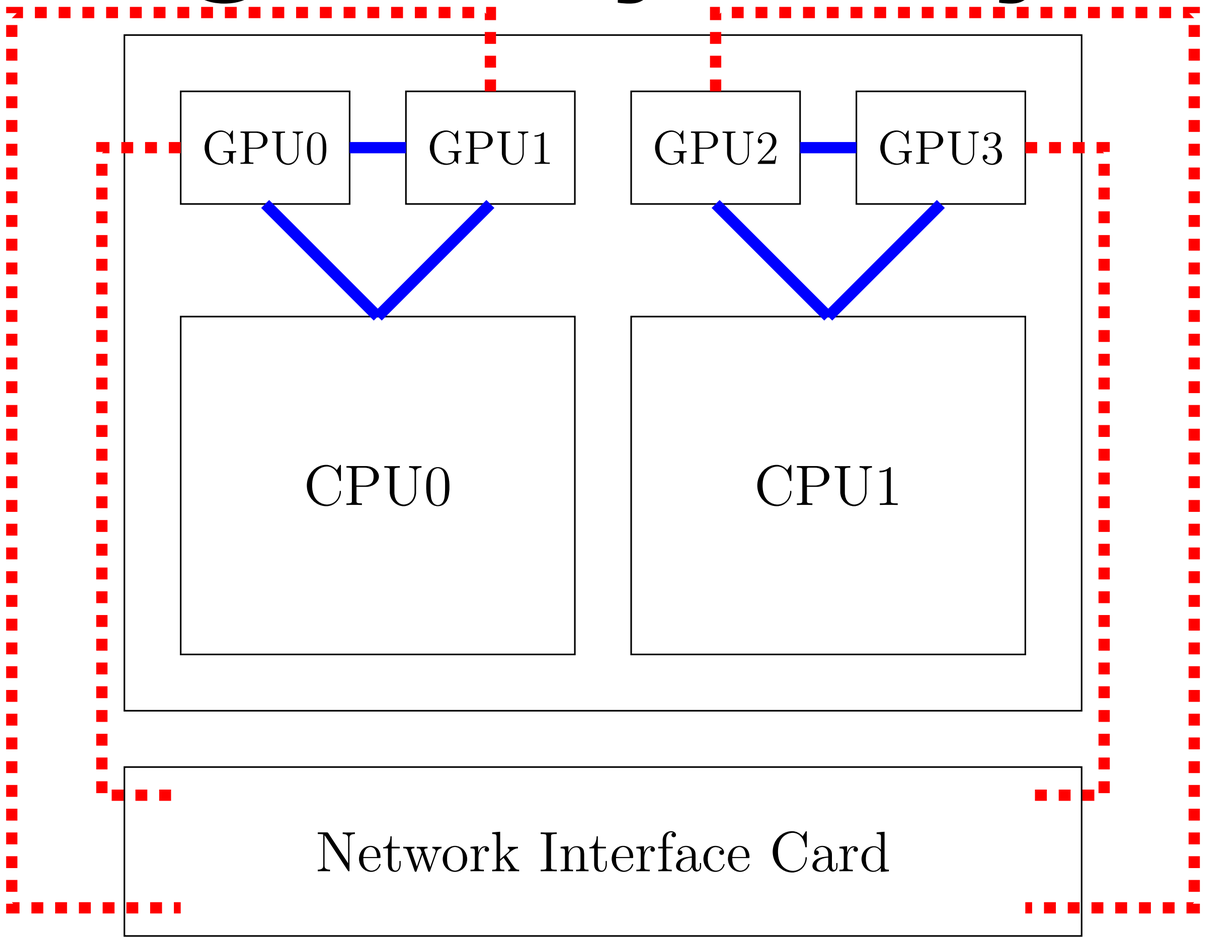
Introduction to Parallel Processing

Lecture 26 : MPI + X

12/05/2022

Professor Amanda Bienz

Heterogeneity in Systems



Want to use MPI between nodes, but can further optimize among processes within a node!

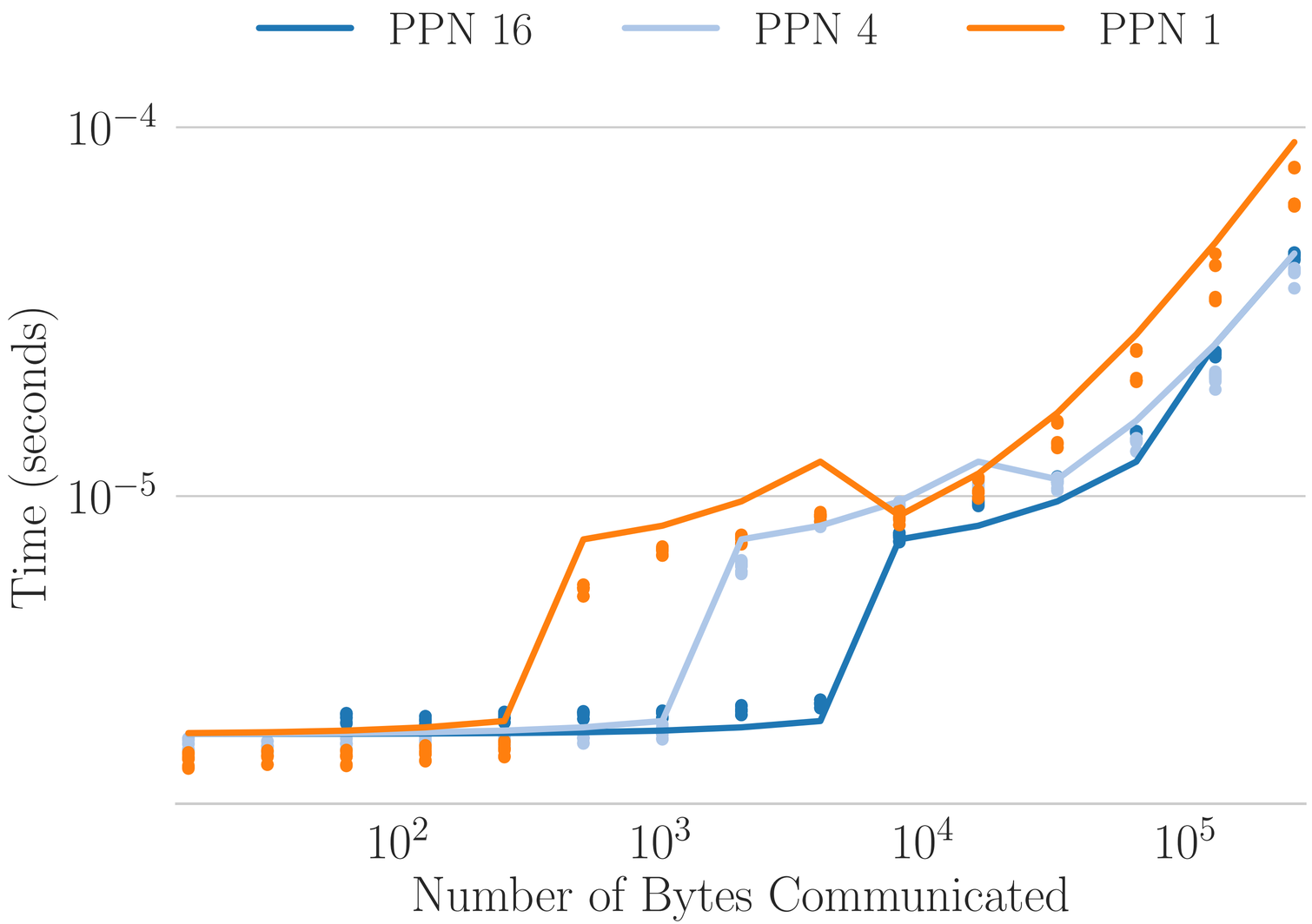
MPI + X

- MPI between nodes
- Something else (x) for on-node or accelerated computing
  - **OpenMP : Shared Memory CPUs and GPUs**
  - **MPI Shared Memory : Shared Memory CPUs**
  - CUDA (Compute Unified Device Architecture) : NVIDIA GPUs
  - HIP (Heterogeneous-Computing Interface) : NVIDIA or AMD GPUs
  - OpenCL : Any GPU (lower-level programming)
  - OpenACC : Any GPU (higher-level programming)

MPI + OpenMP

- Use MPI between nodes
- Use OpenMP on each node (within shared memory region)
- Standard approach : run with a small number of MPI processes per node (maybe 1) and have each MPI process utilize multiple OpenMP threads
  - Changes very little in program, easy to do
  - Downside : communication requirements now all on small number of processes

Performance : MPI Cores Per Node



Task-Based MPI + OpenMP

- Instead of using #pragma omp parallel for, use #pragma omp parallel around entire program
- Then, have one (or a few) threads do all communication
- Have remaining OpenMP threads do all computation
- Can see overlap between communication and computation

MPI Thread Safety

- When using threads, change MPI_Init to MPI_Init_thread

- MPI_Init_thread(int** argc,
  char** argv[],
  MPI_THREAD_LEVEL,
  int* provided)

  1. MPI_THREAD_SINGLE : only one thread in application (similar to MPI_Init)

  2. MPI_THREAD_FUNNELED : multithreaded, but only thread calling MPI_Init_thread can make any MPI calls (similar to first MPI + OpenMP example)

  3. MPI_THREAD_SERIALIZED : multithreaded, but only one thread makes MPI calls at any time

  4. MPI_THREAD_MULTIPLE : multithreaded, any thread can make MPI calls at any time

- Thread levels in order… if program works with MPI_THREAD_FUNNELED, it would also work with MPI_THREAD_SERIALIZED or MPI_THREAD_MULTIPLE

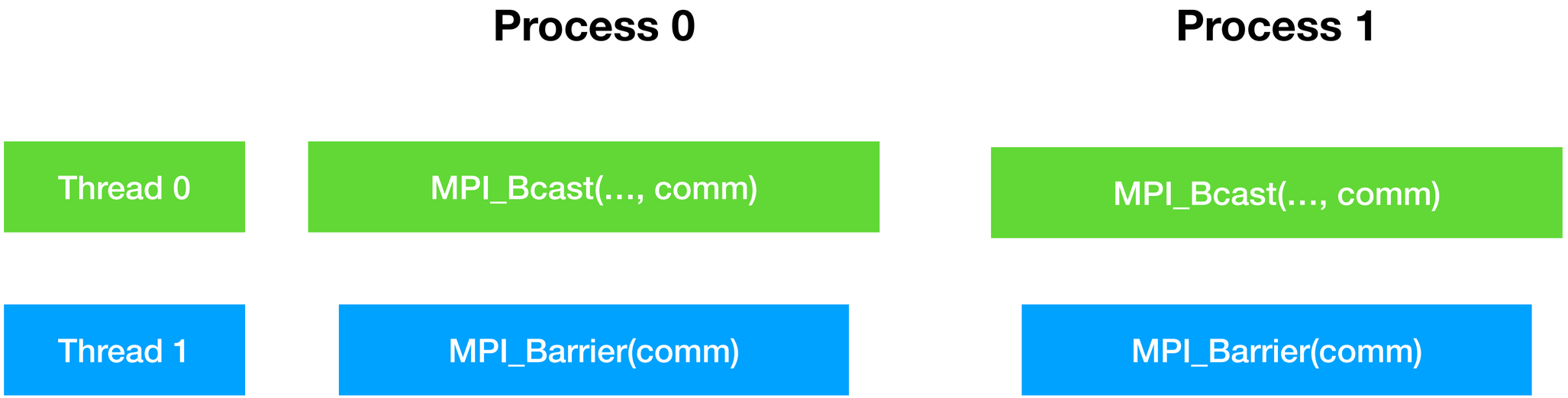
MPI Thread Safety

- MPI Implementations are not required to support thread safety… may only support MPI_THREAD_SINGLE
- If calling MPI_Init, instead of MPI_Init_thread, assume only MPI_THREAD_SINGLE is supported

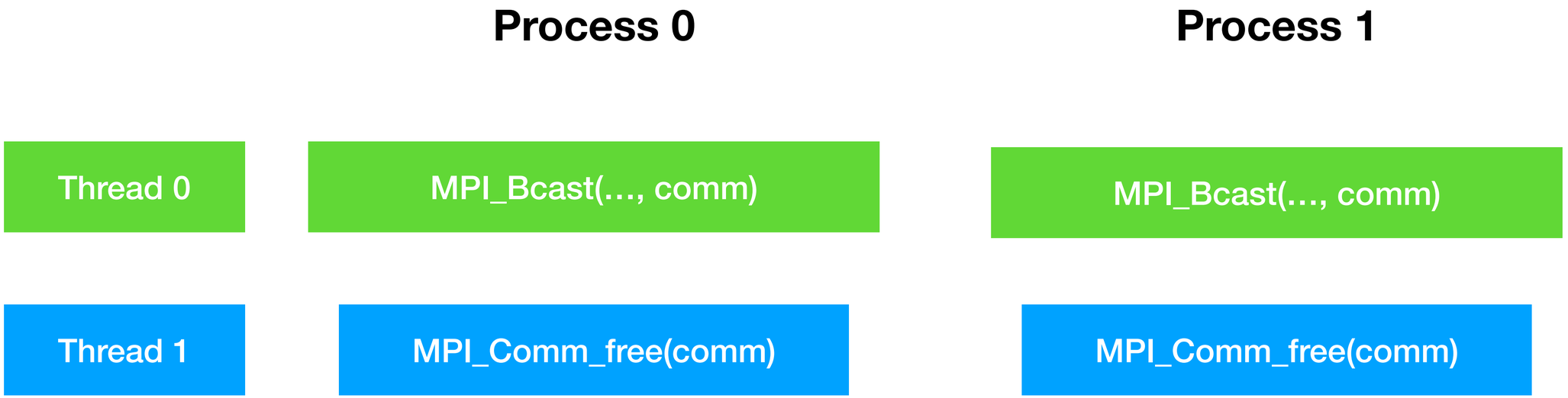
MPI_THREAD_MULTIPLE

- Ordering : We do not know which order threads will be executed… MPI_THREAD_MULTIPLE allows program to make MPI calls concurrently, which could result in any sequential execution
  - Ordering within each thread is maintained
  - Must be sure all collective operations on same communicator (or window / file handle) are ordered correctly
    - Cannot call broadcast on one thread and reduce on another thread in same communicator!
  - User is responsible to prevent race conditions with threads
- Blocking MPI calls : block only calling thread… other threads can continue running

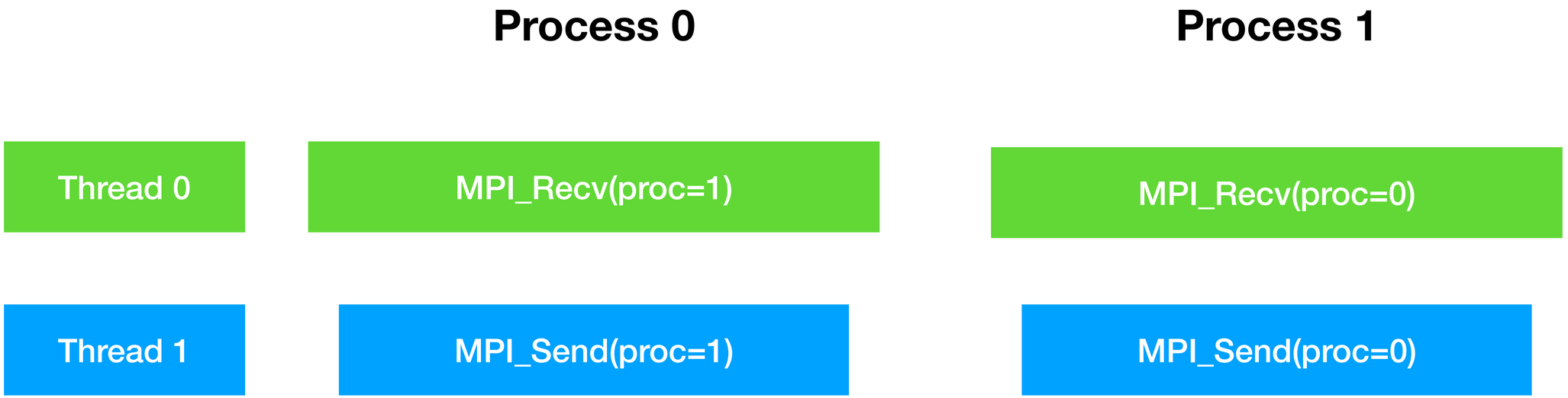
MPI_THREAD_MULTIPLE Orderings



MPI_THREAD_MULTIPLE Orderings



MPI_THREAD_MULTIPLE Orderings



MPI_THREAD_MULTIPLE Performance

- Performance overhead (sometimes large) with using MPI_THREAD_MULTIPLE
- Thread safety requires implementation to add locks or critical sections around parts of code
- May see large slowdown in baseline performance when turning on MPI_THREAD_MULTIPLE

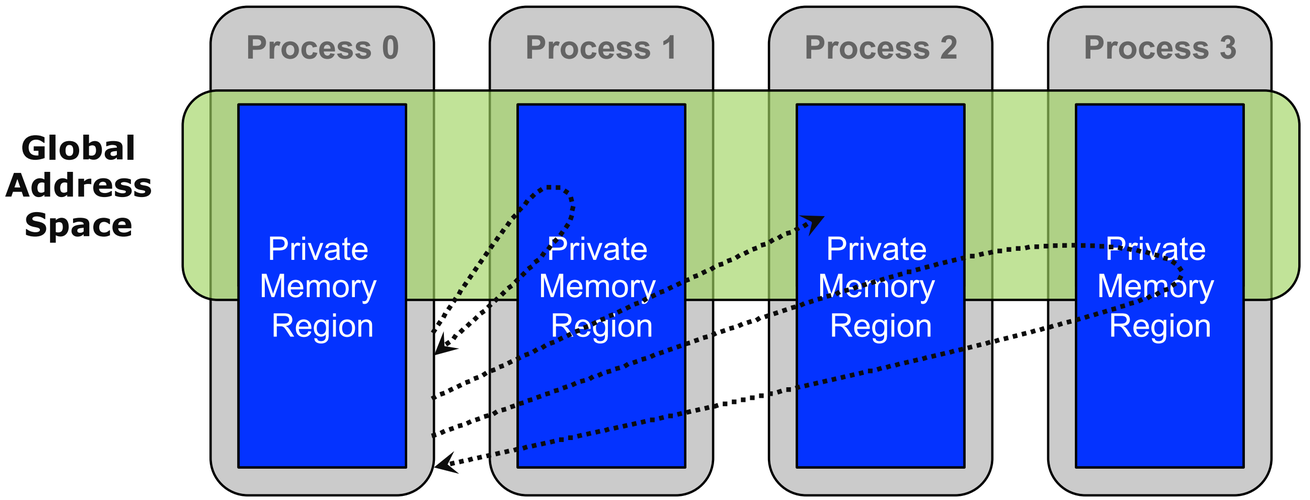
MPI Shared Memory

- Think back to MPI one-sided communication… similar concepts can be used to exploit shared memory on-node, rather than using OpenMP threads
- Create shared memory windows : MPI processes can perform loads and stores on any memory in the window
- If use of threads is to get access to all memory on the node, MPI shared memory will likely be a more efficient route

MPI Shared Memory

Basic Idea

- Think back to MPI one-sided communication… similar concepts can be used to exploit shared memory on-node, rather than using OpenMP threads
- Create shared memory windows : MPI processes can perform loads and stores on any memory in the window
- If use of threads is to get access to all memory on the node, MPI shared memory will likely be a more efficient route



First, create a window:

- MPI_WIN_CREATE : have already allocated buffer that would like to be remotely accessible
- MPI_WIN_ALLOCATE : want to create a buffer and directly make it remotely accessible
- MPI_WIN_CREATE_DYNAMIC : don’t have a buffer yet but will in the future
- **MPI_WIN_ALLOCATE_SHARED : want multiple processes on same node to share a buffer**

MPI_Win_allocate_shared

- This method will create an MPI Window object that is capable of shared memory accesses.

- It will allocate memory for each process

- MPI_Win_allocate_shared(MPI_Aint size,
  int disp_unit,
  MPI_Info info,
  MPI_Comm comm,
  void* base-to,
  MPI_Win* win)

MPI Info

- Haven’t talked much about this, but know that MPI Info can be used to give hints to the MPI implementation about how to get best performance
- MPI_Info_create()
- MPI_Info_set(…) : adds a key, value to the info
- Important key for MPI shared memory:
  - “alloc_shared_noncontig” : if not set to true, memory is continuous across process ranks

Need for locks

- MPI_Win_lock_all : locks all members of the window so that only one process can operate on window object at a time
- MPI_Win_unlock_all : unlocks so that next process can use memory in window
- Note : all does not refer to collective routine… just that all processes in window are locked
- Also note : all processes can move past this line, but cannot access window until unlocked (can have barrier inside of locks)

Flush and Sync

- MPI_Win_flush (int rank, MPI_Win) : completes all operations that calling process performed on rank's portion of window
- MPI_Win_flush_all(MPI_Win) : completes all operations issued by calling process (on any target) in window
- MPI_Win_sync(MPI_Win) : synchronizes private and public copies in the window

MPI_Win_shared_query

- The method queries local address for remote memory segments created with MPI_Win_allocate_shared
- Find out which parts of shared memory belong to a given process
- Returns a buffer, which can be used to load/store data

MPI Shared Memory vs MPI + OpenMP

- OpenMP parallel for loops:
  - Pros : very simple to program
  - Cons : fewer MPI processes are active in communication, starting up threads has a cost
- OpenMP task based programming:
  - Pros : can have more threads active in communication, can overlap communication and computation
  - Cons : slightly more difficult to program, still need to launch threads (overhead)
- MPI Shared Memory:
  - Pros : all processes can still be active in MPI communication, no need for costly MPI_THREAD_MULTIPLE
  - Cons : most difficult to program, can have to change large portions of your code

OpenMP Offloading

OpenMP Offloading

- OpenMP parallelizes for loops or blocks of code, by splitting across multiple threads
- Similar to accelerators (CUDA) where tasks are split across many many threads on the GPU
- Recent OpenMP versions support GPUs:
  - Write OpenMP code
  - Behind the scenes, data is moved to the GPU, threads are executed on the GPU, and data is copied back to the CPU

Matrix-Matrix Multiplication

- Let’s look back at our example of matrix-matrix multiplication with OpenMP
- Each of our outermost loops (both setting values and multiplying) has #pragma omp parallel
- Remember, we need to declare variables as private

Moving to the GPU

- #pragma omp target : instructs compiler to map variables to device data and execute enclosed block of code on GPU
- Map : which variables need to be copied between CPU and GPU (each of the matrices)
  - Takes a direction
    - to : copy original values from CPU to GPU
    - from : copy final values from GPU to CPU
    - tofrom : copy both directions
    - alloc : data is allocated on device
  - Must use array notation for pointers : map(to:A[0:N])

Matrix-Matrix Multiplication Times

- Serial : 1.74e00 seconds
- OpenMP Threads (CPU) : 1.12e-01 seconds
- OpenMP Offloading : 1.12e01 seconds

OpenMP Target Data Only

- #pragma omp target data
  - Offloads data from CPU to GPU, but not execute
  - Manually specify which executions should happen on the GPU and everything else will happen on the CPU

OpenMP Teams

- Distribute iterations to teams (or groups) of threads
- Workshare within the teams
- Ideally want to parallelize within the teams : call parallel for loop within each team (inner loop)

Collapse

- Add “collapse(n)” to your pragma statement to combine n of the loops into one
- Distribute then splits up all iterations between teams

Schedule

- GPUs only support static scheduling
- On CPUs, schedule(static) works best, splitting n/p continuous iterations to each process
- On GPUs, this is probably not what we want (think memory coalescing)
  - Would probably rather have schedule(static,1) to partition iterations in a round-robin style

Reduction

- Reduction does what you would expect (same as OpenMP conceptually)
- However, the variable that is being reduced must appear in a map(to from) clause to get the value back at the end