Part 1: Problem Introduction

For the CSC718 Project, I intend to create one sequential and two parallel programs that will parse a text file and query the data to return the number of occurrences for a specified search term. These programs will focus on a singular file of any size, which will then be parsed and analyzed. For the parallel programs, this parsing will include distributing blocks of the original text file to n temp files and mapping those files to n threads. I intend to create this program using C, with the possibility of some C++ libraries for natural language processing. Each parallel program will utilize either semaphores or Open MP.

The intent for these programs is to streamline the process of searching large files within the command line, as an intuitive GUI is not always available. Not only will they provide a simple solution for finding text patterns, but the parallel programs will also (theoretically) function at a faster rate than their terminal text-editing counterpart such as vim, vi, nano, etc.

Part 2: Solution Implementation

I have successfully created three programs, one sequential and two parallel, which parse a file to count occurrences of a specified search term. Each program is provided in the same final project folder as this report and contains a unique `README` and `MakeFile`. There is also `testFile.txt` which is a randomly generated AI text file; the information within it is nonfactual and mostly nonsensical. While each folder contains a separate copy of `testFilet.txt`, these files are identical and used as the default parsing file.

While the `README's` provide an in-depth overview of compilation and execution instructions, each program can be compiled using the provided `MakeFile`:

`make`	to make executable
`make clean`	to remove executable

Furthermore, each program contains specific and optional arguments which are also explained in their corresponding `README` files or accessed using the `-h` flag:

```
./proj_XXX -h
```

However, the common arguments between all three programs are as follows:

```
./proj_XXX -h -c -v -f <file> -s <string>
```

COMMAND	PURPOSE	DEFAULT	EXAMPLE
`-h`	Display Usage summary		`./proj_XXX -h`
`-c`	Set Case Sensitivity	`False`	`./proj_XXX -c`
`-v`	Set Verbose Mode	`False`	`./proj_XXX -v`
`-f <file>`	Specify File	`testFile.txt`	`./proj_XXX -f myFile.txt`
`-s <string>`	Specify Search Term	`life`	`./proj_XXX -s search_term`

It is important to note that while each program accepts arguments, they are optional; if no argument is given, the previously listed defaults will be used. To change the program defaults, the code can be modified under `Constants` then `Default Settings`. There are also additional arguments specific to the parallel programs which are not listed here. Lastly, if the user would like to test out the programs with other search words, without the labor of finding another file, some other common words in `testFile.txt` are `Earth` (tip: try both with and without case sensitivity on), `science`, and `according`.

One of the reasons I chose to parallelize file parsing is because it can be computationally expensive, especially on a large file. To find matches for a specified search term, the file must iterate the file, word-by-word, and then compare each word to the predefined search term. Therefore, when there are enough words, this can take some time. I initially believed splitting this computation between processes and/or threads would drastically speed up the process. However, I forgot a key detail: before the file data can be split, it must be parsed and mapped. Therefore, the two parallel programs end up parsing the file twice, thus negating any time saved. For example, the sequential program follows these basic steps:

1. Open File
2. Parse File
   a. Compare Search Terms
   b. Count Matches
3. Display Final Data

File Parse Time

As shown above in blue, the file is only read one time. Alternatively, the parallel programs follow these basic steps:

1. Open File
2. Parse File
   a. Split File
   b. Map Processes/Threads to a Data Block
   ---- BEGIN PARALLELIZATION ----
3. Parse File Block
   a. Compare Search Terms
   b. Count Matches
   ---- END OF PARALLEL REGION ----
4. Display Final Data

File Parse Time

While these are rudimentary overviews of the steps in each program, they illustrate how a parallel program must parse through a file twice, and a sequential program once. However, since the second iteration is done in parallel, it completes much faster than the first. To further illustrate this, each program displays the total program runtime and the file parse time upon completion. The program runtime begins at the start of the program and ends after all tasks have been completed, while the file parse time only calculates the time it takes to compare search terms and count matches (as shown above in green). A benchmark of the program, including program runtime and the file parse time is shown below.

	TOTAL PROGRAM	% CHANGE	PARSE ONLY	% CHANGE	*threads*
`proj_seq`	0.0251		0.0259		
`proj_sem`	0.0357	1.4223	0.0122	0.4710	*5*
	0.0391	1.5578	0.0139	0.5367	*9*
`proj_omp`	0.0732	2.9163	0.0113	0.4363	*5*
	0.0700	2.7888	0.0203	0.7838	*9*

As expected, the parallel versions of the solution increase the total computation time of the program, due to the extra file iteration. In fact, it increases up to 291% for the five-thread version of

OMP. However, when calculating only the time it takes to analyze and compare the text, the parallel programs significantly outperform the sequential program – the fastest completing in less than half of the time at 43% for the five-thread version of OMP. I believe this distinction is important because, while these programs do not provide a direct solution to the problem introduced here, they would be ideal for similar, pre-blocked data. For example, when there are multiple files to read in and each thread can be assigned a separate file.

One of the most important things I learned in this project is that just because parallelization seems best in theory, this is not always true. Sometimes the overhead to implementing a parallel program negates any time saved – such is the case here. I also gained a better understanding of OMP during this process. I struggle with parallelization in general due to the limited ability to debug. I learn how a program works by stepping through it and using watchers. Similarly, when I execute a sequential-to-parallel transformation, the implementation of each step in the process helps me understand the methods used. OMP handles many of these steps automatically, so I tend to struggle with what is implied and what I still need to implement. My insecurity regarding OMP is the reason I chose it for this project – I wanted to try it with a program I completely understood.