Laboratory Exercise 1
Introduction to MPI Programming

Multiprocessor Systems, DV2544

Charlie Svahnberg and Håkan Grahn
Blekinge Institute of Technology

April 4, 2016

The objective of this laboratory exercise is to give an introduction to how applications can be parallelized using MPI, get a hands-on introduction to the MPI runtime environment.

Home Assignment 1. Preparations

Read through this laboratory manual and do the **home assignments** in this manual.

End of home assignment 1.

1 Introduction

MPI stands for "Message Passing Interface" and it is more or less the communication de-facto-standard for high performance computing on clusters. MPI provides a standardized interface towards the communication primitives, giving MPI-programs the potential to be highly portable.

There are a number of implementations of the MPI standard. But most of them have the following key components:

- Commands to "boot up" and maintain the run-time environment, i.e. processes handling the underlying communication between nodes, at least one of these processes is started at each node in the cluster.
- Front-ends to the different c, C++ and Fortran compilers, the front-end makes sure the right MPI-libraries are used when a program is compiled.
- Commands to start and maintain programs, these commands interacts with the run-time environment and makes, for instance, sure that a program is started on the right number of nodes.

There exists a huge number of MPI-functions, luckely only a few of them are needed to get started. In the following sections some simple examples are

given. To get started, we recommend that you experiment with these on the laboratory environment. In our laboratory exercise and the project assigment we will use the MPICH2 implementation of the MPI-standard [2]. MPICH2 is one of the most popular, freely available, and portable implementations of MPI.

The MPI Primer [1] and the MPI-standard [2] are good starting points for MPI programming. For more complete reference material see [3] and [4].

The computer that we will use in the laboratory is called *kraken.tek.bth.se* and is a Linux server with 2 quad-core processors, i.e., it is in essence an 8 cpu machine, with 16GB primary memory. You will have a new user account on the machine, which should have been sent to you by email. The source code files and programs that are necessary for the laboratory are copied or linked to your account by executing the command `/home/hgr/bin/init_mpi`.

2 Examples

2.1 Getting started

2.1.1 Help

There are man-pages for all commands and functions, consult them when in doubt! For instance, to see the man-page for the command `mpicc` type:

```
man mpicc
```

In the same way, to see the man-page for the function `MPI_Init` type:

```
man MPI_Init
```

2.1.2 Compiling

The command:

```
mpicc -o hn hostname_mpi.c
```

Would compile the MPI-program `hostname_mpi.c` and produce an executable named `hn`.

2.1.3 Start the run-time environment

The run-time envirnment needs to be started so that instances of an MPI program can communicate. It will also make sure that output from the instances are redirected to the terminal from which the program was started. The infrastructure is started with the command:

```
mpd &
```

Before the first time we start the MPI daemon `mpd`, we need to do a few things. A file named `.mpd.conf` file must be present in the user's home directory with read and write access only for the user, and it must contain at least a line with:

```
MPD_SECRETWORD=<secretword>
```

One way to safely create this file is to do the following:

```
cd $HOME
touch .mpd.conf
chmod 600 .mpd.conf
```

and then use an editor to insert a line like

```
MPD_SECRETWORD=mr45-j9z
```

into the file. (Of course use some other secret word than mr45-j9z.) All these steps (including starting the `mpd` daemon) is done when you execute `/home/hgr/bin/init_mpi`.

2.1.4 Executing

To execute one instance of the MPI program using N processes/nodes, the command `mpirun` can be used. For example, to start one instance of the program `hn` on four nodes type:

```
mpirun -n 4 ./hn
```

The run-time environment will make sure that output generated by `hn`, e.g. by printf, will be redirected to the terminal from which `mpirun` was executed.

> **Task 1: Log in**
>
> Log in on *kraken.tek.bth.se* using the username and password that have been provided to you. Use `ssh` (Secure Shell Client) to connect to *kraken*.
> Initialize the MPI environment and copy relevant files to your account on *kraken* by executing the command `/home/hgr/bin/init_mpi`.

2.2 Example: hostname_mpi.c

The following program will try to print the hostname of the nodes it is started on.

```
/* ****************************************************************************
 * File : hostname_mpi.c
 * Author(s) : Charlie Svahnberg
 * Created : 2005-03-10
 * Last Modified : $LastChangedDate: 2008-04-04 10:33:36 +0200 (Fri, 04 Apr 2008) $
 * Last Modified by : $LastChangedBy: skalle $
 *
 * $Id: hostname_mpi.c 18381 2008-04-04 08:33:36Z skalle $
 * © 2005 by Charlie Svahnberg, Blekinge Institute of Technology.
 * All Rights Reserved
 * ************************************************************************** */
/*
 * Compile with:
 * mpicc -o hn hostname_mpi.c
 */

#include <mpi.h>

void
name(int rank, int size)
```

```
{
    char name[255];

    gethostname(name, 255);
    printf("Rank: %d / %d, hostname is: %s\n", rank, size, name);
}

int
main(int argc, char** argv)
{
  int rank;
  int size;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  printf("Starting %d/%d\n", rank, size);

  name(rank, size);

  MPI_Finalize();
  return 0;
}
```

Some notes on the code above:

- `MPI_Init` must be called before any other MPI functions are called and initiates the execution environment.
- `MPI_COMM_WORLD` is the communicator used, and refers to all processes in the MPI-application.
- `MPI_Comm_rank` is a unique number in the specified communicator. The rank can be used to differentiate the work done by the different instances[1]. The rank is also used when communicating, e.g. send a message from node with rank 0 to the node with rank 2.
- `MPI_Comm_size` will give the total number of instances running. Note that while rank starts counting at zero, the size starts with one.
- `MPI_Finalize` is used to terminate the execution environment.

[1] This is similar to way the return value of the system call `fork` is used to differentiate between the parent and the child process.

Task 2: Compile and run the `hostname_mpi` application

1. Compile the `hostname_mpi` application using the command `mpicc -o hn hostname_mpi.c`.
2. Run the application by typing `mpirun -n 4 ./hn` in the console window.

Questions:

1. Which rank numbers do the nodes get?

2.3 Example: pingpong_mpi.c

The following program sends an integer to the first node which passes the integer to node number two, etc. The last node passes the integer back the chain again. At each node some status information is added to keep track of which nodes the integer has passed. A logical view of the communication can be seen in figure 2.3.

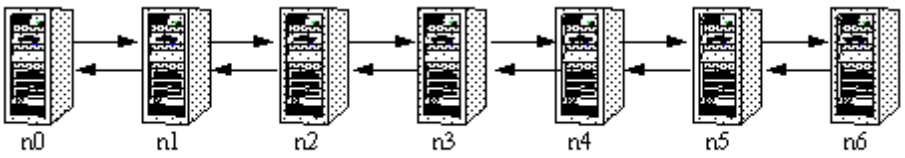

Figure 1: Logical view of communication for the pingpong program

```
/* ************************************************************************
* File : pingpong_mpi.c
* Author(s) : Charlie Svahnberg
* Created : 2005-03-01
* Last Modified : $LastChangedDate: 2008-04-04 10:33:36 +0200 (Fri, 04 Apr 2008) $
* Last Modified by : $LastChangedBy: skalle $
*
* $Id: pingpong_mpi.c 18381 2008-04-04 08:33:36Z skalle $
* © 2005 by Charlie Svahnberg, Blekinge Institute of Technology.
* All Rights Reserved
* ************************************************************************ */
/*
 * Compile with:
 * mpicc -o pingpong pingpong_mpi.c
 */

#include <mpi.h>

const int left = 1;
const int right = 2;

void
```

```
ping()
{
    short counter = 0;
    int ball;
    int fetch;

    while(1)
    {
        ball = (++counter << 16) | 0x1;
        MPI_Send(&ball, 1, MPI_INT, 1, right, MPI_COMM_WORLD);
        MPI_Recv(&fetch, 1, MPI_INT, 1, left, MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);
        fetch |= (0x1 << 8);
        printf("Sent %d, ", ball >> 16);
        fetch &= 0xffff;
        printf("-> passed %d%d%d%d %d%d%d%d, ",
                (fetch & 0x1) >> 0, (fetch & 0x2) >> 1,
                (fetch & 0x4) >> 2, (fetch & 0x8) >> 3,
                (fetch & 0x10) >> 4, (fetch & 0x20) >> 5,
                (fetch & 0x40) >> 6, (fetch & 0x80) >> 7);
        fetch >>= 8;
        printf("<- passed %d%d%d%d %d%d%d%d\n",
                (fetch & 0x1) >> 0, (fetch & 0x2) >> 1,
                (fetch & 0x4) >> 2, (fetch & 0x8) >> 3,
                (fetch & 0x10) >> 4, (fetch & 0x20) >> 5,
                (fetch & 0x40) >> 6, (fetch & 0x80) >> 7);
    }
}

/* Could have been included in forward */
void
pong(int rank)
{
    int fetch;
    int id_right = (0x1 << rank);
    int id_left = (id_right << 8);

    while(1)
    {
        MPI_Recv(&fetch, 1, MPI_INT, rank - 1, right, MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);
        fetch |= (id_right | id_left);
        /* Simulate some delay */
        usleep(200);
        MPI_Send(&fetch, 1, MPI_INT, rank - 1, left, MPI_COMM_WORLD);
    }
}

void
forward(int rank)
{
    int fetch;
    int id_right = (0x1 << rank);
    int id_left = (id_right << 8);

    while(1)
    {
        MPI_Recv(&fetch, 1, MPI_INT, rank - 1, right, MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);
        fetch |= id_right;
        /* Simulate some delay */
        usleep(100);
```

```
        MPI_Send(&fetch, 1, MPI_INT, rank + 1, right, MPI_COMM_WORLD);

        MPI_Recv(&fetch, 1, MPI_INT, rank + 1, left, MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);
        fetch |= id_left;
        /* Simulate some delay */
        usleep(100);
        MPI_Send(&fetch, 1, MPI_INT, rank - 1, left, MPI_COMM_WORLD);
    }
}

int
main(int argc, char** argv)
{
    int rank;
    int size;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    printf("Starting %d/%d\n", rank, size);

    if(rank == 0)
        ping();
    if(rank == size - 1)
        pong(rank);

    forward(rank);

    MPI_Finalize();
    return 0;
}
```

Some notes on the code above:

- `MPI_Send` is used for blocking sending of data. The arguments are:
  - Address to the data to be sent
  - Number of elements to send, i.e. 1 if the data is not in an array.
  - The MPI data type, see page 27 in [1] for a list of data types.
  - Rank of destination
  - Message tag, used by the receiver to distinguish one message from the other if multiple messages are received.
  - The communicator to use, the communicators `MPI_COMM_WORLD`, `MPI_COMM_SELF` and `MPI_COMM_PARENT` are available by default.
- `MPI_Recv` makes a blocking receive. The arguments are:
  - Address to where to place the received data
  - Number of elements to receive
  - The MPI data type.
  - Rank of source.
  - Message tag.
  - The communicator to use.

  - Address of where status information should be placed, `MPI_STATUS_IGNORE` simply ignores the status information.

- The bitmask operations are not related to MPI-programming and need not be fully understood.

Task 3: Compile and run the `pingpong_mpi` application

1. Compile the `pingpong_mpi` application using the command `mpicc -o pingpong pingpong_mpi.c`.
2. Run the application using 4 processes/nodes by typing `mpirun -n 4 ./pingpong` in the console window.

Questions:

1. Which nodes (rank numbers) do execute the functions `ping`, `pong`, and `forward`, respectively?

2. How can you tell that the "ball" has passed all the nodes in the cluster?

Task 4: Print information for each node the "ball" passes

We shall now add some code that prints some status information for each node the "ball" passes.

- Add a code line in the function `pong` that prints both the rank number and the value of `fetch` (in hexadecimal number). Put the `printf` statement after the `MPI_Recv` call.
- Add two lines of code in the function `forward` that prints both the rank number and the value of `fetch` (in hexadecimal number). Put the first `printf` statement after the first `MPI_Recv` call, and the second `printf` statement after the second `MPI_Recv` call.

References

[1] Ohio Supercomputer Center, *Mpi primer / developing with lam*, nov 1996.

[2] MPI Forum, *Message passing interface forum*, `http://www.mpi-forum.org`.

[3] M Snir, *Mpi: The complete reference: Core v. 1 (scientific & engineering computation)*, MIT Press, 1998.

[4] Marc Snir, William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, and William Saphir, *Mpi: The complete reference:*

The mpi-2 extensions vol 2 (scientific & engineering computation), The MIT Press, 1998.