Homework 1 — Due Monday April 23, 2018, 11 PM

In this programming assignment you will implement Radix Sort, and will learn about OpenMP, an API which simplifies parallel programming on shared memory CPUs. OpenMP is an API which enables simple yet powerful multi-threaded computing on shared memory systems. To link the OpenMP libraries to your C++ code, you simply need to add `-fopenmp` to your compiler flags. You can then specify the number of threads to run with from within the program, or set environment variables:

```
export OMP_NUM_THREADS=4 (for sh/ksh/bash shell, e.g., on cme213-cluster)
setenv OMP_NUM_THREADS 4 (for csh shell)
```

We will cover OpenMP in class. You can learn more about OpenMP at the official website:
`http://openmp.org/`

If you find yourself struggling, there are many excellent examples at:
`https://computing.llnl.gov/tutorials/openMP/exercise.html`

Please do not modify the filenames, Makefile or any of the test files. Only files you need to modify are `main_q1.cpp` and `main_q2.cpp`. Do not forget to set the number of threads before running your program.

Typing `make` will make all the files; typing `make main_q1` will only make the first problem, etc.

Problem 1

In this short problem you will implement a parallel function that sums separately the odd and even values of a vector. The values are of type `unsigned int`.
For example, on

$$\texttt{v} = \boxed{1}\boxed{5}\boxed{2}\boxed{8}\boxed{8}\boxed{1}\boxed{0}\boxed{5}$$

the output should be:

$$\texttt{sums} = \boxed{18}\boxed{12}$$

in which:

$$18 = 2 + 8 + 8 + 0$$

$$12 = 1 + 5 + 1 + 5$$

The starter code for this problem contain the following files (* means that you should *not* modify this file):

- `main_q1.cpp`: This is the file that you will need to modify in this problem. It contains the prototypes for the functions you need to implement.
- *`tests_q1.h`: This is the header file to the test utility functions, e.g., `ReadVectorFromFile`.
- *`tests_q1.cpp`: This contains the implementation of the test utility functions.
- *`Makefile`: To compile just the Problem 1 code, run `make main_q1`. This compiles `main_q1.cpp` (and the file containing the tests). Once the code is compiled, you can run it by typing `./main_q1`. However, to get meaningful performance results of the parallel program, we'll need to run the program on a cluster, e.g., `cme213-cluster`. See Submission Instructions in the Appendix for more details.
- `hw1.sh`: This script is used to submit jobs in the queue for the cme213-cluster (see Appendix B).

Question 1 (18 points)
Implement `serialSum` (for test purposes) and `parallelSum` that compute the sums of even and odd elements Function skeletons have been provided for the same.

Problem 2

In this problem, you will implement Radix Sort in parallel. If you need a refresh on the details of Radix Sort, you should refer to the accompanying Radix Sort Tutorial.

Radix Sort sorts an array of elements in several passes. To do so, it examines, starting from the least significant bit, a group of `numBits` bits, sorts the elements according to this group of bits, and proceeds to the next group of bits.
More precisely:

1. Select the number of bits `numBits` you want to compare per pass.
2. Fill a histogram with `numBuckets` $= 2^{\text{numBits}}$ buckets, i.e., make a pass over the data and count the number of elements in each bucket.
3. Reorder the array to take into account the bucket to which an element belongs.
4. Process the next group of bits and repeat until you have dealt with all the bits of the elements (in our case 32 bits).

Here is the code you are given to get started (* means that you should *not* modify this file):

- `main_q2.cpp`: This contains a serial implementation of lsd radix sort, and shell code for a parallel implementation of radix sort. Do not modify the function signatures, but instead only implement the bodies of the functions. You should modify this file. In particular you should implement: `computeBlockHistograms`, `reduceLocalHistoToGlobal`, `scanGlobalHisto`, `computeBlockExScanFromGlobalHisto`, `populateOutputFromBlockExScan`.
- *`tests_q2.h`: This is the header file to the test functions.
- *`tests_q2.cpp`: This contains the implementation of the test functions.
- *`test_macros.h`: This header files contains some macros that are useful for testing. (If your are using a Windows system or the output looks garbled you will need to uncomment the line `// #define NO_PRETTY_PRINT` for your own debugging).
- *`Makefile`: To compile the code for just problem 2, run `make main_q2`. This compiles `main_q2.cpp` (and the file containing the tests). Once the code is compiled, you can run it by typing `./main_q2`.

To illustrate the role of each function you need to implement, we will use the following example:

`keys` =

001	101	011	000	010	111	110	100

Question 1 (20 points)

Write a **parallel** function `computeBlockHistograms` using OpenMP to create the local histograms. The prototype is given in the starter code. `Test1` should pass if your routine is implemented correctly. This will run a test we implemented. If the code runs with no error, this means that your function was correctly implemented. This is also the procedure we will adopt to grade your code.

Here are some details on the role of `computeBlockHistograms`. We first divide the array into blocks of size `sizeBlock` (here `sizeBlock` = 2).

`keys` =

001	101	011	000	010	111	110	100

block 0 block 1 block 2 block 3

The goal of `computeBlockHistograms` is to create local histograms (a histogram per block). In this case, we use just two buckets (bucket 0 for elements ending with bit 0 and bucket 1 for elements ending with bit 1). The result is:

`blockHistograms` = | 0 | 2 | 1 | 1 | 1 | 1 | 2 | 0 |

block 0 block 1 block 2 block 3

Question 2 (10 points)
Implement a function `reduceLocalHistoToGlobal` that combines the local histograms into a global histogram. The prototype is given in the starter code. `Test2` should pass if your routine is implemented correctly. This will run a test we implemented.
On our example, the output of `reduceLocalHistoToGlobal` should be:

`globalHisto` = | 4 | 4 |

Question 3 (10 points)
Implement `scanGlobalHisto` that scans the global histogram. The prototype is given in the starter code. `Test3` should pass if your routine is implemented correctly. In our case the result of the function is:

`globalHistoExScan` = | 0 | 4 |

Question 4 (12 points)
Implement `computeBlockExScanFromGlobalHisto` that computes the offsets at which each block will write in the sorted vector. The prototype is given in the starter code. `Test4` should pass if your routine is implemented correctly.
In our case the output is:

`blockExScan` = | 0 | 4 | 0 | 6 | 1 | 7 | 2 | 8 |

This means that block 0 will start writing:

- the elements ending with bit 0 at offset 0 in the sorted array
- the elements ending with bit 1 at offset 4 in the sorted array

Question 5 (20 points)
Implement the **parallel** function `populateOutputFromBlockExScan` that populates the sorted array. The function `populateOutputFromBlockExScan` should use the work done in the previous steps to populate the (partially) sorted array. The prototype is given in the starter code. `Test5` should pass if your routine is implemented correctly.
In our case, the result would be:

`keys` = | 000 | 010 | 110 | 100 | 001 | 101 | 011 | 111 |

To get the sorted array, you need to do two others passes on the array.

Question 6 (10 points)
Run Radix Sort for 1, 2, 4, 8, 16, 32, and 64 threads.

For this question, execute `main_q2_part6` instead of `main_q2`, which runs `radixSortParallel` with 1, 2, ..., 64 threads and prints the running time. To build only this executable, you can type `make main_q2_part6` instead of make.

Plot the `running_time` as a function of `number_of_threads`.

Ideally, a sequential program that takes t time to finish should take t/N time if it can be **fully** parallelized among N threads. We define efficiency e

$$e = \frac{\texttt{number\_of\_elements}}{\texttt{running\_time}} \times \frac{1}{\texttt{number\_of\_threads}}$$

to explore this with our Radix Sort program. Plot the efficiency e as a function of `number_of_threads`. Does e remain constant as `number_of_threads` increases as in an "ideal" situation, and why? (Hint: Since it can sometimes be difficult to determine the exact causes for the reduction in efficiency, we will accept any reasonable explanations. You may consider the composition of the program, costs of parallelization, and physical hardware limitations, etc.)

Total number of points: 100

A Submission instructions

To submit:

1. For all questions that require explanations and answers besides source code, put those explanations and answers in a separate PDF file. The name of the file should be: `hw1.pdf`
2. The homework should be submitted using a submission script on `cardinal`. The submission script must be run on `cardinal.stanford.edu`.
3. Copy your submission files to `cardinal.stanford.edu`. You can use the following command in your terminal:
   `scp <your submission file(s)> <your SUNetID>@cardinal.stanford.edu:`
   The script will copy only the files below to a directory accessible to the CME 213 staff. Only these files will be copied. The rest of the files required (tests.cpp's etc.) will be copied by us. Therefore, make sure you make changes only to the files below. You are free to change other files for your own debugging purposes, but make sure you test it with the default test files before submitting. Also, do not use external libraries, additional header files, etc, that would prevent the teaching staff from compiling the code successfully. Here is the list of files we are expecting and that will be copied:

   ```
   main_q1.cpp
   main_q2.cpp
   hw1.pdf
   ```

   The script will fail if one of these files does not exist. Make sure the `hw1.pdf` you upload is the solution file and not the original pdf that comes with the homework!
4. Make sure your code compiles on `cme213-cluster` and runs. We will deduct most points off if your code does not compile on `cme213-cluster`. To check your code, we will run:
   `$ make`
   This should produce 3 executables: `main_q1`, `main_q2` and `main_q2_part6`.
5. Type:
   `/usr/bin/python /usr/class/cme213/WWW/script/submit.py hw1 <directory with your submission files>`
6. You can submit at most 10 times before the deadline; each submission will replace the previous one.
7. There will be a 10% penalty per 24 hours for late submission. We will not accept submissions that are submitted past two days.

B Running on the cme213-cluster

You can find a simple introduction to the cluster on `https://hpcc-intranet.stanford.edu/how-do-i/`. Remember to use VPN before attempting to log in.

After logging in, you may need to load some modules. These are useful commands:

```
$ module list
$ module avail
```

You can copy the load command into your `.bashrc` for convenience.

B.1 Using the queue

For our purposes, it will be sufficient to only use two commands: `sbatch` to submit jobs, and `scancel` to delete jobs.

To run a job with `sbatch`, you must provide a script containing the commands you wish to run. We have provided the script `hw1.sh` for you to run your code on `cme213-cluster`. Note that this script runs all the 3 executables. You can change the script if you would like to run only one. You should change the `WORKDIR` in the script. The command to submit the job script is

```
$ sbatch hw1.sh
```

You can delete a job with

```
$ scancel jobID
```

You can check the status of the queue with either one of these commands:

```
$ squeue
$ squeue -p gpu
$ squeue -u <user name>
```

C Advice

We gather here a few advice for a successful assignment:

- Review the basics of the STL. In particular, you can look at:
  `http://www.cplusplus.com/reference/vector/vector/`
  Make sure to take a look at the methods which return const iterators.
- Review the basic bitwise operations.
- Before you attempt implementing the parallel Radix Sort, make sure that you understand how the serial version works.
- Do not jump straight into the code. Come up first with a strategy to implement parallel Radix Sort and then code it.
- If a part is not working, it is useless to keep going. Always fix the bug(s) before moving to the next part.