PGAS
Partitioned Global Address Space Languages

Coarray Fortran (CAF)
Unified Parallel C (UPC)

Dr. R. Bader
Dr. A. Block

May 2010

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

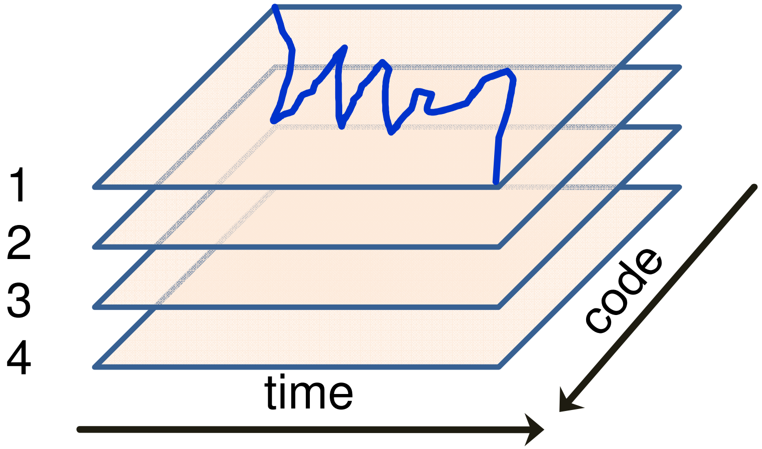
memory management for dynamic sharable entities

Standardization efforts:

- Fortran 2008 draft standard (now in DIS stage, publication targeted for August 2010)
- separately standardized C extension (work in progress; existing document is somewhat informal)

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 replicate**
  - program-controlled exchange of data
  - may necessitate synchronization

Execution model: Resource mappings

- **One-to-one:**
  - each image / thread executed by a single physical processor core
- **Many-to-one:**
  - some (or all) images / threads are executed by multiple cores each (e.g., socket could support OpenMP multi-threading within an image)
- **One-to-many:**
  - fewer cores are available to the program than images / threads
  - 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

Simplest possible program

- **CAF – intrinsic integer functions for orientation**

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

between 1 and `num_images()`

- **UPC**
  - uses integer expressions for the same purpose

non-repeatably unsorted output if multiple images/threads used

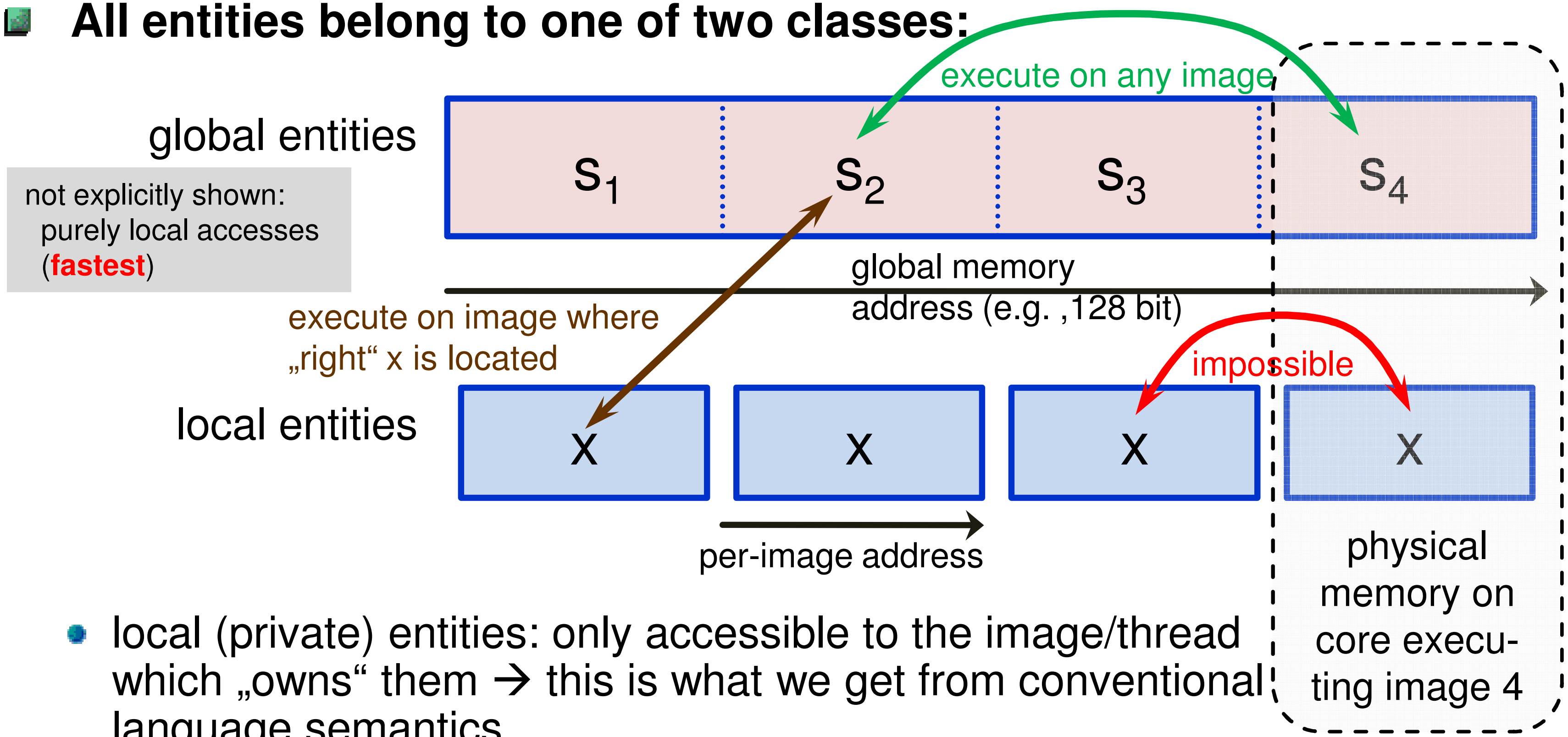
```
#include <upc.h>
#include <stdlib.h>
#include <stdio.h>

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

between 0 and `THREADS - 1`

PGAS 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“:
→ **different** semantics than in OpenMP (esp. for CAF)

Declaration of corrays/shared entities
(simplest case)

- **CAF:**
  - coarray notation with **explicit** indication of location (coindex)
  - symmetry is enforced (asymmetric data must use derived types – see later)

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

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

  - more images → additional coindex value

- **UPC:**
  - uses **shared** attribute
  - implicit locality; various **blocking** strategies
  - asymmetry – threads may have uneven share of data

```
shared [1] int A[10];
shared     int A[10];
```

Thread 0	1	2	3
A[0]	A[1]	A[2]	A[3]
A[4]	A[5]	A[6]	A[7]
A[8]	A[9]		

  - more threads → e.g., A[4] moves to another physical memory

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 A[N][N];
```

  - complete matrix rows on each thread (≥1 per thread)

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

  - storage sequence **matches with rank 1 coarray** from previous slide (→ symmetry restored)
  - static THREADS environment may be required (compile-time thread number determination)

CAF shared data: coindex to image mapping

Coarrays

- may be scalars

```
integer s[-2:*]
```

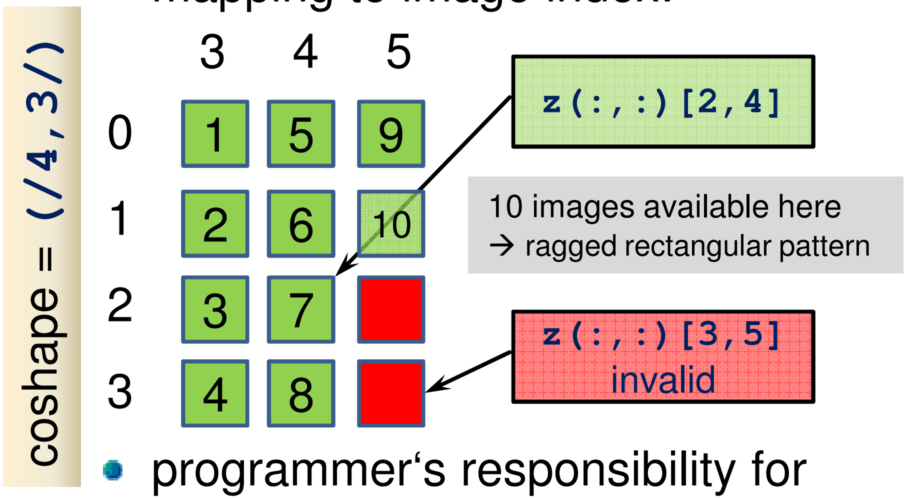
Image	1	2	3	4
	S[-2]	S[-1]	S[0]	S[1]

- and/or have **corank** > 1

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

lower cobound of codimension 1 — upper cobound of last codimension

- mapping to image index:



- programmer's responsibility for **valid** coindex reference (see later for additional intrinsic support)

Intrinsic functions

- query lower and upper cobounds

```
lcobound(coarray[,dim,kind])
ucobound(coarray[,dim,kind])
```

- return an integer array or scalar
- scalar if optional `dim` argument is present
- **example:**

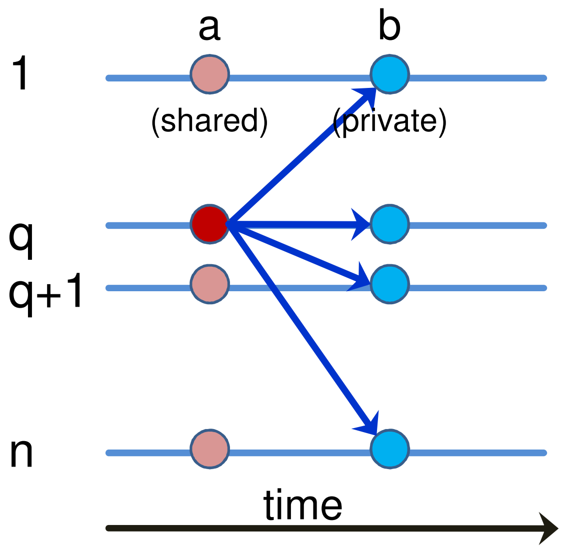
```
write(*, *) lcobound(z)
```

will produce the output `0 3`

Accessing shared data

Simplest example

- collective scatter: each thread/image executes **one** statement implying data transfer



- one-sided semantics:

b = a (from image/thread q)
„Pull"

CAF syntax:

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

b  =  a(:)[q]
```

q: **same** value on all images/threads

UPC syntax:

remember enforced symmetry

```
int b[3];
shared [*] int a[THREADS][3];

for (i=0; i<3; i++) {
  b[i] = a[q][i];
}
```

Note:

- initializations of a and q omitted – there's a catch here …

Locality control

CAF:

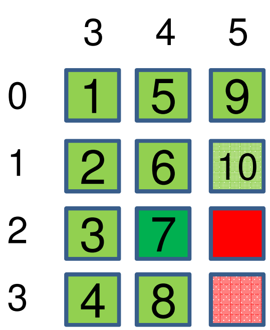
- local accesses are fastest and trivial

```
integer :: a(3)[*]
a(:) = (/ … /)
```

same as `a(:)[this_image()]` but probably faster

- coarray ↔ coindexed object
- explicit coindex: usually a visual indication of communication
- supporting intrinsics:

10 images




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

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 **any** image, returns 0

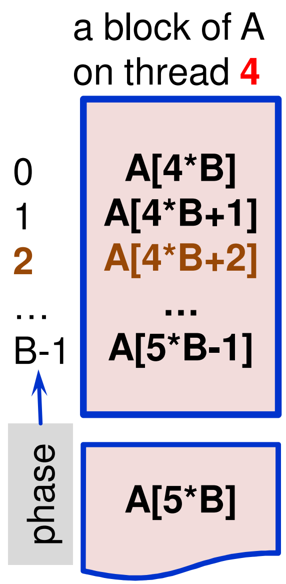
UPC:

- implicit locality → cross-thread accesses easier to write
- ensuring local access: explicit mapping of array index to thread may be required (see next slide)
- supporting intrinsics:

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

img = \
upc_threadof(&A[4*B+2]);
```
on **any** thread, returns 4

```
pos = \
upc_phaseof(&A[4*B+2]);
```
on **any** thread, returns 2



Work sharing + avoiding non-local accesses

- **Typical case**
  - loop or iterator execution
- **CAF:**
  - index transformations between local and global

```
integer :: a(nlocal)[*]
do i=1, nlocal
   j = … ! global index
   a(i) = …
end do
```

- **UPC:**
  - loop over all, work on subset

```
shared int a[N];
for (i=0; i<N; i++) {
  if (i%THREADS == MYTHREAD) {
     a[i] = … ;
  }
}
```

  - conditional may be inefficient
  - cyclic distribution may be slow
- **upc_forall**
  - integrates affinity with loop construct

```
shared int a[N];
upc_forall (i=0; i<N; i++; i) {
   a[i] = … ;
}
```

afffinity expression

  - affinity expression:

    an integer → execute if
    `i%THREADS == MYTHREAD`

    a global address → execute if
    `upc_threadof(…) == MYTHREAD`

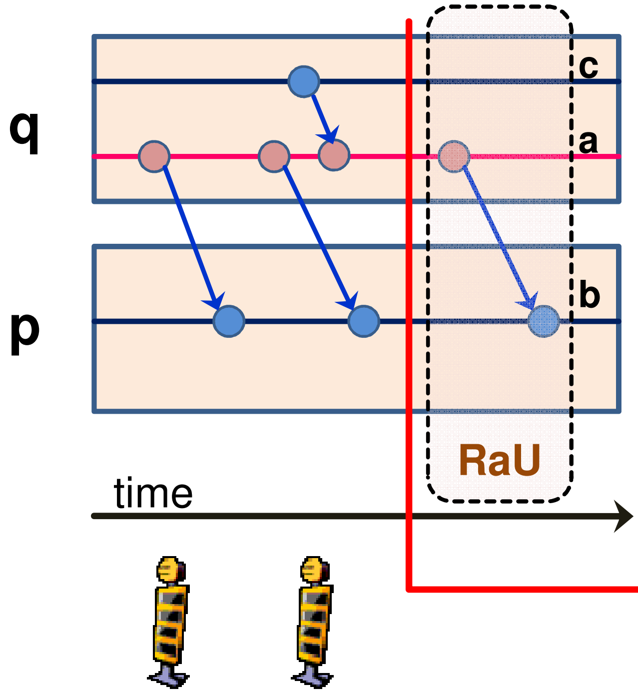
    example above: could replace „i“ with „&a[i]“

Race conditions – need for synchronization

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

a = c
b = a(:)[q]
```

- **Focus on image pair q, p:**
  - three scenarios



- **Serial semantics**
  - execution sequence
- **Parallel semantics**
  - relaxed consistency
  - unordered **segments** of p, q
  - explicit synchronization by user required to prevent races
- **Imposed by algorithm:**
  - RaU („reference after update“) is correct

CAF terminology

global barrier:
enforce ordering of segments – on **all** images

```
a = c
sync all
b = a(:)[q]
```

```
for (…) a[n][i]= …;
upc_barrier;
for (…) b[i]=a[q][i];
```

UPC: Consistency modes

- **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 implicit barriers
  - 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
```

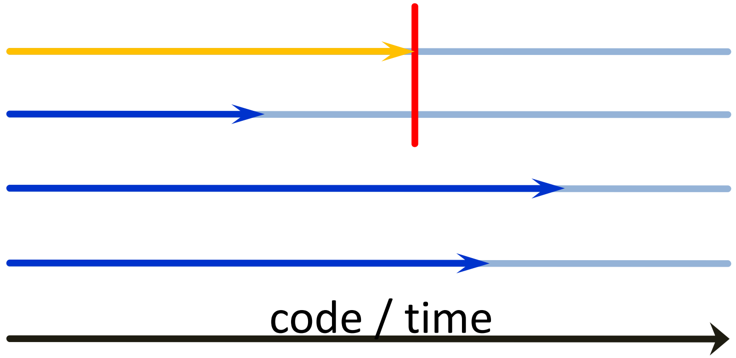
  - program level

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

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

CAF: partial synchronization

Synchronizing subsets



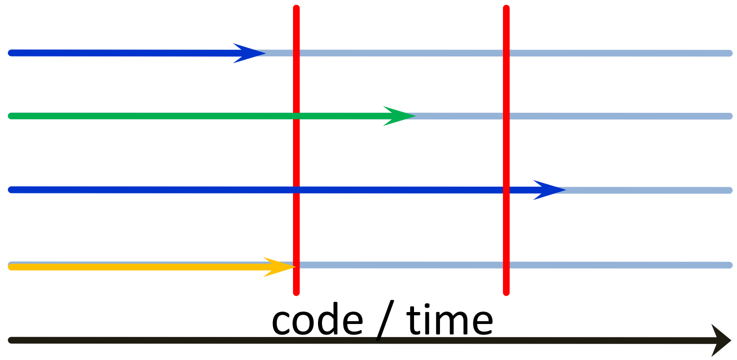

```
if (this_image() < 3) then
  sync images ( (/ 1, 2 /) )
end if
```

- all images against one:

```
if (this_image() == 1) then
  : ! send data
  sync images ( * )
else
  sync images ( 1 )
  : ! use data
end if
```

images 2 etc. don‘t mind stragglers

Critical regions



- only one thread at a time executes
- order is unspecified

```
critical
  : ! statements in region
end critical
```

- can have a name, but this has no specific semantics

Memory fences and atomic subroutines – user-defined light-weight synchronization

atomic entities are **exempt** from the synchronization rules
programmer‘s responsibility for proper handling

CAF: spin-lock example

```
logical(ATOMIC_LOGICAL_KIND),save :: &
        ready[*] = .false.
logical :: val

me = THIS_IMAGE()
if (me == p) then
  : ! produce
  sync memory     ! A
  call ATOMIC_DEFINE(ready[q], .true.)
else if (me == q)
  val = .false.
  do while (.not. val)
    call ATOMIC_REF(val, ready)
  end do
  sync memory     ! B
  : ! consume
end if
```

segment P_i ends

segment Q_j starts

- memory fence: prevents reordering of statements (A), enforces memory loads (for coarrays, B)

UPC:

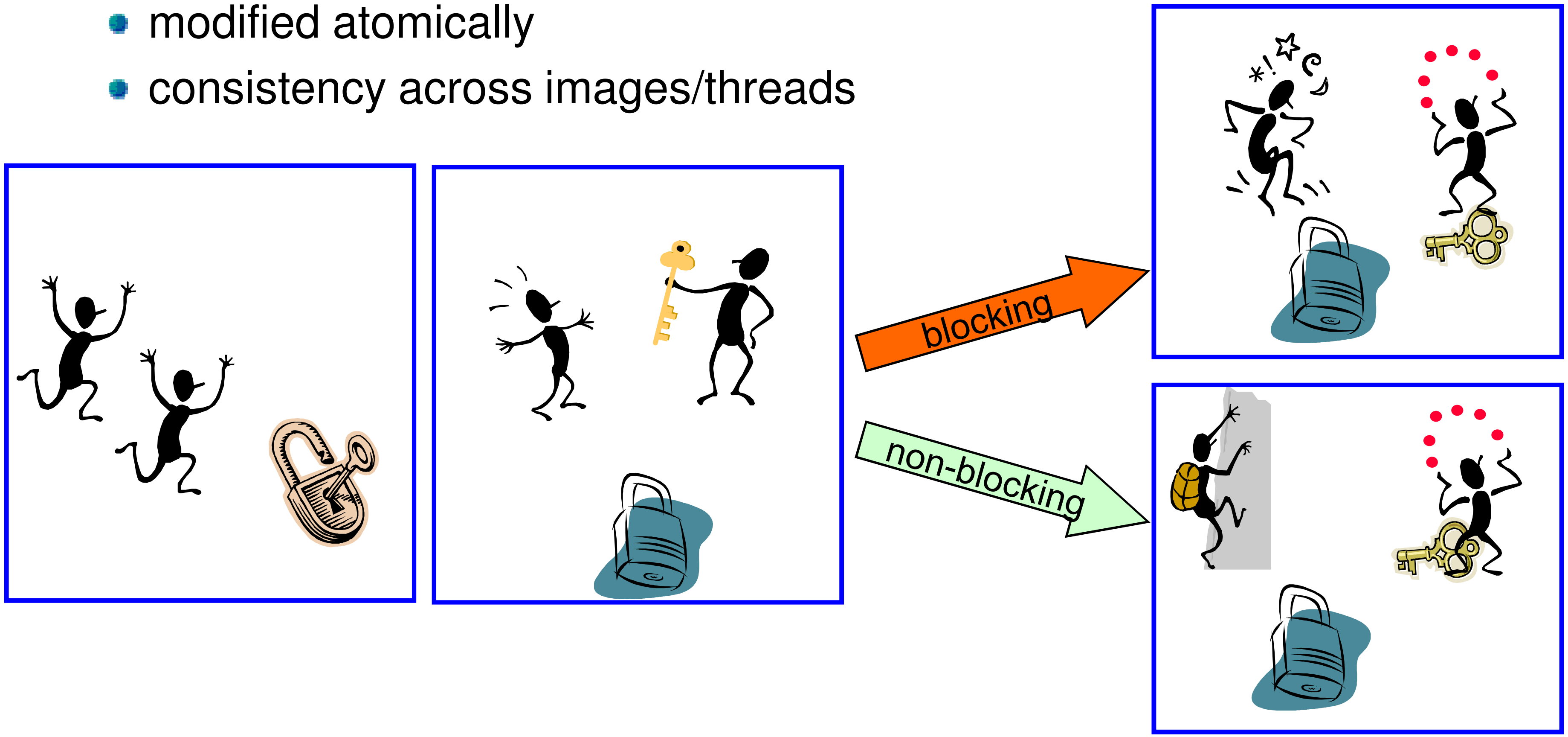
- memory fence is defined by **`upc_fence;`**
- atomic functions: **extension** supported by Berkeley UPC

Remarks

- memory fence: implied by many other synchronization constructs
- atomic operations:
  - guarantee undivided state change, but not a particular ordering or appearance
  - light-weight – if hardware supports atomic operations, better-performing than the big global barrier hammer

Locks – fine grain synchronization

- **Coordinate access to shared (=sensitive) data**
  - sensitive data represented as “red balls”
- **Use a coarray/shared lock variable**
  - modified atomically
  - consistency across images/threads



Locks – fine grain synchronization

CAF:

- simplest examples

```
use, intrinsic :: iso_fortran_env

type(lock_type) :: lock[*]
!  default initialized to unlocked
logical :: got_it

lock(lock[1])
: !  play with red balls
unlock(lock[1])

do
  lock(lock[2], acquired_lock=got_it)
  if (got_it) exit
: ! do other stuff
end do
: ! play with other red balls
unlock(lock[2])
```

like **critical**, but more flexible

- lock must be a coarray → as many locks as there are images
- lock/unlock: no memory fence, only **one-way** segment ordering

UPC:

- single pointer lock variable

```
#include <upc.h>

upc_lock_t *lock;    // local pointer
                     // to a shared entity

lock = upc_all_lock_alloc();

upc_lock(lock);
: // play with red balls
upc_unlock(lock);

for (;;) {
  if (upc_lock_attempt(lock)) break;
  : // do other stuff
}
: // play with red balls
upc_unlock(lock);
upc_lock_free(lock);
```

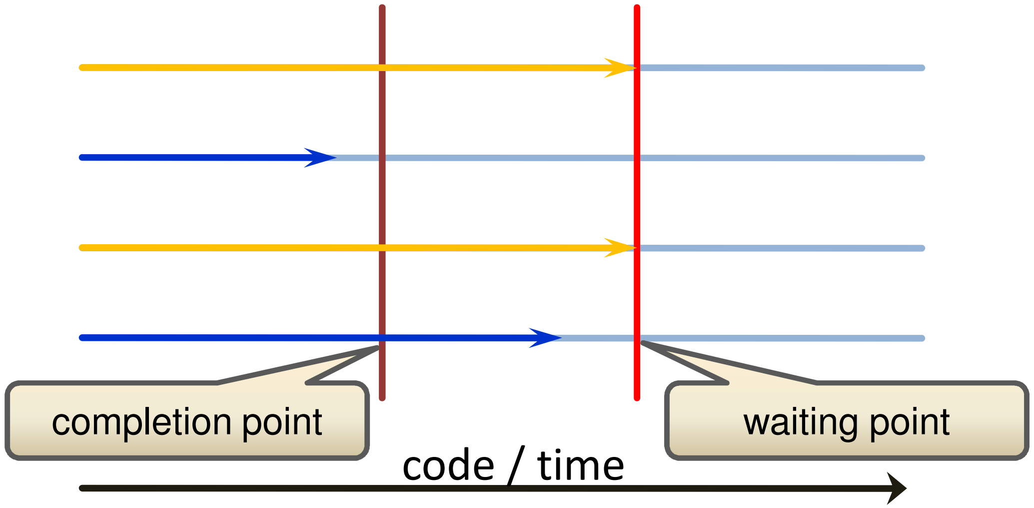
collective call same result on each thread

- thread-individual lock generation is also possible (non-collective)
- lock/unlock imply memory fence

UPC: Split-phase barrier

- **Separate barrier completion point from waiting point**
  - this allows threads to perform other computations before they are required to wait

```
for (…) a[n][i]= …;
upc_notify;
// do work not
// involving a
upc_wait;
for (…) 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
  - can define using locks

Dynamic entities: Pointers

Remember pointer semantics

- different between C and Fortran

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

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

no pointer arithmetic
type and rank matching

```
<type> *ptr;

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

pointer arithmetic
rank irrelevant
pointer-to-pointer
pointer-to-void

Pointers and PGAS memory categorization

- both pointer entity and pointee might be private or shared → **4 combinations** possible
- UPC: three of these combinations are realized
- CAF: only two of them allowed, and only in a limited manner ← aliasing only to local entities

Pointers continued …

CAF:

```
integer, target :: i1[*]
integer, pointer :: p1

type :: ctr
  integer, pointer :: p2(:)
end type
type(ctr) :: o[*]
integer, target :: i2(3)
```

a coarray **cannot** have the `pointer` attribute

- entity „o“: typically asymmetric

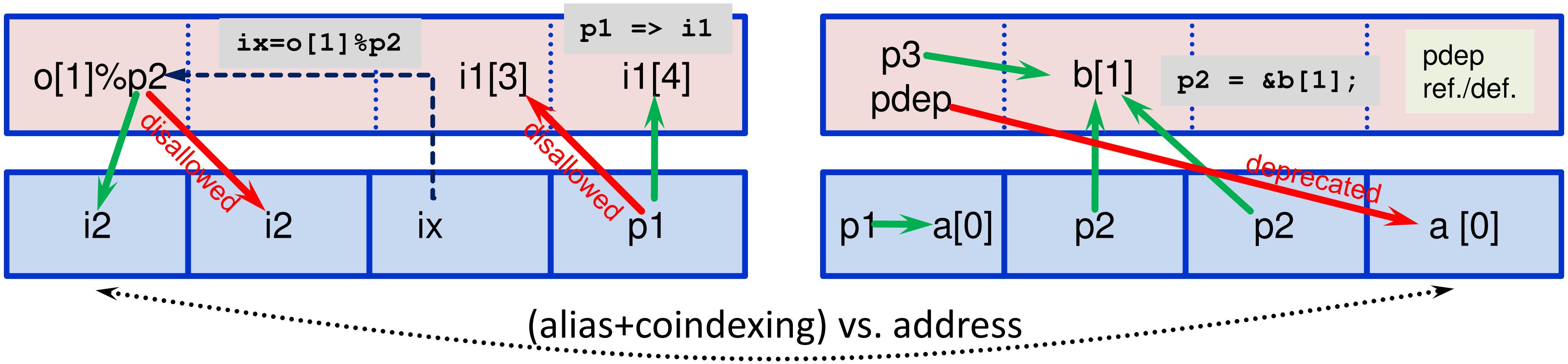
UPC:

```
int *p1;
shared int *p2;
int *shared p3;
int *shared pdep;

int a[N];
shared int b[N];
```

problem: where does pdep point? all other threads may not reference

- pointer to shared: addressing overhead



Dynamic entities: Memory management

collective allocation facilities which **synchronize all** images/threads

- **CAF:**

deferred shape and coshape

```
integer, &
  allocatable :: id(:)[:]

allocate(id(100)[2:*])
```

- **symmetric** allocation required: same type, type parameters, bounds and cobounds on every image

```
deallocate(id)
```

- deallocation: synchronizes before carried out

- **UPC:**

```
shared [100] int *id;
id = (shared [100] int *) \
      upc_all_alloc( \
      THREADS,100*sizeof(int));
```

number of blocks

bytes per block

- layout equivalent to coarray on the left (note compile time constants)
- arguments of type `size_t`
- result is a pointer to shared (same value on each thread)
- deallocation

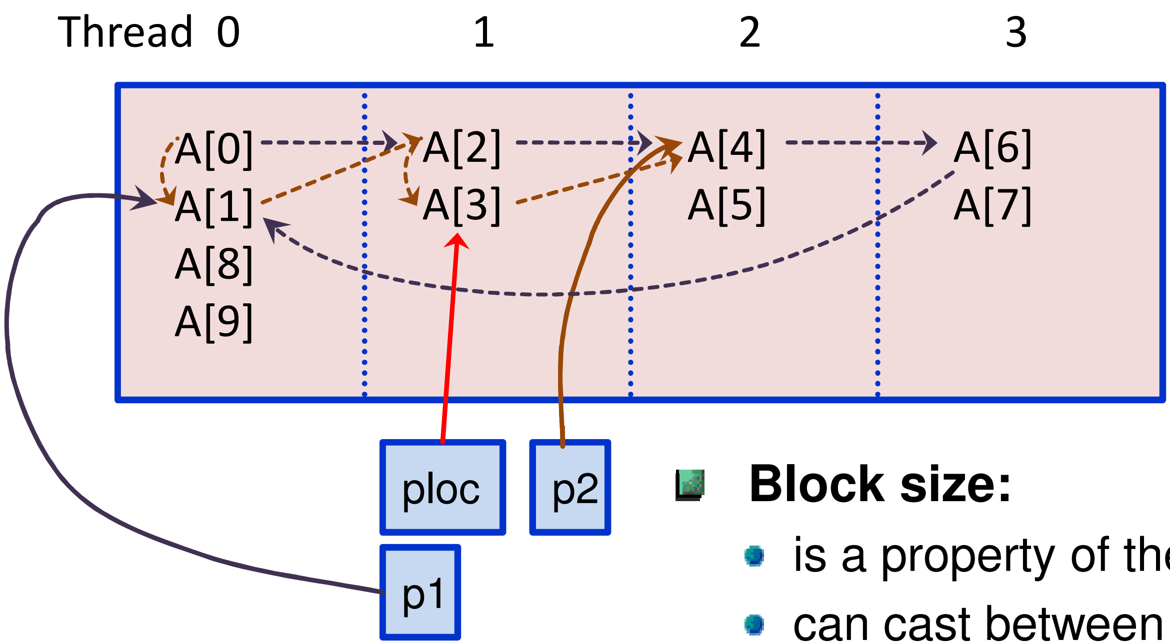
```
upc_barrier;
if (MYTHREAD==0) upc_free(id);
```

is **not** collective

UPC: Pointer blocking and casting

- **Assume 4 threads:**

```
shared [2] int A[10];
shared int *p1;
shared [2] int *p2;
int *ploc;
```




```
if (MYTHREAD == 1) {
 p1 = &A[0]; p2 = &A[0];
 p1 += 4; p2 += 4;
 ploc = (int *) &A[2];
 ploc += 1;
}
```

- **Block size:**
  - is a property of the shared type used
  - can cast between different block sizes → pointer arithmetic follows blocking of pointer!
- **Cast to a pointer to local:**
  - must point to data with affinity to current thread

CAF: coarray type components

- **Declaration:**

```
type :: shared_stuff
  real, allocatable :: r(:)[:]
  : ! other (maybe non-coarray) components
end type
```

  - component must be allocatable
  - type extension: base type must already have a coarray component
- **Usage / Semantics**
  - much like allocatable coarrays
  - entities must be scalar, may not be allocatable or pointers

```
type(shared_stuff) :: o
allocate(o%r(100)[-4:*])) ! synchronizes
: ! use o%r
deallocate(o%r)
```

Non-synchronizing memory management facilities

CAF:

- allocatable type components

```
type :: ragged
  integer, allocatable :: a(:)
end type
type(ragged) :: o[*]

allocate(o%a(10*this_image()))
```

dynamic allocation: could be in shared space

must be local

can be asymmetric

- remote accesses require synchronization

```
sync all
b(1:size(o[p]%a)) = o[p]%a
```

… assuming b is large enough

UPC:

- two routines, both called by individual threads …

```
shared void *upc_global_alloc(
                NBLOCKS,NBYTES);
```

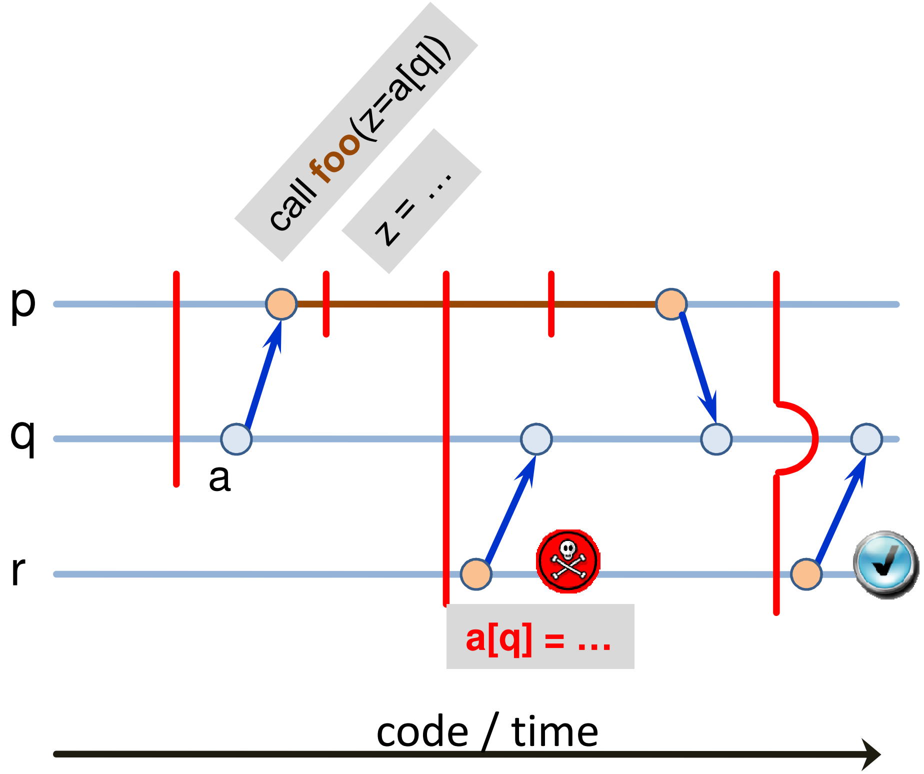
- per-thread pointer to first element of multiple distributed blocks

```
shared void *upc_alloc(NBYTES);
```

- memory allocated in shared space on calling thread (→ single block with affinity)
- pointer to first element of allocated memory
- require shared pointer to handle data transfers ("directory")

Coarrays in subprograms

- **Important cases:**
  1. coarray dummy arguments
  2. local coarray entities
  3. non-coarray dummy arguments associated with a coindexed object
- **Case 1:**
  - restrictions ensure that no copy-in/out can occur
  - for allocatable entities, synchro-nization can occur inside subpro-gram, symmetric call is required
- **Case 2:**
  - SAVE attribute required, **no** automatic entities
  - allocatable is allowed → synchronization is implied
- **Case 3:**
  - copy-in/out will usually occur → **additional** synchronization rule needed



UPC collective operations

- **Separate include file**

```
#include <upc_collective.h>
```

- **Two types:**
  - data redistribution (e.g., scatter, gather)
  - computation operations (reduce, prefix, sort)
- **Synchronization mode:**
  - constants of type `upc_flag_t`

```
                NOSYNC
UPC_  IN        
      OUT   _   MYSYNC
                ALLSYNC
```

  - entry or exit point, synchronize not at all / wrt data on entered threads / wrt all threads → allow function to read/write data
  - can combine using „|“

- **Example:**

```
void upc all reduceT(
     shared void *restrict dst,
     shared const void *restrict src,
     upc_op_t op, size_t nelems,
     size_t blk size, T(*func)(T, T),
     upc_flag_t flags);
```

  - `T` is one of the following types:

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

  - **op** is one of the following operations: UPC_ADD, UPC_MULT, UPC_AND, UPC_OR, UPC_XOR, UPC_LOGAND, UPC_LOGOR, UPC_MIN,UPC_MAX, UPC_FUNC, UPC_NONCOMM_FUNC

Parallel I/O extensions in UPC

- **Extension to UPC**
  - defined in `upc_io.h`
  - provide collective I/O functions
  - local and shared reads and writes (individual vs. shared file pointer)
- **Individual file pointer**
  - read thread-specific sections of a file
  - write thread-specific sections of a file
  - special flag for atomicity and consistency semantics (writes from multiple threads)
  - otherwise weak semantics (`upc_all_fclose()`, `upc_all_fsync()`)
- **Shared file pointer**
  - one pointer shared by all threads
  - cannot use in conjunction with pointers to local buffers
  - consistency requirements for arguments
- **I/O on shared vs. private data**
  - can do both using individual file pointers

Possible future developments

- **Teams**
  - load imbalanced problems (partial synchronization)
  - recursive algorithms
- **Asyncs and Places**
  - memory and function shipping
  - support for accelerator devices?
- **Collective calls in CAF**
  - maybe even asynchronous?
- **Process topologies in CAF**
  - more general abstraction than multiple coindices
- **Global variables and co-pointers in CAF**
  - increase programming flexibility
- **Split-phase barrier in CAF**
- **Parallel I/O in CAF**

Thank you for your attention!

Any questions?