KOÇ UNIVERSITY

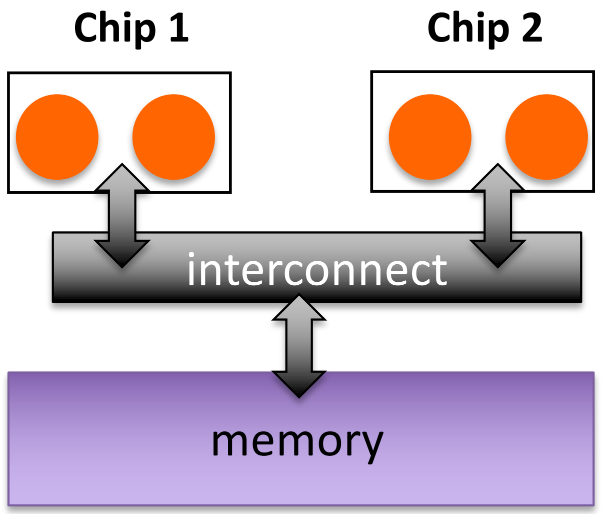
Partitioned Global Address Space (PGAS)

Didem Unat

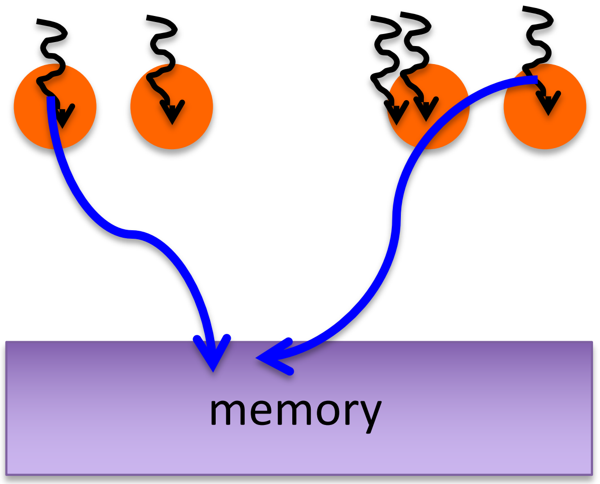
COMP 429/529 Parallel Programming

Shared-Memory Programming Model

- Shared-address space programming
  - Threads communicate through shared memory as opposed to messages
  - Threads coordinate through synchronization (also through shared memory).

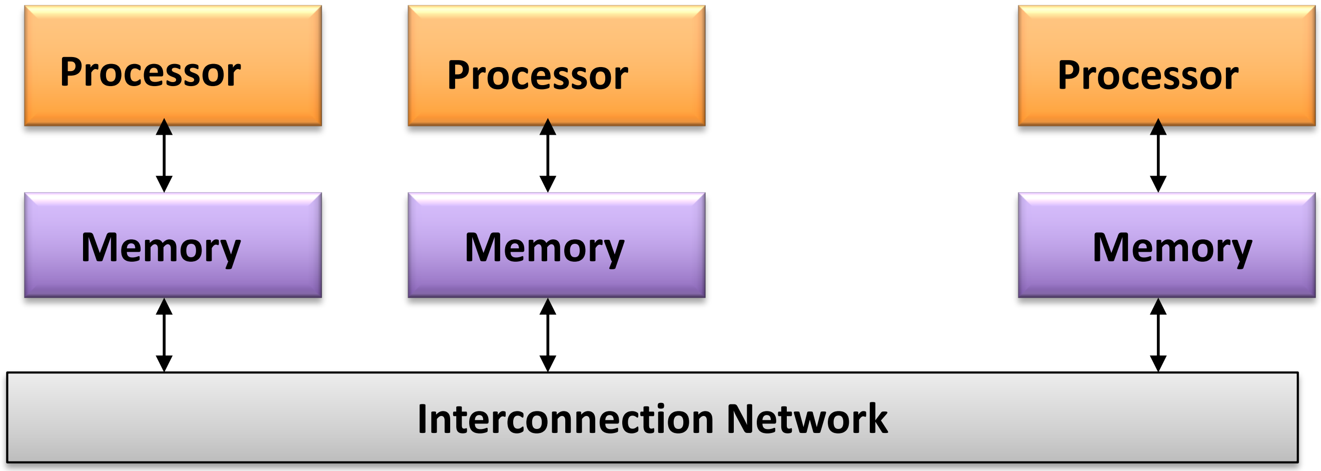


Recall shared memory system
(can be either UMA, NUMA)



Message Passing Programming Model

- Programs execute as a set of P processes (user specifies P)
- Each processor has its own private address space
  - Processors share data by *explicitly* sending and receiving information (message passing)
  - Coordination is built into message passing primitives (message send and message receive)



Shared Memory vs. Message Passing

Shared Memory

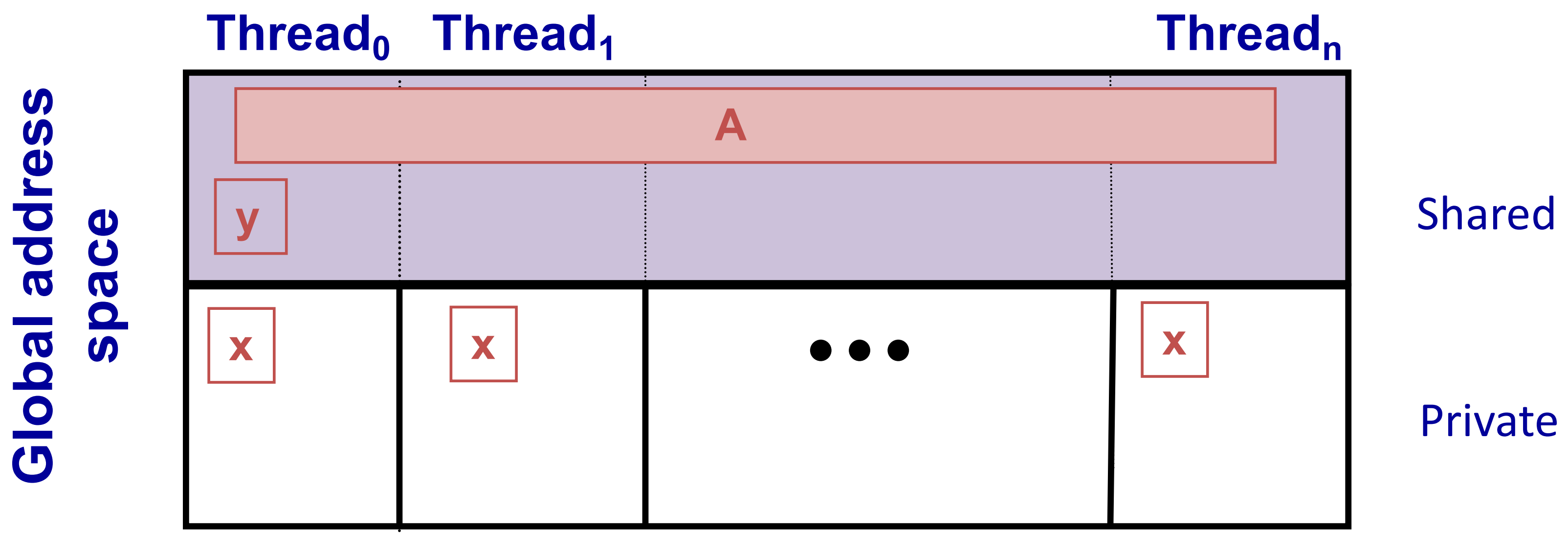
- Advantage: Convenience
  - Can share data structures
  - Just annotate loops
  - Closer to serial code
- Disadvantages
  - No locality control
  - Does not scale
  - Race conditions

Message Passing

- Advantage: Scalability
  - Locality control
  - Communication is all explicit in code (cost transparency)
- Disadvantage
  - Need to rethink entire application / data structures
  - Lots of tedious pack/unpack code

Partitioned Global Address Space (PGAS) Model

- *Global address space:* thread may directly read/write remote data
  - Hides the distinction between shared/distributed address spaces
- *Partitioned:* data is designated as **local or global**
  - Does not hide this: critical for locality and scaling



- Each process has declared a private copy of variable x
- Process 0 has declared a shared variable y
- A shared array A is distributed across the global address space

UPC++

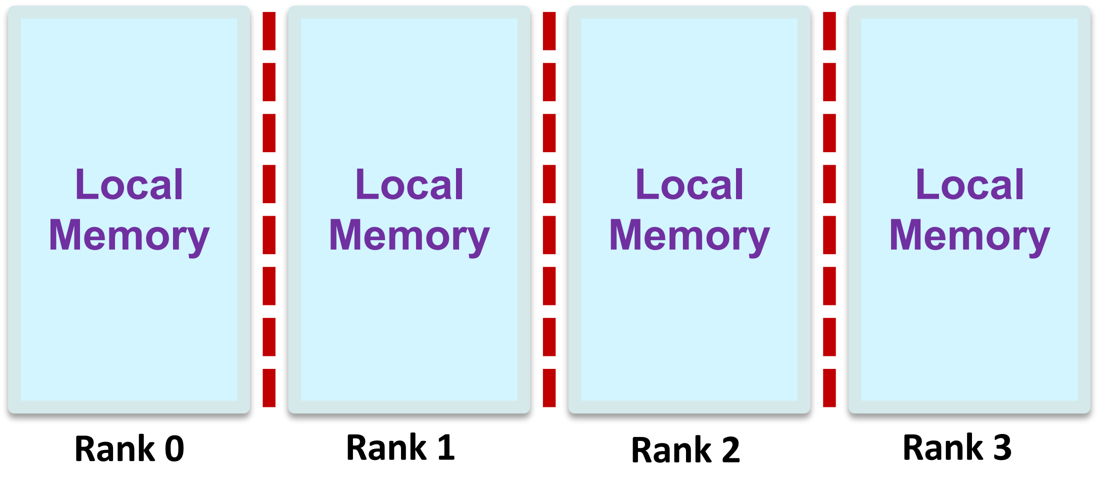
- Unified Parallel C (**UPC++**) is an extension of the C++ programming language designed
  - Provides a uniform programming model for both shared and distributed memory hardware

- A number of threads working independently in a SPMD fashion
  - Number of threads specified at compile-time or run-time; available as program variable **THREADS**
  - **MYTHREAD** specifies thread index (**0..THREADS-1**)
  - **upc_barrier** is a global synchronization: all wait
  - There is a form of parallel loop that we will see later

- There are other PGAS languages such as X10, Chapel and OpenShmem

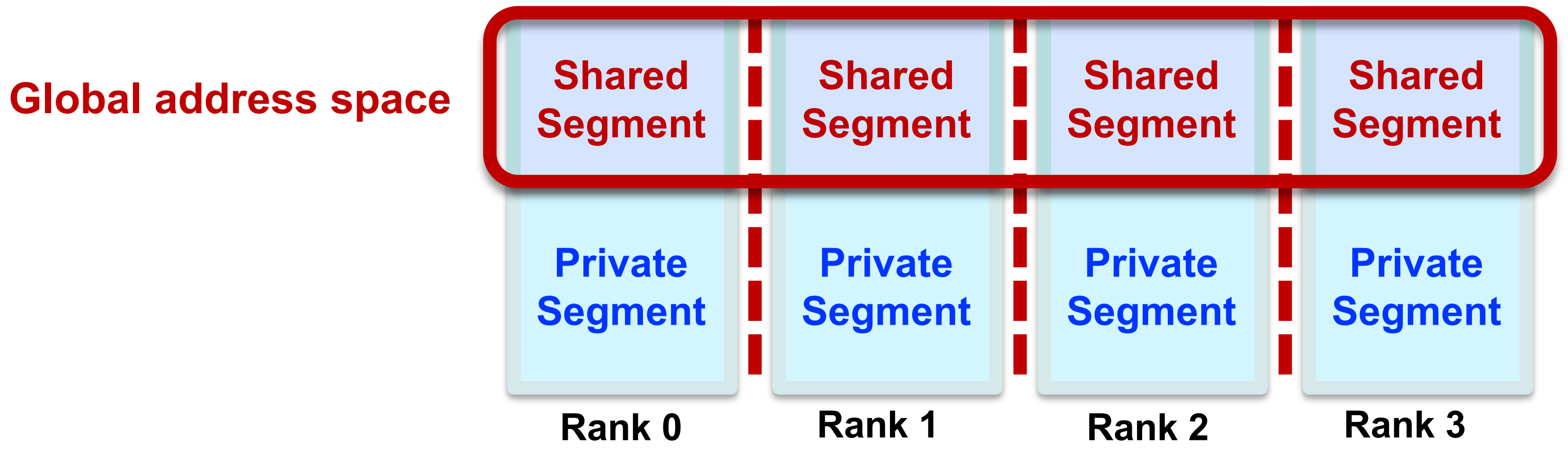
Preliminaries for UPC++

- UPC++ is an SPMD model, like MPI
- A distributed memory parallel computer is an abstract collection of processing elements, an indivisible computing resource with local memory, AKA a *rank*
- The number of ranks is fixed throughout the program



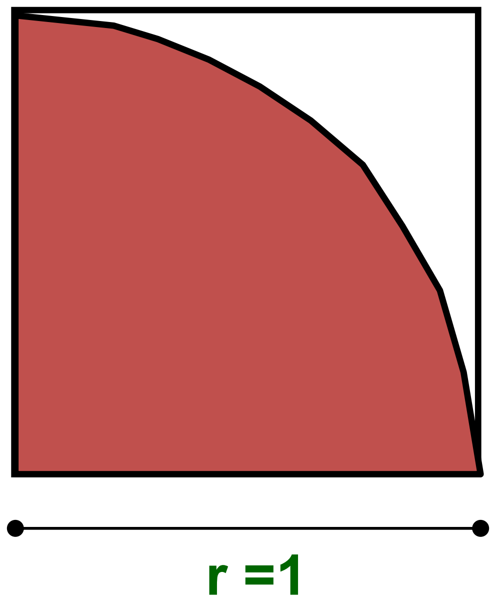
A Partitioned Global Address Space

- Global Address Space (private address spaces only)
  - Ranks read & write *shared segments* of memory, set up at program initialization time, via *1 sided communication*
  - explicit sender & receiver to copy data between private address spaces
- Partitioned (Not applicable to message passing)
  - *Global pointers* to shared segments have affinity to a particular rank
  - Explicitly managed by the programmer to optimize for locality



Example: Monte Carlo Pi Calculation

- Estimate Pi by throwing darts at a unit square
- Calculate percentage that fall in the unit circle
  - Area of square = $r^2 = 1$
  - Area of circle quadrant = ¼ * $p\ r^2 = p/4$
- Randomly throw darts at x,y positions
- If $x^2 + y^2 < 1$, then point is inside circle
- Compute ratio:
  - # points inside / # points total
  - $p = 4 * \text{ratio}$



Independent Estimation of Pi in UPC

```
void main(int argc, char **argv) {

    int i, hits, trials = 0;
    double pi;

    if  (argc != 2)trials = 1000000;
    else trials = atoi(argv[1]);

    srand(MYTHREAD*17);

    for (i=0; i < trials; i++)
       hits += hit();
    pi = 4.0*hits/trials;
    printf("PI estimated to %f.", pi);
  }
```

Each thread gets its own copy of these variables

Each thread can use input arguments

Initialize random in math library

Each thread calls “hit” function separately

Helper Code for Pi in UPC

- Required includes:

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

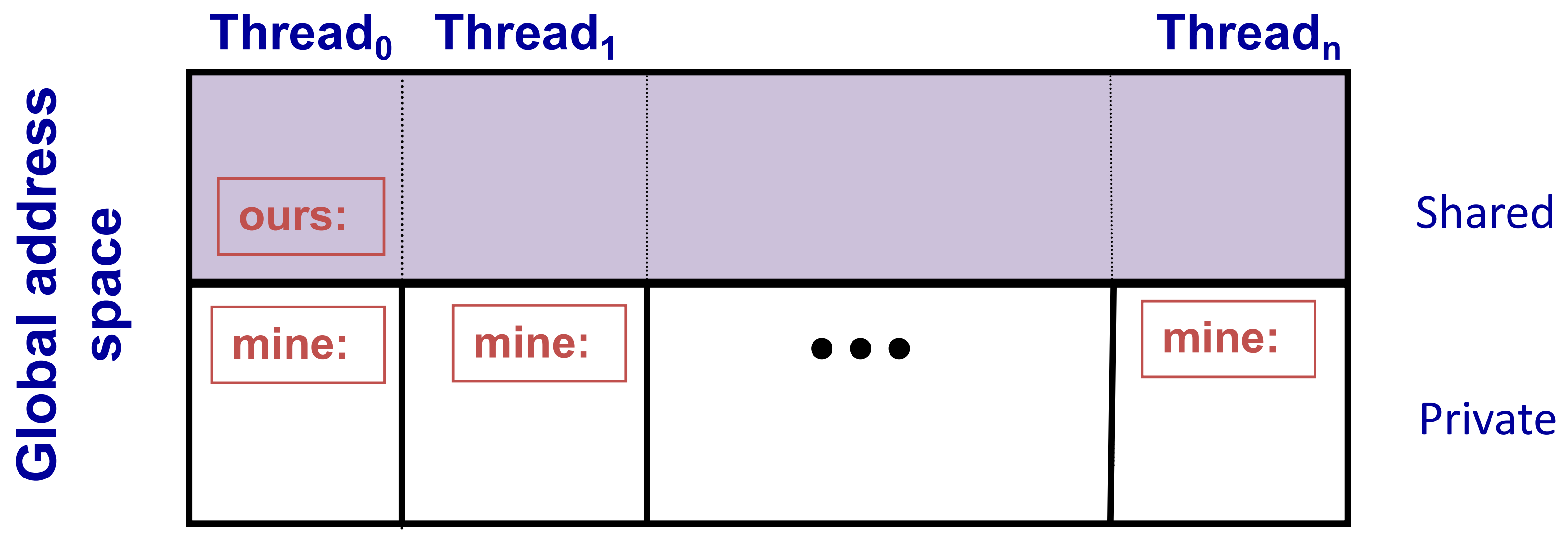
- Function to throw dart and calculate where it hits:

```
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);
  }
}
```

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;
```



Pi in UPC: Shared Memory Style

```
 shared int hits;
void 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);
    }
 }
```

shared variable to record hits

divide work up evenly

accumulate hits

There is a bug in the program. What is the problem with this program?

Pi in UPC: Shared Array Version

- Alternative fix to the race condition
- Have each thread update a separate counter:
  - But do it in a shared array
  - Have one thread compute sum

```
shared int all_hits [THREADS];
main(int argc, char **argv) {
   … declarations an initialization code omitted
   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);
   }
}
```

all_hits is shared by all processors, just as hits was

update element with local affinity

Synchronization - Locks

- Locks in UPC are represented by an opaque type:

  `upc_lock_t`
- Locks must be allocated before use:

  `upc_lock_t *upc_all_lock_alloc(void);`

  allocates 1 lock, pointer to all threads

  `upc_lock_t *upc_global_lock_alloc(void);`

  allocates 1 lock, pointer to one thread
- To use a lock:

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

  use at start and end of critical region
- Locks can be freed when not in use

  `void upc_lock_free(upc_lock_t *ptr);`

Pi in UPC: Shared Memory Style

Use locks like pthreads, but use shared accesses judiciously

```
shared int hits;
```

one shared scalar variable

```
main(int argc, char **argv) {
    int i, my_hits, my_trials = 0;
    upc_lock_t *hit_lock = upc_all_lock_alloc();
```

other private variables

create a lock

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

accumulate hits locally

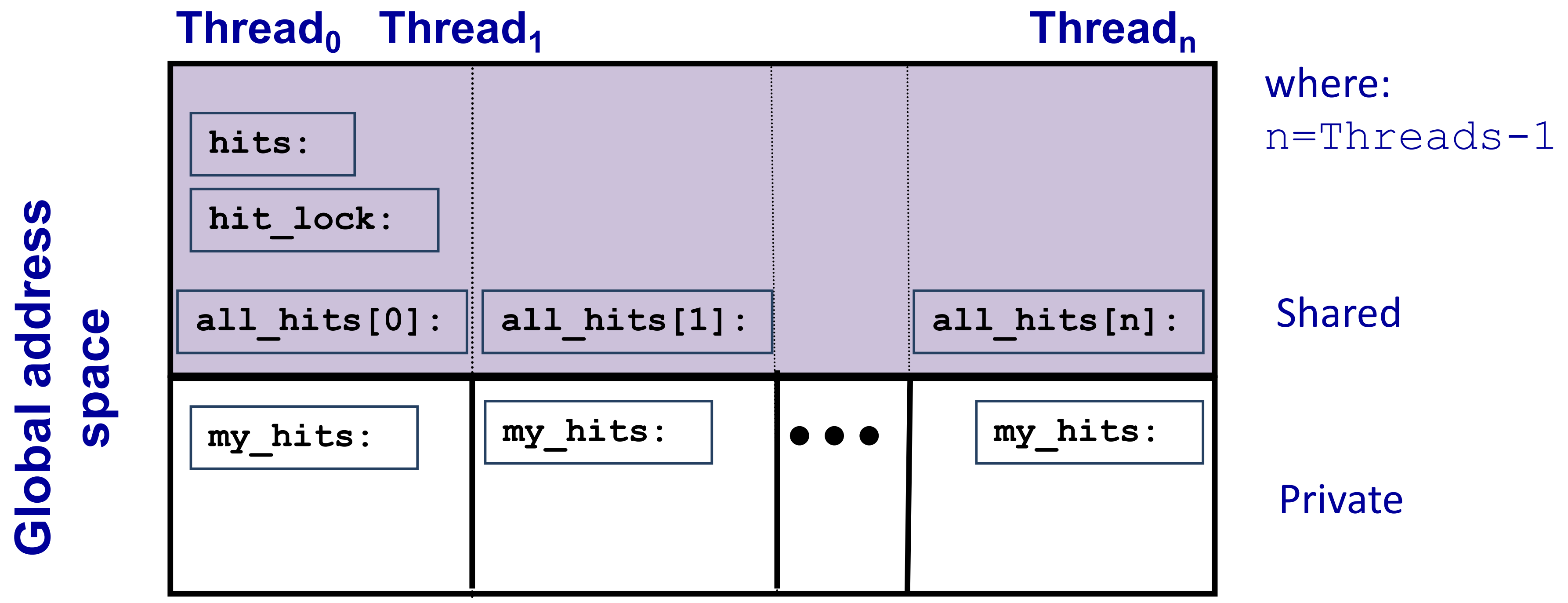
```
    upc_lock(hit_lock);
    hits += my_hits;
    upc_unlock(hit_lock);
    upc_barrier;
```

accumulate across threads using a lock

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

Recap: Private vs. Shared Variables in UPC

- We saw several kinds of variables in the pi example
  - Private scalars (**`my_hits`**)
  - Shared scalars (**`hits`**)
  - Shared arrays (**`all_hits`**)
  - Shared locks (**`hit_lock`**)



One-Sided Communication

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

1-sided Communication in UPC++

- A rank can read or write memory in another address space via a *global pointer*, *target* is unaware
- Also called Remote Memory Access (RMA, MPI, too)
- Unlike message passing, no explicit sender & receiver

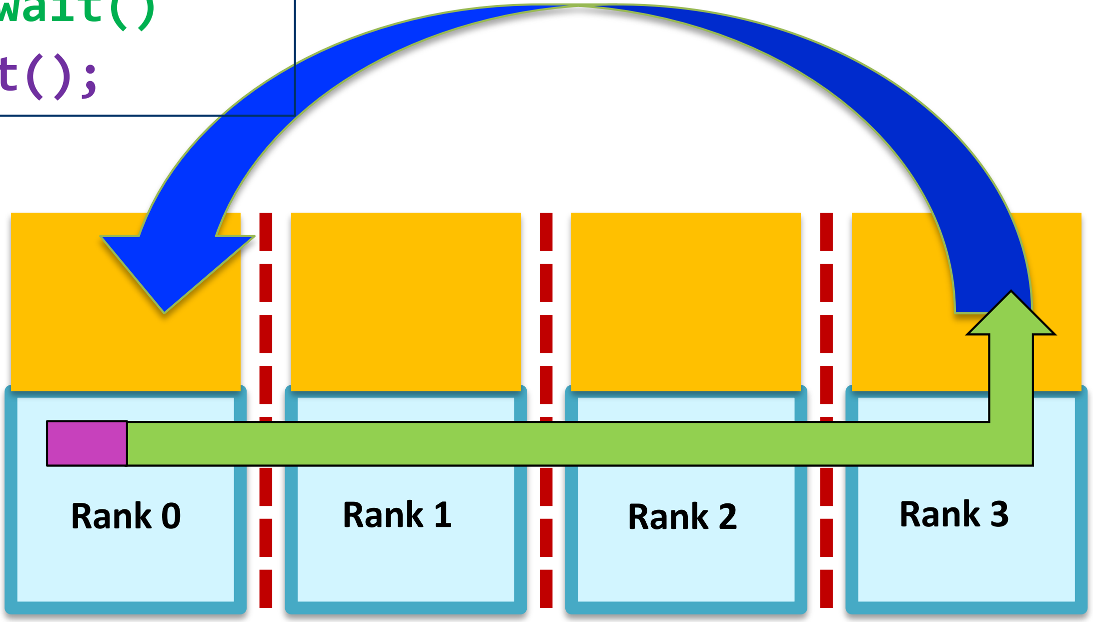
```
global_ptr<T> gptr1 = . . .;
T t1 = rget(gptr1).wait()
rput(gptr2,x,2).wait();
```

Wait returns with result to the requestor when rget completes

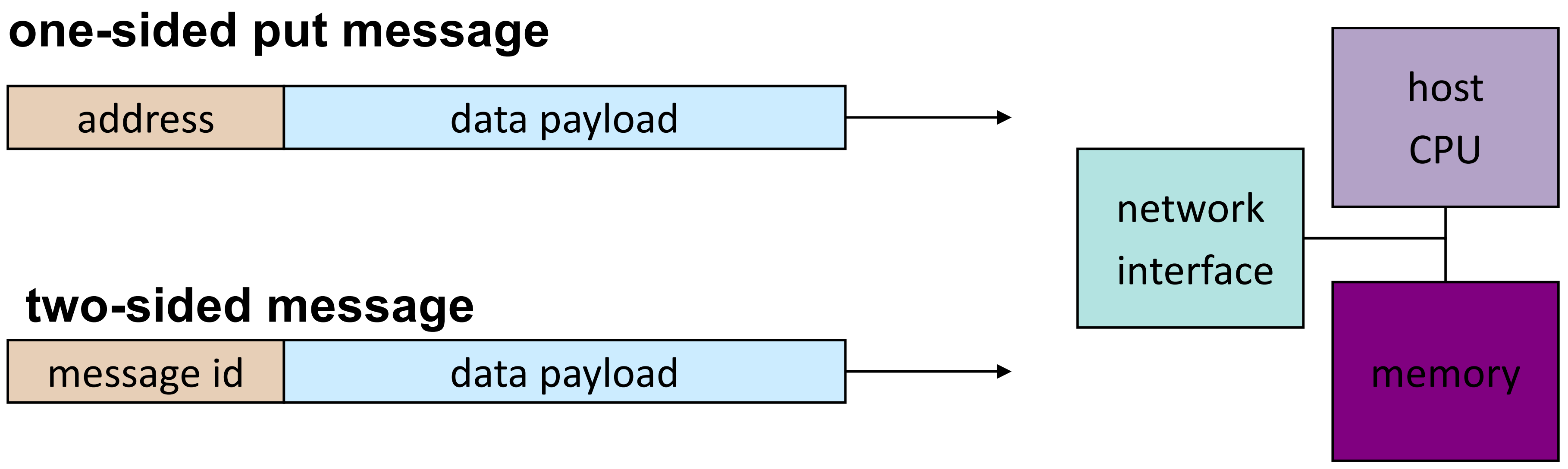
Global address space

Start the get

Private memory



One-Sided vs Two-Sided



- A one-sided put/get message can be handled directly by a network interface with RDMA support
  - Avoid interrupting the CPU or storing data from CPU (preposts)
  - Put() and Get()
- A two-sided messages needs to be matched with a receive to identify memory address to put data
  - Send() and Receive()

Remote Memory Access (RMA)

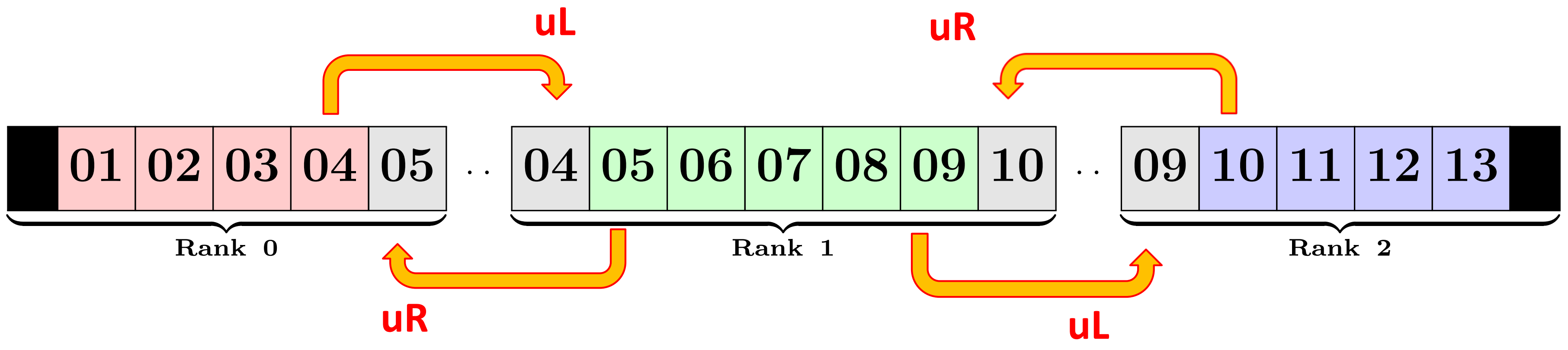
- If communication *pattern* is not known *a priori*, but the data locations are known, the send-receive model requires an extra step to determine how many sends- receives to issue
- RMA, however, can handle it easily because only the origin or target process needs to issue the put or get call
- This makes dynamic communication easier to code in RMA

MPI support for One-Sided Comm

- **MPI_Win_create** exposes local memory to RMA operation by other processes in a communicator
  - Collective operation
  - Creates window object
- **MPI_Win_free** deallocates window object
- **MPI_Put** moves data from local memory to remote
- memory
- **MPI_Get** retrieves data from remote memory into local memory
- **MPI_Accumulate** updates remote memory using local values
- Data movement operations are non-blocking
- Subsequent synchronization on window object needed to ensure operation is complete

1-D Point Jacobi in UPC++

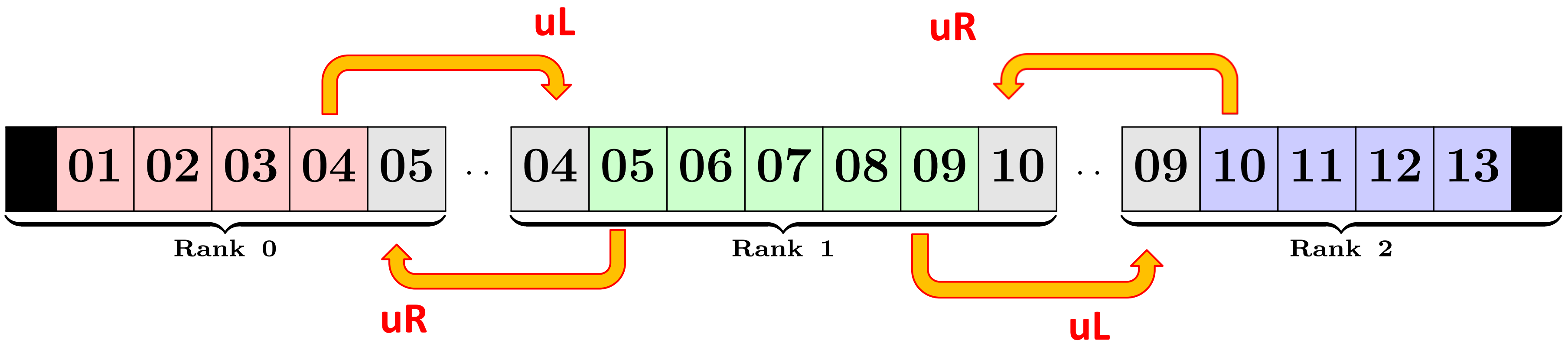
- Each rank allocates solution U in global memory



```
global_ptr<double> U  = new_array<double>(n);

global_ptr<double> uL = … ,  uR = … ;
```

1-D Point Jacobi in UPC++

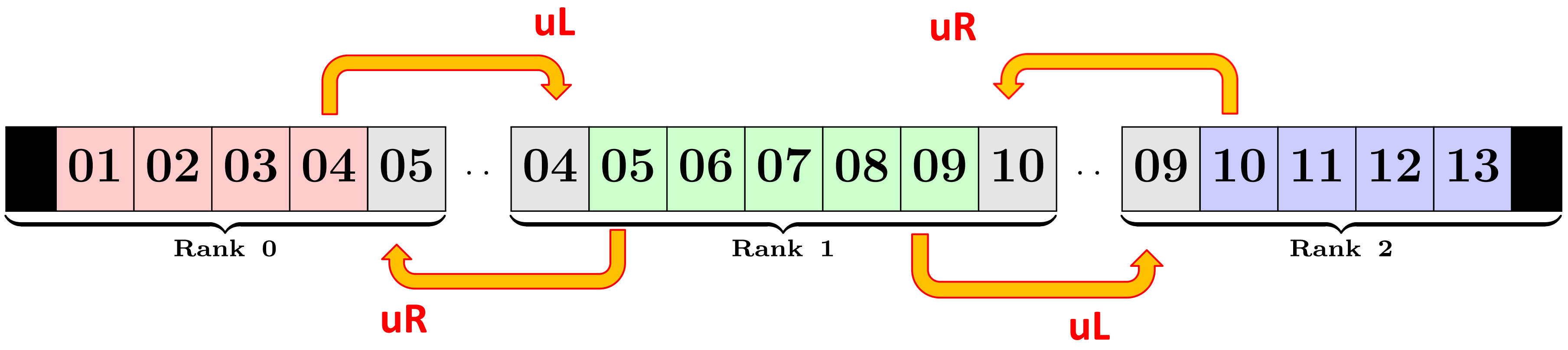
- Each rank allocates solution U in global memory
- u is a raw pointer to local data in the global segment



```
global_ptr<double> U  = new_array<double>(n);
double *u = U.local() ; // U^new allocated similarly
global_ptr<double> uL = … ,  uR = … ;
for (i = 1 to MaxIter ){
     fillGhost(u,n, uL, uR);
     Sweep(U^new, U);
     Swap pointers…
 }
```

FillGhost in UPC++

- uL and uR have been set up previously to point to left and right neighboring solution arrays



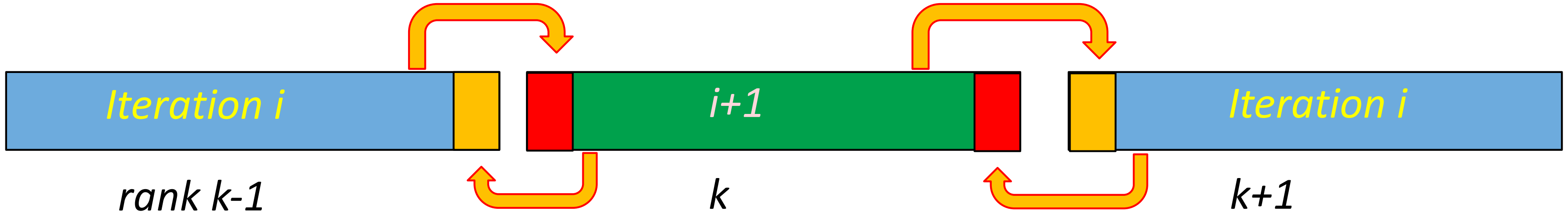
```
fillGhost(u,n,uL,uR):

 if ( !(uL.is_null()))
     u[0] = rget(uL).wait();    //get 1-sided comm

 if ( !(uR.is_null()))

     u[n-1] = rget(uR).wait();  //get 1-sided comm
```

The UPC++ Memory Model

- UPC++ runs under a different memory model than message passing
- Like shared memory, there can be *race conditions*
- A race condition arises because we are reading and writing global storage
- The timing of the accesses can affect the outcome
- We say that we have a *non-deterministic computation* when the outcome can vary from run to run
- Message passing avoids the race condition, since data movement is coupled with synchronization
- We'll illustrate a race condition (and the solution) in 1-D Point Jacobi

Where is the Race Condition?

- Since ranks run at different rates, its possible that a neighboring rank can capture its ghost cells and move on to the next iteration (i+1) before its neighbors do
- A *straggler* that's still in iteration i could obtain data from a neighbor that is computing a future iteration i+1
- This behavior is unpredictable, & we may not observe it



```
for (i = 1 to MaxIter ){
  fillGhost(u,n, uL, uR);
  Sweep();
  Swap pointers…
 }
```

```
fillGhost(u,n,uL,uR):
 if ( !(uL.is_null()))
      u[0] = rget(uL).wait();
  if ( !(uR.is_null()))
      u[n-1] = rget(uR).wait();
```

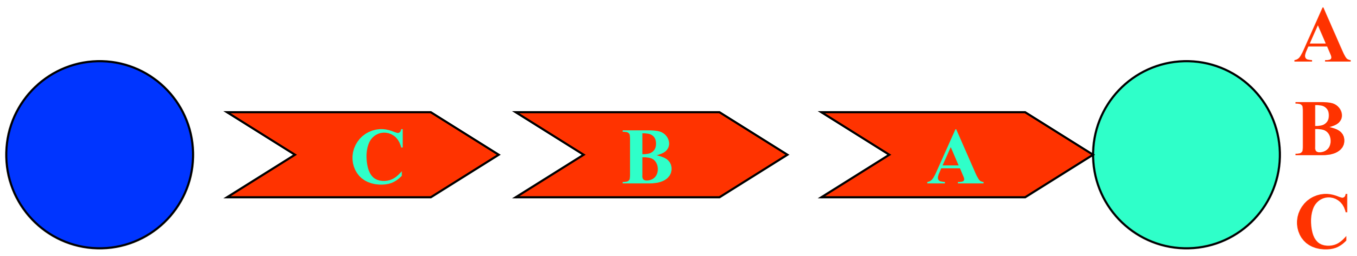
The Solution: a Barrier

- No rank can proceed past the barrier until all have arrived, ensuring that no rank runs ahead of others
- Barrier runs in time $\log_2(P)$
  allreduce = reduce + bcast

```
for (i = 1 to MaxIter ){
  fillGhost(u ,n, uL, uR);
  Sweep();
  Swap pointers…
  barrier();
}
```

Comparing message passing with PGAS

- Recall that message passing is 2-sided
  - There is an explicit sender and receiver
  - Data movement and synchronization are coupled
  - Messages have an associated context (e.g. tag) that must be matched to handle incoming messages correctly
- Ordering guarantees are not semantically matched to the hardware
- PGAS change various communication attributes
  - Execute fewer instructions to perform a transfer (reduce α)

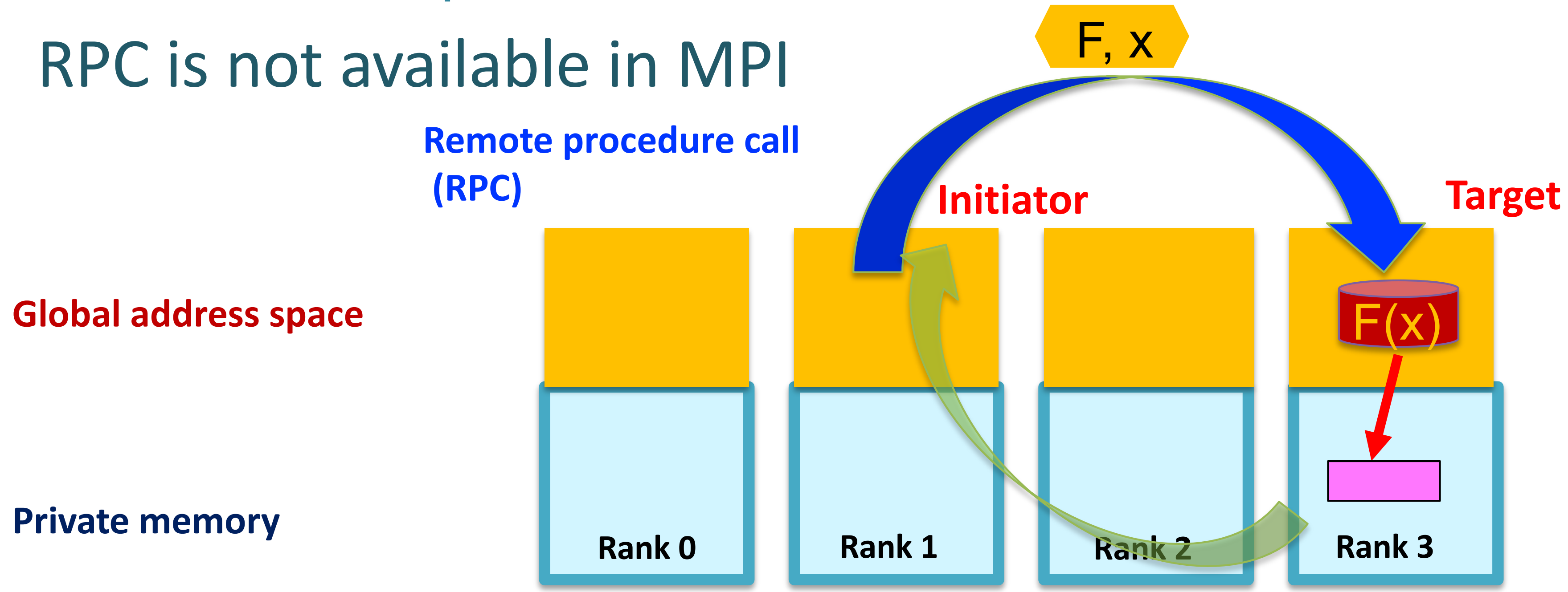


UPC++ reduces the Overheads

- RMA lets each rank directly access one another’s memory via a global reference
  - We don’t need to match sends to receives
  - We don’t need to guarantee message ordering
  - MPI also supports RMA, too
- Looks like shared memory, so we need to handle with race conditions
  - Unlike message passing, synchronization and data movement are separate
- Technology trends provide support that benefits RMA
  - Modern network hardware provides the capability to directly access memory on another node: Remote Direct Memory Access (RDMA)

Remote Procedure Call

- Remote Procedure call (RPC) is different from RMA
  - Executes a function on another rank (the *target*), sending any arguments
  - Returns an optional result to the initiator
- RPC is not available in MPI



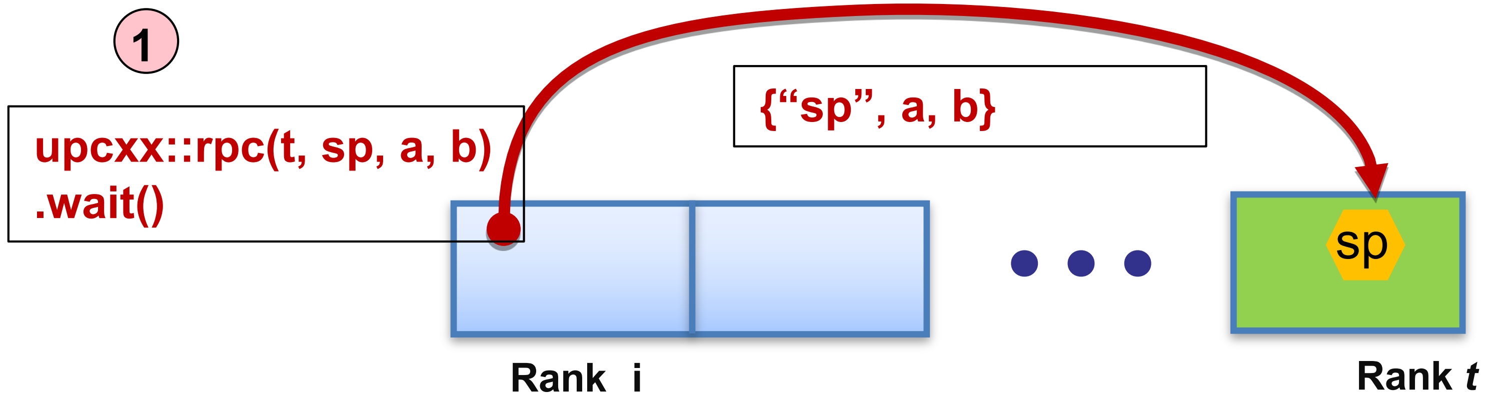
Issuing a Remote Procedure Call

1

- Rank i executes sp( ) on rank *t* via an RPC

  int sp (int a, int b) { return a + b; }

- int sum = rpc(t, sp, a, b).wait();



Issuing a Remote Procedure Call

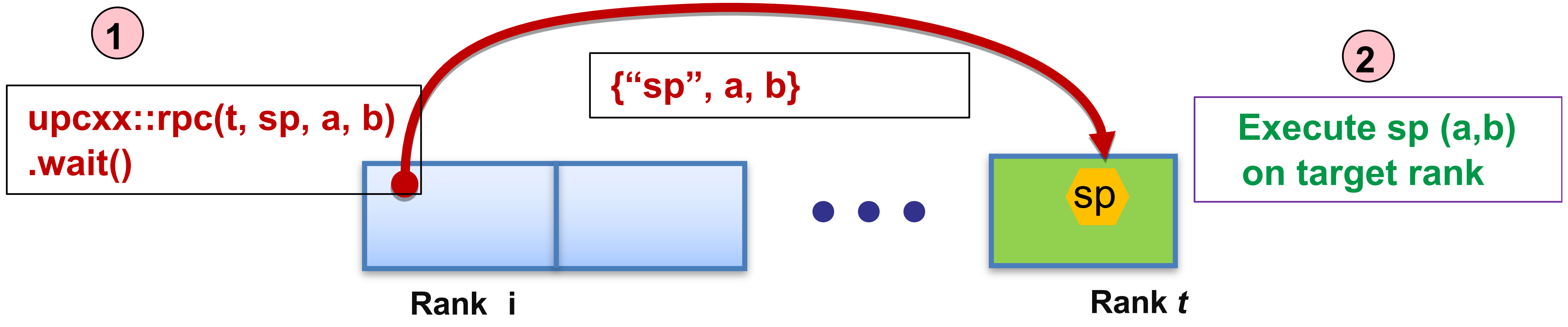
- Rank i executes sp( ) on rank *t* via an RPC ①

```
int  sp (int  a, int  b) { return  a + b; }
```

-
```
int sum = rpc( t, sp, a, b).wait();
```

- The target rank t will execute the handler function sp( ) at some future time determined at the target ("makes progress") ②

- The details of progress are hidden from the programmer



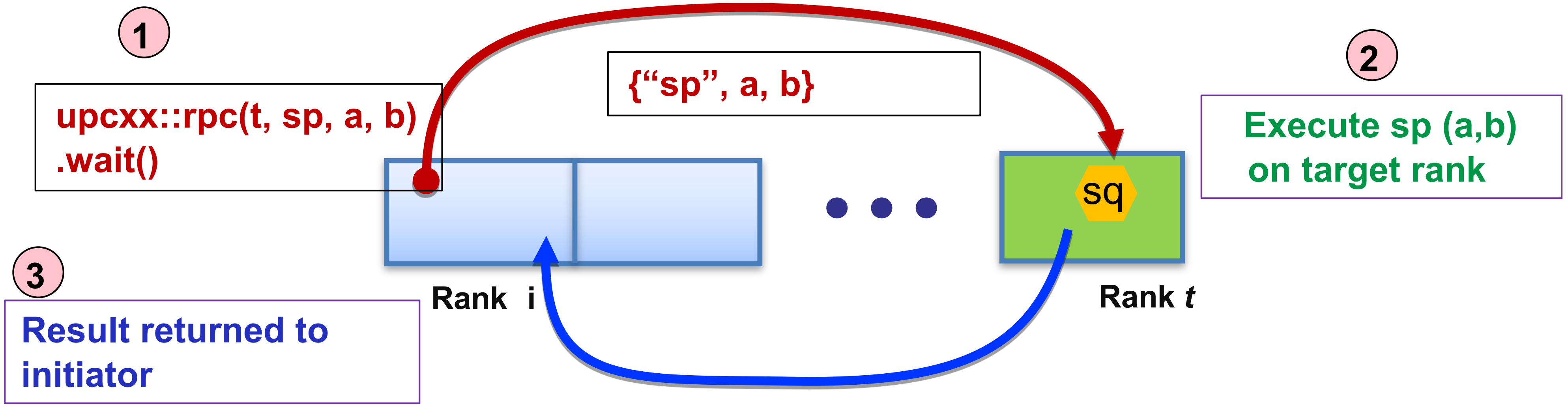
Issuing a Remote Procedure Call

- Rank i executes sp( ) on rank *t* via an RPC (1)

  ```
  int  sp (int  a, int  b) { return  a + b; }
  ```

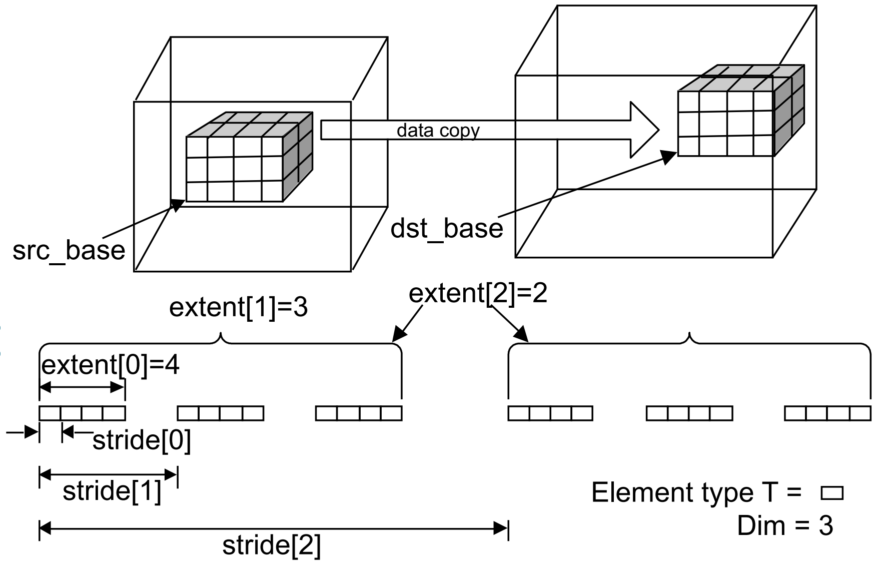
-
  ```
  int sum = rpc( t, sp, a, b),wait();
  ```

- The target rank t will execute the handler function sp( ) at some future time determined at the target ("makes progress") (2)
- When the handler completes, result returned to rank i (3)



Other Features of UPC++

- Remote Atomic operations
- Completions
  - Know when the source memory can be modified, when the operation has completed at the target
  - Attach an RPC to RMA completion
- Non-contiguous transfers
- Teams
  (like MPI communicators)
- Memory Kinds
  Unified treatment of host
  & device memory



Acknowledgments

- These slides are inspired and partly adapted from
  - Kathy Yelick (Univ. of California, Berkeley, LBL)
  - Jim Demmel (Univ. of California, Berkeley)
  - William Gropp (UIUC)
  - Scott B. Baden (c) 2020 / PGAS Programming