Parallel Programming Languages

Hwansoo Han
Message Passing Interface - MPI

- A message-passing library specification
  - Message-passing model
  - Neither a compiler specification nor a specific product
- Parallel computers, clusters, and heterogeneous networks
- Designed to aid the development of portable parallel SW
An MPI Program Example

- **MPI_Init()**
  - Starting MPI environment
- **MPI_Finalize()**
  - Exiting MPI environment
- **MPI_Comm_size()**
  - Number of processes (= size)
- **MPI_Comm_rank()**
  - Its own MPI process ID
  - 0 … (size-1)
- **MPI_Send()**
  - Send a message
- **MPI_Recv()**
  - Receive a message

```c
#include <mpi.h>

main(int argc, char **argv) {
    MPI_Init (&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0)
        MPI_Send(…);
    else
        MPI_Recv(…);
    MPI_Finalize();
}
```
Communicator

- Communicator = context + group
- Communication in MPI takes place with respect to communicators
  - MPI_COMM_WORLD is a predefined one
  - All processes are initially in the group
- Processes may belong to many different communicators
Point-to-Point Communications

- **Sending and receiving messages**

- **Message**
  - **Data**
    - Buffer: address
    - Count
    - Datatype: MPI_CHAR, MPI_INT, MPI_FLOAT, MPI_DOUBLE, …
  - **Envelope**
    - Process ID (source/destination rank)
    - Message tag
    - Communicator

*MPI_ANY_SOURCE can be used as a wildcard value in a receive*
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

Basic point-to-point communication

- MPI_Send (buf, count, datatype, destID, tag, comm);
- MPI_Recv(buf, count, datatype, sourceID, tag, comm, status);
Blocking vs. non-Blocking Comm.

- **Blocking communication**
  - MPI_Send() does not complete until all data are sent
  - MPI_Recv() does not complete until the receive buffer is full

- **Non-blocking communication**
  - MPI_Isend() – non-blocking send
  - MPI_Irecv() – non-blocking receive
  - MPI_Test() – check if MPI_Irecv() is completed
  - MPI_Wait() – wait until the completion of MPI_Irecv()

  - MPI_Testall(), MPI_Testany(), MPI_Waitall(), MPI_Waitany()
Collective Communication

- Coordinated communication among a group of processes
  - Group is specified by communicator
  - All collective operations are blocking and no message tags
  - All processes in the communicator group call the collective operation

- Basic collective operations
  - MPI_Bcast(buf, count, datatype, root, comm)
    - Send data from one process (root) to all others
  - MPI_Reduce(buf, result, count, datatype, operation, root, comm)
    - Combine data from all processes, using a specified operation
    - Return the result to a single process (root)
**Broadcast and Reduce**

- **Basic collective operations**
  - **MPI_Bcast(buf, count, datatype, root, comm)**
    - Send data from one process (root) to all others
  - **MPI_Reduce(buf, result, count, datatype, operation, root, comm)**
    - Combine data from all processes, using a specified operation
    - Return the result to a single process (root)

<table>
<thead>
<tr>
<th>rank</th>
<th>send buffer</th>
<th>send buffer</th>
<th>rank</th>
<th>send buffer</th>
<th>result buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
<td>A</td>
<td>0</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>?</td>
<td>A</td>
<td>1</td>
<td>B</td>
<td>?</td>
</tr>
<tr>
<td>2</td>
<td>?</td>
<td>A</td>
<td>2</td>
<td>C</td>
<td>?</td>
</tr>
<tr>
<td>3</td>
<td>?</td>
<td>A</td>
<td>3</td>
<td>D</td>
<td>?</td>
</tr>
</tbody>
</table>

Bcast (root=0)

Reduce (root=0)

\[ X = A \text{ op } B \text{ op } C \text{ op } D \]
Scatter and Gather

- **MPI_Scatter**
  - Distribute the data from the root to all processes

- **MPI_Gather**
  - Collect the data from all processes to the root

![Diagram showing Scatter and Gather operations with buffers and ranks](attachment:image.png)
Other Collective Communications

- **MPI_Allreduce**

  - Rank send buffer:
    - Rank 0: A
    - Rank 1: B
    - Rank 2: C
    - Rank 3: D

  - Result buffer:
    - X

  - Allreduce operation:
    - X = A op B op C op D

- **MPI_Allgather**

  - Rank send buffer:
    - Rank 0: A
    - Rank 1: B
    - Rank 2: C
    - Rank 3: D

  - Result buffer:
    - ABCD

- **MPI_Alltoall**

  - Rank send buffer:
    - Rank 0: \(A_0B_0C_0D_0\)
    - Rank 1: \(A_1B_1C_1D_1\)
    - Rank 2: \(A_2B_2C_2D_2\)
    - Rank 3: \(A_3B_3C_3D_3\)

  - Result buffer:
    - \(A_0A_1A_2A_3\)
    - \(B_0B_1B_2B_3\)
    - \(C_0C_1C_2C_3\)
    - \(D_0D_1D_2D_3\)

- **MPI_Scan**

  - Rank send buffer:
    - Rank 0: A
    - Rank 1: B
    - Rank 2: C
    - Rank 3: D

  - Result buffer:
    - \(W = A\)
    - \(X = A \text{ op } B\)
    - \(Y = A \text{ op } B \text{ op } C\)
    - \(Z = A \text{ op } B \text{ op } C \text{ op } D\)
Collective Computation Operations

- **Allreduce, Reduce, Scan, ... take combination functions**
- **Built-in collective computation operations**
  - MPI_MAX, MPI_MIN // max, min
  - MPI_MAXLOC, MPI_MINLOC // max, min with location
  - MPI_PROD, MPI_SUM // product, sum
  - MPI_LAND, MPI_LOR, MPI_LXOR // logical and/or/xor
  - MPI_BAND, MPI_BOR, MPI_BXOR // bitwise and/or/xor
- **User-defined collective computation operations**
  - MPI_Op_create(user_func, commute_flag, user_op);
  - MPI_Op_free(user_op);

```c
user_func (invec, inoutvec, len, datatype) {
    for (i=0; i<len; i++) inoutvec[i] = invec[i] op inoutvec[i];
}
```
Synchronization

- **MPI_Barrier (comm)**
  - Blocks until all processes in `comm` reach the point and call the function
  - Often not needed at all in many message-passing codes
  - Mostly needed for highly asynchronous programs
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 subgroup contains the processes with the same color
  - The rank in the subgroup is defined by the key

```c
color = (rank % 2 == 0)? 0 : 1;
key = rank / 2;
MPI_Comm_split(comm, color, key, &newcomm);
```
Communicator Manipulation (cont’d)

- **Subdivide a communicator**
  - MPI_Comm_create(comm, group, newcomm)

- **Obtain an existing group and free a group**
  - MPI_Comm_group (comm, group) – obtain existing group
  - 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()
Timing MPI Programs

- **MPI Wtime()**
  - Returns a double precision floating-point number of seconds, which represents elapsed wall-clock time since some time in the past
  - Can measure the interval by calling it twice within a process
  - But it is not synchronized with other processes

- **MPI Wtick()**
  - Return the resolution of the tick in a double precision FP number of seconds
  - i.e., interval between two successive ticks