Lecture 36: MPI, Hybrid Programming, and Shared Memory

William Gropp
www.cs.illinois.edu/~wgropp

Thanks to

- This material based on the SC14 Tutorial presented by
  - Pavan Balaji
  - William Gropp
  - Torsten Hoefler
  - Rajeev Thakur

MPI and Threads

- MPI describes parallelism between processes (with separate address spaces)
- Thread parallelism provides a shared-memory model within a process
- OpenMP and Pthreads are common models
  - OpenMP provides convenient features for loop-level parallelism. Threads are created and managed by the compiler, based on user directives.
  - Pthreads provide more complex and dynamic approaches. Threads are created and managed explicitly by the user.

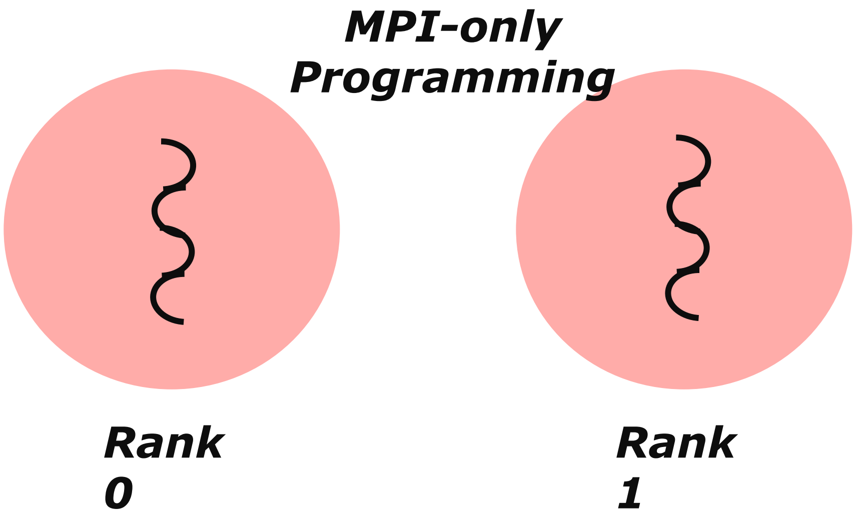
1867

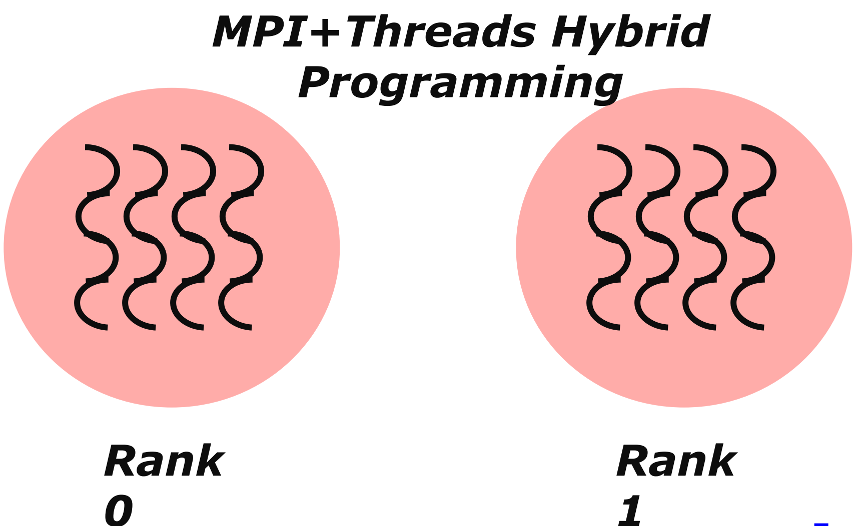
PARALLEL@ILLINOIS

Programming for Multicore

- Common options for programming multicore clusters
  - All MPI
    - MPI between processes both within a node and across nodes
    - MPI internally uses shared memory to communicate within a node
  - MPI + OpenMP
    - Use OpenMP within a node and MPI across nodes
  - MPI + Pthreads
    - Use Pthreads within a node and MPI across nodes
- The latter two approaches are known as "hybrid programming"

Hybrid Programming with MPI+Threads





- In MPI-only programming, each MPI process has a single program counter
- In MPI+threads hybrid programming, there can be multiple threads executing simultaneously
  - All threads share all MPI objects (communicators, requests)
  - The MPI implementation might need to take precautions to make sure the state of the MPI implementation is consistent

PARALLEL@ILLINOIS

MPI's Four Levels of Thread Safety

- MPI defines four levels of thread safety -- these are commitments the application makes to the MPI
  - MPI_THREAD_SINGLE: only one thread exists in the application
  - MPI_THREAD_FUNNELED: multithreaded, but only the main thread makes MPI calls (the one that called MPI_Init_thread)
  - MPI_THREAD_SERIALIZED: multithreaded, but only one thread at a time makes MPI calls
  - MPI_THREAD_MULTIPLE: multithreaded and any thread can make MPI calls at any time (with some restrictions to avoid races – see next slide)
- Thread levels are in increasing order
  - If an application works in FUNNELED mode, it can work in SERIALIZED
- MPI defines an alternative to MPI_Init
  - MPI_Init_thread(requested, provided)
    - Application specifies level it needs; MPI implementation returns level it supports

MPI_THREAD_SINGLE

- There are no threads in the system
  - ♦ E.g., there are no OpenMP parallel regions

```
int main(int argc, char ** argv)
{
    int buf[100];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    for (i = 0; i < 100; i++)
        compute(buf[i]);

    /* Do MPI stuff */

    MPI_Finalize();

    return 0;
}
```

MPI_THREAD_FUNNELED

- All MPI calls are made by the master thread
  - ♦ Outside the OpenMP parallel regions
  - ♦ In OpenMP master regions

```
int main(int argc, char ** argv)
{
    int buf[100], provided;

    MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, &provided);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

#pragma omp parallel for
    for (i = 0; i < 100; i++)
        compute(buf[i]);

    /* Do MPI stuff */

    MPI_Finalize();

    return 0;
}
```

MPI_THREAD_SERIALIZED

- Only one thread can make MPI calls at a time
    - Protected by OpenMP critical regions

```
int main(int argc, char ** argv)
{
    int buf[100], provided;

    MPI_Init_thread(&argc, &argv, MPI_THREAD_SERIALIZED, &provided);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

#pragma omp parallel for
    for (i = 0; i < 100; i++) {
        compute(buf[i]);
#pragma omp critical
        /* Do MPI stuff */
    }

    MPI_Finalize();

    return 0;
}
```

MPI_THREAD_MULTIPLE

- Any thread can make MPI calls any time (restrictions apply)

```
int main(int argc, char ** argv)
{
    int buf[100], provided;

    MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

#pragma omp parallel for
    for (i = 0; i < 100; i++) {
        compute(buf[i]);
        /* Do MPI stuff */
    }

    MPI_Finalize();

    return 0;
}
```

Threads and MPI

- An implementation is not required to support levels higher than MPI_THREAD_SINGLE; that is, an implementation is not required to be thread safe
- A fully thread-safe implementation will support MPI_THREAD_MULTIPLE
- A program that calls MPI_Init (instead of MPI_Init_thread) should assume that only MPI_THREAD_SINGLE is supported
- A threaded MPI program that does not call MPI_Init_thread is an incorrect program (common user error)

Specification of MPI_THREAD_MULTIPLE

- ***Ordering:*** When multiple threads make MPI calls concurrently, the outcome will be as if the calls executed sequentially in some (any) order
  - Ordering is maintained within each thread
  - User must ensure that collective operations on the same communicator, window, or file handle are correctly ordered among threads
    - E.g., cannot call a broadcast on one thread and a reduce on another thread on the same communicator
  - It is the user's responsibility to prevent races when threads in the same application post conflicting MPI calls
    - E.g., accessing an info object from one thread and freeing it from another thread
- ***Blocking:*** Blocking MPI calls will block only the calling thread and will not prevent other threads from running or executing MPI functions

1867

Ordering in MPI_THREAD_MULTIPLE: Incorrect Example with Collectives

	Process 0	*Process 1*
Thread 1	MPI_Bcast(comm)	MPI_Bcast(comm)
Thread 2	MPI_Barrier(comm)	MPI_Barrier(comm)

- P0 and P1 can have different orderings of Bcast and Barrier
- Here the user must use some kind of synchronization to ensure that **either** thread 1 **or** thread 2 gets scheduled first on both processes
- Otherwise a broadcast may get matched with a barrier on the same communicator, which is not valid in MPI

Ordering in MPI_THREAD_MULTIPLE: Incorrect Example with RMA

```
int main(int argc, char ** argv)
{
    /* Initialize MPI and RMA window */

#pragma omp parallel for
    for (i = 0; i < 100; i++) {
        target = rand();
        MPI_Win_lock(MPI_LOCK_EXCLUSIVE, target, 0, win);
        MPI_Put(..., win);
        MPI_Win_unlock(target, win);
    }

    /* Free MPI and RMA window */

    return 0;
}
```

Different threads can lock the same process causing multiple locks to the same target before the first lock is unlocked

Ordering in MPI_THREAD_MULTIPLE: Incorrect Example with Object Management

	Process 0	*Process 1*
Thread 1	MPI_Bcast(comm)	MPI_Bcast(comm)
Thread 2	MPI_Comm_free(comm)	MPI_Comm_free(comm)

- The user has to make sure that one thread is not using an object while another thread is freeing it
  - ♦ This is essentially an ordering issue; the object might get freed before it is used

Blocking Calls in MPI_THREAD_MULTIPLE: Correct Example

	Process 0	*Process 1*
Thread 1	MPI_Recv(src=1)	MPI_Recv(src=0)
Thread 2	MPI_Send(dst=1)	MPI_Send(dst=0)

- An implementation must ensure that this example never deadlocks for any ordering of thread execution
- That means the implementation cannot simply acquire a thread lock and block within an MPI function. It must release the lock to allow other threads to make progress.

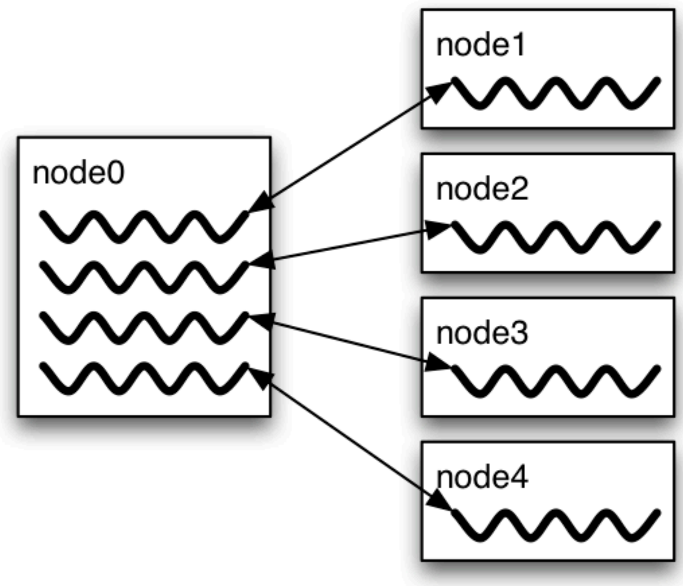
The Current Situation

- All MPI implementations support MPI_THREAD_SINGLE (duh).
- They probably support MPI_THREAD_FUNNELED even if they don't admit it.
  - ♦ Does require thread-safe malloc
  - ♦ Probably OK in OpenMP programs
- Many (but not all) implementations support THREAD_MULTIPLE
  - ♦ Hard to implement efficiently though (lock granularity issue)
- "Easy" OpenMP programs (loops parallelized with OpenMP, communication in between loops) only need FUNNELED
  - ♦ So don't need "thread-safe" MPI for many hybrid programs
  - ♦ But watch out for Amdahl's Law!

Performance with MPI_THREAD_MULTIPLE

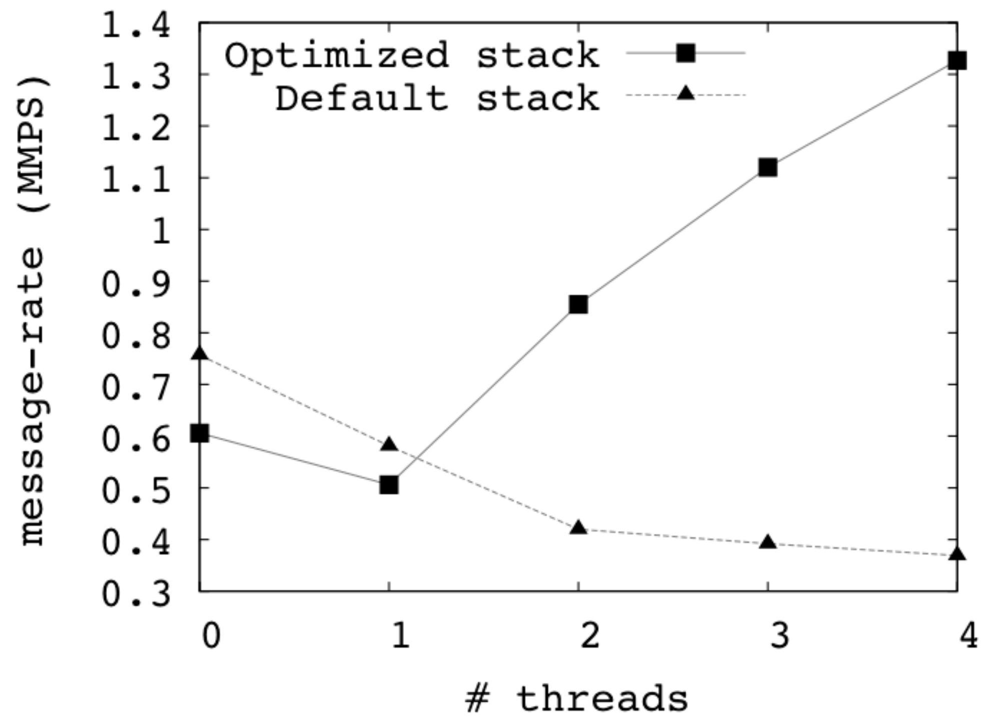
- Thread safety does not come for free
- The implementation must protect certain data structures or parts of code with mutexes or critical sections
- To measure the performance impact, we ran tests to measure communication performance when using multiple threads versus multiple processes
  - For results, see Thakur/Gropp paper: "Test Suite for Evaluating Performance of Multithreaded MPI Communication," Parallel Computing, 2009

Message Rate Results on BG/P



Message Rate Benchmark

"Enabling Concurrent Multithreaded MPI Communication on Multicore Petascale Systems" EuroMPI 2010



Why is it hard to optimize MPI_THREAD_MULTIPLE

- MPI internally maintains several resources
- Because of MPI semantics, it is required that all threads have access to some of the data structures
  - E.g., thread 1 can post an Irecv, and thread 2 can wait for its completion – thus the request queue has to be shared between both threads
  - Since multiple threads are accessing this shared queue, it needs to be locked – adds a lot of overhead

Hybrid Programming: Correctness Requirements

- Hybrid programming with MPI+threads does not do much to reduce the complexity of thread programming
  - Your application still has to be a correct multi-threaded application
  - On top of that, you also need to make sure you are correctly following MPI semantics
- Many commercial debuggers offer support for debugging hybrid MPI +threads applications (mostly for MPI +Pthreads and MPI+OpenMP)

Example of The Difficulty of Thread Programming

- Ptolemy is a framework for modeling, simulation, and design of concurrent, real-time, embedded systems
- Developed at UC Berkeley (PI: Ed Lee)
- It is a rigorously tested, widely used piece of software
- Ptolemy II was first released in 2000
- Yet, on April 26, 2004, four years after it was first released, the code deadlocked!
- The bug was lurking for 4 years of widespread use and testing!
- A faster machine or something that changed the timing caught the bug
- See "The Problem with Threads" by Ed Lee, IEEE Computer, 2006

An Example Encountered Recently

- The MPICH group received a bug report about a very simple multithreaded MPI program that hangs
- Run with 2 processes
- Each process has 2 threads
- Both threads communicate with threads on the other process as shown in the next slide
- Several hours spent trying to debug MPICH before discovering that the bug is actually in the user's program ☹

2 Proceses, 2 Threads, Each Thread Executes this Code

```
for (j = 0; j < 2; j++) {
    if (rank == 1) {
        for (i = 0; i < 2; i++)
          MPI_Send(NULL, 0, MPI_CHAR, 0, 0,MPI_COMM_WORLD);
        for (i = 0; i < 2; i++)
          MPI_Recv(NULL, 0, MPI_CHAR, 0, 0, MPI_COMM_WORLD, &stat);
    }
    else {  /* rank == 0 */
        for (i = 0; i < 2; i++)
          MPI_Recv(NULL, 0, MPI_CHAR, 1, 0, MPI_COMM_WORLD, &stat);
        for (i = 0; i < 2; i++)
          MPI_Send(NULL, 0, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
    }
}
```

Intended Ordering of Operations

Rank 0	***Rank 1***
2 recvs (T1)	2 sends (T1)
2 sends (T1)	2 recvs (T1)
2 recvs (T1)	2 sends (T1)
2 sends (T1)	2 recvs (T1)
2 recvs (T2)	2 sends (T2)
2 sends (T2)	2 recvs (T2)
2 recvs (T2)	2 sends (T2)
2 sends (T2)	2 recvs (T2)

- Every send matches a receive on the other rank

Possible Ordering of Operations in Practice

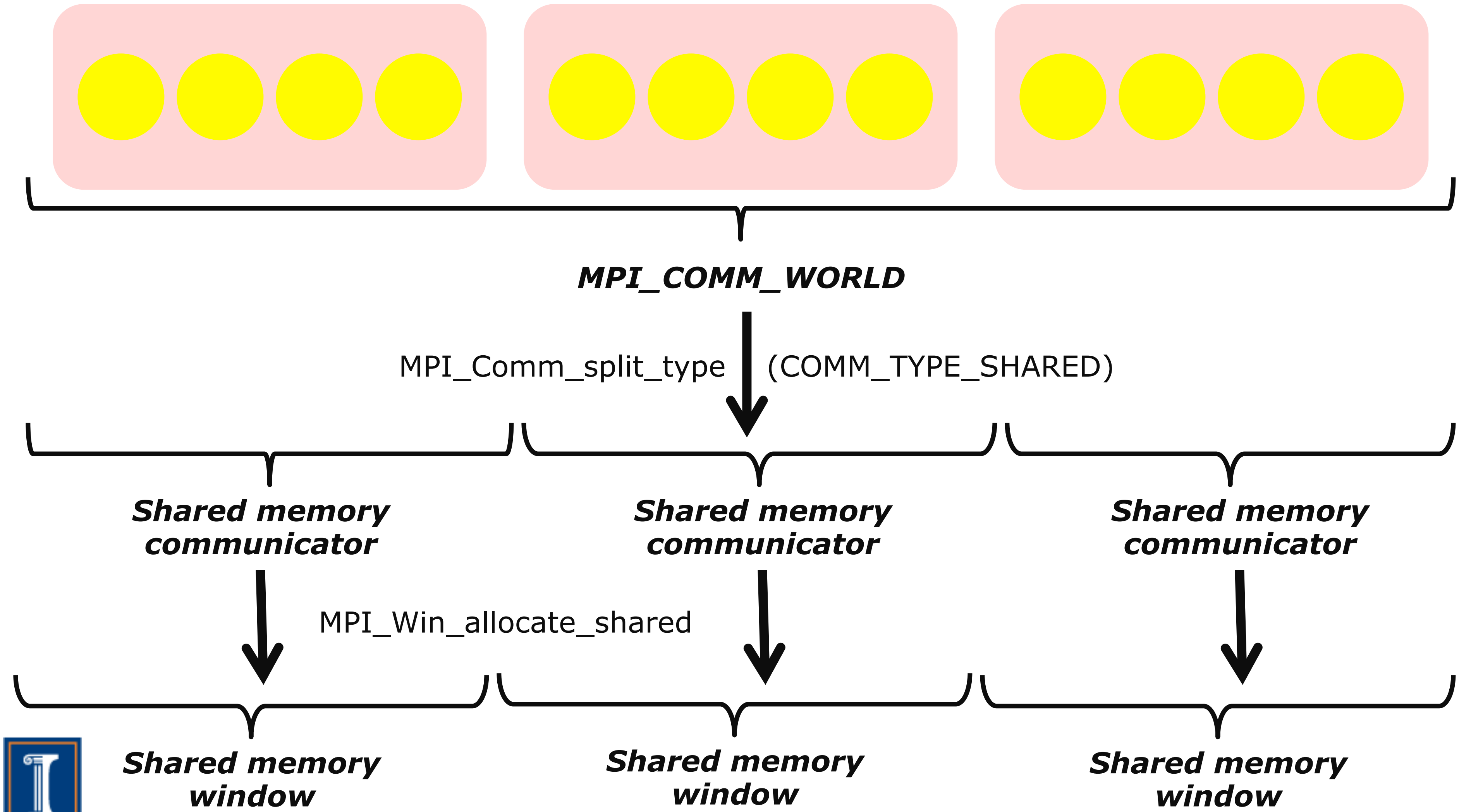
Rank 0		***Rank 1***	
2 recvs (T1)		2 sends (T1)	
2 sends (T1)		1 recv (T1)	
1 recv (T1)			2 sends (T2)
	1 recv (T2)		1 recv (T2)
--------	--------	--------	--------
1 recv (T1)	1 recv (T2)	1 recv (T1)	1 recv (T2)
--------	--------	--------	--------
2 sends (T1)	2 sends (T2)	2 sends (T1)	2 sends (T2)
	2 recvs (T2)	2 recvs (T1)	2 recvs (T2)
	2 sends (T2)		

- Because the MPI operations can be issued in an arbitrary order across threads, all threads could block in a RECV call

Hybrid Programming with Shared Memory

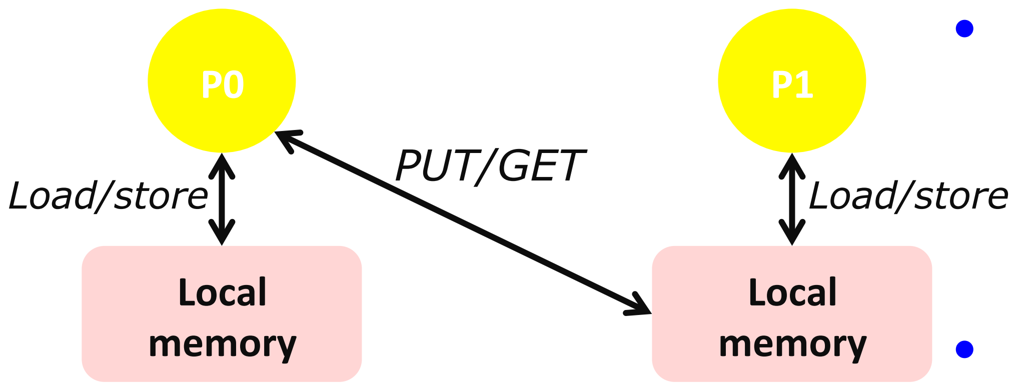
- MPI-3 allows different processes to allocate shared memory through MPI
  - MPI_Win_allocate_shared
- Uses many of the concepts of one-sided communication
- Applications can do hybrid programming using MPI or load/store accesses on the shared memory window
- Other MPI functions can be used to synchronize access to shared memory regions
- Can be simpler to program than threads

Creating Shared Memory Regions in MPI

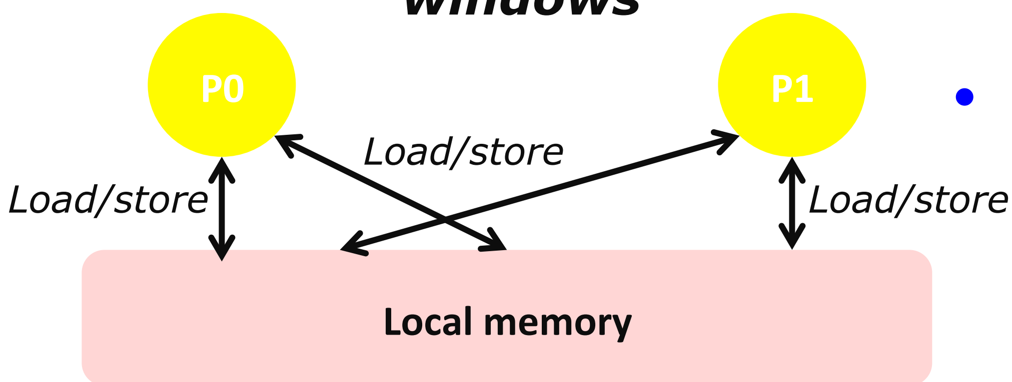


PARALLEL@ILLINOIS

Regular RMA windows vs. Shared memory windows



Traditional RMA windows



Shared memory windows

- Shared memory windows allow application processes to directly perform load/store accesses on all of the window memory
  - E.g., x[100] = 10
- All of the existing RMA functions can also be used on such memory for more advanced semantics such as atomic operations
- Can be very useful when processes want to use threads only to get access to all of the memory on the node
  - You can create a shared memory window and put your shared data

Memory Allocation And Placement

- Shared memory allocation does not need to be uniform across processes
  - Processes can allocate a different amount of memory (even zero)
- The MPI standard does not specify where the memory would be placed (e.g., which physical memory it will be pinned to)
  - Implementations can choose their own strategies, though it is expected that an implementation will try to place shared memory allocated by a process "close to it"
- The total allocated shared memory on a communicator is contiguous by default
  - Users can pass an info hint called "noncontig" that will allow the MPI implementation to align memory allocations from each process to appropriate boundaries to assist with placement

Shared Arrays with Shared Memory Windows

```
int main(int argc, char ** argv)
{
    int buf[100];

    MPI_Init(&argc, &argv);
    MPI_Comm_split_type(..., MPI_COMM_TYPE_SHARED, .., &comm);
    MPI_Win_allocate_shared(comm, ..., &win);

    MPI_Comm_rank(comm, &rank);

    MPI_Win_lockall(win);

    /* copy data to local part of shared memory */
    MPI_Barrier(comm);

    /* use shared memory */

    MPI_Win_unlock_all(win);
    MPI_Win_free(&win);
    MPI_Finalize();
    return 0;
}
```

Summary

- MPI + X a reasonable way to handle
  - Extreme parallelism
  - SMP nodes; other hierarchical memory architectures
- Many choices for X
  - OpenMP
  - pthreads
  - MPI (using shared memory)