CMPS109 Spring 2018 : Lab 6

In this lab 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 7% of your final grade.

Submissions are due NO LATER than 23:59, Sunday May 13, 2018.

Late submissions will not be graded.

Background information

In Lab 5 you developed an embarrassingly parallel radix sorter when each additional thread simply ran the single-threaded algorithm. By sorting many lists at the same time, an approximately super-scalar speedup was achieved.

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.

Setup

SSH in to any of the CMPS109 teaching servers using your CruzID Blue credentials:

```
$ ssh <cruzid>@<server>.soe.ucsc.edu
```
(use Putty http://www.putty.org/ if on Windows)

Authenticate with Kerberos and AFS: ***(do this EVERY time you log in)***

```
$ kinit
$ aklog
```

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

```
$ mkdir -p ~/CMPS109/Lab6
$ cd ~/CMPS109/Lab6
```

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

```
$ tar xvf /var/classes/CMPS109/Spring18/Lab6.tar.gz
```

Build the skeleton system:

```
$ make
```

Check your implementation's functionality:

```
$ make check
```

Check your implementation's performance:

```
$ make perf
```

Calculate your expected grade:

```
$ make grade
```

Note that using shared machines like the CMPS109 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 perform far more predictably.

What to submit

In a command prompt:

```
$ cd ~/CMPS109/Lab6
$ make submit
```

This creates a gzipped tar archive named `CMPS109-Lab6.tar.gz` in your home directory.

UPLOAD THIS FILE TO THE APPROPRIATE CANVAS ASSIGNMENT.

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

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 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 labs, if there's something you don't understand, do this, in this order:

1. Google it
2. Post a discussion on Canvas
3. Come along to a section and/or 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. (70 Marks) **Does it work?**

    a. Functional Requirement (20 marks)
    b. Non-Functional Requirement (40 marks)
    c. Code free of warnings (10 marks)

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 CMPS109 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:

        - < 50% copied code No deduction
        - 50% to 75% copied code (-50%)
        - > 75% (-100%)

