November 7, 2017

DVA336 Parallel Systems

Assignment 4

Introduction to Distributed Memory Programming

Introduction

The purpose of this lab is to get acquainted with distributed memory programming using MPI (Message Passing Interface). The MPI run-time system can use SSH (Secure SHell) to distribute the execution of a program to multiple computers. Note that each process only has its own *local* view of the world - this requires explicit data movement/distribution between the processes.

You can compile all the C source files found in `src/` at once, with the command make, unless stated otherwise below. You could also compile an individual `target` with the command `make target`. Study the Makefile (`src/Makefile`) to see how the targets are compiled.

Preparation

Read Chapter 6 *Programming Using the Message-Passing Paradigm* in the main textbook *Introduction to Parallel Computing, 2nd Edition*, by Grama et al., and you may also want to look at the tutorial listed above. The MPI man-pages, the MPI standards or e.g. this tutorial hold more information about the Message Passing Interface.

Examination

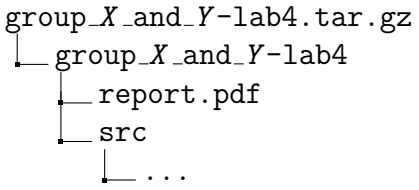
You should solve and hand in a solution for this lab in groups of **two**. Of course, it is required that each member of the group can explain the details of the groups solution individually. Therefore, you are encouraged to work on solving the different exercises on your own, and then discuss different alternatives together with your partner. This will help you get a deeper understanding of what you are doing and why you are doing it. Furthermore, having solved the exercises on your own will be to your advantage on the written exam.

Send in your gzipped tar archive-file by e-mail to the course assistant. The archive file should be called `group_`*X*`_and_`*Y*`-lab4.tar.gz`. This archive should contain

only one folder, called `group_X_and_Y-lab4`. This folder should in turn contain a report, called `report.pdf`, and a folder called `src`. The file `report.pdf` should contain answers to, results from, and reasoning about the questions in this lab-pm (explain the reasons for the phenomena that occur). The `src` folder should contain your modified source code and all files needed to compile your programs; you should not need to modify the provided `Makefile`. In the file/folder names above, the name `X_and_Y` should identify you, i.e., `X` and `Y` should be replaced with 10-character strings that clearly identifies you as the two students of the group. Example: Katy Andersson, born March 11, 1991, and Larry David, born July 2, 1947 name their file

```
group_19910311KA_and_19470702LD-lab4.tar.qz
```

The structure and naming convention of the archive file is depicted below.



Assuming you have a folder called `group_X_and_Y-lab4`, you can create the archive-file with the following command.

```
$ tar -czf group_X_and_Y-lab4.tar.gz  group_X_and_Y-lab4
```

Handling the MPI Run-Time Environment

The code you write using MPI will be part of every single process that is run. But since the address spaces of the processes are *not* shared (MPI is a local view language), all the declared variables in the program will be separate instances for each process.

This introduces a new way of thinking when writing parallel programs, compared to threaded programming. The most notable thing is that if a process writes to some variable, this will *not* be visible in the other processes' variables with the same name. Compare this to how things work when you write a program using threads, where the address space is shared.

Compiling and Running MPI Programs

To see how MPI programs are compiled, study `src/Makefile`.

You execute your MPI programs with the following command.

```
$ mpirun -np i your_program
```

Here, `i` is the number of processes to use. Note that the above given command will execute your program on your local computer only, assuming that the (Open-) MPI default hostfile is empty. To execute the program in a distributed manner, you need to define your network of computing nodes using the mpirun hostfile format.

Since you do not have privileges enough to edit the (Open-) MPI default hostfile, you will have to pass the `-hostfile` argument to the (Open-) MPI run time environment. This is done with the following command.

```
$ mpirun -hostfile your_hostfile -np i your_program
```

You can find more information about the hostfile on the Internet or in your local `man`-pages for `mpirun`. A simple example of a hostfile could look like this:

```
localhost     slots=2  max_slots=2
# node1       slots=2  max_slots=3
192.168.0.23  slots=3
last_node              max_slots=2
```

Note: the line containing `node1` is a comment and it will not be processed.

Preparation

Create a new hostfile, `src/hostfile`, in which you specify a cluster of your own computer, called `localhost`, and two other computers near by (use their IP-addresses). For example, you can set `slots=4` (or perhaps even `max_slots=4` if you want to be even more restrictive) for each node if you are using quad-core machines. (You can use the command `nproc` to print the number of processing units available.)

If you have not already set up any ssh-key for password-less logins between nodes in the computer rooms, you should execute the script `src/set_up_ssh_keys.sh`. This is done by the following command (assuming that your current working directory is `src/`):

```
$ ./set_up_ssh_keys.sh
```

If you are unsure, run the script. In order to finish the setup, you need to log in to another computer in the room (from your current computer) using ssh.

Basics

In the file `src/hello.c`, there is a simple Hello World program written using MPI. Study, compile and run the program. Run it one time on your local computer and one time distributed on the network specified in `src/hostfile`. Set the number of processes to a suitable number, e.g., use 12 processes if you are targeting 3 different quad-core machines.

Modify the program so that the slave processes wait for the master process (rank 0) to tell them to start before printing.

Hint: Consider the `MPI_Send` and `MPI_Recv` routines.

The master should print "`Master sent` ***`a_char`*** `to process` ***`i`***", where ***`a_char`*** is 'a' when sending to the process of rank 1, 'b' when sending to process 2, and so on (yes, the characters should actually be sent to the slave processes). ***`i`*** should be the rank of the process that is to receive the message. Also, let the master "do some work" inbetween sending the messages to the slave processes.

The slave/receiving processes should print "`Process` ***`i`*** `received` ***`a_char`*** `from master`", when receiving a message from the master process. ***`i`*** and ***`a_char`*** should be interpreted as described above (note that you should print the character received in the message from the master process).

While developing the solution, you can test your program on your local computer, using, e.g., 12 processes. When finished, also verify that it is able to run on the cluster specified in `src/hostfile`.

Finding the Maximum Value of an Array

In the file `src/maximum.c`, there is a skeleton program in which you should add your solution to this problem – finding the maximum value of an array of integers.

1. First, you are supposed to use `MPI_Scatter` to distribute the data to all processes. Next, each process should find their local maximum values. Finally, `MPI_Reduce` should be used to determine the global maximum value. Try running your solution with some different numbers of processes. Make sure that you always find the maximum value!

   Hint: Pay special attention to how the global problem array, `problem`, and the process local arrays, `my_problem`, are initialized. Note that some extra space is allocated in the global array to make its size a multiple of the number of processes used (i.e. its size is larger than the actual problem size). Also note that $n/p + 1$ (where n is the problem size and p is the total number of processes used) elements of the array should be distributed to each process. This is to guarantee that all elements of the global array are distributed to the (arbitrary number of) processes.

2. There is a commented code snippet that initializes the problem array the naïve – wrong – way, only taking the given problem size into account. Study this part of the code.

3. Copy your program into a new file, `src/maximum_wrong.c`, so that you save your old solution. Now, in the new file, use the wrong initialization (uncomment this part of the code and comment out the current initialization). Also make `my_problem` n/p in size (remember to adjust your solution as well). Compile your program using:

   ```
   $ make maximum_wrongmpi
   ```

   What happens if you run this solution using a problem size of 100000000 and 4 processes?

   What happens if you run this solution using a problem size of 100000000 and 7 processes?

   Hint: You can let each process print their local maximum value to ease your understanding of what is happening.

Matrix Multiplication

In the file `src/matmul.c`, there is a skeleton program that initiates three square dimensional (with dimension DIM) matrices, A, B and C (note that you should ignore all code including the matrix called D). Your task is to implement the following algorithm which calculates $C = AB$.

Note that in this exercise, you will not extend the size of the problem to be a multiple of the number of processors. Instead you will calculate a submatrix of C, with dimension $\lfloor DIM/(p-1) \rfloor \times DIM$, in each slave process and let the master process handle any leftover elements. Here, p is the total number of processes (including the master).

1. Distribute $\lfloor DIM/(p-1) \rfloor$ (different) rows of A to each process, not including the master process.

2. Distribute the entire matrix B to all slave processes.

3. Perform the process specific calculations. Let the master process handle any leftover elements in C, i.e. elements that have not been calculated by a slave process.

   Hint: The master probably needs to consider the last rows of C.

4. Gather the problem so that the result is available at the master process.

5. Run your solution both locally (on your computer only) and on the cluster specified in `src/hostfile`. What is the reason for the difference in execution times? When will the opposite case occur (i.e. when will the slowest case turn into being the fastest case)?

Groups & Communicators

In the file `src/communicators.c`, you will implement a program that divides the program-processes into separate groups, using different communicators.

1. Make every third process within `MPI_COMM_WORLD` belong to a unique group (`group1`) and make the separate (three) groups synchronize on the barrier using separate communicators (`comm1`). You are encouraged to use the predefined variables; you should *not* need any other variables – remember that the variables are local to every process.

   You must be able to handle the case when the number of used processes are not evenly dividable by 3. All processes must belong to the correct group.

   Compile and execute the program using 7, 8 and 9 processes. Make a note of the output.

   Hint: The groups should contain the following processes, using the processes' `MPI_COMM_WORLD` ranks.

First group:	`{0,3,6,...}`
Second group:	`{1,4,7,...}`
Third group:	`{2,5,8,...}`

2. **(Optional)** Further divide the processes in the first group into two new groups (`group2`) of (sometimes approximately) equal size, based on the first group (`group1`) and communicator (`comm1`). Make the processes with the lowest ranks belong to the first group, and the rest of the processes to the second group. Use the communicator `comm2` for the processes in the new groups (`group2`).

   Note that there are some commented variables in the code (of which some where mentioned in the previous paragraph). You are encouraged to uncomment and use these; you should *not* need any other variables – remember that the variables are local to every process.

   You must be able to handle the case when the number of processes in the first first-level group are not evenly dividable by 2. Compile and execute the program using 12, 13 and 14 processes. Make a note of the output.

   Hint: The groups should contain the following processes, using the processes' `MPI_COMM_WORLD` ranks. This example applies to the case when the number of used processes is 14.

First group:	`{0,3,6,9,12}`
First group, subgroup 1:	`{0,3,6}`
First group, subgroup 2:	`{9,12}`
Second group:	`{1,4,7,10,13}`
Third group:	`{2,5,8,11}`

NOTE: When submitting your solutions to the course assistant, remember to follow the directions given in the section Examination!