PGAS

Partitioned Global Address Space Languages

Incorporating parallelism into C and Fortran

An Overview

Reinhold Bader

Alexander Block

Leibniz Supercomputing Centre

February 2010

Coarray Fortran and UPC

- **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

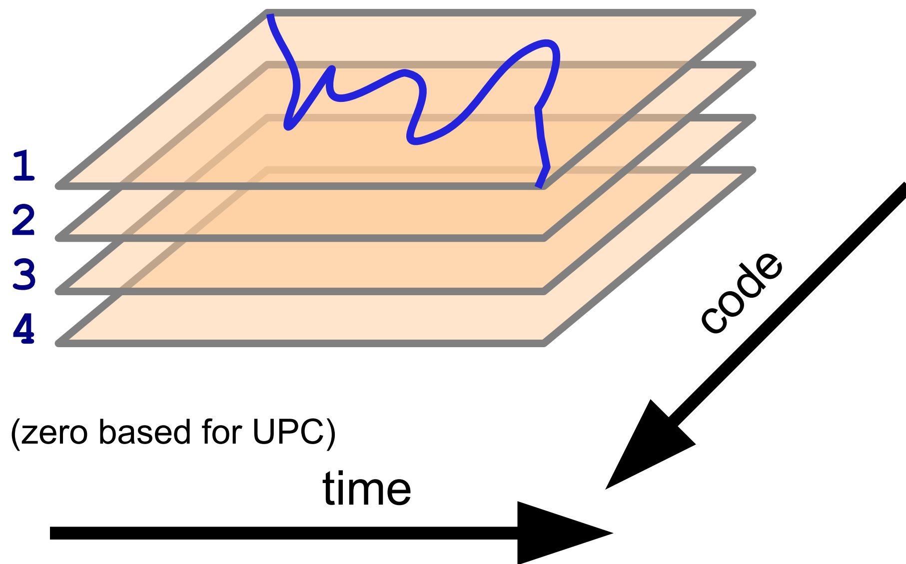
    data distribution: partitioned memory model

    synchronization vs. race conditions

    memory management for „sharable“ entities

- **Standardization efforts:**
  - Fortran 2008 draft standard (now FDIS, publication targeted for August 2010)
  - separately standardized C extension (work in progress; existing document is far too informal)

Program execution: The concept of [CAF: image | UPC: thread]



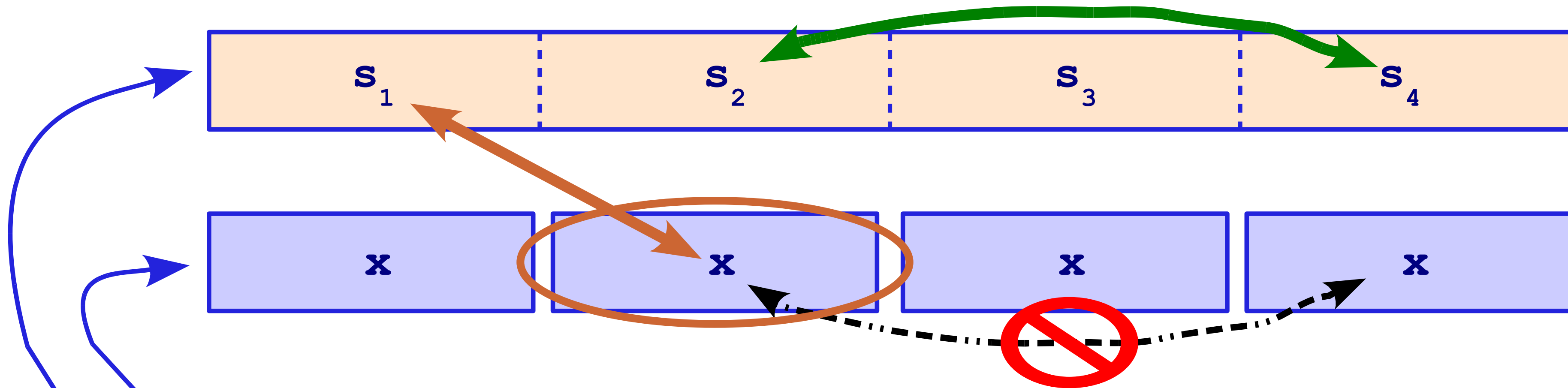
- **replicate single program a fixed number of times**
  - set number of replicates at compile time or at execution time (UPC: impact on semantics)
  - asynchronous execution – loose coupling unless program-controlled synchronization occurs
- **separate set of data objects on each replicate**
  - program-controlled exchange of data
  - may necessitate synchronization

Execution models / Resource availability

- **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 multithreading **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**

Memory model

- **All data belongs to one of two classes:**



  - local („private") data: **only** accessible to the image/thread which „owns" them
  - global („sharable") data – partitioned global memory: data declared on and physically assigned to one image/thread may be accessed by **any** other one

- **UPC and CAF provide (slightly different) syntax to implement these memory semantics**
  - improved locality control compared to OpenMP

Declarations of sharable data

CAF:

- coarray notation with **explicit** indication of location
- symmetry is enforced – must use derived types with dynamic components for asymmetric objects

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

Image 1	Image 2	Image 3	Image 4	Image 5
A(1)[1]	A(1)[2]	A(1)[3]	A(1)[4]	A(1)[5]
A(2)[1]	A(2)[2]	A(2)[3]	A(2)[4]	A(2)[5]
A(3)[1]	A(3)[2]	A(3)[3]	A(3)[4]	A(3)[5]

UPC:

- uses **shared** attribute
- various blocking strategies
- asymmetry – some images typically do not have data

```
shared [1] int A[12];
shared     int A[12];
```

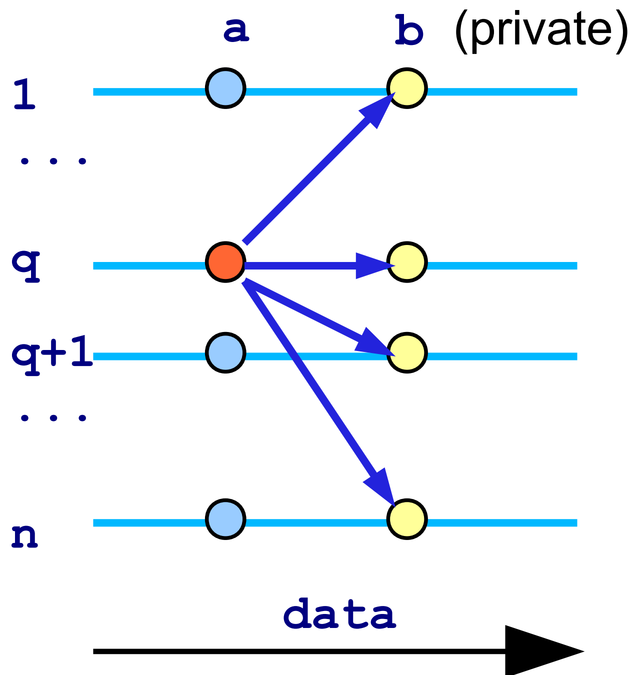
Thread 0	Thread 1	Thread 2	Thread 3	Thread 4
A[0]	A[1]	A[2]	A[3]	A[4]
A[5]	A[6]	A[7]	A[8]	A[9]
A[10]	A[11]			

- nearest analogue to CAF:

```
shared [*] int A[12];
```

Cross-[image|thread] access to sharable data

- **One-sided semantics**



- **Coarray notation:**

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

  - executed on all images

- **UPC notation:**

```
for (i=0; …) {
  int j = …;
  b[j] = a[i];
}
```

  - blocking-dependent subset of index values (note: `upc_forall` makes this easier)
  - executed on all threads

- **corresponds to MPI collective scatter operation**
  - but simpler to handle
  - no library call → processor can perform **optimization** of com-munication or even non-blocking behaviour

© 2010 LRZ

Integrated type system

- **Sharable entities (for UPC and CAF)**
  - may be of any intrinsic or derived type
  - another conciseness advantage compared to MPI

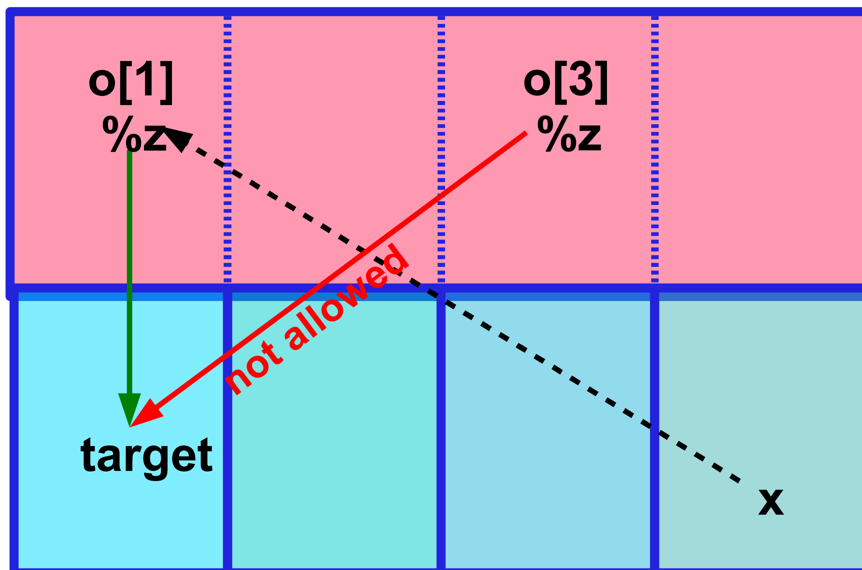
- **In particular, types with dynamic components may be used …**

Use of pointer entities

- **CAF**
  - requires a derived type

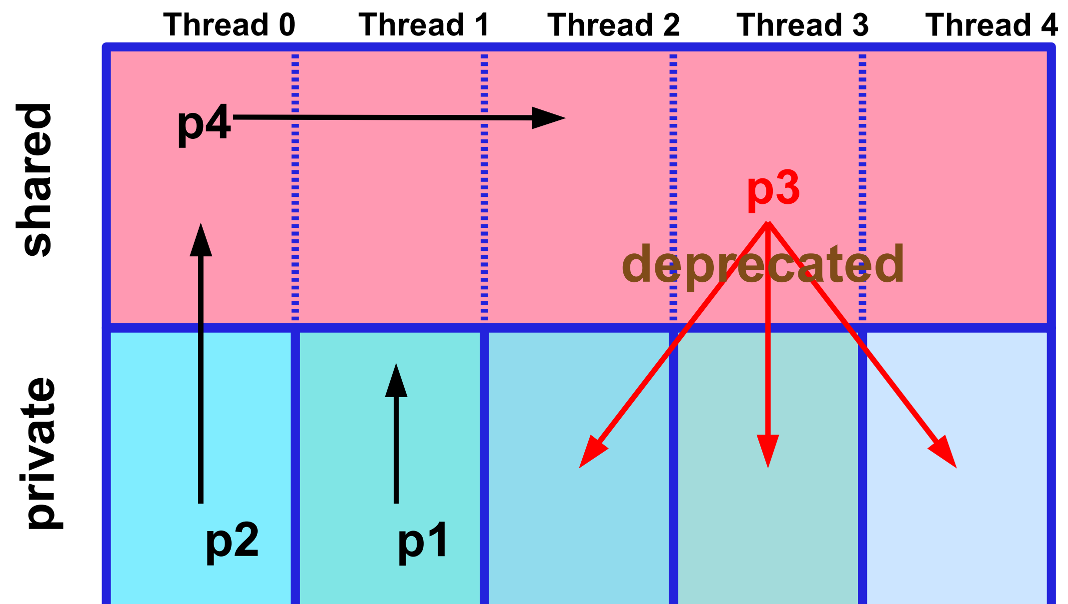
```
type :: my_ptype
  integer, pointer :: z(:)
end type
type(my_ptype) :: o[*]
```

  - remember alias semantics



- **UPC:**
  - three kinds of pointers

```
int *p1;              // private to private
shared int *p2;       // private to shared
int *shared p3;       // shared to private
shared int *shared p4;
                      // shared to shared
```

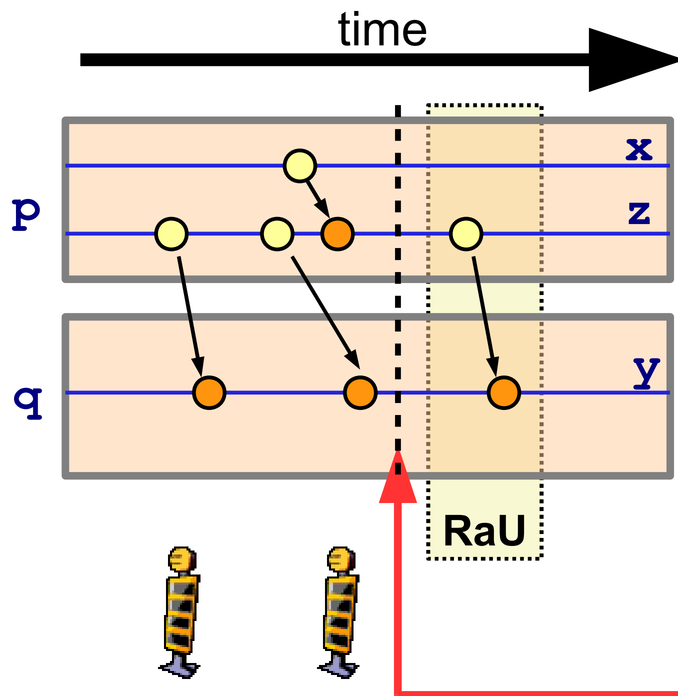


UPC vs. CAF syntax and semantics

- **UPC has more flexibility in setting up data**
  - can only add `shared` to a declaration – voilà a distributed object
  - performance impact of access to sharable data → need to design for minimal communication anyway
  - but: non-local accesses to shared data not easily visible (to programmer) in the program text
- **Fragmented model of CAF**
  - gives processor more freedom for optimization
  - gives programmer visual clue for occurrence of communication
- **Conclusion:**
  - well-optimizing compilers and efficient program implementations probably easier to do with CAF

Race conditions – need for synchronization

```
real :: z(100)[*]
real :: x(100), y(100)
:
z = x
y = z(:)[p]
```



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

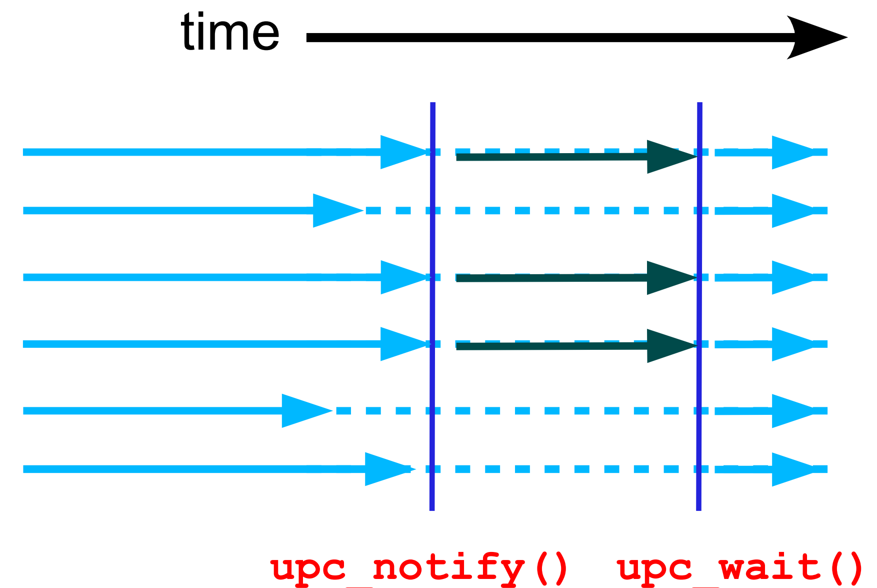
```
:
z = x
sync all
y = z[p]
```

enforce ordering
→ two segments

Available synchronization mechanisms

- **Global barrier**
  - sync all / upc_barrier()
- **Image subsets**
  - sync images – CAF only
- **Split-phase global barrier**
  - upc_notify / upc_wait - UPC only
- **Locks**
  - protect specific data against multiple image access
- **Atomic updates / references**
  - CAF only
- **Critical regions**
  - CAF only

- **Memory fence**
  - sync memory, upc_fence
  - user-defined synchronization



- **UPC synchronization modes**
  - relaxed vs. strict
  - strict: implicit synchronization

Memory management for sharable entities

- **CAF:**
  - synchronizing ALLOCATE statement for coarrays

```
real, dimension(:), &
  allocatable :: dyn[:]
:
allocate(dyn(100)[*])
: ! use dyn for calculation
deallocate(dyn)
```

- **UPC:**
  - three variants of shared pointer allocation
  - all return a `shared void *`
  - block distribution parameterized by `NBLOCKS, NBYTES` arguments
  - `upc_free()`

- `upc_all_alloc(NBLOCKS,NBYTES)`
  - called by all threads, allocates data in sharable memory
  - on each thread, returns pointer to first element of single distributed block
- `upc_global_alloc(NBLOCKS, NBYTES)`
  - called by one or more threads
  - per-thread pointer to first element of multiple distributed blocks
- `upc_alloc(NBYTES)`
  - memory allocated in shared space on calling thread (→ affinity)
  - pointer to first element of allocated memory on all calling threads

Further items

- **Intrinsics**
  - who am I / how many?
  - efficient memory copying (UPC only)
  - data location queries
  - collectives (reductions etc.): UPC only
- **Support for parallel I/O**
  - UPC only, full implementations?
- **Future CAF:**
  - planned TR, John Mellor-Crummey's new concepts
  - teams (much more efficient handling of image subsets → scalability improvements)
  - collectives
  - topologies?
  - I/O

Comparison with other paradigms

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

Feature	MPI	OpenMP	CAF / UPC
Portability	yes	yes	yes
Interoperability (Fortran/C/C++)	yes	yes	not yet
Scalability	4	2	4
Performance	4	2	3+
Ease of Use	1	4	3
Data parallelism	no	partial	partial
Distributed memory	yes	no	yes

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