CSE502: Foundations of Parallel Programming

Lecture 24: End Semester Review

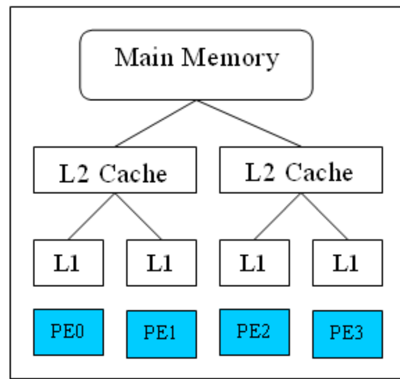
Vivek Kumar

Computer Science and Engineering

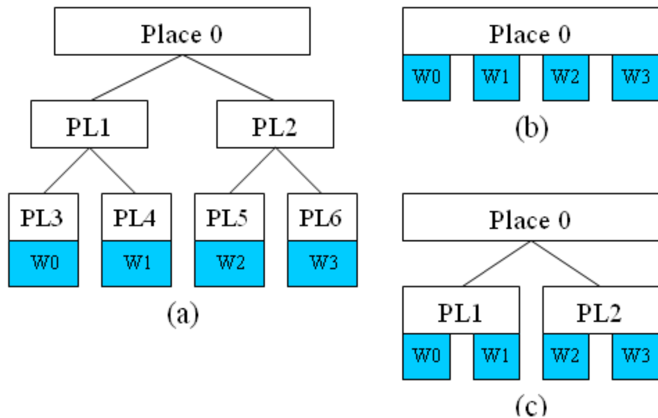
IIIT Delhi

vivekk@iiitd.ac.in

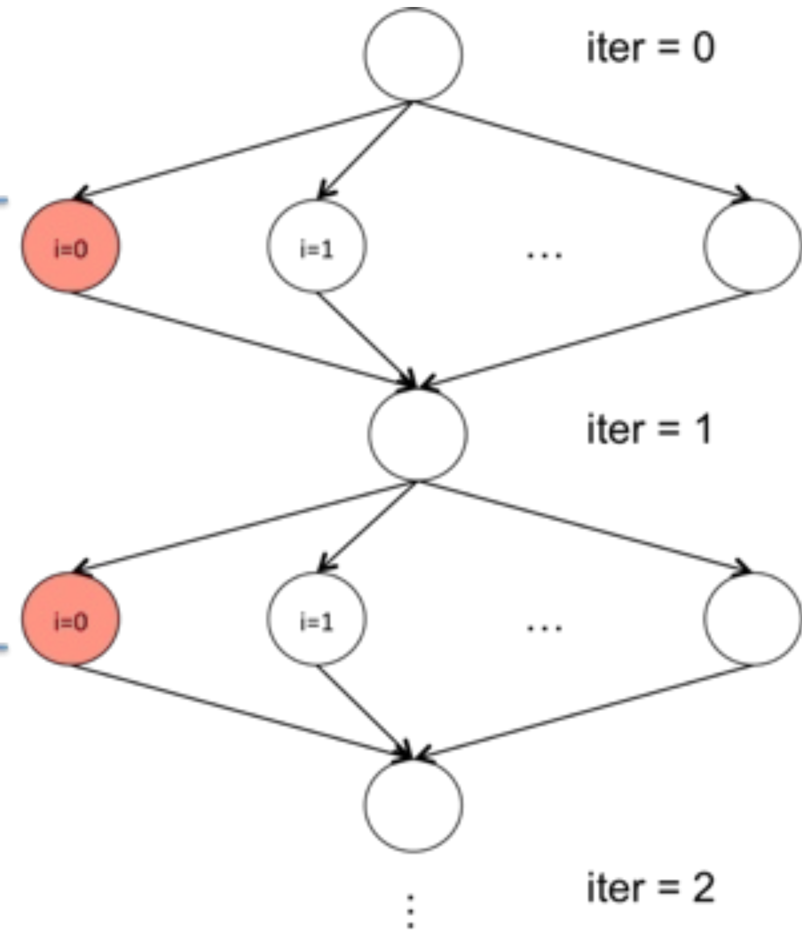
Locality Benefits: Analyzing Locality Iterative Averaging



A Quad-core workstation



Locality benefits will be realized if all instances of chunk 0 execute on the same core and reuse data from the same cache



Source: https://wiki.rice.edu/confluence/download/attachments/24426821/comp322-s16-lec32-slides-v1.pdf?version=1&modificationDate=1483206145705&api=v2

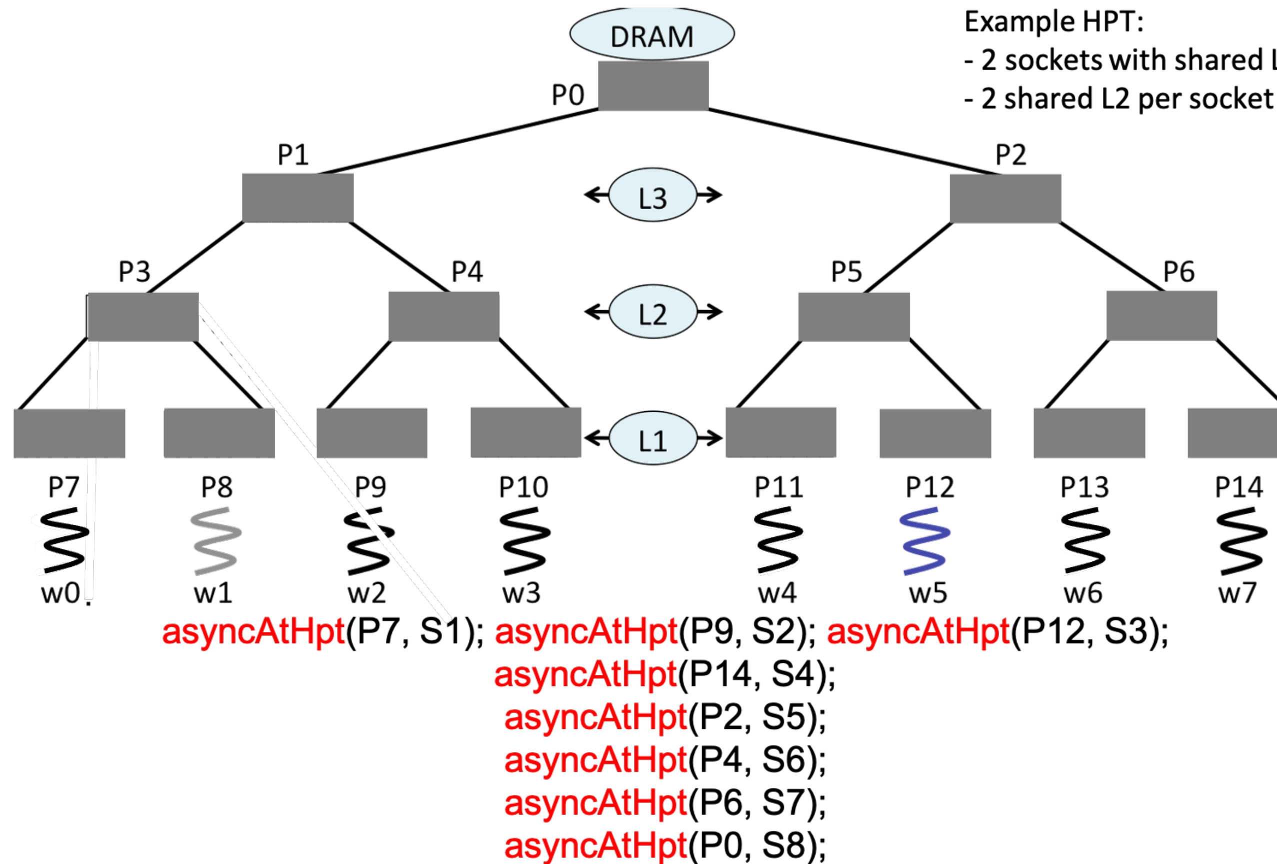
Hierarchical Place Trees in HClib

```
<?xml version="1.0"?>
<!DOCTYPE HPT SYSTEM "hpt.dtd">

<HPT version="0.1" info="HPT test">
   <place num="1" type="mem">
      <place num="2" type="cache">
         <worker num="1"/>
      </place>
   </place>
</HPT>
```

- Abstraction of the memory hierarchy that a HClib program is executed on (using XML document)
- Place denoting affinity group at memory hierarchy level
  - E.g., L1 cache, L2 cache, DRAM

© Vivek Kumar

Hierarchical Place Trees in HClib



- Leaf places include worker threads
  - E.g., W0, W1, W2, W3
- Workers can push task to any place
  asyncAtHpt(place*, lambda)
- Workers can pop/steal only from their parent place hierarchy

HClib Futures: Tasks with Return Values

future_t<T> *f = async_future { S }

- Creates a new child task that executes S, which must terminate with a return statement and return value
- Async expression returns a pointer to a container of type future_t

T result = f.get();

- get() evaluates f and blocks if f's value is unavailable
- Unlike finish which waits for all tasks in the finish scope, a get operation only waits for the specified async_future

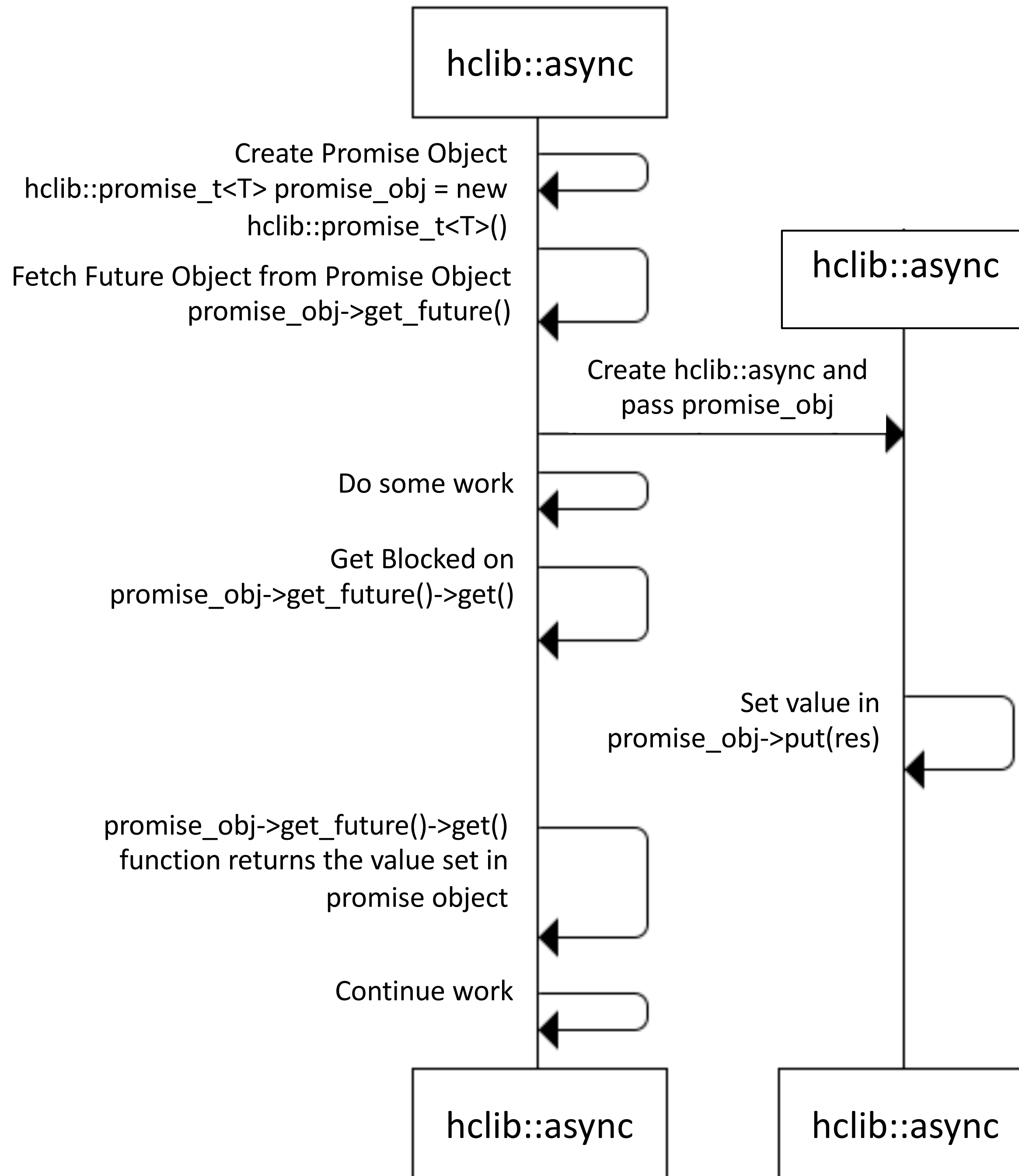
Source: https://wiki.rice.edu/confluence/download/attachments/24426821/comp322-s16-lec5-slides.pdf?version=1&modificationDate=1483206145961&api=v2

hclib::promise v/s hclib::future

- *"A promise is an object that can store a value of type T to be retrieved by a future object (possibly in another thread), offering a synchronization point"*
  - Writable end of an object
- *"A future is an object that can retrieve a value from some provider object or function, properly synchronizing this access if in different threads"*
  - Readable end of an object

Source: http://www.cplusplus.com/reference

hclib::promise_t and hclib::future_t workflow



Modification of Source: http://thispointer.com/c11-multithreading-part-8-stdfuture-stdpromise-and-returning-values-from-thread/

Data-Driven Task (DDT) in HClib

async_await(lambda, fObj_1, fObj_2,…..,fObj_n)

- Unlike any other async tasks that we have seen so far (async, asyncAtHpt, async_future), async_await task is pushed to the deque ONLY after all the future objects in the parameter list are ready with values inside them
  - i.e. the put has been performed on the promise end of each of the future objects

Source: https://wiki.rice.edu/confluence/download/attachments/24426821/comp322-s16-lec14-slides-v1.key.pdf?version=1&modificationDate=1483206145028&api=v2

Introducing Cilk

Identifies a function as a Cilk procedure, capable of being spawned in parallel

Cilk is a faithful extension of C, i.e., it supports serial elision

The named Child procedure can execute in parallel with the parent caller

```
cilk uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x, y;
    x = spawn fib(n-1);
    y = spawn fib(n-2);
    sync;
    return (x + y);
  }
}
cilk int main(int argc, char** argv) {
  int result = spawn fib(40);
  sync;
}
```

Control cannot pass this point until all spawned children have returned

spawn keyword can only be applied to a Cilk function, and cannot be used in a C function

Cilk function cannot be called as normal C function, and must be called with spawn & waited for by a sync

Fully-strict v/s Terminally-strict

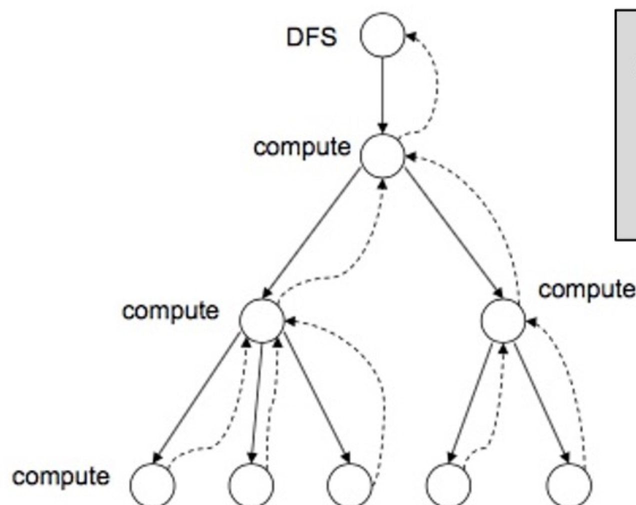
- What is a "strict" computation?
  - A strict computation is one in which all join edges from a task go to one of its ancestor tasks in the computation graph

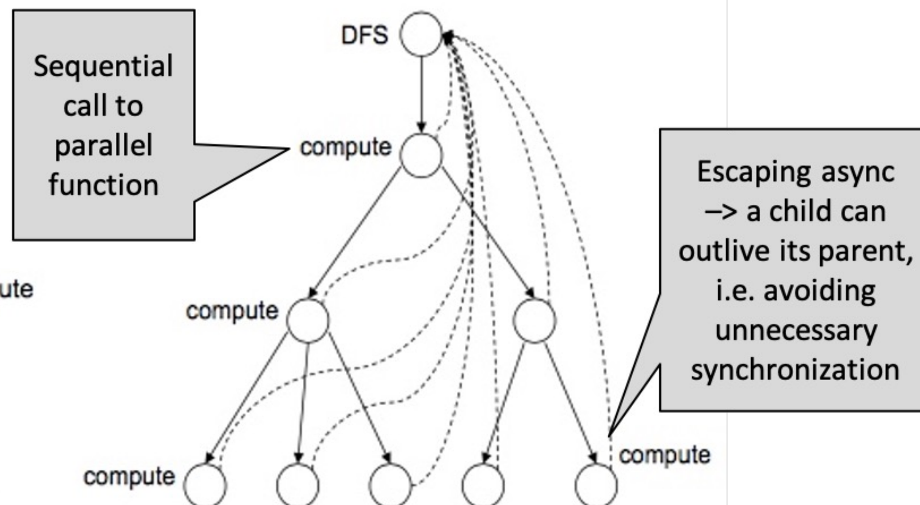
```
cilk void compute(Node* node) {
  int i;
  process(node); //sequential
  for(i=0; i<node->numChild; i++) {
    spawn compute(node->child[i]);
  }
  sync;
}
cilk void DFS(Node* root) {
  spawn compute(root);
  sync;
}
```

Cilk

```
void compute(Node* node) {
  int i;
  process(node); //sequential
  for(i=0; i<node->numChild; i++) {
    async compute(node->child[i]);
  }

}
void DFS(Node* root) {
  finish compute(root);
}
```

HClib

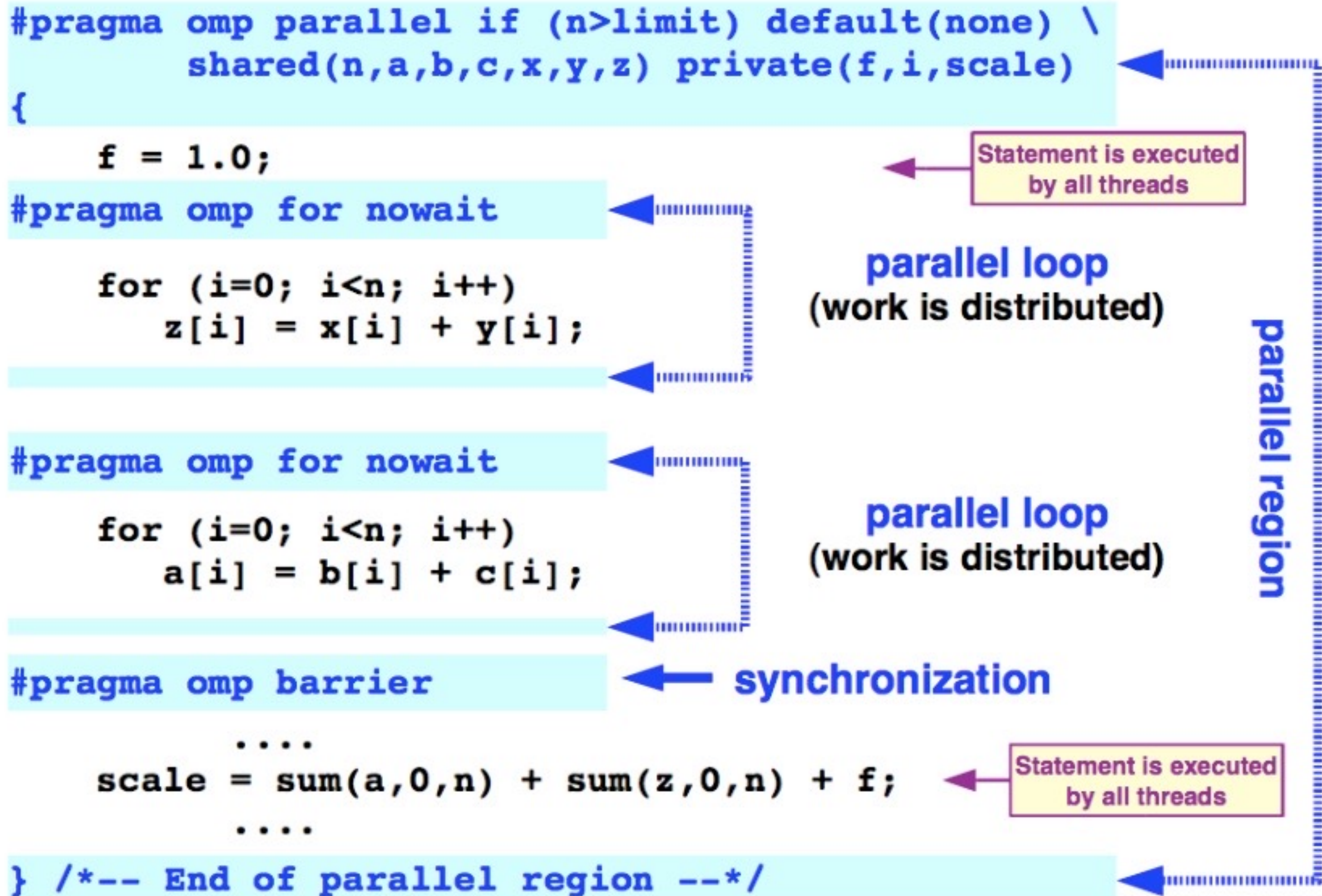




Computing a Product in Parallel using **inlet** & **abort**

```
cilk int product(int *A, int n) {
  int p = 1;
  inlet void mult(int x) {
    p *= x;
    if (p == 0) {
      abort; /* Aborts existing children, */
    }        /* but not future ones.     */
    return;
  }

  if (n == 1) {
    return A[0];
  } else {
    mult( spawn product(A, n/2) );
    if (p == 0) { /* Add check for future */
      return 0;   /* children             */
    }
    mult( spawn product(A+n/2, n-n/2) );
    sync;
    return p;
  }
}
```

OpenMP Parallel Programming Model



Task Directive

Task Synchronization

- **Task Synchronization explained:**

```
#pragma omp parallel num_threads(np)
{
#pragma omp task
    function_A();
#pragma omp barrier
#pragma omp single
    {
#pragma omp task
        function_B();
    }
}
```

np Tasks created here, one for each thread

All Tasks guaranteed to be completed here

1 Task created here

B-Task guaranteed to be completed here

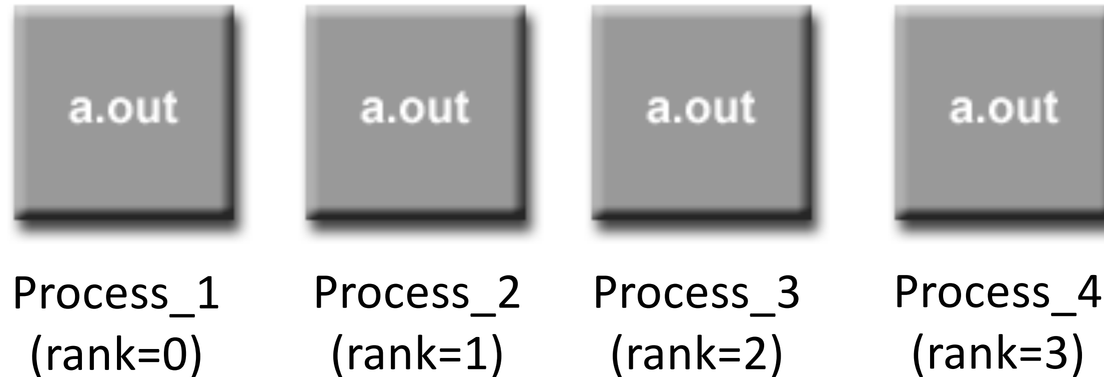
Data Scoping with Tasks

Best Practice to avoid unexpected results !!

```
#pragma omp parallel default(none) shared(A) private(B)
{
   ...
#pragma omp task
   {
       int C;
       compute(A, B, C);
   }
}
```

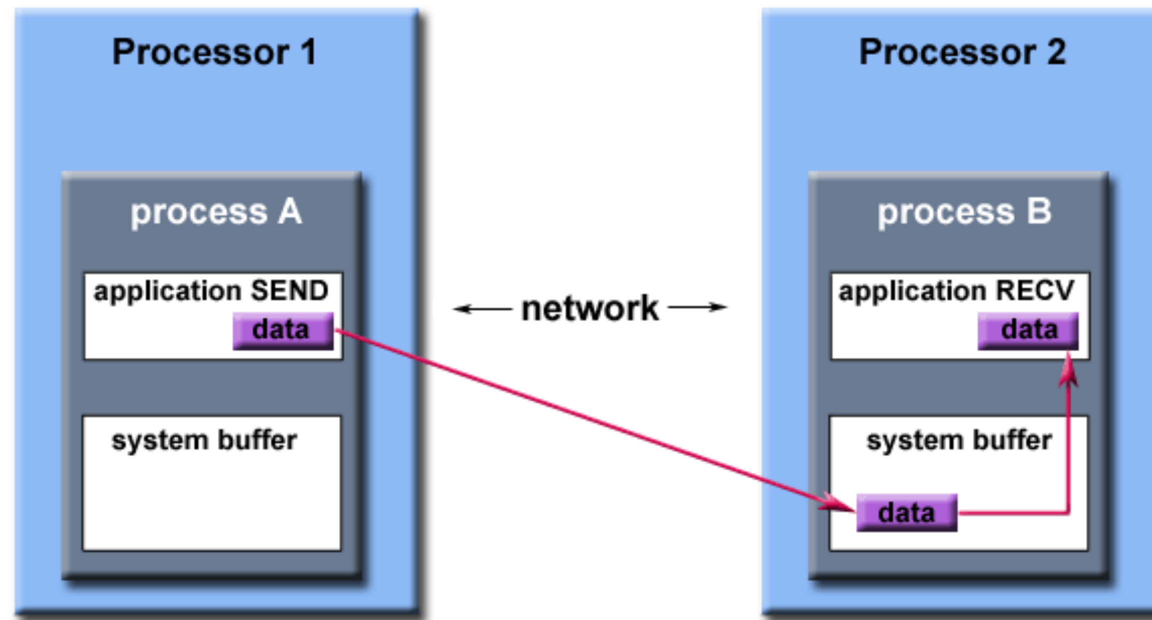
A is shared
B is firstprivate
C is private

MPI: SPMD Pattern



- SPMD: Single Program Multiple Data
- Run the same program on P processing elements (PEs)
- Use the “rank” … an ID ranging from 0 to (P-1) … to determine what computation is performed on what data by a given PE
- Different PEs can follow different paths through the same code

Message Buffering



Path of a message buffered at the receiving process

- Not possible to synchronize every MPI_Send with matching MPI_Recv
  - How to deal if a send arrives before a matching recv is posted?
  - How to deal with multiple sends arriving?
- "**MPI Implementations**" (not MPI standard!) typically reserves a system buffer to hold data in transit

Picture source: https://computing.llnl.gov/tutorials/mpi/

Message Ordering Guarantee

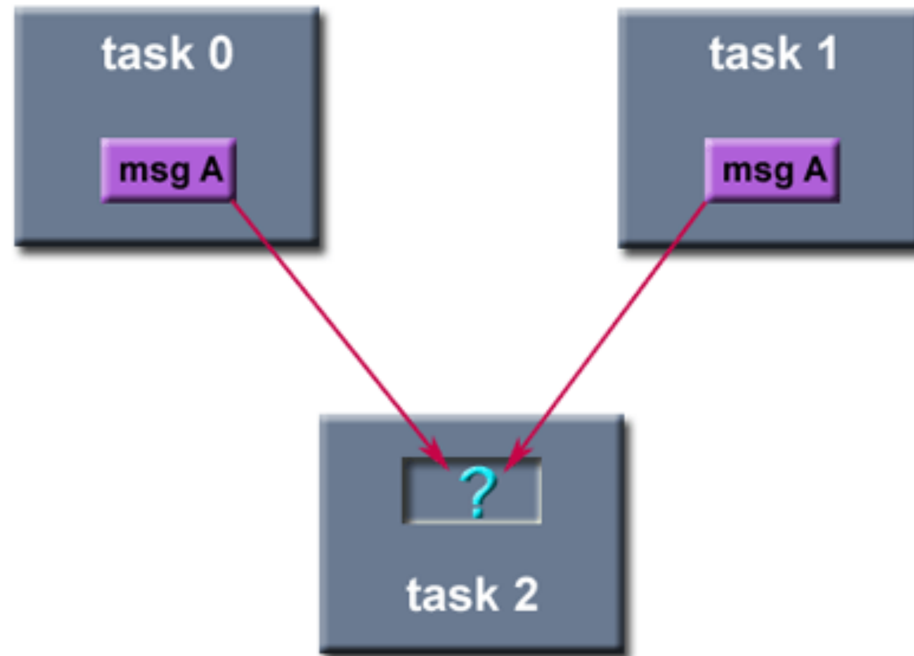
```
main(int argc, char **argv) {
  int rank, nproc;
  ............
  ............

  if(rank == 1) {
    for(int i=0; i<MAX; i++) {
      MPI_Send(&i, 1, MPI_INT, 0, tag, MPI_COMM_WORLD);
    }
  }
  else if(rank == 0) {
    int buffer[MAX];
    for(int i=0; i<MAX; i++) {
      MPI_Recv(&buffer[i], 1, MPI_INT, 1, tag, MPI_COMM_WORLD, &status);
      assert(buffer[i] == i); // Never fails
    }
  }
  ............
}
```

Code available on github: https://github.com/vivkumar/cse502/tree/master/mpi

- If a sender sends two messages (Msg_1 and Msg_2) in succession to same destination, and both match the same receive, the recv operation will always receive Msg_1 before Msg_2

© Vivek Kumar

No Guarantee for Fairness



- MPI does not guarantee fairness
- Example: task 0 sends a message to task 2. However, task 1 sends a competing message that matches task 2's receive. Only one of the sends will complete

Picture source: https://computing.llnl.gov/tutorials/mpi/

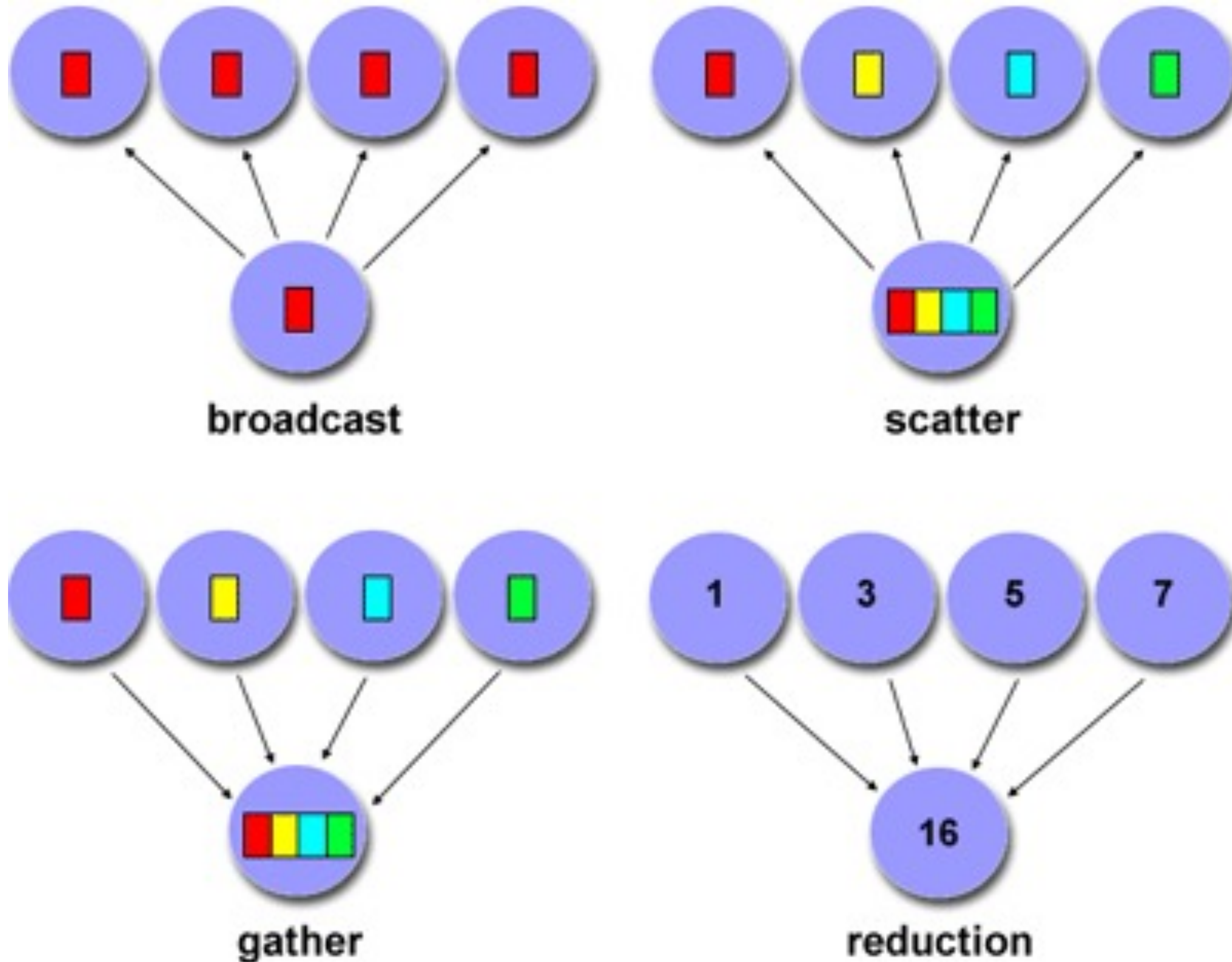
© Vivek Kumar

Non-Blocking Point-to-Point Communications

- MPI_Isend
- MPI_Irecv

1. These APIs returns immediately. They do not wait for any communication events to complete, such as message copying from user memory to system buffer space or the actual arrival of message
2. Provide opportunities to overlap computations and communications – unlike their blocking counterparts

Collective Communications in MPI

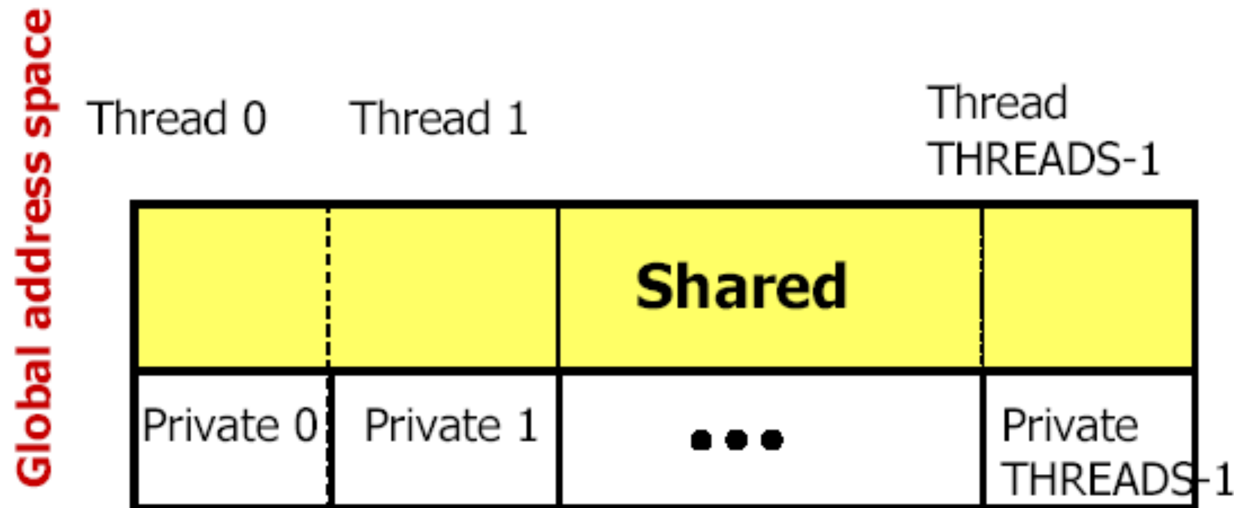


Picture source: https://computing.llnl.gov/tutorials/mpi/

PGAS Programming: General View

- A collection of threads operating in a partitioned global address space that is logically distributed among threads
- Each thread has affinity with a portion of the globally shared address space. Each thread has also a private space.
- Elements in partitioned global space belonging to a thread are said to have affinity to that thread.

UPC++ Memory Model



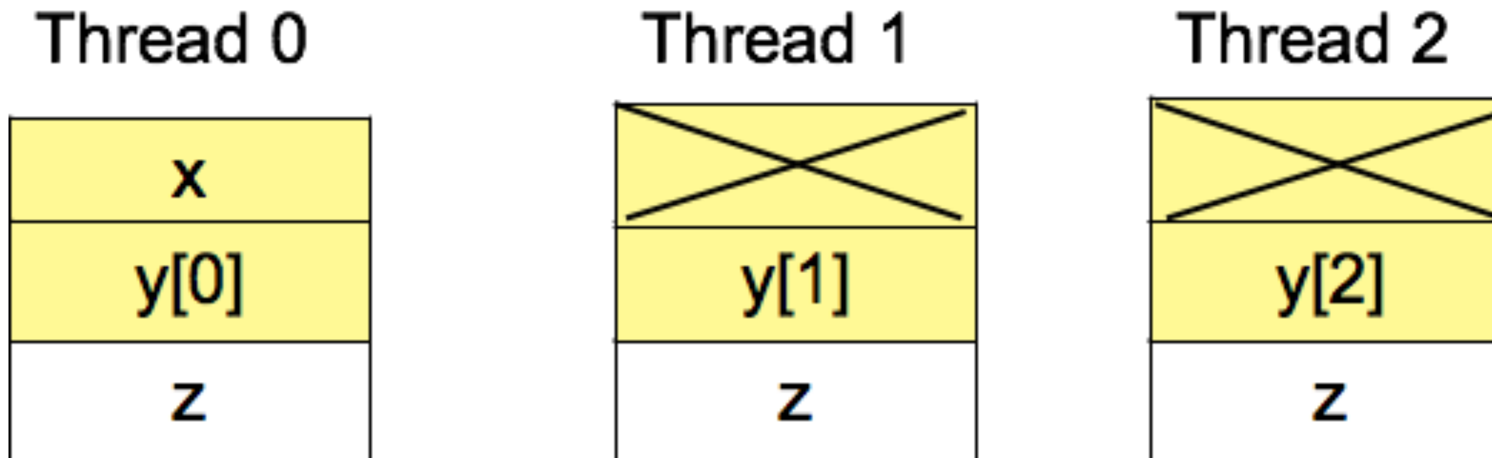
- A pointer-to-shared can reference all locations in the shared space
- A pointer-to-local ("plain old C pointer") may only reference addresses in its private space or addresses in its portion of the shared space
- Static and dynamic memory allocations are supported for both shared and private memory

Shared and Private Data (UPC++)

- Consider the following data layout directives

```
upcxx::shared_var<int> x; // x has affinity to thread 0
upcxx::shared_array<int> y(3);
int z;                    // private
```

- For THREADS = 3, we get the following layout



Next Two Lectures

- Student seminar
  - 10+2 min slot