SOFE 3950U / CSCI 3020U: Operating Systems

TUTORIAL #10: MPI

Objectives

- Learn the fundamentals of parallel clustered computing
- Gain experience writing parallel code using MPI

Important Notes

- Work in groups of **four** students
- All reports must be submitted as a PDF on blackboard, if source code is included submit everything as an archive (e.g. zip, tar.gz)
- Save the file as <tutorial_number>_<first student's id>.pdf (e.g. tutorial10_100123456.pdf)
- If you cannot submit the document on blackboard then please contact the TA with your submission at **jonathan.gillett@uoit.net**

Notice

It is recommended for this lab activity and others that you save/bookmark the following resources as they are very useful for C programming.

- http://en.cppreference.com/w/c
- http://www.cplusplus.com/reference/clibrary/
- http://users.ece.utexas.edu/~adnan/c-refcard.pdf
- http://gribblelab.org/CBootcamp

The following resources are helpful as you will need to use MPI to complete this tutorial.

- https://computing.llnl.gov/tutorials/mpi/
- http://mpitutorial.com/tutorials/
- http://condor.cc.ku.edu/~grobe/docs/intro-MPI-C.shtml

Conceptual Questions

1. Explain what MPI is, what are it's benefits?

2. Explain briefly how MPI supports running on separate computers in a cluster.

3. Name **four** MPI data types (hint they all start with **MPI_**).

4. Read the documentation for and explain the **MPI_Bcast** function.

5. Read the documentation for and explain the **MPI_Send** and **MPI_Recv** functions.

Application Questions

All of your programs for this activity can be completed using the template provided, where you fill in the remaining content. A makefile is not necessary, to compile your programs use the following command in the terminal. **For this tutorial you MUST use mpicc to compile MPI programs. If you are still having issues please ensure that you have installed MPI on your system.**

You must install MPI on your computer in order to execute MPI programs, on most Ubuntu systems it can be installed using the following commands.

```
sudo apt-get install openmpi-bin libopenmpi-dev
```

Your programs can be compiled using mpicc as follows:

```
mpicc -Wall -Wextra -std=c99 <program name>.c -o <program name>
-lm
```

Example:

```
mpicc -Wall -Wextra -std=c99 q1.c -o q1 -lm
```

You can then execute and test your program by running it with the following command.

```
mpiexec -np <num processes> ./<program>
```

Example:

```
mpiexec -np 4 ./q1
```

Template

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <mpi.h>

int main(void)
{ ... }
```

Notice

You **MUST** use **mpicc** for this tutorial to compile your programs.

1. Ensure that you are able to execute and compile the following two demo programs: **demo_1.c** and **demo_2.c.**

- Once you have both programs compiled and running add comments above each **MPI specific** line in the code explaining what each statement does.
- You should be able to get both examples working if you have installed MPI.

2. Create a program using MPI that does the following.
    - Performs a “ping” and “pong” operation to send a message back and forth between two programs.
    - You will need to use **MPI_Send** and **MPI_Recv** to send data back and forth, each time a client receives data, it should should send the exact same data back to the sender (a pong).
    - Increase the size of data sent and perform some benchmarks, how long does it take to send an array of **10 million** double values back and forth 100 or 1000 times?
    - You **MUST** use **MPI_Wtime()** in order to accurately record the time it takes for the master to send and receive all the data back.
    - You will likely need to use malloc() to allocated the 10 million double values.

3. Create a program using MPI that does the following.
    - Create two 2-dimensional integer arrays **A,** and **B** that are 100x100, and a third matrix **C,** that is 100x100 as well to store the results.
    - Initialize each of the 2 dimensional arrays **(A, B)** to a specified integer value (e.g. to the current index of the for loop for example).
    - Implement **parallel matrix multiplication** using MPI.
    - You will need to send the second matrix **B** to each MPI slave process using the **Broadcast** functionality in MPI.
    - For each row of matrix **A** use MPI **Send** and **Recv** functions to send the row to a slave process, which performs matrix multiplication using that row on matrix **B** and returns the row of calculations back as the result.
    - You will have to take extra precaution to make sure that the row returned back as a result is copied into the **correct row** in the resultant array **C**.
    - You should use another tool to perform the matrix multiplication to verify that your parallel solution is correct, it is likely you may have issues combining your results back into the final matrix correctly.
    - For details on matrix multiplication please see: https://en.wikipedia.org/wiki/Matrix_multiplication

4. Create a program using MPI that does the following.
    - Finds the **prime numbers** within a specified search space.
    - You will need to create a two dimensional integer array **primes** that stores the primes found by each slave process for the search space found.
    - Look up the theory on **prime density function** to get the number of primes you can expect to find for each search space, otherwise an easy solution is to assume that all odd numbers could be prime.
    - Your program should use the MPI **Send** to send an array of two integers containing the start and stop bounds for each, for example if you are searching for all the primes between 1 - 1000, and you have **three** slave processes, you would **Send** the following three arrays: [1, 333], [334, 666], [667, 1000].
    - After each slave process goes through the range of of numbers to search for primes, it then returns back an **array** containing the prime numbers found.
    - There are a number of more efficient ways to search for prime numbers, however you can use the easiest method (exhaustive search for roots).
    - The master process then receives each of the primes found from the slaves and adds it to the **primes array,** which is a total list of primes found.
    - Your program should then print the final list of all prime numbers found, you can easily check online to validate your answer.
    - If you need a refresher on prime number theory please see: https://en.wikipedia.org/wiki/Prime_number