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

Dr. R. Bader
Dr. A. Block

January 2011
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:

- separately standardized C extension (work in progress; existing document is somewhat informal)
Execution model: UPC threads / CAF images

1. Going from single to multiple execution contexts
   - CAF - images:
   - UPC uses zero-based counting
   - UPC uses the term thread where CAF has images
   - will occasionally use „task“ as a synonym for both

2. 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

3. Separate set of entities on each replicate
   - program-controlled exchange of data
   - may necessitate synchronization

© 2010-2011 LRZ
## Comparison with other parallelization methods

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

### Ratings:
- 1-low
- 2-moderate
- 3-good
- 4-excellent

---

**Coarray Fortran (and PGAS in general):**
good scalability for fine-grain parallelism in distributed memory systems will require use of special interconnect hardware features
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., implementation 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
„Hello world“ with PGAS

CAF – intrinsic integer functions for orientation

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

UPC

- uses integer expressions for the same purpose

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

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

between 0 and THREADS - 1

between 1 and num_images()
All entities belong to one of two classes:

- **Global entities**: Declared on and physically assigned to one image/thread may be accessed by any other one. This allows implementation for distributed memory systems.

- **Local (private) entities**: Only accessible to the image/thread which "owns" them. This is what we get from conventional language semantics.

The term "shared": different semantics than in OpenMP (esp. for CAF).

© 2010-2011 LRZ
Declaration of coarrays/shared entities
(simplest case)

CAF:
- coarray notation with **explicit** indication of location (coindex in square brackets)
- symmetry is enforced
  (asymmetric data must use derived types)

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

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

```
shared [1] int B[10];
// or shared int B[10];
```

**Mapping is trivial in the simplest case**

**more images → additional coindex value**

**more threads → e.g., B[4] located on a different physical memory**
Enforcing symmetry for UPC shared objects
(make them as similar as possible to coarrays)

Two methods
- extra dimension indexes threads
- THREADS macro in declaration

Method 1:
```c
shared int A[3][THREADS];
```

Method 2:
- use a non-default block size
  (number of subsequent elements placed on any thread)
```c
shared [3] int A[THREADS][3];
```

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
- programmers may prefer implicit distribution for simplicity of use (but beware unintentioned cross-thread accesses)
Further possibilities (not covered in this talk)

- Declaration variants
  - coarrays with corank > 1, and/or lower cobounds /= 1
    ```fortran
    integer, &
    codimension[5,0:*] :: sc
    ```
  - UPC shared objects with maximal or minimal asymmetry:
    ```fortran
    shared [0] int loc[10]
    // all of loc on thread 0
    // but still shared
    shared [*] int d[10]
    // distribute d as evenly
    // as possible
    ```

- Support for shared objects of derived type
  - both CAF and UPC
  - static type components
  - dynamic type components (with usage restrictions)

- Object oriented programming
  - CAF only
  - many restrictions when combined with coarrays
  - implementations not yet mature
Accessing shared data locally
(fastest access method)

CAF
- omit square brackets
- restores sequential semantics
- implementation may optimize as for sequential code

```
integer :: a(3)[*], b
b = 0
do i=1, 3
   b = b + a(i)
end do
```

the expression

```
b = b + &
a(i)[this_image()]
```

has the same semantics (but will probably execute slower)

UPC
- either access directly

```
shared int A[3][THREADS];
int B = 0, i;
for (i=0; i<2; i++) {
   B += A[i][MYTHREAD];
}
```

- or cast to local pointer \( \rightarrow \) better performance, guarantees no cross-image references are performed

```
shared int A[3][THREADS];
int B = 0, i;
int *A_loc;
A_loc = (int *) A;
for (i=0; i<2; i++) {
   B += A_loc[i];
}
```

on thread 2, A_loc selects
A[0][2]
A[1][2]
A[2][2]
**Simplest example:**
- collective scatter: each thread/image executes one statement implying data transfer

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

**Notes:**
- initializations of a (and q) omitted – there’s a catch here …
- `upc_threadof(&a[j][i])` for UPC tells you on which thread the argument lives
Race conditions – need for synchronization

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

serial semantics:
- execution sequence

parallel semantics:
- relaxed consistency
- unordered segments
- explicit synchronization by user required to prevent races

Imposed by algorithm:
- RaU ("reference after update") is correct

Global barrier:
- enforce ordering of segments – on all images

```fortran
integer :: b(3), a(3)[*]
a = c
b = a(:) [q]
```

```upc
for (...) a[n][i]= ...; upc_barrier;
for (...) b[i]=a[q][i];
```
**Example:**
- Iterative solvers require repeated evaluation of matrix-vector product

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

- Calculate only a block of \( b \) on each task (but that completely)
- Color codes assignment of data to tasks
- Grey: replicated data

**Alternatives:**
- Cyclic, block-cyclic
- Column, row and column

**Memory requirement:**
- \( n^2 / \langle \# \text{ of tasks} \rangle + n \) words per image/thread
- Load balanced
both codes use an external function `matval` to calculate matrix elements

compared to Fortran: different index order, same storage order, zero based
Data distribution: Changes to declarations

**Assumption:** MB == N / (# of tasks)
- dynamic allocation more flexible
- if mod(N, # of tasks) > 0 \(\rightarrow\) conditioning required

**CAF:**

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

- no point in declaring \texttt{Mat} as a coarray \(\leftarrow\) no communication between blocks required
- \texttt{B} must be a coarray so results can be broadcast to other images later if needed there

**UPC:**

```c
float V[N];
shared [MB] float Mat[N][N];
shared [MB] float B[N];
```

- shared declaration for \texttt{Mat}: not necessary, but coding easier to handle
- \texttt{Mat} and \texttt{B} are \textit{globally} addressed (no evidence of affinity in executable statements)
- tradeoff vs. performance possible in some situations

block size may have implementation limits

© 2010-2011 LRZ
**Work sharing: Assignment and $M^*v$ processing**

### CAF:
- Need to calculate global row index from local iteration variable (or vice versa)

```fortran
do icol=1,N
  do i=1,MB ! i is local index
    irow = (this_image()-1)*MB+i
    mat(i,icol) = &
      matval(irow,icol)
  end do
end do

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

- Degenerates into serial version of code for 1 image

### UPC:
- Global entities $\rightarrow$ no need for index transformation
- Work sharing construct: restrict processing to local portion on each thread

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

cblas_sgemv(CblasColMajor,
  CblasNoTrans,MB,N,1.0,
  (float *)Mat,MB,V,1,0.0,
  (float *)B,1);
```
Comments on `upc_forall`:
- Integrates data affinity to threads with loop construct.
- Fourth argument is an affinity expression:
  1. An integer \( i \) → execute if \( i \% \text{THREADS} == \text{MYTHREAD} \)
  2. A global address (as in example) → execute if \( \text{upc_threadof}(&...) == \text{MYTHREAD} \)
  3. `continue` or empty → all threads (use for nested `upc_forall`)
- Beware multiple shared entities with incommensurate block sizes inside code block → performance hit due to communication.

Casting to a local pointer:
- See earlier slide.
- Function arguments:
  - Type safety for prototype specifying a local pointer.
  - Local part of shared array is passed to a local formal parameter (this can turn out to be NULL for asymmetric objects!)

© 2010-2011 LRZ
UPC: One-sided memory block transfer

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

- **Note:**
  - CAF array transfers should do this by default (and in a more type safe manner)

**prototypes from `upc.h`**

```c
void upc_memcpyp(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);
```
Completing the \( M^*v \): Broadcast results to all tasks

Assumption: update \( V \) on each task with values from \( B \)

CAF:
- pull ...

\[
call \ sgemv(...) \\
sync \ all \ ! \ remote \ B \ available \\
do \ m=1, \ num\_images() \\
\quad V((m-1)\times MB+1:m\times MB) = B(:)[m] \\
end \ do \\
: ! use \( V \) again
\]

- \( \ldots \) or push (\( V \) must be a coarray then)

\[
call \ sgemv(...) \\
sync \ all \ ! \ assure \ reads \ on \ V \ done \\
me = this\_image() \\
do \ m=1, \ num\_images() \\
\quad V((me-1)\times MB+1:me\times MB)[m] = B \\
end \ do \\
sync \ all \\
: ! local \ access \ to \ V
\]

UPC:
- must use \texttt{upc\_memget()} for existing declarations
- could also use a loop with a scalar assignment in body

\[
upc\_barrier; \\
for \ (m=0; \ m<\text{THREADS}; \ m++) \{ \\
\quad upc\_memget(&V[m\times MB],&B[m\times MB],MB\times sizeof(float)); \\
\} \\
// ! use \( V \) again
\]

- cannot simply declare \( V \) shared since locality required by library call
Timing results (N=20000, MB=2500)

- **Hardware**
  - 2 sockets with 4 Nehalem cores (2.53 MHz E5540)

- **BLAS**
  - MKL 10.2 for serial SGEMV

- **CAF / serial Fortran**
  - Intel 12.0.1 (both)

- **UPC / C**
  - BUPC 2.12.0 / gcc 4.4

- **Number of Tasks**: 8

- **GCC compiler-specific optimization**
  - -mtune=barcelona (serial code only)

### Variant | Execution time (s)
--- | ---
serial Fortran | 4.8
CAF w/o Bcast | 1.42
CAF w Bcast | 1.48 (speedup 3.2)
serial C | 26.6 (gcc)
UPC w/o | 13
UPC w Bcast | 13
UPC Fortran style | 7.8 (speedup 3.4)

### UPC:
- assignment did not use cast to local pointer
- **upc forall** as inner loop (not so good)
- BUPC cannot pass on gcc optimization options (?)
Collectives: Reductions

- **M*v with column blocks**
  - only partial sums are formed by each task
  - need to sum these over all tasks to obtain final result

- **Reduction concept:**
  - distributed set of objects
  - operation defined on type

  ![Reduction Diagram]

- **Reduction types**

<table>
<thead>
<tr>
<th>Numeric</th>
<th>Logical</th>
<th>User-defined function</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/UC</td>
<td>L/UL</td>
<td></td>
</tr>
<tr>
<td>S/US</td>
<td>F/D/LD</td>
<td></td>
</tr>
<tr>
<td>I/UI</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Operations:**
  - are constants of type `upc_op_t`

- **Availability:**
  - UPC only
  - CAF: need to roll your own for now (future CAF may include this feature)
Collectives: Reduction prototype

```c
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);
```

- **src** and **dst** may not be aliased
- replace **T** by type (C, UC, etc.)
- function argument will be **NULL** unless user-defined function is configured via **op**

**Further UPC calls:**
- data redistribution collectives
- prefix reductions (each thread gets an incremental partial sum)
Dynamic entities: Pointers

- **Remember pointer semantics**
  - different between C and Fortran
  
  ```
  Fortran  
  <type>, [dimension (:,[,:,...])], pointer :: ptr  
  ptr => var  ! ptr is an alias for target var
  
  C
  <type>*ptr;
  ptr = &var;  ! ptr holds address of var
  ```

- **Pointers and PGAS memory categorization**
  - both pointer and entity pointed at might be private or shared
    - → 4 combinations theoretically possible
  
  - **UPC**: three of these combinations are realized
  - **CAF**: only two of the combinations allowed, and only in a limited manner
    - → aliasing is allowed only to local entities
Pointers continued ...

**CAF:**

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

type :: ctr
  integer, pointer :: p2(:)
end type

type(ctr) :: o[*]

integer, target :: i2(3)
```

- entity "o" typically asymmetric

**UPC:**

```fortran
int *p1;
shared int *p2;
int *shared p3;
int *shared p4;

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

- pointer to shared: addressing overhead

**UPC: four combinations:**
- p1: private pointer to private memory
- p2: private to shared
- p3: shared to private
- p4: shared to shared

**Problem:** where does p3 point?

all other threads may not reference

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

© 2010-2011 LRZ

PGAS Languages: Coarray Fortran/Unified Parallel C
UPC: Shared Pointer blocking and casting
(have already seen casts to local)

Assume 4 threads:

```
shared int *p1;
shared [2] int *p2;
```

Thread 0 1 2 3


if (MYTHREAD == 1) {
  p1 = &A[0]; p1 += 4;
  p2 = &A[0]; p2 += 4;
}

Block size:
- is a property of the shared entity used
- can cast between different block sizes
- pointer arithmetic follows blocking ("phase") of pointer!

© 2010-2011 LRZ
Dynamic entities: Memory management

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

**CAF:**

```fortran
integer, allocatable :: id(:)[:,]
allocate(id(100)[*])
```

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

```fortran
deallocate(id)
```

- deallocation: synchronizes all images before carried out

**UPC:**

```fortran
shared [100] int *id;
id = (shared [100] int *) \nupc_all_alloc( \nTHREADS,100*sizeof(int));
```

- 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

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

is not collective
Improving the M*v example

- **Flexibility of coding:**
  - use dynamic allocation
  - use arbitrary number of tasks

- **Conditioning:**
  - adjust block size

```fortran
real, allocatable :: &
    Mat(:,:),V(:),B(:)[:,:]
:
nim = num_images()
me = this_image()
MB = N / nim
if (mod(N, nim) /= 0) MB = MB + 1
MB_loc = MB
if (nim == me) MB_loc = &
    N - MB*(num_images() - 1)
allocate(Mat(MB,N), &
    V(MB*num_images()),B(MB)[*])
```

- all but the last block have the same size
- last block may be smaller
- need to replace MB by MB_loc in some (but not all) places in the code
- deallocate at the end

- **UPC-specific issue:**
  - need to go to block cyclic distribution because block size must be compile time constant
  - this is no problem for assignment, but 
cblas_sgemv() must be invoked more than once on each thread

a bit larger than necessary → broadcast easier to handle
Non-synchronizing memory management facilities

**CAF:**

- allocatable type components

```fortran
type :: ragged
    integer, allocatable :: a(:)
end type

type(ragged) :: o[*]
```

- remote accesses require synchronization

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

- must be local

- can be asymmetric

- dynamic allocation: could be in shared space

**UPC:**

- two routines, both called by individual threads …

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

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

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

- memory allocated in shared space on calling thread (→ single block with affinity to that thread)

- pointer to first element of allocated memory

- both calls require shared pointers to enable data transfers

... assuming b is large enough
Image subsets

- Sometimes, it is sufficient to synchronize only a few images.

UPC does not explicitly support this; note that in

```c
upc_barrier exp;
```

exp only serves as a label, with the same value on each thread.

More than 2 images:

- Need not have the same image set on each image.
- But: eventually all image pairs must be resolved, else deadlock occurs.

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

Executing image implicitly included in image set.
Example: Simple Master-Worker

- **General Scenario:**
  - one image sets up data for computations
  - others do computations

- **Minimal example in M*v:**
  - read problem size $N$ from stdin
  - in CAF, stdin (officially) only available on image 1

```fortran
integer :: N[*]

if (this_image() == 1) then
    read(*,*) N
    sync images ( * )
else
    sync images ( 1 )
    N = N[1]
end if
```

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

© 2010-2011 LRZ

PGAS Languages: Coarray Fortran/Unified Parallel C
Partial synchronization: Best Practices

- **Localyze complete set of synchronization statements**
  - **avoid** interleaved subroutine calls which do synchronization of their own

```fortran
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

```fortran
subroutine mysub(…)
:
if (this_image() == 2) sync images (/ 1 /)
:
end subroutine
```

- May produce wrong results even if no deadlock occurs
Final remarks

**Implementations**
- Cray Fortran (XT and vector)
- g95 (Fortran 9x level)
- Intel V12.0 compiler
- gfortran single-image (4.6)

**Not covered at all here:**
- mutual exclusion (locks, critical regions)
- rules for subprogram invocation and program termination
- UPC split phase barrier
- UPC strict vs. relaxed memory semantics
- UPC parallel I/O
- atomic functions and light-weight synchronization
- distributed structures (object-based programming patterns)

**Future developments:**
- teams
  - load imbalanced problems (partial synchronization)
  - recursive / hybrid / MPMD-like
- asyncs and places
  - memory and function shipping
  - support for accelerator devices?
- collective calls in CAF
  - maybe even asynchronous?
- further atomic functions
- process topologies in CAF
  - more general abstraction than multiple coindices
- global variables and shared pointers in CAF
  - increase programming flexibility
- split-phase barrier in CAF
- parallel I/O in CAF
- CAF+UPC interoperation
Thank you for your attention!

Any questions?