MPI—Message Passing Interface

Goals
- Portable application-level interface
- Support efficient communication across a wide variety of machines
- Support heterogeneous computing environments
- Provide a reliable communication interface

History
- Defined by a large consortium (60 individuals, 40 organizations)
- First standard presented in 1992
- Widely adopted
  - Many implementations, including vendor-specific implementations
  - Widely used
- MPI2
  - Extensions proposed starting in 1995

MPI—Message Passing Interface (cont)

History (cont)
- MPI 2.0 (1997)
  - Adds many features
    - Process management
    - One-sided communication
    - Parallel I/O
  - Rarely implemented or used
The Basic Model

Distributed memory
- Each process sees a local address space
- Processes send messages to communicate with other processes

SPMD code
- Write one piece of code that executes on each processor

Basic Model (cont)

SPMD code
- Write one piece of code that executes on each processor

SPMD vs. SIMD?
- SIMD is a hardware execution model
- Each instruction executes in lock step
- SPMD is a software execution model—each process executes independently
Execution Models

SPMD execution
- Execute the same binary on each processor
- Can mimic MIMD execution by using control flow that depends on a process’ rank

MIMD execution
- Execute different binary on different processors

How do SPMD and MIMD differ?
- Fundamentally, no difference
- MIMD supports heterogeneous processors
- MIMD has lower control flow overhead
- MIMD has smaller code size
- MIMD code may be easier for a compiler to analyze (?)

#include <stdio.h>
#include "mpi.h"
int main(argc, argv)
int argc;
char ** argv;
{
  int rank, value, size;
  MPI_Status status;
  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  /* do something interesting */
  MPI_Finalize();
  return 0;
}

MPI Example: Initialization and Cleanup

This is a communicator, which is a scoping mechanism for grouping sets of related communication operators

The rank is this process’s ID within this communicator

The size is this size of this communicator
MPI Example: Point-to-Point Communication

```c
/* do something interesting */
do {
    if (rank==0) {
        scanf ("%d", &value);
        MPI_Send (&value, 1, MPI_INT, rank+1, 0, MPI_COMM_WORLD);
    } else {
        MPI_Recv (&value, 1, MPI_INT, rank-1, 0, MPI_COMM_WORLD);
        if (rank < size-1)
            MPI_Send (&value, 1, MPI_INT, rank+1, 0, MPI_COMM_WORLD);
    }
    printf ("Process %d got %d\n", rank, value);
} while (value >= 0);
```

Point-to-Point Communication

**MPI_Send**
- Blocking send—blocks until the message buffer is safe to reuse

**MPI_Recv**
- Blocking receive—blocks until the message buffer is safe to reuse

Will the following code lead to deadlock?

```c
/* Assume two processes */
MPI_Send (&value, 1, MPI_INT, 1-rank, 0, MPI_COMM_WORLD);
MPI_Recv (&value, 1, MPI_INT, 1-rank, 0, MPI_COMM_WORLD);
```
MPI Example: Point-to-Point Communication (cont)

```c
/* do something interesting */
do {
    if (rank==0) {
        scanf ("%d", &value);
        MPI_Send (&value, 1, MPI_INT, rank+1, 0,
                   MPI_COMM_WORLD);
    } else {
        MPI_Recv (&value, 1, MPI_INT, rank-1, 0,
                   MPI_COMM_WORLD);
        if (rank < size-1)
            MPI_Send (&value, 1, MPI_INT, rank+1, 0,
                       MPI_COMM_WORLD);
    }
    printf ("Process %d got %d\n", rank, value);
} while (value >= 0);
```

Round Trip Message Latency

Latency
- Much copying and synchronization
**Cost of Blocking Communication**

**Implications**
- Lower latency – e.g. `MPI_Send()` returns when data has been copied to the kernel

<table>
<thead>
<tr>
<th>Sending Process</th>
<th>Kernel</th>
<th>Kernel</th>
<th>Receiving Process</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>send</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>latency</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>wait</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>recv</code></td>
</tr>
</tbody>
</table>

**Cost of Non-Blocking Communication**

**Implications**
- Lower latency
- Buffer might be overwritten before being copied to the kernel

<table>
<thead>
<tr>
<th>Sending Process</th>
<th>Kernel</th>
<th>Kernel</th>
<th>Receiving Process</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>isend</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>wait</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>recv</code></td>
</tr>
</tbody>
</table>
Collective Communication

**Barriers**
- Pure synchronization

**Gather**
- Collect data from all processes to a single process

**Scatter**
- Spread data from one process to all other processes

**Reductions**
- Compute max, min, sum of values that reside on multiple processes
- Can also compute some user-defined function

**Scans**
- Parallel prefix