COL380

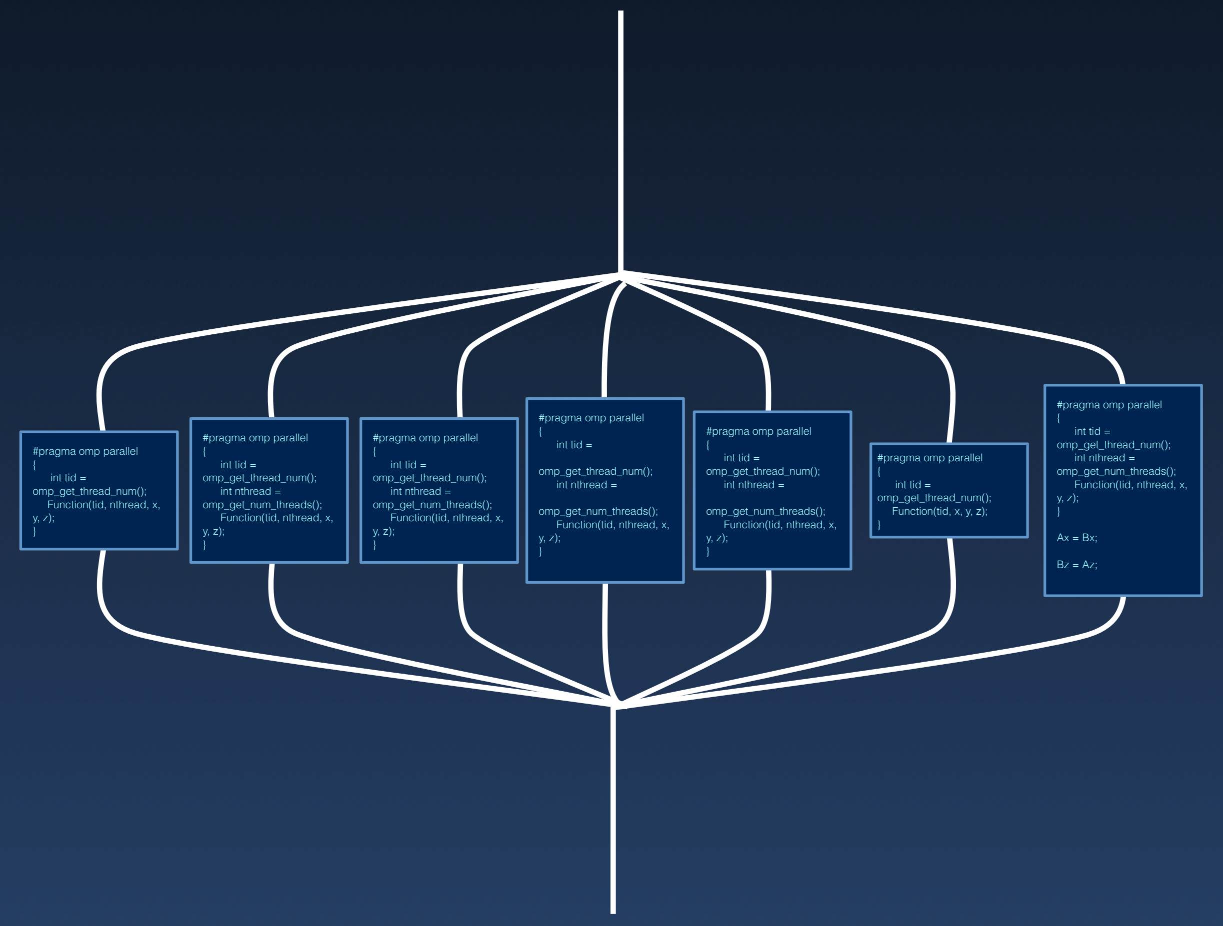
Introduction to
Parallel & Distributed Programming

Agenda

- Introduction to Message Passing
  - ➡ OpenMP review

OpenMP Review

- Fork-Join programming model
  - ➡ Teams of threads
  - ➡ Implicit barriers
- Shared Memory programming model
- Well defined memory model
- High level synchronization



```
#pragma omp parallel
{
    int tid = omp_get_thread_num();
    int nthread = omp_get_num_threads();
    Function(tid, nthread, x, y, z);
}
```

Subodh Kumar

More OpenMP

- Shared and private variables
  - ➡ Temporary view of shared memory
  - ➡ Option to reduce private variables
- Work sharing
  - ➡ Section, for, simd

```
#pragma omp parallel for simd
for(i = 0; i < num; i++) {
    c[i] += a[i] * b[i];
}
```

See: nowait, ordered

```
int sum;
#pragma omp parallel reduction(+:sum)
sum = partial_sum(data, omp_get_thread_num());
```

```
#pragma omp target map(tofrom:c[0:num])
for (int i = 0; i < num; i++) {
    c[i] += a[i] * b[i];
}
```

Subodh Kumar

Efficiency Issues

- Minimize synchronization
  - ➡ Avoid BARRIER, CRITICAL, ORDERED, and locks
  - ➡ Use NOWAIT
  - ➡ Use named CRITICAL sections for fine-grained locking
  - ➡ Use MASTER (instead of SINGLE)
- Parallelize at the highest level possible
  - ➡ such as outer FOR loops
  - ➡ keep parallel regions/tasks large

Efficiency Issues

- FLUSH is expensive
- LASTPRIVATE has synchronization overhead
- Thread safe malloc/free are expensive
- Reduce False sharing
  - ➡ Careful design of data structures
  - ➡ Use PRIVATE

Avoiding Errors

- Try to
  - Avoid nested locks
  - Release locks religiously
  - Avoid “while true” (especially, during testing)
- Be careful with
  - Non thread-safe libraries
  - Concurrent access to shared data
  - IO inside parallel regions
  - Differing views of shared memory (FLUSH)
  - NOWAIT

Programming Models

- Shared Memory model
- Distributed Memory model
- Task based model
- Work-queue model
- Stream processing model
- Map-reduce model
- Client-server model

Other Models

pthread

```
void *threadFn(void *argp)
{
    // Do something with argp
}
…
{
    int arg;
    pthread_t thread_id;;
    pthread_create(&thread_id, NULL, threadFn, &arg);
    // Continue doing other things
    pthread_join(thread_id, NULL);
}
```

Other Models

go

```
func threadFn(id int, arg string, wg *sync.WaitGroup)
{
    defer wg.Done()
    // Do thing 'id' with 'arg'

}
…
{
    var wg sync.WaitGroup

    wg.Add(1)
    go threadFn(1, "a string", &wg)
    // Continue doing other things
    wg.Wait()

}
```

pthread

```
void *threadFn(void *argp)
{
    // Do something with argp
}
…
{
    int arg;
    pthread_t thread_id;
    pthread_create(&thread_id, NULL, thr
    // Continue doing other things
    pthread_join(thread_id, NULL);
}
```

Other Models

pthread

```
void *threadFn(void *argp)
{
    // Do something with argp
}
…
{
    int arg;
    pthread_t thread_id;;
    pthread_create(&thread_id, NULL, thr
    // Continue doing other things
    pthread_join(thread_id, NULL);
}
```

go

```
func threadFn(id int, arg string, wg *sync.WaitGroup,
                                              ch chan int)
{
    defer wg.Done()
    // Do thing ‘id’ with ‘arg’
                        x <- ch
}
…
{
    var wg sync.WaitGroup
                        chA := make(chan int)
    wg.Add(1)
    go threadFn(1, “a string”, &wg, chA)
    // Continue doing other things
    wg.Wait()
                        chA <- result
}
```

Subodh Kumar

Other Models

chapel

```
var loc = 0;
sync {
    while((taski = taskQ.dequeue()) != nil) { // Next queue item
        begin { on Locales[loc] workOnTask(taski); } // Non-local task
        loc = (loc + 1)%numLocales
    }
} // sync ⇒ implicit join with children tasks
```

Subodh Kumar

Other Models

chapel

```
var loc = 0;
sync {
    while((taski = taskQ.dequeue()) != nil) { // Next queue item
        begin { on Locales[loc] workOnTask(taski); } // Non-local task
        loc = (loc + 1)%numLocales
    }
```

```
const D1 = {0..#n} dmapped BlockCyclic(startIdx=(0), blocksize=(8));
var distA: [D1] int; // Distributed distA as
on Locales[1] { // Executes on Locales[1]
    var second = 2; // variable is native to Locales[1]
    coforall loc in Locales { // Create concurrent tasks, 1 per iteration
        on loc { // On Locales[loc]
            var local = distA[0] + distA[here.id*8] + second; // Fetch non-local
        }
    } // Implicit join with children tasks
}
```

Message Passing

- Communication network (and infrastructure)
  - ➡ Ethernet, Infiniband, Custom-made
- Processor-local memory
- Access to other threads' data through explicit instructions
  - ➡ Implicit synchronization semantics
- Can double as inter-process synchronization

Shared Memory

<u>Multiple Threads of Execution</u>

OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands

Shared Variable Store

(May not share actual memory)

- How are they instantiated?
- Execution and memory model
- How do the interact?

Separate from HW
(Who does the execution?)

Subodh Kumar

Message Passing

<u>Multiple Threads of Execution</u>

OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands

- How are they instantiated?
- Execution and memory model
- How do the interact?

Separate from HW
(Who does the execution?)

Message Passing

Multiple Threads of Execution

OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
Memory	Memory

(Could share actual memory)

- How are they instantiated?
- Execution and memory model
- How do the interact?

Separate from HW
(Who does the execution?)

Message Passing

<u>Multiple Threads of Execution</u>

OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
Memory	Memory

(Could share actual memory)

- How are they instantiated?
- Execution and memory model
- How do the interact?

Separate from HW
(Who does the execution?)

Message Passing

<u>Multiple Threads of Execution</u>

OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
Memory	Memory

(Could share actual memory)

- How are they instantiated?
- Execution and memory model
- How do the interact?

Separate from HW
(Who does the execution?)

Message Passing

Multiple Threads of Execution

OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
Memory	Memory

(Could share actual memory)

- How are they instantiated?
- Execution and memory model
- How do the interact?

Separate from HW
(Who does the execution?)

Message Passing

Multiple Threads of Execution

OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
Memory	Memory

(Could share actual memory)

- How are they instantiated?
- Execution and memory model
- How do the interact?

Separate from HW
(Who does the execution?)

Multiple Threads of Execution

OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
Memory	Memory

(Could share actual memory)

- How are they instantiated?
- Execution and memory model
- How do the interact?

Separate from HW
(Who does the execution?)

- Multiple "threads" of execution
  - Do not share address space (Processes)
  - Each process may further have multiple threads of control that share memory with each other

- Multiple “threads” of execution
    - Do not share address space (Processes)
    - Each process may further have multiple threads of control that share memory with each other

Shared Memory Model

Read Input
Create Sharing threads:
Process(sharedInput, myID)

Message Passing

- Multiple “threads” of execution
  - Do not share address space (Processes)
  - Each process may further have multiple threads of control that share memory with each other

<u>Shared Memory Model</u>

Read Input
Create Sharing threads:
Process(sharedInput, myID)

<u>Message Passing Model</u>

Read Input
Create Remote Processes
Loop: Send data to each process
Wait and collect results

- Multiple "threads" of execution
  - Do not share address space (Processes)
  - Each process may further have multiple threads of control that share memory with each other

<u>Shared Memory Model</u>

Read Input
Create Sharing threads:
Process(sharedInput, myID)

<u>Message Passing Model</u>

Read Input
Create Remote Processes
Loop: Send data to each process
Wait and collect results

: Recv data
Process(data)
Send results

+/- of Shared Memory

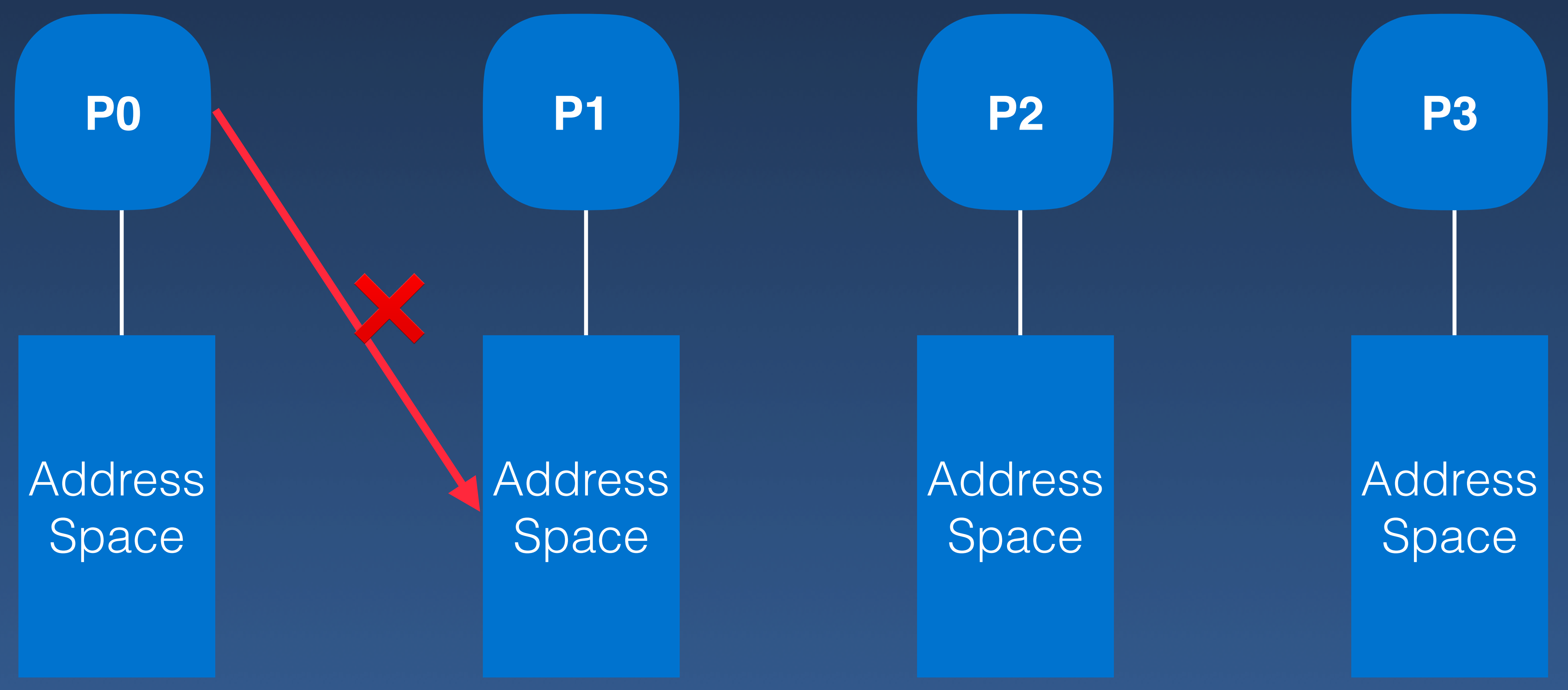
- \+ Easier to program with global address space
- \+ Typically fast memory access
  - ➡ (when hardware supported)
- \- Hard to scale
  - \- Adding CPUs (geometrically) increases traffic
- \- Programmer initiated synchronization of memory accesses

+/- of Message Passing

+ Memory is scalable with number of processors
+ Local access is fast (no cache coherency overhead)
+ Cost effective, with off-the-shelf processor/ network

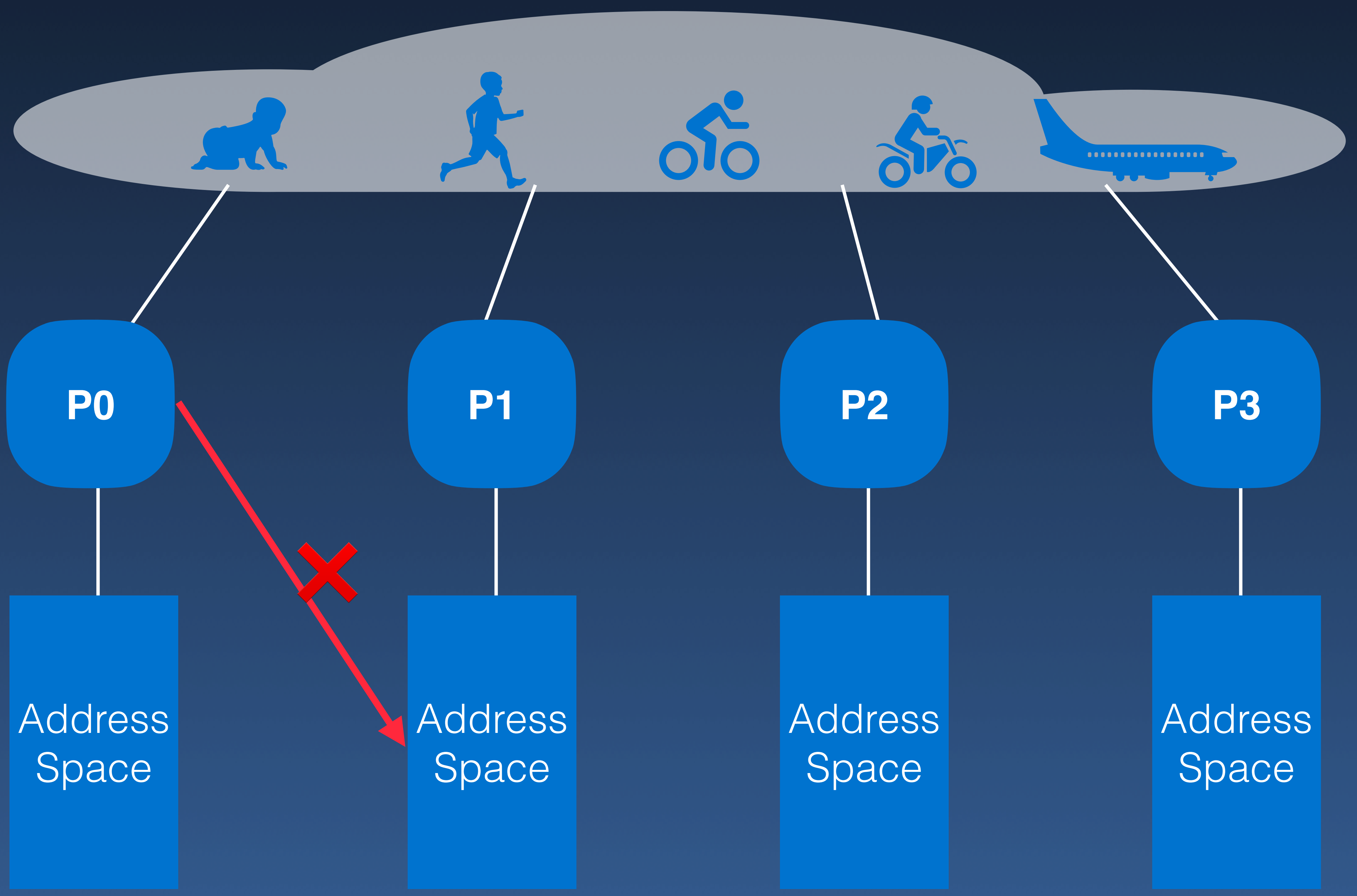
- Programs often looks more complex
- Data communication needs to be managed

Distributed Memory

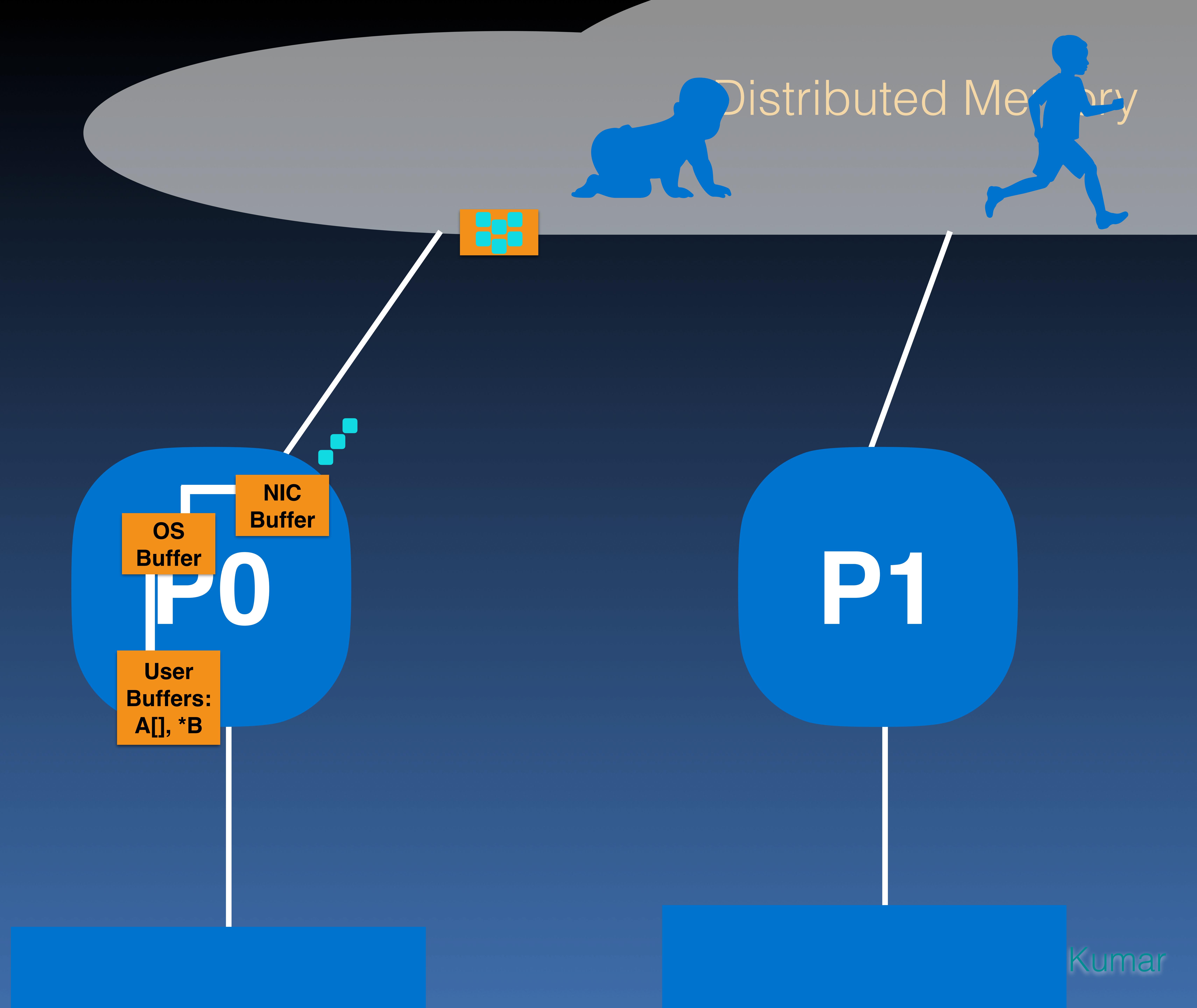


Subodh Kumar

Distributed Memory



Subodh Kumar


Distributed Me ry
NIC
Buffer
OS
Buffer
P0
User
Buffers:
A[], *B
P1
Kumar


istributed Memory
NIC
Buffer
OS
Buffer
P0
User
Buffers:
A[], *B
NIC
Buffer
OS
Buffer
P1
User
Buffers:
A[], *B
P2
Subo


istributed Memory
NIC Buffer
OS Buffer
P0
User Buffers: A[], *B
NIC Buffer
OS Buffer
P1
User Buffers: A[], *B
P2
RDMA solutions also exist
Subo

- MPI is for inter-process communication
  - ➡ Process creation
  - ➡ Data communication (Buffering, Book-keeping ..)
  - ➡ Synchronization
- Allows
  - ➡ Synchronous communication
  - ➡ Asynchronous communication
    - compare to shared memory

- MPI is for inter-process communication
  - ➡ Process creation
  - ➡ Data communication (Buffering, Book-keeping ..)
  - ➡ Synchronization

Functions, Types, Constants

- Allows
  - ➡ Synchronous communication
  - ➡ Asynchronous communication
    - compare to shared memory

- High-level constructs
  - broadcast, reduce, scatter/gather message
  - Collective functions
- Interoperable across architectures

Running MPI Programs

- **Compile:** mpiCC -o exec code.cpp
  - script to compile and link
  - Automatically adds include, library flags
- **Run:**
  - mpirun -host host1,host2 exec args
  - Or, use hostfile
- **Useful:**
  - mpirun -mca <key> <value>

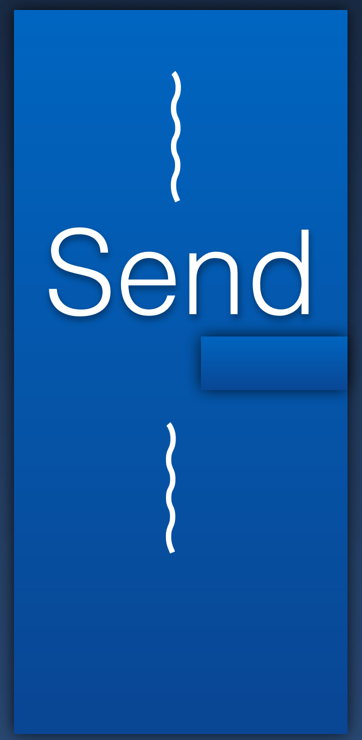
- mpirun -mca mpi_show_handle_leaks 1
- mpirun -mca btl openib,tcp
- mpirun -mca btl_tcp_min_rdma_size
- Check out "ompi_info"

- Key based remote shell execution
- Use ssh-keygen to create public-private key pair
  - ➡ Private key stays in subdirectory ~/.ssh on your client
  - ➡ Public key on server in ~/.ssh/authorized_keys
  - ➡ Test: ‘ssh <server> ls’ works
  - ➡ On hpc/css, client and server share the same home directory
- PBS automatically creates appropriate host files
  - ➡ See also: -l select=2:ncpus=1:mpiprocs=1 -l place=scatter

- Variables, Buffers, and Packets
  - ➡ Application to application
- Lossy?
  - ➡ Deal with loss
  - ➡ Acks
- FIFO?
- Point to Point vs Collective?
- Addressing?

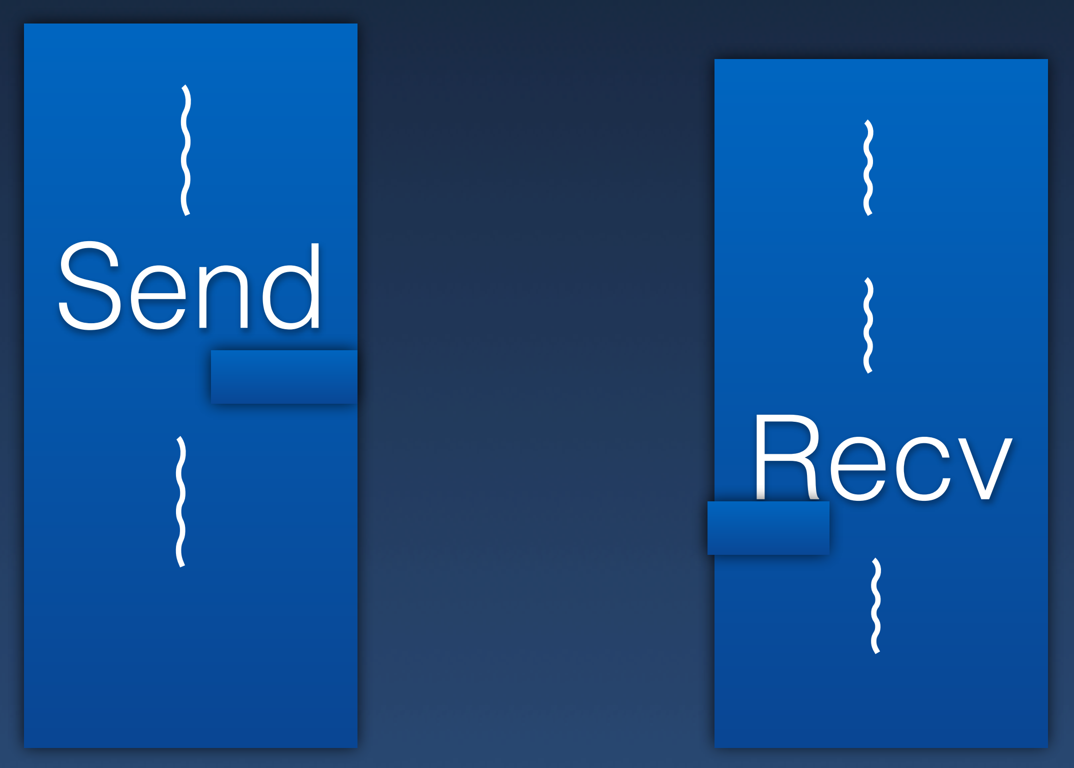
Message Semantics

- Variables, Buffers, and Packets
  - ➡ Application to application
- Lossy?
  - ➡ Deal with loss
  - ➡ Acks
- FIFO?
- Point to Point vs Collective?
- Addressing?



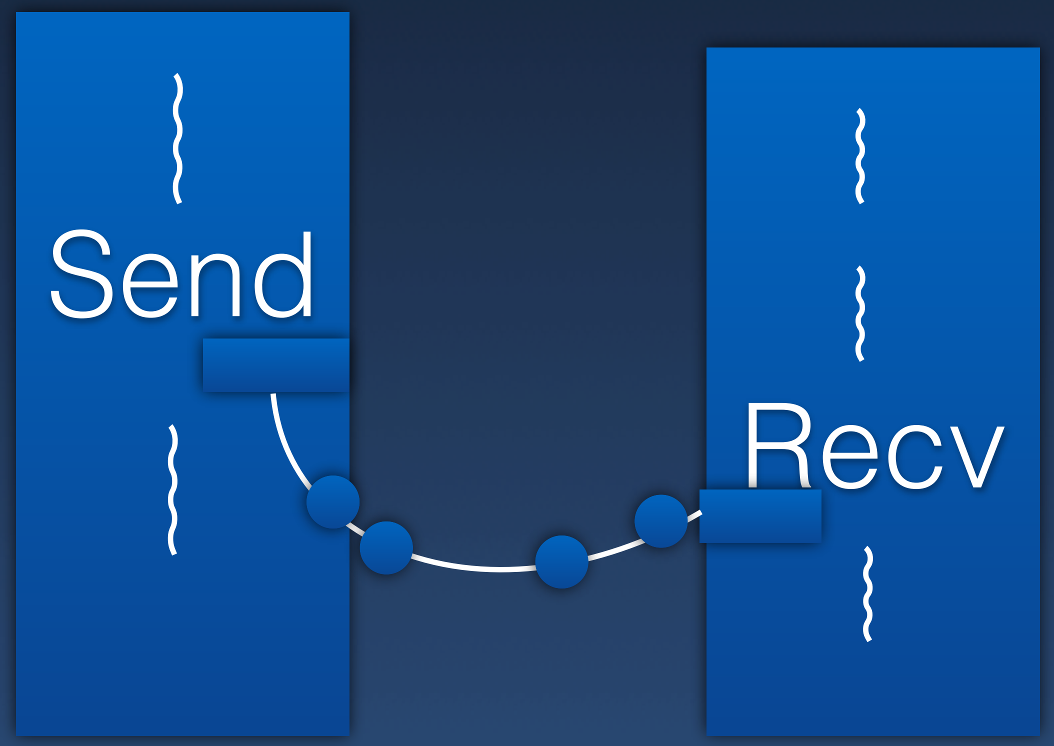
Subodh Kumar

- Variables, Buffers, and Packets
    - ➡ Application to application
- Lossy?
    - ➡ Deal with loss
    - ➡ Acks
- FIFO?
- Point to Point vs Collective?
- Addressing?

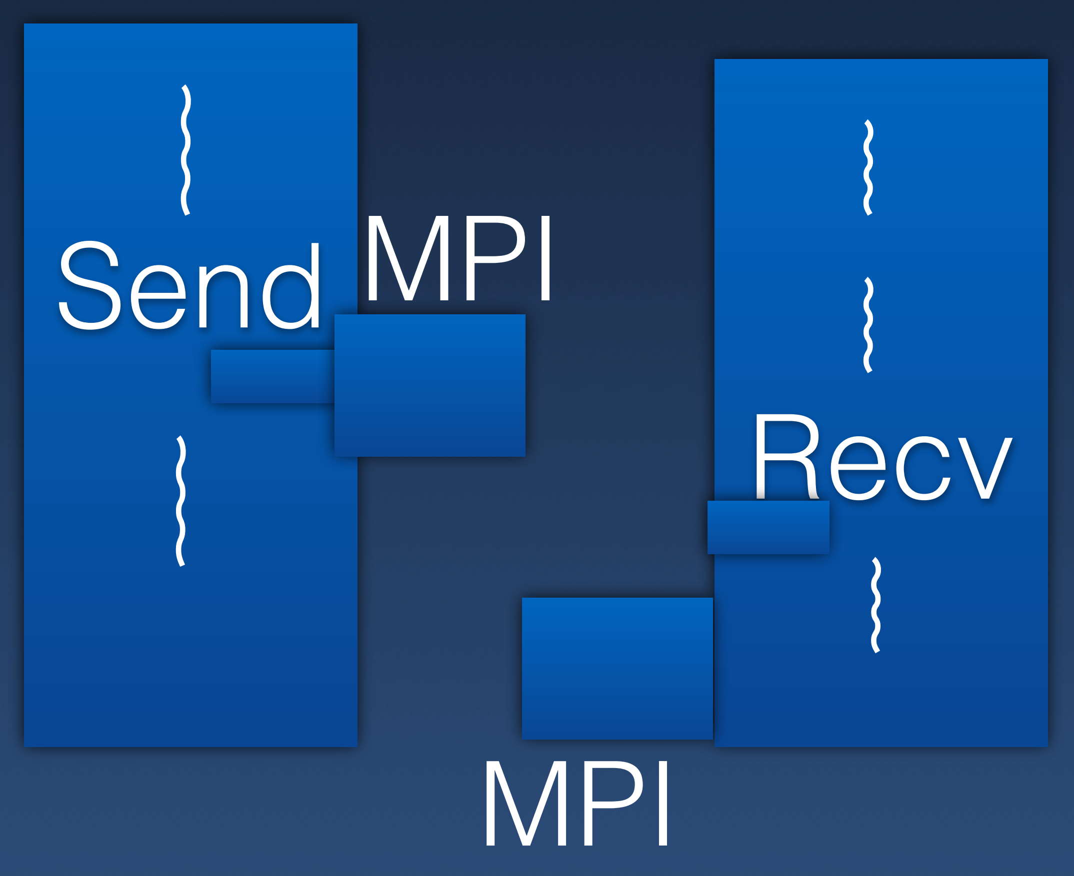


Message Semantics

- Variables, Buffers, and Packets
  - ➡ Application to application
- Lossy?
  - ➡ Deal with loss
  - ➡ Acks
- FIFO?
- Point to Point vs Collective?
- Addressing?

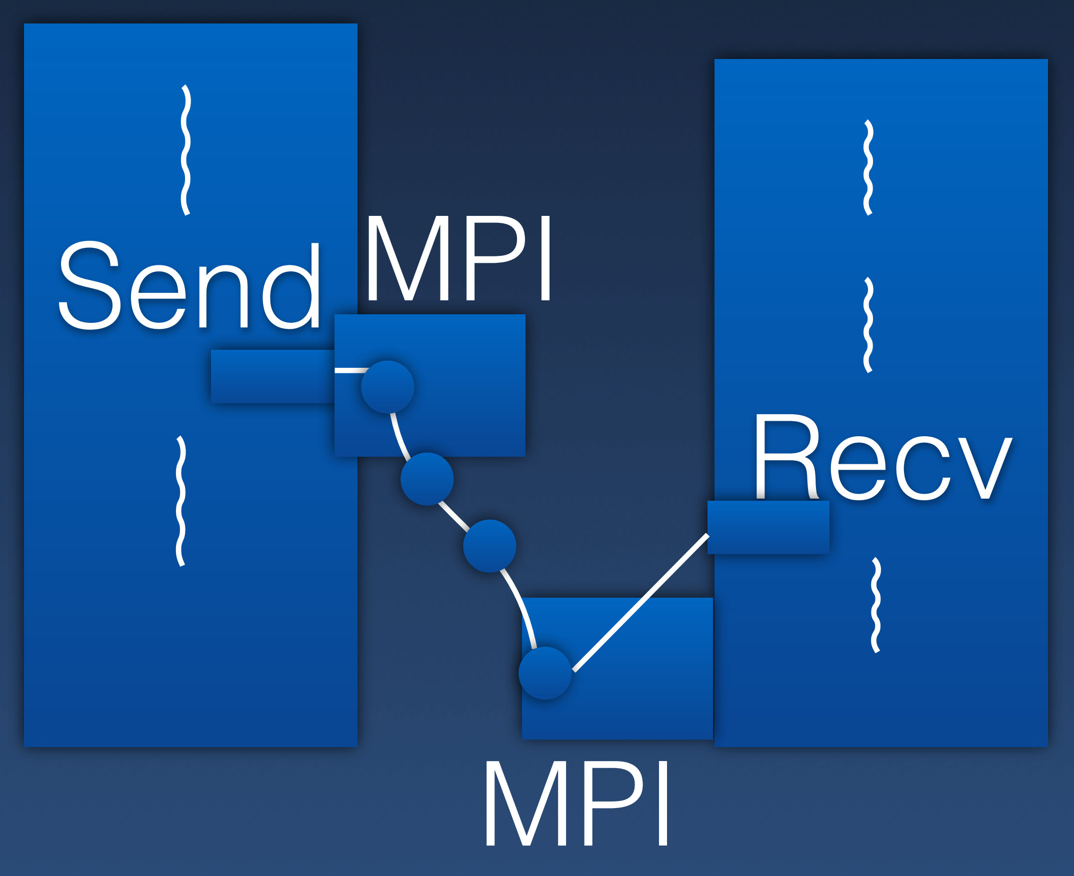


- Variables, Buffers, and Packets
  - ➡ Application to application
- Lossy?
  - ➡ Deal with loss
  - ➡ Acks
- FIFO?
- Point to Point vs Collective?
- Addressing?



Message Semantics

- Variables, Buffers, and Packets
  - ➡ Application to application
- Lossy?
  - ➡ Deal with loss
  - ➡ Acks
- FIFO?
- Point to Point vs Collective?
- Addressing?



Process Organization

- Communicator

  Topology

  - Groups of processes sharing a context
  - Intra and inter-communicator

  Predefined constant: MPI_COMM_WORLD

- Context
  - "communication universe"
  - Messages across context have no 'interference'
- Groups
  - Collection of processes (can build hierarchy)
  - Ordered

  Use group-rank to address

Hello MPI

```
#include "mpi.h"      /* includes MPI library code specs */

#define MAXSIZE 100

int main(int argc, char* argv[])
{
  MPI_Init(&argc, &argv);                      // start MPI
  int nProcs, myRank, dat[2] = {5,6};
  MPI_Status status;
  MPI_Comm_size(MPI_COMM_WORLD, &nProcs); // Group size
  MPI_Comm_rank(MPI_COMM_WORLD, &myRank); // get my rank
  If(myRank == 0)
      MPI_Send(dat, 2, MPI_INT, nProcs-1, 11, MPI_COMM_WORLD);
  If(myRank == nProcs-1)
      MPI_Recv(dat, 9, MPI_INT, 0, 11, MPI_COMM_WORLD, &status);
  MPI_Finalize();                              // stop MPI
}
```

Hello MPI

```
#include "mpi.h"      /* includes MPI library code specs */

#define MAXSIZE 100

int main(int argc, char* argv[])
{
  MPI_Init(&argc, &argv);                    // start MPI
  int nProcs, myRank, dat[2] = {5,6};
  MPI_Status status;
  MPI_Comm_size(MPI_COMM_W
  MPI_Comm_rank(MPI_COMM_W
  If(myRank == 0)
      MPI_Send(dat, 2, MPI                                      LD);
  If(myRank == nProcs-1)
      MPI_Recv(dat, 9, MPI_INT, 0, 11, MPI_COMM_WORLD, &status);
  MPI_Finalize();                            // stop MPI
}
```

```
status.MPI_TAG

status.MPI_SOURCE

MPI_Get_count(&status, MPI_INT,  &count);
```

Subodh Kumar

Review

- Message Passing and MPI
- The spectrum from shared memory to Message passing style programing