Danny Lopez and Ryan Lanphear

Group 8

ECEL 353 Project 3 Report

Professor Michael Lui

8/7/2017

**Mini-grep Multi-Threaded Program**

For Assignment 3, the class was tasked with developing a C program that would act as a UNIX grep, searching for a provided string within files in a specified directory and its sub-directories beneath it. This would be accomplished through a single-threaded serial search, then with a multi-threaded static or dynamic search for comparison. The user specifies the number of threads to use, as well as whether to search statically or dynamically. Choosing static will split the threads to search an even number of files per thread, but what this runs into is having different file sizes. Choosing dynamic will split the files more evenly, allocating space to the threads as time goes on.

To compile mini\_grep.c, here is the bash syntax:

$ gcc -o mini\_grep mini\_grep.c queue\_utils.c -std=c99 -lpthread -Wall

To run the compiled file, use bash syntax similar to the specifications and example below.

To run:

./mini\_grep <string> /<path>/<to/<search>/ <num\_threads> <static/dynamic>

For example:

$ ./mini\_grep lui /home/DREXEL/mdl45/ 4 static

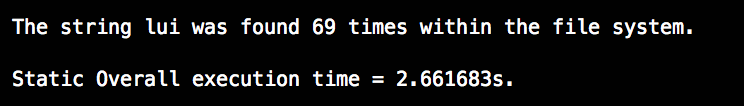
$ ./mini\_grep lui /home/DREXEL/mdl45/ 4 dynamic

The first example run above will begin searching /home/DREXEL/mdl45/ for string “lui” as a single-threaded task and returns the files that contain the string and the number of times that the string is found, along with the time elapsed for the task. Next, the string will be searched for in the same directory again as a multi-threaded statically or dynamically load balanced task depending on the user input for method and threads used. For both examples above, 4 threads are allocated for multi-threading.

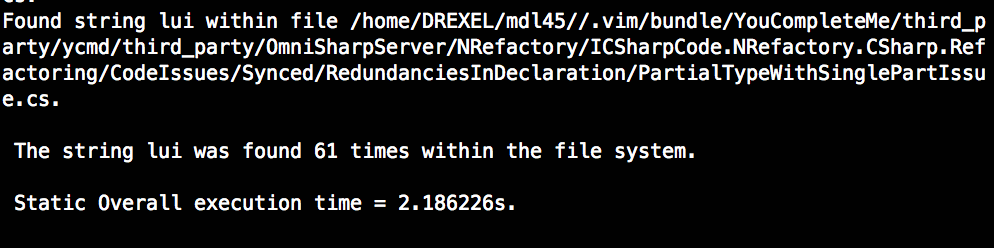
Running the program as a multi-threaded statically load balanced task allocates memory for the queue and adds the path to the queue and recursively searches through the path in the queue, searching for files and subdirectories to add subdirectories to the queue to continue recursively searching through them, and ignoring symbolic links, “.”, and “..” files. This retrieves all the files and increments a counter for the number of files to divide up the work that the threads will be responsible for. With the files and the work obtained, the threads themselves and the division of work is coded. The work is divided up and sent to the queue, then the for loop recursively works through each queue task and assigns the specified number of threads to handle the work. The threads take the next task available from the queue, allocates memory needed for its process, and works through it to search for the string in a portion of the gathered files by opening them and searching through the file until there’s nothing left to iterate through. If the string is found, then the line that the string is found is printed and an integer is incremented that will be returned as the counter for number of times the specified string is found. Any mutex locks used are locked to prevent an overlap of work. Once all of the files are iterated over, the information gathered over the separate threads are joined together and the mutex locks are destroyed. The counter for the number of times that the string was found is returned and the execution time is printed to the terminal for comparison against the serial search.

When ran as a multi-threaded dynamically load balanced task, the method begins in a similar manner, creating variables and allocating memory, then populating the queue by gathering the files as before, but without incrementing the number of files as it does not matter, since all files should finish at approximately the same time through the use of locking and unlocking mutexes. The queue is initialized, the threads are created, and the thread function defines how the work is to be handled. The while loop runs for as long as there are files in the queue, assigning a thread to handle the next available task in the queue. The threads will then increment a global variable the number of times that the string was found and return that global to be printed. The execution time for the dynamic function is also printed to the terminal for comparison to the serial search.

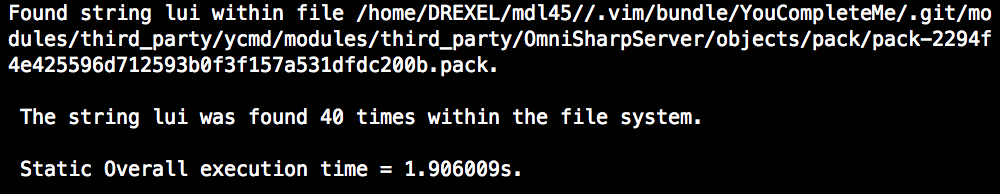
In all instances of using multithreading, there was a significant decrease of execution time observed across both methods. The terminal outputs below display the reduction in speed for the number of threads used in the static load balanced method.



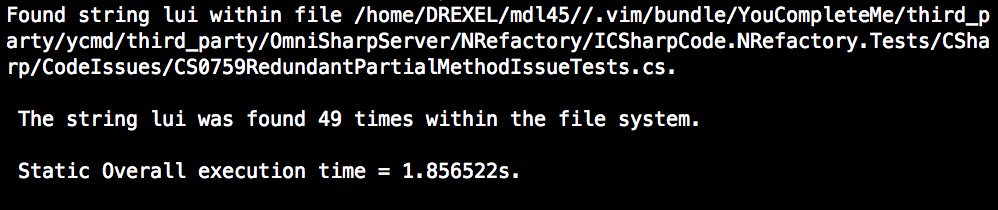
**Figure 1.** The number of times the string “lui” was found, searched statically with two threads.



**Figure 2.** The number of times the string “lui” was found, searched statically with four threads.

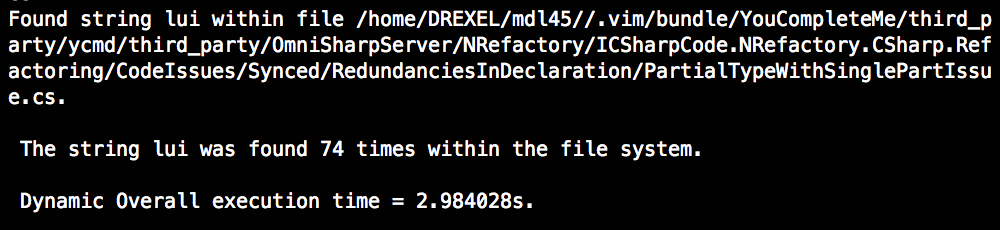


**Figure 3.** The number of times the string “lui” was found, searched statically with eight threads.

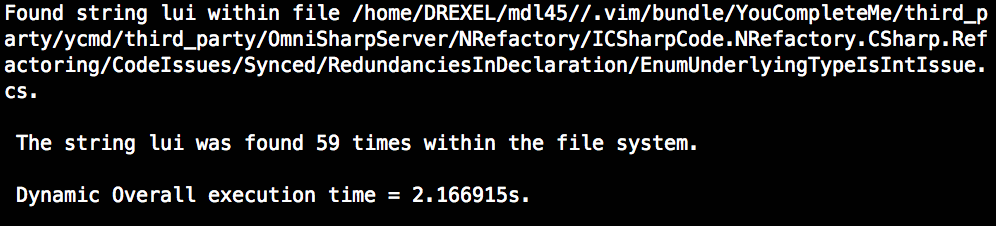


**Figure 4.** The number of times the string “lui” was found, searched statically with sixteen threads.

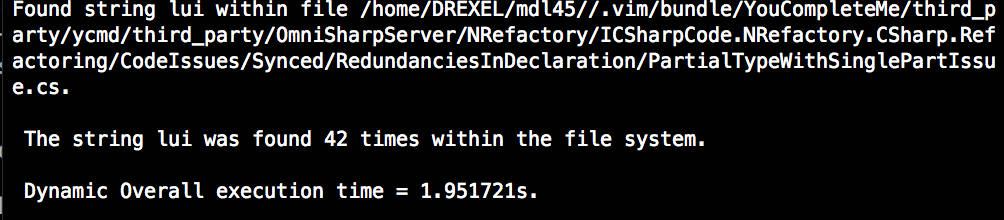
The dynamic load balancing results are as displayed below.



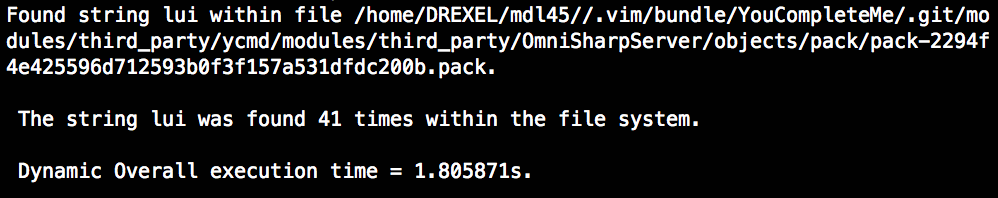
**Figure 5.** The number of times the string “lui” was found, searched dynamically with two threads.

****

**Figure 6.** The number of times the string “lui” was found, searched dynamically with four threads.



**Figure 7.** The number of times the string “lui” was found, searched dynamically with eight threads.



**Figure 8.** The number of times the string “lui” was found, searched dynamically with sixteen threads.

Note that despite the decrease in execution time, the number of times that the string was found would often decrease as more cores were added across both methods. Table 1 below displays the time speedup in elapsed time reduction in an organized fashion.

|  |  |  |
| --- | --- | --- |
| **Cores used** | **Static time elapsed (seconds)** | **Dynamic time elapsed (seconds)** |
| 1 | 8.52 (serial search) | 8.52 (serial search) |
| 2 | 2.661 | 2.984 |
| 4 | 2.186 | 2.167 |
| 8 | 1.906 | 1.951 |
| 16 | 1.857 | 1.806 |

**Table 1.** The reduction in time elapsed from the terminal organized within a table.

As seen from the table, dynamic search is not entirely beneficial at first for lower numbers of allocated threads for the task. As more threads are allocated, however, the dynamic load balancing search’s execution time decreases at a faster rate than the static load balancing. As for why the grep search finds the string less frequently as more threads are allocated, this could be due to the mutex locks overlapping for an increasing number of threads. Further experimentation of lock implementations would be needed to attempt to rectify this.