COSC 6374
Parallel Computation

Message Passing Interface (MPI) - II
Advanced point-to-point operations

Edgar Gabriel
Fall 2015

Overview

• Point-to-point taxonomy and available functions
• What is the status of a message?
• Non-blocking operations
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);
```

Point-to-point operations

• Data exchange between two processes
  - both processes are actively participating in the data exchange ➔ two-sided communication

• Large set of functions defined in MPI-1 (50+)

<table>
<thead>
<tr>
<th></th>
<th>Blocking</th>
<th>Non-blocking</th>
<th>Persistent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard</td>
<td>MPI_Send</td>
<td>MPI_Isend</td>
<td>MPI_Send_init</td>
</tr>
<tr>
<td>Buffered</td>
<td>MPI_Bsend</td>
<td>MPI_Ibsend</td>
<td>MPI_Bsend_init</td>
</tr>
<tr>
<td>Ready</td>
<td>MPI_Rsend</td>
<td>MPI_Irsend</td>
<td>MPI_Rsend_init</td>
</tr>
<tr>
<td>Synchronous</td>
<td>MPI_Ssend</td>
<td>MPI_Issend</td>
<td>MPI_Ssend_init</td>
</tr>
</tbody>
</table>
A message contains of...

- the data which is to be sent from the sender to the receiver, described by
  - the beginning of the buffer
  - a data-type
  - the number of elements of the data-type
- the message header (message envelope)
  - rank of the sender process
  - rank of the receiver process
  - the communicator
  - a tag

Rules for point-to-point operations

- Reliability: MPI guarantees, that no message gets lost
- Non-overtaking rule: MPI guarantees, that two messages posted from process A to process B arrive in the same order as posted
- Message-based paradigm: MPI specifies, that a single message cannot be received with more than one Recv operation (in contrary to sockets!)
Message matching (I)

- How does the receiver know, whether the message which he just received is the message for which he was waiting?
  - the sender of the arriving message has to match the sender of the expected message
  - the tag of the arriving message has to match the tag of the expected message
  - the communicator of the arriving message has to match the communicator of the expected message

Message matching (II)

- What happens if the length of the arriving message does not match the length of the expected message?
  - the length of the message is not used for matching
  - if the received message is shorter than the expected message, no problems
  - the received message is longer than the expected message
    - an error code (MPI_ERR_TRUNC) will be returned
    - or your application will be aborted
    - or your application will deadlock
    - or your application writes a core-dump
Message matching (III)

- Example 1: correct example

```c
if (rank == 0) {
    MPI_Send(buf, 3, MPI_INT, 1, 1, MPI_COMM_WORLD);
}
else if (rank == 1) {
    MPI_Recv(buf, 5, MPI_INT, 0, 1, MPI_COMM_WORLD, &status);
}
```

Message

Recv buffer

untouched elements in the recv buffer

Message in the Recv buffer

Message matching (IV)

- Example 2: erroneous example

```c
if (rank == 0) {
    MPI_Send(buf, 5, MPI_INT, 1, 1, MPI_COMM_WORLD);
}
else if (rank == 1) {
    MPI_Recv(buf, 3, MPI_INT, 0, 1, MPI_COMM_WORLD, &status);
}
```

Message

Recv buffer

potentially writing over the end of the recv buffer

Message in the Recv buffer
Deadlock (I)

- **Question**: how can two processes safely exchange data at the same time?
- **Possibility 1**

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Send(buf, ...)</td>
<td>MPI_Send(buf, ...)</td>
</tr>
<tr>
<td>MPI_Recv(buf, ...)</td>
<td>MPI_Recv(buf, ...)</td>
</tr>
</tbody>
</table>

  - can deadlock, depending on the message length and the capability of the hardware/MPI library to buffer messages

Deadlock (II)

- **Possibility 2**: re-order MPI functions on one process

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Recv(rbuf, ...) ; MPI_Send(buf, ...) ;</td>
<td>MPI_Send(buf, ...) ; MPI_Recv(rbuf, ...) ;</td>
</tr>
</tbody>
</table>

- Other possibilities:
  - asynchronous communication - shown later
  - use buffered send (MPI_Bsend) - not shown here
  - use MPI_Sendrecv - not shown here
Example

- Implementation of a ring using Send/Recv

  Rank 0 starts the ring

          MPI_Comm_rank (comm, &rank);
          MPI_Comm_size (comm, &size);
          if (rank == 0 ) {
              MPI_Send(buf, 1, MPI_INT, rank+1, 1,comm);
              MPI_Recv(buf, 1, MPI_INT, size-1, 1,comm,&status);
          }
          else if ( rank == size-1 ) {
              MPI_Recv(buf, 1, MPI_INT, rank-1, 1,comm,&status);
              MPI_Send(buf, 1, MPI_INT, 0, 1,comm);
          }
          else {
              MPI_Recv(buf, 1, MPI_INT, rank-1, 1,comm,&status);
              MPI_Send(buf, 1, MPI_INT, rank+1, 1,comm);
          }

Wildcards

- **Question:** can I use wildcards for the arguments in Send/Recv?

- **Answer:**
  - for Send: no
  - for Recv:
    - tag: yes, MPI_ANY_TAG
    - source: yes, MPI_ANY_SOURCE
    - communicator: no
Status of a message (I)

- the MPI status contains directly accessible information
  - who sent the message
  - what was the tag
  - what is the error-code of the message

- ... and indirectly accessible information through function calls
  - how long is the message
  - has the message bin cancelled

Status of a message (II) - usage in C

```c
MPI_Status status;
MPI_Recv ( buf, cnt, MPI_INT, ..., &status);
/* directly access source, tag, and error */
src = status.MPI_SOURCE;
tag = status.MPI_TAG;
err = status.MPI_ERROR;

/* determine message length and whether it has been cancelled */
MPI_Get_count ( status, MPI_INT, &rcnt);
MPI_Test_cancelled ( status, &flag);
```
Status of a message (IV)

• If you are not interested in the status, you can pass
  - MPI_STATUS_NULL
  - MPI_STATUSES_NULL

to MPI_Recv and all other MPI functions, which return a status.

Non-blocking operations (I)

• A regular MPI_Send returns, when ‘... the data is safely
  stored away’
• A regular MPI_Recv returns, when the data is fully
  available in the receive-buffer
• Non-blocking operations initiate the Send and Receive
  operations, but do not wait for its completion.
• Functions, which check or wait for completion of an
  initiated communication have to be called explicitly
• Since the functions initiating communication return
  immediately, all MPI-functions have an I prefix (e.g.
  MPI_Isend or MPI_Irecv).
Non-blocking operations (II)

MPI_Isend (void *buf, int cnt, MPI_Datatype dat, int dest, int tag, MPI_Comm comm, MPI_Request *req);
MPI_Irecv (void *buf, int cnt, MPI_Datatype dat, int src, int tag, MPI_Comm comm, MPI_Request *req);

Non-blocking operations (III)

- After initiating a non-blocking communication, it is not allowed to touch (=modify) the communication buffer until completion
  - you can not make any assumptions about when the message will really be transferred
- All Immediate functions take an additional argument, a request
- a request uniquely identifies an ongoing communication, and has to be used, if you want to check/wait for the completion of a posted communication
Completion functions (I)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Wait</td>
<td>Wait for one communication to finish</td>
</tr>
<tr>
<td>MPI_Waitall</td>
<td>Wait for all comm. of a list to finish</td>
</tr>
<tr>
<td>MPI_Waitany</td>
<td>Wait for one comm. of a list to finish</td>
</tr>
<tr>
<td>MPI_Waitsome</td>
<td>Wait for at least one comm. of a list</td>
</tr>
<tr>
<td></td>
<td>Content of the status not defined for Send operations</td>
</tr>
</tbody>
</table>

```c
MPI_Wait (MPI_Request *req, MPI_Status *stat);
MPI_Waitall (int cnt, MPI_Request *reqs, MPI_Status *stats);
MPI_Waitany (int cnt, MPI_Request *reqs, int *index, MPI_Status *stat);
MPI_Waitsome (int cnt, MPI_Request *reqs, int *outcnt, int *indices, MPI_Status *stats);
```

Completion functions (II)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Test</td>
<td>Check, whether a comm. has finished</td>
</tr>
<tr>
<td>MPI_Testall</td>
<td>Check, whether all comm. of a list finished</td>
</tr>
<tr>
<td>MPI_Testany</td>
<td>Check, whether one of a list of comm. finished</td>
</tr>
<tr>
<td>MPI_Testsome</td>
<td>Check, how many of a list of comm. finished</td>
</tr>
</tbody>
</table>

```c
MPI_Test (MPI_Request *req, int *flag, MPI_Status *stat);
MPI_Testall (int cnt, MPI_Request *reqs, int *flag, MPI_Status *stats);
MPI_Testany (int cnt, MPI_Request *reqs, int *index, int *flag, MPI_Status *stat);
MPI_Testsome (int cnt, MPI_Request *reqs, int *outcnt, int *indices, int *flag, MPI_Status *stats);
```
Deadlock problem revisited

- **Question**: how can two processes safely exchange data at the same time?

- **Possibility 3**: usage of non-blocking operations

  ```
  Process 0
  MPI_Irecv(rbuf, ..., &req);
  MPI_Send(buf, ...);
  MPI_Wait(req, &status);

  Process 1
  MPI_Irecv(rbuf, ..., &req);
  MPI_Send(buf, ...);
  MPI_Wait(req, &status);
  ```

- **note:**
  - you have to use 2 separate buffers!
  - many different ways for formulating this scenario
  - identical code for both processes