# Project 2 Report

CS 4500

Daniel Frey, Justin Hale
3-27-2019

1. **Project Members**: Daniel Frey and Justin Hale.

   **We have neither given nor received unauthorized assistance on this work.**

2. **VM Name:** dfrey-CS4500
   instructor account password: zhuang
   *Files for Project 2 are in /home/instructor/Project\ 2
   **Each specific Task is in Project 2 directory

3. **Description of how solved and lessons learned**

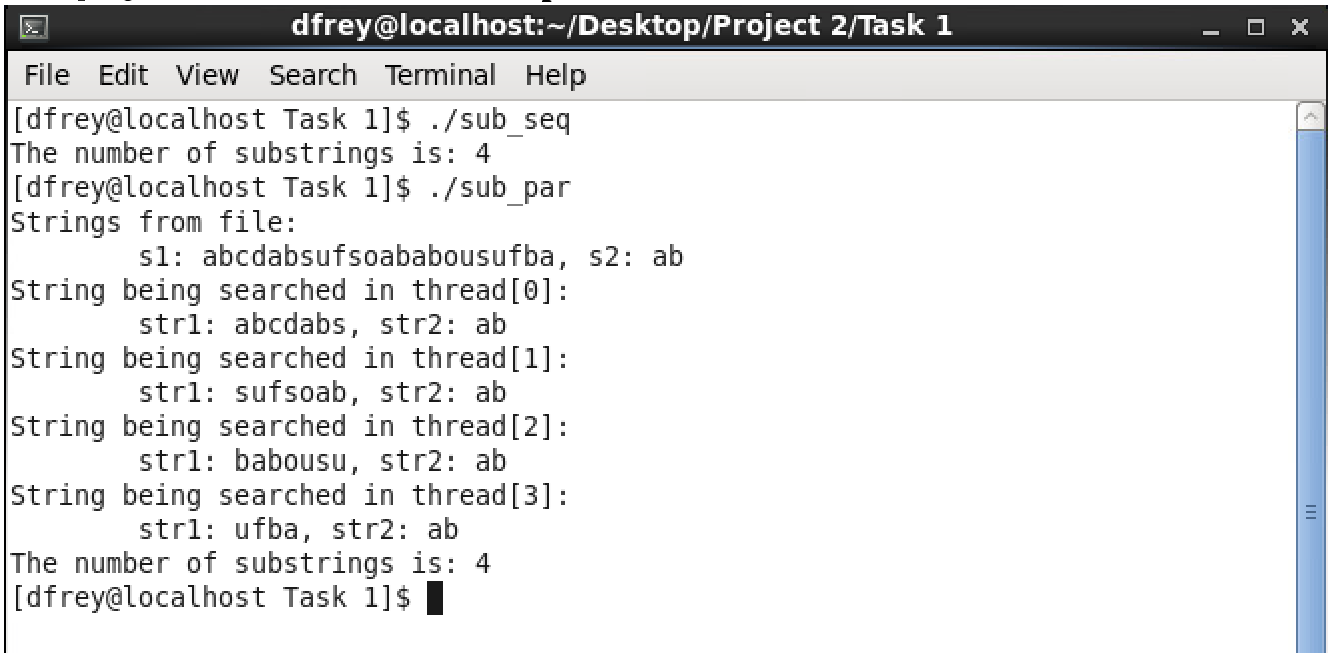
**3.1. Task 1 /home/instructor/Project\ 2/Task\ 1**

To solve this problem we began by dividing the input string, s1, among the arbitrary number of threads. Each thread has a struct of data for the string it is searching, str1, and the string it is trying to find, s2. Each thread then searches its partitioned string for substrings of s2.

To try and address the issue of s2 appearing at the partition of s1, we allowed a specific number of chars to be repeated in the partitioned search strings, str1. The number of chars is dependent on s2. We set the repeated chars to one less than the size of s2. This way it seemed that it would not find extra matches when repeating characters.

From this Task we were able to better understand using threads to implement a divide and conquer approach to a problem. The issues faced in this task were how to divide s1 and make sure substrings were not missed at the partition locations.

*The program to run for this task is **sub_par**


dfrey@localhost:~/Desktop/Project 2/Task 1
File Edit View Search Terminal Help

```
[dfrey@localhost Task 1]$ ./sub_seq
The number of substrings is: 4
[dfrey@localhost Task 1]$ ./sub_par
Strings from file:
        s1: abcdabsufsoababousufba, s2: ab
String being searched in thread[0]:
        str1: abcdabs, str2: ab
String being searched in thread[1]:
        str1: sufsoab, str2: ab
String being searched in thread[2]:
        str1: babousu, str2: ab
String being searched in thread[3]:
        str1: ufba, str2: ab
The number of substrings is: 4
[dfrey@localhost Task 1]$
```

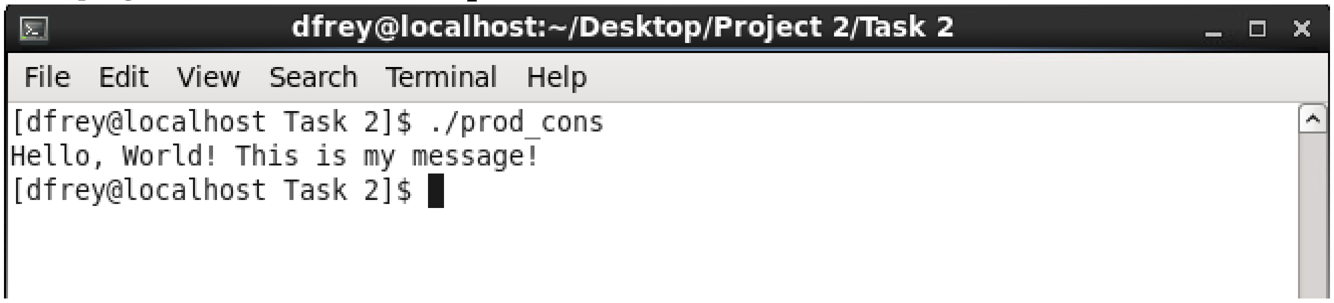
**3.2. Task 2 /home/instructor/Project\ 2/Task\ 2**

The producer uses a condition variable to wait if the buffer is full. Once the buffer is deemed not full, it will copy over one char from the string that was stored in message.txt to the buffer. The producer then signals the consumer that there are items to be consumed in the buffer. If the producer is using the buffer queue, it sets a lock on it so that it cannot be changed by the consumer.

The consumer will run so long as the characters from the string in message.txt are not yet exhausted. The consumer begins by waiting until there is an item in the buffer to consume. Once there are items ready, the consumer will display the characters that are in the buffer. After that, consumer then signals the producer that it should produce more items to consume. Just like the producer, the consumer will place a lock on the buffer so that the producer cannot edit it while it is being consumed.

Solving this problem seemed to go easier than the first task. This task helped conceptualize the issue with waiting and signaling so that different parts did not interfere with the other.

*The program to run for this task is **prod_cons**


dfrey@localhost:~/Desktop/Project 2/Task 2
File Edit View Search Terminal Help

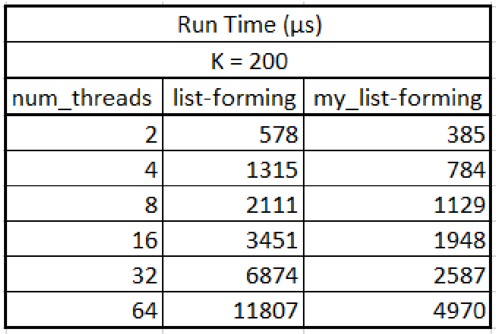
```
[dfrey@localhost Task 2]$ ./prod_cons
Hello, World! This is my message!
[dfrey@localhost Task 2]$ 
```

### 3.3. Task 3 /home/instructor/Project\ 2/Task\ 3

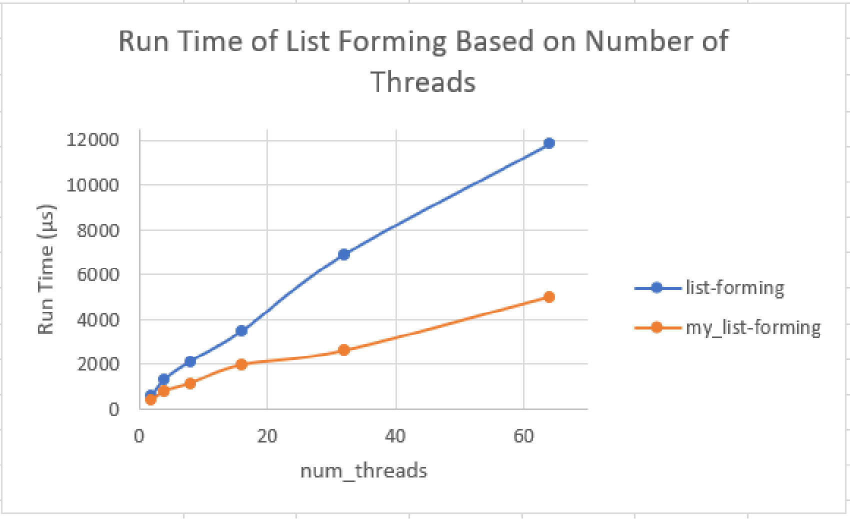
To solve this problem it seemed like it would be faster to create a local list of nodes and then add all at once. This way mutex lock and unlock are only being called once per thread instead of K times per thread. The producer thread will create a local list of K nodes. Once these nodes are all created and stored in a list, a mutex is used when transferring the local list to the global list.

The performance results from this change were easily noticeable. Instead of changing the amount of K, as this would require recompiling each time, we changed the number of threads to gather rest results. We tested each program with 2-64 threads. Sometimes the programs ran quickly or slowly, so we ran the them several times to obtain a more accurate run time. The modified program, my_list-forming, ran quicker than the original list-forming program for each number of threads, thus proving the changes were a successful improvement.

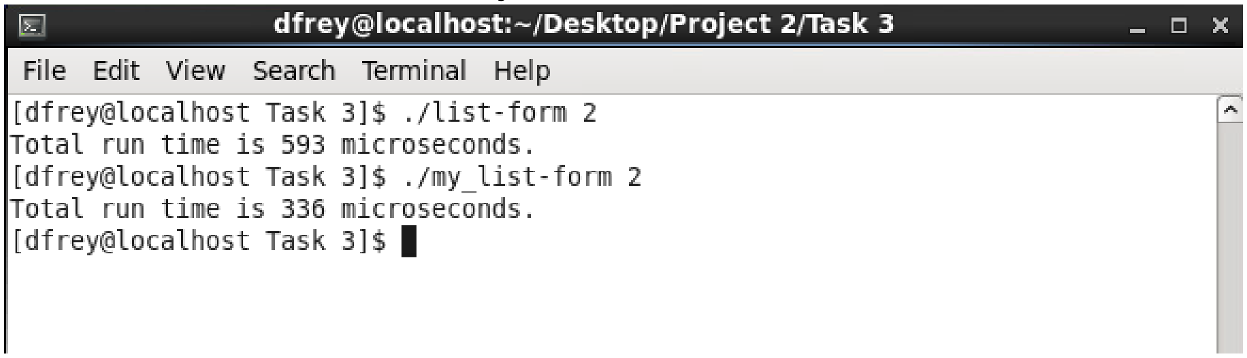
This allowed us to learn how to code programs to run more efficiently.



| Run Time (µs) | | |
|---|---|---|
| K = 200 | | |
| num_threads | list-forming | my_list-forming |
| 2 | 578 | 385 |
| 4 | 1315 | 784 |
| 8 | 2111 | 1129 |
| 16 | 3451 | 1948 |
| 32 | 6874 | 2587 |
| 64 | 11807 | 4970 |



*The program to run for this task is **my_list-form**



dfrey@localhost:~/Desktop/Project 2/Task 3
File Edit View Search Terminal Help

```
[dfrey@localhost Task 3]$ ./list-form 2
Total run time is 593 microseconds.
[dfrey@localhost Task 3]$ ./my_list-form 2
Total run time is 336 microseconds.
[dfrey@localhost Task 3]$ 
```

***Note: Source codes and test files can be found in the attached Project 2 folder.**