,UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC);
// broadcast to a local scalar
  rl = *res;
  printf("Reduce variant 2 - Thread %i: %12.4f\n", MYTHREAD, rl);
```

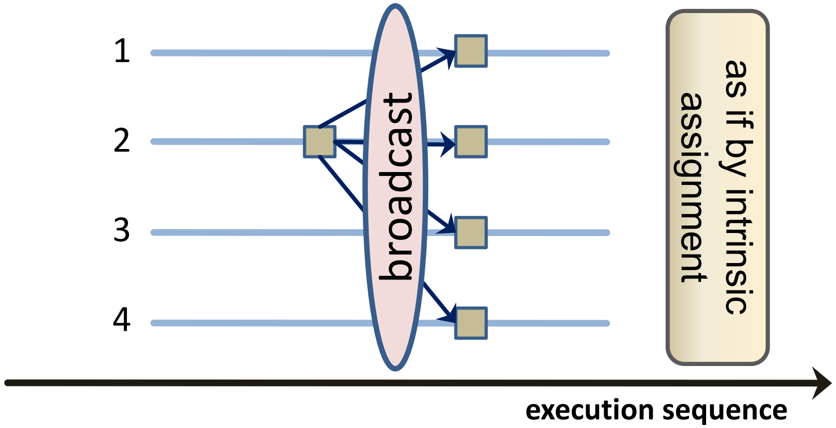
reduction from a localized source variable `cc_0`

```
  upc_all_reduceD(&res,cc_0,UPC_ADD,THREADS,0,NULL,UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC);
  printf("Reduce variant 3 - Thread %i: %12.4f\n", MYTHREAD, (double) *res);
}
```

- **Array reductions are not supported**

© 2010-19 LRZ

CAF: Data redistribution with CO_BROADCAST



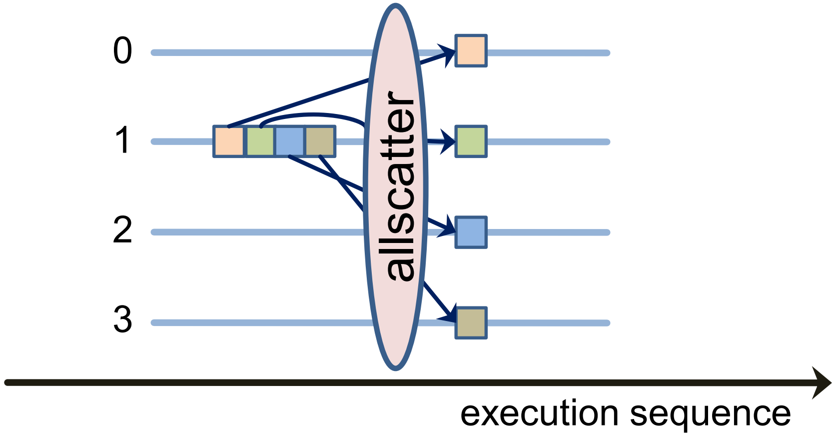
```
type(matrix) :: xm
:
call co_broadcast(a=xm, source_image=2)
```

- **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()

UPC: Example for redistribution

UPC Allscatter

```
void upc_all_scatter (
     shared void *dst,
     shared const void *src,
     size_t nbytes,
     upc_flag_t sync_mode);
```



- i-th block of `src` with size **`nbytes`** is copied to `dst` with affinity to thread i
- each block in `src` must have affinity to a single thread

Further UPC collectives

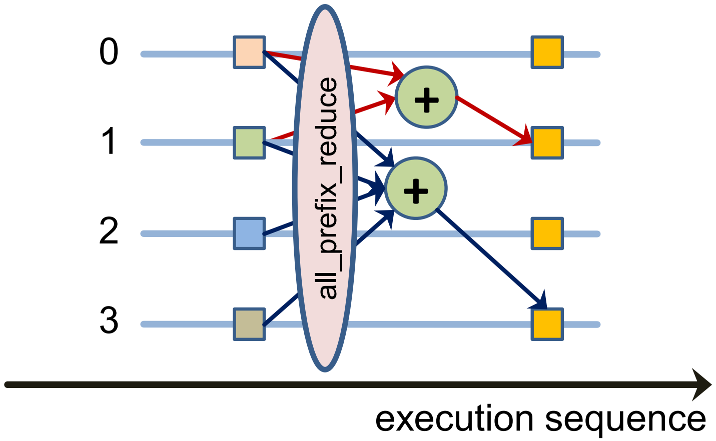
Redistribution functions

- upc_all_broadcast()
- upc_all_gather_all()
- upc_all_gather()
- upc_all_exchange()
- upc_all_permute()

→ **consult the UPC language specification for details**

Prefix reductions

- upc_all_prefix_reduce**T**()
- semantics:



for UPC_ADD,

object `dst[i]` hosted on thread i gets $\sum_{k=0}^{i} d_k$

(thread-dependent result)

Part 4a: Advanced Synchronization Concepts

Partial synchronization

One-sided synchronization

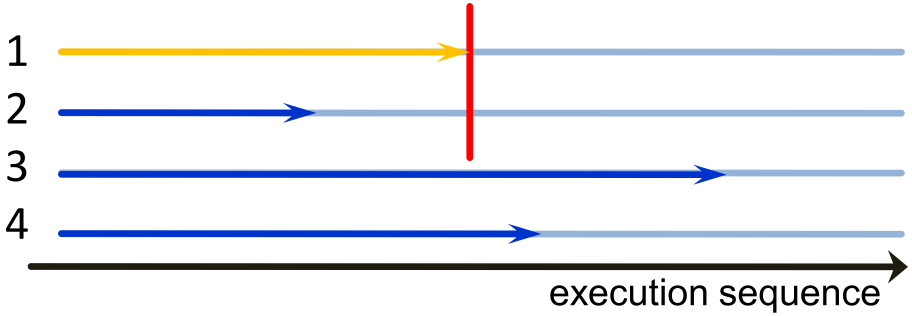
Mutual exclusion (locks)

UPC: split phase barrier and memory consistency

Partial synchronization

Image subsets

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



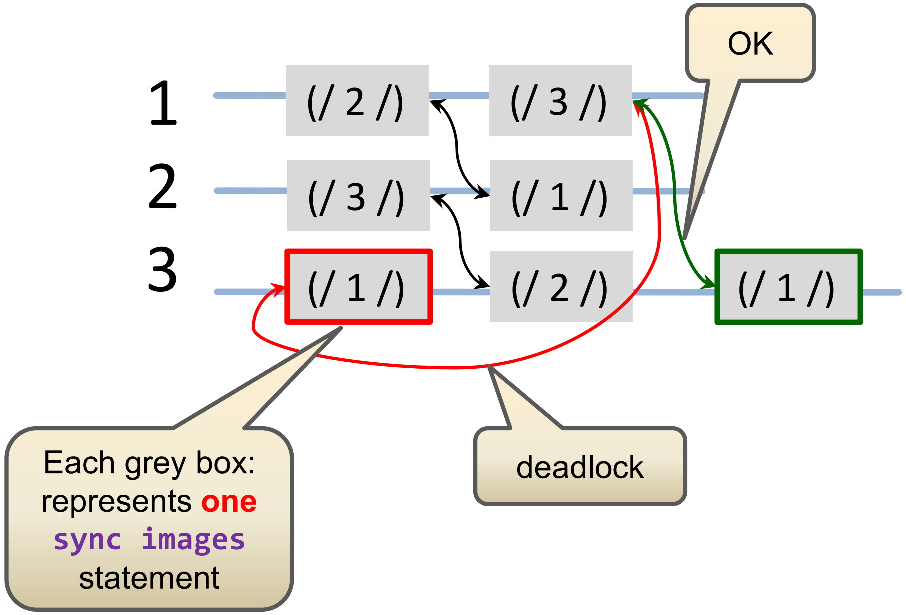
- synchronization statement:

```
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
- ordering can be relevant:



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
```

- „Push" / "Put" mode

  an optimizing implementation 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
```

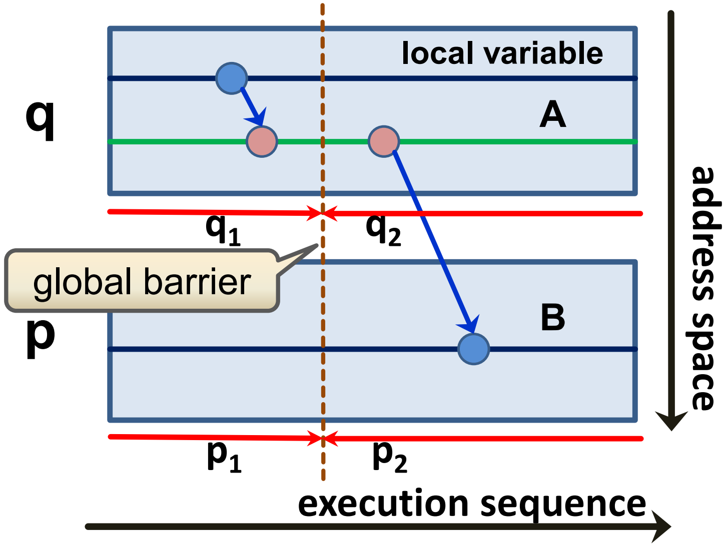
`sync images` is not context-safe

  - likely to produce wrong results even if no deadlock occurs

© 2010-19 LRZ

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**

concept applies for UPC also!

```
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**

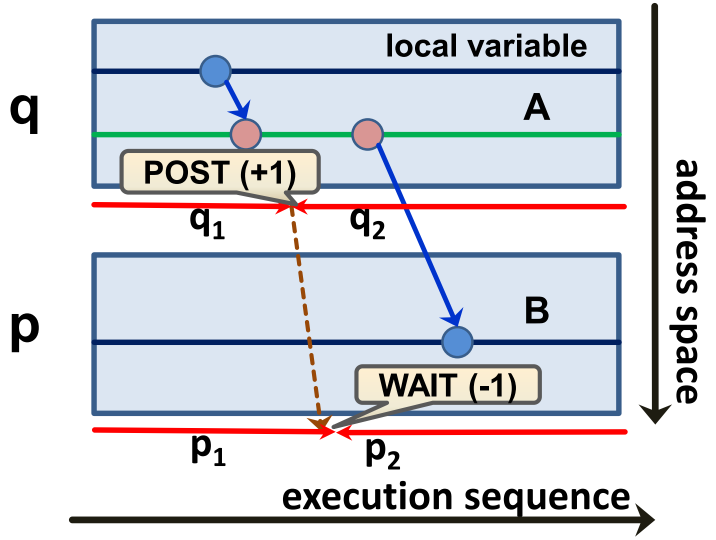
```
event wait ( ev )
b = a(:)[q]
```

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 )
```

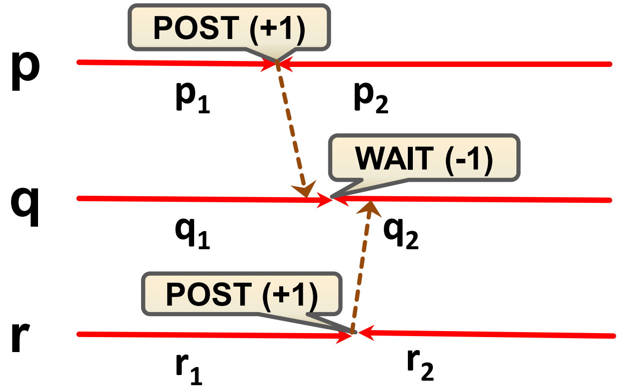
  - Image **r** executes

```
event post ( ev[q] )
```

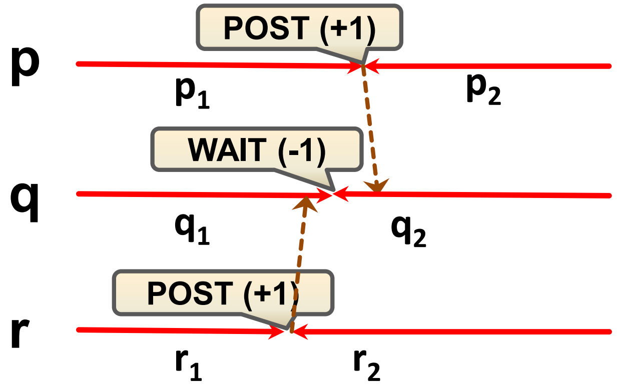
- **Question:**
  - what synchronization effect results?
- **Answer: 3 possible outcomes**
  - which one happens is **indeterminate**

Avoid over-posting from multiple images!

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



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

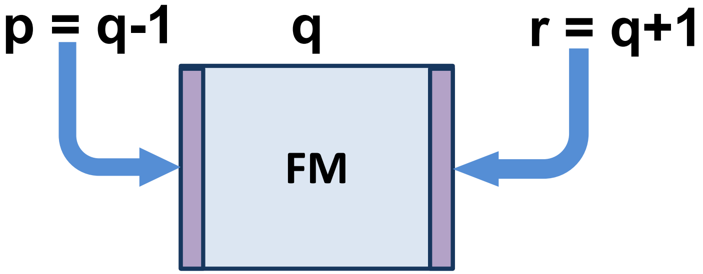


- **Case 3: ordering as given on next slide**

© 2010-19 LRZ

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

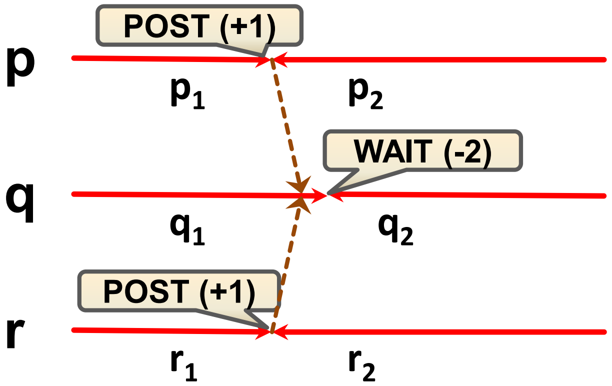
```
fm(:,n)[q] = …
event post ( ev[q] )
```

  - 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

The EVENT_QUERY intrinsic

- **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
    (note that the actual count may **change** before you can inspect the result!)
  - the facility can be used to implement non-blocking execution on the WAIT side of event processing
  - invocation has **no** synchronizing effect

© 2010-19 LRZ

Berkeley UPC extension: Semaphores

Optional slide

Setting up a semaphore

```
#include <upc_sem.h>
upc_sem_t *shared ev[THREADS];
int flags;

flags = …;
ev[MYTHREAD] = upc_sem_alloc(flags);

: // use ev for synchronization

upc_barrier;
upc_sem_free(&ev[MYTHREAD]);
```

non-collective

Possible flag values

Value	Semantics
`UPC_SEM_[BOOLEAN,INTEGER]`	binary vs. counted semaphore
`UPC_SEM_[S,M]PRODUCER`	increment by only one thread or by all threads
`UPC_SEM_[S,M]CONSUMER`	decrement by hosting thread or by all threads

- entries along rows can be combined
- for example,

```
flags = UPC_SEM_INTEGER | UPC_SEM_MPRODUCER | UPC_SEM_SCONSUMER;
ev[MYTHREAD] = upc_sem_alloc(flags);
```

supplies semantics equivalent to Fortran‘s events

© 2010-19 LRZ

Using the semaphore for one-sided synchronization

Optional slide

- Single-post

```
// thread q executes
p = …;
a[p] = …;
upc_sem_post( ev[p] );

// thread p executes
upc_sem_wait( ev[MYTHREAD] );
… = a[MYTHREAD];
```

- Non-blocking wait

```
// thread q does the same as above
…
// thread p executes
for (;;) {
  if (upc_sem_try( ev[MYTHREAD] ))
     break;
  : // do something unrelated to 'a'
}
… = a[MYTHREAD];
```

- Multiple-post

```
// thread q executes
p = …;
a[p] = …;
upc_sem_post( ev[p] );

// thread r executes
p = …;
b[p] = …;
upc_sem_post( ev[p] );

// thread p executes
upc_sem_waitN( ev[MYTHREAD], 2 );
… = a[MYTHREAD] + b[MYTHREAD];
```

For details, read **upc_sem.pdf**

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 parallel 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

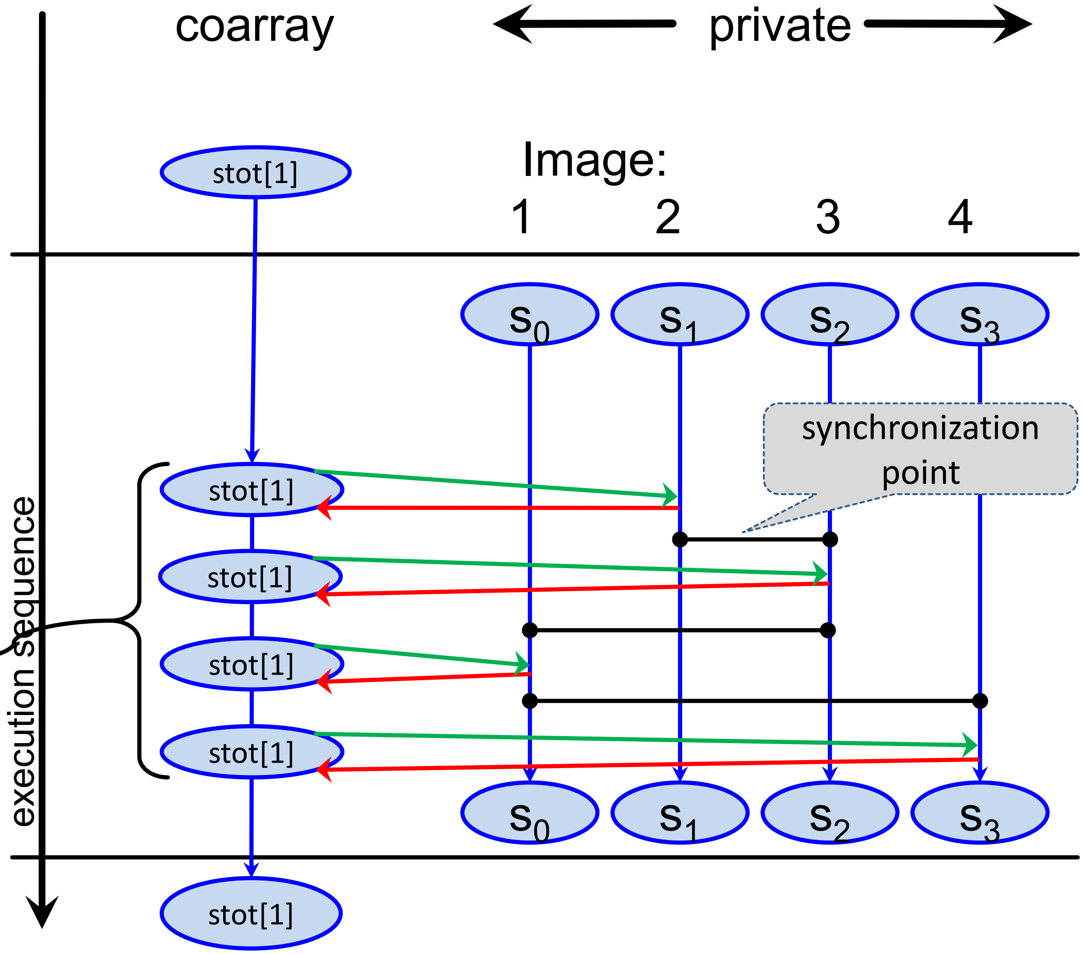
Example for mutual exclusion via a critical region

```
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



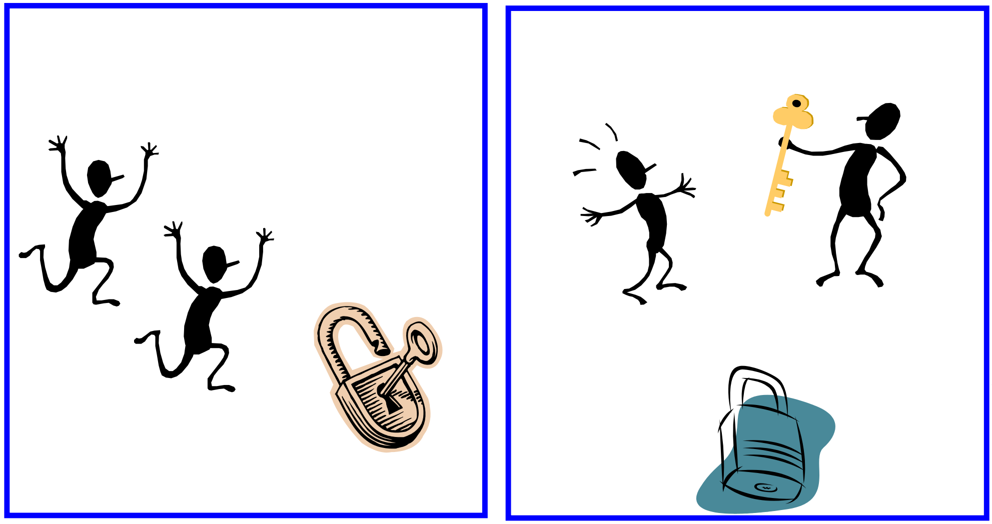
- **Only one image at a time can execute the critical region**
  - others must wait → code in region is effectively serialized

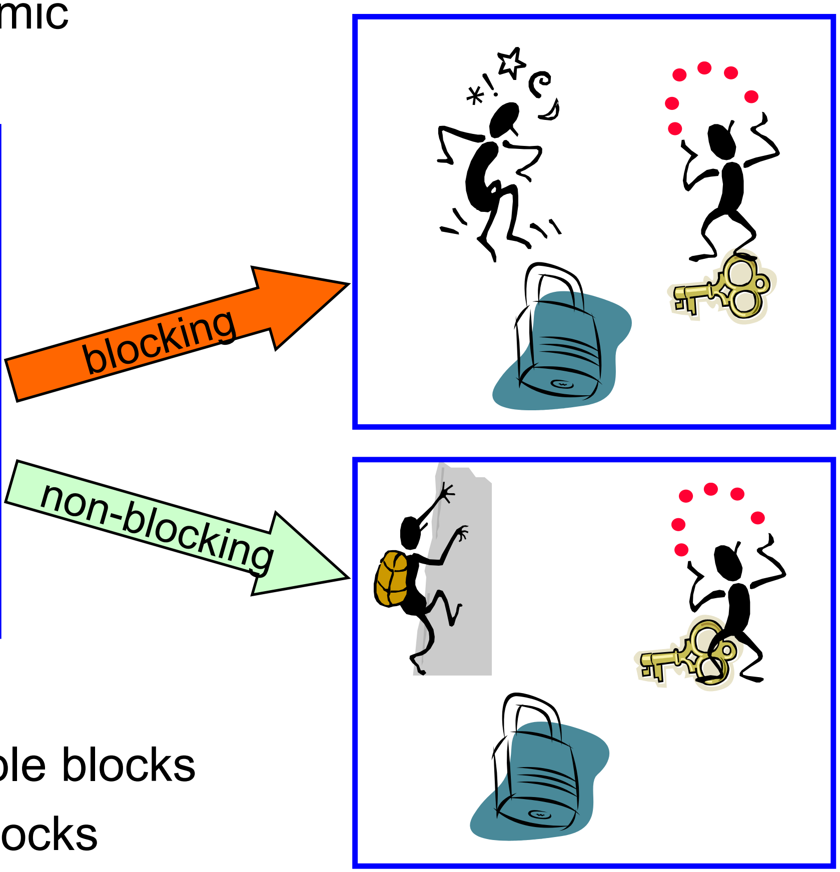
Locks – a more flexible mechanism for mutual exclusion

- **Coordinate access to shared (= sensitive) data**

„red balls“

- **Use a coarray/shared lock variable**
  - modifications are guaranteed to be atomic
  - consistency across images/threads





- **Problems with CAF critical region:**
  - objects may require protection in multiple blocks
  - different objects protected by different locks
    → improved scalability

Declaration and initial state

CAF

```
use, intrinsic :: iso_fortran_env

type(lock_type) :: my_lock[*]
```

default initialized to the "unlocked" state

Lock variable:

- must be a coarray (here, this implies one lock per image!)
- two states - unlocked or locked
- locked means: acquired by a specific image (until that image releases the lock again)

UPC

```
#include <upc.h>

upc_lock_t *lock;

lock = upc_all_lock_alloc();
: // do stuff with lock
if (MYTHREAD == 0)
   upc_lock_free(lock);
```

collective call
same result on each thread

Lock variable:

- typically, one or more pointers to a single shared object (included in type)
- explicit setup and teardown required
- otherwise, like CAF

Simplest example for blocking locks

- **CAF**

Example works analogous
to a CRITICAL region

```
use, intrinsic :: iso_fortran_env
type(lock_type) :: my_lock[*]
:
rb = …
sync all

lock( lock[1] )
  i = …
  rb[i] = rb[i] + …
unlock( lock[1] )

sync all
: ! access rb
```

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/unlock: no memory fence, only **one-way** segment ordering

Quiz: why image 1 in the example?

- **UPC**

```
#include <upc.h>
upc_lock_t *lock;
:
lock = upc_all_lock_alloc();
rb = …;
upc_barrier;

upc_lock( lock );
  i = …;
  rb[i] = rb[i] + …;
upc_unlock( lock );

upc_barrier;
: ! access rb
if (MYTHREAD == 0)
   upc_lock_free(lock);
```

might also be needed to prevent race against teardown

- lock/unlock imply memory fence

Non-blocking lock semantics

- **CAF:**

```
use, intrinsic :: iso_fortran_env

type(lock_type) :: nb_lock[*]
logical :: got_it

activity : do
  lock( nb_lock[1], &
        acquired_lock=got_it )
  if ( got_it ) exit activity
  : ! go climb that mountain
end do activity
: ! play with red balls
unlock( nb_lock[1] )
```

Always continues.
Result is **true** if
lock was acquired.

- **UPC:**

```
#include <upc.h>
upc_lock_t *nb_lock;

nb_lock = upc_all_lock_alloc();

for (;;) {

  if (upc_lock_attempt( nb_lock ))
       break;
  : // go climb that mountain
}
: // play with red balls
upc_unlock( nb_lock );

upc_all_lock_free( nb_lock );
```

collective teardown (UPC 1.3)
includes barrier

potentially needed explicit barriers are omitted here

Locks – an expensive synchronization mechanism

- **Best case timing for lock acquisition**

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

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)
- **Advice:**
  - prefer use of events for synchronization (where possible)

CAF: 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
```

UPC: lock components

- **Establish a component inside a struct definition**

```
typedef struct queue {
  upc_lock_t *lock;
  shared work_item *work;
  shared queue *next;
}

typedef shared struct queue Queue;
```

all objects are shared

- **Constructor for a Queue object** (called on a per-thread basis)

```
Queue *Queue_add ( Queue *in, work *w ) {
  Queue *this;
  : // establish "this"
  this->lk = upc_global_lock_alloc();
  : // upc_memput w to this->work (after locking)
}
```

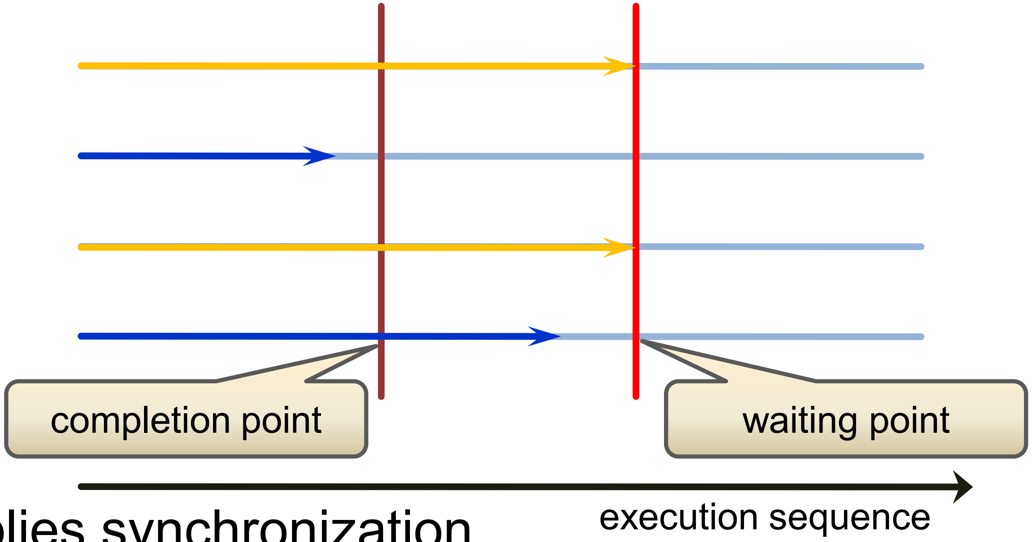
non-collective lock allocation

lock is needed for modifying the queue

UPC: split-phase barrier

- **Separate barrier completion point from waiting point**
  - this allows threads to continue computations once all others have reached the completion point → may reduce impact of load imbalance

```
for (…) a[n][i]= …;
upc_notify;
// do work (on b?) or comm.
// not involving a
upc_wait;
for (…) b[i]= b[i]+a[q][i];
```



  - completion of `upc_wait` implies synchronization
  - collective – **all** threads must execute sequence
- **CAF:**
  - presently does not have this facility in statement form

    (one can implement this concept using events)

© 2010-19 LRZ

UPC: Memory consistency modes

Optional slide

How are shared entities accessed?

- relaxed mode → program **assumes** no concurrent accesses from different threads
- strict mode → program **ensures** that accesses from different threads are separated, and **prevents** code movement across these synchronization points
- relaxed is default; strict may have **large** performance **penalty**

Options for synchronization mode selection

- variable level:

  (at declaration)

```
strict shared int flag = 0;
relaxed shared [*] int c[THREADS][3];
```

example for a spin lock

Thread q

```
c[q][i] = …;
flag = 1;
```

Thread p

```
while (!flag) {…};
… = c[q][j];
```

q has same value on thread p as on thread q

- code section level:

```
{ // start of block
  #pragma upc strict
  … // block statements
}
// return to default mode
```

  – program level

```
#include <upc_strict.h>
// or upc_relaxed.h
```

consistency mode on variable declaration **overrides** code section or program level specification

What strict memory consistency does and doesn‘t do for you

Optional slide

- **„strict“ cannot prevent all race conditions**
  - example: „ABA“ race

```
strict shared int flag;
int val, val1, val2;
```

thread 0

```
flag = 0;
upc_barrier;
flag = 1;
flag = 0;
```

thread 1

```
upc_barrier;
val = flag;
```

may end up with 0 or 1

- **„strict“ does not** make a[i]+=j atomic (read/modify/write)
- **„strict“ does assure** that changes on (complex) objects appear in the same order on other threads

thread 0

```
flag = 0;
upc_barrier;
flag = 1;
flag = 2;
```

thread 1

```
upc_barrier;
val1 = flag;
val2 = flag;
```

may obtain (val1 <= val2)
but **not** (val1 > val2)
e.g., (2, 1) or (2,0) are not possible

Part 4b: PGAS programming scenarios

Optional Interaction with OO semantics

Library Design:

Subprogram interfaces

Factory procedures

PGAS and MPI programming

Using coarrays together with object-oriented features

- Shaky ground due to implementation issues
- Limited semantics

Combining coarrays with object orientation

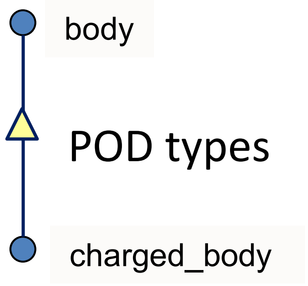
Optional slide

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

Optional slide

- **Coindexing is not permitted:**

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

Not permitted

```
select type(particles[2])
  :
end select
```

```
associate(p => asteroids[2])
  p = …
end associat
```

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

Optional slide

- **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

Optional slide

```
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)

```
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

Optional slide

- **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

Shared objects as dummy arguments

- **Library codes may need**
  - to communicate and synchronize argument data

  → declare dummy arguments as coarrays / pointers to shared
- **Preserve ability for exchanging data between images**
  - implies that data must not be copied when calling a procedure
  - 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
    - the `VALUE` attribute is prohibited
    - a coarray descriptor might be copied
  - UPC shared data:
    - private pointers to shared might be copied, but not shared-to-shared

Shared dummy interface

CAF

- an **explicit interface** is required for using coarray dummy arguments

```
subroutine subr(n,w,x,y)
  integer :: n
  real :: w(n)[n,*]
  real :: x(n,*)[*]
  real :: y(:,:)[*]

  : ! local computations
  sync all
  : ! exchange data
  sync all
  : ! etc
end subroutine
```

explicit shape

assumed size

assumed shape

F18 updating a coarray dummy through coindexing is permitted (exception to aliasing rules)

UPC

```
void subr(int n,
          shared float *w) {
  int i;
  float *wloc;
  wloc = (float *) &w[MYTHREAD];
  for (i=0; i<n; i++){
     … = wloc[i] + …
  }
  upc_barrier;
  // exchange data
  upc_barrier;
  // etc.
}
```

- assumes local size is n
- cast to local pointer for safety of use **and performance** if only local accesses are required
- declarations with *fixed* block size > 1 also possible (default is 1, as usual)

© 2010-19 LRZ

Calling the procedure

CAF

```
real :: a(ndim)[*], b(ndim,2)[*]
real, allocatable :: c(:,:,:)[:]
allocate(c(10,20,30)[*])
: ! initialize a, b, c
call subr(ndim, a, b, c(1,:,:))
```

- actual argument **must** be a coarray if the dummy is
- argument **a**: corank mismatch is permitted. Inside the procedure, coindices are remapped.

recommendation: avoid image-dependent cobounds

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

UPC

```
shared [*] float x[THREADS][NDIM]
int main(void) {
  : // initialize x
  upc_barrier;
  subr(NDIM, (shared float *) x);
}
```

- cast to cyclic to match the prototype
- this approach of passing cyclic pointer and block size as arguments is a common solution to UPC library design.
- cyclic is “good enough” in most cases because function can recover actual layout via pointer arithmetic
- in this example w[i] aliases x[i][0]

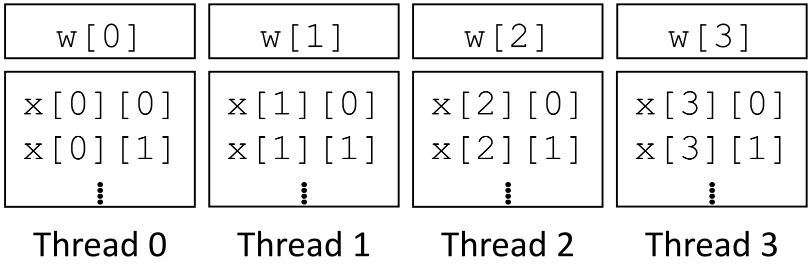


Image-dependent coarray arguments

Example CAF 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:

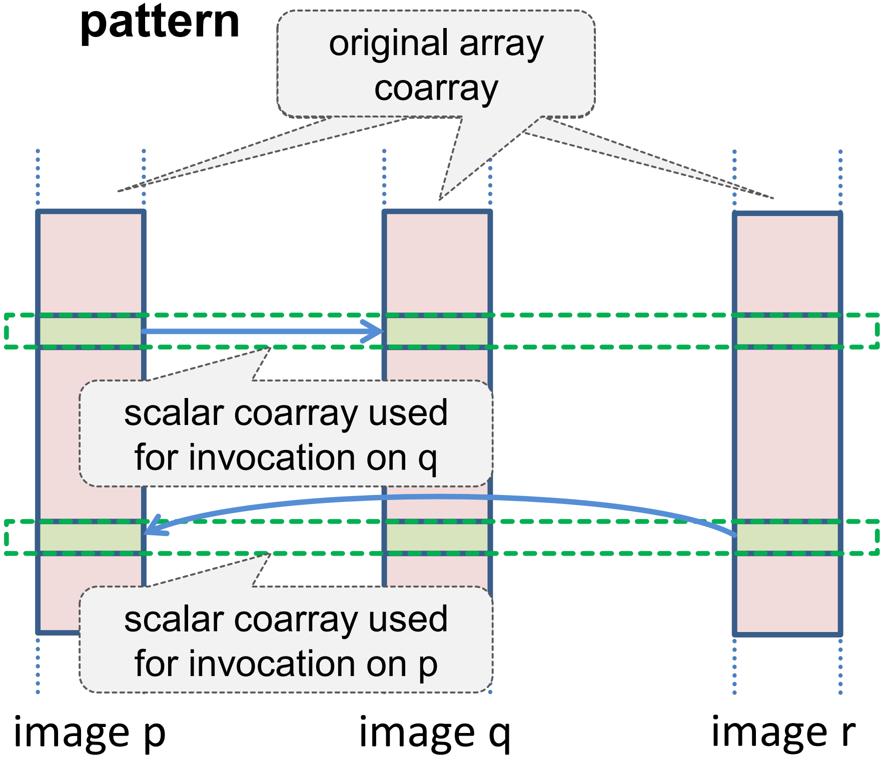
- with a different coarray (subobject) on each image

```
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



original array coarray

scalar coarray used for invocation on q

scalar coarray used for invocation on p

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

Image-dependent shared object passing

UPC version

```
void add_pivot(shared float *x,
         float y[], int n) {
  y[n] = y[n] + *x;
}
```

- with invocation

```
shared float x[NDIM][THREADS];
int main() {
  int p = …; int n = …;
  : // initialize x
  upc_barrier;
  add_pivot( &x[n][p], (float *) x, n);
}
```

p /= MYTHREAD, n and **p** are different on each thread

Beware:

- if synchronization is done within a procedure, all images must execute a consistent sequence of synchronizations
- else, deadlocks or data races will result

CAF: 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 shared objects

- **CAF 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!

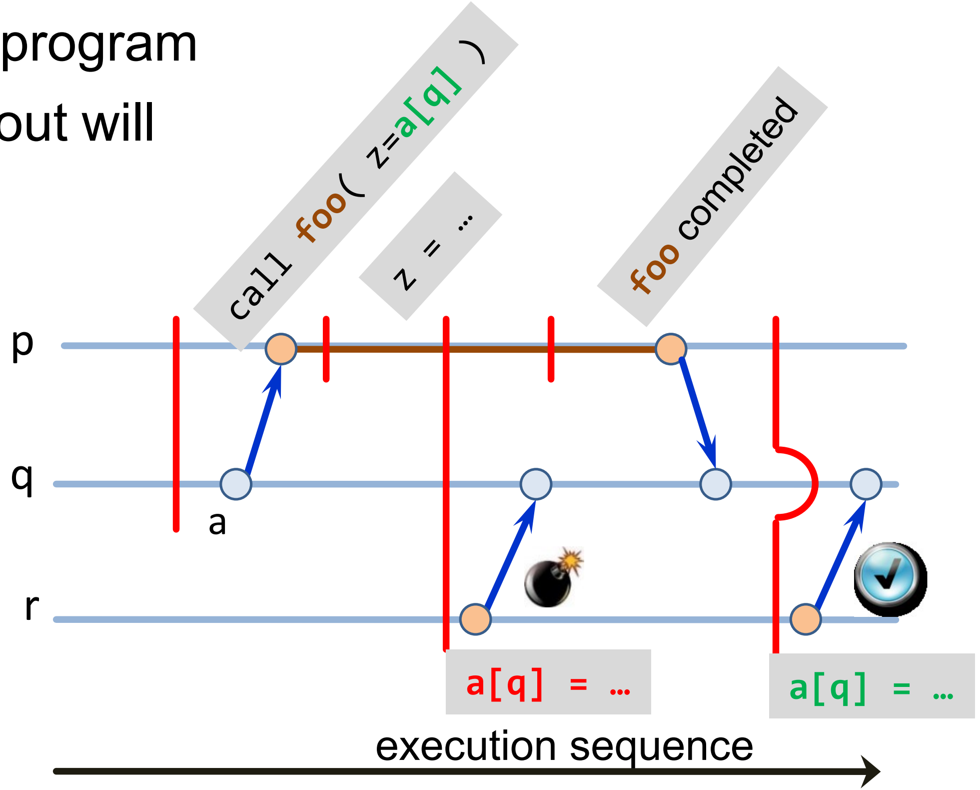
    If that coarray does not also have the SAVE attribute, it will be auto-deallocated at exit from the procedure if no explicit DEALLOCATE was previously issued.
- **UPC: has similar restrictions**
  - statically declared shared objects cannot be automatic

CAF: Coindexed actual arguments

Note: this has no UPC equivalent

- **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!)

CAF: Factory procedures for coarrays

Allocatable dummy argument is a coarray:

deferred coshape

```
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 into simulation_field
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

UPC Factory – shared pointer function result

- **Analogous functionality as for CAF is illustrated**

```
shared *float factory(char *file_name) {
  shared float *wk;
  : // determine size n to allocate
  wk = (shared float *) upc_all_alloc(THREADS, sizeof(float)*n);
  : // fill wk with data
  return wk;
}
```

typically, a multiple of THREADS

  - i.e., requires collective execution

- **Remember:**
  - other allocation functions `upc_global_alloc` (single thread distributed entity), `upc_alloc` (single thread shared entity) do not synchronize
  - this permits to implement factory functions that do not require collective execution

CAF: Overloading the assignment

Optional slide

- **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)

CAF: Using type-bound procedures to implement communication

Optional slide

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

`put_success` and `put_fail`
are distinct integer constants

```
type(polynomial) :: s[*]
integer :: status[*]
```

Execution

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

```
s = …
sync all
:
status[q] = s%put(q)
event post (ev[q])
```

- of consuming code on image **q**

```
s = …
sync all
:
event wait (ev)
if ( status == put_success ) then
  : ! reference local part of s
end if
```

Optional slide

Implementation sketch

```
integer function put(this, img)
  class(polynomial), intent(inout) :: this[*]
  integer, intent(in) :: img
  integer :: rem_size
  if ( .not. allocated( this[img]%f ) ) then
     put = put_fail
     return
  end if
  rem_size = size( this[img]%f, 1 )
  if ( rem_size >= size( this%f ) ) then
     put = put_success
     this[img]%f(:this%f) = this%f
     this[img]%f(this%f+1:) = 0.0
  else
     put = put_fail
  end if
end function
```

failure is determined to 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**

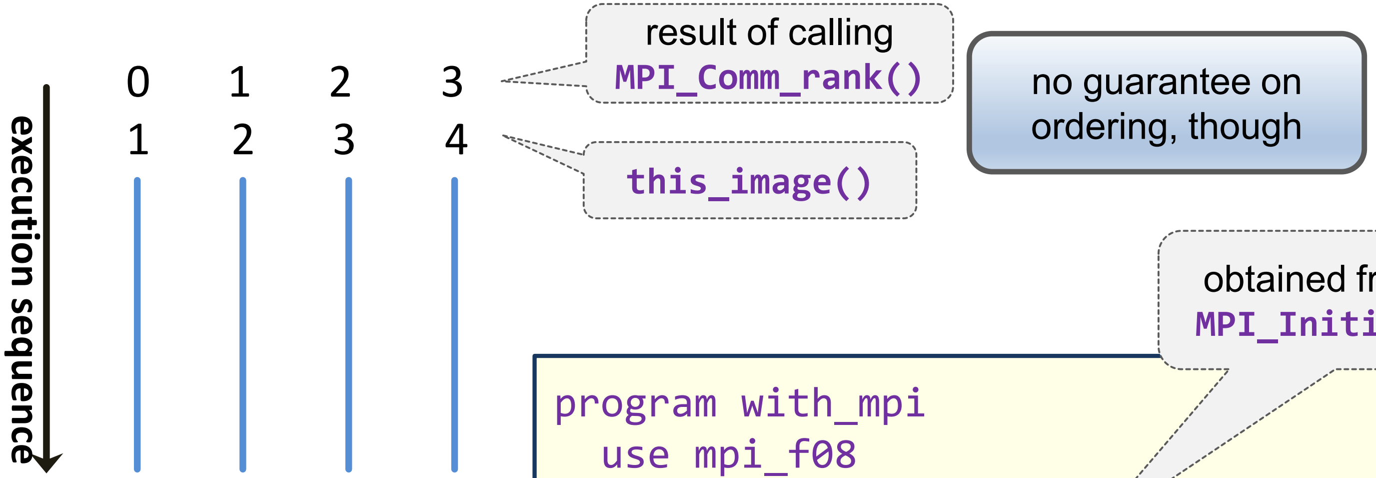
Documenting the synchronization behaviour

- **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

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 ☺

Appendix

Implementations

- **CAF**
  - Cray Fortran compiler on Cray systems
  - Intel 12.0 and higher (current release: 19.0)
  - gfortran (since 4.6: single image)
    - partial implementation in 5.0
    - more features in 8.0
  - Rice coarray Fortran (research vehicle, deviates from the standard, development stalled)
  - g95 (development stalled)

- **UPC**
  - Cray UPC
  - Berkeley UPC
  - GCC UPC

- **Note:**
  - performance problems still exist (tuning one-sided communication is a challenge)

  → do not expect MPI-like performance and scalability, except for the Cray compiler on appropriate networks

References

UPC references

- https://upc-lang.org/upc-documentation (language specification, release level 1.3)
- UPC Manual, by Sébastien Chauvin, Proshanta Saha, François Cantonnet, Smita Annareddy, Tarek El-Ghazawi, May 2005 http://upc.gwu.edu/downloads/Manual-1.2.pdf
- UPC Distributed Memory Programming, by Tarek El-Ghazawi, Bill Carlson, Thomas Sterling, and Katherine Yelick, Wiley & Sons, June 2005

Coarray references

- Coarrays in the next Fortran Standard, by John Reid, N1824 from https://wg5-fortran.org
- Fortran 2018 international standard
- Modern Fortran explained, by Michael Metcalf, John Reid and Malcolm Cohen (OUP, September 2018)
- Coarray compendium, by Andy Vaught, http://www.g95.org/compendium.pdf
- TS18508 „Additional parallel features in Fortran“, draft specification available as document N2074 from https://wg5-fortran.org
- The New Features of Fortran 2018, by John Reid, N2161 from https://wg5-fortran.org

Omitted topics

- **Omitted:**
  - rules for program termination
  - parallel I/O (mostly UPC)
  - asynchronous block transfers (UPC only)
- **Further CAF TS18508 features**
  - teams
    - composable splitting of execution contexts
    - allow data transfer and sync across team boundary
    - recursive / hybrid / MPMD-like
  - atomic functions (similar to those added in UPC 1.3)
  - limited fail-safe execution
- **Possible futures**
  - process topologies in CAF
    - more general abstraction than multiple coindices
  - global variables and shared pointers in CAF
    - increase programming flexibility
  - parallel I/O in CAF
  - asynchronous transfers in CAF
  - CAF+UPC interoperation
  - UPC++
    - https://bitbucket.org/berkeleylab/upcxx/wiki/Home
- **Recent development**
  - Coarray C++
    - presently available on Cray systems
    - uses template mechanism and leverages existing Fortran run time to map coarrays to C++

© 2010-19 LRZ

Acknowledgements

- Significant parts of this slide set are based on the SC12 tutorial notes:

„Introduction to PGAS (UPC and CAF) and Hybrid for Multicore Programming"

by

Alice E. Koniges – NERSC, Lawrence Berkeley National Laboratory (LBNL)
Katherine Yelick – University of California, Berkeley and LBNL
Rolf Rabenseifner – High Performance Computing Center Stuttgart (HLRS), Germany
Reinhold Bader – Leibniz Supercomputing Centre (LRZ), Munich/Garching, Germany
David Eder – Lawrence Livermore National Laboratory
Filip Blagojevic and Robert Preissl – Lawrence Berkeley National Laboratory