Course Wrap-Up

Vivek Sarkar

Department of Computer Science
Rice University

vsarkar@cs.rice.edu

Schedule

- **In-class Final Exam on 4/22/08**
  - **Duration: 1 hour**
  - **Weightage: 20%**
  - **Three written questions to cover the following areas from second half of course**
    - **MPI (Lectures 16, 17, 18)**
    - **UPC (Lectures 21, 22)**
    - **Parallel Graph Algorithms (Lecture 24)**
- **REMINDER: Deadline for Assignment #4 is 4/30/08**

Lecture 16: Programming Using the Message Passing Paradigm (Chapter 6)

The minimal set of MPI routines.

`MPI_Init`	**Initializes MPI.**
`MPI_Finalize`	**Terminates MPI.**
`MPI_Comm_size`	**Determines the number of processes.**
`MPI_Comm_rank`	**Determines the label of calling process.**
`MPI_Send`	**Sends a message.**
`MPI_Recv`	**Receives a message.**

Avoiding Deadlocks

Consider the following piece of code, in which process i sends a message to process i + 1 (modulo the number of processes) and receives a message from process i - 1 (module the number of processes).

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
        MPI_COMM_WORLD);
MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
MPI_COMM_WORLD);
...
```

We have a deadlock because MPI_Send is blocking.

Avoiding Deadlocks

We can break the circular wait to avoid deadlocks as follows:

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank%2 == 1) {
        MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
                MPI_COMM_WORLD);
        MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
                MPI_COMM_WORLD);
}
else {
        MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
                MPI_COMM_WORLD);
        MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
                MPI_COMM_WORLD);
}
```

Another approach is to use the combined MPI_SendRecv operation

Non-Blocking Communications

- In order to overlap communication with computation, MPI provides a pair of functions for performing non-blocking send and receive operations ("I" stands for "Immediate"):

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype,
            int dest, int tag, MPI_Comm comm,
            MPI_Request *request)
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
            int source, int tag, MPI_Comm comm,
            MPI_Request *request)
```

- These operations return before the operations have been completed. `MPI_Wait` waits for the operation to complete.

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

- NOTE: Non-blocking send/receive can also be used to avoid deadlocks

Collective Communication Operations

- The barrier synchronization operation is performed in MPI using:

```
int MPI_Barrier(MPI_Comm comm)
```

  The one-to-all broadcast operation is:

```
int MPI_Bcast(void *buf, int count, MPI_Datatype datatype,
  int source, MPI_Comm comm)
```

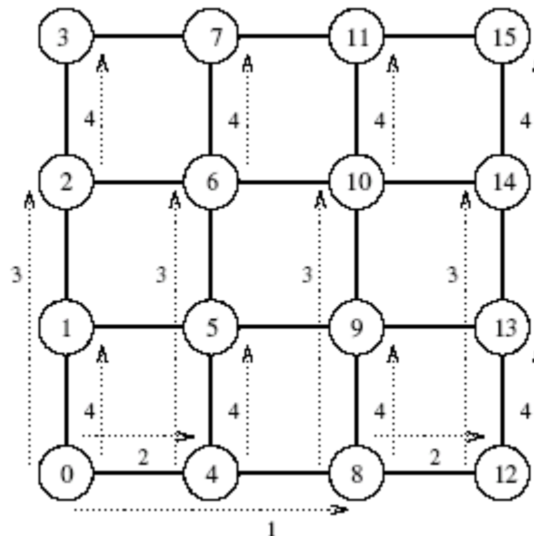
- The all-to-one reduction operation is:

```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count,
  MPI_Datatype datatype, MPI_Op op, int target,
MPI_Comm comm)
```

Lecture 17: Basic Communication Operations (Chapter 4)

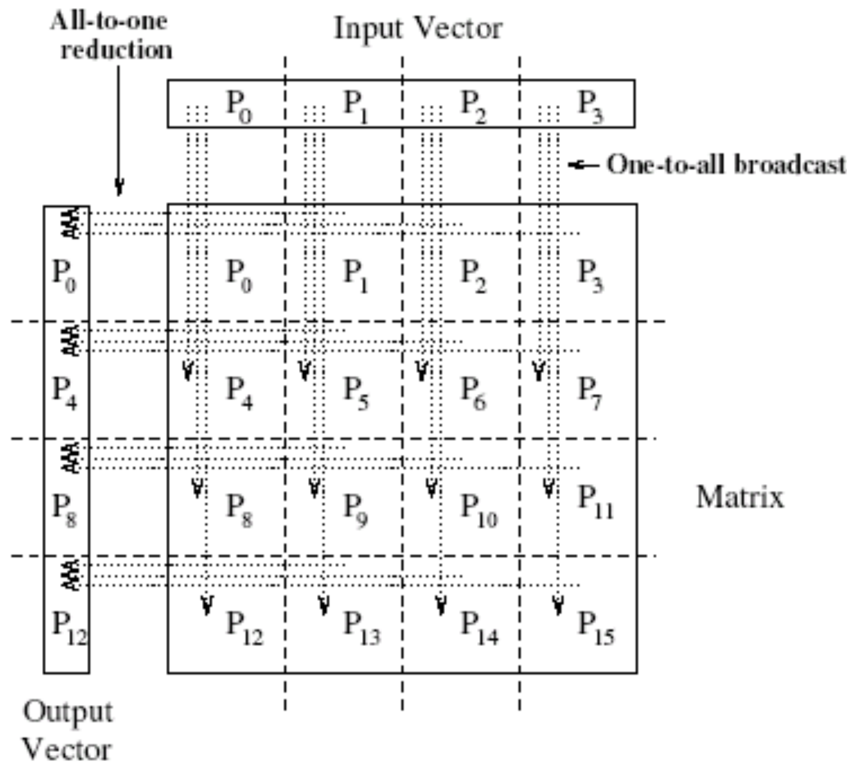
Broadcast and Reduction on a Mesh

- We can view each row and column of a square mesh of *p* nodes as a linear array of $\sqrt{p}$ nodes.
- Broadcast and reduction operations can be performed in two steps - the first step does the operation along a row and the second step along each column concurrently.
- Example: one-to-all broadcast on a 16-node mesh

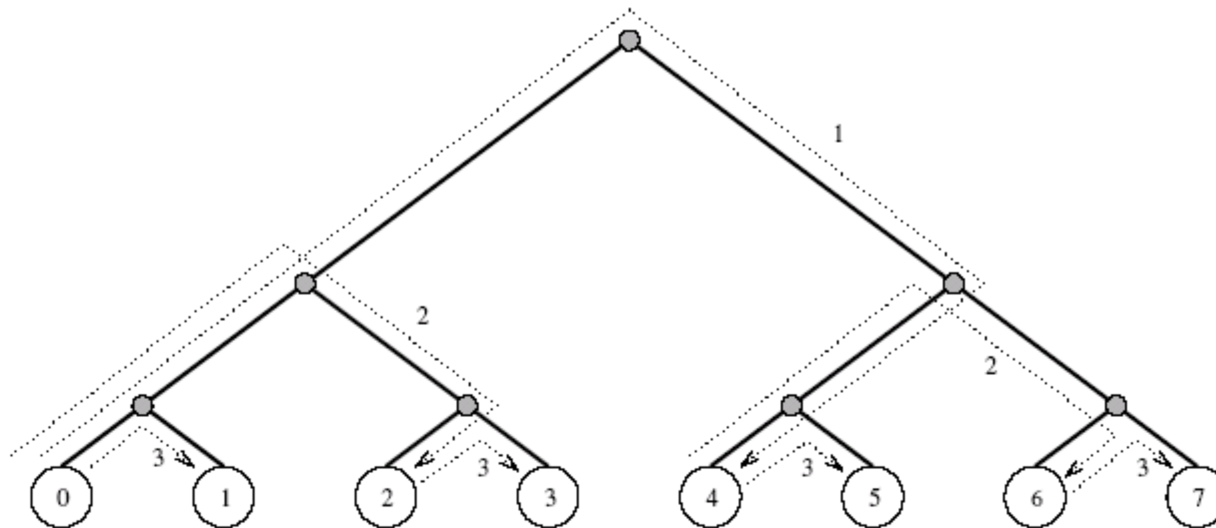


Broadcast and Reduction: Matrix-Vector Multiplication Example

One-to-all broadcast and all-to-one reduction in the multiplication of a *4 x 4* matrix with a *4 x 1* vector.



Broadcast and Reduction on a Balanced Binary Tree



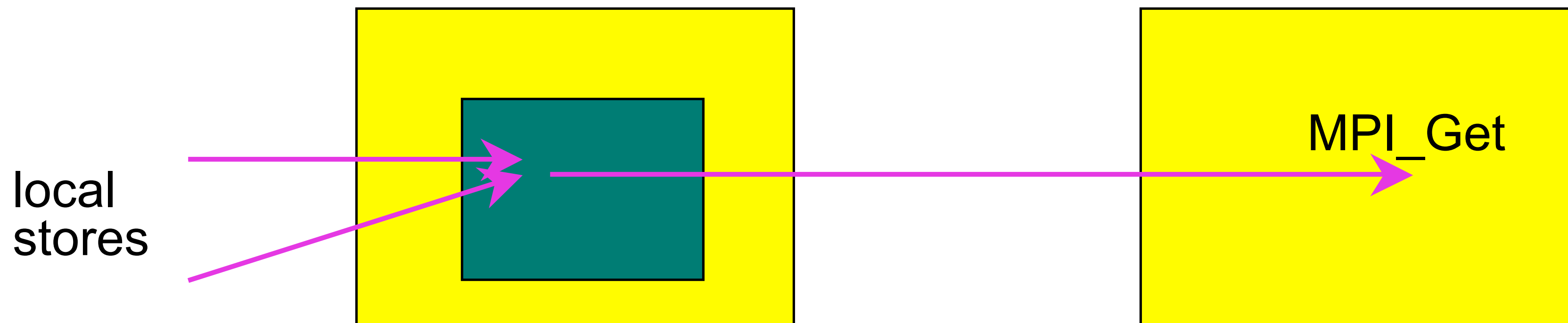
The Prefix-Sum Operation

- Given p numbers $n_0, n_1, \ldots, n_{p-1}$ (one on each node), the problem is to compute the sums $s_k = \sum_{i=0}^{k} n_i$ for all k between 0 and $p-1$.
- Initially, n_k resides on the node labeled k, and at the end of the procedure, the same node holds S_k.

Put, Get, and Accumulate

- ```
  MPI_Put(origin_addr, origin_count,
          origin_datatype,
          target_rank, target_offset,
          target_count, target_datatype,
          window)
  ```
- `MPI_Get( ... )`
- `MPI_Accumulate( ..., op, ... )`
- `op` **is as in** `MPI_Reduce`**, but no user-defined operations are allowed**
  ```

The Synchronization Issue



- Issue: Which value is retrieved?
  - Some form of synchronization is required between local load/stores and remote get/put/accumulates
- MPI provides multiple forms

Synchronization with Fence

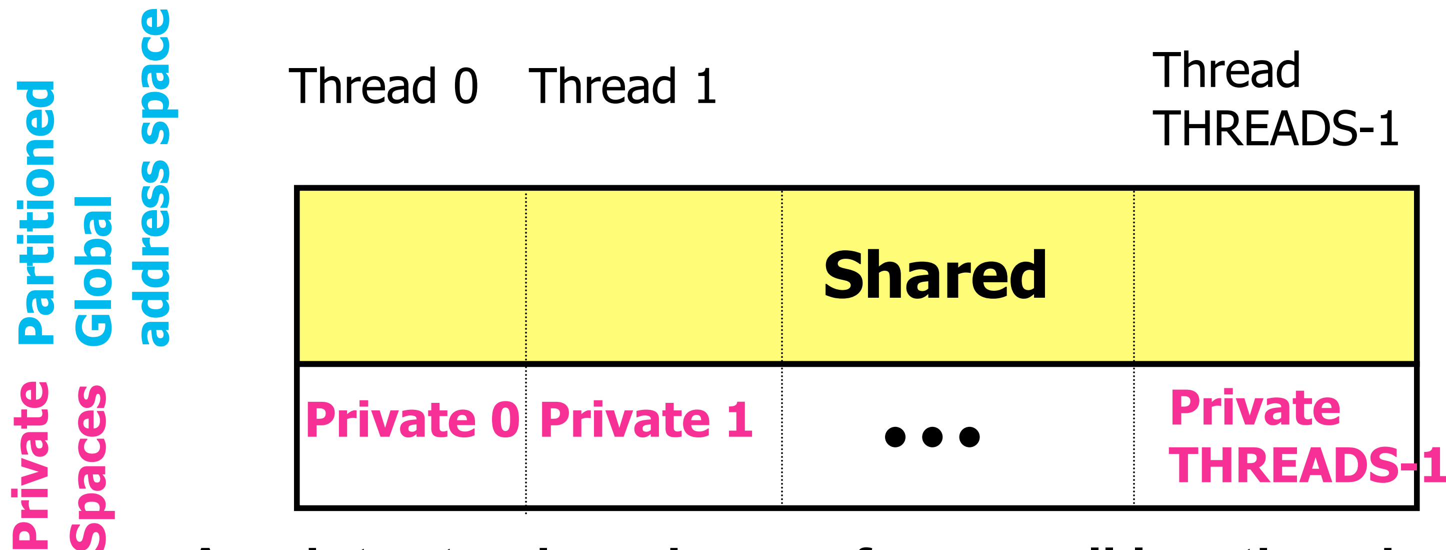
Simplest methods for synchronizing on window objects:

- `MPI_Win_fence` **- like OpenMP flush**

Process 0	Process 1
MPI_Win_fence(win)	MPI_Win_fence(win)
MPI_Put	
MPI_Put	
MPI_Win_fence(win)	MPI_Win_fence(win)

- **MPI_Win_fence is collective over the group of the window object**
- **MPI_Win_fence is used to *separate*, not just complete, RMA and local memory operations**

Lectures 21, 22: Unified Parallel C (UPC)



- **A pointer-to-shared can reference all locations in the shared space, but there is data-thread affinity**
- **A private pointer may reference addresses in its private space or its local portion of the shared space**
- **Static and dynamic memory allocations are supported for both shared and private memory**

Example of upc_forall w/ affinity clause

- upc_forall(init; test; loop; affinity)
  statement
- Affinity can be an int expression, or the address of a shared location

```
//vect_add.c

#include <upc_relaxed.h>
#define N 100*THREADS

shared int v1[N], v2[N], v1plusv2[N];

void main()
{
     int i;
     upc_forall(i=0; i<N; i++; i)
          v1plusv2[i]=v1[i]+v2[i];
}
```

Iteration #:	Thread 0	Thread 1
	0	1
	2	3

Shared Space

v1[0]	v1[1]
v1[2]	v1[3]

…

v2[0]	v2[1]
v2[2]	v2[3]

…

v1plusv2[0]	v1plusv2[1]
v1plusv2[2]	v1plusv2[3]

…

Block-Cyclic Distributions for Shared Arrays

- Default block size is 1
- Shared arrays can be distributed round robin on a block-per-thread basis (block-cyclic with default block size of 1)
- A block size is specified in the declaration as follows:
  - `shared [block-size] type array[N];`
  - e.g.: `shared [4] int a[16];`

Block size and THREADS determine affinity

Element i of a blocked array has affinity to thread:

$$\left\lfloor \frac{i}{blocksize} \right\rfloor \bmod THREADS$$

Shared and Private Data

Assume THREADS = 4

```
shared [3] int A[4][THREADS];
```

will result in the following data layout:

Thread 0	Thread 1	Thread 2	Thread 3
A[0][0]	A[0][3]	A[1][2]	A[2][1]
A[0][1]	A[1][0]	A[1][3]	A[2][2]
A[0][2]	A[1][1]	A[2][0]	A[2][3]
A[3][0]	A[3][3]		
A[3][1]			
A[3][2]			

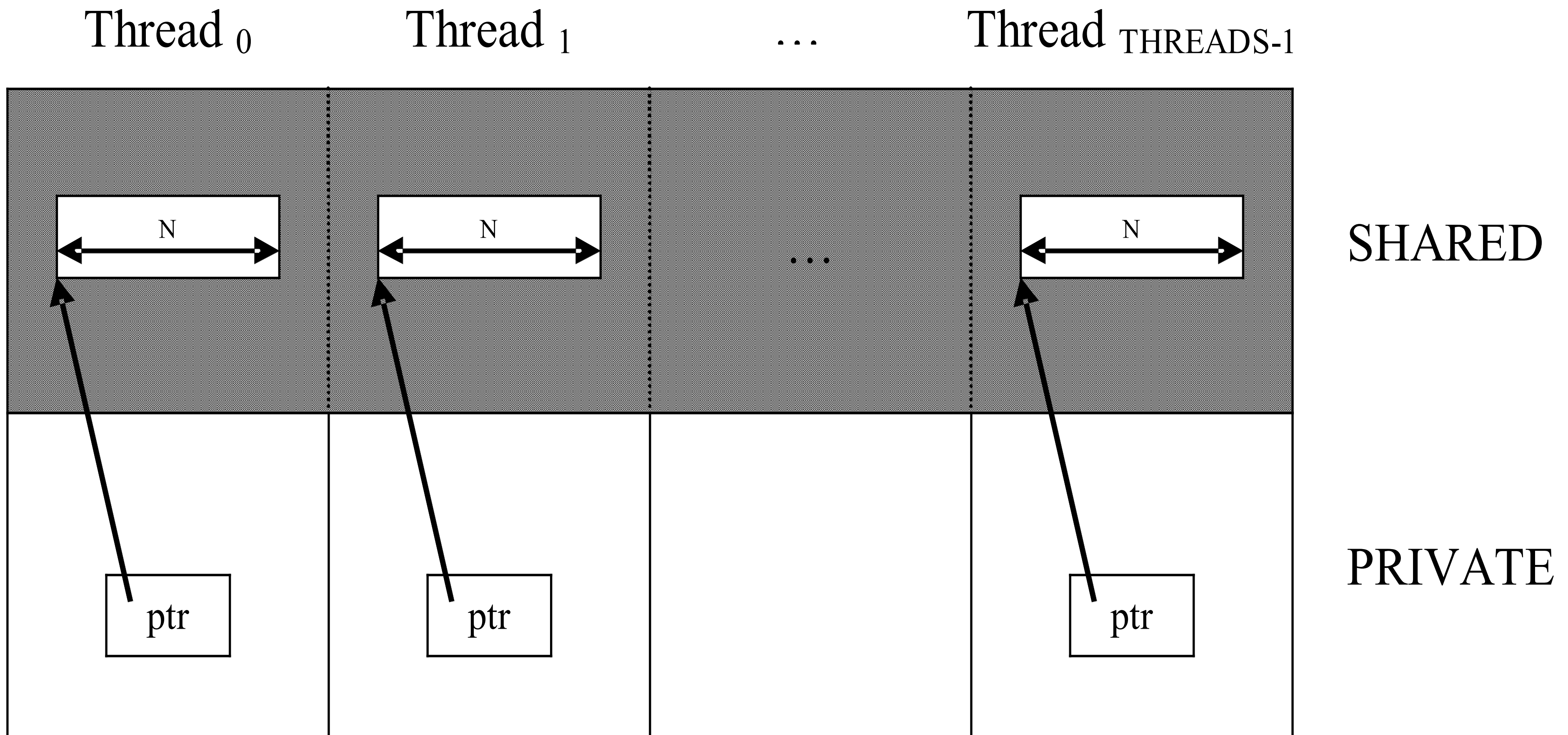
Local-Shared Memory Allocation

```
shared void *upc_alloc (size_t nbytes);
```

nbytes: block size

- **Non collective, expected to be called by one thread**
- **The calling thread allocates a contiguous memory region in the local-shared space of the calling thread**
- **Space allocated per calling thread is equivalent to :**
  `shared [] char[nbytes]`
- **If called by more than one thread, multiple regions are allocated and each calling thread gets a different pointer**

Local-Shared Memory Allocation Example




```
shared [] int *ptr;
ptr = (shared [] int *)upc_alloc(N*sizeof( int ));
```

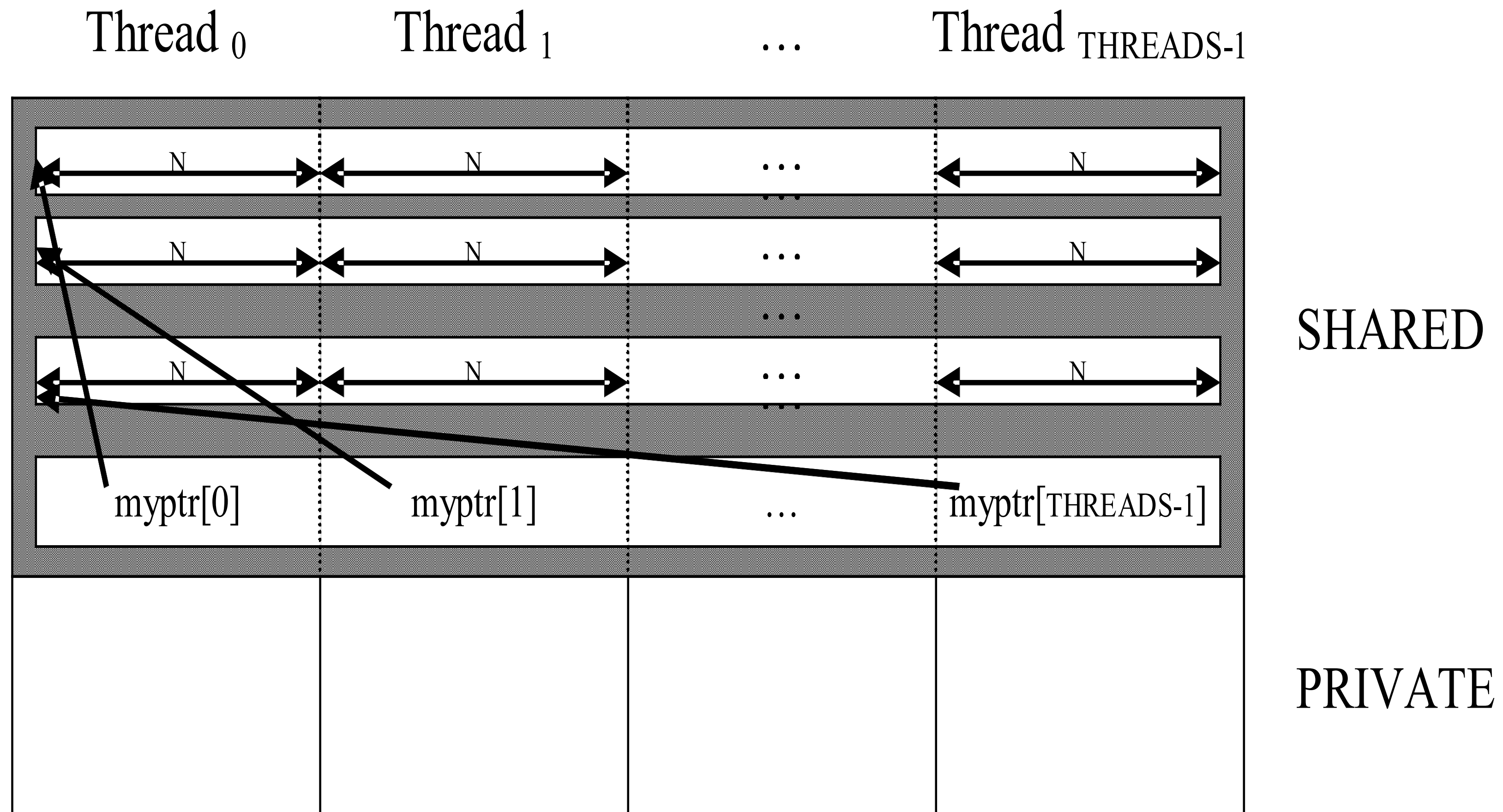
Global Memory Allocation

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

    nblocks : number of blocks
    nbytes : block size
```

- **Non collective, expected to be called by one thread**
- **The calling thread allocates a contiguous memory region in the shared space**
- **Space allocated per calling thread is equivalent to :**
  `shared [nbytes] char[nblocks * nbytes]`
- **If called by more than one thread, multiple regions are allocated and each calling thread gets a different pointer**

Global Memory Allocation Example




```
shared [N] int *shared myptr[THREADS];
myptr[MYTHREAD] = (shared [N] int *)
  upc_global_alloc( THREADS, N*sizeof( int ));
```

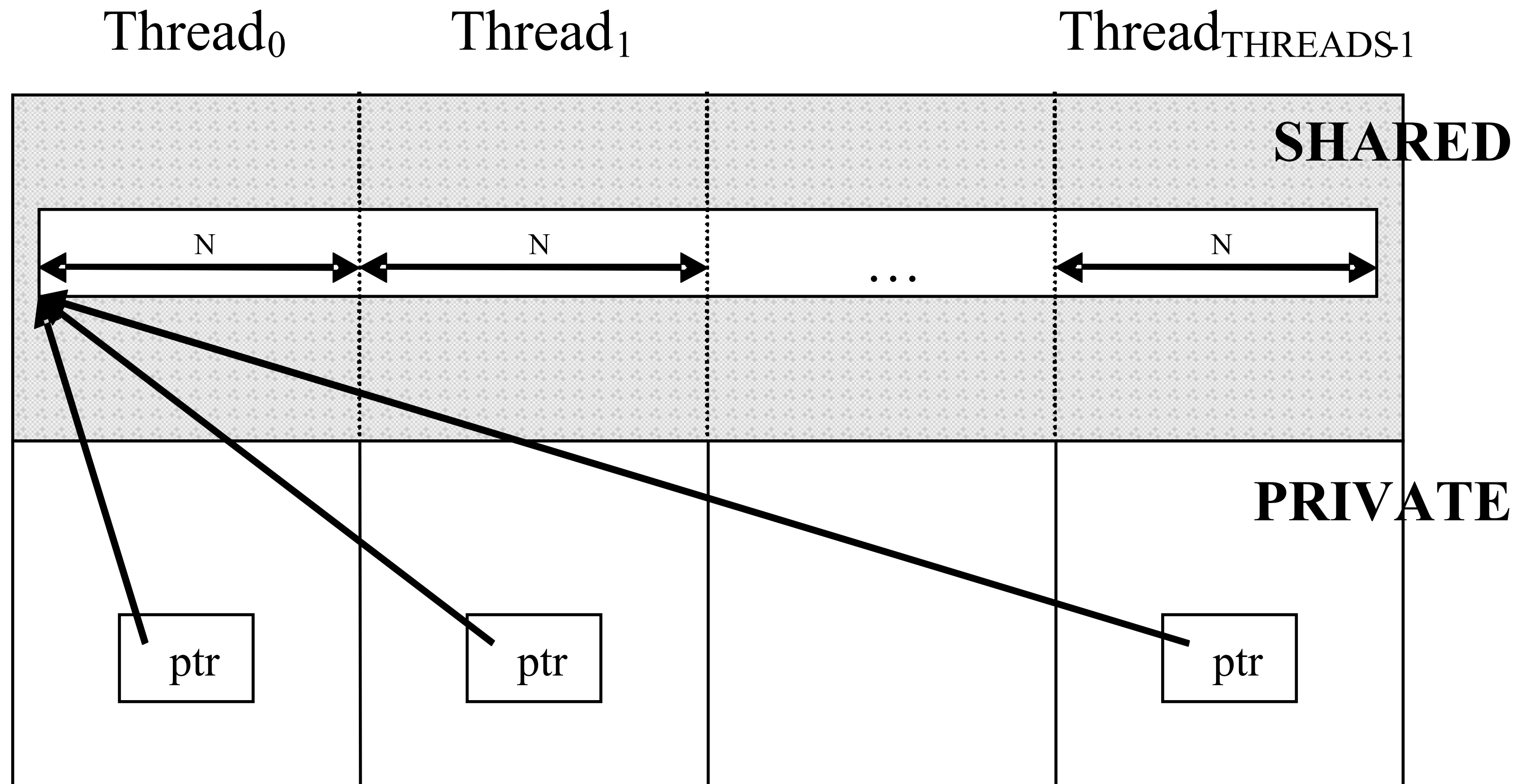
Collective Global Memory Allocation

```
shared void *upc_all_alloc
            (size_t nblocks, size_t nbytes);
```

nblocks: number of blocks
nbytes: block size

- **This function has the same result as** upc_global_alloc**. But this is a collective function, which is expected to be called by all threads**
- **nblocks and nbytes must be *single-valued* i.e., must have the same value on every thread. (The behavior of the operation is otherwise undefined.)**
- **All the threads will get the same pointer**
- **Equivalent to :**
  shared [nbytes] char[nblocks * nbytes]

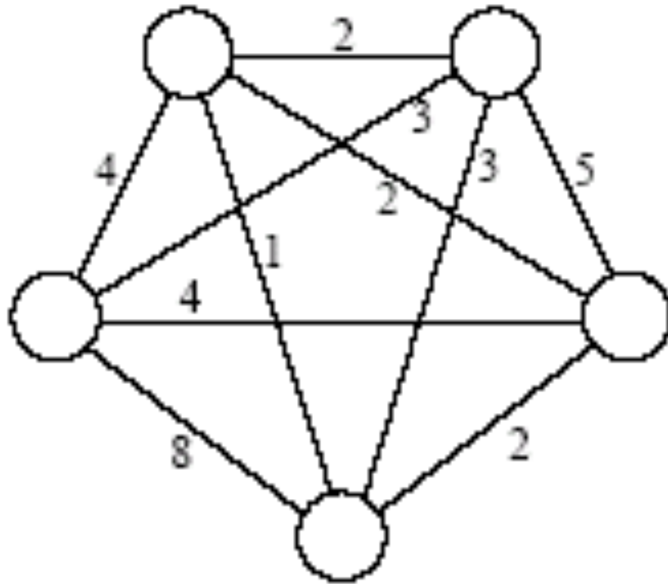
Collective Global Memory Allocation Example



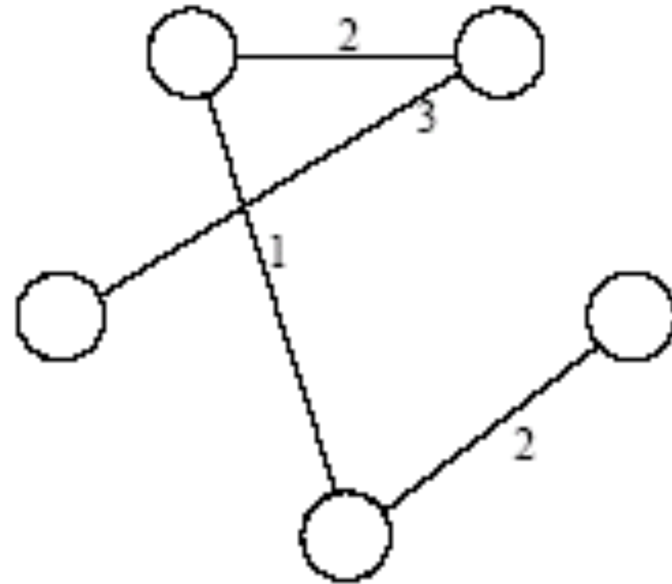

```
shared [N] int *ptr;
ptr = (shared [N] int *)
        upc_all_alloc( THREADS, N*sizeof( int ) );
```

Parallel Graph Algorithms (Lecture 24)

- **Minimum Spanning Tree**



Undirected graph



Minimum spanning tree

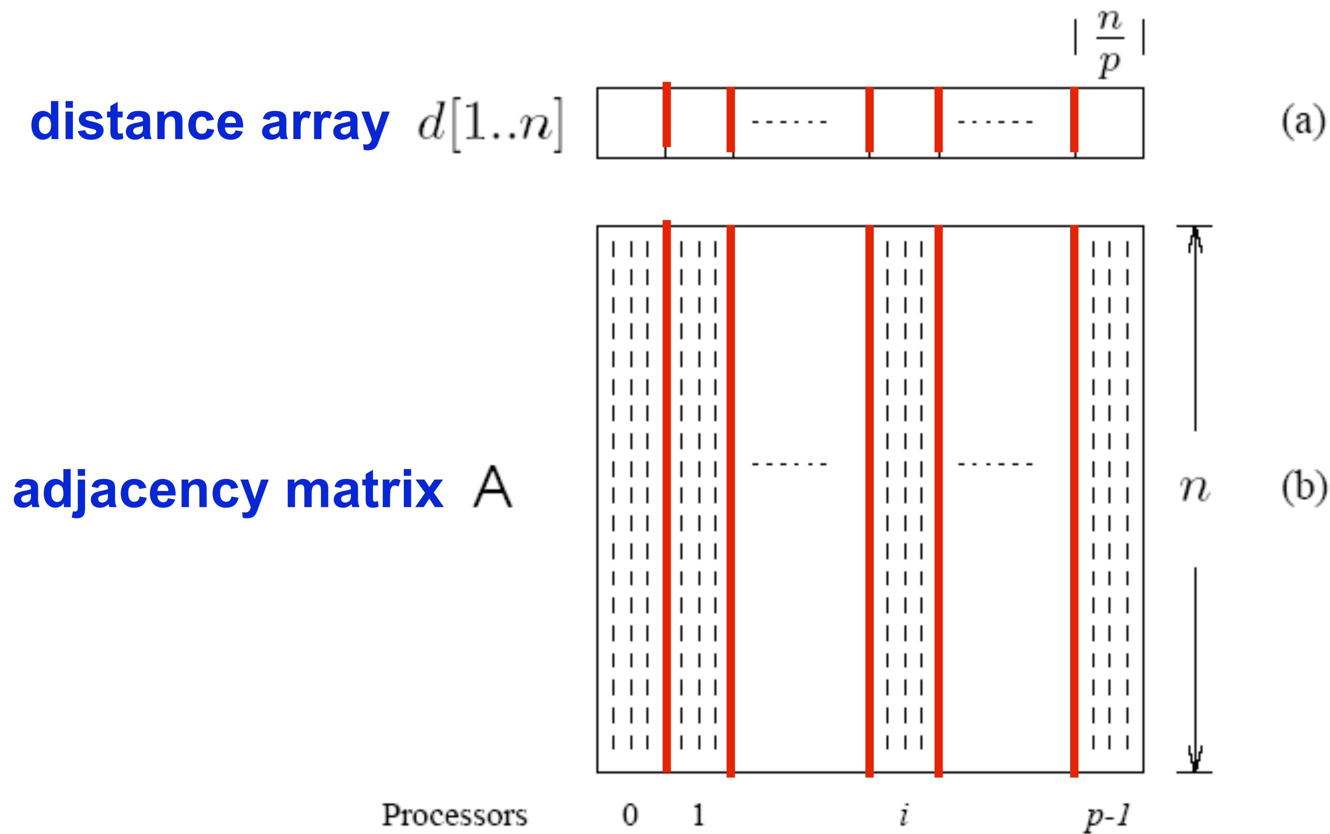
Computing a Minimum Spanning Tree

Prim's sequential algorithm

```
1.    procedure PRIM_MST(V, E, w, r)
2.    begin
3.        V_T := {r};  // initialize spanning tree vertices V_T with vertex r, the designated root
4.        d[r] := 0;                                            // compute d[:], the
5.        for all v ∈ (V − V_T) do                              // weight between
6.            if edge (r, v) exists set d[v] := w(r, v);        // r and each
7.            else set d[v] := ∞;                               // vertex outside V_T
8.        while V_T ≠ V do   // while there are vertices outside T
9.        begin
10.           find a vertex u such that d[u] := min{d[v] | v ∈ (V − V_T)};
11.           V_T := V_T ∪ {u};  // add u to T
12.           for all v ∈ (V − V_T) do                    // recompute d[:] now
13.               d[v] := min{d[v], w(u, v)};             // that u is in T
14.       endwhile
15.   end PRIM_MST
```

// use d[:] to find u,
// vertex closest to T

Parallel Formulation of Prim's Algorithm



partition *d* and *A* among *p* processes

Complexity Analysis of Prim's Algorithm

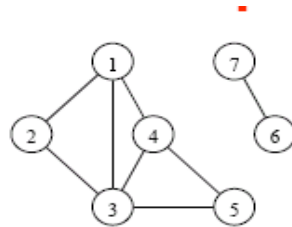
- **Cost to select the minimum entry**
  - — *$O(n/p)$: scan n/p local part of d vector on each processor*
  - —*$O(\log p)$* all-to-one reduction across processors
- **Broadcast next node selected for membership**
  - —*$O(\log p)$*
- **Cost of locally updating *d* vector**
  - — *$O(n/p)$: replace d vector with min of d vector and matrix row*
- **Parallel time per iteration**
  - —*$O(n/p + \log p)$*
- **Total parallel time**
  - —*$O(n^2/p + n \log p)$*

Connected Components Parallel Formulation

- **Partition the graph across processors**
- **Step 1**
  - —run independent connected component algorithms on each processor
  - —result: p spanning forests.
- **Step 2**
  - —merge spanning forests pairwise until only one remains

Connected Components Parallel Formulation

Graph *G*



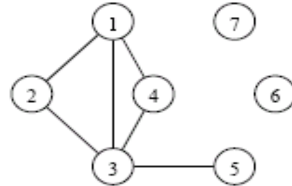
(a)

	1	2	3	4	5	6	7	
1	0	1	1	1	0	0	0	Processor 1
2	1	0	1	0	0	0	0	
3	1	1	0	1	1	0	0	
4	1	0	1	0	1	0	0	Processor 2
5	0	0	1	1	0	0	0	
6	0	0	0	0	0	0	1	
7	0	0	0	0	0	1	0	

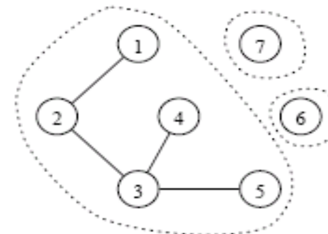
(b)

1. Partition adjacency matrix of the graph *G* into two parts

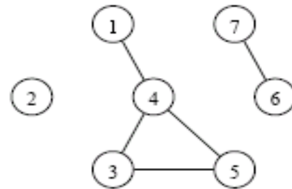
2. Each process gets a subgraph of graph *G*



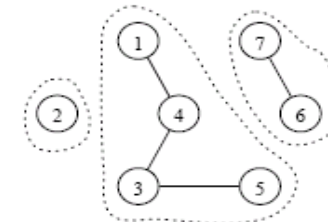
(c)



(d)



(e)



(f)

3. Each process computes the spanning forest of its subgraph of *G*

4. Merge the two spanning trees to form the final solution

Connected Components Parallel Formulation

- **Merge pairs of spanning forests using disjoint sets of vertices**
- **Consider the following operations on the disjoint sets**
  - —*find*(x)
    - **returns pointer to representative element of the set containing x**
    - **each set has its own unique representative**
  - —*union*(x, y)
    - **merges the sets containing the elements x and y**
    - **the sets are assumed disjoint prior to the operation**

Connected Components Parallel Formulation

- **To merge forest *A* into forest *B***
  - **for each edge *(u,v)* of *A*,**
    - **perform *find* operations on u and v**
      - **determine if u and v are in same tree of *B***
    - **if not, then union the two trees (sets) of *B* containing *u* and *v***
      - **result: u and v are in same set, which means they are connected**
    - **else , no *union* operation is necessary.**
- **Merging forest *A* and forest *B* requires at most**
  - *2(n-1)* ***find* operations**
  - *(n-1)* ***union* operations**

at most n-1 edges must be considered because A and B are forests

Connected Components

Analysis of parallel 1-D block mapping

- **Partition an *n x n* adjacency matrix into *p* blocks**
- **Each processor computes local spanning forest: *Θ(n^2/p)***
- **Merging approach**
  - —embed a logical tree into the topology
    - – *log p* merging stages
    - – each merge stage takes time *Θ(n)*
  - —total cost due to merging is *Θ(n log p)*
- **During each merging stage**
  - —spanning forests are sent between nearest neighbors
  - —*Θ(n)* edges of the spanning forest are transmitted
- **Parallel execution time**

$$T_P = \overbrace{\Theta\left(\frac{n^2}{p}\right)}^{\text{local computation}} + \overbrace{\Theta(n \log p)}^{\text{forest merging}}.$$

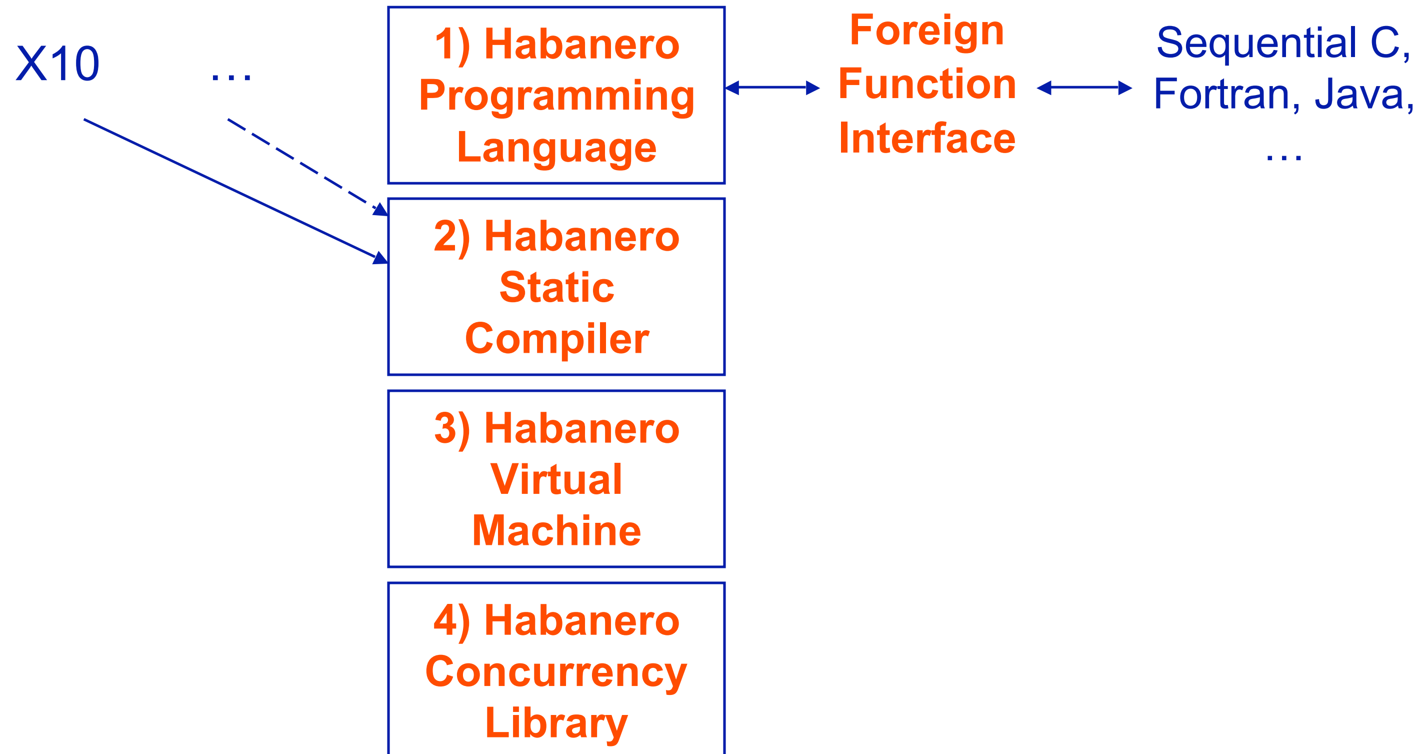
Course Topics

- Introduction (Chapter 1)
- Parallel Programming Platforms (Chapter 2)
  — New material: homogeneous & heterogeneous multicore platforms
- Principles of Parallel Algorithm Design (Chapter 3)
- Programming Shared Address Space Platforms (Chapter 7)
  — New material (beyond threads and OpenMP): Java Concurrency Utilities, Intel Thread Building Blocks, .Net Parallel Extensions (Task Parallel Library & PLINQ), Cilk, X10
- Analytical Modeling of Parallel Programs (Chapter 5)
  — New material: theoretical foundations of task scheduling
- Dense Matrix Operations (Chapter 8)
- Graph Algorithms (Chapter 10)
- Programming Using the Message-Passing Paradigm (Chapter 6)
  — New material (beyond MPI): Unified Parallel C (UPC)
- New material: Problem Solving on Large Scale Clusters using MapReduce
- Guest lectures: High Performance Fortran (HPF), Mutual Exclusion with Locks and Atomic Operations, Co-array Fortran (CAF)

Parallel Software Challenges

Layer	Challenges
Domain-specific Programming Models	Domain-specific implicitly parallel programming models e.g., Matlab, stream processing, map-reduce (Sawzall),
Middleware	Parallelism in middleware e.g., transactions, relational databases, web services, J2EE containers
Application Libraries	Parallel application libraries e.g., linear algebra, graphics imaging, signal processing, security
Programming Tools	Parallel Debugging and Performance Tools e.g., Eclipse Parallel Tools Platform, TotalView, Thread Checker
Languages	Explicitly parallel languages e.g., OpenMP, Java Concurrency, .NET Parallel Extensions, Intel TBB, CUDA, Cilk, MPI, Unified Parallel C, Co-Array Fortran, X10, Chapel, Fortress
Static & Dynamic Optimizing Compilers	Parallel intermediate representation, optimization of synchronization & data transfer, automatic parallelization
Multicore Back-ends	Code partitioning for accelerators, data transfer optimizations, SIMDization, space-time scheduling, power management
Parallel Runtime & System Libraries	Parallel runtime and system libraries for task scheduling, synchronization, parallel data structures
OS and Hypervisors	Virtualization, scalable management of heterogeneous resources per core (frequency, power)

Habanero Project (habanero.rice.edu)



Opportunities for Broader Impact

- **Education**
  - —Influence how parallelism is taught in future Computer Science curricula
- **Open Source**
  - —Build an open source testbed to grow ecosystem for researchers in Parallel Software area
- **Industry standards**
  - —Use research results as proofs of concept for new features that can be standardized
  - —Infrastructure can provide foundation for reference implementations

Send email to vsarkar@rice.edu if you are interested in a PhD, postdoc, research scientist, or programmer position in the Habanero project, or in collaborating in our research!