S3774430 Myeonghoon Sun https://github.com/hooniez/osp_assessment1

Assignment 1

All of the tasks were implemented before the updated assessment specification was made available after the due date had changed and a week before the extended due date. I have done as best as I can despite the lack of response to my questions put forth in the discussion forum. Where the course coordinator has failed to clarify the unclear assessment specification, I have spent an extraordinary amount of time trying to guess what is meant by each sentence in the ambiguous assessment specification that hasn't even been proof-read.

The Source Data

I have downloaded all the 12 files, and combined them into *combined.txt* in *original_data* so that it can be used as input data for Task 1 (the input data does have duplicate words).

Task 1

In the original assessment spec, there is no mention of how the task 1 shell script should be written (in terms of whether it should force the user to pass arguments in the command line). As such, please run Task1.sh without passing any argument. If you have to change input or output file names, edit the file.

Task 1's implementation is correct because the task 2 specification requires the student to ignore words of lengths shorter than 3 characters and longer than 15 characters. Thus, after the task 1 is run, these words should be present in order for it to be ignored. There was no mention of sorting whatsoever in the original specification. I have sorted the words only to remove duplicates.

- the filtering rule: select words that are made solely of English letters without special symbols such as the apostrophe, comma, and hyphen.

- the coreutils tools used: awk (to use regex to apply the filtering rule) and sort (to remove duplicates)

- the number of words of length 3 to

S3774430 Myeonghoon Sun https://github.com/hooniez/osp_assessment1

15 letters in the dataset: 1,367,007 words

- the performance on my machine (Intel(R) Core(TM) i7-4710MQ CPU @ 2.50GHz):

- time ./Task1.sh => 5.319s (an average time of 5 executions)
- time ./Task1Filter ../original_data/combined.txt ../input.txt => 5.751s (an average time of 5 executions)

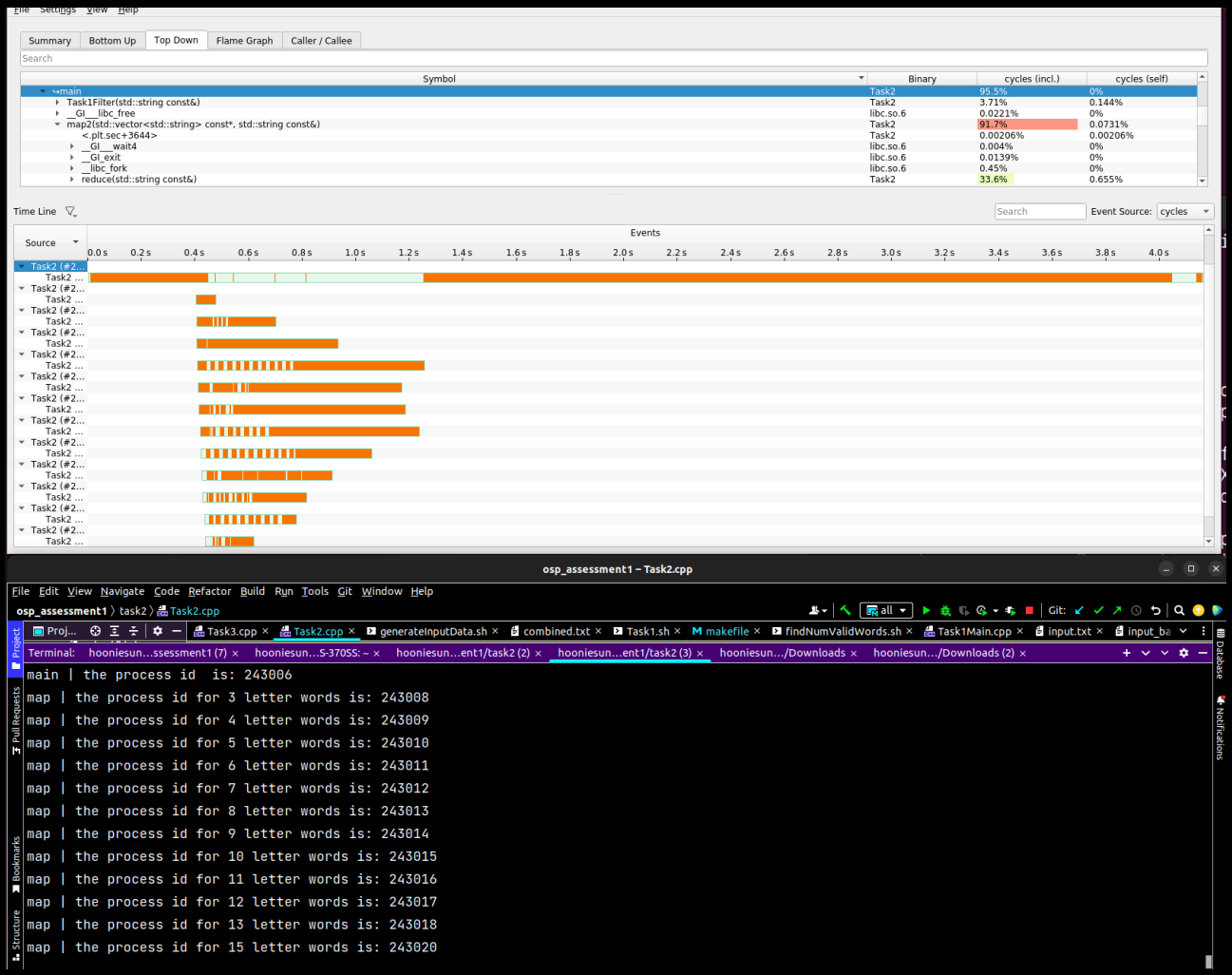
Many hours were spent to come up with a way to shuffle both of the filtered datasets such that they are identical. Because of the limitation not to use the same coreutils tools used in the bash script in the corresponding C program, I couldn't just use the system() call in my C program to shuffle identically. I finally devised a simple yet effective way to shuffle by creating a 2d array to store the filtered data and then transposing it both in the bash script and the C program only to be told by the course coordinator that shuffling is in fact unnecessary, which is at odds with the specification. I believe this implementation, which is now commented out, be recognised as a deed exceeding the assignment requirements.

Task 2

- the performance (I have chosen to use the perf tool because it allows for profiling on per-thread and per-process basis, which is necessary to optimise thread performance as required in Task 4; *perf record* collects samples for event *cycles*, which then can be analysed to find out the workload of each thread or process):

- Run Time: 3.962s

As seen below, it takes twice as much of the total clock cycle for the map2() to sort and write outputs as for the reduce2() to perform 13 to 1 merge operations; a 91.7% of the total clock cycle can be attributed to operations within map2() in which reduce2() is also called with its share being 33.6% of the total clock cycle.

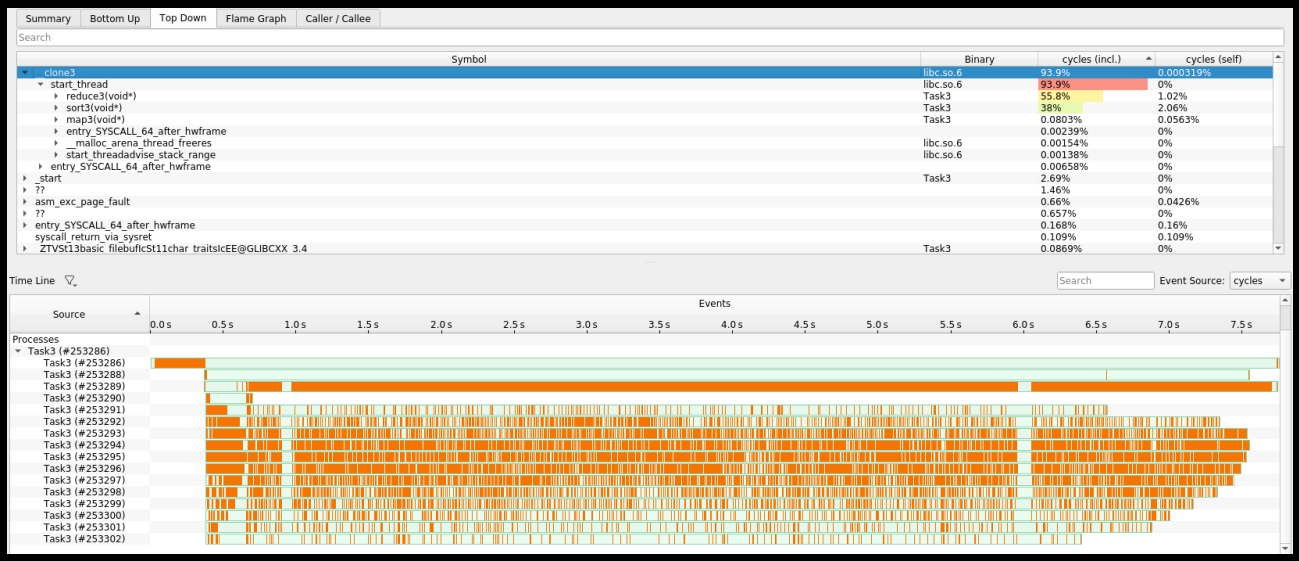


Task 3

- the performance:

- Run time: 7.761s

As seen below, the reduce3() accounts for 55.8% of the total clock cycle, which is a significantly greater proportion than the reduce2()'s share of its own clock cycle. By the design of the program, reduce3() starts its operation only when every thread spawned from map3() finishes its sorting.



	name	tid	Cycles before FIFO	Cycles after FIFO
0	main	253286	3.58	0.08
1	map	253288	0.67	0.00
2	reduce	253289	0.01	69.50
3	sort	253290	0.74	0.25
4	sort	253291	5.89	1.22
5	sort	253292	9.73	2.46
6	sort	253293	12.09	4.04
7	sort	253294	13.93	4.62
8	sort	253295	10.86	4.68
9	sort	253296	12.57	4.11
10	sort	253297	9.21	3.11
11	sort	253298	7.38	2.27
12	sort	253299	6.04	1.57
13	sort	253300	3.27	1.02
14	sort	253301	2.90	0.67
15	sort	253302	1.91	0.41

From the left screenshot, it is clear that some threads require more time to sort their respective index array as indicated by the column "Cycles before FIFO" (this information was derived from the visualisation above coupled with the command *perf report -s pid –time x%-y%* where x is the starting time and y is the finishing time). The rightmost column shows that reduce3() will be the workhorse when every FIFO file is open for writing and reading.

As Ron has noted in the discussion forum, the single-threaded reduce solution is good as long as it works (Paul also confirmed the reduce function is supposed to be single-threaded). For this reason, the subsequent tasks (3, 4, and 5) all execute the reduce function in a single thread.

Task 3 is the only task that has implemented the graceful exiting since implementing it in all the subsequent tasks could make code hard to read. Also the program is carefully designed so that it ends within 10 seconds, making the graceful exiting somewhat unnecessary. Regardless, to see the normal execution without forced waiting in the main, please comment out the lines from 280 to 284.

Task 4

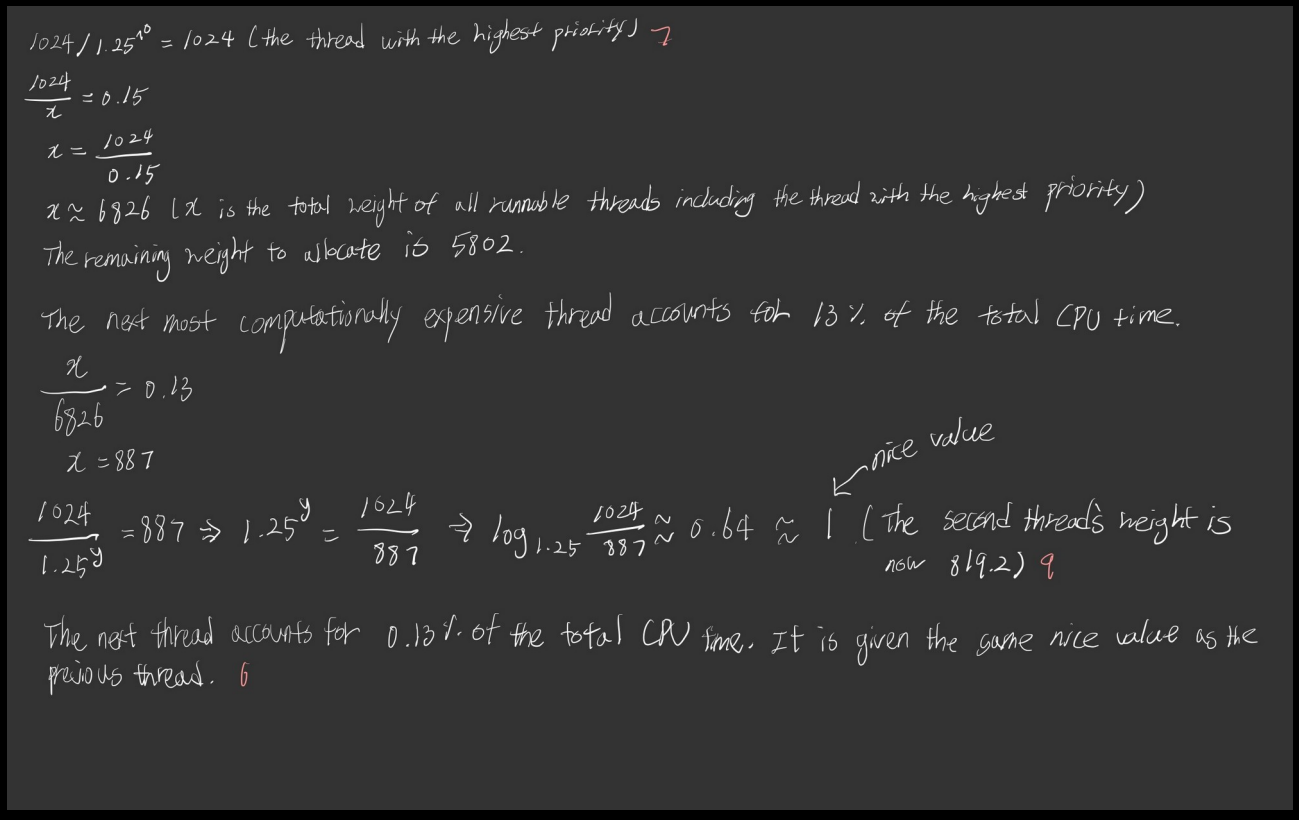
- the performance:

- Run time: 5.994s

As per the table above as well as the total number of permissible nice values a user can set being 20 (0 to 19), we can derive nice values for the threads spawned from map4() in the initial sorting phase. To begin with, the 7-letter-sorting thread will receive the nice value of 0 as it requires the largest timeslice. The nice function has been chosen as the setpriority function requires sudo privilege (originally pthread scheduling functions were used, but they were replaced by the nice function since the submitted code should work on titan according to the original assessment specification).

To decide on reasonable nice values to tweak later on, the following knowledge was applied: the Linux Completely Fair Scheduler calculates a weight based on the niceness. The weight can be calculated using the formula: 1024 / (1.25 ^ niceValue). The timeslice given to a process is in proportion to the weight of the process divided by the total weight of all runnable threads.

The images below detail the reasoning behind the initial nice values for the threads (each number in pink represents the identical number of letters in words assigned to each thread [e.g. 7 below represents a thread that is given an index array for 7-letter words]).



The next thread's ratio of the total clock cycle : 0.11

$\frac{x}{6826} = 0.11 \Rightarrow x \approx 751 \Rightarrow \frac{1024}{1.25^y} \approx 751 \Rightarrow 1.25^y = \frac{1024}{751} \Rightarrow \log_{1.25} \frac{1024}{751} \approx 1.4$ 8

the nice value 1 is given.

Current remaining weight : 3344

the next thread's ratio of the total clock cycle : 0.10 (nice value 2) 5

Current remaining weight : 2688

the next thread's ratio of the total clock cycle : 0.09 (nice vale 2) 10

Current remaining weight : 2032

The next thread's ratio of the total clock cycle : 0.07

$\frac{x}{6826} = 0.07 \Rightarrow x \approx 477 \Rightarrow \frac{1024}{1.25^y} \approx 477 \Rightarrow 1.25^y = \frac{1024}{477} \Rightarrow \log_{1.25} \frac{1024}{477} \simeq 3$ 11

Current remaining weight : 1507

the next thread's ratio of the total cycle : 0.06 nice value 4 12

Current remaining weight : 1087

The next thread's ratio of the total cycle : 0.06 nice value 4 4

Current remaining weight : 667

The next thread's ratio of the total cycle : 0.03 nice value 7 13

Current remaining weight : 452

The next thread's ratio of the total cycle : 0.03 nice value 7 14

Current remaining weight : 237

The next thread's ratio of the total cycle : 0.02 nice value 9 15

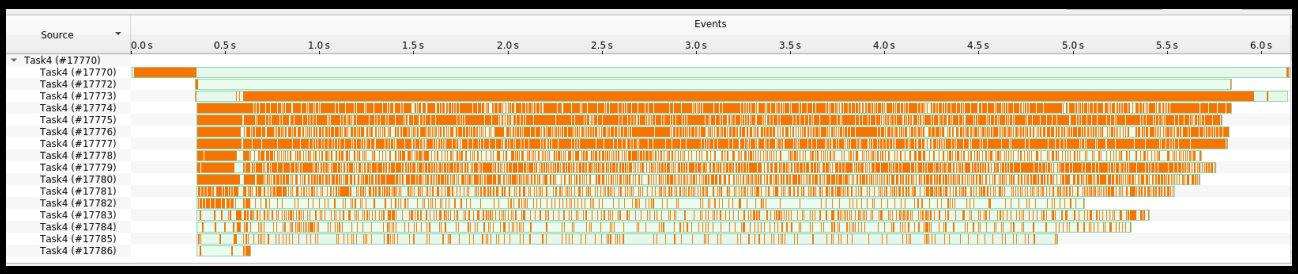
Current remaing weight : 99

The next thread's ratio of the total cycle : 0.01 nice value 12 3

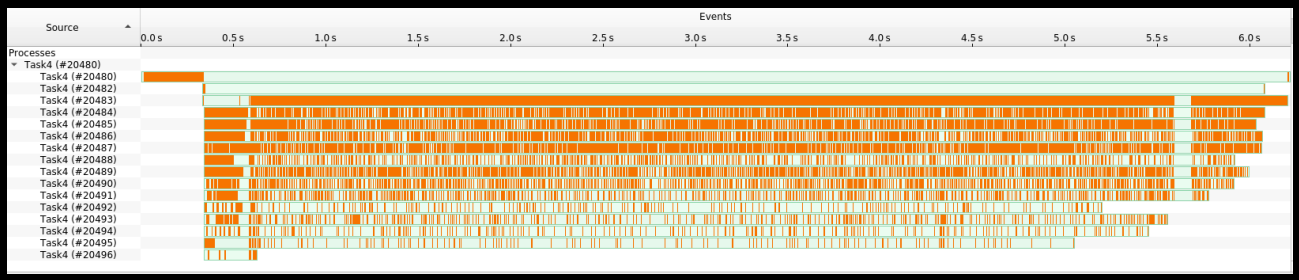
Current remaining weight : 22

The finalised nice values applicable to the initial sorting phase can be found in the map data structure named niceValueBeforeFIFO in Task4.cpp. After the index arrays are all sorted, the reduce4 does all the CPU-bounded work whereas the threads spawned from map4() are IO bounded. For this reason, IO bounded threads are given the nicest value of 19 to let the reduce4() thread claim more CPU time.

As seen from the screenshot below, all the threads in the phase of sorting index arrays now finish at about the same time thanks to the tweaked nice values.



On the other hand, the screenshot below shows the result from the initial nice values. We can see that many threads roughly finish at the same time except for the thread #20495, 20496, and a few others. After changing the nice values for these threads and others, the screenshot above illustrates all the threads can finish at about the same time.



As seen in the screenshots above, the majority of the program's running time is spent in reduce4() in which words are read from FIFO files and then compared to determine the lowest word in lexical order. Therefore, all the changes so far made in places other than reduce4() have brought about improvement too small to contribute to the overall performance.

As for the last question "w hat is the time complexity of the program?", we can divide the program into two parts where most computationally expensive operations execute: one before FIFO files are open for writing (during which the sorting of index arrays takes places) and the other after FIFO files are read (during which the files are read and the 13 words at a time are sorted).

1. O(nlogn) – n words are divided into 13 parts and the largest part determines the time complexity; the largest part can be no greater than n.
2. O(n) – 13 words are sorted *n* number of times (n*(13log13))

In conclusion, the multi-threaded programming has helped decrease the role the first part plays in the time complexity of the whole program, as this most computational work is now divided across multiple threads. The time dominating time complexity is O(n) with the caveat that the constant in front of n is large, resulting in the time complexity of O(nlogn) in practice. The empirical performance does not reflect the theoretical time complexity as it would in single-thread programming since these threads run with varying priorities.

Task 5

In the discussion forum , I have left a long list of questions regarding the task 5 only to be ignored. Lack of response has led me to spend an extraordinary amount of time trying to get at what needs to be done despite the extremely unclear assessment spec. I have tried to implement the task 5 in the same way as the previous tasks while meeting the ambiguous requirements as best as I can.

On a stream size of 300,000 words with a delay of 1000 microseconds in between streams, the runtime of the program is 11.634s.

- cat - | sort (cat's output becomes input of sort; when all of the lines are read [a binary tree is built with each node containing a block of data and sorting it in a multi-threaded fashion and, all the nodes then get merged to the root node], sort then sorts them, after which you "get to see the output")
- In Task 5, each block of data is streamed from the stream server to the map thread. Since words are randomly picked to be streamed, the number of words for each length can initially differ from each other significantly and, hence, nice values are more likely to fluctuate as the greater number of words of a particular length, the higher priority should it be given per stream. However, as more streams are processed, the nice values become less prone to change as the cumulative percentages representing the distribution of cumulative words stay constant.
- Blocking the data (streaming) requires the use of conditional variables to prevent the map thread from busy waiting and wait for a conditional signal from the stream server.

In Task 5, raw data is fed through std::cin and filtered using Task1Filter. It is then shuffled to allow for random entries to be selected in a stream of data to the map thread. The streaming server is connected to the map thread via a FIFO file. A stream size and delay seconds can be changed to enable streaming at a constant rate. The map method does not engage in busy waiting for the streaming server to write by keeping the FIFO open, but rather waits for a conditional signal from the streaming server if it finishes processing first (unlikely as the map maps entries in addition to reading). In most cases, the streaming server will finish writing first and then wait for a conditional signal from the map to sleep for a certain amount of time before streaming again. Well-coordinate use of conditional variables has helped streamline this process so that no deadlocks occur. As in the previous tasks, the map thread maps the incoming streaming to appropriate worker threads, which are spawned from it. Because now there are blocks of data to process, each worker thread first takes one block at a time and sorts it as before. What differs this time is that at the end of processing, the worker thread will need to reduce vectors of sorted words into one vector that can be further reduced into one output with other vectors reduced from worker threads. From this point on, the rest

of the steps are identical. When it comes to scheduling, because of a varying number of words each worker thread will be assigned to process, nice values are updated according to the formula used in Task 4 per stream.