Programming Using the Message Passing Interface

Jinkyu Jeong (jinkyu@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
Topics

• Principles of Message-Passing Programming
• Building Blocks
  – Send and Receive Operations
• MPI: the Message Passing Interface
• Topologies and embedding
• Overlapping communication with Computation
• Collective communication and computation Operations
• Groups and communicators
• MPI-derived data types
A Distributed Address Space System
Principles of Message-Passing

• The logical view of a message-passing paradigm
  – $p$ processes
  – Each with its own exclusive address space
• Data must be explicitly partitioned and placed.
• All interactions (read-only or read/write) are two-sided
  – Process that has the data
  – Process that wants to access the data.
  – Underlying costs are explicit
• Using the *single program multiple data* (SPMD) model
Send and Receive Operations

• Prototypes

  send(void *sendbuf, int nelems, int dest)
  receive(void *recvbuf, int nelems, int source)

• Consider the following code segments:

  P0
  a = 100;
  send(&a, 1, 1);
  P1
  receive(&a, 1, 0)
  send(&a, 1, 1);
  printf("%d\n", a);
  a = 0;

• The semantics of the send
  – Value received by process P1 must be 100, not 0
  – Motivates the design of the send and receive protocols
    • Non-buffered blocking message passing
    • Buffered blocking message passing
    • Non-blocking message passing
Non-Buffered Blocking Message Passing

• A simple method
  – Send operation to return only when it is safe to do so
  – Send does not return until the matching receive has been encountered

• Issues
  – Idling and deadlocks
    • Deadlock example

P0:
  send(&a, 1, 1);
  receive(&b, 1, 1);

P1:
  send(&a, 1, 0);
  receive(&b, 1, 0);
Non-Buffered Blocking Message Passing

Idling occurs when sender and receiver do not reach communication point at similar times.
Buffered Blocking Message Passing

• **Process**
  – Sender copies data into buffer
  – Sender returns after the copy completes
  – Data may be buffered at the receiver

• **A simple solution to idling and deadlock**

• **Trade-off**
  – Buffering trades idling overhead for buffer copying overhead

P0:
```c
send(&a, 1, 1);
receive(&b, 1, 1);
```

P1:
```c
send(&a, 1, 0);
receive(&b, 1, 0);
```
Buffered Blocking Message Passing

Blocking buffered transfer protocols

(a) With communication hardware

(b) w/o communication hardware: sender interrupts receiver and deposits data in buffer at receiver end.
Buffered Blocking Message Passing

Bounded buffer sizes can have significant impact on performance

P0:
for (i = 0; i < 1000; i++){
    produce_data(&a);
    send(&a, 1, 1);
}

P1:
for (i = 0; i < 1000; i++){
    receive(&a, 1, 0);
    consume_data(&a);
}

Buffer overflow leads to blocking sender. Programmers need to be aware of bounded buffer requirements
Buffered Blocking Message Passing

Deadlocks are still possible with buffering since receive operations block.

P0:
receive(&a, 1, 1);
send(&b, 1, 1);

P1:
receive(&a, 1, 0);
send(&b, 1, 0);
Non-Blocking Message Passing

• Send and receive returns before it is semantically safe
  – Sender: data can be overwritten before it is sent
  – Receiver: data can be read before it is received
• Programmer must ensure semantics of the send and receive.
  – A check-status operation is accompanied

• Benefit
  – Capable of overlapping communication overheads with useful computations

• Message passing libraries typically provide both blocking and non-blocking primitives
Non-Blocking Message Passing

(a) Without communication hardware

(b) With communication hardware
MPI: Message Passing Interface

• Standard library for message-passing
  – Portable
  – Ubiquitously available
  – High performance
  – C and Fortran APIs

• MPI standard defines
  – Syntax as well as the semantics of library routines

• Details
  – MPI routines, data-types, and constants
  – Prefixed by “MPI_”

• 6 Golden MPI functions
  – 125 functions but 6 most used functions
### MPI: Message Passing Interface

The minimal set of MPI routines.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Init</td>
<td>Initializes MPI.</td>
</tr>
<tr>
<td>MPI_Finalize</td>
<td>Terminates MPI.</td>
</tr>
<tr>
<td>MPI_Comm_size</td>
<td>Determines the number of processes.</td>
</tr>
<tr>
<td>MPI_Comm_rank</td>
<td>Determines the label of calling process.</td>
</tr>
<tr>
<td>MPI_Send</td>
<td>Sends a message.</td>
</tr>
<tr>
<td>MPI_Recv</td>
<td>Receives a message.</td>
</tr>
</tbody>
</table>
Starting and Terminating MPI Programs

• int MPI_Init(int *argc, char ***argv)
  – Initialize the MPI environment
    • Strips off any MPI related command-line arguments.
  – Must be called prior to other MPI routines

• int MPI_Finalize()
  – Must be called at the end of the computation
  – Performs various clean-up tasks to terminate the MPI environment.

• Return code
  – MPI_SUCCESS
  – MPI_ERROR
Communicators

• A communicator defines a *communication domain*
  – A set of processes allowed to communicate with each other

• **Type** MPI_Comm
  – Specifies the communication domain
  – Used as arguments to all message transfer MPI routines

• A process can belong to many different (possibly overlapping) communication domains

• **MPI_COMM_WORLD**
  – Default communicator
  – Includes all the processes
Querying Information in Communicator

- int MPI_Comm_size(MPI_Comm comm, int *size)
  - Determine the number of processes
- int MPI_Comm_rank(MPI_Comm comm, int *rank)
  - Index of the calling process
  - $0 \leq rank < \text{communicator size}$
Sending and Receiving Messages

• int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

• int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
Hello World using MPI

```c
#include <stdio.h>
#include <string.h>  /* For strlen */
#include <mpi.h>     /* For MPI functions, etc */

const int MAX_STRING = 100;

int main(void) {
    char greeting[MAX_STRING];
    int comm_sz;  /* Number of processes */
    int my_rank;  /* My process rank */

    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    if (my_rank != 0) {
        sprintf(greeting, "Greetings from process %d of %d!",
                my_rank, comm_sz);
        MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
                 MPI_COMM_WORLD);
    } else {
        printf("Greetings from process %d of %d!\n", my_rank, comm_sz);
        for (int q = 1; q < comm_sz; q++) {
            MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
                      0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            printf("%s\n", greeting);
        }
    }

    MPI_Finalize();
    return 0;
} /* main */
```
Compilation & Execution

• Compile

$ mpicc -g -Wall -o mpi_hello mpi_hello.c

• Execution

$ mpiexec -n <number of processes> <executable>

Greetings from process 0 of 1 !
Greetings from process 0 of 4 !
Greetings from process 1 of 4 !
Greetings from process 2 of 4 !
Greetings from process 3 of 4 !
Sending and Receiving Messages

• int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
• int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)

• buf
  – Pointer to a sending/receiving buffer

• count
  – # of items to transfer
  – Of type specified by datatype
## MPI Datatypes

<table>
<thead>
<tr>
<th>MPI Datatype</th>
<th>C Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>signed short int</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>signed long int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>MPI_UNSIGNED</td>
<td>unsigned int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>long double</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td>8 bits</td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td>Packed sequence of bytes</td>
</tr>
</tbody>
</table>
Sending and Receiving Messages

• Message tag
  – Tags allow programmers to deal with the arrival of messages in an orderly manner
  – Range of tag
    • $0 \ldots 32767$ ($2^{15} -1$) are guaranteed
    • The upper bound is provided by MPI_TAG_UB
    • MPI_ANY_TAG can be used as a wildcard value
Message matching

MPI_Send(send_buf_p, send_buf_sz, send_type, dest, send_tag, send_comm);

MPI_Send
src = q

MPI_Recv(dest = r)

MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag, recv_comm, &status);

q
r
Receiving Messages

- Two wildcards of MPI_recv
  - MPI_ANY_SOURCE
  - MPI_ANY_TAG

MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag, recv_comm, &status);

- A receiver can get a message without knowing
  - The amount of data in the message
  - Sender of the message
  - Tag of the message
Receiving Messages

- **MPI_Status**
  - Stores information about the MPI_Recv operation.
  - Data structure contains:
    ```c
    typedef struct MPI_Status {
        int MPI_SOURCE;
        int MPI_TAG;
        int MPI_ERROR;
    };
    ```

- `int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)`
  - Returns the precise count of data items received
  - Not directly accessible
Deadlock Pitfall (1)

```c
int a[10], b[10], myrank;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
    MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
    MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (myrank == 1) {
    MPI_Recv(b, 10, MPI_INT, 0, 2, MPI_COMM_WORLD);
    MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD);
}
...

If MPI_Send is non-buffered blocking, there is a deadlock. MPI standard does not specify whether the implementation of MPI_Send is buffered blocking or not.
Deadlock Pitfall (2)

Circular communication

```c
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
          MPI_COMM_WORLD);
MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
          MPI_COMM_WORLD);
...

Once again, we have a deadlock if MPI_Send is non-buffered blocking.
```
Avoiding Deadlocks

We can break the circular wait to avoid deadlocks as follows:

```c
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank%2 == 1) {
    MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
             MPI_COMM_WORLD);
    MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
             MPI_COMM_WORLD);
}
else {
    MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
             MPI_COMM_WORLD);
    MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
             MPI_COMM_WORLD);
}
...
```
Sending and Receiving Messages Simultaneously

• Exchange messages
  
  ```c
  int MPI_Sendrecv(void *sendbuf, int sendcount,
                  MPI_Datatype senddatatype, int dest, int sendtag,
                  void *recvbuf, int recvcount, MPI_Datatype recvdatatype,
                  int source, int recvtag, MPI_Comm comm,
                  MPI_Status *status)
  ```

  - Requires both send and receive arguments
  - Avoid the circular deadlock problem

• Exchange messages using the same buffer
  
  ```c
  int MPI_Sendrecv_replace(void *buf, int count,
                            MPI_Datatype datatype, int dest, int sendtag,
                            int source, int recvtag, MPI_Comm comm,
                            MPI_Status *status)
  ```
Non-blocking Sending and Receiving Messages

• Non-blocking send and receive operations
  int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
  int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)

• Tests whether non-blocking operation is finished
  int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)

• Waits for the operation to complete
  int MPI_Wait(MPI_Request *request, MPI_Status *status)
Avoiding Deadlocks

Using non-blocking operations remove most deadlocks

```c
int a[10], b[10], myrank;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
    MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
    MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (myrank == 1) {
    MPI_Recv(b, 10, MPI_INT, 0, 2, &status, MPI_COMM_WORLD);
    MPI_Recv(a, 10, MPI_INT, 0, 1, &status, MPI_COMM_WORLD);
}
...
```

Replacing either the send or the receive operations with non-blocking counterparts fixes this deadlock.
Overlapping Communication with Computation

• Example: Cannon’s matrix-matrix multiplication

(a) Initial alignment of A
(b) Initial alignment of B
Overlapping Communication with Computation

- Example: Cannon's matrix-matrix multiplication

(c) A and B after initial alignment

(d) Submatrix locations after first shift
Overlapping Communication with Computation

- Example: Cannon’s matrix-matrix multiplication

(e) Submatrix locations after second shift
(f) Submatrix locations after third shift
Overlapping Communication with Computation

- Using blocking communications

```c
/* Get into the main computation loop */
for (i=0; i<dims[0]; i++) {
    MatrixMultiply(nlocal, a, b, c); /*c=c+a*b*/

    /* Shift matrix a left by one */
    MPI_Sendrecv_replace(a, nlocal*nlocal, MPI_DOUBLE,
                         leftrank, 1, rightrank, 1, comm_2d, &status);

    /* Shift matrix b up by one */
    MPI_Sendrecv_replace(b, nlocal*nlocal, MPI_DOUBLE,
                         uprank, 1, downrank, 1, comm_2d, &status);
}
```

Code snippet of Cannon’s matrix-matrix multiplication
Overlapping Communication with Computation

• Using non-blocking communications

```c
/* Get into the main computation loop */
for (i=0; i<dim[0]; i++) {
    MPI_Isend(a_buffers[i%2], nlocal*nlocal, MPI_DOUBLE, 
              leftrank, 1, comm_2d, &reqs[0]);
    MPI_Isend(b_buffers[i%2], nlocal*nlocal, MPI_DOUBLE, 
              uprank, 1, comm_2d, &reqs[1]);
    MPI_Irecv(a_buffers[(i+1)%2], nlocal*nlocal, MPI_DOUBLE, 
              rightrank, 1, comm_2d, &reqs[2]);
    MPI_Irecv(b_buffers[(i+1)%2], nlocal*nlocal, MPI_DOUBLE, 
              downrank, 1, comm_2d, &reqs[3]);

    /* c = c + a*b */
    MatrixMultiply(nlocal, a_buffers[i%2], b_buffers[i%2], c);
    for (j=0; j<4; j++)
        MPI_Wait(&reqs[j], &status);
}
```

Code snippet of Cannon’s matrix-matrix multiplication
Collective Communication

- MPI provides an extensive set of functions of collective communication operations
- Operations are defined over a group corresponding to the communicator
- All processors in a communicator must call these operations

- Simple collective communication: barrier

  int MPI_Barrier(MPI_Comm comm)

  - Waits until all processes arrive
One-to-All Broadcast

```c
int MPI_Bcast(void *buf, int count, MPI_Datatype datatype,
              int source, MPI_Comm comm)
```

![Diagram showing data broadcast from one rank to all other ranks](image)
All-to-One Reduction

```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count,
               MPI_Datatype datatype, MPI_Op op, int target,
               MPI_Comm comm)
```

\[ X = A_0 \text{ op } A_1 \text{ op } A_2 \text{ op } A_3 \]
# Predefined Reduction Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Meaning</th>
<th>Datatypes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MAX</td>
<td>Maximum</td>
<td>C integers and floating point</td>
</tr>
<tr>
<td>MPI_MIN</td>
<td>Minimum</td>
<td>C integers and floating point</td>
</tr>
<tr>
<td>MPI_SUM</td>
<td>Sum</td>
<td>C integers and floating point</td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>Product</td>
<td>C integers and floating point</td>
</tr>
<tr>
<td>MPI_LAND</td>
<td>Logical AND</td>
<td>C integers</td>
</tr>
<tr>
<td>MPI_BAND</td>
<td>Bit-wise AND</td>
<td>C integers and byte</td>
</tr>
<tr>
<td>MPI_LOR</td>
<td>Logical OR</td>
<td>C integers</td>
</tr>
<tr>
<td>MPI_BOR</td>
<td>Bit-wise OR</td>
<td>C integers and byte</td>
</tr>
<tr>
<td>MPI_LXOR</td>
<td>Logical XOR</td>
<td>C integers</td>
</tr>
<tr>
<td>MPI_BXOR</td>
<td>Bit-wise XOR</td>
<td>C integers and byte</td>
</tr>
<tr>
<td>MPI_MAXLOC</td>
<td>max-min value-location</td>
<td>Data-pairs</td>
</tr>
<tr>
<td>MPI_MINLOC</td>
<td>min-min value-location</td>
<td>Data-pairs</td>
</tr>
</tbody>
</table>
Collective Communications (I)

- **All** processes must call the same collective function.
  - Ex. `MPI_Recv()` in P0 + `MPI_Reduce()` in P1 (X)

- The arguments must be compatible to each other
  - Ex.

```c
Process 0
MPI_Reduce(in_buf, out_buf, 1, MPI_CHAR, MPI_SUM, 0, MPI_COMM_WORLD);
```

```c
Process 1
MPI_Reduce(in_buf, out_buf, 1, MPI_CHAR, MPI_SUM, 1, MPI_COMM_WORLD);
```

Mismatch!!
Collective Communications (2)

- Output argument is only used in the destination process
  - But, other processes should provide destination argument, even if it is NULL
  - Ex.

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MPI_Reduce(in_buf, out_buf, 1, MPI_CHAR, MPI_SUM, 0, MPI_COMM_WORLD);</code></td>
<td><code>MPI_Reduce(in_buf, NULL, 1, MPI_CHAR, MPI_SUM, 0, MPI_COMM_WORLD);</code></td>
</tr>
</tbody>
</table>
Collective vs. Point-to-Point Comm.

• **Point-to-point communication**
  – MPI_Send/Recv are matched on the basis of **tags** and **ranks**

• **Collective communication**
  – Do NOT use tags
  – They’re matched solely on the basis of receiver’s **rank** and **order**
MPI_MAXLOC and MPI_MINLOC

- **MPI_MAXLOC**
  - Combines pairs of values \((v_i, l_i)\)
  - Returns the pair \((v, l)\)
    - \(v\) is the maximum among all \(v_i\)’s
    - \(l\) is the corresponding \(l_i\)
      - (if there are more than one, it is the smallest among all these \(l_i\)’s).

- **MPI_MINLOC** does the same, except for minimum value of \(v_i\).

<table>
<thead>
<tr>
<th>Value</th>
<th>15</th>
<th>17</th>
<th>11</th>
<th>12</th>
<th>17</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

\[
\text{MinLoc(Value, Process)} = (11, 2) \\
\text{MaxLoc(Value, Process)} = (17, 1)
\]
# MPI_MAXLOC and MPI_MINLOC

MPI datatypes for data-pairs used with the MPI_MAXLOC and MPI_MINLOC reduction operations.

<table>
<thead>
<tr>
<th>MPI Datatype</th>
<th>C Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_2INT</td>
<td>pair of ints</td>
</tr>
<tr>
<td>MPI_SHORT_INT</td>
<td>short and int</td>
</tr>
<tr>
<td>MPI_LONG_INT</td>
<td>long and int</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE_INT</td>
<td>long double and int</td>
</tr>
<tr>
<td>MPI_FLOAT_INT</td>
<td>float and int</td>
</tr>
<tr>
<td>MPI_DOUBLE_INT</td>
<td>double and int</td>
</tr>
</tbody>
</table>
All-to-All Reduction

```c
int MPI_Allreduce(void *sendbuf, void *recvbuf,
                  int count, MPI_Datatype datatype, MPI_Op op,
                  MPI_Comm comm)
```

\[ X = A_0 \text{ op } A_1 \text{ op } A_2 \text{ op } A_3 \]
Prefix Operation (Inclusive)

`int MPI_Scan(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)`

```
A0  A1  A2  A3

X0 = A0,
X1 = A0 op A1,
X2 = A0 op A1 op A2,
X3 = A0 op A1 op A2 op A3
```
Prefix Operation (Exclusive)

int MPI_Exscan(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

<table>
<thead>
<tr>
<th>rank</th>
<th>A0</th>
<th>A1</th>
<th>A2</th>
<th>A3</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Scan(op)

<table>
<thead>
<tr>
<th></th>
<th>X0</th>
<th>X1</th>
<th>X2</th>
<th>X3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X0 = 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X1 = A0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X2 = A0 \text{ op } A1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X3 = A0 \text{ op } A1 \text{ op } A2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Scatter and Gather

• Gather data at one process
  int MPI_Gather(void *sendbuf, int sendcount,
                  MPI_Datatype senddatatype, void *recvbuf,
                  int recvcount, MPI_Datatype recvdatatype,
                  int target, MPI_Comm comm)

• Scatter data from source to all processes
  int MPI_Scatter(void *sendbuf, int sendcount,
                  MPI_Datatype senddatatype, void *recvbuf,
                  int recvcount, MPI_Datatype recvdatatype,
                  int source, MPI_Comm comm)
Allgather

- Gather and scatter them to all processes

```c
int MPI_Allgather(void *sendbuf, int sendcount,
                  MPI_Datatype senddatatype, void *recvbuf,
                  int recvcount, MPI_Datatype recvdatatype,
                  MPI_Comm comm)
```

```
<table>
<thead>
<tr>
<th>rank</th>
<th>A0</th>
<th>A1</th>
<th>A2</th>
<th>A3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

allgather:
```
<table>
<thead>
<tr>
<th></th>
<th>A0</th>
<th>A1</th>
<th>A2</th>
<th>A3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td>A0</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
</tr>
<tr>
<td>A1</td>
<td>A0</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
</tr>
<tr>
<td>A2</td>
<td>A0</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
</tr>
<tr>
<td>A3</td>
<td>A0</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
</tr>
</tbody>
</table>
```
All-to-All

• The all-to-all personalized communication

```c
int MPI_Alltoall(void *sendbuf, int sendcount,
                 MPI_Datatype senddatatype, void *recvbuf,
                 int recvcount, MPI_Datatype recvdatatype,
                 MPI_Comm comm)
```

– Analogous to a matrix transpose

Data

```
   A0  A1  A2  A3
   B0  B1  B2  B3
   C0  C1  C2  C3
   D0  D1  D2  D3
```

alltoall

```
   A0  B0  C0  D0
   A1  B1  C1  D1
   A2  B2  C2  D2
   A3  B3  C3  D3
```
Communicators

- All MPI communication is based on a communicator which contains a context and a group
  - Contexts define a safe communication space for message-passing – viewed as system-managed tags
  - Contexts allow different libraries to co-exist
  - Group is just a set of processes
  - Processes are always referred to by the unique rank in a group

- Pre-defined communicators
  - MPI_COMM_WORLD
  - MPI_COMM_NULL  // initial value, cannot be used as for comm
  - MPI_COMM_SELF  // contains only the local process
Communicator Manipulation

- **Duplicate communicator**
  - MPI_Comm_dup(comm, newcomm)
  - Create a new context with similar structure

- **Partition the group into disjoint subgroups**
  - MPI_Comm_split(comm, color, key, newcomm)
  - Each sub-communicator contains the processes with the same *color*
  - The rank in the sub-communicator is defined by the *key*

```c
color = (rank % 2 == 0)? 0 : 1;
key  = rank / 2;
MPI_Comm_split(comm, color, key, &newcomm);
```
Communicator Manipulation – con’t

• Obtain an existing group and free a group
  – MPI_Comm_group(comm, group) – create a group having processes in the specified communicator
  – MPI_Group_free(group) – free a group

• New group can be created by specifying members
  – MPI_Group_incl(), MPI_Group_excl()
  – MPI_Group_range_incl(), MPI_Group_range_excl()
  – MPI_Group_union(), MPI_Group_intersect()
  – MPI_Group_compare(), MPI_Group_translate_ranks()

• Subdivide a communicator
  – MPI_Comm_create(comm, group, newcomm)
Creating Cartesian Topologies

• Creates cartesian topologies

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims,
                    int *dims, int *periods, int reorder,
                    MPI_Comm *comm_cart)
```

– Creates a new communicator with dims dimensions.

• ndims = number of dimensions
• dims = vector of length of each dimension
• periods = vector indicates which dims are periodic (wrap-around link)
• Reorder = flag – ranking may be reordered

P0, P1, P2, P3

MPI_Cart_create(comm, 2, {2,2}, {0,0}, ..., new_comm)
Using Cartesian Topologies

• Sending and receiving messages still require ranks (or process IDs)

• Convert ranks to cartesian coordinates and vice-versa

\[
\text{int } \text{MPI\_Cart\_coord}(\text{MPI\_Comm } \text{comm\_cart}, \text{int } \text{rank}, \text{int } \text{maxdms}, \text{int } *\text{coords})
\]

\[
\text{int } \text{MPI\_Cart\_rank}(\text{MPI\_Comm } \text{comm\_cart}, \text{int } *\text{coords}, \text{int } *\text{rank})
\]

• The most common operation on cartesian topologies is a shift

\[
\text{int } \text{MPI\_Cart\_shift}(\text{MPI\_Comm } \text{comm\_cart}, \text{int } \text{dir}, \text{int } \text{s\_step}, \text{int } *\text{rank\_source}, \text{int } *\text{rank\_dest})
\]

Examples:

- P1 calls MPI_Cart_coord() → (0, 1)
- MPI_Cart_rank({0, 1}) → 1 (P1)
- MPI_Cart_shift(0 (direction), 2 (step)) → src:P2, dst:P0 (assuming all dims are periodic)
Splitting Cartesian Topologies

• Partition a Cartesian topology to form lower-dimensional grids:

\[
\text{int } \text{MPI\_Cart\_sub}(\text{MPI\_Comm } \text{comm\_cart}, \text{int } *\text{keep\_dims}, \\
\text{MPI\_Comm } *\text{comm\_subcart})
\]

  – \text{keep\_dims}[i] determines whether to split or not the \text{i}th dimension

• The coordinate of a process in a sub-topology

  – Derived from its coordinate in the original topology
  – Disregarding the coordinates that correspond to the dimensions that were not retained
  – Example: (2, 3) → (2) if \text{dims} = \{\text{true, false}\}
Splitting Cartesian Topologies

2 x 4 x 7

four 2 x 1 x 7

eight 1 x 1 x 7
Limitations of MPI Data Types

• Only primitive data types can be exchanged through MPI_Send/Recv

• Many programs use more complex data structures
  – Ex. struct in C

<table>
<thead>
<tr>
<th>MPI datatype</th>
<th>C datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>signed short int</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>signed long int</td>
</tr>
<tr>
<td>MPI_LONG_LONG</td>
<td>signed long long int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>long double</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td></td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td></td>
</tr>
</tbody>
</table>

Basic data types in MPI
MPI Derived Data Types

• To make more complex data types to be exchanged through MPI communication methods
  – MPI should know the size of a data structure
  – MPI should know the members within the data structure

• Location

```c
struct a {
    MPI_DOUBLE x[2];
    MPI_DOUBLE y[2];
    MPI_LONG value[2];
};
```

<table>
<thead>
<tr>
<th>Member</th>
<th>Offset in bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0</td>
</tr>
<tr>
<td>y</td>
<td>16</td>
</tr>
<tr>
<td>value</td>
<td>32</td>
</tr>
</tbody>
</table>
MPI_Type create_struct

- Builds a derived datatype consisting of individual elements

```c
int MPI_Type_create_struct(
    int count, /* in */,
    int array_of_block_lengths[], /* in */,
    MPI_Aint array_of_displacements[], /* in */,
    MPI_Datatype array_of_types[], /* in */,
    MPI_Datatype* new_type_p /* out */);
```

- array_of_block_lengths
  - Each member can be either a variable or an array
  - Ex. \{2, 2, 2\};

- array_of_displacements
  - Offsets of each member from start address
  - Ex. \{0, 16, 32\}

- array_of_types
  - Types of each member
  - Ex. \{MPI_DOUBLE, MPI_DOUBLE, MPI_LONG\}

```c
struct a {
    MPI_DOUBLE x[2];
    MPI_DOUBLE y[2];
    MPI_LONG value[2];
};
```
MPI_Get_address

- To know the address of the memory location referenced by `location_p`
- The address is stored in an integer variable of type `MPI_Aint`

```c
struct a {
    MPI_DOUBLE x[2];
    MPI_DOUBLE y[2];
    MPI_LONG value[2];
};
```

```c
int MPI_Get_address(
    void* location_p /* in */,
    MPI_Aint* address_p /* out */);
```

```c
#include "mpi.h"

struct a a;
MPI_Get_address(&a.x, &x_addr);
MPI_Get_address(&a.y, &y_addr);
MPI_Get_address(&a.value, &value_addr);
array_of_displacements[0] = x_addr - &a;
array_of_displacements[1] = y_addr - &a;
```
Other methods

- **MPI_Type_commit**
  - To let MPI know the new data type
  - After calling this function, the new data type can be used in MPI communication methods

```c
int MPI_Type_commit(MPI_Datatype* new_mpi_t_p /* in/out */);
```

- **MPI_Type_free**
  - When the new data type is no longer used, this function frees any additional storages used for the new data type

```c
int MPI_Type_free(MPI_Datatype* old_mpi_t_p /* in/out */);
```
MPI_Pack/Unpack

- An alternative method to send/receive a complex data structure
- Pack multiple data types into a single buffer
- One pair of MPI_Send & MPI_Recv
- Sender and receiver have to know which data types are packed in the single buffer

```c
buffer = malloc();
MPI_Pack(A);
MPI_Pack(B);
MPI_Pack(C);
MPI_Send(buffer, MPI_PACKED);
```

```c
buffer = malloc();
MPI_Recv(buffer, MPI_PACKED);
MPI_Unpack(A);
MPI_Unpack(B);
MPI_Unpack(C);
```
Concluding Remarks

• MPI or the Message-Passing Interface
  – An interface of parallel programming in distributed memory system
  – Supports C, C++, and Fortran
  – Many MPI implementations
    • Ex, MPICH2

• SPMD program

• Message passing
  – Communicator
  – Point-to-point communication
  – Collective communication
  – Safe use of communication is important
    • Ex. MPI_Sendrecv()
References

• COMP422: Parallel Computing by Prof. John Mellor-Crummey at Rice Univ., 2015.