An Introduction to MPI

Edgar Gabriel
Overview

- Distributed memory machines
- Basic principles of the Message Passing Interface (MPI)
  - addressing
  - startup
  - data exchange
  - process management
  - communication
Distributed memory machines

- Compute node
- Processor 1
- Processor 2
- Memory
- Network card 1
- Network card 2
- Local disks
- Message passing network
- Administrative network

An Introduction to MPI
Edgar Gabriel
Communication between different machines

1st Process (Client)

Processor 1
- Memory
- Network card 1
- Local disks

Processor 2

2nd Process (Server)

Processor 1
- Memory
- Network card 1
- Local disks

Processor 2

Host Name and Host Address
- mypc.my-university.edu 183.69.14.54
- webserver.provider.com 129.74.11.55

http request

Protocol

An Introduction to MPI
Edgar Gabriel
Communication between different machines on the Internet

• Addressing:
  – hostname and/or IP Address

• Communication:
  – based on protocols, e.g. http or TCP/IP

• Process start-up:
  – every process (= application) has to be started separately
The Message Passing universe

- Process start-up:
  - Want to start \( n \)-processes which shall work on the same problem
  - mechanisms to start \( n \)-processes provided by MPI library

- Addressing:
  - Every process has a unique identifier. The value of the rank is between 0 and \( n-1 \).

- Communication:
  - MPI defines interfaces/routines how to send data to a process and how to receive data from a process. It does not specify a protocol.
History of MPI

• Until the early 90’s:
  – all vendors of parallel hardware had their own message passing library
  – Some public domain message passing libraries available
  – all of them being incompatible to each other
  – High efforts for end-users to move code from one architecture to another
• June 1994: Version 1.0 of MPI presented by the MPI Forum
• June 1995: Version 1.1 (errata of MPI 1.0)
• 1997: MPI 2.0 – adding new functionality to MPI
Simple Example (I)

MPI command to start process: `mpirun -np 2 ./t1`

Number of processes to be started: 2

Number of processes which have been started

Rank of the 1st process

Rank of the 2nd process

Name of the application to start: `./t1`
Simple example (II)

mpirun starts the application t1

- two times (as specified with the -np argument)
- on two currently available processors of the parallel machine
- telling one process that his rank is 0
- and the other that his rank is 1
Simple Example (III)

```c
#include "mpi.h"

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

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

    printf ("Mpi hi von node %d job size %d\n", rank, size);

    MPI_Finalize ();
    return (0);
}
```
MPI basics

- `mpirun` starts the required number of processes
- every process has a unique identifier (`rank`) which is between 0 and n-1
  - no identifiers are duplicate, no identifiers are left out
- all processes which have been started by `mpirun` are organized in a process group (`communicator`) called `MPI_COMM_WORLD`
  - `MPI_COMM_WORLD` is static
    - number of processes can not change
    - participating processes can not change
MPI basics (II)

- The rank of a process is always related to the process group
  - e.g. a process is uniquely identified by a tuple (rank, process group)
- A process can be part of the several groups
  - i.e. a process has in each group a rank

MPI_COMM_WORLD, size=7

new process group, size = 5
Simple Example (IV)

Function returns the rank of a process within a process group

---snip---

MPI_Comm_rank (MPI_COMM_WORLD, &rank);
MPI_Comm_size (MPI_COMM_WORLD, &size);

---snip---

Rank of a process within the process group

MPI_COMM_WORLD

Default process group containing all processes started by mpirun

Number of processes in the process group

MPI_COMM_WORLD

Function returns the size of a process group
Simple Example (V)

Function sets up parallel environment:
- processes set up network connection to each other
- default process group (MPI_COMM_WORLD) is set up
- should be the first function executed in the application

---snip---
MPI_Init (&argc, &argv);
---snip---

Function closes the parallel environment
- should be the last function called in the application
- might stop all processes

---snip---
Second example – scalar product of two vectors

- two vectors are distributed on two processors
  - each process holds half of the original vector

<table>
<thead>
<tr>
<th>Process with rank=0</th>
<th>Process with rank=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a(0\ldots N/2-1)$</td>
<td>$b(0\ldots N/2-1)$</td>
</tr>
<tr>
<td>$a(N/2\ldots N)$</td>
<td>$b(N/2\ldots N)$</td>
</tr>
</tbody>
</table>
Second example (II)

- Logical/Global view of the data compared to local view of the data

<table>
<thead>
<tr>
<th>Process with rank=0</th>
<th>Process with rank=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a(0\ldots N/2-1)$</td>
<td>$a(N/2\ldots N)$</td>
</tr>
<tr>
<td>$a_{local}(0) \Rightarrow a(0)$</td>
<td>$a_{local}(0) \Rightarrow a(N/2)$</td>
</tr>
<tr>
<td>$a_{local}(1) \Rightarrow a(1)$</td>
<td>$a_{local}(1) \Rightarrow a(N/2+1)$</td>
</tr>
<tr>
<td>$a_{local}(2) \Rightarrow a(2)$</td>
<td>$a_{local}(2) \Rightarrow a(N/2+2)$</td>
</tr>
<tr>
<td>\vdots</td>
<td>\vdots</td>
</tr>
<tr>
<td>$a_{local}(n) \Rightarrow a(N/2-1)$</td>
<td>$a_{local}(n) \Rightarrow a(N-1)$</td>
</tr>
</tbody>
</table>
Second example (III)

- Scalar product:

\[ s = \sum_{i=0}^{N-1} a[i] \times b[i] \]

- Parallel algorithm

\[ s = \sum_{i=0}^{N/2-1} (a[i] \times b[i]) + \sum_{i=N/2}^{N-1} (a[i] \times b[i]) = \sum_{i=0}^{N/2-1} (a_{local}[i] \times b_{local}[i]) + \sum_{i=0}^{N/2-1} (a_{local}[i] \times b_{local}[i]) \]

- requires communication between the processes
Second example (IV)

```c
#include "mpi.h"

int main ( int argc, char **argv )
{
    int i, rank, size;
    double a_local[N/2], b_local[N/2];
    double s_local, s;

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

    s_local = 0;
    for ( i=0; i<N/2; i++ ) {
        s_local = s_local + a_local[i] * b_local[i];
    }
```
An Introduction to MPI

Edgar Gabriel

Second example (V)

```c
if ( rank == 0 ) {
    /* Send the local result to rank 1 */
    MPI_Send ( &s_local, 1, MPI_DOUBLE, 1, 0,
               MPI_COMM_WORLD);
}

if ( rank == 1 ) {
    MPI_Recv ( &s, 1, MPI_DOUBLE, 0, 0,
                MPI_COMM_WORLD, &status );

    /* Calculate global result */
    s = s + s_local;
}
```
Second example (VI)

```c
/* Rank 1 holds the global result and sends it now to rank 0 */
if ( rank == 0 ) {
    MPI_Recv (&s, 1, MPI_DOUBLE, 1, 1, MPI_COMM_WORLD, &status);
}
if ( rank == 1 ) {
    MPI_Send (&s, 1, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD);
}

/* Close the parallel environment */
MPI_Finalize ();
return (0);
```
An Introduction to MPI
Edgar Gabriel

Sending Data

---snip---

MPI_Send (&s_local, 1, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);

---snip---

Data element which shall be send
Number of elements which shall be send
Data Type of the element which shall be send

Process group containing all processes started by `mpirun`
a user defined integer (tag) for uniquely identifying a message

Rank of processes in the process group `MPI_COMM_WORLD`
to which the message shall be sent
Receiving Data

Data element where the data shall be received

Number of elements which shall be recvd

Data Type of the element which shall be recvd

---snip---

MPI_Recv (&s_local, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);

---snip---

Process group

Rank of processes in the process group which sent the message

Status information about the message

a user defined integer (tag) for uniquely identifying a message
Faulty examples (I)

- Sender mismatch:
  - if rank does not exist (e.g. rank > size of MPI_COMM_WORLD), the MPI library can recognize it and return an error
  - if rank does exist (0<rank<size of MPI_COMM_WORLD) but does not send a message => MPI_Recv waits forever => deadlock

```c
if ( rank == 0 ) {
    /* Send the local result to rank 1 */
    MPI_Send ( &s_local, 1, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);
}
if ( rank == 1 ) {
    MPI_Recv ( &s, 1, MPI_DOUBLE, 5, 0, MPI_COMM_WORLD, &status );
}
```
Faulty examples (II)

- Tag mismatch:
  - if tag outside of the allowed range (e.g. \(0 < \text{tag} < \text{MPI\_TAG\_UB}\)) the MPI library can recognize it and return an error
  - if tag in \text{MPI\_Recv} then the tag specified in \text{MPI\_Send} => \text{MPI\_Recv} waits forever => deadlock

```c
if ( rank == 0 ) {
    /* Send the local result to rank 1 */
    MPI_Send ( &s_local, 1, MPI\_DOUBLE, 1, 0, 
                MPI\_COMM\_WORLD);
}
if ( rank == 1 ) {
    MPI\_Recv ( &s, 1, MPI\_DOUBLE, 0, 18, 
                MPI\_COMM\_WORLD, &status );
}
```
What you’ve learned so far

- Six MPI functions are sufficient for programming a distributed system memory machine:

```c
MPI_Init(int *argc, char ***argv);
MPI_Finalize();

MPI_Comm_rank (MPI_Comm comm, int *rank);
MPI_Comm_size (MPI_Comm comm, int *size);

MPI_Send (void *buf, int count, MPI_Datatype dat, int dest, int tag, MPI_Comm comm);
MPI_Recv (void *buf, int count, MPI_Datatype dat, int source, int tag, MPI_Comm comm, MPI_Status *status);
```
So, why not stop here?

- **Performance**
  - need functions which can fully exploit the capabilities of the hardware
  - need functions to abstract typical communication patterns

- **Usability**
  - need functions to simplify often recurring tasks
  - need functions to simplify the management of parallel applications
So, why not stop here?

- **Performance**
  - asynchronous point-to-point operations
  - one-sided operations
  - collective operations
  - derived data-types
  - parallel I/O
  - hints

- **Usability**
  - process grouping functions
  - environmental and process management
  - error handling
  - object attributes
  - language bindings
Collective operation

- all process of a process group have to participate in the same operation
  - process group is defined by a communicator
  - all processes have to provide the same arguments
  - for each communicator, you can have one collective operation ongoing at a time
- collective operations are abstractions for often occurring communication patterns
  - eases programming
  - enables low-level optimizations and adaptations to the hardware infrastructure
MPI collective operations

MPI_Barrier
MPI_Bcast
MPI_Scatter
MPI_Scatterv
MPI_Gather
MPI_Gatherv
MPI_Allggather
MPI_Allgatherv
MPI_Alltoall
MPI_Alltoallv
MPI_Reduce
MPI_Allreduce
MPI_Reduce_scatter
MPI_Scan
MPI_Exscan
MPI_Alltoallw
More MPI collective operations

- Creating and freeing a communicator is considered a collective operation
  - e.g. `MPI_Comm_create`
  - e.g. `MPI_Comm_spawn`
- Collective I/O operations
  - e.g. `MPI_Write_all`
- Window synchronization calls are collective operations
  - e.g. `MPI_Win_fence`
MPI_Bcast

MPI_Bcast (void *buf, int cnt, MPI_Datatype dat, int root, MPI_Comm comm);

• The process with the rank root distributes the data stored in buf to all other processes in the communicator comm.
• Data in buf is identical on all processes after the bcast
• Compared to point-to-point operations no tag, since you cannot have several ongoing collective operations
MPI_Bcast (II)

\[
\text{MPI\_Bcast (buf, 2, MPI\_INT, 0, comm)};
\]

buf on root

rbuf on rank=0
rbuf on rank=1
rbuf on rank=2
rbuf on rank=3
rbuf on rank=4
MPI_Scatter (II)

MPI_Scatter (sbuf, 2, MPI_INT, rbuf, 2, MPI_INT, 0, comm);

sbuf on root

rbuf on rank=0

rbuf on rank=1

rbuf on rank=2

rbuf on rank=3

rbuf on rank=4
MPI Error handlers

• An error handler is a function which is called by the MPI library in case an error occurs
  – Wrong input parameters
  – Network or process failures

• MPI defines two predefined error handlers:
  – MPI_ERRORS_ARE_FATAL (Default): Abort the application on the first error
  – MPI_ERRORS_RETURN: Return error-code to user
    • State of MPI undefined
    • does not necessarily allow the user to continue to use MPI after an error is detected

• User can register its own error handler functions
Some implementation aspects: Open MPI

[Diagram showing a flow from User application to MPI API to MPI Component Architecture (MCA), with components like PML, PTL, Memory Pooling, Memory Management, Proc Private, Shared, Pinned, Pow 2 binning, Best fit, TCP/IP, Shared Mem, IB]

An Introduction to MPI
Edgar Gabriel
Some Links

• MPI Forum:
  – http://www.mpi-forum.org

• My personal MPI home page:

• Open MPI:
  – http://www.open-mpi.org

• MPICH: