COSC 6374
Parallel Computation

Message Passing Interface (MPI) - I
Introduction

Edgar Gabriel
Fall 2011

Overview

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

Communication between different machines

1st Process (Client)

2nd Process (Server)

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
- 2008: MPI 2.1
- 2009: MPI 2.2 and 3.0 in progress

Simple Example (I)

- MPI command to start process
- name of the application to start
- number of processes to be started
- Number of processes which have been started
- Rank of the 1st process
- Rank of the 2nd process
- Connected to cactus.de.de
**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

```bash
cmpc> mpirun -np 2 ./t1
```

**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
    
    0 1 2 3 4 5 6
    0 1 2 3 4

Edgar Gabriel
Simple Example (IV)

Function returns the rank of a process within a process group

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

---snip---

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

---snip---

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---

MPI_Finalize ();

---snip---

Function closes the parallel environment

• should be the last function called in the application
• might stop all processes
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...N/2-1) ]</td>
<td>[ a(N/2...N-1) ]</td>
</tr>
<tr>
<td>[ b(0...N/2-1) ]</td>
<td>[ b(N/2...N-1) ]</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...N/2-1) ]</td>
<td>[ a(N/2...N-1) ]</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:
  \[ s = \sum_{i=0}^{N-1} a[i] * b[i] \]

• Parallel algorithm
  \[ s = \sum_{i=0}^{N/2-1} (a[i] * b[i]) + \sum_{i=N/2}^{N-1} (a[i] * b[i]) \]
  \[ = \sum_{i=0}^{N/2-1} (a_{\text{local}}[i] * b_{\text{local}}[i]) + \sum_{i=0}^{N/2-1} (a_{\text{local}}[i] * b_{\text{local}}[i]) \]
  - 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];
    }
}```
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);
```
Sending Data

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

---snip---

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

---snip---

- Process group containing all processes started by mpirun
- Rank of processes in the process group to which the message shall be sent
- a user defined integer (tag) for uniquely identifying a message

Receiving Data

- Data element where the data shall be received
- Number of elements which shall be recv
- Data Type of the element which shall be recv

---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, 0, 
                MPI_COMM_WORLD, &status );
}
```

Faulty examples (II)

- Tag mismatch:
  - if tag outside of the allowed range (e.g. 0<tag<MPI_TAG_UB) the MPI library can recognize it and return an error
  - if tag in MPI_Recv then the tag specified in MPI_Send => 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, 
                MPI_COMM_WORLD, &status );
}
```
What you’ve learned so far

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

```
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

Some Links

- **MPI Forum:**
  - [http://www.mpi-forum.org](http://www.mpi-forum.org)

- **Open MPI:**
  - [http://www.open-mpi.org](http://www.open-mpi.org)

- **MPICH:**