CME 213, Introduction to parallel computing
Eric Darve
Spring 2024

Stanford University

Homework 7

Total number of points: 100.

In this programming assignment, you will implement Radix Sort, and will learn about OpenMP, an API which simplifies parallel programming on shared memory CPUs.

Read carefully the tutorial on Radix Sort. You are welcome to check online resources as well. The algorithm is not simple so make sure you spend some time reviewing it before starting the programming assignment.

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)
setenv OMP_NUM_THREADS 4 (for csh shell)
```

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

If you find yourself struggling, there are many excellent examples at:
`https://hpc-tutorials.llnl.gov/openmp/`

Please do not modify the filenames, Makefile or any of the test files. 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.

We provide script `hw7.sh` to compile and run all executables.

For the sake of uniformity in all students' results, you must run your code using the teaching cluster.

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):

- `sum.h`: This is the file that you will need to modify in this problem. It contains the prototypes for the functions you need to implement.
- *`main_q1.cpp`: This is the file that uses the functions you wrote and tests them.

- *`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`.

The questions in this homework can be completed using parallel openMP `for` loops and the reduction construct when required.

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 refresher 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^{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):

- `parallel_radix_sort.h`: This contains helper functions used for serial and parallel implementations 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`.
- *`main_q2.cpp`: This contains the serial lsd radix sort, and benchmark codes.
- *`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`).

$$\texttt{keys} = \underbrace{\begin{array}{|c|c|}\hline 001 & 101 \\ \hline\end{array}}_{\text{block 0}}\underbrace{\begin{array}{|c|c|}\hline 011 & 000 \\ \hline\end{array}}_{\text{block 1}}\underbrace{\begin{array}{|c|c|}\hline 010 & 111 \\ \hline\end{array}}_{\text{block 2}}\underbrace{\begin{array}{|c|c|}\hline 110 & 100 \\ \hline\end{array}}_{\text{block 3}}$$

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:

$$\texttt{blockHistograms} = \underbrace{\begin{array}{|c|c|}\hline 0 & 2 \\ \hline\end{array}}_{\text{block 0}}\underbrace{\begin{array}{|c|c|}\hline 1 & 1 \\ \hline\end{array}}_{\text{block 1}}\underbrace{\begin{array}{|c|c|}\hline 1 & 1 \\ \hline\end{array}}_{\text{block 2}}\underbrace{\begin{array}{|c|c|}\hline 2 & 0 \\ \hline\end{array}}_{\text{block 3}}$$

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.
In our example, the output of `reduceLocalHistoToGlobal` should be:

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

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:

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

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:

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

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 (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)

In `main_q2.cpp`, you have code that is inside blocks like

```
#ifdef QUESTION6
...
#endif
```

The macro `QUESTION6` is used to repeatedly run the parallel sorting function with different configurations.

The unit test named `radixSortTest` initializes a single vector `keys_orig` with the original, unsorted input data. It then runs the benchmark for different combinations for number of threads and number of blocks. Complete the `runBenchmark` function in `parallel_radix_sort.h` by filling in the `TODO`s.

Run this code (independently or using the common `hw7.sh` script) that varies the block size and the number of threads to use. This should be run on ICME GPU cluster for consistency of results. The terminal output should show that the unit tests all passed.

Turn in the code you used to run these benchmarks.

Print the running times you obtain in the form of a table where each row corresponds to a number of threads and each column to a number of blocks.

Comment on your result. For example, what is the optimal number of threads and blocks? How does the timing vary as a function of the number of threads and blocks? Be aware of the fluctuations and uncertainty in your timings.

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 and upload this file on gradescope. The name of the file should be: `hw7.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`. The script `submit.py` 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:

```
sum.h
parallel_radix_sort.h
```

The script will fail if one of these files does not exist.

4. To check your code, we will run:
   `$ make`
   This should produce 3 executables: `main_q1`, `main_q2` and `main_q2_part6`.

5. To submit, type:

```
$ /afs/ir.stanford.edu/class/cme213/script/submit.py hw7 <directory with your submission files>
```

B Advice

We gather here a few advice for a successful assignment:

- Review the basics of the STL. In particular, you can look at:
  `https://en.cppreference.com/w/cpp/container/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. First come up 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.