COSC 4397
Parallel Computation

Message Passing Interface (MPI) - 1
Introduction

Edgar Gabriel
Spring 2010

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)
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
- Rank of the 1st process
- Rank of the 2nd process
- Number of processes which have been started
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
    0 1 2 3 4 5 6
    
    new process group, size = 5
    0 1 2 3 4
    ```
Simple Example (IV)

Function returns the rank of a process within a process group

```c
---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

```c
---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... \frac{N}{2} - 1) )</td>
<td>( a(\frac{N}{2} ... N - 1) )</td>
</tr>
<tr>
<td>( b(0... \frac{N}{2} - 1) )</td>
<td>( b(\frac{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... \frac{N}{2} - 1) )</td>
<td>( a(\frac{N}{2} ... N - 1) )</td>
</tr>
<tr>
<td>( a_{\text{local}}(0) \Rightarrow a(0) )</td>
<td>( a_{\text{local}}(0) \Rightarrow a(\frac{N}{2}) )</td>
</tr>
<tr>
<td>( a_{\text{local}}(1) \Rightarrow a(1) )</td>
<td>( a_{\text{local}}(1) \Rightarrow a(\frac{N}{2} + 1) )</td>
</tr>
<tr>
<td>( a_{\text{local}}(2) \Rightarrow a(2) )</td>
<td>( a_{\text{local}}(2) \Rightarrow a(\frac{N}{2} + 2) )</td>
</tr>
<tr>
<td>( \vdots )</td>
<td>( \vdots )</td>
</tr>
<tr>
<td>( a_{\text{local}}(n) \Rightarrow a(\frac{N}{2} - 1) )</td>
<td>( a_{\text{local}}(n) \Rightarrow a(N - 1) )</td>
</tr>
</tbody>
</table>
Second example (III)

- Scalar product:
  \[ s = \sum_{i=0}^{N-1} a[i] \cdot b[i] \]

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

  - 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];
    }
    ...
}
```
Second example (V)

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)

/* 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  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

---snip---

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<tag<MPI_TAG_UUB) 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, 18,
               MPI_COMM_WORLD, &status );
}
```
What you’ve learned so far

- Six MPI functions are sufficient for programming a distributed 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 processes 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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Barrier</td>
<td>synchronizes processes on send</td>
</tr>
<tr>
<td>MPI_Bcast</td>
<td>broadcast message to all processes</td>
</tr>
<tr>
<td>MPI_Scatter</td>
<td>scatter message to processes</td>
</tr>
<tr>
<td>MPI_Scatterv</td>
<td>scatter message to processes</td>
</tr>
<tr>
<td>MPI_Gather</td>
<td>gathers message from processes</td>
</tr>
<tr>
<td>MPI_Gatherv</td>
<td>gathers message from processes</td>
</tr>
<tr>
<td>MPI_Allgatherv</td>
<td>gathers message from processes</td>
</tr>
<tr>
<td>MPI_Alltoall</td>
<td>all-to-all communication</td>
</tr>
<tr>
<td>MPI_Allgatherv</td>
<td>all-to-all gather</td>
</tr>
<tr>
<td>MPI_Alltoallv</td>
<td>all-to-all scatter</td>
</tr>
<tr>
<td>MPI_Allreduce</td>
<td>all-to-all reduction</td>
</tr>
<tr>
<td>MPI_Reduce</td>
<td>reduces message from processes</td>
</tr>
<tr>
<td>MPI_Reduce_scatter</td>
<td>reduces message from processes</td>
</tr>
<tr>
<td>MPI_Scan</td>
<td>performs a binary scan</td>
</tr>
</tbody>
</table>

### 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_File_write_all
- Window synchronization calls are collective operations
  - *e.g.* MPI_Win_fence
**MPI_Barrier**

```
MPI_Barrier (MPI_Comm comm);
```

- Synchronizes all processes of the communicator
  - no process can continue with the execution of the application until all processes of the communicator have reached this function
  - often used before timing certain sections of the application
- MPI makes no statement about the quality of the synchronization
- Advice: no scenario is known to me, which requires a barrier for correctness. Usage of `MPI_Barrier` strongly discouraged.

---

**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)

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

Example: distributing global parameters

int rank, problemsize;
float precision;
MPI_Comm comm=MPI_COMM_WORLD;

MPI_Comm_rank ( comm, &rank );
if (rank == 0 ) {
    FILE *myfile;
    myfile = fopen("testfile.txt", "r");
    fscanf (myfile, “%d”, &problemsize);
    fscanf (myfile, “%f”, &precision);
    fclose (myfile);
}

MPI_Bcast (&problemsize, 1, MPI_INT, 0, comm);
MPI_Bcast (&precision, 1, MPI_FLOAT, 0, comm);
**MPI_Scatter**

MPI_Scatter (void *sbuf, int scnt, MPI Datatype sdat, 
void *rbuf, int rcnt, MPI Datatype rdat, 
int root, MPI_Comm comm);

- The process with the rank root distributes the 
data stored in sbuf to all other processes in the 
communicator comm
- Difference to Broadcast: every process gets 
different segment of the original data at the root 
process
- Arguments sbuf, scnt, sdat only relevant 
and have to be set at the root-process

---

**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
Example: partition a vector among processes

```c
int rank, size;
float *sbuf, rbuf[3];
MPI_Comm comm=MPI_COMM_WORLD;

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

if (rank == root ) {
    sbuf = malloc (3*size*sizeof(float));
    /* set sbuf to required values etc. */
}

/* distribute the vector, 3 Elements for each process */
MPI_Scatter (sbuf, 3, MPI_FLOAT, rbuf, 3, MPI_FLOAT, root, comm);
if ( rank == root ) {
    free (sbuf);
}
```

MPI_Gather

```c
MPI_Gather (void *sbuf, int scnt, MPI_Datatype sdat, 
            void *rbuf, int rcnt, MPI_Datatype rdat, 
            int root, MPI_Comm comm);
```

- Reverse operation of MPI_Scatter
- The process with the rank root receives the data stored in sbuf on all other processes in the communicator comm into the rbuf
- Arguments rbuf, rcnt, rdat only relevant and have to be set at the root-process
**MPI_Gather (II)**

```c
MPI_Gather (sbuf, 2, MPI_INT, rbuf, 2, MPI_INT, 0, comm);
```

- `sbuf` on rank=0
- `sbuf` on rank=1
- `sbuf` on rank=2
- `sbuf` on rank=3
- `sbuf` on rank=4

**rbuf** on root

**MPI_Allgather**

```c
MPI_Allgather (void *sbuf, int scnt, MPI_Datatype sdat,
              void *rbuf, int rcnt, MPI_Datatype rdat,
              MPI_Comm comm);
```

- **Identical to MPI_Gather**, except that all processes have the final result

**sbuf** on rank=3

**on rank=1**

**sbuf** on rank=0

**on rank=0**

**sbuf** on rank=2

**rbuf** on rank=2

**rbuf** on rank=1

**rbuf** on root
Example: matrix-vector multiplication with row-wise block distribution

```c
int main( int argc, char **argv)
{
    double A[nlocal][n], b[n];
    double c[nlocal], cglobal[n];
    int i,j;

    ...  
    for (i=0; i<nlocal; i++) {
        for ( j=0; j<n; j++) {
            c[i] = c[i] + A[i][j]*b[j];
        }
    }

    MPI_Allgather( c, nlocal, MPI_DOUBLE, cglobal, nlocal,
                   MPI_DOUBLE, MPI_COMM_WORLD );
}
```

Reduction operations

- Perform simple calculations (e.g. calculate the sum or the product) over all processes in the communicator
  - MPI_Reduce
    - `outbuf` has to be provided by all processes
    - `result` is only available at `root`
  - MPI_Allreduce
    - `result` available on all processes
Predefined reduction operations

- MPI_SUM: sum
- MPI_PROD: product
- MPI_MIN: minimum
- MPI_MAX: maximum
- MPI_BAND: binary and
- MPI_BOR: binary or
- MPI_BXOR: binary exclusive or
- MPI_MAXLOC: maximum value and location
- MPI_MINLOC: minimum value and location

Reduction operations on vectors

- Reduce operation is executed element wise on each entry of the array

<table>
<thead>
<tr>
<th>Rank 0 inbuf</th>
<th>Rank 1 inbuf</th>
<th>Rank 2 inbuf</th>
<th>Rank 3 inbuf</th>
<th>Rank 0 outbuf</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>18</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>22</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>26</td>
</tr>
</tbody>
</table>

- Reduction of 5 elements with root = 0

```
MPI_Reduce (inbuf, outbuf, 5, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
```
Example: scalar product of two vectors

```c
int main(int argc, char **argv) {
    int i, rank, size;
    double a_local[N/2];
    double b_local[N/2];
    double s_local, s;
    ...
    s_local = 0;
    for (i=0; i<N/2; i++) {
        s_local = s_local + a_local[i] * b_local[i];
    }
    MPI_Allreduce (&s_local, &s, 1, MPI_DOUBLE, MPI_SUM,
                    MPI_COMM_WORLD);
    ...
}
```

Example: matrix-vector multiplication with column-wise block distribution

```c
int main(int argc, char **argv) {
    double A[n][nlocal], b[nlocal];
    double c[n], ct[n];
    int i,j;
    ...
    for (i=0; i<n; i++) {
        for (j=0; j<nlocal; j++) {
            ct[i] = ct[i] + A(i,j)*b(j);
        }
    }
    MPI_Allreduce (&ct, &c, n, MPI_DOUBLE, MPI_SUM,
                    MPI_COMM_WORLD);
    ...
}
```
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:**