

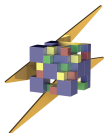
Lecture 19

MPI

이재진

서울대학교 컴퓨터공학부

<http://aces.snu.ac.kr>



THUNDER Research Group
Seoul National University
서울대학교 천둥 연구실



MPI

- Message Passing Interface
 - Provide a standard for writing message passing programs
 - Portable, efficient, flexible
- Specification of message passing libraries for developers and users
 - Specifies what such a library should be
 - API for such libraries
 - MPI standard
 - <http://www.mpi-forum.org>
 - <http://www-unix.mcs.anl.gov/mpi/>
- Language bindings
 - C, C++, and FORTRAN



MPI (cont'd)

- For parallel computers, clusters, and heterogeneous networks
- Designed to provide access to advanced parallel hardware for
 - End users
 - Library writers
 - Tool developers



MPI (cont'd)

- 1994, MPI 1.0
- 1995, MPI 1.1, revision and clarification to MPI 1.0
 - Major milestone
 - C, FORTRAN
- 1997, MPI 1.2
 - Corrections and clarifications to MPI 1.1
- 1997, MPI 2.0
 - Major extension (and clarifications) to MPI 1.1
 - C++, C, FORTRAN
 - Partially implemented in most libraries
 - A few full implementations (e.g. ANL MPICH2)
- 2012, MPI 3.0
- 2015, MPI 3.1



MPI (cont'd)

- De facto standard for parallel computing
 - Industry standard
- Portability
 - Supported on virtually all HPC platforms
- Performance
 - So far the best
 - High performance and scalability
- Rich functionality
 - MPI 1.1 – 125 functions
 - MPI 2 – 152 functions



Message Passing Programming Model

- MPI is for communication among processes
 - Each process has a separate address space
- Interprocess communication consists of
 - Synchronization
 - Movement of data from one process's address space to another's
- The exchange of data is cooperative between processes
 - Data is explicitly sent by one process and received by another
 - Point-to-point communications
 - Any change in the receiving process's memory is made with the receiver's explicit participation



Message Passing Programming Model (cont'd)

- Number of CPUs is statically determined
 - MPI 2 specifies dynamic process creation, but not available in most implementations
 - In general, we assume one-to-one mapping of MPI processes to processors
- SPMD
 - All processes the same program, but act on different data
- MPMD
 - Multiple Program Multiple Data
 - Each process may be a different program



MPI Program Structure

- MPI include file
- MPI environment initialization
- Message passing calls
- MPI environment termination



Hello, World

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

int main(int argc, char **argv)
{
    int my_rank, size;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("Hello, I am %d of %d\n",
           my_rank, size);
    MPI_Finalize();
    return 0;
}
```

```
Hello, I am 0 of 4
Hello, I am 2 of 4
Hello, I am 1 of 4
Hello, I am 3 of 4
```



Running MPI Programs

- `mpiexec <args>` is part of MPI-2, as a recommendation, but not a requirement
- In general, starting an MPI program is dependent on the implementation of MPI you are using
- For example,
 - `mpicc -o hello hello.c`
 - `mpirun np 4 ./hello`
 - 4 processes but not necessarily 4 processors



MPI Naming Conventions

- All names have MPI_ prefix
- In C: mixed uppercase/lowercase
 - `ierr = MPI_XXxx(arg1, arg2, ...);`
- MPI constants are all uppercase
 - `MPI_COMM_WORLD, MPI_SUCCESS, MPI_DOUBLE, MPI_SUM, ...`



Error Handling

- An error causes all processes to abort
 - The user can cause routines to return with an error code instead
 - In C++, exceptions are thrown (MPI-2)
- `ierr = MPI_...;`
 - If `ierr == MPI_SUCCESS`, everything is fine



Environment

- The elements of an application are
 - N processes numbered through $N - 1$
 - Communication paths between them
 - Communicator is a group of processes that can communicate with one another
 - Most MPI routines require a communicator argument to specify the collection of processes the communication is based on



Environment (cont'd)

- All processes in the computation form the communicator `MPI_COMM_WORLD`
 - `MPI_COMM_WORLD` is pre-defined by MPI, available anywhere
 - Can create subgroups/subcommunicators within `MPI_COMM_WORLD`
- `MPI_COMM_SIZE()`: the number of processes (N)
- `MPI_COMM_RANK()`: the specific number of a process
 - Ranks are used to specify source and destination of communications
- A process may belong to different communicators, and have different ranks in different communicators



MPI Initialization

- `int MPI_Init(int *argc, char ***argv)`
 - Initializes MPI environment
 - Must be called before any other MPI routine (so put it at the beginning of code) except `MPI_Initialized()` routine.
 - Can be called only once; subsequent calls are erroneous.

- `int MPI_Initialized(int *flag)`
 - Checks if `MPI_Init()` is called



Termination

- `int MPI_Finalize(void)`
 - Cleans up MPI environment
 - Must be called before exits
 - No other MPI routine can be called after this call
 - Even `MPI_INIT()`
 - Exceptions
 - `MPI_Initialized()`, `MPI_Get_version()`, `MPI_Finalized()`
- `int MPI_Abort(MPI_Comm comm, int errcode)`
 - Abnormal termination
 - Makes a best attempt to abort all tasks



MPI Initialization and Termination

```
int main(int argc, char ** argv)
{
    MPI_Init(&argc, &argv);

    int flag;

    MPI_Initialized(&flag);
    if(flag != 0) ...
    ... ..
    MPI_Finalize();

    return 0;
}
```



MPI Communications

- Point-to-point communications
 - Involves a sender and a receiver, one process to another process
- Collective communications
 - All processes within a communicator participate in communication
 - Barrier, reduction operations, gather, ...



MPI Data Types

- Recursively defined as
 - Predefined, corresponding to a data type from the language
 - e.g., MPI_INT, MPI_DOUBLE_PRECISION
 - A contiguous array of MPI datatypes
 - A strided block of datatypes
 - An indexed array of blocks of datatypes
 - An arbitrary structure of datatypes
- There are MPI functions to construct custom datatypes
 - An array of (int, float) pairs
 - A row of a matrix stored column-wise



Basic MPI Data Types

MPI datatype	C datatype
<code>MPI_CHAR</code>	<code>signed char</code>
<code>MPI_SHORT</code>	<code>signed short</code>
<code>MPI_INT</code>	<code>signed int</code>
<code>MPI_LONG</code>	<code>signed long</code>
<code>MPI_UNSIGNED_CHAR</code>	<code>unsigned char</code>
<code>MPI_UNSIGNED_SHORT</code>	<code>unsigned short</code>
<code>MPI_UNSIGNED</code>	<code>unsigned int</code>
<code>MPI_UNSIGNED_LONG</code>	<code>unsigned long int</code>
<code>MPI_DOUBLE</code>	<code>double</code>
<code>MPI_FLOAT</code>	<code>float</code>
<code>MPI_LONG_DOUBLE</code>	<code>long double</code>
<code>MPI_BYTE</code>	
<code>MPI_PACKED</code>	



MPI Tags

- Messages are sent with an accompanying user-defined integer tag
 - To assist the receiving process in identifying the message
 - Messages can be screened at the receiving end by specifying a specific tag
 - Not screened by specifying MPI_ANY_TAG



Blocking Send

- `int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
- Buf: address of send buffer
 - Where the message is
- Count: number of data items
- Datatype: type of data items
- Dest: rank of destination process
- Tag: message tag
- Comm: communicator, usually `MPI_COMM_WORLD`



Blocking Send (cont'd)

- MPI_Send is blocking
 - When this function returns, the data has been delivered to the system
 - User can safely access or overwrite the send buffer
 - The message may not have been received by the destination process



Blocking Receive

- `int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`
- Buf: address of receive buffer
- Count: number of elements in receive buffer
- Datatype: data type of receive buffer elements
- Source: rank of source process or `MPI_ANY_SOURCE`
- Tag: message tag or `MPI_ANY_TAG`
- Status: status object containing additional information of received message



Blocking Receive (cont'd)

- MPI_Recv is blocking
 - Waits until a matching message is received from the system
 - After it returns, the data is in the buffer and ready for use
- Receiving fewer than count occurrences of datatype is OK, but receiving more is an error



MPI_Recv Status

- The C MPI_Status structure has 3 members
 - status.MPI_TAG – tag of received message
 - status.MPI_SOURCE – source rank of message
 - status.MPI_ERROR – error code

- `int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)`
 - Length of received message



Example

```
// Send an array from process 0 to 1.

int main(int argc, char **argv) {
    int rank, size;
    double a[10];
    int tag = 1001;
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) {
        for (int i=0; i<10 ; i++)
            a[i] = i;
        MPI_Send(a, 10, MPI_DOUBLE, 1, tag, MPI_COMM_WORLD);
    } else if (rank == 1) {
        MPI_Recv(a, 10, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD,
                &status);
    }

    MPI_Finalize();
}
```



MPI is Simple

- Many parallel programs can be written using just the six functions below
- MPI_INIT
- MPI_FINALIZE
- MPI_COMM_SIZE
- MPI_COMM_RANK
- MPI_SEND
- MPI_RECV



Deadlocks

- Must be careful to avoid deadlock
 - Carefully sequence all messages

Process 0

```
MPI_Send(..., 1, 1001, ...)  
MPI_Recv(..., 1, 1001, ...)
```

Process 0

```
MPI_Send(..., 1, 1001, ...)  
MPI_Send(..., 1, 1002, ...)
```

Process 0

```
MPI_Recv(..., 1, 1001, ...)  
MPI_Send(..., 1, 1001, ...)
```

Process 1

```
MPI_Send(..., 0, 1001, ...)  
MPI_Recv(..., 0, 1001, ...)
```

Process 1

```
MPI_Recv(..., 0, 1002, ...)  
MPI_Recv(..., 0, 1001, ...)
```

Process 1

```
MPI_Recv(..., 0, 1001, ...)  
MPI_Send(..., 0, 1001, ...)
```



Buffering

- Send and matching receive operations may not be synchronized in reality
 - Implementation dependent
 - Typically, a system buffer holds data in transit



Communication Modes for Send

- Standard mode: MPI_Send
 - System decides buffering
 - Small messages --> buffering; large messages --> no buffering
- Buffered mode: MPI_Bsend
 - Message is copied to buffer, then returns
 - User can provide the buffer
- Synchronous mode: MPI_Ssend
 - Block, no buffering
- Ready mode: MPI_Rsend
 - Can be used only if a matching receive is already posted
 - Otherwise erroneous
- Only one MPI_Recv



Delivery Order

- If a sender sends two messages in succession to the same destination, and both match the same receive, then this receive will receive the first message
- If a receiver posts two receives in succession and both match the same message, then the first receive will receive it
- If a receive matches two messages from two different senders, the receive receives either one

```
if(rank==0) {  
    MPI_Bsend(buf1, count, MPI_DOUBLE, 1, tag, comm) ;  
    MPI_Bsend(buf2, count, MPI_DOUBLE, 1, tag, comm) ;  
} else if(rank==1) {  
    MPI_Recv(buf1, count, MPI_DOUBLE, 0, MPI_ANY_TAG, comm, &status) ;  
    MPI_Recv(buf2, count, MPI_DOUBLE, 0, tag, comm, &status) ;  
}
```



Send-recv

- To avoid deadlock
 - Combines send and receive in one call
 - Executing a non-blocking send and a non-blocking recv, and then wait for them to complete
- ```
int MPI_Sendrecv(void *sendbuf, int sendcount,
MPI_Datatype sendtype, int dest, int sendtag,
void *recvbuf, int recvcount, MPI_Datatype
recvtype, int source, int recvtag, MPI_Comm comm,
MPI_Status *status)
```



# Non-blocking Communication

- Sender proceeds even if message has not been sent
- Receiver proceeds even if message has not be received
- Either can check on the status of the message
- Avoids idle time and (some) deadlocks
- Overlap computation with communication
  
- Split communication operations into two parts
  - Initiates the operation (non-blocking)
  - Waits for the operation to complete



## Non-blocking Communication (cont'd)

- `MPI_Send(buf, count, type, dest, tag, comm)`
  - `MPI_Isend(buf, count, type, dest, tag, comm, &request)`
  - `MPI_Wait(&request, &status)`
- `MPI_Recv(buf, count, type, dest, tag, comm, status)`
  - `MPI_Irecv(buf, count, type, dest, tag, comm, &request)`
  - `MPI_Wait(&request, &status)`



# Non-blocking Communication (cont'd)

- Checking completion
  - Use the request record
- `MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)`
  - To check for completion
  - `MPI_TestAll`, `MPI_TestAny`, `MPI_TestSome`
- `MPI_Wait(MPI_Request *request, MPI_Status *status)`
  - To block for completion (non-blocking --> blocking)
  - `MPI_WaitAll`, `MPI_WaitAny`, `MPI_WaitSome`



## Non-blocking Communication (cont'd)

```
#define MYTAG 1000
#define WORLD MPI_COMM_WORLD
MPI_Request request;
MPI_Status status;
...
if(rank==0) {
 MPI_Irecv(B, 100, MPI_DOUBLE, 1, MYTAG, WORLD, &request);
 MPI_Send(A, 100, MPI_DOUBLE, 1, MYTAG, WORLD);
 MPI_Wait(&request, &status);
} else if(rank==1) {
 MPI_Irecv(B, 100, MPI_DOUBLE, 0, MYTAG, WORLD, &request);
 MPI_Send(A, 100, MPI_DOUBLE, 0, MYTAG, WORLD);
 MPI_Wait(&request, &status);
}
```



## Non-blocking Communication (cont'd)

```
#define MYTAG 1000
#define WORLD MPI_COMM_WORLD
MPI_Request request;
MPI_Status status;
...
if(rank==0) {
 MPI_Isend(A, 100, MPI_DOUBLE, 1, MYTAG, WORLD, &request);
 MPI_Recv(B, 100, MPI_DOUBLE, 1, MYTAG, WORLD, &status);
 MPI_Wait(&request, &status);
} else if(rank==1) {
 MPI_Isend(A, 100, MPI_DOUBLE, 0, MYTAG, WORLD, &request);
 MPI_Recv(B, 100, MPI_DOUBLE, 0, MYTAG, WORLD, &status);
 MPI_Wait(&request, &status);
}
```



# Collective Communications

- Called by all processes in a communicator
- `MPI_Barrier(MPI_Comm comm)`
  - Global barrier synchronization
- `MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int source, MPI_Comm comm)`
  - Distributes data from one process (the root) to all others in a communicator
  - One-to-all send of a single data block



## Collective Communications (cont'd)

- `MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int target, MPI_Comm comm)`
  - Combines data from all processes in communicator and returns it to one process
  - Target processor's `recvbuf` holds the result at the end
  - `Datatype` and `op` control the actual reduction (add, max, etc)
- `SEND/RECEIVE` can be replaced by `BCAST/REDUCE`
  - Improving both simplicity and efficiency





## Collective Communications (cont'd)

- `MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype datatype, void *recvbuf, ..., int target, MPI_Comm comm)`
  - Collects together sendbufs from all in comm
  - Each input is `sendcount*sizeof(datatype)`
  - Outputs all into target's `recvbuf` in rank order
  - Output size must be `p*sendcount*sizeof(datatype)`
  - `MPI_Allgather` lacks a target input, outputs to everyone



## Collective Communications (cont'd)

- `MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype datatype, void *recvbuf, ..., int source, MPI_Comm comm)`
  - Spreads a sendbuf to all in comm
  - Input size must be  $p \times \text{sendcount} \times \text{sizeof}(\text{datatype})$
  - Puts  $\text{sendcount}/p$  into each  $p$ 's recvbuf
  - Outputs each get part of input in rank order
  - Each output is  $\text{sendcount} \times \text{sizeof}(\text{datatype})$



## Collective Communications (cont'd)

- `MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype datatype, void *recvbuf, ..., int source, MPI_Comm comm)`
  - Trades portions of sendbuf to all in comm
  - Input size must be  $p \times \text{sendcount} \times \text{sizeof}(\text{datatype})$
  - Each  $p$  puts  $\text{sendcount}/p$  into each  $p$ 's recvbuf
  - Outputs each get part of ranked input in rank order
  - Each output is  $p \times \text{sendcount} \times \text{sizeof}(\text{datatype})$



# Computing $\pi$

- The integral of  $\frac{4}{1+x^2}$  between 0 and 1 is equal to  $\pi$ 
  - The integral is approximated by a sum of  $n$  intervals
  - The approximation to the integral in each interval is  $\frac{1}{n} \times \frac{4}{1+x^2}$
- 1. The master process (rank 0) asks the user for the number of intervals
- 2. The master should then broadcast this number to all of the other processes
- 3. Each process then adds up every  $n$ 'th interval ( $x = rank/n, rank/n + size/n, \dots$ )
- 4. Finally, the sums computed by each process are added together using a reduction



# Computing $\pi$ (cont'd)

```
#include "mpi.h"
#include <math.h>
int main(int argc, char *argv[])
{
 int done = 0, n, myid, numprocs, i, rc;
 double PI25DT = 3.141592653589793238462643;
 double mypi, pi, h, sum, x, a;

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

 while (!done) {
 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;
 }
}
```



## Computing PI (cont'd)

```
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_Finalize();
return 0;
}
```

