Parallel Programming with Unified Parallel C (UPC)

Richard Walsh

CUNY, High Performance Computing Center

New York, NY
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: Declaring Shared Data**

- **Minimal** changes to standard C declarations; array syntax retained
- Adds “shared” keyword and blocking/layout qualifier, “[BSIZE]”
- Cannot be automatic, must be either global (before main) or declared static

Private:  

```c
long count1;
double mass[ATOMS];
```

Shared:  

```c
static shared long scount1;
static shared [1] double smass[ATOMS];
```

- 3 data layout idioms, expressed explicitly at declaration, implicitly thereafter
  - **Indefinite (unblocked):** `shared [0] float indx[ATOMS][THREADS];`
    - All elements have affinity to thread 0, using just “[]” is identical
    - For indefinite blocking, THREADS must be statically defined at compile time
  - **Round-robin:** `shared [1] float indx[ATOMS][THREADS];`
    - Blocks [1] row-element per thread, rows cross threads (the “[1]” is optional)
    - Cray needs THREADS-multiple, right-most dimension
  - **Blocked:** `shared [ATOMS] float indx[THREADS][ATOMS];`
    - Blocks [ATOMS] row-elements per thread => a single, whole row here
    - Blocksize is the number of array-sequential elements per thread
    - Blocked with “[*]” and THREADS-multiple dimension => one or more whole rows per thread
      - Threads-multiple array size: blocksize = (size of array in elements) / (# threads)
      - If not threads-multiple array size: blocksize = floor (size of array in elements / # threads) + 1
• The keyword “shared” places one copy of the object in shared memory

- Shared **scalars** (scount1) are singular with an affinity for thread 0
  - CAF creates a copy on each image
- Shared **arrays** (smass[ATOMS]) are distributed across threads
  - First element is usually on thread 0 (not for some dynamic memory allocations)
  - Three blocking schemes: indefinite (infinite), round-robin, blocked
  - CAF creates a copy on each image
- The block size, thread count, and an array’s size and rank determine distribution pattern.
UPC: Unblocked Data Layout

- **Indefinite (unblocked):** shared [] float indx[ATOMS][THREADS];
  - Rows and columns of indx are **not** distributed, all within thread 0

```
Thread 0
  Private
  Shared
  indx[0][0]
  indx[0][1]
  ..
  indx[A-1][T-1]

Thread 1
  Private
  Shared

Thread THREADS-1
  Private
  Shared
```

- Think of the blocking factor as infinite
- Although stored on thread 0, the array can be referenced by any thread
- If THREADS is used in declaration, it must be defined statically (at compile time)
- Memory on other threads need not be allocated
  - Equal extents are **not** required by the UPC definition
**UPC: Round-Robin Data Layout**

- **Round-robin (unit blocking):** `shared float indx[ATOMS][THREADS];`
  - Rows cross threads, columns within threads
    - Whole column locality if last dimension is THREADS
  - Example: Blocksize = 1, ATOMS = 2, THREADS = 8

  - This is also called **cyclic** or **unit blocking**, can use “**shared [1]**”
  - If THREADS is defined dynamically …
    - Array size must be a thread multiple, [THREADS*4] is OK  [THREADS+4] is not
    - THREADS may appear in dimensions only once
**UPC: Blocked Data Layout**

- **Blocked:** shared [ATOMS] float indx[THREADS][ATOMS];
  - Rows within thread, columns cross threads
    - Whole row locality if left dimension is THREADS, right blocksize
  - Blocksize (ATOMS here) must be a constant and one dimensional
    - The block is the unit of distribution, one per thread
  - Example: Blocksize = ATOMS = 4, THREADS = 8:

- Wild card blocking “[*]” produces blocks of ARRAY_SIZE/THREADS
  - To put one row per thread: shared [*] float indx[THREADS][ATOMS];
  - To put N-sequential rows per thread: shared [*] float indx[N*THREADS][ATOMS];
  - As with round-robin, THREADS can be defined dynamically
    - Same restrictions apply and you want asize%THREADS = 0
**UPC: Blocked, Non-congruent**

- **Non-congruent blocking**: shared [3] float indx[ATOMS][THREADS];
  - Messy, something to be avoided
  - Example: Blocksize = 3, ATOMS = 4, THREADS = 4

  - Shared memory space has unequal extents
  - Blocking cuts up array’s natural dimensions
  - May not be allowed on the Cray XE6 (have not tried it)
  - CAF specification does not allow anything like this
• Qualifier “shared” cannot appear within a structure
  – Except to qualify a private pointer to shared
    • struct OK { shared char *p1}; ! P1 is private, but points to shared
    • struct BAD {char * shared p2}; ! P2 is located in shared, but struct isn’t
    • shared struct OK {int bignum}; ! Whole structure is shared
  – Avoids mixing shared and private data in structures
• Block size declaration qualifier “[BLOCK]” is part of type
  – Therefore retained in pointer aliasing, assignment
    • shared [] long *p0; ! p0 de-references to parent thread, however assigned
• Block size of “shared [*]” array is:
  – ARRAY_SIZE/THREADS … (ARRAY_SIZE in elements)
• “[*]” cannot not be used in declaring data of pointer type
• Shared variables must be statically or globally defined
  – May not be automatic variables
    void foo(void) {
      shared int v; ! Automatic shared int is not allowed
      static shared int x; ! Explicitly declared static so OK
      shared int *y; ! Private pointer-to-shared is allowed
      int * shared z; ! Automatic shared pointer to int is not allowed
    ...
    }
• Block size cannot exceed UPC_MAX_BLOCK_SIZE
• The i-th element’s thread affinity is found with:
  – floor(( i / block size )) mod THREADS, or
  – upc_threadof(element address);
• With dynamic thread allocation, the compiler must be able to compute the number of elements per thread without knowing the thread count
  – May include THREADS only once in one dimension
  – Allow only integer, multiple expressions: shared int a[ATOMS][THREADS*4]
  – May not use THREADS for indefinite blocked arrays
  – Fixed size arrays are not allowed: shared int x[15];
• Arrays of indefinite block size, “shared [] array”, may not include THREADS in dimension when dynamically allocated
• One block size, one access type (strict or relaxed) per typedef
• Blocking factor applies to the underlying primitive type in a typedef
shared [*] double d[4*THREADS][8];  (run on 4 threads)
  • What is the blocking factor?
  • How many rows per thread?

shared [5] long list[4][8];  (run on 3 threads)
  • How many rows per thread?
  • How many elements are on each thread?
  • Where is 20th element?

shared int m[32+THREADS];
  • When will this generate a compiler error?

shared float x; float y;  (run on 32 threads)
  • How many copies of x? Of y?

typedef int P[8];
shared [8] P P2[THREADS];
  • How is this blocked?
  • Eight Ps (64 ints), or 8 ints per thread?

shared int *table[THREADS];  (run on 4 threads)
  • How many ints are allocated? Where are they located?
  • How many pointers total? Where are they located?
  • Where must the pointers point? (more on pointers later)
  • When will this generate a compiler error? Why?
shared [*] sarray[N*THREADS];
sarray[i] = MYTHREAD ! Run on 4 threads in a for(i=0; i<N*THREADS; i++) loop
   • Redundantly and unpredictably initializes sarray 4 times
shared int a=0;
a += sarray[i] ! Run only on thread 0 in a for(i=0; i<N*THREADS; i++) loop
   • Computes the sum (reduction) of all the elements of sarray
   • Result accumulates on thread 0 in shared scalar a, but accessible by all threads
   • What would be the result of all threads ran the loop?

shared int x[256][THREADS]; int *p;
p = (int *)&x[0][MYTHREAD]; ! Set on all threads
   • Casts the shared address of the thread-local piece of a shared array to a local-private pointer
   • Speeds pointer arithmetical references to the local part of shared array

typedef struct ptable {
   int pnum;
   shared float *pmass;
} Atom;
Atom atoms; float max = 0.0; shared float mass[THREADS]
atoms.pmass = &mass[i];
if ( *(atoms.pmass) > max ) max = *(atoms.pmass) ! Run in loop on all threads
   • Uses a private structure with a private pointer-to-shared to find largest shared value
   • Each thread finds the same global result and stores it locally in its own max
   • Would locks be needed?
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
Process synchronization constructs in UPC

• Important concepts:
  – Implicit vs. Explicit
  – Global vs. Partial vs. Local
  – Single-phase vs. Split-phase (synchronization phase)
  – Thread (instructions) synchs vs. Memory (data) synchs
  – Critical regions (one-thread-at-a-time regions)

• **Implicit** synchronization, off by default (relaxed mode)
  – You are responsible for it, with some exceptions:
  – UPC collective functions:
    • upc_all_alloc(), upc_all_lock_alloc(), upc_all_broadcast(), etc.
  – UPC special extension statements:
    • upc_forall(;;;)

16 Feb 2011
Process synchronization constructs in UPC (cont.)

- **Explicit** synchronization tools, **barriers**
  - **Single-phase**, blocking barrier statement, **all-threads**
    - `upc_barrier [tag];`
    - All threads *post and wait* at barrier until all have posted
    - The tag is used to distinguish different barriers
    - Includes global memory sync
  - **Split-phase**, non-blocking barrier statement, **all-threads**
    - `upc_notify [tag]; ... /* free to compute */ ... ; upc_wait [tag];`
    - All threads first *post* first `upc_notify;`
    - Posting threads may compute until *wait* at `upc_wait`
    - When all have posted (not reached wait), all are released
    - The tag is used to distinguish different barriers
    - Includes global memory sync
Process synchronization constructs in UPC (cont.)

- **Explicit** synchronization tools, *locks*
  - Locked/critical (one thread) region definition with function calls
    - **Declare** lock variable with type: `upc_lock_t *a_lock;`
      - Shared data type, but opaque to the programmer
    - **Allocate and free** memory for locks:
      - Allocated in the unlocked state
        - Collectively on all threads: `upc_lock_t * upc_all_lock_alloc(void);`
        - Locally on calling thread: `upc_lock_t * upc_global_lock_alloc(void);`
        - Free from any thread: `void upc_lock_free(upc_lock_t *);`
          » Frees any lock in any state
    - **Set and unset** locks:
      - Set and proceed if not locked: `void upc_lock(upc_lock_t *ptr);`
        » Else wait
      - Set, return 1 if not locked: `int upc_attempt_lock(upc_lock_t *ptr);`
        » Else return 0
      - Unset, proceed (locking thread only): `void upc_unlock(upc_lock_t *ptr);`
What is a “Memory Consistency Model”??:

The practices governing the availability of the most current global (shared) data values to all processes in the correct inter- and/or intra-program order. “Correct order” is usually defined as the intended, sequential program order.

• General concerns
  – Not yet stored, still processor-local results
    • Stored in registers, cache, or in-flight
    • Needed by another processor
  – Out-of-program-order statement execution
    • Driven by compiler optimization techniques
    • Compiler sees only processor-local data dependencies, but …
    • Programmer must control for inter-processor dependencies
  – How and when to use different modes (UPC’s relaxed vs. strict)
    • Getting it right
    • Minimizing negative performance effects
UPC memory consistency/coherency: two modes

- Default is **relaxed mode**, `#include <upc_relaxed.h>`
  - Programmer ensures consistency when required
  - Consistency is applied (one hopes) only when necessary
  - Compiler is free to optimize, reorder for **fast code**

- Also, **strict mode**, `#include <upc_strict.h>`
  - All shared data references are sync’d, globally observable
  - Does in software what is done on hardware in cc-NUMA memory
  - Strict “program order” is observed, safe, debugging tool, **slow code**
  - **Warning:** your intended “sequential program order” must be clear
UPC memory consistency/coherency: two modes

- Controlled with *includes*, *directives*, or *type* qualifiers
  - 3 distinct scopes here
  - Global scope: 
    
  
  - Block scope: 
    
  - Variable scope: 
  
- Controlled through *statements* at specific locations
  - Thread-local, shared-data sync: upc_fence;
  - Equivalent to a “null” strict shared memory reference
    - strict static shared in x = 1; x = x;
    - forces all prior relaxed shared references on thread to complete
    - they become globally “observable”
  - Program-global, instruction and shared-data sync: upc_barrier;
  - Also dynamic memory and other collective routines
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. Support Functions and Operators
8. **Work Sharing in Loops**
9. Using Pointers
10. Dynamic Memory Allocation
11. IO Model and Constructs
12. Exercises
In the SPMD model, all processors execute the same code, but …

– The same code works largely on different data
– The path through code is conditionally modified using process ID
– The code’s looping structures are exploited to distribute work

UPC needs a mechanism for distributing the work in loops

– UPC extends the C for (;;) statement with a new *affinity field*
  • `upc_forall(;; &a[i])` (iteration i is given to thread local to &a[i])
  • In UPC loops, iterations/array indices invisibly trip across PGAS
  • UPC array syntax *does not* explicitly express element location, so …
  • Index and affinity used to divide iterations *implicitly* among processors
UPC work sharing in more detail:

upc_forall(expr;expr;expr;affinity) {} 

- upc_forall(;;;) is a UPC-specific **collective/synchronous** construct 
- Iterations are distributed to threads based on shared data affinity 
- The affinity field can take one of **three** forms 
  - An *integer expression*, often a function of index ==> i or i/4 
    - Thread iterations assigned using: i\%THREADS == MYTHREAD 
    - Good for round-robin thread assignment 
  - A *shared memory address*, often indexed as ==> &a[i][0] 
    - Thread iterations assigned using: upc_threadof(&a[i][0]) == MYTHREAD 
    - Good for round-robin or blocked thread assignment 
  - The *reserved word* “continue” or nothing specified 
    - All threads execute all iterations (revert to simple for() loop pattern) 
    - Used when nesting upc_forall() loops
UPC work sharing in more detail (cont.):

- Outer most `upc_forall(;;;)` loop with affinity is the:
  - Controlling (work distributing) loop
  - Only loop in nest where work is distributed
  - All other loops behave as if their affinity was “continue”

- Loop dependencies not allowed (that span loop threads)
  - Compiler generates scalar code (not vector code), but …
  - Does not manage sync’s between threads, so in …
  - `a[i+1] = a[i] + b[i]`
  - `a[i+1]` may be read on next thread before write on this one

- Individual threads must not exit loop block early
  - No breaks, gotos, etc.
UPC: Work Distribution/Sharing

Simple UPC example, matrix vector multiply:

```c
#include <upc_relaxed.h>
#define BLOCK 100
#define ASIZE BLOCK*THREADS
shared [ASIZE] double a[ASIZE][ASIZE];
shared double b[ASIZE], x[ASIZE];
void main()
{
    int i, j;

    upc_farray(i=0; i<ASIZE; i++;)
    upc_farray(j=0; j<ASIZE; j++; continue) {
        b[i] += a[i][j] * x[j];
    }
}
```

Array declared as shared. Blocking is explicit in [ASIZE].

Affinity by integer, so loop iterations are divided and assigned to threads using:
```
i%THREADS == MYTHREAD
```

Affinity by “continue”, so all threads execute all iterations in loop j.

UPC/C is row major therefore “j” is placed in inner loop to get memory stride 1.

How could the same thing be achieved using an address in the affinity field?
UPC Tutorial

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. Support Functions and Operators
8. Work Sharing in Loops
9. Using Pointers
10. Dynamic Memory Allocation
11. IO Model and Constructs
12. Exercises
UPC: Using Pointers

UPC/CAF shared data spaces add complexity to pointers

– Pointers to shared objects must identify target process
– Pointers might point to …
  • shared or private memory
– Pointers might reside in …
  • shared or private memory
  • local or non-local memory
– Implies four pointer types (pp, sp, ps, ss)
– First letter defines pointer type, second pointer location

![Pointer Types Diagram]
UPC allows pointers of all four classes (PP, PS, SP, SS)

- Private object => thread-local (private or shared) data (PP): `int *pp1;`
  - Standard C pointer with simple address
  - Occupies a private addressable location on each UPC thread
  - Pointer arithmetic follows local memory order, always on MYTHREAD
    - Local memory order in C is typically row-wise, but …
    - Could be column-wise, if the PP is assigned to local piece of shared object
    - `pp = (int *)&sarray[0][MYTHREAD]`
  - Cannot be cast to shared pointer (why?), but …
  - Can reference shared memory on local thread (as if it were private) How?
    - Speeds references to local shared objects, such references always relaxed
UPC: Using Pointers

UPC pointer types, continued (PP, SP, PS, SS)

- Private object => thread-shared (local or not) data (SP): `shared int* sp1`;
  - Complex 3-tuple-pointer with **thread**, **phase**, and **address** values
  - Occupies a **private** addressable location on each UPC thread
  - Pointer arithmetic follows subscript (row), even across threads
  - Has thread locations (and possible values) managed by parent thread
  - Can be cast to a private pointer to local data (private or shared)
    - Object cast **must** have affinity to casting thread (MYTHREAD)
      » Where would this be allowed in diagram below?
  - Speeds references to local shared objects, but …
  - Thread and phase information is lost

```
<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
<th>Process 3</th>
<th>Process N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Private Memory</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
</tr>
<tr>
<td>Shared Memory</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

16 Feb 2011
**UPC: Using Pointers**

**UPC pointers types continued** (PP, SP, PS, SS)

- Shared object => thread-local (private or shared) data (PS): `int *shared ps1;`
  - Standard C pointer with simple address
  - Occupies a *shared* addressable location on UPC thread 0**
  - Pointer arithmetic follows local memory order, on the referring thread
    - Local memory order in C is typically row-wise, but …
    - Could be column-wise if PS is assigned to point to local piece of shared object
    - `ps = (int *)&sarray[0][MYTHREAD]`
  - Cannot be cast to a shared pointer (why?), but …
  - Can reference shared memory on local thread (as if it were private). How?
    - Speeds references to local shared objects
  - **Warning:** Any thread can change it, always points to referring thread’s address space, no guarantee target is symmetrically placed across threads
UPC pointers types continued (PP, SP, PS, SS)

- Shared object => thread-shared (local or note) data (SS): `shared int *shared ss1;`
  - Complex 3-tuple-pointer with `thread`, `phase`, and `address`
  - Occupies a `shared` addressable location on UPC thread 0**
  - Pointer arithmetic follows subscript (row), even across threads
  - Can be cast to private pointer to local data (private or shared)
    - Object cast `must` have affinity to casting thread (MYTHREAD)
      - Where would this be allowed in diagram below?
    - Speeds references to local shared objects
    - Thread and phase information is lost

- Warning: Any thread can change it

```
<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
<th>Process 3</th>
<th>Process N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Private Memory</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shared Memory</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**ss**
UPC pointer data fields

- **Private pointer** contains only local virtual address
  - virtual address
  - Pointer arithmetic traditional, **simple**, **fast**
    - Formula for \( p_2 = (p_1 + i) \):
      - \( \text{vaddress}(p_2) = \text{vaddress}(p_1) + i \times \text{sizeof}(p_1) \)

- **Shared pointer**, a 3-tuple, contains:
  - Thread number
  - Thread-local virtual address of block
  - Data element offset within block (phase)

<table>
<thead>
<tr>
<th>phase</th>
<th>thread</th>
<th>virtual address of block</th>
</tr>
</thead>
</table>

  - Pointer arithmetic different, **compound**, **slow**
    - Formulas for \( p_2 = (p_1 + i) \):
      - \( \text{phase}(p_2) = (\text{phase}(p_1) + i) \mod \text{blocksize} \)
      - \( \text{thread}(p_2) = (\text{thread}(p_1) + (\text{phase}(p_1) + i) \div \text{blocksize}) \mod \#\text{threads} \)
      - \( \text{blockaddress}(p_2) = f(p_1) = ?? \)

  - Optimized code minimizes shared pointer references, arithmetic
UPC: Using Pointers

UPC pointer casting and arithmetic
- Casting shared to private OK, but only when affinity is local
- Casting shared to shared OK, but …
  - Warning: block size does not transfer to assigned pointer
  - Warning: if block or type sizes differ, phase is set to zero (meaning?)
- Shared pointers increment by following subscript (row)
  - Regardless of thread crossing pattern
  - Warning: assigning or casting to non-congruent objects is complex
- Examples (use thread number of 4, running on thread 0):

```c
shared int y[THREADS*3];
shared [THREADS] int z[THREADS*4];

shared int *sp_y=&y[5];
shared [THREADS] *sp_z=&z[9];

int *pp;
sp_y = sp_y+5; sp_z = sp_z-9;

pp = (int *)sp_z;
sp_z = sp_y; // sp_z still has blocksize of 4
             // so pointer arithmetic through
             // y[] will not be what you expect
             // if sp_z is used
```
UPC pointer related functions and operators

• Functions for determining the location/content of shared objects:
  – Return the parent thread of the shared object pointed to
    • `size_t upc_threadof(shared void *ptr);`
  – Return the phase of the shared object pointed to
    • `size_t upc_phaseof(shared void *ptr);`
  – Return the local address of the shared object pointed to
    • Like a shared-to-private cast
    • `size_t upc_addrfield(shared void *ptr);`
  – Reset the phase of a shard pointer to zero
    • Like a non-congruent cast
    • `shared void *upc_resetphase(shared void *ptr);`
UPC pointer related functions and operators (cont.)

- Operators for determining the size of shared objects:
  - Find the thread-local size in bytes of a shared object or type
    - `size_t upc.localsizeof([typename|expression]);`
    - Returns same size on all threads, so function may pad asymmetric layouts
    - The C `sizeof()` may not return constant size, depends on blocking
  - Find the block size in numbers of elements of a shared object
    - `size_t upc.blocksizeof([typename|expression]);`
  - Find the element size in bytes of the first type not an array
    - `size_t upcelemsizeof([typename|expression]);`
  - Find the size of the entire thread-local portion of a shared object
    - `size_t upc.affinitysize(size_t totalsize, size_t nbytes, size_t threadid);`
    - Totalsize is the object size in bytes, nbytes is the byte size of a block
    - Similar to `upc.localsizeof` above
UPC pointer related functions and operators (cont.)

- Functions for *moving* shared data objects
  - Objects moved in simple memory-order (byte-wise, unit stride)
    - Forced by indefinite shared cast, `shared [] char *dst|*src`
  - Objects locality is governed by affinity of dst and src
  - Not collectives, not synchronization points, so are relaxed copies

Functions:

- **Copy** n bytes *(shared-to-shared)* from one shared object to another
  - No thread affinity requirements; destination is first argument
  - `void upc_memcpy(shared [] char *dst, shared [] char *src, size_t n);`

- **Get** n bytes *(shared-to-private)* from a shared object, place in a private object …
  - On the calling thread; destination is first argument
  - `void upc_memget(char *dst, shared [] char *src, size_t n);`

- **Put** n bytes *(private-to-shared)* into shared object from a private object …
  - On the calling thread; destination is first argument
  - `void upc_memput(shared [] char *dst, char *src, size_t n);`

- **Set** n bytes in a shared object to c, cast as an unsigned char
  - `void upc_memset(shared [] char *dst, int c, size_t n);`
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. Support Functions and Operators
8. Work Sharing in Loops
9. Using Pointers
10. **Dynamic Memory Allocation**
11. IO Model and Constructs
12. Exercises
UPC dynamic, shared-memory allocation functions:

- **Global** functions that allocate shared memory on all threads

  ```c
  shared void *upc_global_alloc (size_t nblocks, size_t nbytes)
  ```

  - **Not a collective function**, all/any thread may call independently
  - Returns a generic pointer-to-shared, on the calling thread
    - Can then be cast and assigned to either an **SP** or **SS**
    - Why not to a **PS** or **PP**? What is affinity of starting address?
  - Points to start (on thread 0) of shared array spanning all threads
  - Multiple calls generate multiple, distinct shared arrays
  - Default generic layout:
    ```c
    shared [nbytes/sizeof(type)] type[nblocks*nbytes/sizeof(type)];
    ```
  - Generic blocksize not actual, that is set by cast at assignment
Global functions that allocate shared memory across all threads

```
shared void *upc_all_alloc (size_t nbblocks, size_t nbytes)
```

- A **collective function**, synchronously called by all threads
- Returns a generic pointers-to-shared, on each thread
  - Can then be cast and assigned to either an **SP** or **SS**
  - Why not to a **PS** or **PP**? What is affinity of starting address?
- Each points to the start (on thread 0) of same array, spanning all threads
- Multiple calls generate multiple, distinct arrays
- Default generic layout:

  ```
  shared [nbytes/sizeof(type)] type[nblocks*nbytes/sizeof(type)]
  ```

- Generic blocksize not actual, that is set by cast at assignment
Global function data layout (4 threads)

- \( \text{ps1} = (\text{shared [4] int * shared}) \ \text{upc\_global\_alloc(4*THREADS, 5*sizeof(int))} \);
  - Run on thread 1
  - \( H \ \text{“height”} = \frac{\text{total\_elements}}{\text{threads}\times\text{blocksize}} = \text{blocks} \) per thread
  - \( 4\times4\times5 / 4\times4 = 5 \)
  - Phase of pointer is zero, affinity is thread 0

- \( \text{ps2} = (\text{shared [4] float *}) \ \text{upc\_all\_alloc(4*THREADS, 2*sizeof(float))} \);
  - Run collectively on all threads
  - \( H \ \text{“height”} = \frac{\text{total\_elements}}{\text{threads}\times\text{blocksize}} = \text{blocks} \) per thread
  - \( 4\times4\times2 / 4\times4 = 2 \)
  - Phase of pointer is zero, affinity is thread 0

![Diagram showing memory allocation](image)
UPC dynamic, shared-memory allocation functions (cont.)

- **Local** functions that allocate shared memory on calling thread only

  shared void *upc_alloc (size_t nbytes);

  - *Not a collective function*, all/any thread may call independently
  - Returns a generic pointer-to-shared, on the calling thread
    » Can be cast to either SP or SS, … or PS or PP. Why?
  - Points to start of a shared array, local/limited to the calling thread
  - Multiple calls generate multiple, distinct thread-local arrays
  - Default layout:
    
    shared [nbytes/sizeof(type)] type[nblocks*nbytes/sizeof(type)];
  - Can be used to allocate shared memory with an affinity to any thread
    » Unlike: shared [] int[THREADS]; allows allocation only on thread 0

  shared void *upc_local_alloc (size_t nbblocks, size_t nbytes);

  - Deprecated function with same placement as upc_alloc() above
  - I like this name better …
UPC dynamic, shared-memory allocation functions (cont.)

- Function to free dynamically allocated shared memory

```c
void upc_free (shared void* ptr);
```

- *Not a collective* function, any thread can call it
- Frees memory allocated with *any* shared allocation function
  
  » Even `upc_all_alloc()` data freed only once by any thread
Local function data layout (4 threads)

- \( \text{ps3} = (\text{shared int } *) \text{ upc_alloc} (\text{ASIZE}\times \text{sizeof(int)}) \);
  - Called on thread 1
  - Block size other than 1 has no meaning? Why?
- \( \text{ps4} = (\text{shared float } * \text{ shared}) \text{ upc_alloc} (2\times \text{ASIZE}\times \text{sizeof(float)}) \);
  - Called on thread 3
  - Block size other than 1 has no meaning? Why?
- \( \text{ps5} = (\text{shared float } *) \text{ upc_local_alloc} (2\times \text{ASIZE}, \text{sizeof(float)}) \);
  - Called on thread 0 ! This routine is deprecated
  - Note that it allocates same amount of local shared memory as \text{ps4} above
Some UPC dynamic memory allocation examples:

/* shared variable declarations */

/* Allocate 25 integers elements per thread, with each thread doing its portion of the allocation. --COLLECTIVE CALL */
p1 = (shared [5] int *) upc_all_alloc( 5*THREADS, 5*sizeof(int) );

/* Allocate same 25 integer elements per thread, but just run the allocation on thread 5. -- NON COLLECTIVE CALL */
if (MYTHREAD == 5)
p2 = (shared [5] int *) upc_global_alloc(5*THREADS, 5*sizeof(int) );

/* Allocate 5 integer elements only on thread 3. -- NON COLLECTIVE CALL */
if (MYTHREAD == 3)
p3 = (shared [5] int *) upc_alloc( 5*sizeof(int) );

/* Allocate 25 integer elements per thread, just run the allocation on thread 4, but make the space accessible everywhere. --NON COLLECTIVE CALL */
if (MYTHREAD == 4)
p4 = (shared [5] int * shared) upc_global_alloc(5*THREADS, 5*sizeof(int) );

/* Allocate 5 elements only on thread 2, but have the result visible on all threads. -- NON COLLECTIVE CALL */
if (MYTHREAD == 2)
p5 = (shared [5] int * shared ) upc_alloc( 5*sizeof(int) );
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. Support Functions and Operators
8. Work Sharing in Loops
9. Using Pointers
10. Dynamic Memory Allocation
11. IO Model and Constructs
12. Exercises
13. (NOT COVERED: UPC Collectives)
UPC IO model, as defined in the extension

- Not sure where its fully implemented
  - Cray’s XE6 UPC does implement UPC IO
- UPC supports both POSIX and a UPC IO library (MPI-IO like)
  - POSIX is singular (one thread calls)
  - UPC IO is collective (all threads must call)
  - Cannot mix use on same file
- UPC IO provides several classes of routines
  - Read/write to shared or private memory
  - Read/write synchronously or asynchronously
  - Read/write list vectors and offsets or not
  - Easily identified by name: `upc_all_fread_shared()`
- UPC IO uses relaxed/weak file consistency by default
  - Overlapping reads/writes produce inconsistent results, unless …
  - Programmer takes the responsibility to synchronize
  - Consistency can be set (weak/strong), “flags” argument
  - Within a thread, consistency is strict/strong
UPC IO model (cont.)

– File descriptor type “fd” is a pointer to type upc_file_t *
  • Includes needed:
    #include <upc.h>
    #include <upc_io.h>
  • This is a shared data type
  • Can be passed between threads and still compare equal

– Signed integer type for file offsets upc_off_t
  • Used with upc_all_fseek()
  • Needs same includes

– Supports global/common and thread/individual file pointers
  • Controlled by argument, “flags”
  • Writes with common file pointers require sync’ing before next read
    – Under the default weak consistency

– Fine tuning done using hints and flags keywords

– Return values (and errno)
  • May differ across threads
  • Only required to report on one thread
UPC IO collective routines (a sample of important ones)

- **Open a file by name:**
  ```c
  upc_file_t *upc_all_fopen(const char *fname, int flags,
                           size_t numhints, struct upc_hint const *hints);
  ```
  - `fname` is filename, "myfile"
  - `flags` controls access modes, pointer type, consistency, more.
    - UPC_RDONLY, UPC_WRONLY, UPC_RDWR
    - UPC_COMMON_FP, UPC_INDIVIDUAL_FP
    - Bitwise inclusively OR'd
  - `numhints` and `hints` are more detailed controls in a keyword pair list
  - Returns a file pointer or NULL on error, sets `errno`

- **Close a file:**
  ```c
  int upc_all_fclose(upc_file_t *fd);
  ```
  - Executes an implicit `upc_all_fsync()`
  - Returns 0 on success or -1 on error, sets `errno`
UPC IO collective routines (cont)

- **Flush buffers** of writes:
  
  ```c
  int upc_all_fsync(upc_file_t *fd);
  ```
  
  - Includes an implied barrier
  - Returns 0 on success or -1 on error, set errno

- **Set position** of current file pointer:
  
  ```c
  upc_off_t upc_all_fseek(upc_file_t *fd, upc_off_t offset, int origin);
  ```
  
  - Seeks relative to int origin
    - UPC_SEEK_SET, UPC_SEEK_CUR, UPC_SEEK_END
  - Offset can be positive or negative
  - For common file pointers offset and origin must be identical
  - Returns current location in bytes, or -1 on error, sets errno

- **Set attributes** of file:
  
  ```c
  int upc_all_fcntl(upc_file_t *fd, int cmd, void *arg);
  ```
  
  - Set/Reset file attributes (pointer type, consistency, etc.)
UPC IO collective routines (cont.)

- Read into a local buffer from a file:
  
  ```c
  upc_off_t upc_all_fread_local(upc_file_t *fd, void *buffer,
  size_t size, size_t nmemb, upc_flag_t flags);
  ```
  
  - `fd` must be defined with individual file pointer
  - `buffer` may be different (in name and/or off-set) on each thread
  - Read `size*nmemb` bytes (possibly different on each thread) into `buffer`
  - Returns number of bytes read on thread or -1 on error, sets errno

- Read into a shared buffer from a file:
  
  ```c
  upc_off_t upc_all_fread_shared(upc_file_t *fd, shared void *buffer,
  size_t blocksize, size_t size, size_t nmemb, upc_flag_t flags);
  ```
  
  - `fd` may be defined with an individual or common file pointer
  - `buffer` must a pointer to shared data, with any affinity
  - `blocksize` is the declared blocksize of buffer in elements
  - If individual pointer, then like above except to shared memory
  - If common pointer, `buffer`, `blocksize`, `size`, `nmemb` same on each thread
    - Each thread fills its blocks of shared memory
  - Returns total number of bytes read or -1 on error, sets errno
UPC IO collective routines (cont.)

- Write routines are similar:
  upc_off_t upc_all_fwrite_local();
  upc_off_t upc_all_fwrite_shared();

- Asynchronous read/write routines:
  void upc_all_fread_local_async();
  void upc_all_fread_shared_async()
  void upc_all_fwrite_local_async();
  void upc_all_fwrite_shared_async();
  upc_off_t upc_all_fwait_async();
  upc_off_t upc_all_ftest_async();

- List IO file-memory vector-offset routines:
  upc_off_t upc_all_fread_list_local();
  upc_off_t upc_all_fwrite_list_local();
  upc_off_t upc_all_fread_list_shared();
  upc_off_t upc_all_fwrite_list_shared();
UPC IO example (taken from UPC Specs. V1.2)

A synchronous, parallel read from a file using a common file pointer into a blocksize-5 shared buffer. Each of 4 threads reads one block into its shared memory space. On success, tbytes will contain the total number of bytes read by all threads (8*20 = 160 bytes). The common file pointer “fd” will be advanced 160 bytes.

```c
#include <upc.h>
#include <upc_io.h>

shared [5] float buffer[20]; /* Assume a total of 4 static THREADS */
upc_file_t *fd;
upc_off_t tbytes;

fd = upc_all_fopen( "myfile", UPC_RDWRONLY | UPC_COMMON_FP, 0, NULL );

tbytes = upc_all_fread_shared( fd, buffer, upc_blocksizeof(buffer),
                                   upc_elementssizeof(buffer), 20, UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC);
```

Reminder: UPC IO routines are not yet implemented on Cray systems
Data layout after this `upc_all_fread_shared()`

- Each thread does its own piece of the read
- 20 floats are read (160 bytes)
- Buffer’s declared blocking factor is used (5 here)
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. Support Functions and Operators
8. Work Sharing in Loops
9. Using Pointers
10. Dynamic Memory Allocation
11. IO Model and Constructs

12. Exercises

13. (NOT COVERED: UPC Collectives)