**CSCI 3431: Operating System  
Winter 2022  
Project 2-Multi threading and Synchronization**

**Group Members:**

**Tushar Mahat**

**Nabin Bhandari**

**Submitted By: Instructor:**

**Tushar Mahat Dr. Jiju Poovvancheri**

Table of Contents

[1.Introduction: 3](#_Toc99657034)

[2.Methodology: 3](#_Toc99657035)

[3.Findings: 5](#_Toc99657036)

[4.Analysis: 5](#_Toc99657037)

[5.Conclusion: 7](#_Toc99657038)

[6.References 8](#_Toc99657039)

# **1.Introduction:**

The main task in this project is to design and develop a multi-threading application which involves synchronization. The program implements Floyd-Warshall (FW) All-Pairs-Shortest-Path algorithm in a multi-threaded fashion along with enforcing the readers-writers problem.

# **2.Methodology:**

The single threaded version of Floyd-Warshall (FW) All-Pairs-Shortest-Path algorithm was given in the project description, and the pseudo-code of the multi-threaded version was also taught in one of the recitations. Pseudo code approach for multi-threaded version of FW algorithm in C is described below:

*1.Define global variables INF 1000000 as large infinite value, graph which holds adjacency matrix and dist to hold distance matrix.*

*2.Make struct arg\_s to hold arguments which will be passed in each thread.*

*3.Make struct timeval to hold start and end time and a double totalTime to save total time taken by the threads in execution.*

*4.As given in Readers-Writers program, define int readCount=0, mutex readCountLock and semaphore dbLock.*

*5.The main() method of program:*

1. *Create a pointer arguments which is struct arg\_s.*
2. *Get N (# of nodes) and M (# of edges) from the user, show error message if the user inputs a negative N or M value and ask for a new value.*
3. *Allocate the size of arguments based on the value of N.*
4. *Initialize the NxN graph and dist matrices with initial values using the initializeMatrix(arguments,N,M) function. The function sets the values as follows:*

*=> graph- all the elements = 0.*

*=>dist- the diagonal elements = 0, non-diagonal elements = “INF”.*

*An example of the 3x3 matrices look like the following initially:*

*graph=, dist=*

1. *Use fillMatrix(N,M) function to get M number of u,v,w from the user and set graph[v][u]=1 if nodes i and j are connected, and set dist[v][u] =w. If the user inputs invalid u or v or w, the program shows error message and asks for u,v,w again.*
2. *Initialize mutex lock and semaphore.*
3. *Allocate memory for N threads.*
4. *Call* FloydWarshallAlg(m,arguments,threads)*[[1]](#footnote-1)*

void FloydWarshallAlg(int n, struct arg\_s \*args, pthread\_t \*threads)

{

for (int k = 0; k < n; k++)

{

//get thread start time

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

{

// set up argument to be passed to, n,k,i

// create threads -> pthread\_create

}

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

{

// join threads -> pthread\_join

}

// get thread end time

//calculate total elapsed time[[2]](#footnote-2)

}

//print time taken

void \*floydWarshall(void \*args)[[3]](#footnote-3)

{

// get n,i,k from args

for (int j = 0; j < n; j++)

{

// acquire read lock[[4]](#footnote-4)

if ((dist[i][k] + dist[k][j]) < dist[i][j])

{

// release readCount lock[[5]](#footnote-5)

// acquire write lock[[6]](#footnote-6)

dist[i][j] = dist[i][k] + dist[k][j];

// release write lock[[7]](#footnote-7)

}

else

{

// release readCount lock[[8]](#footnote-8)

}

}

pthread\_exit(NULL);

}

*6.Display the result using displayMatrix(arguments)*

*7.Destroy locks.*

# **3.Findings:**

Sample output for 10 nodes:

*A black screen with white text

Description automatically generated with low confidence Text

Description automatically generated with low confidence*

After compiling and executing both single-threaded and multi-threaded version of Floyd-Warshall Algorithm the following results were found for varying number of nodes. The table in *Figure 1* shows the number of nodes and execution time taken by both versions in seconds and the speed up (execution time of single-threaded/execution time of multi-threaded). In the case of multi-threaded version, the number of threads created are equal to the number of nodes.

|  |  |  |  |
| --- | --- | --- | --- |
| **# Of Nodes** | **Single-threading execution time in seconds(S)** | **Multi-threading execution time in seconds(M)** | **Speed Up(S/M)** |
| 10 | 0.000111 | 0.010454 | 0.01061795 |
| 50 | 0.000947 | 0.044485 | 0.02128807 |
| 100 | 0.005841 | 0.165964 | 0.03519438 |
| 500 | 0.437152 | 13.130904 | 0.03329184 |
| 1000 | 3.50294 | 102.752699 | 0.03409098 |
| 2000 | 28.858258 | 792.346074 | 0.03642128 |
| 5000 | 434.193687 | 12206.6947 | 0.03557013 |

*Figure 1: Table showing number of nodes and execution time taken by single and multi-threaded execution*

# **4.Analysis:**

The program was tested in the system with the following system specifications:

Model Name: MacBook Air

Chip: Apple M1

Total Number of Cores: 8 (4 performance and 4 efficiency)

Clock rate: 2.06 - 3.2 GHz

Cache: 16 MB

Architecture: ARM

Memory: 16 GB LPDDR4

System Firmware Version: 7459.101.2

OS Loader Version: 7459.101.2

The limit on the number of threads per process in the system is 4096.



Before testing the program, our belief was that the multi-threaded version of Floy-Warshall All Shortest Path Algorithm would be way faster than the single-threaded version. But when ran the test the results were very surprising. The graph plotted between time taken and number of nodes in *Figure 2* shows that the time taken by single-threaded version linearly increases with the number of nodes. But in case of multi-threaded version as we increase the number of threads, the time taken linearly increases till 1500 nodes and after that it increases exponentially.

*Figure 2: # Nodes vs Time taken*

The graph in *Figure 3* shows that the speedup is less than zero which means that there is no speed up in multi-threading. If we compare the numbers, the speedup is 0.01 when the number of threads is 10 and it increases up to 0.035 when the number of threads is 100. But when the number of threads is more than 100, we can see that the speed up is decreasing, and at 2000 nodes tries to increase but remains about the same after that.

*Figure 3: # Nodes vs Speed up*

After doing some depth study we found that the context switch overheads make multi-threading program slower. When there are more threads, the program goes through all the threads depending on the scheduling algorithm and more context switching makes it slower. Also, creating and destroying too many threads takes too much time. If a thread locks the data, and its time slice expires before it unlocks the data, the context switches to another thread. The thread must wait for another time slice to complete the execution, only after that it will unlock the data. In this case when one of the threads is locking the data, numerous threads cannot access it and they must wait until it is unlocked. Hence, multi-threading is taking more time.

# **5.Conclusion:**

As we worked in a group in this project, we tested the program in different system. The time consumed by a single threaded Quad-Core Intel Core i5 is less than that of an Apple M1 (8 cores). In multi-threading, the Apple M1(8 cores) takes less time than the Intel Core i5 quad-core processor. It concludes that having additional cores enhances the multi-threading as compared to having less cores and multi-threading is much slower due to synchronization of shared variable and context switching overheads.

# **6.References**

1. Project2.pdf
2. 6. Syncronization.pdf
3. https://stackoverflow.com/questions/5248915/execution-time-of-c-program

1. 6 Synchronization.pdf [↑](#footnote-ref-1)
2. https://stackoverflow.com/questions/5248915/execution-time-of-c-program [↑](#footnote-ref-2)
3. 6 Synchronization.pdf [↑](#footnote-ref-3)
4. Project 2.pdf [↑](#footnote-ref-4)
5. Project 2.pdf [↑](#footnote-ref-5)
6. Project 2.pdf [↑](#footnote-ref-6)
7. Project 2.pdf [↑](#footnote-ref-7)
8. Project 2.pdf [↑](#footnote-ref-8)