POSIX Thread Programming

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

**Team name: BestGroup**

Name of the VM: VirtualMachine (in mgibbon2/ & username: BestGroup)

Password: HaveAGreatDay24

Team members names:

Matthew Gibbons | Part 1 Pseudocode + Implementation, Part 3 Secondary Refinement + Data Gathering

Noah Storms | Part 2 Pseudocode + Implementation, Part 3 Primary Refinement + Data Analysis

# Description of Project: 2 – 3 Sentences

Part 1:  
Given a sequential program that accomplishes the task of counting substrings, implement a similar algorithm to accomplish the same task but utilizing multithreading to accomplish a pseudo-parallel program.

Part 2:  
The program utilizes condition variables to synchronize access between the producer and consumer threads, ensuring orderly reading from the file and sequential writing into the circular queue. This enables efficient management of data flow, maintaining the integrity of the original message order throughout the process.

Part 3:  
Given a program that utilizes a K value and a num thread parameter, modify the program to reduce the runtime. Perform analysis on the resulting runtime data.

<START PART 1 PSEUDOCODE>  
n1 >= 2n2  
(‘‘abcdab", ‘‘ab")

// don’t need validation checks because length of string 1 at least twice as long as string 2

NUM\_THREADS = 4

int currentlyAllocatedThreads = 0

int substringCount = 0

// charIndex = 0; 0 < 5 (6 – 2) + 1; 0 += 1  
// charIndex = 1; 1 < 5; 1 += 1

// charIndex = 2; 2 < 5; 2 += 1

// charIndex = 3; 3 < 5; 3 += 1

// charIndex = 4; 4 < 5; 4 += 1

// charIndex = 5; 5 !< 5  
for (int charIndex = 0; charIndex < n1 – n2 + 1; charIndex++)

// start thread 0, “abcdab”, “ab”, 6, 2, &(0 -> 1)

// thread 0 will also start at index 4 on next loop

// thread 0 “abcdab”, “ab”, 6, 2, &(1 -> 2)

// start thread 1, “abcdab”, “ab”, 6, 2, &substringCount

// start thread 2, “abcdab”, “ab”, 6, 2, &substringCount

// start thread 3, “abcdab”, “ab”, 6, 2, &substringCount

start thread (charIndex, s1, s2, n2, &substringCount)

}

// return 2

return substringCount

// thread 0, “abcdab”, “ab”, 2, &(0 -> 1)

// thread 1, “abcdab”, “ab”, 2, &(1)

// thread 2, “abcdab”, “ab”, 2, &(1)

// thread 3, “abcdab”, “ab”, 2, &(1)

// thread 4, “abcdab”, “ab”, 2, &(1 -> 2)

thread (int charIndex, string s1, string s2, int n1, int n2, int\* substringCountAddr) (

do {

// i = 0; 0 < 2; 0 += 1

// i = 1; 1 < 2; 1 += 1

// i = 2; 2 !< 2

for (int i = 0; i < n2; i++)

// THREAD 0

// s1[0 + 0] = ‘a’

// s2[0] = ‘a’

// ‘a’ = ‘a’

// s1[0 + 1] = ‘b’

// s2[1] = ‘b’

// ‘b’ = ‘b’

// completes for loop

// THREAD 1

// s1[1 + 0] = ‘b’

// s2[0] = ‘a’

// ‘b’ != ‘a’

// returns

// THREAD 2

// s1[2 + 0] = ‘c’

// s2[0] = ‘a’

// ‘c’ != ‘a’

// returns

// THREAD 3

// s1[3 + 0] = ‘c’

// s2[0] = ‘a’

// ‘c’ != ‘a’

// returns

// THREAD 0 second loop

// s1[4 + 0] = ‘a’

// s2[0] = ‘a’

// ‘a’ = ‘a’

// s1[0 + 1] = ‘b’

// s2[1] = ‘b’

// ‘b’ = ‘b’

// completes for loop

if s1[charIndex + i] doesn’t equal s2[i]

// thread 1 returns early

// thread 2 returns early

// thread 3 returns early

return

// thread 0 completed for loop

// thread 4 completed for loop

\*substringCountAddr++

charIndex += NUM\_THREAD

}

while (charIndex <= n2 – n1)

}  
  
other test cases:

(‘‘aaa", ‘‘a") = 3

(‘‘abac", ‘‘bc") = 0

# <END PART 1 PSEUDOCODE>

<START PART 2 PSEUDOCODE>

DEFINE BUFFER\_SIZE as 12

DEFINE buffer[BUFFER\_SIZE] // Circular buffer to store characters

DEFINE count = 0 // Number of items in the buffer

DEFINE in = 0 // Index for the producer to write

DEFINE out = 0 // Index for the consumer to read

DEFINE producer\_finished = 0 // Flag to indicate producer finished

DEFINE mutex // Mutex for thread synchronization

DEFINE cond\_producer // Condition variable for the producer

DEFINE cond\_consumer // Condition variable for the consumer

DEFINE producer function

OPEN file for reading

WHILE NOT end of file

LOCK mutex

WHILE count is equal to BUFFER\_SIZE

WAIT on cond\_producer using mutex

WRITE character from file to buffer[in]

in = (in + 1) MOD BUFFER\_SIZE

INCREMENT count

SIGNAL cond\_consumer

UNLOCK mutex

SET producer\_finished to 1

SIGNAL cond\_consumer

CLOSE file

DEFINE consumer function

WHILE TRUE

LOCK mutex

WHILE count is equal to 0 AND NOT producer\_finished

WAIT on cond\_consumer using mutex

IF producer\_finished AND count is equal to 0

UNLOCK mutex

BREAK

PRINT buffer[out]

out = (out + 1) MOD BUFFER\_SIZE

DECREMENT count

SIGNAL cond\_producer

UNLOCK mutex

CREATE producer thread

CREATE consumer thread

JOIN producer thread

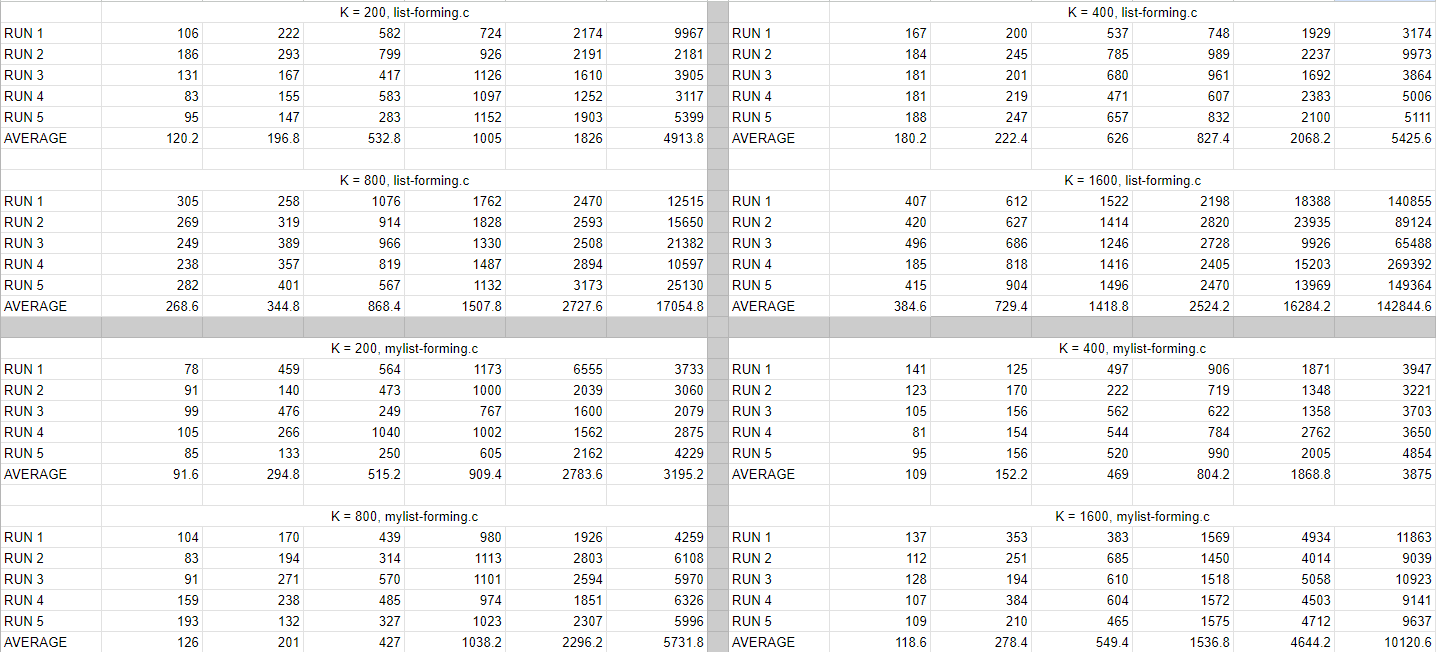
JOIN consumer thread  
<END PART 2 PSEUDOCODE>

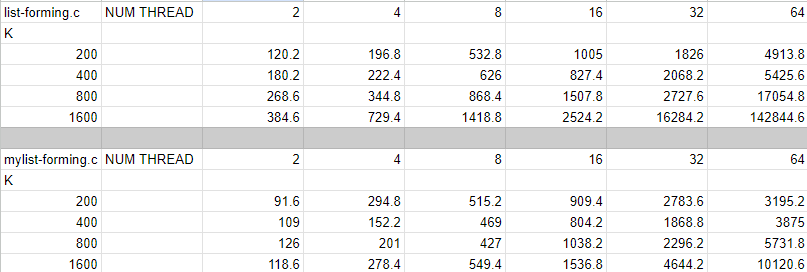
<START PART 3 PSEUDOCODE AND DESIGN OVERVIEW>  
thread\_mutex\_trylock vs thread\_mutex\_lock and thread\_mutex\_unlock  
reduce amount of times the mutex is locked and unlocked -> node insertion strategy (utilize a sublist and connect that to the global list)

This program aims to generate a global linked list by employing multiple threads, where each thread generates a local list of nodes and then merges it into the global list.

**Design Overview:**

* Thread Execution: The program creates a user-defined number of threads (num\_threads). Each thread is bound to a specific CPU core using sched\_setaffinity to potentially improve cache utilization.
* Node Generation: Each thread generates a local list of nodes (K nodes per thread) using the generate\_data\_node function. Nodes are then attached to the local list.
* Local List Formation: After generating the local list, each thread locks the global list using a mutex (pthread\_mutex\_lock). It then merges its local list into the global list and unlocks the mutex.
* Global List Maintenance: The global list is maintained through two pointers (header and tail), allowing efficient addition of nodes.

<END PART 3 PSEUDOCODE AND DESIGN OVERVIEW>  
<START PART 3 ANALYSIS>  




**Performance Results:**

The performance results indicate a notable improvement in execution speed with each run for both the number of threads (num\_threads) and the number of iterations (K). Observing the data, below are some notes taken for why these results have happened:

* Cache Utilization: By binding threads to specific CPU cores, the program potentially enhances cache utilization, reducing cache misses and improving overall performance.
* Locking Strategy: The program uses a mutex to synchronize access to the global list. While this introduces overhead due to locking and unlocking, it ensures thread safety when modifying the shared data structure.
* Local List Formation: The approach of forming local lists and merging them into the global list reduces contention for the global list, thereby minimizing synchronization overhead.
* Scalability: The program's design allows it to scale well with an increasing number of threads (num\_threads) and iterations (K). This scalability is crucial for efficiently utilizing multi-core systems.
* Memory Management: Proper memory management is maintained through the allocation and deallocation of nodes, ensuring there are no memory leaks.

<END PART 3 ANALYSIS>

# Conclusion 1- 2 Paragraphs

Part 1:  
Utilizing pthreads was relatively simple, the difficult part of part 1 came in brainstorming and implanting the parallel algorithm. While we didn’t stay 100% true to the pseudocode, we had the general idea finalized early on. While comparing the sequential solution to the custom parallel one, it was eye-opening to witness the division of the computation amongst multiple threads. Multithreading is definitely an important performance enhancement.

Part 2:  
In task 2, the code encountered a major challenge where it froze the terminal upon execution. This was due to overlooking the need to add a flag to indicate when the producer had finished its task. Once this oversight was addressed and the flag was implemented, the code ran smoothly without freezing the terminal. Despite this setback, I believe our efforts to resolve the issue were our best, and the team collaborated effectively to ensure all components were integrated seamlessly and thoroughly reviewed.

Part 3:  
During task 3, the primary challenge initially encountered was ensuring the modified version of the code ran with comparable efficiency to the original version. Specifically, implementing a method where nodes were added to a global list immediately after creation by a thread proved to be problematic. Despite multiple attempts, this approach did not yield satisfactory results. However, an alternative solution was devised where threads generated a local list of k nodes and subsequently added them to the global list in a batch process. This approach significantly improved testing outcomes.

Regarding potential improvements to the project, there isn’t any that cross my mind. However, the team operated effectively, with one member focusing on code development and the other gathering data for analysis. We believe our best efforts were invested in this project, contributing to its successful completion.

Lessons Learned  
Part 1:  
Passing data to thread functions is tricky, had to use a struct and cast a void \* to the struct pointer when in the thread. I could have just used global variables, but I figured that wouldn’t be up to par with standard programming procedures.

Part 2 and 3:  
Completing the project had its challenges, especially in task 2 where I struggled with writing the code. However, the hints provided for task 3 were helpful in guiding me through the project. The most valuable learning experience for me was during task 2, where I had to write the code independently. It deepened my understanding of thread synchronization, file operations, and queue implementation. This project significantly improved my understanding of computing and operating systems behavior, especially regarding thread management and synchronization. In the future, the skills and knowledge gained from this project will help me design and implement simultaneous systems more effectively.