Parallel Programming with CoArray Fortran (CAF) and Unified Parallel C (UPC)

Richard Walsh
CUNY, High Performance Computing Center
New York, NY
UPC/CAF Tutorial Outline

1. UPC/CAF and MPI Compared
   - Single Program Multiple Data (SPMD) vs. Message Passing
   - Distributed Shared Memory (DSM) vs. Message Passing
   - Partitioned Global Address Space (PGAS) vs. Message Passing

2. Parallel Execution Model

3. Distributed Shared Memory Model

4. A Simple Example

5. Declaring and Assigning Shared Data

6. Process Synchronization and Memory Consistency

7. Special Functions and Operators

8. Work Sharing in Loops

9. Using Pointers

10. Dynamic Memory Allocation

11. IO Model and Constructs

12. Exercises
1. UPC/CAF and MPI Compared
2. Parallel Execution Model
3. Distributed Shared Memory Model
4. A Simple Example
5. Declaring and Assigning Shared Data
6. Process Synchronization and Memory Consistency
7. Special Functions and Operators
8. Work Sharing in Loops
9. Using Pointers
10. Dynamic Memory Allocation
11. IO Model and Constructs
12. Exercises
UPC/CAF and MPI Compared

MPI Assumes Physically Distinct Local Address Spaces

- Processor (p1)
- Memory (m1)

MPI Moves Packaged Data Memory-to-Processor then Processor-to-Memory in Messages

- Interconnect

UPC/CAF Assume Logically Partitioned Global Address Space (PGAS)

- Processors
- Memory (m1, m2, m3, ..., mn)

UPC/CAF Move Raw Data Memory-to-Memory* with Reads and Writes

Interconnect is hidden behind abstraction

MPI Message Passing Model (interconnect’s presence is implied by syntax)

UPC/CAF Distributed Shared Memory (DSM) Model (interconnect is abstracted away in syntax)
MPI Message Passing Interface (MPI) Model

- Generalizable to any parallel architecture
  - Process and address spaces assumed independent
- Current HPC parallel programming de facto standard
- Independent processes
  - That interact formally when necessary
- Library-call based API with collectives, can be clumsy/bulky
  - Fortran, C, and C++ bindings
- 2- to n-sided, engaged transactions among processors
  - Point-to-point and collectives
  - Data is packaged by processors, “messages”
- Includes both implicit and explicit synchronization points
- Potential productivity and performance disadvantages
  - Coding often complex, with large footprint
  - Subroutines unavailable to compiler, higher latencies
  - More data copying/buffering, harder to scale
  - Fine grained algorithms, hard (impossible) to code/scale
UPC/CAF Distributed Shared Memory (DSM) Model

- Not naturally generalizable to “share nothing” architectures
  - Can make do without (fake it), but likes support for:
    - Remote direct memory address (RDMA)
    - Global address space, partitioned (PGAS) or not
- Growing support and interest, but use is still limited
- Independent processes
  - That stay out of each others way
- Economical extensions to familiar language syntax
  - CAF in Fortran 2008; UPC extends ISO 1999 C
  - Try to avoid library calls (CAF does better job)
- 1-sided, largely independent, disengaged transactions
  - “Direct” remote memory reads and writes
- Generally requires programmer explicit synchronization
- Potential productivity and performance advantages
  - Coding more intuitive, smaller footprint, introduce incrementally
  - Compiler can* optimize instructions, lower/hidden latency
  - Less data copying, easier to scale
  - Fine-grained algorithms code
UPC/CAF and MPI Compared

Model Explicit

Remote Data Exchanges

Model Implicit

PGAS, Distributed Shared Memory: UPC, CAF

Shared Memory Fork and Join: OpenMP

Message or Data Passing: MPI, PVM

Work Sharing Within Loops

Model Implicit

Model Explicit
UPC/CAF and MPI Compared

- **Python, Tcl/Tk, Matlab**
  - High Productivity, Expressiveness (especially parallel)
- **C/Fortran and MPI**
  - Low Language Performance (especially parallel)
- **UPC/CAF (cluster)**
  - High Language Performance (especially parallel)
  - Availability, Compiler quality
- **UPC/CAF (Cray X1)**
  - High Performance Computing Center
  - Hardware support, Compiler quality
- **(Cray XE6)**
- **Assembly Code**

Ideal

16 Feb 2011

High Performance Computing Center
The City University of New York
www.csi.cuny.edu/cunyhpc
### UPC/CAF and MPI Compared

<table>
<thead>
<tr>
<th></th>
<th>CAF</th>
<th>UPC</th>
<th>MPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalability/Performance</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Fine Grained Data Control</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Standard and Portable</td>
<td>*</td>
<td>**</td>
<td>--</td>
</tr>
<tr>
<td>Ease of Develop/Support</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Fortran Specification</td>
<td>✓</td>
<td>--</td>
<td>✓</td>
</tr>
<tr>
<td>C/C++ Specification</td>
<td>--</td>
<td>✓</td>
<td>***</td>
</tr>
<tr>
<td>DSM/PGAS Ready</td>
<td>✓</td>
<td>✓</td>
<td>--</td>
</tr>
</tbody>
</table>

* Part of Fortran 2008  
** Extension of ISO C 1999  
*** Does not support C++
real :: data(0:nx+1,0:ny+1,0:nz+1)
real :: local_result, global_result
integer :: mype, ier, nx, myright, myleft
integer :: stag, rtag, status, iz

: call MPI_init (ier)
call MPI_comm_rank (MPI_COMM_WORLD, mype, ier)

! Exchange halo cell data with “left” and “right” processors
do iz = 1, nz
  stag = stag + 1
  rtag = rtag + 1
  call MPI_sendrecv (data(1,ny,iz), nx, MPI_REAL8, myright, stag
                     data(1,0,iz), nx, MPI_REAL8, myleft, rtag,
                     MPI_COMM_WORLD, status, ier)
  stag = stag + 1
  rtag = rtag + 1
  call MPI_sendrecv (data(1,1,iz), nx, MPI_REAL8, myleft, stag
                     data(1,ny+1,iz), nx, MPI_REAL8, myright, rtag,
                     MPI_COMM_WORLD, status, ier)
enddo

! Do some useful work on my new halo cell data then sum results
local_result = use_data(data, nx, ny, nz)

call MPI_reduce( local_result, global_result, 1, MPI_REAL8, MPI_SUM,
                0, MPI_COMM_WORLD, ier)

if (mype .eq. 0) print *, ‘Final = ‘, global_result

call MPI_finalize (ier)
CAF Halo code, with extensions marked in red

```fortran
real :: data(0:nx+1, 0:ny+1, 0:nz+1)[*], local_result, global_result[*]
integer :: myleft, myright, me, ix, iz

me   = this_image()
    :
    (setup)
    :
    ! Exchange halo cell data with “left” and “right” processors
    do  iz  =  1, nz
      do  ix  =  1, nx
        data (ix, 0, iz) = data (ix, ny, iz)[myleft]
        data (ix, ny+1, iz) = data (ix, 1, iz)[myright]
      end do
    end do

    ! Do some useful work on my new halo cell data then sum results
    local_result = use_data(data, nx, ny, nz);

    critical
      global_result[1] = global_result[1] + local_result
    end critical

    sync all

    if (me .eq. 1) print *,’Final = ‘, global_result
```
UPC Halo code, with extensions marked in red

```
shared double data[nz+2][ny+2][nx+2][THREADS], global_result;
int myleft, myright, ix, iz, local_result;
upc_lock_t *lptr = upc_all_lock_alloc();
  :
    (set up)
  :
! Exchange halo cell data with “left” and “right” processors
for (iz=1; iz<=nz; iz++)
  for (ix=1; ix<=nx; ix++) {
    data[iz][0][ix][MYTHREAD] = data[iz][ny][ix][myleft];
    data[iz][ny+1][ix][MYTHREAD] = data[iz][1][ix][myright];
  }
  :
! Do some useful work on my new halo cell data then sum results
local_result = use_data(data, nx, ny, nz);

upc_lock (lptr);
  global_result += local_result;
upc_unlock (lptr);

upc_barrier;

if (MYTHREAD == 0) printf(“Final = %f\n”, global_result);
```
1. UPC/CAF and MPI Compared
2. Parallel Execution Model
3. Distributed Shared Memory Model
4. A Simple Example
5. Declaring and Assigning Shared Data
6. Process Synchronization and Memory Consistency
7. Special Functions and Operators
8. Work Sharing in Loops
9. Using Pointers
10. Dynamic Memory Allocation
11. IO Model and Constructs
12. Exercises
• Independent processes run copies of a Single Program: SP, but …
  – All processes start and finish together (no spawning), but work at own pace (a race)
  – Each process is distinguished by an ID (UPC: MYTHREAD; CAF: this_image() )
  – All processes execute all code, unless ID-based conditionals are used
  – Programmer aligns progress through code via synchronization

![Diagram showing parallel processes]

UPC/CAF serializes/syncs-in parallel processes as needed

versus

OpenMP parallelizes/fans-out a serial process as needed

• Process Multiple Data: shared data (partitioned, global); private data (distinct, local):
  – UPC/CAF are Single Program Multiple Data (SPMD) models, that …
  – UPC/CAF rely on a partitioned, but Global Address Space (PGAS)
    • Remote, shared locations must be reliably addressable
    • Though not equally addressable (NUMA)
  – UPC/CAF extend familiar C and Fortran syntax
    • Minimal modification to express remote addressability
Process count and identification basics in UPC/CAF:

- UPC processes are called **threads**
  - Defined *statically* at compile time (16 here)
    - `cc -h upc X 16 -o upc_prog upc_prog.c` (Cray XE6 compiler)
  - Or, *dynamically* at run-time (8 here, leave –X off compile)
    - `aprun -n 8 upc_prog` (Cray XE6 execution)
  - Number of threads given by keyword **THREADS**
  - Thread ID given by keyword **MYTHREAD** (0 to THREADS-1)
  - Thread of a shared array element returned by `size_t upc_threadof(shared void *)`

- CAF processes are called **images**
  - Defined *statically* at compile time (16 here)
    - `ftn -h caf -X 16 -o caf_prog upc_prog.f` (Cray XE6 compiler)
  - Or, *dynamically* at run-time (8 here, leave –X off compile)
    - `aprun -n 8 caf_prog` (Cray XE6 execution, 8 images)
  - Number of images returned by function **num_images()**
  - Image ID returned by function **this_image()** (1 to num_images())
  - Image of a co-array member returned by **image_index(co-array, (co-subs/))**
    - Part of standard, but still deferred on the Cray
UPC example:

```c
#include <upc_relaxed.h>
#include <stdio.h>

void main() {
  int drem, i;
  drem = MYTHREAD%2;
  for (i=0; i<THREADS; i++) {
    if( i == MYTHREAD ) {
      if ( drem == 0 ) {
        printf("My thread (%d) of %d is even.\n", MYTHREAD, THREADS);
      }
      else {
        printf("My thread (%d) of %d is odd.\n", MYTHREAD, THREADS);
      }
    } 
  }
  upc_barrier;
}
```

CAF example:

```c
program whoami ()
  integer :: drem, i
  drem = mod(this_image(),2)
  do i=1,num_images()
    if(i.eq.this_image()) then
      if(drem.eq.0) then
        print *, "My image (", this_image(), ") of ", num_images(), " is even."
      else
        print *, "My image (", this_image(), ") of ", num_images(), " is odd."
      endif
    endif
    sync all
  enddo
end
```

All threads/images execute each iteration of for/do loops; conditional (if) blocks determine which part of each.

What are the synchronization barriers for … ??
1. UPC/CAF and MPI Compared
2. Parallel Execution Model
3. **Distributed Shared Memory Model**
4. A Simple Example
5. Declaring and Assigning Shared Data
6. Process Synchronization and Memory Consistency
7. Special Functions and Operators
8. Work Sharing in Loops
9. Using Pointers
10. Dynamic Memory Allocation
11. IO Model and Constructs
12. Exercises
UPC/CAF: Memory Model

UPC/CAF Memory Model:

- Defines process-private (local); process-shared (global) data objects
  - Models need underlying local and global addressing capability
  - Shared (global) objects have process local sub-objects

- Underlying implementation details of shared memory are **not specified**
  - Implementations can have a hardware or software emphasis
    - Compile directly to global memory instructions (old Cray X1E)
      - Source-to-instructions
    - Compile to interconnect-specific global memory library (Cluster, Cray XE6)
      - Source-to-(source + library calls)
      - Libraries rely on (are limited by) RDMA capability of interconnect
  - Implementations can be constrained or unconstrained
    - **Aligned, symmetric** shared memory space
      - CAF on Crays uses the symmetric heap
      - Requires shared objects have equal lengths/ extents
    - **Unaligned, asymmetric** shared memory space
      - UPC allows variable lengths/ extents
UPC/CAF Memory Model (cont):

• **Private (local) object** properties
  - Multiple, separate, process-local copies with same name
  - Typically addressed only by the “owning” process
  - You can (try to) address private objects from remote processes, but …

  • Address alignment among process private objects is **not** implied/required/guaranteed, temporality or spatially
UPC/CAF Memory Model (cont):

- **Shared (global) object** properties
  - **Affinity** (locality in CAF) of shared sub-objects to each process
    - Performance implications
  - **Extent** or object length on each process
    - Must be equal in CAF
  - **Addressability** from any process, using same name
    - Exactly how is implementation dependent
    - At a known distance from a common base address in CAF on Cray
      - Data is referred to as “symmetric”, “aligned”
    - Cray imposes symmetry for UPC also
      - Not required by standard
  - **Distribution** pattern across processes
    - UPC ==> *implicit* in standard C array syntax
      - Set by a blocking factor in shared data declaration
    - CAF ==> *explicit* in CAF co-array syntax
      - Local shape of co-array repeated on each image
UPC/CAF: Memory Model

UPC memory layout

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread m-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Private</td>
<td></td>
<td></td>
<td>Private</td>
</tr>
<tr>
<td>p[0] ... p[n]</td>
<td>p[0] ... p[n]</td>
<td>p[0] ... p[n]</td>
<td>p[0] ... p[n]</td>
</tr>
<tr>
<td>base</td>
<td></td>
<td></td>
<td>base</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>base + padd</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>base + padd + sadd</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Private</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>base + padd</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>base + padd + sadd</td>
</tr>
<tr>
<td>Shared</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s[0][0] s[1][0]</td>
<td>s[0][1] s[1][1]</td>
<td>s[0][2] s[1][2]</td>
<td>s[0][m-1] s[1][m-1]</td>
</tr>
<tr>
<td>s[2][0]</td>
<td>s[2][1]</td>
<td>s[2][2]</td>
<td>s[2][m-1]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>affinity, thread 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

private data is replicated not distributed
extents, can be variable

CAF memory layout

<table>
<thead>
<tr>
<th>Image 1</th>
<th>Image 2</th>
<th>Image 3</th>
<th>Image m</th>
</tr>
</thead>
<tbody>
<tr>
<td>Private</td>
<td></td>
<td></td>
<td>Private</td>
</tr>
<tr>
<td>p(1) ... p(n)</td>
<td>p(1) ... p(n)</td>
<td>p(1) ... p(n)</td>
<td>p(1) ... p(n)</td>
</tr>
<tr>
<td>base</td>
<td></td>
<td></td>
<td>base</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>base + padd</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>base + padd + sadd</td>
</tr>
<tr>
<td>Shared</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s(1,1)[1]</td>
<td>s(1,1)[2]</td>
<td>s(1,1)[3]</td>
<td>s(1,1)[m]</td>
</tr>
<tr>
<td>s(2,1)[1]</td>
<td>s(2,1)[2]</td>
<td>s(2,1)[3]</td>
<td>s(2,1)[m]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>s(I,J)[1]</td>
<td>s(I,J)[2]</td>
<td>s(I,J)[3]</td>
<td>s(I,J)[m]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>affinity, image 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

private data is replicated not distributed
extents, must be identical

16 Feb 2011
1. UPC/CAF and MPI Compared
2. Parallel Execution Model
3. Distributed Shared Memory Model
4. **A Simple Example**
5. Declaring and Assigning Shared Data
6. Process Synchronization and Memory Consistency
7. Special Functions and Operators
8. Work Sharing in Loops
9. Using Pointers
10. Dynamic Memory Allocation
11. IO Model and Constructs
12. Exercises
UPC simple example sub-program

- Shows the essential elements of the UPC extension
  - shared and blocking factor [#] declaration qualifiers
  - THREADS and MYTHREADS keywords

- Program sums two shared arrays and assigns the result
  - How are the arrays distributed?
  - What threads execute which iterations of the for() loop?
  - Is there remote data referencing occurring here?

```c
#include <upc.h>

#define N 100
shared [1] int v1[N][THREADS], v2[N][THREADS];
shared [1] int v1pv2[N][THREADS];

void vsum()
{
  int i;
  for (i=0; i<N; i++)
    v1pv2[i][MYTHREAD] = v1[i][MYTHREAD] + v2[i][MYTHREAD];
}
```
• **CAF simple example program**
  - Shows the essential elements of the CAF extension
    • CoArray [*] global/shared data declaration qualifier
    • `this_image()` image identification/indexing function
  - Program sums two global/shared arrays and assigns the result
    • How are the arrays distributed?
    • What images sum which data?
    • Is there remote data referencing occurring here?

```fortran
subroutine vsum(n, v1, v2, v1pv2)
  integer :: n
  integer, dimension(n)[*] :: v1, v2, v1pv2
  integer :: cosub ! inelegant, clumsy, slow (explicit local co-array)
  cosub = this_image(v1) ! get co-subscript for this image
  v1pv2(1:n)[cosub] = v1(1:n)[cosub] + v2(1:n)[cosub] ! sum using array syntax (no do loop)
  v1pv2(1:n) = v1(1:n) + v2(1:n) ! elegant, idiomatic, fast (implicit local co-array)
  v1pv2 = v1 + v2 ! still more compact (same results)
end subroutine vsum
```