MPI – Message Passing Interface

- MPI is used for distributed memory parallelism (communication between nodes of a cluster)
- Interface specification with many implementations
- Portability was a major goal
- Widespread use in parallel scientific computing
- Six basic MPI functions
  - MPI_Init, MPI_Finalize,
  - MPI_Comm_size, MPI_Comm_rank,
  - MPI_Send, MPI_Recv
- Many other functions...
An MPI job consists of multiple processes running on multiple nodes, e.g., 1 or more processes per node.

Processes do not share memory. MPI provides functions for passing messages between processes.

If there are fewer processes than cores (usual case), then multiple threads are used in each process.
```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char *argv[])
{
    int size, rank;

    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    printf("Hello, world! from %d of %d\n", rank, size);

    MPI_Finalize();

    return 0;
}
```
Compiling MPI programs

Use `mpicc` which calls a compiler and links to appropriate libraries, etc.

```
mpicc      # wrapper around gcc
mpiicc     # wrapper around icc
mpiicpc    # wrapper around icpc
```
Running MPI programs

Use `mpirun -n <numproc> .... ./progname`

- `mpirun` will contact all nodes, set up communication between nodes, and run your program on all nodes.
- Usually MPI jobs are run on multiple nodes of a cluster (1 or more processes per node), and multiple threads per MPI process.
Running MPI programs on MIC

- We will use MPI to run multiple processes on a single coprocessor (although this is shared memory hardware)
- It is also possible to use MPI to run multiple processes on multiple coprocessors, and multiple coprocessors and CPU hosts, but we will not do this
- Run in native mode
  - compile on host using `mpiicc -mmic ...`
  - scp executable to coprocessor
  - log into coprocessor and use `mpirun`
- Run from the host
  - compile on host using `mpiicc -mmic ...`
  - scp executable to coprocessor
  - use `mpirun` but must also set `I_MPI_MIC`
    
    ```
    I_MPI_MIC=1 mpirun -host mic0 -n 60 ~/progname
    ```
Blocking Send and Recv

- **MPI_Send**
  - Function does not return until send buffer can be reused
  - Does not imply the message has been sent
  - Must be assured that the receiver posts a receive call

- **MPI_Recv**
  - Function does not return until recv buffer contains received message

- **Deadlock example (will deadlock if no buffering)**
  - Two processes, each performs
    
    ```
    Send(to other)
   Recv(from other)
    ```
```c
void main(int argc, char *argv[]) {
    int size, rank;
    double sum;
    double sendbuf[MSGLEN];
    double recvbuf[MSGLEN];
    MPI_Status status;

    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    double val = (double) rank;
    sendbuf[0] = (double) rank;

    MPI_Send(sendbuf, MSGLEN, MPI_DOUBLE, (rank+1)%size, 0, MPI_COMM_WORLD);

    MPI_Recv(recvbuf, MSGLEN, MPI_DOUBLE, (rank-1+size)%size, 0, MPI_COMM_WORLD, &status);

    printf("Recv on node %d is %f\n", rank, recvbuf[0]);

    MPI_Finalize();
}
```
Non-blocking Send and Recv

- **MPI_Isend**
  - Function returns immediately; the data may be buffered, and the message may not be sent yet

- **MPI_Irecv**
  - Function returns immediately; the message has not necessarily arrived

- **MPI_Wait**
  - Block until Isend/Irecv completes (buffer can only be used at this point)

  - Allows overlap of communication with computation
  - Easier to avoid deadlocks than using blocking calls
  - Can combine blocking and non-blocking calls
MPI_Request request;
MPI_Status status;

MPI_Irecv(recvbuf, length, MPI_CHAR, (rank-1+size)%size, 
          0, MPI_COMM_WORLD, &request);

MPI_Send(sendbuf, length, MPI_CHAR, (rank+1)%size, 
         0, MPI_COMM_WORLD);

MPI_Wait(&request, &status);
Measured latency and bandwidth

2 CPU Xeon host (bidirectional bandwidth, one pair of processes)

- latency is around 1 microsecond
- bandwidth is comparable to single thread memory bandwidth (MPI is using shared memory in this case)
- how does this curve change when multiple pairs are running?
- more efficient to send long messages than short messages (group your messages together if possible, unless you are pipelining computations)
2 CPU Xeon host (bidirectional bandwidth, one pair of processes)
Measured latency and bandwidth

Xeon Phi coprocessor (bidirectional bandwidth, one pair of processes)
Eager and Rendezvous protocols

- Eager protocol: if the message is short, it is sent immediately and buffered on the receiver’s side. On the receiver, the message is copied to the receive buffer when the receive is posted.
- Rendezvous protocol: if the message is long, a short message is first sent to the receiver to indicate that a send has been posted. The receiver sends the address of the receive buffer. The sender then sends the actual message.
- Kink in timing graph is due to the switchover from eager to rendezvous protocols.
MPI process pinning

- When using multiple MPI processes per node, it may be desirable to pin the processes to a socket, or to a set of cores.
- Each MPI process may use multiple threads (within a socket or set of cores).
- Define a *domain* to be a non-overlapping set of logical cores.
- A MPI process can be pinned to a domain; the threads in a process run on the logical cores of the domain (use `KMP_AFFINITY` to pin threads).
- Pinning can be accomplished with environment variables (also with the `mpirun` command, etc.).
- Set `I_MPI_DEBUG=4` to see how processes are pinned.
MPI process pinning with environment variables

**I_MPI_PIN_DOMAIN** can take the following values:

- core
- socket
- numa
- node
- cache1
- cache2
- cache3
- numerical value, which is the size of the domain
- **omp** which sets the domain size to **OMP_NUM_THREADS**
MPI process pinning with environment variables

I_MPI_PIN_ORDER can take the following values:

- scatter
- compact
- spread
- bunch

Reference: https://software.intel.com/en-us/node/528819
Write a code to measure the maximum bandwidth between $2p$ processes on the coprocessor, for different message lengths, and for different values of $p$, e.g., 1, 2, 4, 8, 16, 30. (Use communication between pairs of processes.)

At what value of $p$ does the aggregate bandwidth for long messages no longer improve?

For this value of $p$, plot the average time for sending a single message as a function of message length. Use lengths 1 byte, 2 bytes, 4 bytes, etc., up to 16 MB. Also plot the bandwidth (GB/s). Use log-log axes for both plots.