‘Advanced’ HPC Parallel Programming Models

CoArray Fortran (CAF) and Unified Parallel C (UPC),

An Brief Introduction and Comparison To the Message Passing Interface (MPI)

Richard Walsh
CUNY, High Performance Computing Center
New York, NY
1. **Getting to the Present; One view of the Future**

2. **PGAS (UPC/CAF) and MPI Compared**

3. **UPC/CAF Parallel Execution Model**

4. **UPC/CAF Distributed Shared Memory Model**

5. **A Simple UPC/CAF Example Program**
Parallelism, basic truths (hardware) …

- Intrinsic to most scientific algorithms, yet …
- Absent in almost all early computer designs
  - von Neumann model
- Presented as a hardware design feature first
- Occurs throughout today’s HPC hardware stack
  - (ILP, DLP, TLP, NLP: ‘the parallel—isms’)
- HPC hardware stack is changing constantly
  - Moving target for parallel software
Parallelism, basic truths (software) …

• In programming models, reflects its position in the hardware stack, … consequently …

• Parallel programming models are stack-specific, varied in design and function, … moreover …

• Parallel applications make their own unique demands on the underlying parallel hardware

• Parallel programming is difficult from an algorithmic, coding, and performance perspective
Given all this, how does one design a parallel programming model that is:

--portable,
--comprehensive in scope,
--preserves current software investments,
--delivers performance, and is …
--(relatively) easy to use … ??

What is the answer, the best model … ??
None, all are bad … ;-) … serial programming is easier, avoid parallelism as long as possible. How?

• Higher clock to get more serial performance
  – This approached ended for almost all in ~2004
  – Power/heat grow non-linearly with clock

• Hide it from the programmer: in the instruction set, in the hardware, in the compiler, in the runtime environment
  – Explicitly Parallel Instruction sets (VLIW, Itanium’s EPIC)
  – Wide-processors with ILP and O-o-O execution
  – Vectorizing, thread-parallelizing compilers (Cray, etc.), BUT …

Finally, became too complicated to hide everything, and also get performance, so …
MPI*, an explicit, conservative approach ...

- Not a new language, but a library (API)
  - Worked with standard languages (Fortran, C, C++)
- Made few assumptions about hardware
  - Needed only some interconnect between nodes
- Addressed distributed memory parallelism only
  - Newest source of hardware parallelism (clusters, MPP)
- Local memory parallelism left as a distinct domain
- Put the programmer in charge ... lots of detail ... !!
  - Like a parallel assembly language ... ?? ... ;-) ... 

* We will ignore PVM for this talk ...
The question is, what’s next ... ??

MPI is now ...

--nearly 20 years old,
--the *de facto* parallel standard,
--has gone through several revisions,
--and has been a success, but ...

Is MPI a kind of ‘assembly language’ of parallel programming to be displaced ... or a more persistent standard like Fortran or C?
Forecast a Future from the Past ...

**Past**
- Assembly (IBM)
- Assembly (UNIVAC)
- Assembly (CDC)

**Compiled Standard Serial Languages**

**Present**
- MPI Standard
- Local Memory Parallelism (GPU-Multicore)
- OpenMP
- Compiled Standard Serial Languages with Parallel Extensions

**Future ... ??**
- CUDA, OpenCL
- Distributed Memory Parallelism (MPP-Cluster)
- UPC
- CAF
One Notion of the Future …

• LM parallelism via *compiler directives*
  – Captures all parallelism within a node (GPU, CPU-multicore)
  – Historical precedents: Cray vectors, OpenMP threads
  – Current implementations (Cray, PGI)
  – Compatible with MPI, allows gradual change
  – Works with standard languages (C and Fortran)

• DM parallelism via *language extensions*
  – Captures all parallelism among nodes
  – Current implementations (UPC and CAF)
  – Compatible with MPI, allows gradual change
  – Works with standard languages (C and Fortran)
Let’s focus is on the distributed memory (DM) layer and the *Partitioned Global Address Space* (PGAS) extensions to standard Fortran and C:

- CoArray Fortran (CAF)
- Unified Parallel C (UPC)

We’ll confer the label of ‘advanced’ on these models today to match the brochure’s title for my talk.
1. Getting to the Present; One View of the Future
2. PGAS (UPC/CAF) and MPI Compared
3. UPC/CAF Parallel Execution Model
4. UPC/CAF Distributed Shared Memory Model
5. A Simple UPC/CAF Example Program
**MPI, Message Passing Interface (MPI) Model**

- Generalizable to “share nothing” parallel architectures
  - 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, more clumsy/bulky
  - Fortran, C, and C++ bindings
- 2- to n-sided, engaged transactions among processors
  - Point-to-point or collective copies to partner processor(s)
  - Data is packaged by processors, “messages”, (overhead)
- Includes both implicit and explicit synchronization points
UPC/CAF, Dist. Shared Memory (PGAS) Model

- Not easily generalizable to “share nothing” architectures
  - Can make do without (fake it with MPI), 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 (mostly)
- Economical extensions to familiar language syntax
  - CAF is part of Fortran 2008; UPC extends ISO 1999 C
  - Tries to avoid library calls (CAF does better job)
- 1-sided, largely independent, disengaged transactions
  - “Direct” remote memory reads and writes (unacknowledged)
- Generally requires programmer explicit synchronization
UPC/CAF and MPI Compared

Different memory abstractions ...

MPI Assumes Physically Distinct Local Address Spaces

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

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

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

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

UPC/CAF Distributed Shared Memory (DSM) Model (interconnect is abstracted away in PGAS syntax)
Different communication paradigms …

MPI multiple-copy vs. PGAS single-copy semantics

Two-sided, Multi-copy, Memory and CPU-intensive
(MPI_SendReceive())

One-sided, “Zero-copy”, CPU-limited
(PGAS get operation)
UPC/CAF and MPI Compared

Different communication paradigms …

Simplified version … ;-) …

Two-sided MPI, both processes (CPUs and Memory) participate

One-sided PGAS, only one process (CPU and Memory) participates
Different coding paradigms, halo code example …

- Domain decomposed for parallel execution
  - one domain per CPU
- Subdomains must exchange information across the boundaries, called
  - halo communication
- Halo communication is often the bulk of inter-processor communication in parallel code

Halo points; filled from neighbors’ interiors

Interior points; used to fill neighbors’ halos
MPI Halo code, with MPI code marked in red

```fortran
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 ! Send and receive a vector of nx values at a time
  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)
endo
:
! 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)
```
CoArray Footprint, Halo Code

CAF Halo code, with extensions marked in red

```
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

```c
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
”, global_result);
```
Productivity and performance …

- **UPC/CAF advantages**
  - Intuitive coding, smaller footprint, hides detail
  - Elegant extension to standard languages
  - Introduce incrementally to MPI code
  - Less data copying, lower latencies, better scaling
  - Fine-grained algorithms code more easily

- **MPI advantages**
  - A second generation model, portable standard
  - Works with standard languages
  - Interconnect hardware agnostic (must code with this in mind)
Productivity and performance ...

- **UPC/CAF disadvantages**
  - First generation, still limited support (but growing)
  - Interconnect hardware dependent performance
    - MPI-equivalent performance should be worst-case

- **MPI disadvantages**
  - Coding often complex, larger footprint, lots of detail
  - More data copying/buffering, higher latencies, harder to scale
  - Fine-grained algorithms, hard (impossible) to code/scale
UPC/CAF and MPI Compared

- **Language Productivity, Expressiveness (especially parallel)**
  - Low: Assembly Code
  - High: Python, Tcl/Tk, Matlab

- **Language Performance (especially parallel)**
  - Low: UPC/CAF (cluster), UPC/CAF (Cray X1)
  - High: Assembly Code

- **Availability, Compiler quality**
  - UPC/CAF (Cray XE6)

- **Hardware support, Compiler quality**
  - UPC/CAF (Cray X1)

Ideal point:
- UPC/CAF (cluster)
- UPC/CAF (Cray X1)
- Berkeley UPC

High Performance Computing Center
The City University of New York
www.csi.cuny.edu/cunyhpc
1. Getting to the Present; One View of the Future
2. PGAS (UPC/CAF) and MPI Compared
3. **UPC/CAF Parallel Execution Model**
4. UPC/CAF Distributed Shared Memory Model
5. A Simple UPC/CAF Example Program
SPMD, *Single Program, Multiple Data*

- Each process runs its own copy of a *Single Program*
  - Processes start and finish together (no spawning)
    - Each works at its own pace, *... a race ...*
  - Processes (threads or images) distinguished by a unique ID
    - UPC: *MYTHREAD*; CAF: *this_image()*
  - Processes execute all code, *unless* ...
    - ID-based conditionals are used
  - Progress through the code aligned via synchronization

---

**UPC/CAF Parallel Execution Model**

**SPMD** versus **SMP**

- **SPMD**
  - UPC/CAF serializes/syncs-in parallel processes as needed
- **SMP**
  - OpenMP parallelizes/fans-out a serial process as needed

---

High Performance Computing Center
The City University of New York
www.csi.cuny.edu/cunyhpc
SPMD, Single Program, Multiple Data

- Each process has its own data, Multiple Data
  - Shared data (partitioned, global)
    - Local portion, remote portion
    - Reliably, but not equally addressable (NUMA), via …
    - Partitioned Global Address Space (PGAS)
  - Private data (distinct, local)
    - Familiar to the serial programmer
**UPC example (threads):**

```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);
            }
        } else {
        }
    }
    upc_barrier;
}
```

**CAF example (images):**

```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
```

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. Getting to the Present; One View of the Future
2. PGAS (UPC/CAF) and MPI Compared
3. UPC/CAF Parallel Execution Model
4. **UPC/CAF Distributed Shared Memory Model**
5. A Simple UPC/CAF Example Program
UPC/CAF Memory Model

- Defines process-private (local) data objects
- Defines process-shared (global) data objects
  - Models need underlying local and global addressing capability
  - Shared (global) objects have process local sub-objects
- Implementation details of shared memory are not specified
- Implementation, hardware or software approach:
  - Compile to global memory instructions (old Cray X1E)
    - Source-to-instructions
  - Compile to interconnect-specific global library (Cray XE6, clusters)
    - Source-to-(source + library calls)
    - Relies on (or limited by) RDMA capability of interconnect
  - Compile to MPI library (cluster)
UPC/CAF Memory Model (cont.)

- **Private (local) object** properties
  - Familiar serial-like data objects
  - 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 in different processes 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 length of shared sub-object on each process
    - Must be equal in CAF
  - **Addressability** from any shared object, using same name
    - Implementation, language dependent
    - At a fixed distance from a common base address on Cray
      - Data is referred to as “symmetric”, “aligned”
    - Cray imposes symmetry for UPC also
      - Not required by standard
  - **Distribution** pattern of shared object across processes
    - UPC ==> *implicit* in standard C array syntax (declare *shared*)
      - Set by a blocking factor in shared data declaration
    - CAF ==> *explicit* in CAF co-array syntax (square brackets, coarray[*])
      - Local shape of co-array repeated on each image
UPC/CAF: Memory Model

**UPC memory layout**
- Private: p[0] … p[n]
- Shared:
  - base + padd: s[0][0], s[1][0], s[2][0]
  - base + padd + sadd: s[0][1], s[1][1], s[2][1]

**CAF memory layout**
- Private: p(1) … p(n)
- Shared:
  - base + padd: s(1,1)[1], s(2,1)[1], ... s(l,J)[1]
  - base + padd + sadd: s(1,1)[2], s(2,1)[2], ... s(l,J)[2]

**Notations**
- **UPC extents, can be variable**
- **CAF extents, must be identical**
- **UPC data is replicated, not distributed**
- **CAF data is replicated, not distributed**

**Affinity**
- **UPC**: symmetric
- **CAF**: congruent-copy, co-subscripted
1. Getting to the Present; One View of the Future
2. PGAS (UPC/CAF) and MPI Compared
3. UPC/CAF Parallel Execution Model
4. UPC/CAF Distributed Shared Memory Model
5. A Simple UPC/CAF Example Program
**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 any 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 sub-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 any remote data referencing occurring here?

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)
    v1pv2(1:n)[cosub] = v1(1:n)[cosub] + v2(1:n)[cosub]

    ! elegant, idiomatic, fast (implicit local co-array)
    v1pv2(1:n) = v1(1:n) + v2(1:n)

    ! still more compact (same results)
    v1pv2 = v1 + v2

end subroutine vsum
‘Prediction is hard, especially about the future.’

-- Niels Bohr

**My predictions, HPC parallel programming 2025 …**

- Completely new parallel languages seem *unlikely*
- Even at performance parity, elegant extensions to standard languages will meet the drive to ease parallel coding prob.
- Intra-node, inter-node parallelism addressed separately
  - Inter-node parallelism, *language extensions* …
    - CoArray Fortran best candidate, UPC a close second
  - Intra-node parallelism, *compiler directives* …
    - OpenMP-like augmented ala PGI, Cray for GPU-Many Core
- MPI will still be in use … ;-) … as is assembly …
That concludes my talk … thanks

Questions … ??
## 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++
UPC/CAF and MPI Compared

Model Explicit vs. Model Implicit

Remote Data Exchanges

Shared Memory Fork and Join: OpenMP

PGAS, Distributed Shared Memory: UPC, CAF

Message or Data Passing: MPI, PVM

Work Sharing Within Loops

Model Explicit vs. Model Implicit
• **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-sub/))`
      - Part of standard, but still deferred on the Cray