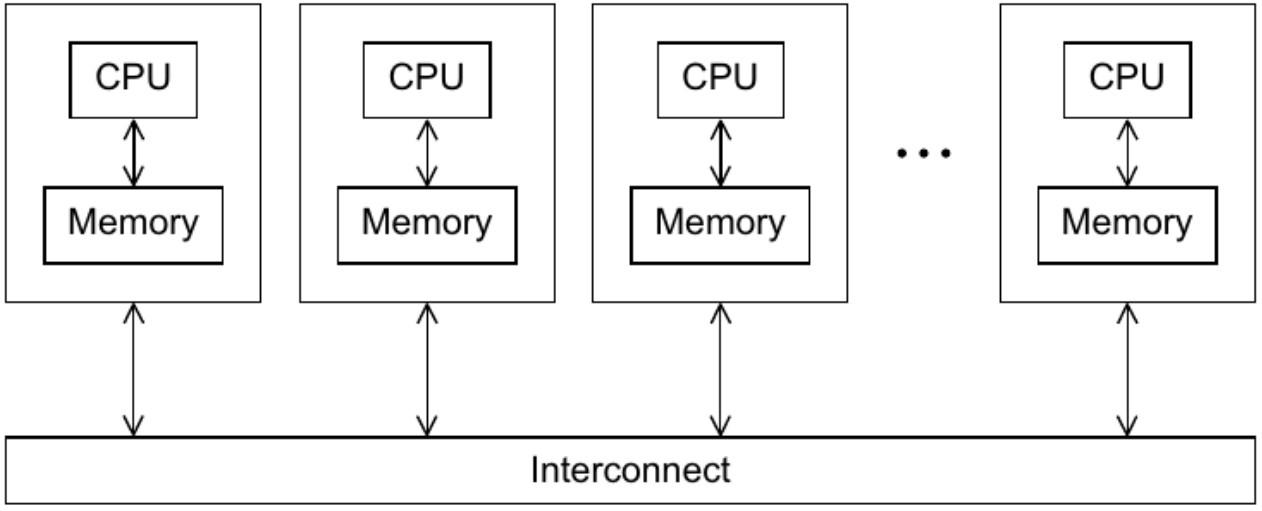
Reference Roadmap

The slide outlines contain references to the main course materials. Not everything has a reference but nearly all the materials can be found in the following references. The references are of the form `(<text>-<pages>)` so (PM-123) means page 123 of An Introduction to Parallel Programming by Peter S. Pacheco and Matthew Malensek.

- (VE-V1) — The Science of Computing: The Art of High Performance Computing, Vol 1 by Victor Eijkhout
- (VE-V2) — Parallel Programming in MPI and OpenMP: The Art of HPC, Vol 2 by Victor Eijkhout
- (VE-V3) — Introduction to Scientific Programming in C++17/Fortran2008: The Art of HPC, Vol 3 by Victor Eijkhout
- (VE-V4) — Tutorials for High Performance Scientific Computing: The Art of HPC, Vol 4 by Victor Eijkhout
- (PM) — An Introduction to Parallel Programming by Peter S. Pacheco and Matthew Malensek.
- (KH) — Programming Massively Parallel Processors: A Hands-on Approach by David B. Kirk and Wen-mei W. Hwu.
- (SAB) — High Performance Computing Modern Systems and Practices by Thomas Sterling, Matthew Anderson, and Maciej Brodowicz.
- (GLS) — Using MPI: Portable Parallel Programming with the Message-Passing Interface by William Gropp, Ewing Lusk, and Anthony Skjellum.
- (GHTL) — Using Advanced MPI: Modern Features of the Message-Passing Interface by William Gropp, Torsten Hoefler, Rajeev Thakur, and Ewing Lusk.

Definitions & Theory

– Distributed Memory Systems



- Basic architecture of large-scale clusters and supercomputers. This is simply another model for parallel hardware.
- Easy to expand interconnect. It is usually an Ethernet or InfiniBand switch.
- Message-passing approach to program utilization, MPI.
- Same theoretical basis, speedup, efficiency, scalability, load balancing, Amdahl's and Gustafson's laws. ....
- Very applicable to super-computer, cluster, and cloud systems. Less applicable to single processor symmetric processing.

MPI — Message Passing Interface

- A program running on one core-memory pair is usually called a process.
- Two processes can communicate by calling functions: one process calls a send function and the other calls a receive function. This is Point-To-Point communication.
- We can also communicate with more than two processes at the same time, even the total number of processes running, called collective communications.
- MPI is not a new programming language it is a library of functions that can be called from C and Fortran programs.
- There are bindings for C++, essentially just wrapper classes around the C functions. These bindings are deprecated and hence we will use the C versions of the calls. We can write in C++ it is just that the calls to the MPI functions will be C and hence there may be some type conversions we need to do.
- Examples
  - MPI Hello World Example (MPIHello). Trace and run.

MPI — Message Passing Interface

- To compile an MPI program we use the MPI compiler which adds on to the gcc compiler we already know. For a C program we run
  `mpicc -g -Wall -o MPIHello MPIHello.cpp`
  or for C++
  `mpic++ -g -Wall -o MPIHello MPIHello.cpp`
- To run an MPI program we use either `mpirun` or `mpiexec`. Both are the same. The following will run a copy of the program on each available core.
  `mpirun ./MPIHello`
  So of you have an 8-core system this will run 8 simultaneous copies of the program.

MPI — Message Passing Interface

- To run it with other numbers of processes use the `-n` flag.
  `mpirun -n <number of processes> ./MPIHello`
  so
  `mpirun -n 1 ./MPIHello`
  will run one copy of the program on one core, hence getting times for serial implementations.
  `mpirun -n 20 ./MPIHello`
  will run 20 copies of the program each on its own core.
- Note here that you are restricted to running one copy on a core. If you have n higher than the number of cores you have available you will get an error. So in the command, if you were running this on a system with only 16 cores the `mpirun` command would give you an error and not run the job.

MPI — Message Passing Interface

- If your cores are multi-threaded you can use all of your hardware threads by adding the flag `-use-hwthread-cpus`, for example,
  `mpirun -use-hwthread-cpus -n <num> ./MPIHello`
  In this case the number you supply must be at mode the total count of hardware threads or as above you will get an error. In addition, the command
  `mpirun -use-hwthread-cpus ./MPIHello`
  will run the program on all of the available hardware threads.
- There is a flag for oversubscribing the number of processors, but we will not be using it. In fact, on most of the hyper-threaded systems I have worked on, I have seen worse performance using hardware threads over just using the cores.
- The big difference between this and our usual program runs is that the same program is started on all of the processors simultaneously. So when we write an MPI program we need

MPI — Message Passing Interface

- In MPI a communicator is a collection of processes that can send messages to each other. `MPI_COMM_WORLD` is the total set of processes available. You can define subsets of processes to other communicators, only processes in the same communicator can pass messages between each other.
- As in most other parallel libraries, each process has a designation, here called a rank. The rank is simply a non-negative integer from 0 to the number of processes minus 1. Process 0 is considered the master process and has more functionality, for example, console input.
- The rank of a process is how we can alter the single program to run differently on different processes and how we can establish point-to-point communication.
- Usually process 0 is the master process that divides the work between the other processes and coordinates the results.
- Send and receive, how they work. General syntax. (PM-150-153)
- Data type constants. (PM-151)
- Conditions for a successful message pass. (PM-154)

MPI — Message Passing Interface

- Examples
  - MPI Hello World Example (MPIHello2). Trace and run.
  - Wildcards `MPI_ANY_SOURCE`, `MPI_ANY_TAG`.
  - Show deadlock with tags not matching or source/destination pair not matching.
- Getting status information with `MPI_Status` struct.
- Examples
  - MPI Hello World Example (MPIHello3). Trace and run.
- A few general notes on sending and receiving.
  - A send may buffer or it may block, depends on the MPI implementation.
  - If it buffers, MPI system will place the message into its own internal storage, and the call to `MPI_Send` will return.
  - If it blocks, it will wait until it can begin transmitting the message (that is, a receive picks it up), and the call to may not return immediately.
  - A receive always blocks until a matching message has been received. when a call to `MPI_Recv` returns, we know that there is a message stored in the receive buffer.
  - There are other send and receive functions that offer more flexibility.

MPI — Message Passing Interface

  - MPI requires that messages be non-overtaking. This means that if process q sends two messages to process r, then the first message sent by q must be available to r before the second message. However, there is no restriction on the arrival of messages sent from different processes.
  - If a process tries to receive a message and there's no matching send, then the process will block forever. That is, the process will hang (deadlock).
  - If a call to `MPI_Send` blocks and there's no matching receive, then the sending process can hang. If, on the other hand, a call to `MPI_Send` is buffered and there's no matching receive, then the message will be lost.

– Examples of simple point-to-point communication, π, what else?
  - PiApproxSerial — Serial with gettimeofday timer.
  - PiApproxRSMPI — Process 0 as coordinator.
  - PiApproxRSMPI2 — Process 0 as coordinator and worker.
  - PiApproxRSMPI3 — Wildcard additions.
  - PiApproxRSMPI4 — Added timing `MPI_Wtime()`.
  - PiApproxSaP — Doing serial and parallel timings, `MPI_Barrier`.

MPI — Message Passing Interface

– Collective Communications

- `MPI_Reduce` syntax (PM-172), (VE-V2-50) and operator syntax (PM-173)
  - `MPI_Reduce(senddata, recvdata, count, type, operator, root, comm);`
  - senddata, recvdata are pointers to the data buffers.
  - MPI that it will not by default overwrite the original data. Hence the senddata and recvdata should be two separate buffers, although this can be overridden.
  - They are the same data type.
  - count is the number of elements in the data.
  - type is the data type using the MPI constants for type designation.
  - operator is the reduction operation as outlined in (PM-173).
  - root is the process that the recvdata is sent to. Non-root processes do not receive the reduced data, they can actually leave the receive buffer undefined.
  - Example: PiApproxRSMPIReduce
  - Example: ArrayReduce
  - Example: ArrayReduce2
- `MPI_Bcast` syntax (PM-178). This is a broadcast communication from one process to all the processes in the communicator. A collective communication in which data belonging to a single process is sent to all of the processes in the communicator.
  - Example: PiApproxRSMPIBcast

MPI — Message Passing Interface

- `MPI_Allreduce` syntax (PM-176).
  - No destination process since it is copied to all processes.
  - This is essentially a reduce followed by a broadcast.
  - Since all processes receive the result they all need a receive buffer.
  - Standard Deviation Calculation.

$$\sigma = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_1 - \mu)^2} \qquad \text{where} \qquad \mu = \frac{1}{n}\sum_{i=1}^{n} x_1$$

  - Divide up the μ sum between processors, reduce to μ and broadcast back to all processors (via Allreduce), use same division of labor on the σ sum, let processor 0 do the final calculation.
  - Example: AllReduceStdDev
- `MPI_Scatter` syntax (PM-183).
  - Two blocks of thee arguments, buffer, count, type. Then source process and communicator, as expected.
  - Divides the data referenced by the send buffer into communicator size pieces, the first piece goes to process 0, the second to process 1, the third to process 2, and so on.
  - Note that send count should be the same as the receive count. The send count is the amount that is being sent to each process, not the total amount of data in the send buffer.

MPI — Message Passing Interface

- ▹ Standard Deviation Calculation.

$$\sigma = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_1 - \mu)^2} \qquad \text{where} \qquad \mu = \frac{1}{n}\sum_{i=1}^{n} x_1$$

  - ▹ This is the same algorithm as before except that we only send to each process the portion of the array it will be working on.
  - ▹ Example: AllReduceStdDevSct

- `MPI_Gather` syntax (PM-185).
  - ▹ Syntax is the same as with scatter, only difference is that there is a destination process instead of a source process.
  - ▹ Combines the data referenced by the send buffer into the receive buffer by blocks from each process in the communicator.
  - ▹ Note that send count should be the same as the receive count. The send count is the amount that is being sent to each process, not the total amount of data in the receive buffer.
  - ▹ Example: VecGather — Simple example of transferring array segments to the different processes, minor alteration, then gather to paste them ll back together.
  - ▹ Example: MatVecMult — Matrix-vector multiplication, $Mx = y$. Broadcast all of x but only 1/commsz rows of M and y. Note we are forcing the number of rows to be evenly divisible by the number of processes.

MPI — Message Passing Interface

  - ▹ Example: MatVecMult2 — Same example but with timing on the serial and parallel portions.
  - ▹ Example: MatVecMult3 — Small update that allows any number of rows and we pad both M and y with enough "garbage" entries so that the rows are evenly divisible by the number of processes. Common technique if storage for the problem size is not an issue.
  - ▹ Example: MatVecMult4 — Built on last example. Using x as the input of the last y output. Essentially the simulation of a dynamical system. Technique of gather on the last value of y, let process 0 copy y to x, then repeat.

- `MPI_Allgather` syntax (PM-190). Same as gather but no destination since all processes receive the gathered data.
  - ▹ Same as gather but no destination since all processes receive the gathered data.
  - ▹ Combines the data referenced by the send buffer into the receive buffer by blocks from each process in the communicator and broadcasts the result to all processes.
  - ▹ Again, in most cases the send count should be the same as the receive count. The send count is the amount that is being sent to each process, not the total amount of data in the receive buffer.
  - ▹ Example: MatVecMult5 — Built on last example, still a dynamical system example. This time the gather, transfer, and broadcast are replaced with a single allgather command.

MPI — Message Passing Interface

- Send and receive timing example to show system-wide latency in communication.
  - ▹ Example: SendRecvTiming — Fairly obvious outcome but nonetheless shows differences in data transfer rates on larger blocks of memory.

– Derived Datatypes (VE-V2-158)

- Basic structure and syntax. (VE-V2-167)
  - ▹ Declare `MPI_Datatype newtype;`
  - ▹ Define, several options here.
  - ▹ Commit the type `MPI_Type_commit`.
  - ▹ Free, `MPI_Type_free`
- The Contiguous data type.
  - ▹ For combining contiguous blocks of data (i.e. an array) into a single data type for sending.
  - ▹ Another way to send multiple data items in one send/receive or broadcast.
  - ▹ Syntax `MPI_Type_contiguous` (VE-V2-169)
  - ▹ Examples:
    ContiguousDT — Point-to-Point one transfer derived to native.
    ContiguousDT2 — Point-to-Point system transfer derived to derived.
    ContiguousDT3 — Broadcast derived to derived.

MPI — Message Passing Interface

- The Vector data type.
  - For sending non-contiguous data.
  - Can divide contiguous data into contiguous chunks by a stride (i.e. skip).
  - Common technique in data transfer between devices. For example, in computer graphics, sending data from the CPU to the GPU.
  - Syntax an layout `MPI_Type_vector` (VE-V2-171–172)
  - Examples:
    VectorDT — Point-to-Point transfer of blocks to processes.
    VectorDT2 — Point-to-Point simple example of extracting columns from a matrix represented as a contiguous array.
- The Subarray data type.
  - Given data that is stored contiguously but with the realization that it represents a higher dimensional array, this datatype will extract a subarray block and send to the receiving process as a contiguous block.
  - Many matrix and linear algebra applications work on submatrices of a matrix, so this technique would be useful in these situations.
  - Syntax an layout `MPI_Type_create_subarray` (VE-V2-176)
  - Examples:
    Subarray — Point-to-Point transfer of blocks to processes.

MPI — Message Passing Interface

- Indexed Type
  - ▹ This allows you to take contiguous data and transfer arbitrary portions of it to contiguous data on other processes using arrays of displacements and block sizes.
  - ▹ Extremely versatile, no predefined system of aggregation.
  - ▹ Syntax an layout `MPI_Type_indexed` (VE-V2-179)
  - ▹ Examples:
    IndexedDT — Point-to-Point transfer of blocks to processes.
- Structured Type
  - ▹ Creating an MPI type that corresponds to a C struct.
  - ▹ Syntax an layout `MPI_Type_create_struct` (VE-V2-182–183)
  - ▹ Examples:
    StructDT — Point-to-Point transfer of single struct. Definition using pointer arithmetic.
    StructDT2 — Point-to-Point transfer of single struct. Definition using `MPI_Get_address`.
    StructDT3 — Point-to-Point transfer of array of structs.
    StructDT4 — Broadcast transfer of array of structs.
    StructDT5 — Broadcast transfer of array of structs, this one using C-Style strings as a data item.