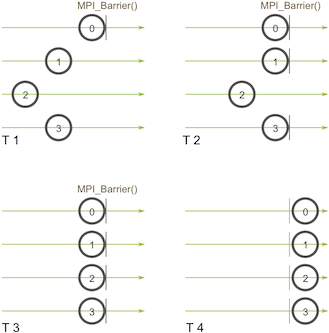
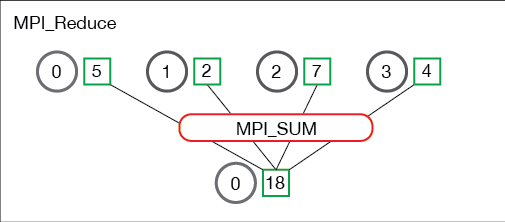
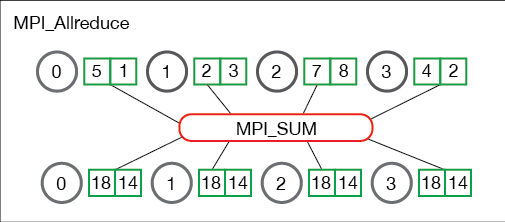
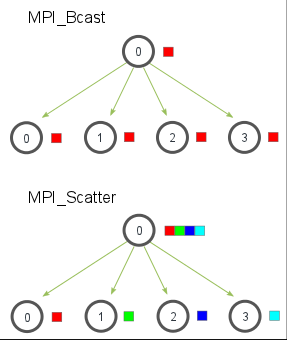
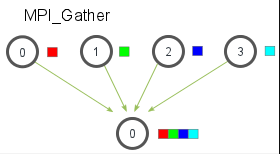
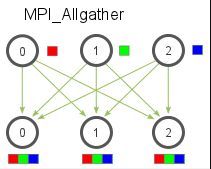
Midterm Review

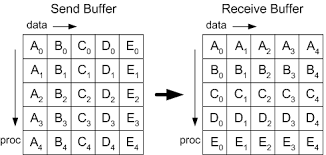
* Memory Hierarchy
  + Types - cache and main memory
  + Cache
    - Very small but very fast to read from
    - Data is first stored in main memory then stored in cache when being read into the program.
    - Cost of reading data: Latency and Bandwidth
      * Bandwidth: how much data your connection can handle
      * Latency: the delays that determine how quickly data gets to ur device
    - Closer the cache is to register, lower latency, higher bandwidth
    - Cache Lines: a group of data is read from memory (say 64 bytes)
    - If cache is full, data is evicted/erased
    - Cache hit: data in cache
    - Cache miss: not in cache
    - During a cache miss, fetch entire line from memory
      * Neighboring items will then be read from cache
    - Writing data to cache is very complicated
      * Write Hit : data to be written already in cache
      * Write-Miss : Cache line first transferred from main memory to cache before being modified
  + Main Memory
    - Very large but very slow to read from
  + Assume I have an array that is small enough to fit in cache…
    - When is the array read from main memory?
      * It is written and read from main memory even if it is small enough to be placed in cache
    - When is the array read from cache?
      * Anytime after the initial writing and reading from main memory, it will always be read from cache after
* Shared Memory
  + OpenMP (open multi-processing) is used for vectorizations and improves performance of programs
    - Vectorization: converting algorithm from operating on single value to operating on a set of values at one time
  + Data Dependencies
    - Situation in which a program statement (instruction) refers to the data of a preceding statement
  + Uses threads which is the same concept as processes
  + If one process changes it’s copy other values are either invalidated or updated otherwise, a process could work with incorrect version of the data
  + Whenever data is written, all copies in system are updated
    - full cache line is written to main memory and sent to the other processes
  + What if x and y are on the same cache line?
    - System only invalidates/updates by cache line, can’t detect the updates aren’t actually the same
  + If x is in the cache for multiple processes and in main memory then it is the correct way to create shared memory
  + If multiple threads want to write to the same variable, can they all write at the same time?
    - No, need to synchronize access to those variables
  + OpenMP Optimizations
    - pragma omp parallel
      * Initialize openmp
    - pragma omp for
      * Vectorize for loop under
    - pragma omp for private(variable)
      * Make a variable a private variable for that thread
    - pragma omp for shared(variable)
      * Make a variable a shared variable for all threads
    - pragma omp parallel reduction(+:sum)
      * Reduction is applying a function like sum to a variable over multiple threads
    - pragma omp simd
      * Performs SIMD (Single-Instruction, Multiple-Data) parallelism
  + In MPI, assume all threads want to calculate a global sum. Can the sum variable be shared?
    - No, use a reduction, not a shared variable
* MPI - Message Passing Interface
  + Typically, entire program is parallelized
  + Each process executes the entire program
  + Typically, don’t want every process executing the same thing
* MPI Point to point communication
  + MPI\_Send
    - MPI\_Send(const void\* send\_buffer, int message\_size, MPI\_Datatype message\_datatype, int destination\_process, int message\_tag, MPI\_Comm communicator)
      * Example of sending process 0 to send a single integer called ‘size’ to process 1
        + MPI\_Send(&size, 1 bytes, MPI\_INT, 1, 1234, MPI\_COMM\_WORLD);
    - If all processes send, and none receive, your program will get stuck and hang indefinitely
  + MPI\_Recv
    - MPI\_Recv(const void\* send\_buffer, int message\_size, MPI\_Datatype message\_datatype, int destination\_process, int message\_tag, MPI\_Comm communicator, MPI\_Status\* status)
      * Example of process 1 receiving from process 0
        + MPI\_Recv(&size, 1, MPI\_INT, 0, 1234, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);
  + MPI\_ISend
    - The standard non-blocking send, does the same as send just is non-blocking
  + MPI\_IRecv
    - The standard non-blocking send, does the same as send just is non-blocking
  + MPI\_Barrier
    - MPI\_Barrier blocks all MPI processes in the given communicator until they all call this routine.

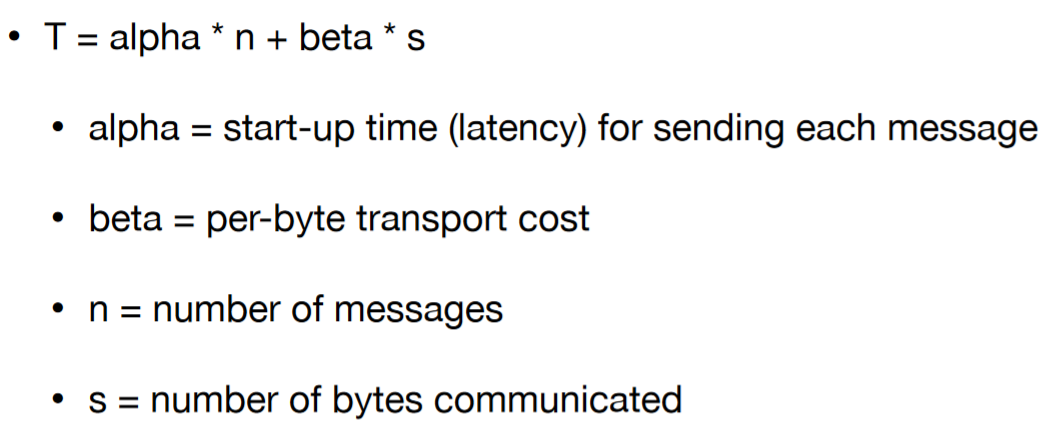


* MPI Collective operations
  + MPI\_Reduce
    - Takes an array of input elements on each process and returns an array of output elements to the root process. The output elements contain the reduced result.
  + MPI\_AllReduce
    - Will reduce the values and distribute the results to all processes
  + MPI\_Scatter
    - Involves a designated root process sending data to all processes in a communicator. Sends chunks of an array to different processes
  + MPI\_BCast
    - Does the same as Scatter, but sends the same piece of data to all processes instead
  + MPI\_Gather
    - Opposite of Scatter, instead of spreading elements from one process to many processes, MPI\_Gather takes elements from many processes and gathers them to one single process



* + MPI\_AllGather
    - Given a set of elements distributed across all processes, MPI\_Allgather will gather all of the elements to all the processes
  + MPI\_AllToAll
    - is a combination of MPI\_Scatter and MPI\_Gather. That is, every process has a buffer containing elements that will be scattered across all processes, as well as a buffer in which store elements that will be gathered from all other processes



* + MPI\_AllToAllV
    - MPI\_Alltoallv is a variant of MPI\_Alltoall; a combination of MPI\_Scatter and MPI\_Gather. That is, every process has a buffer containing elements that will be scattered across all processes, as well as a buffer in which store elements that will be gathered from all other processes
    - However, unlike MPI\_Alltoall, MPI\_Alltoallv allows the messages scattered to have different lengths and be stored at arbitrary locations in the send buffer, similarly for the messages gathered
* Blocking vs non-blocking
  + Blocking communication is done using MPI\_Send() and MPI\_Recv(). These functions do not return (i.e., they block) until the communication is finished. Simplifying somewhat, this means that the buffer passed to MPI\_Send() can be reused, either because MPI saved it somewhere, or because it has been received by the destination. Similarly, MPI\_Recv() returns when the receive buffer has been filled with valid data.
  + In contrast, non-blocking communication is done using MPI\_Isend() and MPI\_Irecv(). These functions return immediately (i.e., they do not block) even if the communication is not finished yet. You must call MPI\_Wait() or MPI\_Test() to see whether the communication has finished.
  + Blocking communication is used when it is sufficient, since it is somewhat easier to use. Non-blocking communication is used when necessary, for example, you may call MPI\_Isend(), do some computations, then do MPI\_Wait(). This allows computations and communication to overlap, which generally leads to improved performance.
  + Blocking operations:
    - MPI\_Send
      * MPI\_send = MPI\_ISend + MPI\_Wait
    - MPI\_Recv
      * MPI\_recv = MPI\_IRecv + MPI\_Wait
  + Non-Blocking operations:
    - MPI\_Isend
    - MPI\_Irecv
* Performance Modeling
  + Postal model calculates the cost of simple operation
  + Can use these simple measurements to analyze the cost of a program
  + Additions are made on depending on the algorithm we are trying to time
* “Barrier” algorithms
  + It is considered a barrier when all processes are waiting on something to finish to continue.