CSE111 Spring 2022

Assignment 4

In this assignment you will implement a truly parallel C++ multi-threaded most significant digit radix sorter to satisfy complimentary functional and non-functional requirements.

This lab is worth 10% of your final grade.

Late submissions will not be graded.

Background information

To be truly parallel, sorting a *single* list when multiple CPU cores are available should show a significant speedup over a single threaded approach. Radix sorting lends itself to truly parallel implementations; consult the literature for approaches you might consider taking.

Remember that MSD is a sorting *outcome*, not a sorting *algorithm* so investigate sorting algorithms that lend themselves to parallel implementation.

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/Assignment4
$ cd ~/ CSE111/Assignment4
```

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

```
$ tar xvf /var/classes/CSE111/Assignment4.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
```

Note that using shared machines like the CSE111 teaching servers leads to variable results when another user suddenly starts executing their tests whilst yours are running. On the automated grading system, your code will have exclusive access to all 24 CPUs, so will performance far more predictably.

Requirements

Your completed radix sorter must do two things:

1. Correctly MSD radix sort large vectors of unsigned integers
2. Exhibit a significant speedup as more CPU cores are used

In addition, you must:

1. Have no compiler warnings or memory leaks
2. Demonstrate 100% function, line, and branch coverage

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"*, whilst always striving to *"make it right"*.

What you need to do

The class `ParallelRadixSort` is provided, but unimplemented. You need to implement it without changing the existing signature. You can add member variables and functions, but do not change existing signatures or overide the default constructor.

Basic steps are as follows:

1. Investigate how a truly parallel MSD radix sort can be implemented
2. Implement a multi-threaded truly parallel version of `ParallelRadixSort::msd()`

To execute your parallel MSD radix sorter after running `make`:

```
$ ./radix 10000 1 9 -d
```

Where "10000" 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 "9" 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.

A more strenuous test might be:

```
$ ./perf 500000 1 4
```

Which will indicate the speedup achieved (or not) by your multi-threaded implementation. 100% indicates you archived no speedup, 200% indicates you doubled the performance, and so on. Speedups around 300% are easily achievable with simple implementations, anything over 400% will require significant effort and a sophisticated implementation.

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

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%) **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 asses a range of speedups in 50% increments from 150% to 900% inclusive, where the % indicates the benefit of adding additional CPU cores to your solution.

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/Assignment4
$ make submit
```

This creates a gzipped tar archive named `CSE111-Assignment4.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.

