Introduction to the MPI programming model
Session 2

Janko Strassburg

PATC Parallel Programming Workshop October 2013
Communicators: Scope of communication operations
Structure of messages: complex data types
Data transfer:
  – Synchronous/asynchronous
  – Blocking/non-blocking
Message tags/identifiers
Communication partners:
  – Point-to-point
  – Wild card process and message tags
Communicators

Scope of processes

- Communicator group processes
- A group defines the set of processes, that can communicate with each other
- Used in point-to-point and collective communication
- After starting a program, its processes subscribe to the “Universe” $\Rightarrow$ MPI_COMM_WORLD
- Each program has its own “Universe”
Usage of Communicators

- Fence off communication environment
- Example: Communication in library
  *What happens, if a program uses a parallel library that uses MPI itself?*

- 2 Kinds of communicators:
  - Intra-communicator: inside a group
  - Inter-communicator: between groups

- Processes in each group are always numbered 0 to $m-1$ for $m$ processes in a group
**Communicators: Scope of communication operations**

**Structure of messages: complex data types**

**Data transfer:**
- Synchronous/asynchronous
- Blocking/non-blocking

**Message tags/identifiers**

**Communication partners:**
- Point-to-point
- Wild card process and message tags
Structure of Messages

**Standard data types:**
- Integer, Float, Character, Byte, ...
- (Continuous) arrays

**Complex data types:**
- Messages including different data: counter + elements
- Non-continuous data types: sparse matrices

**Solutions:**
- Pack/unpack functions
- Special (common) data types:
  - Array of data types
  - Array of memory displacements
- Managed by the message-passing library
Data types for message contents:

- Standard types:
  - MPI_INT
  - MPI_FLOAT
  - MPI_CHAR
  - MPI_DOUBLE
  - ...

- User defined types: derived from standard types
Data Transfer

**Blocking:**
- Function does not return, before message can be accessed again
- Process is “blocked”

**Non-blocking:**
- Function returns, whether data transfer is finished or not
- Requires function to query the status of the data transfer
- Message buffers are needed
  - Length of message is limited
- Overlapping of communication and computation is possible
  - \( \Rightarrow \) Reduction of execution time
Non-blocking send:

Process 1:

Send(&x, 2)

Process 2:

Recv(&y, 1)
Concepts for blocking:

- **Locally blocking:**
  - Function is blocked, until messages has been copied into buffer
  - Transfer needs not be completed

- **Locally non-blocking:**
  - Function returns immediately, whether message has been copied or not
  - User is responsible for message
Standard Send/Receive

**MPI\_Send:**
- Is locally complete as soon as the message is free for further processing
- The message needs not be received
  ⇒ most likely it will have been transferred to communication buffer

**MPI\_Recv:**
- Is locally complete, as soon as the message has been received
Pitfall: Deadlock

Cyclic message exchange in a ring:

```c
if (rank == 0) {
    MPI_Send(buffer,length,MPI_CHAR,1,…);
    MPI_Recv(buffer,length,MPI_CHAR,1,…);
} else if (rank == 1) {
    MPI_Send(buffer,length,MPI_CHAR,0,…);
    MPI_Recv(buffer,length,MPI_CHAR,0,…);
}
```

Problem: both processes are blocked, since each process is waiting on receive to complete send.

Cyclic resource-dependencies
Deadlock Solution

No cyclic dependencies:

```c
if (rank == 0) {
    MPI_Send(buffer,length,MPI_CHAR,1,...);
    MPI_Recv(buffer,length,MPI_CHAR,1,...);
} else if (rank == 1) {
    MPI_Recv(buffer,length,MPI_CHAR,0,...);
    MPI_Send(buffer,length,MPI_CHAR,0,...);
}
```
int MPI_Probe (int source, int tag
MPI_Comm comm, MPI_Status *status)

- **source**: Origin process of message
- **tag**: Generic message tag
- **comm**: Communication handler
- **status**: Status information

- Is locally complete, as soon as a message has been received
- Does not return the message, but provides only status information about it
MPI_Sendrecv

Performs send and receive in one single function call:

```c
MPI_Sendrecv (    
    pointer to send buffer void *sendbuf, 
    size of send message (in elements) int sendcount, 
    datatype of element MPI_Datatype sendtype, 
    destination int dest, 
    int sendtag, 
    pointer to receive buffer void *recvbuf, 
    size of receive message (in elements) int recvcount, 
    datatype of element MPI_Datatype recvtype, 
    source int source, 
    int recvtag, 
    communicator MPI_Comm communicator, 
    return status MPI_Status *status);
```
MPI_Sendrecv_replace

Performs send and receive in one single function call and operates only one single buffer:

```c
MPI_Sendrecv_replace(
    pointer to buffer void *buf,
    size of message (in elements) int count,
    datatype of element MPI_Datatype type,
    destination int dest,
    tag int sendtag,
    source int source,
    tag int recvtag,
    communicator MPI_Comm communicator,
    return status MPI_Status *status);
```
Non-blocking Functions

(MPI_Isend):
- Returns immediately, whether function is locally complete or not
- Message has not been copied
  ⇒ Changes may affect contents of message

(MPI_Irecv):
- Returns immediately, whether a message has arrived or not

(MPI_Iprobe):
- Non-blocking test for a message
Is an operation completed or not?

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

Waits until operation is completed

```
int MPI_Test(MPI_Request *request, int *flag,
             MPI_status *status)
```

Returns immediately.

flag contains status of request (true/false).
Additional Wait-Functions

```c
int MPI_Waitany(int count,
                 MPI_Request *array_of_requests, int *index, MPI_Status *status)

int MPI_Waitall(int count,
                 MPI_Request *array_of_requests,
                 MPI_Status *status)

int MPI_Waitsome(int incount,
                  MPI_Request *array_of_requests,
                  int *outcount, int *array_of_indices,
                  MPI_Status *array_of_statuses)
```
**Additional Test-Functions**

```c
int MPI_Testany(int count,
                 MPI_Request *array_of_requests, int *index,
                 int *flag, MPI_Status *status)

int MPI_Testall(int count,
                 MPI_Request *array_of_requests,
                 int *flag, MPI_Status *status)

int MPI_Testsome(int incount,
                  MPI_Request *array_of_requests, int *outcount, int
                  *array_of_indices,
                  MPI_Status *array_of_statuses)
```
Non-Blocking Functions

Example: Overlapping of Computation and Communication

```
Process 1

MPI_Isend

Message buffer

MPI_Wait

Process 2

MPI_Recv(&y, 1)

completed

Time
```
Example: *Overlapping*

1. if (myrank == 0) {
2.   int x;
3.   MPI_Isend(&x,1,MPI_INT,1,3,MPI_COMM_WORLD,
              req)
4.   compute();
5.   MPI_Wait(req,status);
6. }
7. else {
8.   int x;
9.   MPI_Recv(&x,1,MPI_INT,0,3,MPI_COMM_WORLD,
              stat)
10. }

*Example Code:*

```c
if (myrank == 0) {
    int x;
    MPI_Isend(&x,1,MPI_INT,1,3,MPI_COMM_WORLD,
              req)
    compute();
    MPI_Wait(req,status);
}
else {
    int x;
    MPI_Recv(&x,1,MPI_INT,0,3,MPI_COMM_WORLD,
              stat)
}
```
## Additional Send-Modes

### Possibilities:

<table>
<thead>
<tr>
<th></th>
<th><strong>Blocking</strong></th>
<th><strong>Non-blocking</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard</td>
<td>MPI_Send</td>
<td>MPI_Isend</td>
</tr>
<tr>
<td>Synchronous</td>
<td>MPI_Ssend</td>
<td>MPI_Issend</td>
</tr>
<tr>
<td>Buffered</td>
<td>MPI_Bsend</td>
<td>MPI_Ibsend</td>
</tr>
<tr>
<td>Ready</td>
<td>MPI_Rsend</td>
<td>MPI_Irsend</td>
</tr>
</tbody>
</table>
Additional Send-Modes

All functions are available blocking & non-blocking

**Standard Mode:**
- No assumption about corresponding receive function
- Buffers depend on implementation

**Synchronous Mode:**
- Send/Receive can be started independently but must finish together
Synchronous communication: *Rendezvous*

- Return from function represents end of transfer
- Message buffers are not required
- `Send` function waits until receive finished
- `Recv` function waits until message arrives
- Side effect: synchronization of processes
Asynchronous Communication:
- Send and receive have no temporal connection
- Message buffers are required
- Buffers located at sender or receiver
- Send process does not know, whether message actually arrived or not
- Target process may not receive a message
Synchronous Data Transfer

Case 1: Send is called before receive

Execution blocked

Process 1

Send(&x, 2)

Request

Ack

Message

Process 2

Recv(&y, 1)
Synchronous Data Transfer

Case 2: Recv is called before send

Process 1
Send(&x, 2)

Process 2
Recv(&y, 1)

Time

Execution blocked
## Additional Send-Modes

### Possibilities:

<table>
<thead>
<tr>
<th></th>
<th><strong>Blocking</strong></th>
<th><strong>Non-blocking</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard</td>
<td>MPI_Send</td>
<td>MPI_Isend</td>
</tr>
<tr>
<td>Synchronous</td>
<td>MPI_Ssend</td>
<td>MPI_Issend</td>
</tr>
<tr>
<td>Buffered</td>
<td>MPI_Bsend</td>
<td>MPI_Ibsend</td>
</tr>
<tr>
<td>Ready</td>
<td>MPI_Rsend</td>
<td>MPI_Irsend</td>
</tr>
</tbody>
</table>
Message Tags

Additional Parameter:

- Identifier for message contents
- Supports distinction of different messages (e.g. commands, data, ...)
- Increases flexibility
- *msgtag* is usually arbitrarily chosen integer

Example:

```
send(&x, 2, 5) → recv(&y, 1, 5)
```
Wildcard-Identifiers

Receive-Function:

- Defines message origin and message tag
- Only corresponding messages are accepted
- All other messages are ignored

**Wild card == Joker**

- Permits messages from arbitrary origin
- Permits messages with arbitrary tag
Wild Card

\texttt{recv}(&y,a,b) \quad \text{origin} = a \\
\text{tag} = b

\texttt{recv}(&y,?,b) \quad \text{arbitrary origin} \\
\text{tag} = b

\texttt{recv}(&y,a,?) \quad \text{origin} = a \\
\text{arbitrary tag}

\texttt{recv}(&y,?,?) \quad \text{arbitrary origin} \\
\text{arbitrary tag}
Point-to-Point Communication

MPI Specifics:

- Wild Card at receive operation:
  - for message origin: `MPI_ANY_SOURCE`
  - for message tag: `MPI_ANY_TAG`

Problem:
Race Conditions/Nondeterminism
Collective Operations

Until now:

- Point-to-point operations ==> 1 Sender, 1 Receiver

Now:

- Functions and operations involving multiple processes
Collective Operations

Possibilities:

- **MPI_Barrier**: has to be passed by all processes
- **MPI_Bcast**: one process to all others
- **MPI_Gather**: collect data of other processes
- **MPI_Scatter**: distribute data onto other processes
- **MPI_Reduce**: combine data of other processes
- **MPI_Reduce_scatter**: combine and distribute
- ...

MPI - Message Passing Interface
int MPI_Barrier(MPI_Comm comm)

Communicator *comm* defines a group of processes, that has to wait until each process has arrived at the barrier.
Broadcast/Multicast

Process 0
- data
- msg
- BCast()

Process 1
- data
- BCast()

Process 2
- data
- BCast()
int MPI_Bcast(
    void *buffer, int count,
    MPI_Datatype datatype,
    int root, MPI_Comm comm)

“Message *buf* of process *root* is distributed to all processes within communicator *comm*
Distribute the array `msg_arr` of process `root` to all other processes

- Contents at index i is sent to process i
- Different implementations possible:
  Data may be returned to `root`, ...
- Widely used in SPMD Model
Scatter

**Process 0**
- data
- \( \text{msg\_arr} \)
- \text{Scatter()}\n
**Process 1**
- data
- \text{Scatter()}\n
**Process n-1**
- data
- \text{Scatter()}\n
\(\text{MPI - Message Passing Interface}\)
int MPI_Scatter (void *sendbuf, int sendcount, MPI_Datatype sendtype,
                void *recvbuf, int recvcount, MPI_Datatype recvtype,
                int root, MPI_Comm comm)
Gather

Collect data of all processes on process *root* in array *msg_arr*

- Data of process $i$ is stored at index $i$
- Opposite of Scatter-Operation
- Usually at the end of a distributed computation
- Different implementations possible
Gather

Process 0
- data
- msg_arr
- Gather()

Process 1
- data
- Gather()

Process n-1
- data
- Gather()
int MPI_Gather(
    void *sendbuf, int sendcount, MPI_Datatype sendtype,
    void *recvbuf, int recvcount, MPI_Datatype recvtype,
    int root, MPI_Comm comm)
Example: *Data Collection*

1. `int data[10];`
2. `...`
3. `MPI_Comm_rank(MPI_COMM_WORLD,&myrank);`
4. `if (myrank == 0) {`
5. `MPI_Comm_size(MPI_COMM_WORLD,&grp_size);`
6. `buf = (int*)malloc(grp_size*10*sizeof(int));`
7. `}`
8. `MPI_Gather(data,10,MPI_INT,`
9. `buf,grp_size*10,MPI_INT,0,MPI_COMM_WORLD);`
Global operation on process *root* during data collection

- Combination of **Gather** + global operation
- Logical or arithmetic operation possible
- Different implementations possible:
  - operation on *root*,
  - partial, distributed operations, ...
Reduce

Process 0
- data
- msg
- Reduce() 

Process 1
- data
- Reduce() 

Process n-1
- data
- Reduce() 

Reduce (MPI - Message Passing Interface)
int MPI_Reduce(
    void *sendbuf, void *recvbuf,
    int count, MPI_Datatype datatype, MPI_Op op,
    int root, MPI_COMM comm)

Operations:
MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, ...
Selected Features

Communicators:  
How to create process groups?

Topologies:  
How to create virtual topologies?

General data types:  
How to use your own data types?
Selected Features

Communicators:
*How to create process groups?*

Topologies:
*How to create virtual topologies?*

General data types:
*How to use your own data types?*
Communicators

Standard intra-communicator:

- MPI_COMM_WORLD =
  All processes of a program

Functions:

- MPI_Comm_group ( comm, group )
- MPI_Group_excl ( group, n, ranks, newgroup )
- MPI_Comm_create ( comm, group, comm_out )
- MPI_Comm_free ( comm )
- ...
Example: Communicator

```c
#include <mpi.h>

int main(int argc, char *argv[]) {
    int rank, size;
    int array[8] = {2,3,0,0,0,0,0,0};
    int i, subrank;
    MPI_Status status;
    MPI_Group group;
    MPI_Comm comm;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
```
Example: Communicator

... 

MPI_Comm_group(MPI_COMM_WORLD, &group);
MPI_Group_excl(group, 2, array, &group);
MPI_Group_rank(group, &subrank);
MPI_Group_size(group, &size);

MPI_Comm_create(MPI_COMM_WORLD, group, &comm);
if(subrank != MPI_UNDEFINED) {
    MPI_Gather(&rank, 1, MPI_INT, &array, 1,
               MPI_INT, 0, comm);
    MPI_Comm_free(&comm);
}
Example: Communicator

... 

if(rank == 0) {
    for(i=0;i<size;i++) printf("%d ",array[i]);
    printf("\n");
}
MPI_Finalize();

mpirun -np 8 group
0 1 4 5 6 7
Selected Features

**Communicators:**  
*How to create process groups?*

**Topologies:**  
*How to create virtual topologies?*

**General data types:**  
*How to use your own data types?*
Topologies

Topology:
A graph with the processes as nodes and connections between them as edges.
- A topology is an attribute stored \((cached)\) with a communicator.
- General graph topology and as special case: grid-topology (Cartesian topology)
- Topologies are \textbf{virtual} and are mapped to the underlying hardware topology

Topologies add semantics to the program
Topologies can simplify the code
Some Example Topologies

- **Line**
- **Ring**
- **Mesh**
- **Torus**
- **Star**
<table>
<thead>
<tr>
<th>Name</th>
<th>Dimension</th>
<th>Connected?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line</td>
<td>1</td>
<td>no</td>
</tr>
<tr>
<td>Ring</td>
<td>1</td>
<td>yes</td>
</tr>
<tr>
<td>Cube</td>
<td>2+</td>
<td>no</td>
</tr>
<tr>
<td>Torus</td>
<td>2+</td>
<td>yes</td>
</tr>
<tr>
<td>(Hypercube)</td>
<td>4+</td>
<td>no</td>
</tr>
</tbody>
</table>

Note that star is not a grid topology.
Grid Topology in MPI

```c
MPI_Cart_Create (MPI_Comm old_comm,
                  int number_of_dims,
                  int dim_sizes[],
                  int connected[],
                  int reorder,
                  MPI_Comm *cart_comm)
```

`reorder` determines whether processes in the new communicator can have ranks different to ranks in the old communicator. Reordering may have performance advantages.

Collective operation
Example:

```c
int dims[2], connected[2];
MPI_Comm grid_comm;
dims[0] = 2;
dims[1] = 3;
connected[0] = 0; /* no wrap-around */
connected[1] = 0;
MPI_Cart_create(MPI_COMM_WORLD, 2, dims,
                connected, TRUE, &grid_comm);
```
Processes are numbered in row-major order (2d grids).

Translation of rank to coordinates by

\[ \text{MPI\_Cart\_coords}( \text{MPI\_Comm\ comm, int rank, int number\_of\_dims, int coordinates[]}) \]

Translation of coordinates to rank by

\[ \text{MPI\_Cart\_rank}( \text{MPI\_Comm\ comm, int coordinates[], int *rank}) \]

Local operations
Sub-grid-topologies

```c
int free_coords[2];
MPI_comm row_comm;
free_coords[0] = 0;
free_coords[1] = 1;

MPI_Cart_sub (grid_comm, free_coords, &row_comm);
```

- Creates for each row a new communicator, because the second coordinate (the columns) is declared as `free`. A free coordinate varies, while a non-free coordinate is fixed for each communicator.

- Collective operation
Sparse representation of a graph with one integer and two sets

<table>
<thead>
<tr>
<th>process</th>
<th>neighbours</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>0,1,2,3,4</td>
</tr>
</tbody>
</table>

nnodes = 6;
index = {1,2,3,4,5,9}
edges = {5,5,5,5,5,0,1,2,3,4}
index[0] == degree of node 0
index[i] – index[i-1] == degree of node i, i > 0
Topologies ease the understanding of a program for humans

- Better maintainability
- Lower number of errors
- “Code is written once but read very often.”

Performance implications unclear

- Benefits / penalties of using topologies depends on
  - MPI implementation
  - Underlying network
  - Current actual network partition assigned to user MPI program
Selected Features

**Communicators:**
*How to create process groups?*

**Topologies:**
*How to create virtual topologies?*

**General data types:**
*How to use your own data types?*
General MPI Data Types

**Specification:**
- Array of data types (Type signatures)
- Array of memory displacements (Type map)

**Message construction:**
- `buf` ... Start address in memory
- Typemap = \{(type\(_0\), disp\(_0\)), ..., (type\(_{n-1}\), disp\(_{n-1}\))\}
- Typesig = \{type\(_0\),..., type\(_{n-1}\)\}

**i\(^{th}\) Element:**
- Address = `buf` + `disp\(_i\)` , Data type: `type\(_i\)`
MPI Data Types

Functions for creation of data types:
- MPI_Type_contiguous
- MPI_Type_vector
- MPI_Type_hvector
- MPI_Type_indexed
- MPI_Type_hindexed
- MPI_Type_struct

Additional functions:
- MPI_Address
- MPI_Type_extent
- MPI_Type_size
Minimum Set of Functions?

*For an arbitrary MPI program, only 6 Functions are needed*

- MPI_Init(...)
- MPI_Finalize(...)
- MPI_Comm_rank(...)
- MPI_Comm_size(...)
- MPI_Send(...)
- MPI_Recv(...)