CSE111 Spring 2022

Assignment 5

In this assignment you will implement a Berkeley Socket based, distributed, C++, multi-threaded, most significant digit radix sorter to satisfy complimentary functional and non-functional requirements.

This assignment is worth 10% of your final grade.

Late submissions will not be graded.

Background information

In Assignment 4 you developed a truly parallel radix sorter running stand-alone on a single machine. In this lab you will make the services of that parallel sorter available to clients connecting over the network.

Details of socket programming can be found in the lecture handouts and in many, many on-line locations.

Setup

SSH into any of the CSE111 teaching servers using your CruzID Blue credentials:

```
$ ssh <cruzid>@<server>.soe.ucsc.edu
```

Create a suitable place to work: ***(only do this the first time you log in)***

```
$ mkdir -p ~/CSE111/Assignment5
$ cd ~/ CSE111/Assignment5
```

Install the lab environment: ***(only do this once)***

```
$ tar xvf /var/classes/CSE111/Assignment5.tar.gz
```

Build, test, and check code coverage of the system:

```
$ make test
```

Check your implementation for memory leaks:

```
$ make valgrind
```

Calculate your expected grade:

```
$ make grade
```
(this will take some time)

Reset the system if you get into a mess:

```
$ make clean
```

Requirements

Your completed network-enabled radix sorter must do two things:

1. Correctly MSD radix sort moderately sized vectors of unsigned integers sent to it across the network
2. Cap it's CPU usage to a defined maximum number of cores.

The first being a functional requirement, the second be a non-functional (performance) requirement.

These requirements are NOT equally weighted - see Grading Scheme below. However, it is recommended you work on the functional requirement first, and only then work on the non-functional requirement.

Remember: *"make it work"* always comes before *"make it fast"*.

What you need to do

The classes `RadixServer` and `RadixClient` are provided, but unimplemented. You need to implement them without changing the existing signature. You can add member variables and functions, but do not change existing signatures or override the default or given constructors.

It is recommended your implementation of `RadixServer` delegate to your Assignment 4 implementation of `ParallelRadixSorter`, but this is not required if you wish to create a totally new implementation.

Basic steps are as follows:

1. Implement client end of the distributed radix sorter
2. Implement server end of the distributed radix sorter
    a. Have the server delegate to sorting implementations from previous assignments
3. Ensure you stick to the on-wire protocol as described below

To execute your client-server MSD radix sorter after running `make`:

```
$ ./radix 1000 1 10 -d
```

Where "1000" is the number of random unsigned integers that the test harness will generate, "1" is the number of times it will generate that many random unsigned integers, and "10" is the maximum number of CPU cores to use when sorting the single list.

The `-d` flag requests a sampled dump of the sorted vectors to demonstrate (hopefully) correct ordering.

As in previous assignments, if there's something you don't understand, do this, in this order:

1. Google it
2. Post a discussion on Slack
3. Come along to office hours and ask questions

Note that the test harness has embedded implementations of both client and server. Your implementations are tested against them, then tested against each other.

Client-Server Protocol

Your client and server should obey the following on-wire protocol:

1. Unsigned integers are exchanged in binary form, in order, one-at-a-time, over a stream socket
2. The list termination marker is a zero

A simple implementation of this protocol might be as follows:

- Client receives list of unsigned integers from test harness
- Client sends all unsigned integers in the list to server
- Server stores each unsigned integer as it is received
- Client sends a zero as a list termination flag
- On receipt of the zero, server sorts previously received unsigned integers
- Server returns sorted unsigned integers to the client
- Client stores each unsigned integer as it is received
- Server sends a zero as a list termination flag
- On receipt of the zero, client returns sorted unsigned integers to test harness

Grading scheme

The following aspects will be assessed by executing your code on a machine with an identical configuration to the CMPS109 teaching servers:

1. (100% Marks) **Does it work?**

a.	Functional Requirement	(40%)
b.	Non-Functional Requirement	(30%)
c.	100% Function, line, and branch coverage	(20%)
d.	Your implementation is free of compiler warnings and memory errors	(10%)

For a, marks are deducted for any sort operations failing to produce the correct answer.

For b, marks are deducted for any multi-core sorts failing to perform as required.

Note that the non-functional tests in (b) simply check cpu core capping.

2. (-100%) **Did you give credit where credit is due?**

   a. Your submission is found to contain code segments copied from on-line resources and you failed to give clear and unambiguous credit to the original author(s) in your source code (-100%)

   b. Your submission is determined to be a copy of another student's submission (-100%)

   c. Your submission is found to contain code segments copied from on-line resources that you did give a clear an unambiguous credit to in your source code, but the copied code constitutes too significant a percentage of your submission:

< 25% copied code	No deduction
25% to 50% copied code	(-50%)
> 50%	(-100%)

What to submit

In a command prompt:

```
      $ cd ~/CSE111/Assignment5
      $ make submit
```

This creates a gzipped tar archive named `CSE111-Assignment5.tar.gz` in your home directory and checks it will execute successfully in the automated grading system.

When you’re happy that your submission is working as expected:

UPLOAD THIS FILE TO THE APPROPRIATE CANVAS ASSIGNMENT.