COMP 4901Q: High Performance Computing (HPC)

Lecture 9: Introduction to MPI

Instructor: Shaohuai SHI (shaohuais@cse.ust.hk)

Teaching assistants: Mingkai TANG (mtangag@connect.ust.hk)

Yazhou XING (yxingag@connect.ust.hk)

Course website: https://course.cse.ust.hk/comp4901q/

Outline

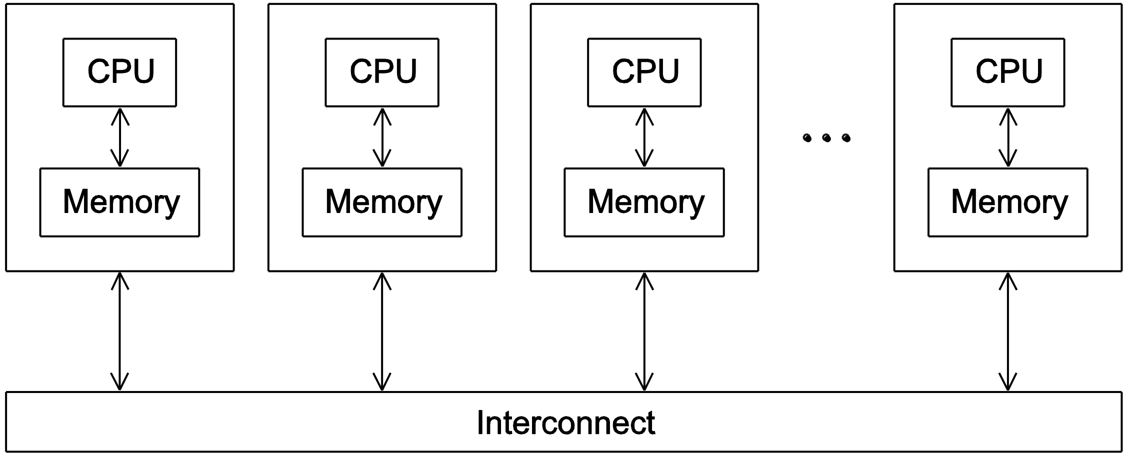
- MPI Overview
  - Basic Concepts
  - Six Core Functions
- Fundamentals of MPI
  - Initialization and Finalization
  - Starting from Simple Examples

Overview of Message Passing Interface (MPI)

- Shared-memory vs. distributed memory
- What is MPI?
- Basic MPI concepts
- Six core functions of MPI

Distributed-Memory System

- Each CPU has its own private memory space
- The CPUs communicate through a high-speed network (e.g., Ethernet, InfiniBand) by sending and receiving messages
  - Pro: The number of CPUs can be very large.
  - Con: The throughput and latency are lower than those of shared-memory systems
- MPI is the most popular programming technique for distributed memory system
- New techniques include Hadoop MapReduce and Apache Spark



What is MPI?

- MPI is a widely used standard for writing message-passing programs
  - http://www.mpi-forum.org
  - It's a specification, not an implementation
- It is implemented as a library, not a programming language
  - Examples include MPICH, Open MPI, Microsoft MPI (MS-MPI), Intel MPI
  - MPI has been supported by C, C++, Fortran, Java, Python, etc.
- History of MPI
  - The first MPI standard, called MPI-1 was completed in May 1994.
    - MPI-1.1 (1995), MPI-1.2 (1997), MPI_1.3 (1998)
  - The second MPI standard, MPI-2, was completed in 1997.
    - MPI-2.1 (2008), MPI-2.2 (2009)
  - MPI-3 was approved in 2012 (more than 800 pages).
  - MPI-4 in 2021 (1,139 pages)

MPI Process and Message Passing

- An MPI program consists of many processes
  - These processes are executed on a set of physical processors which exchange data (by internal bus or a network).

- The processes executing in parallel have separate address spaces.
  - Assume your program has a statement "y = a + b".
  - When process A and process B both execute the above statement, each process has its own set of variables {a, b, y}.

- Message-passing: a portion of one process's address space is copied into another process's address space
  - "message" means "data"
  - "message-passing" means "data transfer"
  - It's usually done by send operation and receive operation

Message Passing

- Sender needs to specify:
  - Who is the receiver (or destination)?
  - How to define the message?

- Receiver needs to specify:
  - Where to store the incoming message?
  - Who is the sender, or where to store the "sender" information?

- Matching between sender and receiver
  - A pair of sender and receiver can use "tag of a message" to control which message should be received

How to Identify a Process?

- Communicator
  - In MPI, a set of processes can form a "group", identified by a communicator
  - A default communicator MPI_COMM_WORLD includes all processes

- Rank
  - If a group contains n processes, then its processes are identified within the group by ranks, which are integers from 0 to n-1
  - Normally, a process is identified by its rank in MPI_COMM_WORLD

How to Define a Message?

- A simple solution: (address, length)
  - Message is stored continuously in memory space
  - “address” refers to the starting memory address of the message
  - “length” refers to the length of the message (in bytes)

- MPI’s solution: (address, count, datatype)
  - Message is an array of items with the same type
  - “address” refers to the starting memory address
  - “count” refers to the total number of items
  - “datatype” refers to the type of the item, which can be a simple elementary data type such as integer, floating-point number, or a complex data type defined by the user

Types of Communications

- Point-to-Point Communications
  - Data is explicitly sent by one process and received by another.

- Collective Communications
  - The communications involve a group or groups of processes.

- One-sided Communications
  - A kind of "Remote Memory Access"
  - One process specifies all communication parameters, both for the sending side and for the receiving side.

Six Core MPI Functions

Function name	Description
MPI_Init	Initialize MPI
MPI_Comm_size	Return the number of processes
MPI_Comm_rank	Return the rank of this process
MPI_Send	Send a message
MPI_Recv	Receive a message
MPI_Finalize	Terminate MPI

Fundamentals of MPI

- Example 1: Compute the value of π

- Example 2: Matrix-Vector multiplication

- Performance measurement

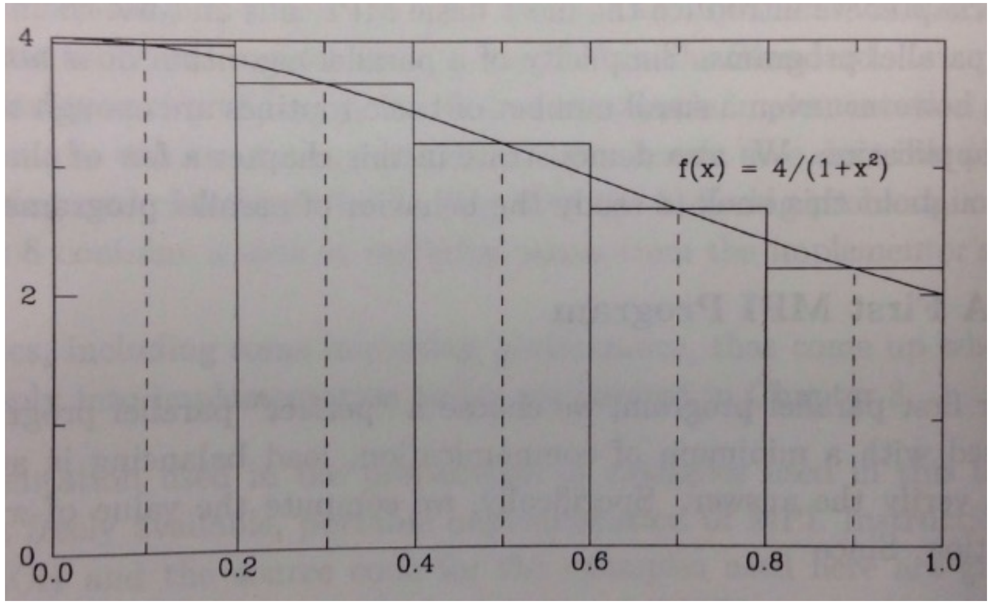
- Communicators

Example 1: Compute the Value of π

- $\int_0^1 \frac{1}{1+x^2} dx = \arctan(x)\,|_0^1 = \arctan(1) - \arctan(0) = \frac{\pi}{4}$

$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$

- Numerical solution:
  - Divide the interval from 0 to 1 into n subintervals
  - Add up the areas of the rectangles
  - The figure shows the case of n = 5



Our First MPI Program: main()

```c
#include "mpi.h"
#include <stdio.h>
#include <math.h>

int main( int argc, char *argv[] )
{
    int n, myid, numprocs, i;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);

    while (1) {
        /* see next page */
    }

    MPI_Finalize();
    return 0;
}
```

Our First MPI Program: the while() body

```
    while (1) {
        if (myid == 0) {
            printf("Enter the number of intervals: (0 quits) ");
            scanf("%d",&n);
        }
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
        if (n == 0)
            break;
        else {
            h   = 1.0 / (double) n;
            sum = 0.0;
            for (i = myid + 1; i <= n; i += numprocs) {
                x = h * ((double)i - 0.5);
                sum += (4.0 / (1.0 + x*x));
            }
            mypi = h * sum;
            MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
            if (myid == 0)
                printf("pi is approximately %.16f, Error is %.16f\n", pi, fabs(pi - PI25DT));
        }
    }
```

MPI in C/C++ Language

- Need to add mpi.h header file.

- Identifiers defined by MPI start with "MPI_".

- First letter following underscore is uppercase.
  - For function names and MPI-defined types.
  - Helps to avoid confusion.

MPI Initialization and Finalization

- MPI_Init( )
  - Tells MPI to do all the necessary setup.

```
int MPI_Init(int *argc_p, char **argv_p);
```

- MPI_Finalize( )
  - Tells MPI we're done, so clean up anything allocated for this program.

```
int MPI_Finalize(void);
```

MPI Initialization and Finalization

- int MPI_Comm_size(MPI_Comm comm, int* comm_sz_p)
  - Tells us the number of processes in a group (communicator)

- int MPI_Comm_rank(MPI_Comm comm, int* my_rank_p)
  - Tells us the rank of the current process

Message Broadcast in MPI

- It is a common pattern to broadcast a message within a process group
  - collective communication

- int MPI_Bcast(void *buf,
  int count,
  MPI_Datatype datatype,
  int root,
  MPI_Comm comm)
  - <buf, count, datatype> specify the "message"
  - root specifies the rank of the source process
  - comm specifies the process group

- Example:
  - MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

MPI Data Types

MPI datatype	C datatype	C++ datatype
MPI::CHAR	char	char
MPI::SHORT	signed short	signed short
MPI::INT	signed int	signed int
MPI::LONG	signed long	signed long
MPI::LONG_LONG	signed long long	signed long long
MPI::SIGNED_CHAR	signed char	signed char
MPI::UNSIGNED_CHAR	unsigned char	unsigned char
MPI::UNSIGNED_SHORT	unsigned short	unsigned short
MPI::UNSIGNED_LONG	unsigned long	unsigned long int
MPI::FLOAT	float	float
MPI::DOUBLE	double	double
MPI::LONG_DOUBLE	long double	long double
MPI::BOOL		bool
MPI::COMPLEX		Complex<float>
MPI::DOUBLE_COMPLEX		Complex<double>
MPI::LONG_DOUBLE_COMPLEX		Complex<long double>
MPI::BYTE		
MPI::PACKED		

Data Reduction in MPI

- It is another common pattern to reduce a set of messages into a single message within a process group
  - Collective communication
  - Examples of operations include max, min, sum, product, etc.
  - Will be discussed in more detail later.
- int MPI_Reduce(void *sendbuf, void *recvbuf, int count,
  MPI_Datatype datatype, MPI_Op op,
  int root, MPI_Comm comm)
  - <sendbuf, count, datatype> specify the "message"
  - <recvbuf, root> specify where to store the reduced result (i.e., recvbuf at process root)
  - op specify the reduction operation
  - comm specifies the process group
- Example:
  - MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD)

MPI Reduction Operators

Operation Value	Meaning
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bitwise and
MPI_LOR	Logical or
MPI_BOR	Bitwise or
MPI_LXOR	Logical exclusive or
MPI_BXOR	Bitwise exclusive or
MPI_MAXLOC	Maximum and location of maximum
MPI_MINLOC	Minimum and location of minimum

Compilation and Execution

- To compile MPI in C language:
  - $mpicc -o mpi_pi mpi_pi.c
- To execute MPI program using a single (multi-core) computer:
  - $mpiexec -n 4 ./mpi_pi
- To execute MPI program using a cluster of computers
  - Create a text file (e.g., my_cluster) that contains the names of the computers in the cluster.
  - $mpiexec -f my_cluster -n 16 ./mpi_pi

Timing MPI Programs

- How to measure the execution time of a piece of program?
  - Different operating systems have different function calls
- MPI provides a platform independent solution
- double MPI_Wtime()
  - It returns the time (in seconds) since an arbitrary time in the past
  - Call it at the beginning and end of a program segment and subtract the values
- double MPI_Wtick()
  - It returns the resolution of MPI_Wtime() in seconds

Running Example

mpimachine:
node1
node2
node3
node4

[shaohuais@node1 mpi]$ mpiexec -f mpimachine -n 4 ./a.out
Enter the number of intervals: (0 quits) 1000000000
pi is approximately 3.141592653589612 8, Error is 0.0000000000001803
It takes 3.878083 seconds.

[shaohuais@node1 mpi]$ mpiexec -f mpimachine -n 2 ./a.out
Enter the number of intervals: (0 quits) 1000000000
pi is approximately 3.1415926535905170, Error is 0.0000000000007239
It takes 7.729113 seconds.

[shaohuais@node1 mpi]$ mpiexec -f mpimachine -n 1 ./a.out
Enter the number of intervals: (0 quits) 1000000000
pi is approximately 3.1415926535921401, Error is 0.0000000000023470
It takes 15.485958 seconds.

# of processes	1	2	4
Execution Time (s)	15.486	7.729	3.878

Example 2: Matrix-Vector Multiplication

$A \in \mathbb{R}^{m \times n}$ $\quad\quad$ $\boldsymbol{x} \in \mathbb{R}^{n \times 1}$

$\boldsymbol{y} = A\boldsymbol{x} \in \mathbb{R}^{m \times 1}$

Matrix-Vector Multiplication

$$
\begin{bmatrix}
a_{00} & a_{01} & \cdots & a_{0,n-1} \\
a_{10} & a_{11} & \cdots & a_{1,n-1} \\
\vdots & \vdots & & \vdots \\
a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\
\vdots & \vdots & & \vdots \\
a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1}
\end{bmatrix}
\begin{bmatrix}
x_0 \\ x_1 \\ \vdots \\ x_{n-1}
\end{bmatrix}
=
\begin{bmatrix}
y_0 \\ y_1 \\ \vdots \\ y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1} \\ \vdots \\ y_{m-1}
\end{bmatrix}
$$

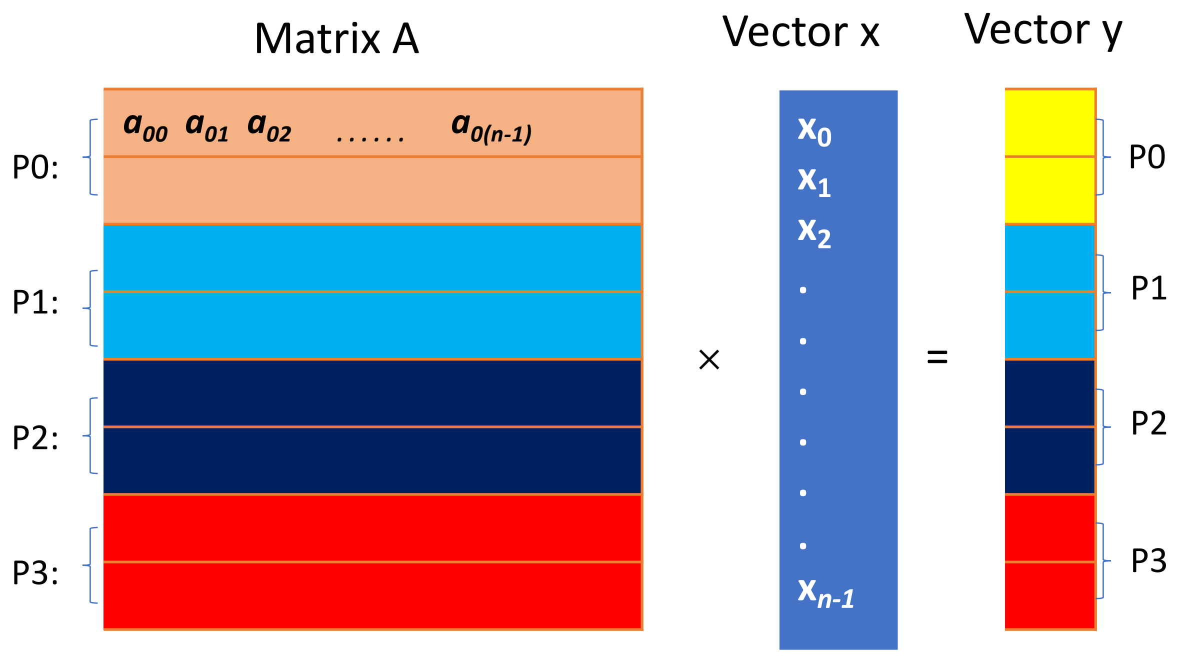
```c
void Mat_vect_mul(double A[], double x[], double y[], int m, int n) {
    int i, j;
    /* For each row of A */
    for (i = 0; i < m; i++) {
        /* From dot product of i-th row with x */
        y[i] = 0.0;
        for (j = 0; j < n; j++) {
            y[i] += A[i*n+j] * x[j];
        }
    }
}
```

Serial Matrix-Vector Multiplication

- Matrix A is stored as a one-dimensional array with m x n elements.
- A[i][j] can be accessed by A[i * n + j].

Row-wise 1-D Partitioning

- Given p processes, Matrix A (m x n) is partitioned into p smaller matrices, each with dimension (m/p x n).
  - For simplicity, we assume p divides m (or, m is divisible by p).



Parallel Matrix-Vector Multiplication: Framework

- Assumptions
  - A total of p processes
  - Matrix A (m x n) and vector x (n x 1) are created at process 0
    - called "master process" because it coordinates the work of other processes (i.e., "slave processes")
- Message passing:
  - Process 0 will send (p-1) sub-matrices to corresponding processes
  - Process 0 will send vector x to all other p-1 processes
- Calculations:
  - Each process carries out its own matrix-vector multiplication
- Message passing:
  - Processes 1 to (p-1) send the results (i.e., part of vector y) back to process 0

Parallel Matrix-Vector Multiplication: Code

```c
int main(int argc, char** argv)
{
    int m = 0, n = 0, myid, numprocs, srow = 0, i;
    double *A, *x, *y;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    if(myid == 0) {
        while ( m <= 0 || n <= 0 || m % numprocs != 0) {
            printf("Please input positive integers m and n: ");
            scanf("%d %d", &m, &n);
        }
    }
    MPI_Bcast(&m, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    srow = m / numprocs;
    if (myid == 0) {
        /* master code */
    } else {
        /* slave code */
    }
    free(A); free(x); free(y);
    MPI_Finalize();
    return 0;
}
```

Key parameters:

(m, n): dimension of the matrix

myid: who am I?

numprocs: total # of processes

srow: # of rows assigned to each process

A: to store the data of the matrix (for process 0) or sub-matrix (for slave processes)

x: to store vector x

y: to store vector y (for process 0) or sub-vector of y (for slaves)

Code of Master Process

```
    if (myid == 0) {
        /* master code */
        /* allocate memory for matrix A, vectors x and y, and initialize them */
        A = (double*) malloc( m * n * sizeof(double) );
        x = (double*) malloc(n * sizeof(double) );
        y = (double*) malloc(m * sizeof(double) );
        init_array(A, m * n); // Remark: this function is written by ourselves
        init_array(x, n);

        /* broadcast vector x */
        MPI_Bcast(x, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);

        /* send sub-matrices to other processes */
        for(i = 1; i < numprocs; i++)
            MPI_Send(A+i*srow*n, srow * n, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);

        /* perform its own calculation for the 1st sub-matrix */
        Mat_vect_mul(A, x, y, srow, n);  // Remark: this function is written by ourselves

        /* collect results from other processes */
        for(i = 1; i < numprocs; i++)
            MPI_Recv(y+i*srow, srow, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Code of Slave Process

```
} else {
    /* slave code */
    /* allocate memory for sub-matrix A, vector x, and sub-sector y */
    A = (double*) malloc( srow * n * sizeof(double) );
    x = (double*) malloc( n * sizeof(double) );
    y = (double*) malloc( srow * sizeof(double) );

    /* receive x from process 0 */
    MPI_Bcast(x, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    /* receive sub-matrix from process 0 */
    MPI_Recv(A, srow * n, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    /* perform the calculation on the sub-matrix */
    Mat_vect_mul(A, x, y, srow, n);

    /* send the results to process 0 */
    MPI_Send(y, srow, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
}
```

Message Passing: MPI_Send and MPI_Recv

- Point-to-point communications
  - A sender process calls MPI_Send() AND a receiver process calls MPI_Recv().

- int MPI_Send(void* buf, int count, MPI_Datatype datatype,
  int dest, int tag, MPI_Comm comm);
  - <buf, count, datatype> specify the message
  - dest specifies the rank of the process that should receive the message
  - tag (an nonnegative integer) is used to distinguish messages
  - comm specifies the process group

- Examples:
  - MPI_Send(A+i*srow*n, srow * n, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
  - MPI_Send(y, srow, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);

Message Passing: MPI_Send and MPI_Recv

- int MPI_Recv(void* buf, int maxsize, MPI_Datatype datatype,

  int source, int tag, MPI_Comm comm, MPI_Status* status_p);
  - <buf, maxsize, datatype> specify where to store the message
  - source specifies the rank of the process from which the message should be received
  - tag should match the "tag" specified by the sender
  - comm specifies the process group
  - status_p can tell us more about the incoming message. Use MPI_STATUS_IGNORE if you don't care.

- Examples
  - MPI_Recv(A, srow * n, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  - MPI_Recv(y+i*srow, srow, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

Message Matching

- Consider process q calls MPI_Send with
  - MPI_Send(q_buf, q_size, q_type, dest, q_tag, q_comm);
- and process r calls MPI_Recv with
  - MPI_Recv(r_buf, r_size, r_type, src, r_tag, r_comm);

- Then r is able to receive the message from q if the following conditions are all satisfied:
  - r_comm = q_comm
  - r_tag = q_tag or MPI_ANY_TAG
  - dest = r
  - src = q or MPI_ANY_SOURCE
  - The buffers in q and r should be compatible
  - E.g., q_type = r_type and r_size >= q_size

What is MPI_Status?

- A receiver may ask the following questions:
  - What is the amount of data in the message?
  - Who is the source? (if MPI_ANY_SOURCE is used)
  - What is the tag of the message? (if MPI_ANY_TAG is used)

- MPI defines a structure named MPI_Status to hold the above three pieces of information.

- E.g., MPI_Status status;
  - status.MPI_SOURCE gives us the “src” information
  - status.MPI_TAG gives us the “tag” information
  - To get the number of data items in the message, do the followings:
    - int count;
    - MPI_Get_count(&status, r_type, &count);

Input and Output

- How to read data from input (such as keyboard) and how to write data to output (such as screen)?

- In MPI, most implementations allow all processes to full access stdout and stderr
  - If multiple processes are accessing the same output, the order will be unpredictable

- But, only process 0 in MPI_COMM_WORLD is allowed to access stdin

Reading List

- Thomas Sterling, Matthew Anderson and Maciej Brodowicz (2018), "High Performance Computing: Modern Systems and Practices," Morgan Kaufmann, **Chapter 8**. [PDF: https://www.sciencedirect.com/book/9780124201583/high-performance-computing]