PPCES 2026: MPI Lab

Marc-André Hermanns, hermanns@itc.rwth-aachen.de

Acknoledgements: Parts of these code example have been developed by: - Christian Iwainsky - Sandra Wienke - Hristo Iliev - Joachim Jenke

Synopsis

The purpose of this hands-on lab is to make you familiar with the basic concepts of MPI. Tasks 1–3 will introduce you to the principles of basic point-to-point communication. Task 4 will practice the concept and usage of collective communications and MPI in general.

Building and executing lab examples

Building

The lab examples are written in C and require a C99 compiler. To ensure the C99 support, the `CFLAGS` in `common/make.def` specify the flag `-std=c99`, which switches on the C99 language support for most compilers. You may need to adapt this to the compiler used on your system.

MPI uses compiler wrappers to take care of all MPI-related command-line arguments for the preprocessor, compiler, and linker. The names for these wrappers is not standardized, yet `mpicc` is a common name for the C compiler wrapper. Therefore, this is the default compiler for the lab examples.

On most systems, you should be able to build an example with just

```
% make
```

Yet, you can override the compiler wrapper used by specifying it directly with the `make` command. The following example chooses the Intel Compiler with the Intel MPI library.

```
% make MPICC=mpiicc
```

Executing

MPI does not specify a specific launch command to be used to launch an MPI application, however, many MPI libraries provide the command `mpiexec`. If you are on a local system (like your laptop), this command will likely allow you to start the MPI application. On HPC platforms however, execution of parallel application may be performed by a different launch command, and an existing `mpiexec` in the path may not work as desired. Always consult your local HPC platform documentation for the details on launching MPI applications.

The CLAIX system at RWTH Aachen University defines environment variables for the convenience of the user. For the sake of these lab exercises, the following command on the cluster frontends will launch your MPI application on a shared partition.

```
% ${MPIEXEC} -n <numberOfProcesses> \
    <application> <application_args>
```

Note: *The `\` followed by a line break allows to spread a single command across multiple lines. It is used here for the sole purpose of clarity, and is not needed if all your command line arguments are on a single line.*

All exercises contain a special make target that will launch you program with an appropriate number of MPI processes.

```
% make run
```

They also contain a special make target that will submit the program as a batch job.

```
% make batch
```

1. Hello, MPI!

Lecture Sessions: MPI Overview, MPI Concepts The purpose of this exercise is to get you familiar with the very basics of MPI programming. Start with the minimal program `hello.(c|f90)` in directory `1_helloMPI` and insert the appropriate code at the `TODO` markers.

2. Ping Pong

Lecture Sessions: Blocking Point-to-Point Communication One basic MPI program using point-to-point communication is the "ping pong" between two MPI processes. A ping-pong program skeleton can be found in directory ***2_pingPong***. Complete the source code parts marked with "TODO".

a) Make the first process of the MPI program transmit its input to the second process. The second process should then print the received value and send it back with an opposite sign to the first process, which should again print the received value.

b) Make each rank send an individually and randomly selected number of elements. Let the other process know in advance the size of the array by explicitly sending it as an additional message.

c) What is the behaviour of the program for `NPROCS=1` and `NPROCS>2`? Modify it to display an error message when started with too few processes and to execute properly with more than two processes.

d) Implement part b) of the assignment without explicitly sending the number of elements.

e) *Bonus task:* Implement a loop to send/receive messages with different sizes. How does the message size influence the time being spent in MPI functions? You may use `MPI_Wtime()` to measure wall-clock time and go with array size as high as 226 elements to make the impact of the data size clearly visible.

3. Send-Receive

Lecture Sessions: Blocking/Non-blocking Point-to-Point Communication When multiple MPI processes exchange messages concurrently using blocking communication, the application may run into a deadlock if the communication pattern is not properly implemented. Use the code example `3_sendReceive/send_receive.(c|f90)` and identify why this application runs into a deadlock. Use different techniques to overcome this problem.

(*Note: You can abort the program execution when running interactively using* `Ctrl-c`*.)*

a) Modify the original code to use `MPI_Send()` and `MPI_Recv()` such that it becomes a correct MPI program and completes execution.

b) Modify the original code to use `MPI_Sendrecv()` or `MPI_Sendrecv_replace` to avoid the deadlock.

c) Modify the original code to use non-blocking communication on one of the point-to-point communication calls to avoid the deadlock.

d) Modify the original code to use non-blocking communication on both of the point-to-point communication calls to avoid the deadlock.

e) Modify the code from **d)** to work with more than 2 MPI processes. In this case the messages should be exchanged between adjacent ranks. *(Hint: Will a special treatment be needed for the last rank?)*

4. Derived Datatypes

Messages can be exchanged with different datatype handles on sender and receiver side, if the respective type signature of the buffers at sender and receiver-side matches.

a) Extend the skeleton file `6_datatypes/derived.c` and create datatypes on the sender and receiver side to transpose a 10 x 10 matrix in flight, by reading the data column-wise (use `MPI_Type_vector`) on the sender side and receiving row-wise (use `MPI_Type_contiguous`) on the receiver side. (*Hint: It is easiest to send separate messages for each column of data)

b) Extend your solution for (a) and create a full matrix type to be able to send the data with a single message. Use a `MPI_Type_create_hindexed` (which is similar to `MPI_Type_indexed`, but uses byte displacements) to combine several columns of data into a single message. Think about why you cannot

use `MPI_Type_indexed` for this easily. ## 5. Simple Collectives ##### Lecture Sessions: Blocking Collective Communication

MPI collective operations describe common communication patterns among multiple processes. Implement the collectives `bcast_int`, `scatter_int`, `gather_int`, `alltoall_int`, and `reduce_sum_int` present in the skeleton file `4_simpleCollectives/collectives.c`. Note that these collectives are simplified to work on integer buffers.

The skeleton file already contains `printf` statements to help you verify the correctness of your implementation. You can provide the rank in `MPI_COMM_WORLD` of the process that should perform the print statements. To test your implementation execute the application with different arguments.

Here is an example for testing with four ranks:

```
% make run NPROCS=4              # rank 0 is default
% make run NPROCS=4 PROG_ARGS=1 # select rank 1 to output
% make run NPROCS=4 PROG_ARGS=2 # select rank 2 to output
% make run NPROCS=4 PROG_ARGS=3 # select rank 3 to output
```

6. Creating new communicators

Lecture Sessions: Derived Datatypes Create new communicators as described in the subtask

a) Create a duplicate of `MPI_COMM_WORLD` using `MPI_Comm_dup` and query the processes' rank and size for this communicator in `dupRank` and `dupSize`, respectively.

b) Split `MPI_COMM_WORLD` such that the resulting communicators hold the MPI processes with odd and even ranks in `MPI_COMM_WORLD`, respectively. Query the processes' rank and size for this communicator in `oddevenRank` and `oddevenSize`, respectively.

c) Split `MPI_COMM_WORLD` such that the resulting communicators hold the MPI processes with the ranks in `MPI_COMM_WORLD` below half of the size of `MPI_COMM_WORLD` and equal and above, respectively. Reorder the ranks such that the lowest rank in `MPI_COMM_WORLD` has the highest rank in the resulting communicator. Query the processes' rank and size for this communicator in `upperlowerRank` and `upperlowerSize`, respectively.