System Programming

Programming Assignment 4

Fall 2015

Due: 23:59 Mon, Jan 18, 2016

1 Problem description

Sorting is one of the most fundamental algorithms and frequently used functions for data processing. In many real world applications, like a *bidding system*, the size of the data file can range from hundreds of megabytes to terabytes. As a result, the data to be sorted may be too large to be read into process memory. How to efficiently sort such huge data set is critical to application performance. One plausible solution is merge sort.

In this assignment, your job is to write a merge sort program using **multiple threads**, which reads numbers from standard input, sorts them, and outputs the sorted results. The execution time using merge sort is significantly less than using a normal sorting algorithm.

2 Sorting

All of you should be familiar with the merge sort algorithm, we show a case here to explain the requirements in this assignment.

Suppose the input sequence is as below:

37 25 28 30 32 26 28 49 28 31 15 32 22 39 2 10 4 25 49 47

As input to this assignment, you will be given an argument specifying *[segment_size]*, the size of segment that the data should be split up. For example, if 4 is given as *[segment_size]*, indicating that the data should be split into segments sized 4, producing 5 segments. *Note that if the input size is not divided by 4, the size of the last segment will be less than 4.* The data in this case will be split in the following way:

Seg#1	Seg#2	Seg#3	Seg#4	Seg#5
37 25 28 30	32 26 28 49	28 31 15 32	22 39 2 10	4 25 49 47

The first part of merge sort requires you to sort each individual sort, in parallel. In our example, this means that you will launch five threads, one for each of the segments, and each thread must sort only their segment of the input, being destroyed after they finish sorting. You don't have to write your own sort here; in fact, you should use the C function qsort() or STL function sort().

After the sorting threads of first round have all completed, the data will look like:

Seg#1	Seg#2	Seg#3	Seg#4	Seg#5
25 28 30 37	26 28 32 49	15 28 31 32	2 10 22 39	4 25 47 49

And each of the threads need to output some information. For example, the thread handling Seg#1 should output the following lines:

```
Handling elements:
37 25 28 30
Sorted 4 elements.
```

Next, the merge sort will proceed several rounds to merge them together.

At each round you should launch $\lfloor \#segments/2 \rfloor$ threads to merge adjoining pairs of segments in parallel namely Seg#{n} and Split#(n+1) for all odd valued n, where #segments is the number of segments you had **at the beginning of this round**. If there is an odd number of segments in a round, you should not merge the last segment in the current round, but simply advance it to the next round of merging. This round must block until all threads finish merging. After this round, you will have [#segments/2] or [(#segments/2) + 1] segments remaining. You must proceed several rounds until only one single, completely sorted list remains.

Remember that merging two splits here is an O(n) operation not O(nlog(n)), so you shouldn't call qsort() or sort() here. Instead, think of how you can merge two sorted lists in O(n) time scan both lists just one time, and keep extracting the smaller numbers to your result until all numbers are extracted so that your sorting is done. To let us know what you're doing, you are required to keep track of how many duplicates you've seen between the two segments while merging two lists. You should count one duplicate only when you find a number appears in both splits **NOT** if a segment contains the same number multiple times.

In the previous example, the first round consists of two parallel merges: (Seg#1 and Seg#2) and (Seg#3 and Seg#4).

Seg#(1+2)	Seg#(3+4)	Seg#5
25 26 28 28 30 32 37 49	2 10 15 22 28 31 32 39	4 25 47 49

The sort of Split#(1+2) will generate an output:

```
Handling elements:
25 28 30 37 26 28 32 49
Merged 4 and 4 elements with 1 duplicates.
```

,where "1 duplicates" is due to 1 same element across Seg#1 and Seg#2.

The sort of Split#(3+4) will generate an output:

```
Handling elements:
15 28 31 32 2 10 22 39
Merged 4 and 4 elements with 0 duplicates.
```

,where "0 duplicates" is due to no same elements across Seg#3 and Seg#4

The second round consists of only one merge:

Seg#(1+2+3+4)	Seg#5
2 10 15 22 25 26 28 28 28 30 31 32 32 37 39 49	4 25 47 49

The sort of Split#(1+2+3+4) will generate an output:

```
Handling elements:
25 26 28 28 30 32 37 49 2 10 15 22 28 31 32 39
Merged 8 and 8 elements with 3 duplicates.
```

,where "3 duplicates" is due to 28 and 32 being across both segments, and there are **two** 28's in the left comparing the one in right

Note that:

1. the two 28's within the same segment will not be counted as a duplicate
2. take the number **in the left segment** into sorted list if the two comparing numbers are equal, hence there are 2 dup for the number 28

And the final round also consists of only one merge:

Seg#(1+2+3+4+5)
2 4 10 15 22 25 25 26 28 28 28 30 31 32 32 37 39 47 49 49

The sort of Split#(1+2+3+4+5) will generate an output:

```
Handling elements:
2 10 15 22 25 26 28 28 28 30 31 32 32 37 39 49 4 25 47 49
Merged 16 and 4 elements with 2 duplicates.
```

,where "2 duplicates" is due to 25 and 49 being across both segments.

You should know the benefit of merge sort, is that you don't have to load the whole data into the memory compared to a regular sort requiring the whole data in the memory. While a regular sorting algorithm will crash when the data is too huge to be loaded, merge sort does not crash, however. Merge sort requires only a segment to be loaded to apply qsort(), and the merging stage, as the segment size becomes larger, still does not need the whole segment to be loaded into memory because it only applies a linear scan, so only the portion around the scanning cursor is required to be loaded. **However, for the easiness of testing, you CAN read the whole data into memory in this assignment.**

3 Format for inputs and outputs

One program is required: "merger".

Your program requires *[segment_size]* as the only argument, specifying the number of each split you have to use.

The merger will first read an integer from standard input, specifying the number of integers to be sorted. And the merger should continue reading from standard input as many integers as specified. All input integers would fit in a 32 bit signed integer, and the number of integers would be less than $5x10^7$.

The merger should implement merge sort with segment size as specified. At the beginning of the merge sort, each qsort() or sort() called within a split should output the following formatted text:

```
Handling elements:
[numbers here]
Sorted [#numbers in the segment] elements.
```

In the merging stage, every merge of **Seg#n** and **Seg#(n+1)** of odd number n generates an output:

```
Handling elements:
[numbers here]
Merged [#numbers in Seg#n] and [#numbers in Seg#(n+1)] elements with [#dup] duplicates.
```

where #dup is the number of elements that appear in **both** segments in this merge.

In the end, output the final sorted result in a line.

4 Sample execution

```
$ ./merger 6 < priceInfo.txt
```

This indicates merger reads integers from priceInfo.txt, which contains 32 bit integers in plain text format, and splits them into 6 sized segments.

Suppose the content of testdata is

```
20
37 25 28 30 32 26 28 49 28 31 15 32 22 39 2 10 4 25 49 47
```

Hence, 4 threads are required to sort them, several other threads to merge them, and so on.

The output will be:

```
Handling elements:
37 25 28 30 32 26
Sorted 6 elements.
Handling elements:
22 39 2 10 4 25
Sorted 6 elements.
Handling elements:
28 49 28 31 15 32
Sorted 6 elements.
Handling elements:
49 47
Sorted 2 elements.
Handling elements:
25 26 28 30 32 37 15 28 28 31 32 49
Merged 6 and 6 elements with 2 duplicates.
Handling elements:
2 4 10 22 25 39 47 49
Merged 6 and 2 elements with 0 duplicates.
Handling elements:
15 25 26 28 28 28 30 31 32 32 37 49 2 4 10 22 25 39 47 49
Merged 12 and 8 elements with 2 duplicates.
2 4 10 15 22 25 25 26 28 28 28 30 31 32 32 37 39 47 49 49
```

Since the program is multi-thread, the output may not be the same order as the output here.

5 Grading

There are 3 subtasks in this assignment. By finishing all subtasks you earn the full 9 points.

1. Completeness (1 point)
   That is, produce merger by Makefile. (DO $ make before you upload!)

2. Correctness (5 points)
   We will compare your results with the correctly sorted result and existing benchmark programs.
   For the sorting correctness, you will get 1 point for each testing data if your program finishes the sorting task correctly.

3. Output format. (1 point)
   Any of your output of one thread should not be interrupted by other threads, that is, your output must be readable.
   Also, please DO NOT output any other unnecessary text to stdout

4. Report (2points)
   We will give credits based on the content in your Report.pdf, please see "Submission" part and slides for details.
   (Excellent design on experients will help you get bonus credits :-))

6 Submission

Your assignment should be submitted to the CEIBA before the deadline. Or you will receive penalty.

At least three files should be included:

1. merger.c (or cpp)
2. Makefile
3. Report.pdf

Since we will directly execute your *Makefile*, therefore you can modify the names of .c files, but *Makefile* should compile your source into one executable named **merger**.

Please generate random test data of size n=100, 10000, 1000000, 10000000.

For each of the data, use segment_size=n/100, n/25, n/10, n/5, n/2 and n accordingly (that is, number of segments would be 100, 25, 10, 5, 2, 1, accordingly). There would be 24 combination in total. You should use linux utility time or any time measurement methods to measure the execution time for each combination, and write them in your Report.pdf.

Briefly state your finding, that is, how's the (real time/user time) affected by number of segments, and why?

Please do NOT add your testdata into the compressed file.

Any optimization parts in your code or other additional designed experiments that worth mentioning are also welcome.

These files should be put **inside a folder named with your student ID (in lower case)** and you should compress the folder into a *.tar.gz* before submission. Do not use .rar or any other file types.

The commands below will do the trick. Suppose your student ID is b03902000:

```
$ mkdir b03902000
$ cp Makefile Report.pdf *.c *.cpp b03902000/
$ tar -zcvf SP_HW4_b03902000.tar.gz b03902000/
$ rm -r b03902000/
```

Please do **NOT** add executable file and testdata files to the compressed file. Errors in the submission file (such as files not in a directory named with your student ID (in lower case), compiled binary not named merger, and so on) may cause deduction of your credits.

Submit the compressed file to CEIBA.

7 Notes

You may try to simply use normal sorting methods such as qsort() to deal with the problem, instead of using merge sort. Since it is noticeable from the execution time that whether you apply merge sort or not when the input size is large, we could see the execution time to tell whether you implement merge sort or not. Of course, we can also look at the source code, so please implement the algorithm stated above, or you will lose credits. When measuring time for your program, remember to redirect output to file or /dev/null, since output large amount of data to console is very slow.

8 Reminder

1. Plagiarism is STRICTLY prohibited.
2. We are approaching the end of semester, hence **late submission is not allowed in this assignment**.
3. If you have any question, please ask questions at the board SysProgram on PTT2, contact us via email or come to R302.
   Check whether your questions has been asked on P2 before you send an email or come.
4. Please start your work ASAP and do not leave it until the last day!