**Parallel Log Likelihood**

In the syllabus this was listed as a parallel zip program with a difficulty rating of 3. Today, in an effort to minimize the likelihood of finding complete solutions on the web, I started looking at other compression algorithms. My goal was to find one that

* was a bit more obscure,
* that was straight forward to understand,
* that there would be some obvious way to parallelize it.

Meeting those requirements proved quite difficult, so I switched to this task, which has the same objectives (plus I am hoping you will find it fun)

**Objectives**

There are three specific objectives to this assignment:

* To familiarize yourself with the Linux pthreads.
* To learn how to parallelize a program.
* To learn how to program for high performance.

**Background**

To understand how to make progress on this project, you should first understand the basics of thread creation, and perhaps locking and signaling via mutex locks and condition variables. These are described in the following book chapters:

* [Intro to Threads](http://pages.cs.wisc.edu/~remzi/OSTEP/threads-intro.pdf)
* [Threads API](http://pages.cs.wisc.edu/~remzi/OSTEP/threads-api.pdf)
* [Locks](http://pages.cs.wisc.edu/~remzi/OSTEP/threads-locks.pdf)
* [Using Locks](http://pages.cs.wisc.edu/~remzi/OSTEP/threads-locks-usage.pdf)
* [Condition Variables](http://pages.cs.wisc.edu/~remzi/OSTEP/threads-cv.pdf)

Read these chapters carefully in order to prepare yourself for this project.

It may also be useful to complete the [HackerRank Ransom Note Problem](https://www.hackerrank.com/challenges/ctci-ransom-note/problem). If you show me your hackerrank score for that problem I will give you that in xp.

**The task**

Ted Dunning wrote an often cited and simply wonderful paper [Accurate Methods for the Statistics of Surprise and Coincidence](http://aclweb.org/anthology/J93-1003)that argued that many of the statistics people use for text analysis are not accurate. In that paper he proposes a method called log likelihood. In the paper he used it to find surprising bigrams (two word combinations). We can also use this method to find out how 'surprising' a word is in one text compared to another. For example, I compared speeches Trump gave when running for president to those given by Obama. Here are the top words sorted by log likelihood:

| **word** | **log likelihood** | **Trump or Obama** |
| --- | --- | --- |
| going | 274.71 | T |
| I | 249.47 | T |
| very | 163.82 | T |
| who | 117.30 | O |
| the | 114.38 | O |
| really | 83.01 | T |

We interpret this as meaning that it is somewhat surprising how many times Trump used the word *going* given how often that word occurs in Obama's speeches. It is also surprising that Trump uses *I* and *very* as often as he does. On the other hand, it is surprising that Obama uses *who* and *the* as often as he does given how often those words appear in Trump's speeches.

**8 years ago in PHP**

8 years ago, I implemented this in PHP. (that was back in the day when PHP was cool). The sites is [Online Text Comparator](http://guidetodatamining.com/ngramAnalyzer/comparator.php). The source code is in [this github repo](https://github.com/zacharski/ngramAnalyzer). On the bottom of the webpage you can try some texts I uploaded and compare the Lotus Sutra to Walden, for example.

When Henry David Thoreau, the author of Walden, was 26, he translated part of the Lotus Sutra and had it published in Ralph Waldo Emerson’s literary journal, *The Dial*. I was curious about the relationship between these two texts.

**Your task: a parallel implementation.**

You are to implement the formula Dr. Dunning gives to compare 2 texts. For example:

vagrant:/log-likelihood lotus.txt walden.txt > output.csv

All output should be written to standard output. The output should be sorted by the log likelihood and be in the following comma-separated format. (The column names must be those specified).

Word,lotus.txt count,walden.txt count,lotus.txt freq,walden.txt freq,Log Likelihood

Buddha,1047,0,0.00959,0.00000,1504.60215

Law,584,1,0.00535,0.00001,831.87029

will,1297,272,0.01187,0.00235,793.336213

beings,568,4,0.00520,0.00003,779.709311

which,88,860,0.00081,0.00743,687.134838

Buddhas,466,0,0.00427,0.00000,662.712284

I,755,1988,0.00691,0.01718,511.937387

a,1431,2950,0.01310,0.02550,462.810632

sutra,325,0,0.00298,0.00000,459.207565

One,511,55,0.00468,0.00048,451.427785

which looks like the following in table form.

| **Word** | **lotus.txt count** | **walden.txt count** | **lotus.txt freq** | **walden.txt freq** | **Log Likelihood** |
| --- | --- | --- | --- | --- | --- |
| Buddha | 1047 | 0 | 0.00959 | 0.00000 | 1504.60215 |
| Law | 584 | 1 | 0.00535 | 0.00001 | 831.87029 |
| will | 1297 | 272 | 0.01187 | 0.00235 | 793.336213 |
| beings | 568 | 4 | 0.00520 | 0.00003 | 779.709311 |
| which | 88 | 860 | 0.00081 | 0.00743 | 687.134838 |
| Buddhas | 466 | 0 | 0.00427 | 0.00000 | 662.712284 |
| I | 755 | 1988 | 0.00691 | 0.01718 | 511.937387 |
| a | 1431 | 2950 | 0.01310 | 0.02550 | 462.810632 |
| sutra | 325 | 0 | 0.00298 | 0.00000 | 459.207565 |
| One | 511 | 55 | 0.00468 | 0.00048 | 451.427785 |

The numbers in the lotus.txt count and walden.txt count columns are the number of times that word appeared in that text file. The freq columns are the frequencies of that word in the respective files. For example, the word *Buddha* accounts for nearly 1% of the words in the Lotus Sutra.

Internally, the program will use POSIX threads to parallelize the process.

**Considerations**

Doing so effectively and with high performance will require you to address (at least) the following issues:

* **How to parallelize the task.** Of course, the central challenge of this project is to parallelize the process. Think about what can be done in parallel, and what must be done serially by a single thread, and design your parallel program as appropriate.

One interesting issue that the "best" implementations will handle is this: what happens if one thread runs more slowly than another? Does your algorithm give more work to faster threads?

* **How to determine how many threads to create.** On Linux, this means using interfaces like get\_nprocs() and get\_nprocs\_conf(); read the man pages for more details. Then, create threads to match the number of CPU resources available.
* **How to efficiently perform each piece of work.** While parallelization will yield speed up, each thread's efficiency in performing its work is also of critical importance. Thus, making the core loop as CPU efficient as possible is needed for high performance.
* **How to access the input file efficiently.** On Linux, there are many ways to read from a file, including C standard library calls like fread() and raw system calls like read(). One particularly efficient way is to use memory-mapped files, available via mmap(). By mapping the input file into the address space, you can then access bytes of the input file via pointers and do so quite efficiently.

**Grading**

Your code should compile (and should be compiled) with the following flags: -Wall -Werror -pthread -O. The last one is important: it turns on the optimizer! In fact, for fun, try timing your code with and without -O and marvel at the difference. (Note: This code will be tested in an Ubuntu 18.04 LTS box)

Your code will first be measured for correctness, ensuring that it analyzes input files correctly.

If you pass the correctness tests, your code will be tested for performance; higher performance will lead to better scores.

**Hand In Instructions**

* Make sure you have included your name in a README file. Also in that file outline your general approach. Create a tar file handin.tar that contains all of your source files, and the README. Attach the tar file to an email sent to the gmail address submit.o.bot with subject line: cpsc405 log likelihood.

**Good luck**