Peri Hassanzadeh
ECE1570 High Performance Computing
Dr. Bigrigg
November 5, 2021

Threads and Parallel Programming

Introduction

In this assignment, I was able to use threads and develop data and task parallel algorithms. Utilizing pthreads and ideas such as busy-wait and mutexes, I was able to develop a better understanding of programming in parallel and how this could be used in other applications as well. In addition, I identified the speedup I could achieve by using multiple processors on the bigrigg.com server.

The main objective of each program no matter it's algorithm was to read and perform actions on certain word documents. I was to read in the data from the text files, split the characters and filter out noise words as well as whitespace, symbols, and numbers. After performing certain actions on the read in text, it was sent to another document in which the frequency of each word was found and then in a final report, the top 10% of most frequent words were listed in alphabetical order.

In order to run each file, make sure that all .cpp files are in the same folder as the working directory. A folder named "certdata" should also be in the working directory with all of the text files within it. This format will allow for successful compilation of each implementation resulting in an executable file with the following commands:

Serial command:

```
g++ serial.cpp -o serial
```

Data Parallel command:

```
Windows: g++ dataParallel.cpp -o dataParallel -lpthread
Linux: g++ dataParallel.cpp -o dataParallel -pthread
```

Task Parallel command:

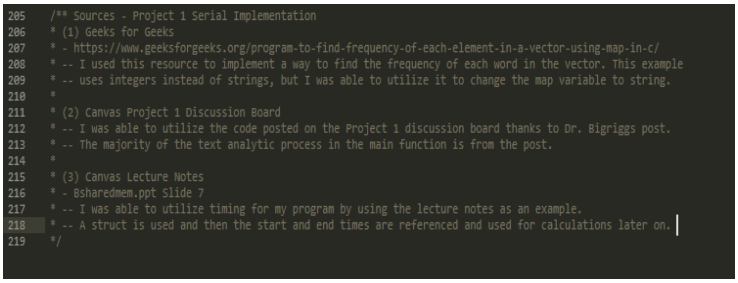
```
Windows: g++ taskParallel.cpp -o taskParallel -lpthread
Linux: g++ taskParallel.cpp -o taskParallel -pthread
```

Finally, note that each implementation could be optimized in various ways. I will go over a few points that I wish I would have been able to develop on given better time management.

Serial Version

The files that correspond with the serial version are as follows:
serial.cpp - implementation of serial version
serial.exe - compiled program file
results.txt - file with all of the filtered words listed from every text file
finalreport.txt - summary file with top 10% of most used words listed

Screenshots of Code:

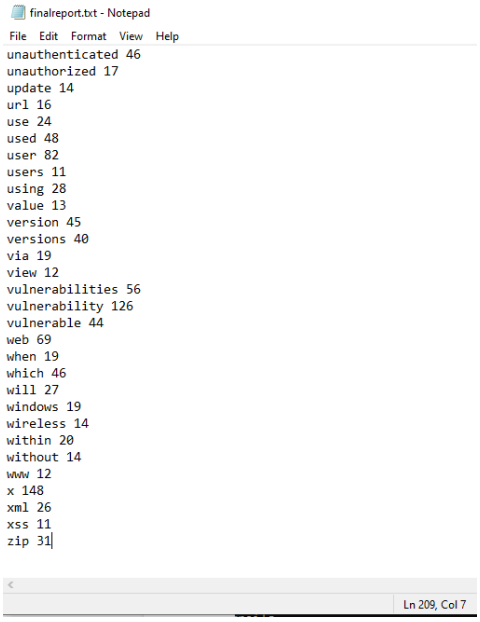


The screenshots on this page are for the serial implementation of the project. The code may also be viewed in the submitted files as serial.cpp. Zoom in for a better view of the code or open the specified file.

Output:

From my local machine:

```
C:\Users\perih\Downloads\ECE1570_Project1\ECE1570_Project1>serial
Total time for serial execution: 269864 microseconds
```



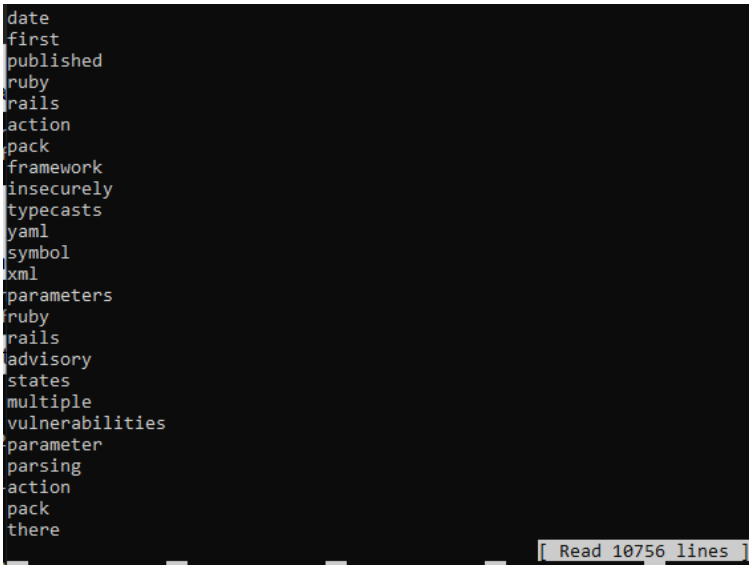
The execution of the program took 269864 microseconds on my local machine. In the text file, it specifies there are 209 lines meaning that the top 10% of most used words summary file listed 209 different words. The results should be the same for the rest of the files if they were successful.

From Bigrigg Server:
Execution time

```
[northampton]$ ./serial
Total time for serial execution: 123376 microseconds
```

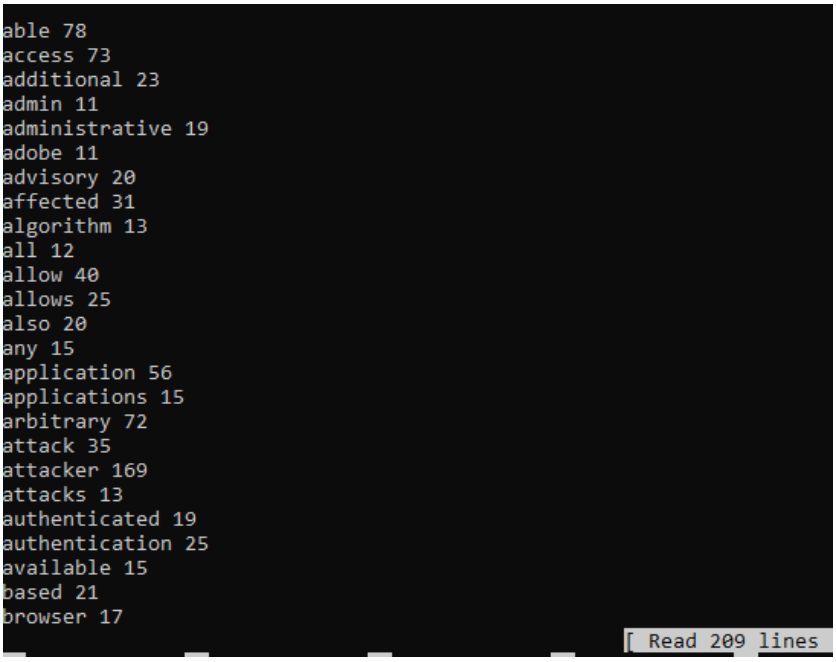
From the ssh server bigrigg.com, the execution took 123376 microseconds which is significantly less than that of my local machine. Additionally, I will make sure to analyze the rest of the execution times for data parallel and task parallel from the server as well as my local machine. Comparing the two values will allow me to properly decide proper scaling and different implementation benefits and drawbacks.

results.txt



The results.txt file holds all of the filtered words before the frequency is counted and analyzed by the top 10% of most frequently used words. Currently there are 10756 total words as listed by the number of lines since there is one word per line.

finalresutls.txt

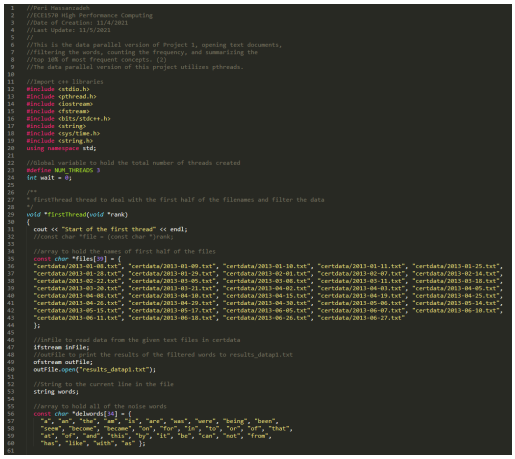


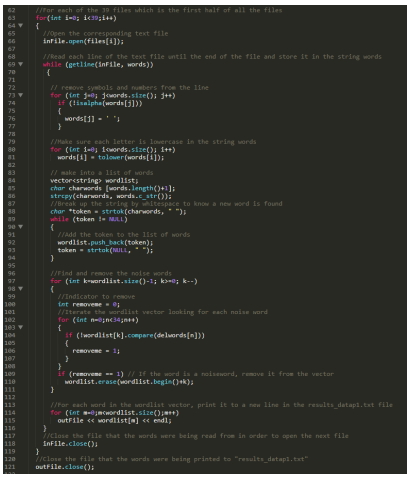
The finalresults.txt file line totals matches up with that on my local machine as well so I can confidently say there should be 209 lines for the following versions of the program as well.

Data Parallel Version

The files that correspond with the data parallel version are as follows:
dataParallel.cpp - implementation of data parallel version
dataParallel.exe - compiled program file
results_datap1.txt - file with all of the filtered words listed from first half of text files
results_datap2.txt - file with all of the filtered words listed from second half of text files
finalreport_datap.txt - summary file with top 10% of most used words listed

Screenshots of Code:





```
/** Sources - Project 1 Data Parallel Implementation
* (1) Geeks for Geeks
* - https://www.geeksforgeeks.org/program-to-find-frequency-of-each-element-in-a-vector-using-map-in-c/
* -- I used this resource to implement a way to find the frequency of each word in the vector. This example
* -- uses integers instead of strings, but I was able to utilize it to change the map variable to string.
*
* (2) Canvas Project 1 Discussion Board
* -- I was able to utilize the code posted on the Project 1 discussion board thanks to Dr. Bigriggs post.
* -- The majority of the text analytic process in the main function is from the post.
*
* (3) Canvas Lecture Notes
* - Bsharedmem.ppt Slide 7
* -- I was able to utilize timing for my program by using the lecture notes as an example.
* -- A struct is used and then the start and end times are referenced and used for calculations later on.
*
* (4) TutorialsPoint
* - https://www.tutorialspoint.com/cplusplus/cpp_multithreading.htm
* -- Used as a reference for the syntax of threads and how to create/join
*/
```

The screenshots on the page above are for the data parallel implementation of the project. The code may also be viewed in the submitted files as dataParallel.cpp. Zoom in for a better view of the code or open the specified file.

Output:

From my local machine:

```
C:\Users\perih\Downloads\ECE1570_Project1\ECE1570_Project1>dataParallel.exe
Creating the first thread that calls firstThread
Creating the second thread that calls secondThread
Start of the first thread
Start of second thread
End of first thread
End of second thread
Creating a third thread to merge results
Start of merge thread
End of merge thread
Process is complete
Total time for data parallel execution: 179443 microseconds
```

finalreport_datap.txt - Notepad

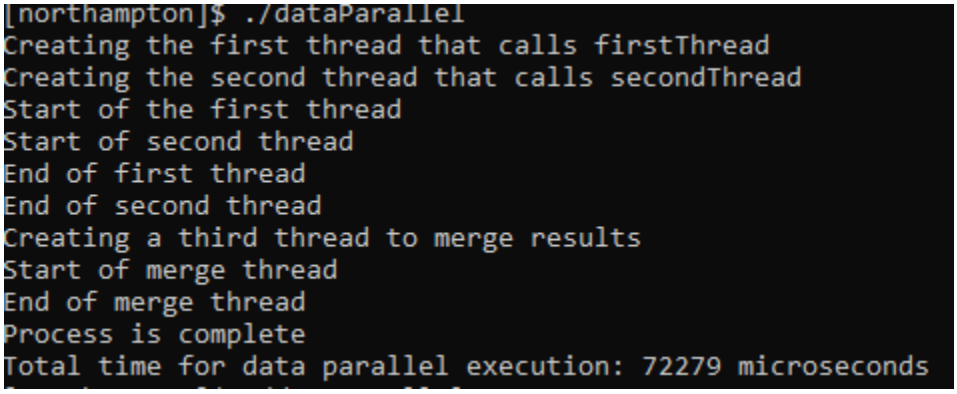
File Edit Format View Help

```
sql 29
stack 20
standard 11
states 13
system 65
systems 13
tf 12
these 26
traversal 15
unauthenticated 46
unauthorized 17
update 14
url 16
use 24
used 48
user 82
users 11
using 28
value 13
version 45
versions 40
via 19
view 12
vulnerabilities 56
vulnerability 126
vulnerable 44
web 69
when 19
which 46
will 27
windows 19
```

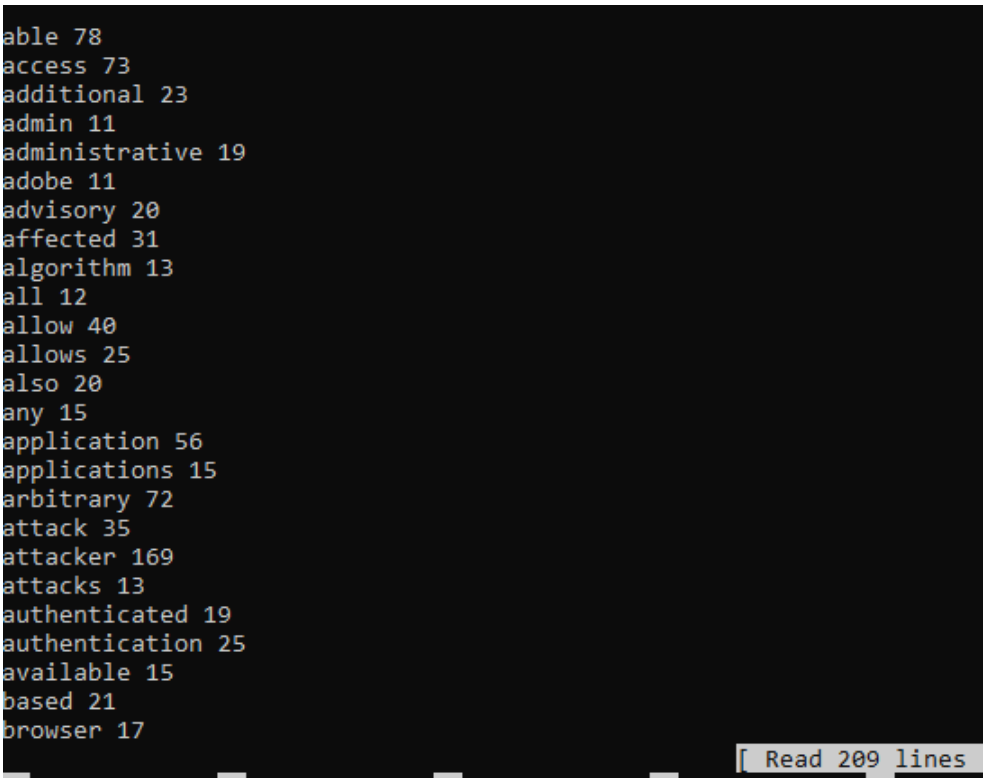
Ln 201, Col 11

The execution of the program took 179443 microseconds on my local machine. In the text file, it specifies there are 201 lines meaning that the top 10% of most used words summary file listed 201 different words. The results are different from the serial version meaning there is some type of error related when introducing threading.

Bigrigg Server:



From the ssh server bigrigg.com, the execution took 72279 microseconds which is also significantly less than that of my local machine similar to what happened in the serial implementation.

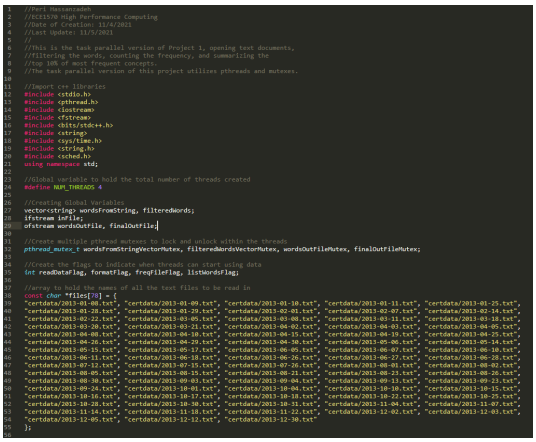


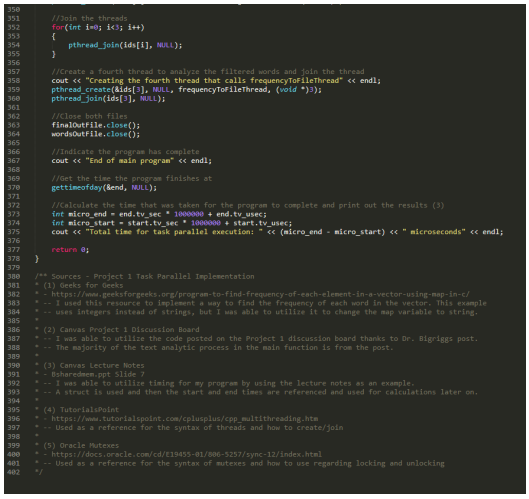
The finalresult_datap.txt file line totals matches up with that on my local machine serial implementation. I am still unsure as to why the results are different when checking from the server and from my local machine. If I had to say, I would think it would be due to the fact of having multiple cores on the server as well as it being a shared machine.

Task Parallel Version

The files that correspond with the task parallel version are as follows:
taskParallel.cpp - implementation of task parallel version
taskParallel.exe - compiled program file
results_taskP.txt - file with all of the filtered words listed from every text file
finalreport_taskP.txt - summary file with top 10% of most used words listed

Screenshots of Code:





```
    //Join the threads
    for(int i=0; i<3; i++)
    {
        pthread_join(ids[i], NULL);
    }

    //Create a fourth thread to analyze the filtered words and join the thread
    cout << "Creating the fourth thread that calls frequencyToFileThread" << endl;
    pthread_create(&ids[3], NULL, frequencyToFileThread, (void *)3);
    pthread_join(ids[3], NULL);

    //Close both files
    finalOutFile.close();
    wordsOutFile.close();

    //Indicate the program has complete
    cout << "End of main program" << endl;

    //Get the time the program finishes at
    gettimeofday(&end, NULL);

    //Calculate the time that was taken for the program to complete and print out the results (3)
    int micro_end = end.tv_sec * 1000000 + end.tv_usec;
    int micro_start = start.tv_sec * 1000000 + start.tv_usec;
    cout << "Total time for task parallel execution: " << (micro_end - micro_start) << " microseconds" << endl;

    return 0;
}

/** Sources - Project 1 Task Parallel Implementation
 * (1) Geeks for Geeks
 * - https://www.geeksforgeeks.org/program-to-find-frequency-of-each-element-in-a-vector-using-map-in-c/
 * -- I used this resource to implement a way to find the frequency of each word in the vector. This example
 * -- uses integers instead of strings, but I was able to utilize it to change the map variable to string.
 *
 * (2) Canvas Project 1 Discussion Board
 * -- I was able to utilize the code posted on the Project 1 discussion board thanks to Dr. Bigriggs post.
 * -- The majority of the text analytic process in the main function is from the post.
 *
 * (3) Canvas Lecture Notes
 * - 8sharedmem.ppt Slide 7
 * -- I was able to utilize timing for my program by using the lecture notes as an example.
 * -- A struct is used and then the start and end times are referenced and used for calculations later on.
 *
 * (4) TutorialsPoint
 * - https://www.tutorialspoint.com/cplusplus/cpp_multithreading.htm
 * -- Used as a reference for the syntax of threads and how to create/join
 *
 * (5) Oracle Mutexes
 * - https://docs.oracle.com/cd/E19455-01/806-5257/sync-12/index.html
 * -- Used as a reference for the syntax of mutexes and how to use regarding locking and unlocking
 */
```

The screenshots on the page above are for the task parallel implementation of the project. The code may also be viewed in the submitted files as taskParallel.cpp. Zoom in for a better view of the code or open the specified file.

Output:

From my local machine:

```
C:\Users\perih\Downloads\ECE1570_Project1\ECE1570_Project1>taskParallel.exe
Start of main program
Creating the first thread that calls readInDataThread
Creating the second thread that calls formatStringThread
Start of readInDataThread
Locking wordsFromStringVectorMutex
Creating the third thread that calls listWordsStringtoVectThread
Joining all threads
Unlocking wordsFromStringVectorMutex
End of readInDataThread1
Start of formatStringThread
Locking wordsFromStringVectorMutex and filteredWordsVectorMutex
Unlocking wordsFromStringVectorMutex and filteredWordsVectorMutex
End of formatStringThread
Start of listWordsStringtoVectThread
Locking filteredWordsVectorMutex and wordsOutFileMutex
Unlocking filteredWordsVectorMutex and wordsOutFileMutex
End of listWordsStringtoVectThread
Creating the fourth thread that calls frequencyToFileThread
Start of frequencyToFileThread
Locking wordsOutFileMutex and finalOutFileMutex
Unlocking wordsOutFileMutex and finalOutFileMutex
End of frequencyToFileThread
End of main program
Total time for task parallel execution: 888178 microseconds
```

The execution of the program took 888178 microseconds on my local machine. In the text file, it specifies there are 209 lines. The results are the same as the serial version meaning there is consistency between the two implementations.

finalReport_taskP.txt - Notepad

File Edit Format View Help

```
unauthenticated 46
unauthorized 17
update 14
url 16
use 24
used 48
user 82
users 11
using 28
value 13
version 45
versions 40
via 19
view 12
vulnerabilities 56
vulnerability 126
vulnerable 44
web 69
when 19
which 46
will 27
windows 19
wireless 14
within 20
without 14
www 12
x 148
xml 26
xss 11
zip 31
```

Ln 209, Col 7

Bigrigg Server:

```
[northampton]$ ./taskParallel
Start of main program
Creating the first thread that calls readInDataThread
Creating the second thread that calls formatStringThread
Start of readInDataThread
Locking wordsFromStringVectorMutex
Creating the third thread that calls listWordsStringtoVectThread
Joining all threads
Unlocking wordsFromStringVectorMutexStart of formatStringThread
End of readInDataThread
Locking wordsFromStringVectorMutex and filteredWordsVectorMutex
1
Unlocking wordsFromStringVectorMutex and filteredWordsVectorMutexStart of listWordsStringtoVectThread
End of formatStringThread

Locking filteredWordsVectorMutex and wordsOutFileMutex
Unlocking filteredWordsVectorMutex and wordsOutFileMutex
End of listWordsStringtoVectThread
Creating the fourth thread that calls frequencyToFileThread
Start of frequencyToFileThread
Locking wordsOutFileMutex and finalOutFileMutex
Unlocking wordsOutFileMutex and finalOutFileMutex
End of frequencyToFileThread
End of main program
Total time for task parallel execution: 649249 microseconds
```

From the ssh server bigrigg.com, the execution took 649240 microseconds which is less than that of my local machine similar to what happened in the serial implementation, but the difference is not as drastic.

```
able 78
access 73
additional 23
admin 11
administrative 19
adobe 11
advisory 20
affected 31
algorithm 13
all 12
allow 40
allows 25
also 20
any 15
application 56
applications 15
arbitrary 72
attack 35
attacker 169
attacks 13
authenticated 19
authentication 25
available 15
based 21
browser 17
                                                    [ Read 209 lines ]
```

The finalresult_taskp.txt file line totals matches up with that on my local machine serial implementation which is 209 lines. The results from my local machine and from the bigrigg server are the same amount of lines, just differing execution times.

Optimizations

There are multiple optimizations that could be made to each implementation, but I would like to address some of the more obvious optimizations that would make a significant difference in the implementation and possibly efficiency of the program.

Starting off specifically with the serial implementation, the data files could be read in from the directory using a command instead of being hardcoded into an array. This could also be similar when regarding the noise words, they could be stored in a text file, read in, and then eliminated from the main data if found. This would be better for cleaner code and less lines as well, possibly less need for for loops to iterate through the array rather than just finding the files from the directory itself and directly accessing the data. The program would be able to recursively search through the files and this implementation could be applied to each part of the project.

Another large optimization that could be made is to create more threads in the task parallel version to be more of a streamlined process being able to break up the tasks even more. Some of my threads have long processes that would take awhile or need to wait for a previous thread to finish where as if they were broken up into smaller parts there would be less waiting time and more tasks would be able to run at the same time.

Finally, the last optimization I wanted to mention was to be able to pass data into a thread from the main thread or main method. I had the code set up, but to speed up my understanding of the parallel programming portion of the project, I decided to hardcode some information in the threads or make the variables global. In the future I would plan to pass these variables into the thread so as not to cause dangerous access to such global variables.