Programming Assignment 1 — Due April 17, 2013 12:50 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 (on icme-gpu1)
setenv OMP_NUM_THREADS 4 (on leland machines)
```

We will cover OpenMP in lecture. 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`

See also the documents uploaded on piazza in Resources.

Problem 1

In this short problem you will implement a parallel function that sums separately the odd and even elements of a vector.
For example, on

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

the output should be:

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

The starter code for this problem contain the following files:

- `sums.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.h`: This is the header file to the test functions. You do not need to modify this file.
- `tests.cpp`: This contains the implementation of the test functions. You do not need to modify this file.
- `Makefile`: to compile the code, run `make`. This compiles sums (and the file containing the tests). Once the code is compiled, you can run it by typing `./sums`.

Question 1 (15 points)
Implement `serialSum` (for test purposes) and `parallelSum` that compute the sums of even and odd elements.

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 Radix Sort Tutorial on Coursework.

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:

- `radixsort.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 headers, but instead only implement the bodies of the functions. You should modify this file. In particular you should implement: `computeBlockHistograms`, `reduceLocalHistoToGlobal`, `computeBlockExScanFromGlobalHisto`, `computeBlockExScanFromGlobalHisto`, `populateOutputFromBlockExScan`, `radixSortParallelPass`.
- `tests.h`: This is the header file to the test functions. You do not need to modify this file.
- `tests.cpp`: This contains the implementation of the test functions. You do not need to modify this file.
- `Makefile`: to compile the code, run `make`. This compiles radixsort (and the file containing the tests). Once the code is compiled, you can run it by typing `./radixsort`. You do not need to modify this file.
- `pa1.pbs`: This is used to submit jobs in the queue.

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

$$\texttt{keys} = \begin{array}{|c|c|c|c|c|c|c|c|}\hline 001 & 101 & 011 & 000 & 010 & 111 & 110 & 100 \\ \hline\end{array}$$

Question 1 (15 points)

Write a parallel function `computeBlockHistograms` using OpenMP to create the local histograms. The prototype is given in the starter code. To test that your routine is implemented correctly, uncomment the line: `Test1` in the `main` function of your starter code. 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).

$$\texttt{keys} = \underbrace{\boxed{001}\boxed{101}}_{\text{block 0}}\underbrace{\boxed{011}\boxed{000}}_{\text{block 1}}\underbrace{\boxed{010}\boxed{111}}_{\text{block 2}}\underbrace{\boxed{110}\boxed{100}}_{\text{block 3}}$$

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:

$$\texttt{blockHistograms} = \underbrace{\boxed{0}\boxed{2}}_{\text{block 0}}\underbrace{\boxed{1}\boxed{1}}_{\text{block 1}}\underbrace{\boxed{1}\boxed{1}}_{\text{block 2}}\underbrace{\boxed{2}\boxed{0}}_{\text{block 3}}$$

Question 2 (8 points)
Implement a function `reduceLocalHistoToGlobal` that combines the local histograms into a global histogram. The prototype is given in the starter code. To test that your routine is implemented correctly, uncomment the line: `Test2` in the `main` function of your starter code. This will run a test we implemented. On our example, the output of `reduceLocalHistoToGlobal` should be:

$$\texttt{globalHisto} = \begin{array}{|c|c|}\hline 4 & 4 \\ \hline\end{array}$$

Question 3 (7 points)
Implement `scanGlobalHisto` that scans the global histogram. The prototype is given in the starter code. To test that your routine is implemented correctly, uncomment the line: `Test3` in the `main` function of your starter code. In our case the result of the function is:

$$\texttt{globalHistoExScan} = \begin{array}{|c|c|}\hline 0 & 4 \\ \hline\end{array}$$

Question 4 (10 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. To test that your routine is implemented correctly, uncomment the line: `Test4` in the `main` function of your starter code.
In our case the output is:

$$\texttt{blockExScan} = \begin{array}{|c|c|c|c|c|c|c|c|}\hline 0 & 4 & 0 & 6 & 1 & 7 & 2 & 8 \\ \hline\end{array}$$

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 (15 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. To test that your routine is implemented correctly, uncomment the line: `Test5` in the `main` function of your starter code.
In our case, the result would be:

$$\texttt{keys} = \begin{array}{|c|c|c|c|c|c|c|c|}\hline 000 & 010 & 110 & 100 & 001 & 101 & 011 & 111 \\ \hline\end{array}$$

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. Plot the efficiency e

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

as a function of `number_of_threads`. Comment your result. In particular, discuss what would be the "ideal" case (where parallelization is maximal) and how your plot compares with it.

Question 7 (10 points)
Explain how the construction of the global histogram from the block histograms (`reduceLocalHistoToGlobal`) could be done in parallel. Provide a snippet of code (we do not ask for a running program) that implements your idea.

Question 8 (10 points)
Explain how `computeBlockExScanFromGlobalHisto` could be written in parallel. Provide a snippet of code (we do not ask for a running program) that implements your idea.

Total number of points: 100

A Submission instructions

To grade your paper, we will perform the following steps:

- `make`

- `./sums` (for Problem 1) and `./radixsort` (for Problem 2)

Please follow the submission instructions to help facilitate the grading process. To submit follow these steps:

1. Create one zip file with the name: `PA1.zip`. This file should contain one directory named `FirstName_LastName_SUNetID_PA1`[1]. The folder should contain the folders `radixsort` and `parallelsum` (each one of these folders contains your code for the problem) and a file called `readme.pdf`
2. Submit this single file on Coursework. Go to Assignments and upload your file in PA1.

Remember that Coursework records the time at which you submit and allows multiple submissions.

B Hardware

We will use `icme-gpu1`, even though we are now only taking advantage of its CPU capabilities. You may directly run your code on any of the nodes or you may use the torque queuing system.

B.1 Getting assigned to a node

If you want to run your code interactively, you need to run:

```
qsub -I
```

You will be assigned to a node where you can run your code without enqueuing your jobs. This mode should be more convenient to write / debug your code because contrary to the queue, you do not need ot open files to look at the results of your computation.

B.2 Using the queue

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

To run a job with `qsub`, you must provide a script containing the commands you wish to run. We have created an example script in `pa1.pbs`. To run this job, simply execute:

```
qsub pa1.pbs
```

If you would like to request a specific number of processors, you can specify this at the command line

```
qsub pa1.pbs -l nodes=1:ppn=#
```

Where # is the number of cores you would like to occupy on the node.

You may need to update the `$USER` or `$pa1_home` values within the script if you have configured your system with something other than your username and put your code in something other than `/home/YOUR_SUNET_ID/pa1/NAME_OF_PROBLEM`.

The output will be stored in a file named `<script>.o.<jobID>` and `<script>.e.<jobID>`. For example, if this was job 15, the standard output would be placed into `pa1.pbs.o15` and standard error in `pa1.pbs.e15`. You should not be executing more than five jobs concurrently, or else you may get kicked off the cluster.

You can delete a job by running

```
qdel jobID
```

You can check the status of the queue with

```
qstat
```

Torque is designed around throughput, not latency, so an individual job may take longer to run than you would expect. We encourage you to develop locally and only test with `qsub` once you have a final version of your program.

[1]For instance, in my case the folder would be named: `Sammy_ElGhazzal_selghazz_PA1`

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/`
- 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.