AMS 250: An Introduction to High Performance Computing

PGAS

Shawfeng Dong

shaw@ucsc.edu

(831) 502-7743

Applied Mathematics & Statistics

University of California, Santa Cruz

Outline

- PGAS Overview
- Coarray Fortran (CAF)
- Unified Parallel C (UPC)
- UPC++

Recap: Parallel Programming Models

- Process Interaction:
  - Shared Memory
  - Message Passing
  - **Partitioned Global Address Space (PGAS)**
- Problem Decomposition
  - Task Parallelism
  - Data Parallelism

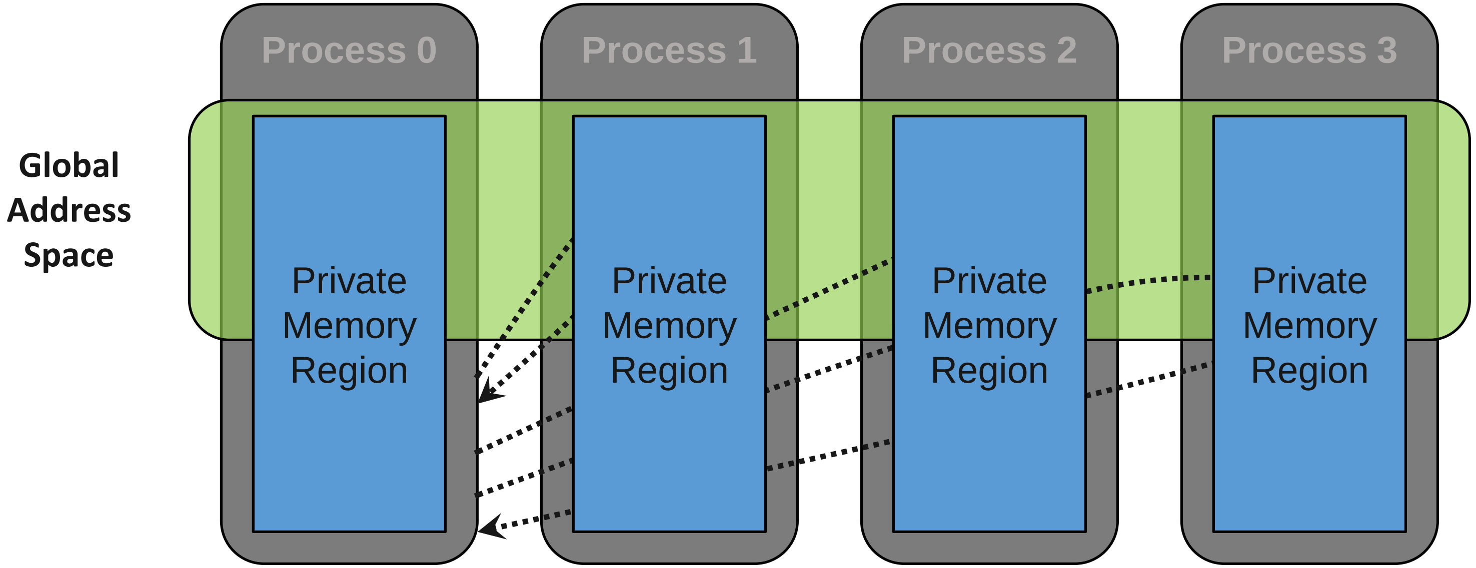
PGAS (Partitioned Global Address Space)

- A global memory address space that is logically partitioned and a portion of it is local to each process or thread
- One-sided communication
- Explicit synchronization, as opposed to (mostly) implicit for MPI
- PGAS libraries:
  - MPI One-Sided (RMA), OpenSHMEM, Global Arrays, UPC++, etc.
- PGAS languages:
  - UPC (Unified Parallel C) – an extension to C
  - CAF (Coarray Fortran) – part of Fortran 2008 standard
- APGAS (asynchronous partitioned global address space) languages, which permit both local and remote asynchronous task creation:
  - Chapel: http://chapel.cray.com/
  - X10: http://x10-lang.org/

Recap: One-Sided Communications

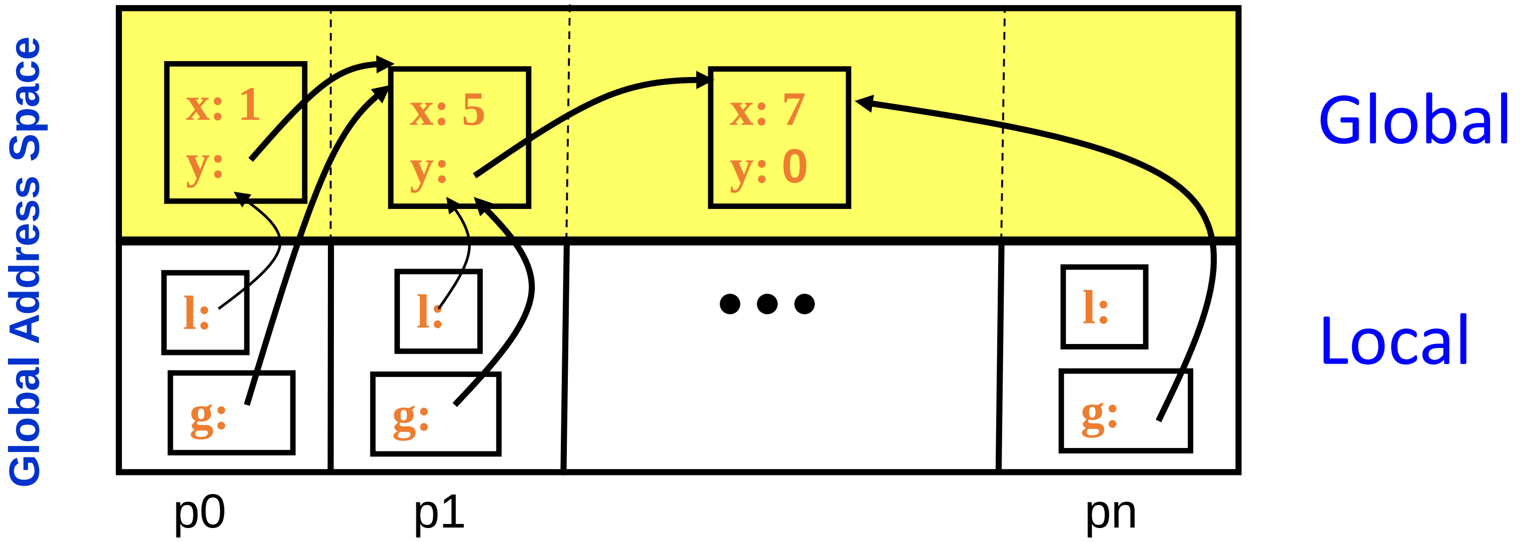
The basic idea of one-sided communication models is to decouple data movement with process synchronization

- Should be able move data without requiring that the remote process synchronize
- Each process exposes a part of its memory to other processes
- Other processes can *directly* read from or write to this memory



Partitioned Global Address Space

- **Global Address Space**: thread/process may directly read/write remote data
  - Convenience of shared memory
- **Partitioned**: data is designated as local or global
  - Locality and scalability of message passing



Fortran 2008

- Fortran 2008 (latest draft) http://www.j3-fortran.org/doc/year/10/10-007.pdf
- Fortran 2008 is a natively parallel language
  - SPMD programming model
  - Simple syntax for one-sided communication, using normal assignment statements
- Executable is replicated across processors (MPI-like)
- Each instance is called an "**IMAGE**"
- Each image has its own data objects
- Each image executes *asynchronously* except when syncs are indicated

“Hello, world!” in Coarray Fortran (CAF)

```
program caf_hello

   character*80 hostname

   call hostnm(hostname)

   write(*,*) "Hello from CAF image ", &
                this_image(), &
                "running on ", trim(hostname), &
                " out of ", num_images()

end program caf_hello
```

Compared to MPI “Hello, world!” in Fortran 90

```
program hello

  use mpi
  implicit none
  integer :: ierr, rank, size

  call MPI_INIT(ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
  call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
  print *, "Hello, world! I am process ", rank, " of ", size
  call MPI_FINALIZE(ierr)

end program hello
```

Coarray Fortran on Hyades (Intel Compilers)

- Intel Fortran compiler supports parallel programming using coarrays as defined in the Fortran 2008 standard
- A CAF program can be built to run on:
  - either a ***shared memory*** system
  - or a ***distributed memory*** cluster
- To compile the example CAF program for a shared memory system:

```
ifort -coarray caf_hello.f90 -o caf_hello.x
```

- By default, when a CAF program is compiled with the Intel compiler, the invocation creates as many images as there are processor cores on the host platform. For example, on the master node:

```
$ ./caf_hello.x
 Hello from CAF image  6 running on hyades.ucsc.edu out of 32
 Hello from CAF image  7 running on hyades.ucsc.edu out of 32
 Hello from CAF image 14 running on hyades.ucsc.edu out of 32
...
```

CAF on Shared Memory System

There are 2 ways to control the number of images on a shared memory system:

- Use the coarray-num-images=*N* compiler option. E.g.:

```
$ ifort -coarray -coarray-num-images=2 caf_hello.f90 -o caf_hello.x
$ ./caf_hello.x
 Hello from image           1 running on hyades.ucsc.edu out of           2
 Hello from image           2 running on hyades.ucsc.edu out of           2
```

- Use the environment variable FOR_COARRAY_NUM_IMAGES. E.g.:

```
$ export FOR_COARRAY_NUM_IMAGES=4
$ ./caf_hello.x
 Hello from CAF image           1 running on hyades.ucsc.edu out of           4
 Hello from CAF image           2 running on hyades.ucsc.edu out of           4
 Hello from CAF image           3 running on hyades.ucsc.edu out of           4
 Hello from CAF image           4 running on hyades.ucsc.edu out of           4
```

CAF on Distributed Memory Cluster

- Set up a machine file, e.g. (*hosts*):

```
gpu-1:2
gpu-2:2
```

- Set up a CAF configuration file, e.g. (*cafconfig.txt*):

```
-genvall -genv I_MPI_FABRICS shm:ofa -machinefile hosts -n 4 ./caf_hello.dist
```

- Compile the example CAF program for a distribute memory cluster:

```
ifort -coarray=distributed -coarray-config-file=cafconfig.txt caf_hello.f90 -o caf_hello.dist
```

- Run the CAF application:

```
$ mpdboot -n 3 -f hosts
$ ./caf_hello.dist
 Hello from CAF image           1 running on gpu-1.local out of           4
 Hello from CAF image           2 running on gpu-1.local out of           4
 Hello from CAF image           3 running on gpu-2.local out of           4
 Hello from CAF image           4 running on gpu-2.local out of           4
$ mpdallexit
```

CAF on Cori

- CAF is supported on Cori through 2 different implementations: Cray CAF and Intel CAF
  http://www.nersc.gov/users/computational-systems/cori/programming/additional-programming-models/
- Cray CAF
  Switch to the Cray compiler environment:
  cori09> module swap PrgEnv-intel PrgEnv-cray

  Supply the '**-h caf**' option when calling ftn:
  cori09> ftn -h caf caf_hello.f90 -o caf_hello.x

  cori09> salloc -N 2 -t 10:00 -p debug -C haswell

  nid00461> ulimit -v unlimited # may not be necessary
  nid00461> srun -n 64 ./caf_hello.x

Coarrays in Fortran 2008

- The array syntax of Fortran is extended with additional trailing subscripts in square brackets ([ ])to provide a concise representation of references to data that is spread across images:
  - e.g., `real :: a(3)[*]`
- Any time a coarray appears without [ ], the reference is to the data on the local image
- The number inside the [ ] can reference any image in the job, including itself
- If a reference with [ ] appears to the right of the =, it is often called a "get"
  - e.g., `b(:) = a(:)[ri]`
- If a reference with [ ] appears to the left of the =, it is often called a "put"
  - e.g., `a(:)[ri] = b(:)`

Fortran 2008 Parallel Programming

- Declaration and allocation

```
real(8), ALLOCATABLE :: rcvbuf(:,:)[*]
! Allocate m*n elements on each processor
ALLOCATE( rcvbuf(m,n)[*] )
```

- Reference

```
! Put data from my local buf into rcvbuf on image k
rcvbuf(:,:)[k] = localbuf(:,:)
```

- PE (processing elements) information

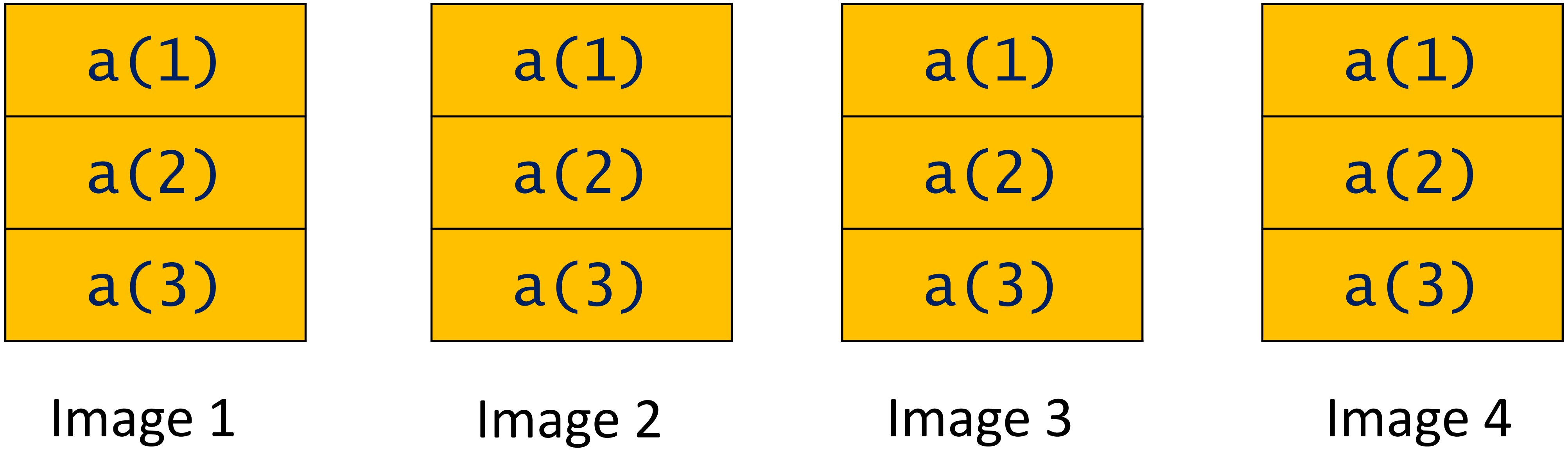
```
this_image(), num_images()
```

- Synchronization

```
sync all
sync images(array_of_images)
```

Array Example

```
real :: a(3)
```



Coarray Example

```
real :: a(3)[*]
```

a(1)[1]	a(1)[2]	a(1)[3]	a(1)[4]
a(2)[1]	a(2)[2]	a(2)[3]	a(2)[4]
a(3)[1]	a(3)[2]	a(3)[3]	a(3)[4]
Image 1	Image 2	Image 3	Image 4

Fortran 2008 Synchronization

Explicit statements:

- sync all
- sync images (array_of_images)
- sync memory
- critical / end critical
- lock / unlock

Implicit synchronization:

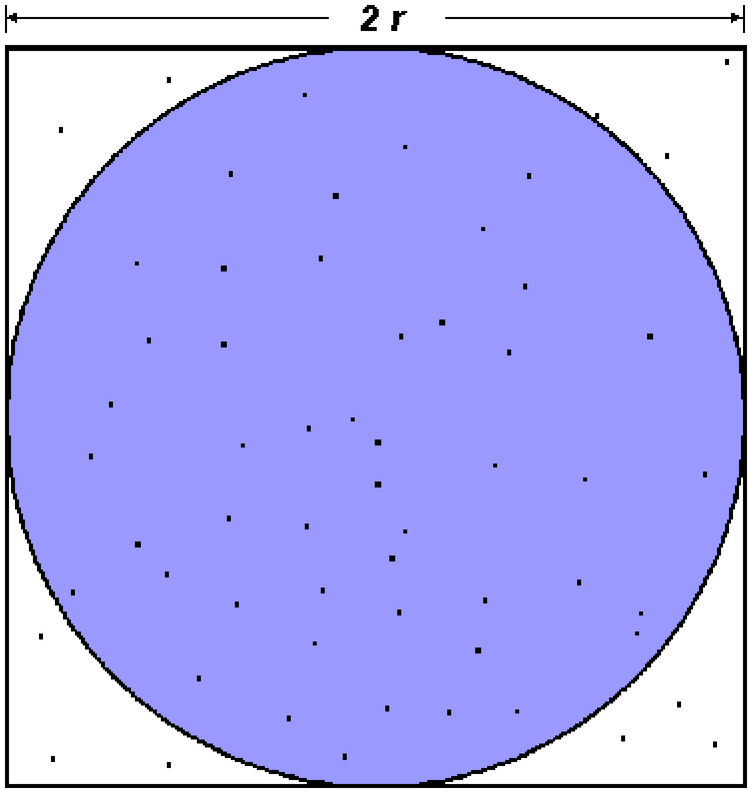
- allocation of a coarray
- deallocation of a coarray (either explicit or implicit)

RYO (roll-your-own) synchronization:

- atomic_ref / atomic_def

Example: Monte Carlo Pi Calculation

- Estimate π using the “dartboard method” ($r=1$)
  - Area of square = $(2r)^2 = 4$
  - Area of circle = $\pi r^2 = \pi$
- Randomly throw darts at (x, y) positions inside the square
- If $x^2 + y^2 < 1$, then dart is inside the unit circle
- Calculate percentage that fall in the unit circle:
  - ratio = # darts inside circle / # darts inside square
- Compute π:
  - $\pi = 4$*ratio



$$A_S = (2r)^2 = 4r^2$$
$$A_C = \pi r^2$$
$$\pi = 4 \times \frac{A_C}{A_S}$$

```fortran
program caf_pi
  implicit none

  integer    :: j
  integer    :: seed(2)
  integer*8  :: N_steps, i_step, hits
  double precision :: x, y
  double precision :: pi_sum, pi
  double precision :: pi_global[*]

  seed(1) = 17*this_image()
  call random_seed(put=seed)
  hits = 0_8
  N_steps = 10000000_8
  do i_step=1_8, N_steps
    call random_number(x)
    call random_number(y)
    if ( (x*x + y*y) <= 1.d0) then
      hits = hits + 1_8
    endif
  enddo

  pi_global = &
       4.d0*dble(hits)/dble(N_steps)

  SYNC ALL

  if (this_image() == 1) then

    pi_sum = 0.d0
    do j=1,num_images()
      pi_sum = pi_sum + pi_global[j]
    enddo

    pi = pi_sum / num_images()
    print *, 'pi = ', pi

  endif

end program caf_pi
```

Unified Parallel C (UPC)

- https://upc-lang.org/
- **UPC** (Unified Parallel C) is an extension to C, with the following constructs:
  - An explicitly parallel execution model (SPMD)
  - A shared address space
  - Synchronization primitives and a memory consistency model
  - Explicit communication primitives, e.g. upc_memput
  - Memory management primitives
- Multiple implementations:
  - Cray UPC
  - gcc version of UPC: http://www.gccupc.org/
  - Berkeley UPC: http://upc.lbl.gov/
- Most widely used on irregular / graph problems today

“Hello, world!” in UPC

```
/* needed for UPC extensions */
#include <upc.h>
#include <stdio.h>

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

- Any legal C program is also a legal UPC program
- If you compile and run it as UPC with *P* threads, it will run *P* copies of the program.

Note: some of the materials are borrowed from Kathy Yelick's presentation and Tarek El-Ghazawi's

- https://people.eecs.berkeley.edu/~demmel/cs267_Spr14/Lectures/lecture08-PGAS-yelick14_4pp.pdf
- http://upc.gwu.edu/downloads/upc_tut04.pdf

Compared to MPI “Hello, world!” in C

```
#include "mpi.h"
#include <stdio.h>

int main(int argc, char* argv[])
{
  int rank, size;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  printf("Hello, world! I am process %d of %d\n", rank, size);
  MPI_Finalize();

  return 0;
}
```

Compared to OpenMP "Hello, world!" in C

```
#include <stdio.h>
#include <omp.h>
int main()
{
  int nthreads, tid;
  #pragma omp parallel private(tid)
  {
    tid = omp_get_thread_num();
    printf("Hello, world! I am thread %d\n", tid);
    #pragma omp barrier
    if (tid == 0)
    {
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d\n",nthreads);
    }
  }
  return 0;
}
```

UPC on Cori

UPC is supported on Cori through 2 different implementations: Berkeley UPC and Cray UPC
http://www.nersc.gov/users/computational-systems/cori/programming/additional-programming-models/

- **Berkeley UPC**
  Berkeley UPC (BUPC) provides a portable UPC programming environment consisting of a **source translation front-end** (which in turn relies on a user-supplied C compiler underneath) and a runtime library based on **GASNet**. The latter is able to take advantage of advanced communications functionality of the Cray Aries interconnect on Cori, such as remote direct memory access (RDMA).
- **Cray UPC**
  UPC is directly supported under Cray's compiler environment through their PGAS runtime library.
  Man page: intro_pgas(7)

Berkeley UPC on Cori

- BUPC is available via the **bupc** module on Cori, which provides
  - The **upcc** compiler wrapper (all 3 programming environments, Intel, GNU & Cray, are supported by BUPC for use as the underlying C compiler
  - The **upcrun** launcher wrapper (which initializes the environment and calls **srun**)
- Compiling and running UPC application with BUPC on Cori:

```
cori08> module load bupc

cori08> upcc upc_hello.c -o upc_hello.x

cori08> salloc -N 2 -t 10:00 -p debug -C haswell

nid00461> upcrun -n 64 ./upc_hello.x
```

Cray UPC on Cori

- To use Cray UPC on Cori, simply switch to the Cray compiler environment and supply the '**-h upc**' option when calling *cc*.
- Compiling and running UPC application with Cray UPC on Cori:

```
cori08> module swap PrgEnv-intel PrgEnv-cray

cori08> cc -h upc upc_hello.c -o upc_hello.x

cori08> salloc -N 2 -t 10:00 -p debug -C haswell

nid00461> ulimit -v unlimited  # may not be necessary
nid00461> srun -n 64 ./upc_hello.x
```

UPC Execution Model

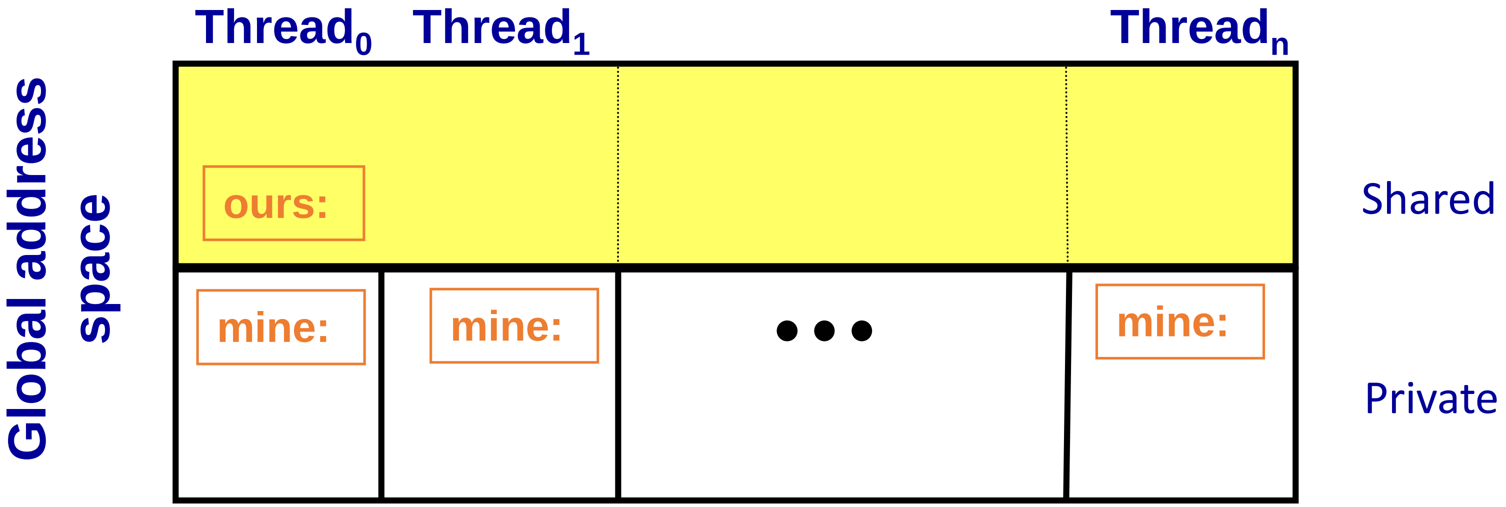
- A number of **threads** working independently in a SPMD fashion
  - UPC threads usually implemented as OS **processes**!
  - Number of threads specified at compile-time or run-time; available as program variable **THREADS**
  - **MYTHREAD** specifies thread index (0 .. **THREADS**-1)
- Synchronization when needed
  - Barriers
  - Locks
  - Memory consistency control

Private vs. Shared Variables in UPC

- Normal C variables and objects are allocated in the private memory space for each thread.
- Shared variables are allocated only once, with thread 0

```
shared int ours;  // use sparingly: performance
int mine;
```

- Shared variables may not have dynamic lifetime: may not occur in a function definition, except as static.



Pi in UPC: Shared Memory Style

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

int hit(){
  int const rand_max = 0xFFFFFF;
  double x = ((double) rand()) /
             RAND_MAX;
  double y = ((double) rand()) /
             RAND_MAX;
  if ((x*x + y*y) <= 1.0) {
    return(1);
  } else {
    return(0);
  }
}
```

```c
shared int hits;

int main(int argc, char **argv) {
  int i, my_trials = 0;
  int trials = atoi(argv[1]);
  my_trials = (trials + THREADS - 1) /
              THREADS;
  srand(MYTHREAD*17);
  for (i=0; i < my_trials; i++)
    hits += hit();

  upc_barrier;
  if (MYTHREAD == 0)
    printf("PI estimated to %f.",
           4.0*hits/trials);
}
```

Race Condition!

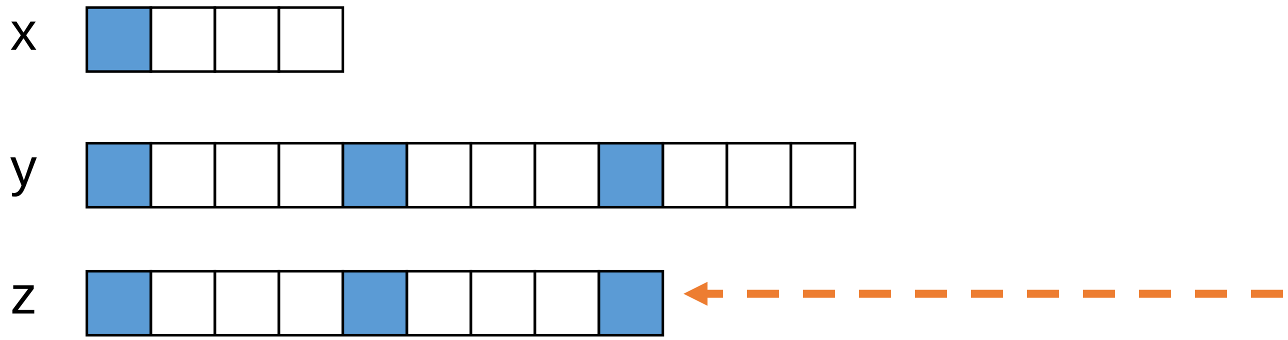
http://people.eecs.berkeley.edu/~demmel/cs267_Spr16/Lectures/lecture08-PGAS-yelick_4pp.pdf

Shared Arrays are Cyclic by default

- Shared scalars always live in thread 0
- Shared arrays are spread over the threads
- Shared array elements are spread across the threads

```
shared int x[THREADS]        /* 1 element per thread */
shared int y[3][THREADS]     /* 3 elements per thread */
shared int z[3][3]           /* 2 or 3 elements per thread */
```

- In the pictures below, assume THREADS = 4
  - Blue elements have affinity to thread 0



z is not logically blocked by columns!

UPC Shared Array Example

```
shared double a[3][THREADS]
```

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]
Thread 0	Thread 1	Thread 2	Thread 3

Pi in UPC: Shared Array Version

```
shared int all_hits[THREADS];

int main(int argc, char **argv) {
  int i, hits, my_trials = 0;
  int trials = atoi(argv[1]);
  my_trials = (trials + THREADS - 1) / THREADS;
  srand(MYTHREAD*17);
  for (i=0; i < my_trials; i++)
    all_hits[MYTHREAD] += hit();

  upc_barrier;
  if (MYTHREAD == 0) {
    for (i=0; i < THREADS; i++)
      hits += all_hits[i];
    printf("PI estimated to %f.", 4.0*hits/trials);
  }
```

UPC Global Synchronization

- UPC has two basic forms of barriers:
  - Barrier: block until all other threads arrive

    upc_barrier
  - Split-phase barriers

    upc_notify; // this thread is ready for barrier

    // do computation unrelated to barrier

    upc_wait; // wait for others to be ready
- Optional labels allow for debugging

```
#define MERGE_BARRIER 12
if (MYTHREAD%2 == 0) {
    …
    upc_barrier MERGE_BARRIER;
} else {
    ...
    upc_barrier MERGE_BARRIER;
}
```

Synchronization - Locks

- Locks in UPC are represented by an opaque type: **upc_lock_t**
- Locks must be allocated before use
  - Allocates 1 lock, pointer to all threads:

  ```
  upc_lock_t *upc_all_lock_alloc(void);
  ```

  - Allocates 1 lock, pointer to one thread:

  ```
  upc_lock_t *upc_global_lock_alloc(void);
  ```

- To use a lock (at start and end of critical region)

  ```
  void upc_lock(upc_lock_t *l);
  void upc_unlock(upc_lock_t *l);
  ```

- Locks can be freed when not in use

  ```
  void upc_lock_free(upc_lock_t *ptr);
  ```

Pi in UPC: Shared Memory Style

```
shared int hits;

int main(int argc, char **argv) {
  int i, my_hits, my_trials = 0;
  upc_lock_t *hit_lock = upc_all_lock_alloc();
  int trials = atoi(argv[1]);
  my_trials = (trials + THREADS - 1) / THREADS;
  srand(MYTHREAD*17);
  for (i=0; i < my_trials; i++) my_hits += hit();

  upc_lock(hit_lock);
  hits += my_hits;
  upc_unlock(hit_lock);

  upc_barrier;
  if (MYTHREAD == 0)
    printf("PI estimated to %f.", 4.0*hits/trials);
}
```

accumulate across threads

UPC (Value-Based) Collectives

- A portable library of collectives on scalar values (not arrays)

  Example: `x = bupc_allv_reduce(double, x, 0, UPC_ADD);`

  `TYPE bupc_allv_reduce(TYPE, TYPE value, int root, upc_op_t op);`

  - 'TYPE' is the type of value being collected
  - root is the thread ID for the root (e.g., the source of a broadcast)
  - 'value’ is both the input and output (must be a “variable” or l-value)
  - op is the operation: UPC_ADD, UPC_MULT, UPC_MIN, …
- Computational Collectives: reductions and scan
- Data Movement Collectives: broadcast, scatter, gather
- Portable implementation available from:
  - http://upc.lbl.gov/download/dist/upcr_preinclude/bupc_collectivev.h
- UPC also has more general collectives over arrays
  - https://upc-lang.org/assets/Uploads/spec/upc-lang-spec-1.3.pdf

Pi in UPC: Data Parallel Style

```
#include <bupc_collectivev.h>
shared int hits;

int main(int argc, char **argv) {
  int i, my_hits, my_trials = 0;
  int trials = atoi(argv[1]);
  my_trials = (trials + THREADS - 1) / THREADS;
  srand(MYTHREAD*17);
  for (i=0; i < my_trials; i++) my_hits += hit();

  //                type, input, thread, op
  bupc_allv_reduce(int, my_hits, 0, UPC_ADD);
  // upc_barrier;

  if (MYTHREAD == 0)
    printf("PI estimated to %f.", 4.0*my_hits/trials);
}
```

UPC++

- https://bitbucket.org/upcxx/upcxx/wiki/Home
- UPC++ paper:
  https://web.eecs.umich.edu/~akamil/papers/ipdps14.pdf
- UPC++ is a PGAS extension for C++
  - "Compiler-free" approach using C++ templates and runtime libraries
- UPC++ provides significantly more expressiveness than UPC:
  - an object-oriented PGAS programming model in the context of C++
  - useful parallel programming idioms unavailable in UPC, such as **asynchronous remote function invocation** and **multidimensional arrays**
  - an easy on-ramp to PGAS programming through interoperability with other existing parallel programming systems (e.g., MPI, OpenMP, CUDA)
- UPC++ performance is close to UPC!

“Hello, world!” in UPC++

```
// upcxx_hello.cpp - Hello world in UPC++
#include <upcxx.h>
#include <iostream>

using namespace upcxx;

int main (int argc, char **argv)
{
  init(&argc, &argv);
  std::cout << "I'm rank " << myrank() << " of " << ranks() << "\n";
  finalize();
  return 0;
}
```

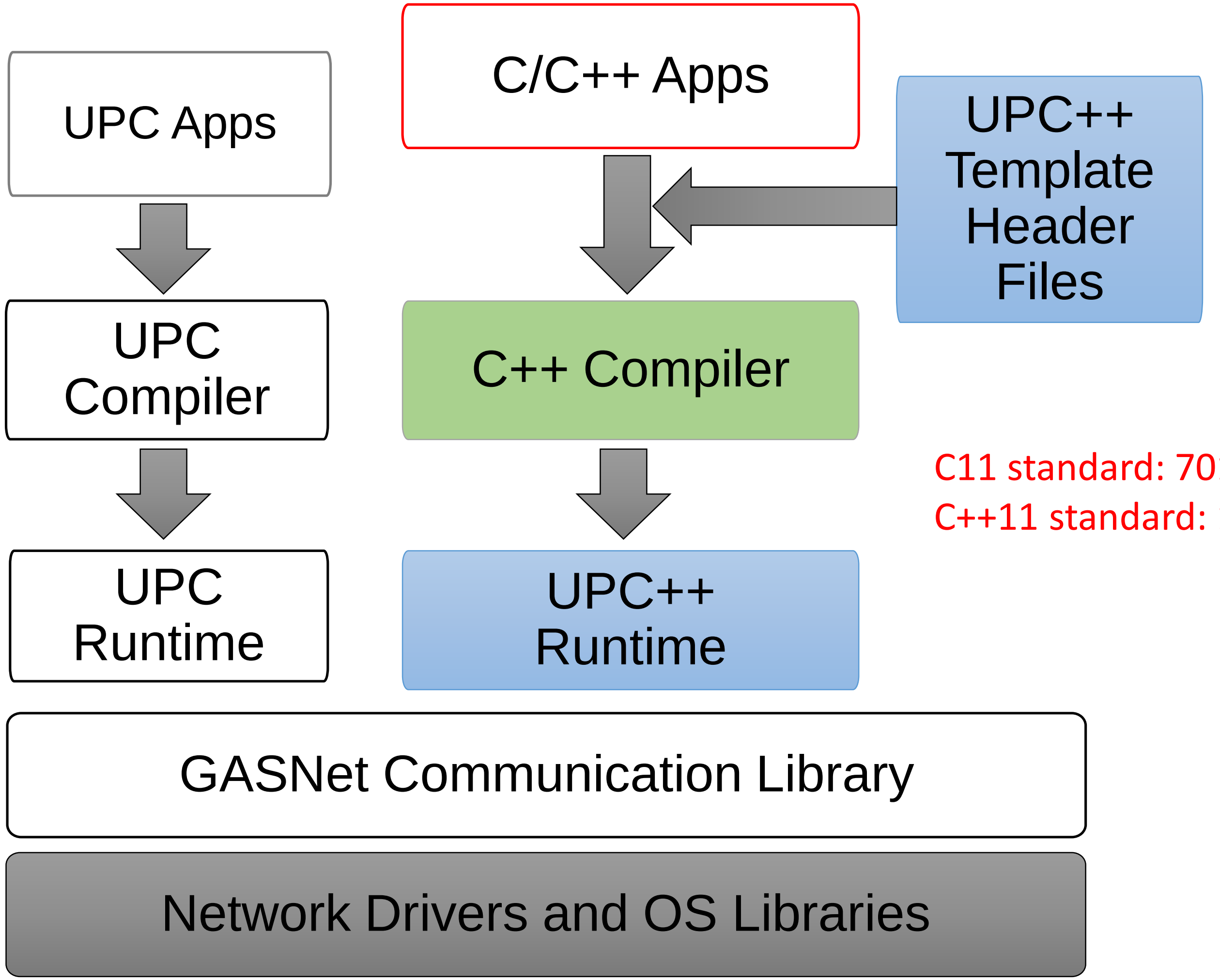
UPC++ on Cori

- UPC++ is available via the **upcxx** module on Cori, which provides the **upc++** compiler wrapper
- Compiling and running UPC++ application on Cori:

```
cori08> module load upcxx

cori08> upc++ -std=c++11 upcxx_hello.cpp -o upcxx_hello.x

cori08> salloc -N 2 -t 10:00 -p debug -C haswell

nid00461> srun -n 64 ./upcxx_hello.x
```

1. Compiling UPC++ Applications: https://bitbucket.org/upcxx/upcxx/wiki/Compiling%20UPC++%20Applications
2. Running UPC++ Applications: https://bitbucket.org/upcxx/upcxx/wiki/Running%20UPC++%20Applications

UPC++ Software Stack



C11 standard: 701 pages
C++11 standard: 1334 pages

UPC++ Introduction

- Shared variable

```
shared_var<int> s;    // int in the shared space
```

- Global pointers (to remote data)

```
global_ptr<LLNode> g;    // pointer to shared space
```

- Shared arrays

```
shared_array<int> sa(8);    // array in shared space
```

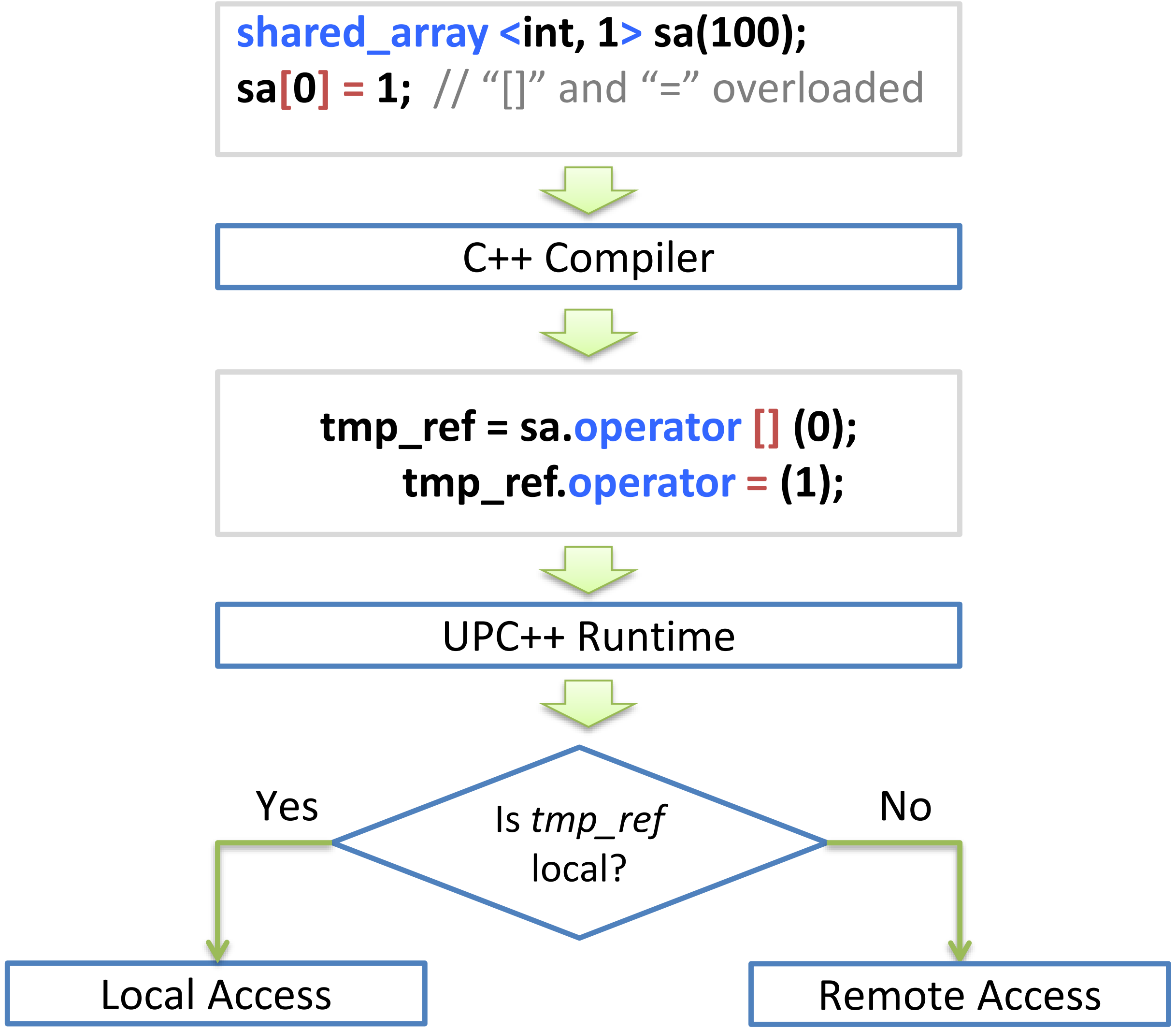
- Locks

```
shared_lock l;    // lock in shared space
```

- Default execution model is SPMD, but with optional async

```
async(place)(Function f, T1 arg1,…);
wait();    // other side does poll()
```

UPC++ Translation Example



Dynamic Global Memory Management

- Global address space pointers (pointer-to-shared)

```
global_ptr<data_type> ptr;
```

- Dynamic shared memory allocation

```
global_ptr<T> allocate<T>(uint32_t where, size_t count);
void deallocate(global_ptr<T> ptr);
```

- Example: allocate space for 512 integers on rank 2

```
global_ptr<int> p = allocate<int>(2, 512);
```

One-Sided Data Transfer Functions

```
// Copy count elements of T from src to dst
upcxx::copy<T>(global_ptr<T> src, global_ptr<T> dst, size_t count);

// Non-blocking version of copy
upcxx::async_copy<T>(global_ptr<T> src, global_ptr<T> dst,
                             size_t count);

// Synchronize all previous asyncs
upcxx::async_wait();
```

Asynchronous Task Execution

- C++ 11 **async** function

```
std::future<T> handle = std::async(Function&& f, Args&&… args);
handle.wait();
```

- UPC++ **async** function

```
// Remote Procedure Call
upcxx::async(place)(Function f, T1 arg1, T2 arg2,…);
upcxx::wait();

// Explicit task synchronization
upcxx::event e;
upcxx::async(place, &e)(Function f, T1 arg1, …);
e.wait();
```

Further Readings

- Coarrays in the next Fortran Standard: ftp://ftp.nag.co.uk/sc22wg5/N1801-N1850/N1824.pdf
- OpenSHMEM Specification 1.3: http://www.openshmem.org/site/sites/default/site_files/OpenSHMEM-1.3.pdf
- OpenSHMEM Tutorials: http://openshmem.org/site/Documentation/Tutorials
- UPC Language Specifications, v1.3: https://upc-lang.org/assets/Uploads/spec/upc-lang-spec-1.3.pdf
- Berkeley UPC User's Guide: http://upc.lbl.gov/docs/user/index.shtml
- PGAS with HPC, *by* Kathy Yelick
- Programming in UPC, *by* Tarek El-Ghazawi
- GASNet: http://gasnet.lbl.gov/
- A tutorial of UPC++: https://bitbucket.org/upcxx/upcxx/wiki/Tutorial
- UPC++, *by* Yili Zheng

Thank you!