PROJECT 2

Floyd-Warshall (FW) All-Pairs-Shortest-Path
Single thread and multi-threaded

Name: Agowun Muhammad Altaf

Contents

Introduction

This report covers the steps and pseudocode used to implement Floyd-Warshall All-pairs shortest path algorithm to find the shortest path between 2 nodes with paths being undirected (paths from A to B can also be taken from B to A). This report has both a single threaded and multithreaded implementation of the algorithm and goes over the performance of these implementations.

Data structure

To store the nodes and paths we used a matrix with n-by-n dimension, where n is the number of nodes.

$$i\left\{\overbrace{\begin{array}{|c|c|c|c|}\hline 0 & 1 & 2 & 3 \\ \hline 1 & 0 & 4 & 5 \\ \hline 2 & 4 & 0 & \text{INF} \\ \hline 3 & 5 & \text{INF} & 0 \\ \hline \end{array}}^{j}\right.$$

Both i and j represents a node, thus we may consider i to be either the starting / destination node and j as either the destination / starting node respectively.
For example, if we consider i = 1 and j = 4 this means the weight of the path connecting node 1 and node 4 is 3 see figure below. We can interpret it as the weight of the path to travel from node 1 to node 4 is 3.

$$\begin{array}{c|c|c|c|c|} & & & & \overset{j}{\vee} \\ \hline \text{I} > & 0 & 1 & 2 & \boxed{3} \\ \hline & 1 & 0 & 4 & 5 \\ \hline & 2 & 4 & 0 & \text{INF} \\ \hline & 3 & 5 & \text{INF} & 0 \\ \hline \end{array}$$

NOTE:

1. All the nodes in the diagonal are 0, this is because these cells represent the weight of the path that would connect the starting node to itself (i = j), hence the weight would be 0, as the destination is where we start from.
2. Since we are assuming undirected path, the matrix is always symmetric, because the shortest path from i to j would also be the shortest path from j to i.
3. If the value is INF, this means that there is no path connecting them (ex: node 3 and node 4).

We also store a matrix that keeps track of all the nodes which are connected. (Nodes 3 and 4 are not connected, hence they are set to 0)

0	1	1	1
1	0	1	1
1	1	0	0
1	1	0	0

Single threaded implementation

For the single threaded implementation, three “for loops” are used to loop through the whole matrix (all paths), the outer loop changes the “transit” nodes (index denoted as k), and the first inner loop changes the “starting” nodes (index denoted as i) and the second inner loop changes the “destination” nodes (index denoted as j), finally the “if condition” checks to see if the path from the starting node to the destination node passing by the transit node has a lower weight than the direct connection between the starting node to the destination node, if that is the case then update both the weight from the starting to destination as well as destination to starting node.

Pseudocode for single threaded

```
for (int k = 0 : num_nodes)
{
    for (int i = 0 : num_nodes)
    {
        for (int j = 0 : num_nodes)
        {
            if (dist[i][k] + dist[k][j] < dist[i][j])
            {
                dist[i][j] = dist[i][k] + dist[k][j];
                dist[j][i] = dist[i][k] + dist[k][j];
            }
        }
    }
}
```

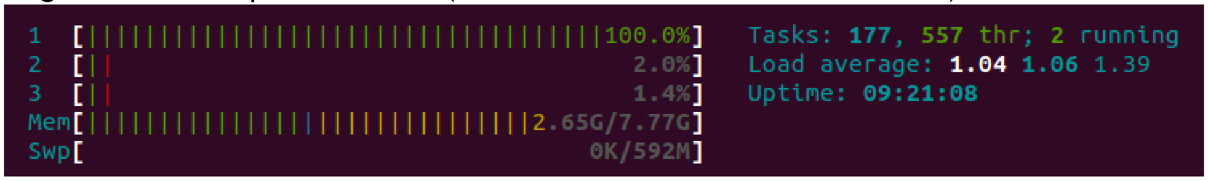
Explanation of (dist[i][k] + dist[k][j] < dist[i][j]):

dist[i][k] + dist[k][j] < dist[i][j]

If the <u>alternative path</u> has a lower weight, then overwrite the previous path (direct path)

(From start node to transit node + From transit node to destination node)
< From start node to destination node

Example screen of htop (see Appendix D for explanation of htop command) when running the single threaded implementation. (Number of nodes in screenshot= 5000)



```
1  [||||||||||||||||||||||||||||||||||||100.0%]   Tasks: 177, 557 thr; 2 running
2  [||                                   2.0%]   Load average: 1.04 1.06 1.39
3  [||                                   1.4%]   Uptime: 09:21:08
Mem[||||||||||||||||||||||||||||||||2.65G/7.77G]
Swp[                                  0K/592M]
```

General observation, only one core is at 100% utilization with user processes, the number of threads stayed around 500.

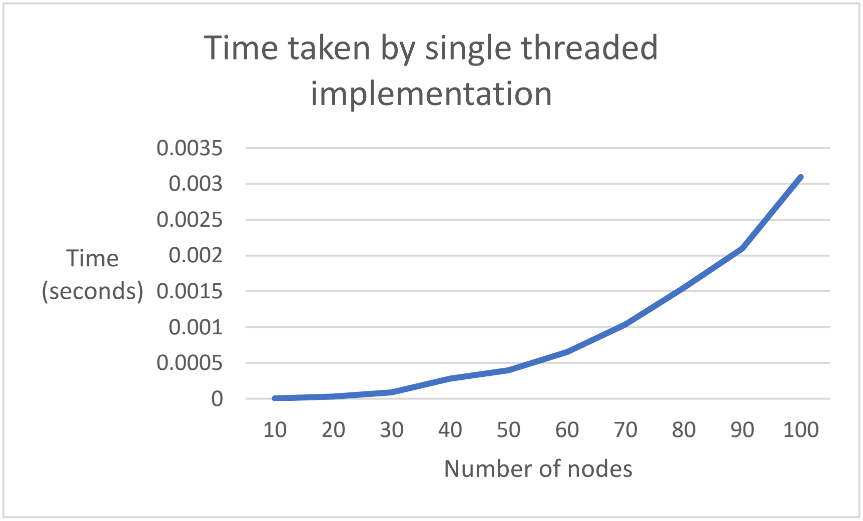
Performance of single threaded implementation

The performance of the single threaded implementation of the Floyd-Warshall all pairs shortest path algorithm was measured by encompassing the three “for loops” of the algorithm with the start and end clock then calculating the time difference (see Appendix A for explanation of the timer).

```
float fwa(int num_nodes)
{
   struct timespec start, end;

   clock_gettime(CLOCK_MONOTONIC, &start);

   for (int k = 0 : num_nodes)
   {
      for (int i = 0 : num_nodes)
      {
         for (int j = 0 : num_nodes)
         {
            if (dist[i][k] + dist[k][j] < dist[i][j])
            {
               dist[i][j] = dist[i][k] + dist[k][j];
               dist[j][i] = dist[i][k] + dist[k][j];
            }
         }
      }
   }

   clock_gettime(CLOCK_MONOTONIC, &end);
   return timeTaken(start, end);
}
```

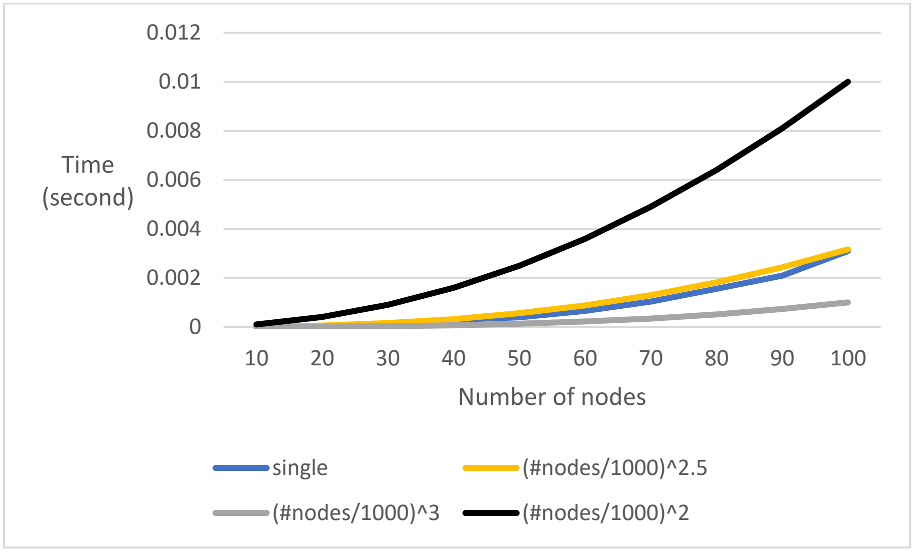
By running the algorithm with different number of nodes and taking the average of the time taken for each number of nodes (see Appendix B), the following table and graph was created.

#nodes	Time(s)
10	0.000004
20	0.00003
30	0.000089
40	0.000283
50	0.000397
60	0.000653
70	0.001036
80	0.001551
90	0.002097
100	0.003095

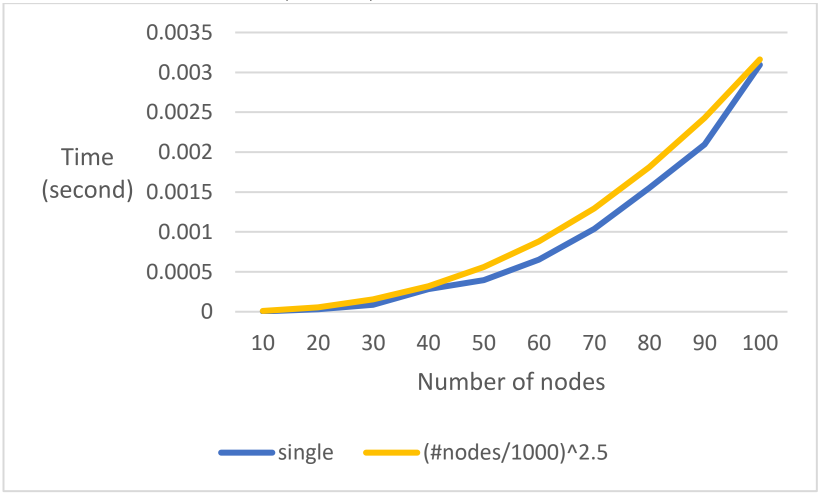


Comparing the values from the above table to common scales, we can get an idea of the time complexity of the algorithm based on the number of nodes.

#nodes	single	(#nodes/1000)^2.5	(#nodes/1000)^3	(#nodes/1000)^2
10	0.000004	0.00001	0.000001	0.0001
20	0.00003	5.65685E-05	0.000008	0.0004
30	0.000089	0.000155885	0.000027	0.0009
40	0.000282667	0.00032	0.000064	0.0016
50	0.000396667	0.000559017	0.000125	0.0025
60	0.000652667	0.000881816	0.000216	0.0036
70	0.001036	0.001296418	0.000343	0.0049
80	0.001550667	0.001810193	0.000512	0.0064
90	0.002096667	0.00243	0.000729	0.0081
100	0.003095	0.003162278	0.001	0.01



Considering only the empirical time taken represented by “single” and (#nodes/1000)^2.5 we get the following graph, which seems to suggest a correlation between time taken by the single threaded implementation and $\left(\frac{\#\,nodes}{1000}\right)^{\frac{5}{2}}$.



Multi-threaded implementation

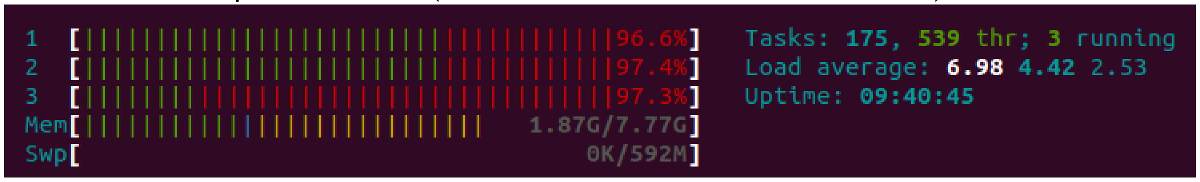
The multithreaded implementation still goes through the outer "for loops" (changes "transit" nodes) and first inner "for loops" (changes "starting" nodes) but inside the 2nd "for loop" instead of running a 3rd "for loop" it creates a thread for each starting node. Each thread then goes through the destination nodes by using a "for loop".

Pseudocode for the first 2 "for loops"

```
for (int k = 0 : num_nodes)
{

    for (int i = 0 : num_nodes)
    {
        arguments[i].n = num_nodes;
        arguments[i].i = i;
        arguments[i].k = k;
        pthread_create(threads[i], NULL, worker, arguments[i]);
    }

    for (int i = 0 : num_nodes)
    {
        pthread_join(threads[i], NULL);
    }
}
```

Explanation for thread joining:
Join the threads, which waits for the threads to exit before continuing the code, this ensures that an edge whose weight depends on a change made in an earlier iteration is made before it runs.

Example screen of htop (see Appendix D for explanation of htop command) when running the multithreaded implementation. (Number of nodes in screenshot= 1000)



```
1  [|||||||||||||||||||||||||||||||||||||96.6%]   Tasks: 175, 539 thr; 3 running
2  [|||||||||||||||||||||||||||||||||||||97.4%]   Load average: 6.98 4.42 2.53
3  [|||||||||||||||||||||||||||||||||||||97.3%]   Uptime: 09:40:45
Mem[||||||||||||||||||||||||||||   1.87G/7.77G]
Swp[                                   0K/592M]
```

General observation, all the cores/CPUs were at 100% utilization with a user processes taking about two third of the total and the rest taken by the system processes, the number of threads stayed around 500 as in single threaded implementation.

Pseudocode for each thread

```
void worker(struct arguments)
{
    // get the value of n,i and k
    int n = arguments.n;
    int i = arguments.i;
    int k = arguments.k;

    // loop through destination nodes
    for (int j = 0 : n)
    {
        ReaderEnter();
        if (dist[i][k] + dist[k][j] < dist[i][j])
        {
            ReaderLeave();

            sem_wait(&semWriter); // writer acquires the lock, or wait if any reader is in
            dist[i][j] = dist[i][k] + dist[k][j];
            dist[j][i] = dist[i][k] + dist[k][j];
            sem_post(&semWriter); // writer releases the lock as it leaves
        }
        else
        {
            ReaderLeave();
        }
    }
    pthread_exit(NULL);
}
```

Explanation:

"dist[i][k] + dist[k][j] < dist[i][j]"

is the part of the algorithm that reads from the shared variable (reader).

While,
"dist[i][j] = dist[i][k] + dist[k][j];
 dist[j][i] = dist[i][k] + dist[k][j];"

is the part that writes to the shared variable (writer).

Multiple readers can read at the same time but only one writer can change the shared variable at a time and no other readers or writers should be accessing the shared variable at that time, hence we need to implement locks to ensure the proper mutual exclusion required. Below is the pseudocode and further explanation of the high-level synchronization constructs.

Explanation for the Writer lock

Writer uses semaphore, which allows signaling so that it can be locked and unlocked by the readers.

Pseudocode for writer locks

```
sem_wait(&semWriter);
critical section
sem_post(&semWriter);
```

sem_wait will block the writer from entering the critical section, either because

1. another writer is already in the critical section or
2. there is at least one reader in the critical section

sem_post once the writer has completed its change, it unlocks the semaphore for the next writer.

Explanation for the Reader lock

When a reader enters the critical section, if it is the first reader then it should block any writer from entering (does so by locking the semaphore that the writer depends on to enter the critical section) inversely if a writer is already in the critical section, then it is the reader which locks, until the writer leaves. All readers also need to record their presence in the critical section, since multiple readers can enter the critical section at the same time, but no writer can enter, we need to keep track both of whether it is the first reader so that the write can be locked, but also if it is the last reader leaving the critical section so that the writer can be unlocked. Mutex locks are also used to ensure that the variable used to track the number of readers is not incorrectly updated.

```
void ReaderEnter()
{
    pthread_mutex_lock(&readLock);
    numReadersIn++;
    if (numReadersIn == 1)
    {
        sem_wait(&semWriter);
    }
    pthread_mutex_unlock(&readLock);
}

void readerLeave()
{
    pthread_mutex_lock(&readLock);
    numReadersIn--;
    if (numReadersIn == 0)
    {
        sem_post(&semWriter);
    }
    pthread_mutex_unlock(&readLock);
}
```

Performance of multi-threaded implementation

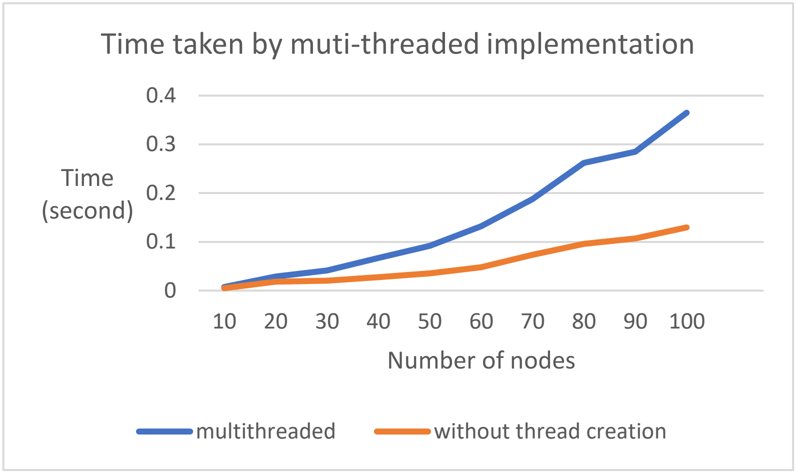
The overall performance of the multithreaded implementation of the Floyd-Warshall all pairs shortest path algorithm was measured by encompassing the outer “for loop” with the start and end timer. Moreover, since the algorithm took more time, another timer was placed to encompass the thread creation part of the algorithm to record the time taken to create all the threads (see “Reasons for multithreading taking more time” below for reason of timing thread creation).

```
float fwa(int num_nodes)
{
    struct timespec start, end, startThreadTime, endThreadTime;
    pthread_t *threads = (pthread_t *)malloc(num_nodes * sizeof(pthread_t));
    struct arg_s arguments[num_nodes];

    clock_gettime(CLOCK_MONOTONIC, &start);
    // loop through transit nodes
    for (int k = 0; k < num_nodes; k++)
    {
        clock_gettime(CLOCK_MONOTONIC, &startThreadTime);
        // loop through start nodes
        for (int i = 0; i < num_nodes; i++)
        {
            arguments[i].n = num_nodes;
            arguments[i].i = i;
            arguments[i].k = k;
            pthread_create(&threads[i], NULL, worker, (void *)&arguments[i]);      Ex: Context switch
        }

        clock_gettime(CLOCK_MONOTONIC, &endThreadTime);
        threadsTime += timeTaken(startThreadTime, endThreadTime);

        for (int i = 0; i < num_nodes; i++)
        {
            pthread_join(threads[i], NULL);
        }
    }
    clock_gettime(CLOCK_MONOTONIC, &end);
    return timeTaken(start, end);
}
```

NOTE: the time recorded by the timers for thread creation might be affected by context switches, since a context switch could happen before the second clock_gettime call, which records the end time and hence increasing the recorded amount of time taken to create the threads, this could be more likely especially with larger number of threads. Although we could place the timers closer to the thread creation itself (pthread_create), it would still not be immune to context switches but also increase the amount of time recorded due to the system having to do the code for the timers but also calculating the time taken through the formula in the timeTaken function.

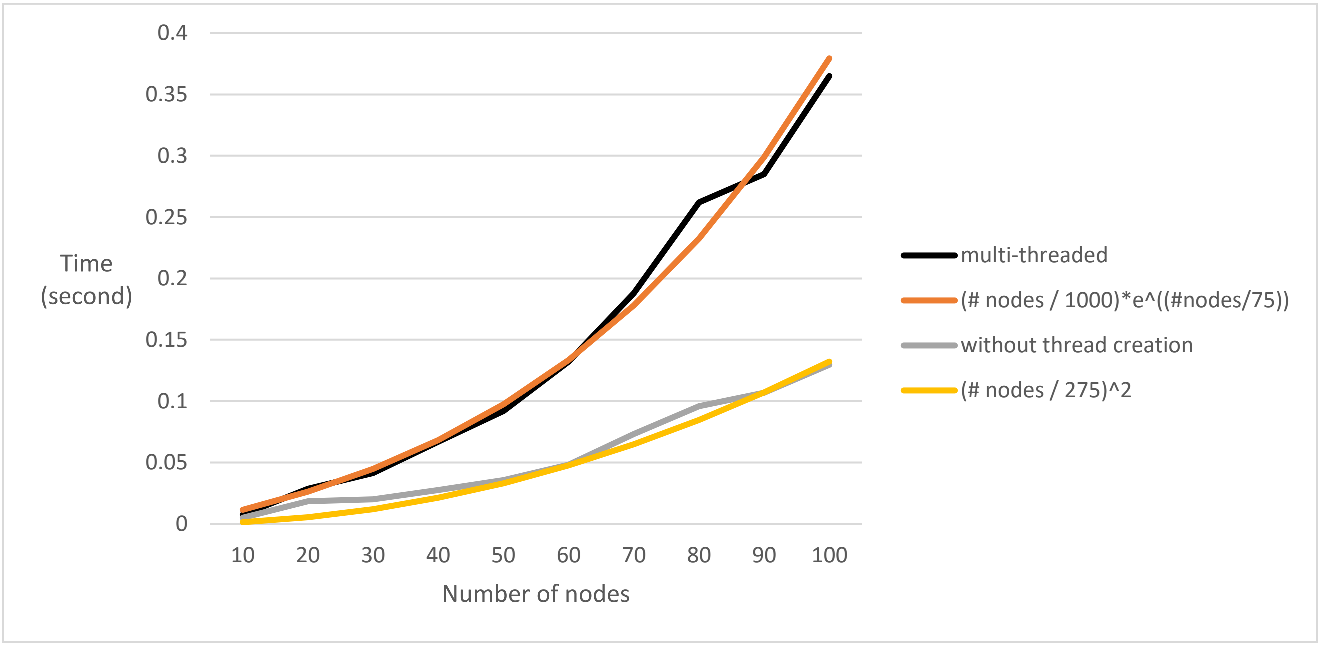
By running the algorithm with different number of nodes and taking the average of the time taken for each number of nodes (see Appendix B), the following table and graph was created.

#nodes	Multithreaded	Without thread creation
10	0.007461	0.004962667
20	0.028714667	0.018435333
30	0.041462	0.020058667
40	0.066952667	0.027438333
50	0.091985667	0.035627
60	0.132167	0.048051333
70	0.187810667	0.073204
80	0.261927	0.095938333
90	0.285036667	0.106821333
100	0.364927667	0.129640333

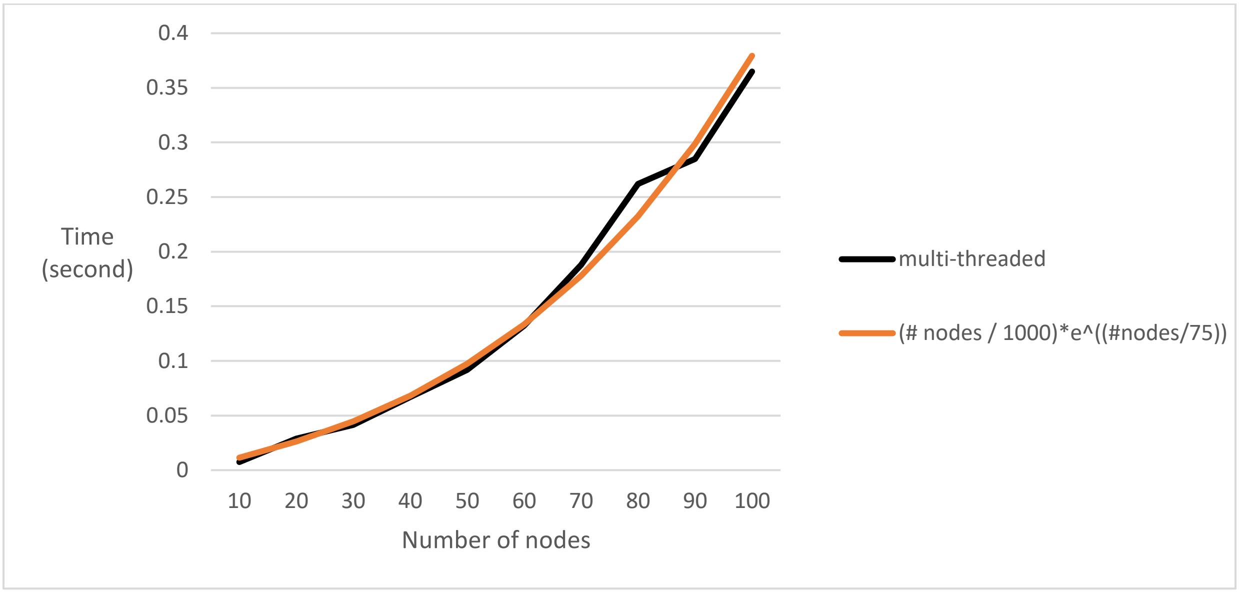


Comparing the values from the above table to common scales, we can get an idea of the time complexity of the algorithm based on the number of nodes.

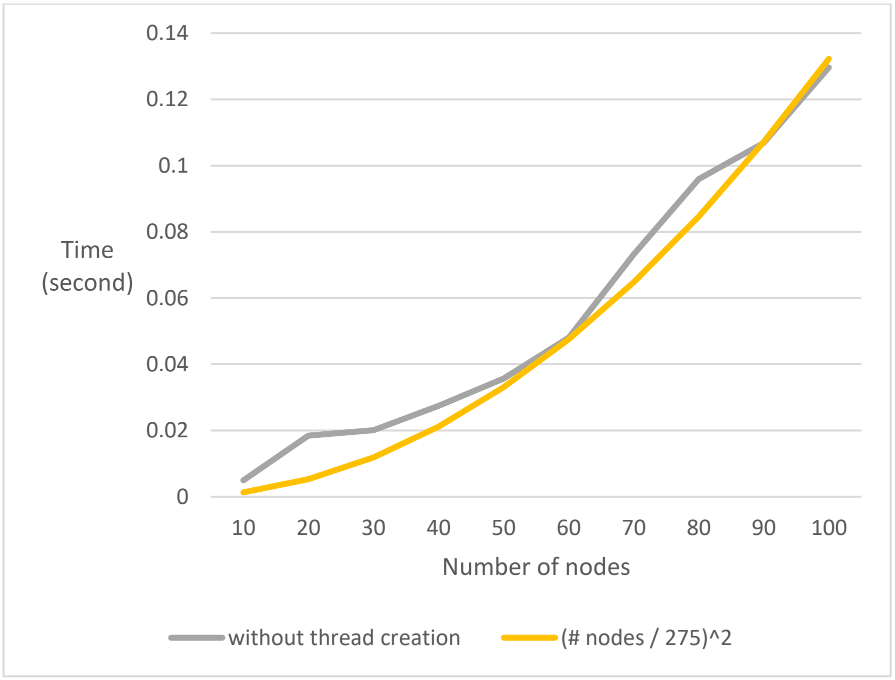
#nodes	multi-threaded	(# nodes / 1000)*e^((#nodes/75))	without thread creation	(# nodes / 275)^2
10	0.007461	0.011426308	0.004962667	0.001322314
20	0.028714667	0.026112103	0.018435333	0.005289256
30	0.041462	0.044754741	0.020058667	0.011900826
40	0.066952667	0.068184195	0.027438333	0.021157025
50	0.091985667	0.097386702	0.035627	0.033057851
60	0.132167	0.133532456	0.048051333	0.047603306
70	0.187810667	0.178008015	0.073204	0.064793388
80	0.261927	0.23245422	0.095938333	0.084628099
90	0.285036667	0.298810523	0.106821333	0.107107438
100	0.364927667	0.379366789	0.129640333	0.132231405



Considering only the empirical time taken represented by "multithreaded" and "(# nodes / 1000)*e^((#nodes/75))" we get the following graph, which seems to suggest a correlation between time taken by the multithreaded threaded implementation and $\left(\frac{\#nodes}{1000}\right)e^{\left(\frac{\#nodes}{75}\right)}$.



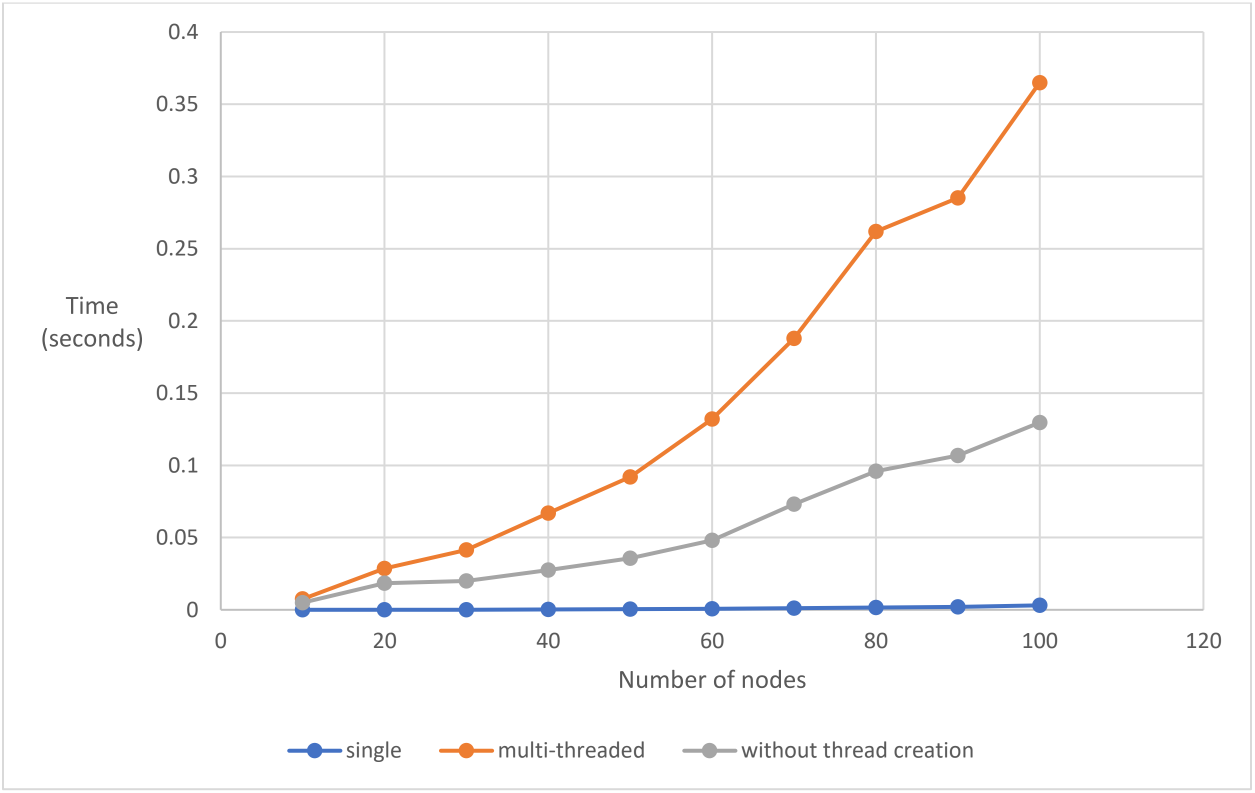
Considering only the empirical time taken without thread creation represented by "without thread creation" and "(# nodes / 275)^2" we get the following graph, which seems to suggest a correlation between time taken by the multithreaded implementation without thread creation and $\left(\frac{\#nodes}{275}\right)^2$.



Comparing the performance of single and multithreaded implementation

Comparing the graph of single threaded, multithreaded, and multithreaded subtracted the time taken to create the threads, we get the following table and graph.

#nodes	single	multi-threaded	without thread creation
10	0.000004	0.007461	0.004962667
20	0.00003	0.028714667	0.018435333
30	0.000089	0.041462	0.020058667
40	0.000282667	0.066952667	0.027438333
50	0.000396667	0.091985667	0.035627
60	0.000652667	0.132167	0.048051333
70	0.001036	0.187810667	0.073204
80	0.001550667	0.261927	0.095938333
90	0.002096667	0.285036667	0.106821333
100	0.003095	0.364927667	0.129640333



From the graph we can see that the multithreaded version is taking increasingly more time than the single threaded version, which seems to be somewhat flat due to the considerable difference it has compared to the multithreaded version.

Reasons for multithreading taking more time

The reasons behind why the multithreaded version is taking more time is partly due to the additional time in creating threads, and as seen from the graph under “Comparing the performance of single and multithreaded implementation”, we can see that by subtracting the time taken to create threads from the total time taken by the multithreaded program (line graph “without thread creation”) we can see that a considerable amount of time is lost due to thread creation.

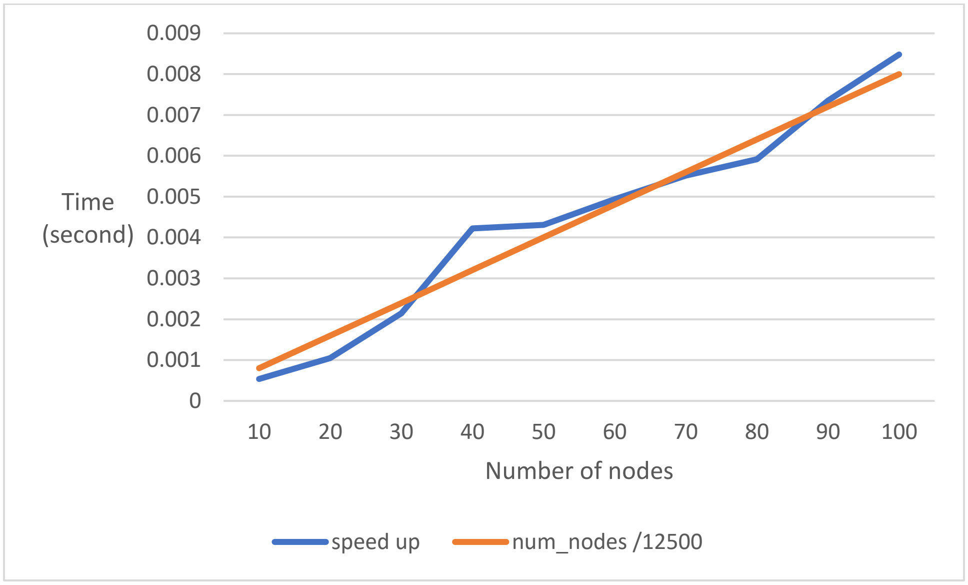
But there are also other reasons as even without the thread creation time, the multithreading version is still taking more time. This additional time could be due to the high-level synchronization construct (semaphore and mutex) as they also block the threads, which nullify the advantage of using multithreading.

In case of the Reader locks, although it allows multiple threads to read at the same time it will still block all the threads if **one** thread acquires the writer lock, similarly every thread which is trying to write to the matrix will be blocked by the semaphore if there is at least one thread reading the matrix or another thread writing to the matrix, thus considerable amount of time could be lost due to the high-level synchronization constructs.

Referring to the table in Appendix B, we can get the speed up between the multithreaded and single threaded version in relation to the number of nodes.

$$Speed\ up = \frac{time\ taken\ by\ single\ threaded\ version}{time\ taken\ by\ multithreaded\ version}$$

#nodes	single	multi-threaded	speed up	#nodes/12500
10	0.000004	0.007461	0.000536121	0.0008
20	0.00003	0.028714667	0.001044762	0.0016
30	0.000089	0.041462	0.002146544	0.0024
40	0.000282667	0.066952667	0.004221893	0.0032
50	0.000396667	0.091985667	0.00431227	0.004
60	0.000652667	0.132167	0.004938199	0.0048
70	0.001036	0.187810667	0.005516194	0.0056
80	0.001550667	0.261927	0.005920226	0.0064
90	0.002096667	0.285036667	0.00735578	0.0072
100	0.003095	0.364927667	0.008481133	0.008

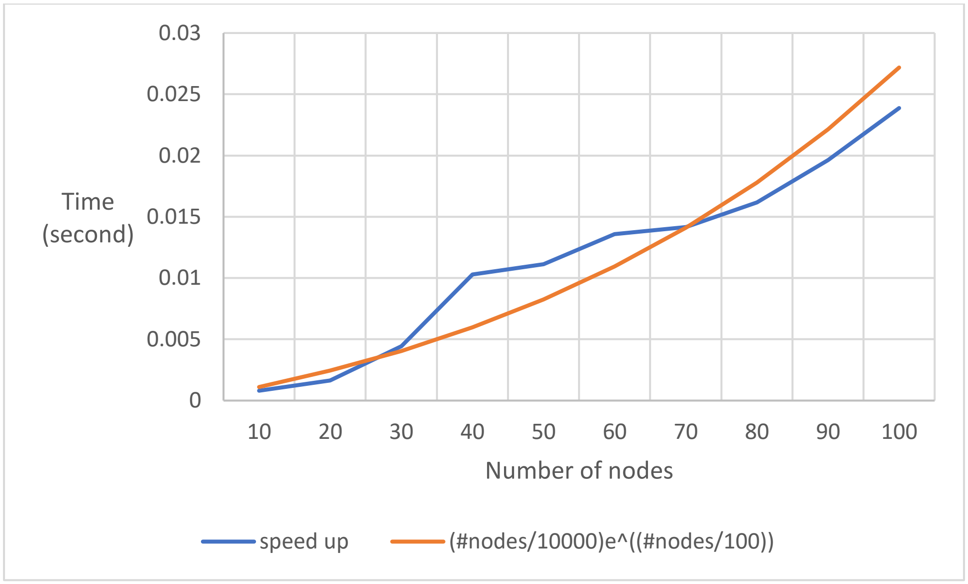


NOTE: the value on the y-axis are less than 1, thus there is no speed up, instead the multithreaded version is actually slower, although it seems that the multithreaded version will eventually cause a speed up (when speed up > 1) but not only will this happen approximately when the number of nodes exceeds 12500 nodes according to the trend line created, but this might not be the case as shown below under Performance of large number of nodes.

We can also look at the speed up considering no thread creation,

$$Speed\ up = \frac{time\ taken\ by\ single\ threaded\ version}{time\ taken\ by\ multithreaded\ version\ (without\ thread\ creation)}$$

#nodes	single	without thread creation	speed up	(#nodes/10000)e^((#nodes/100))
10	0.000004	0.004962667	0.000806018	0.001105171
20	0.00003	0.018435333	0.00162731	0.002442806
30	0.000089	0.020058667	0.004436985	0.004049576
40	0.000282667	0.027438333	0.010301889	0.005967299
50	0.000396667	0.035627	0.011133878	0.008243606
60	0.000652667	0.048051333	0.013582696	0.010932713
70	0.001036	0.073204	0.014152232	0.014096269
80	0.001550667	0.095938333	0.01616316	0.017804327
90	0.002096667	0.106821333	0.01962779	0.022136428
100	0.003095	0.129640333	0.023873743	0.027182818



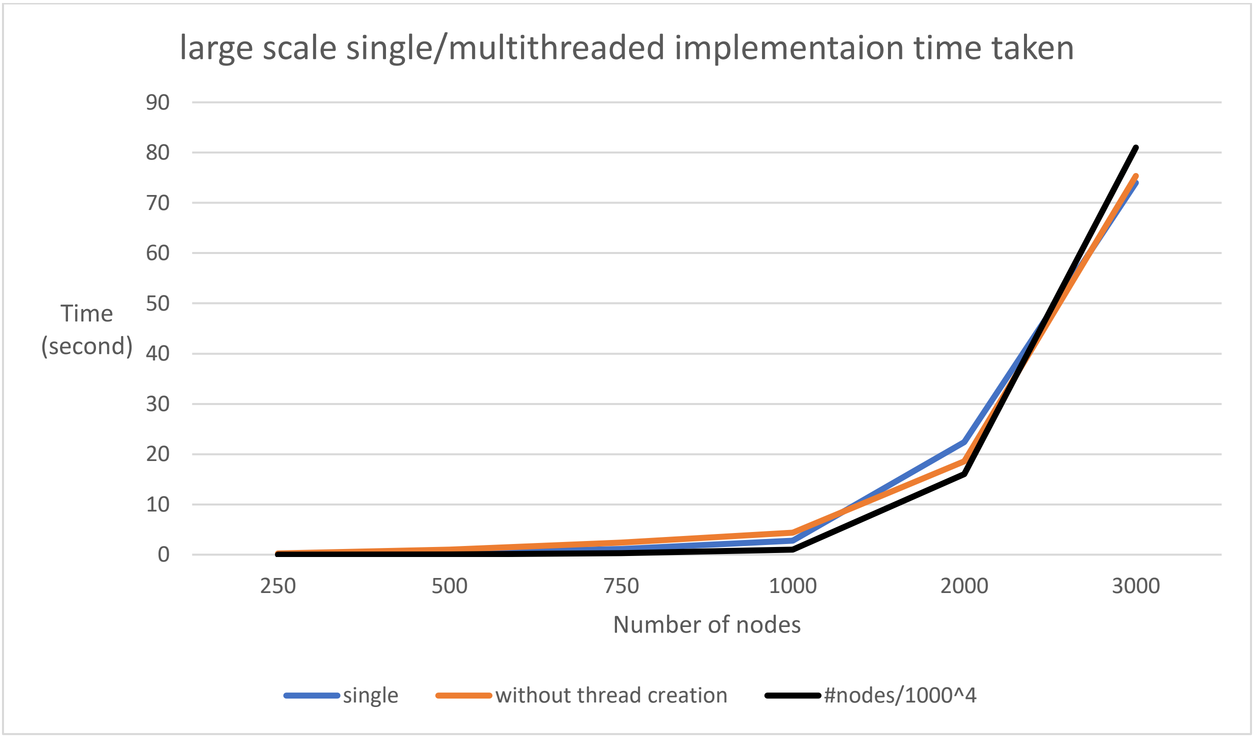
Although it seems that eventually, the multithreaded application will be faster than the single threaded version (speed up > 1), it seems to not be the case as at larger number of nodes, the multithreaded implementation becomes even worst. (see Performance of large number of nodes)

Performance of Large number of nodes

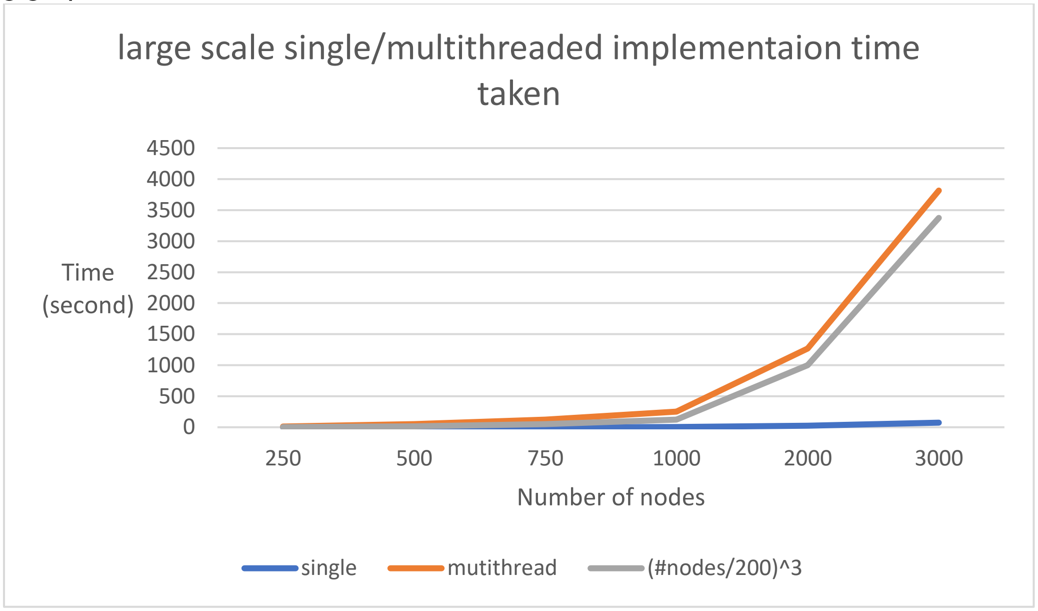
Comparing the time taken for the two implementations on large number of nodes to see if there is a difference as the number of nodes gets larger. The table below was made using the average data's shown in Appendix E.

#nodes	single	without thread creation	#nodes/1000^4	mutithread	(#nodes/200)^3
250	0.043786	0.245938333	0.00390625	11.87787433	1.953125
500	0.3765364	1.032854667	0.0625	51.34816867	15.625
750	1.133599	2.389307667	0.31640625	121.8588843	52.734375
1000	2.779345333	4.407684333	1	252.4441323	125
2000	22.41691533	18.57686367	16	1269.611572	1000
3000	73.99520633	75.35188767	81	3815.743083	3375

Considering only single threaded and multithreaded without thread creation the following graph is obtained, suggesting that the trend would follow $\left(\frac{\#nodes}{1000}\right)^4$. Which is worst than before $\left(\frac{\#nodes}{1000}\right)^{2.5}$ for single threaded and $\left(\frac{\#nodes}{275}\right)^2$ for multithreaded without thread creation.



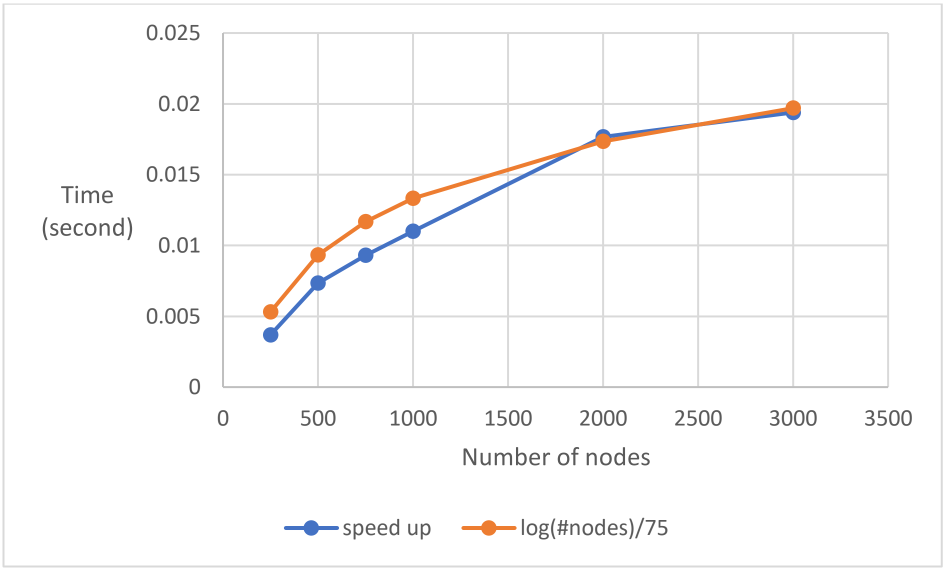
Now considering single threaded and multithreaded with thread creation, we get the following graph.



As we can see even at large number of nodes, the time taken by the multithreaded version is still considerably higher than that taken by the single threaded.

Calculating the speed up we can see that the speed up seems to flatten $\left(\frac{\log(\#nodes)}{75}\right)$, thus showing that it might not approach 1 as the number of nodes continues to increase.

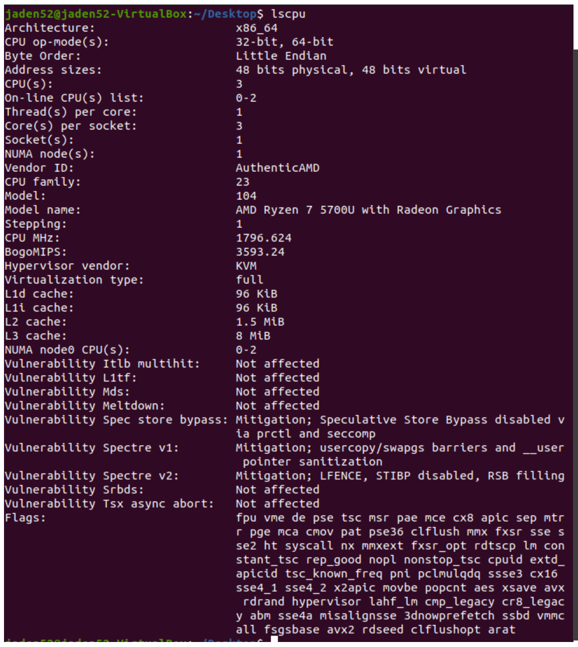
#nodes	speed up	log(#nodes)/75
250	0.003686	0.005306
500	0.007333	0.00932
750	0.009303	0.011667
1000	0.01101	0.013333
2000	0.017657	0.017347
3000	0.019392	0.019695



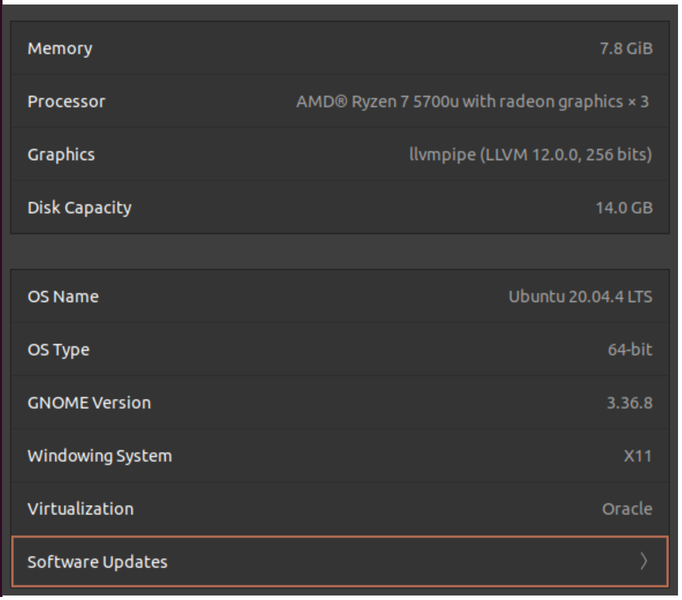
System specification

Using the command “lscpu” the following details were obtained from the system (virtual machine).

OS: ubuntu 20.04.4 LTS
Manufacturer: AMD
CPU model: Ryzen 7 5700U with Radeon graphics
Number of cores/CPU: 3
Architecture: x86_64
RAM: 7.8 GiB
Clock rate: 1796.624 MHz



```
jaden52@jaden52-VirtualBox:~/Desktop$ lscpu
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   48 bits physical, 48 bits virtual
CPU(s):                          3
On-line CPU(s) list:             0-2
Thread(s) per core:              1
Core(s) per socket:              3
Socket(s):                       1
NUMA node(s):                    1
Vendor ID:                       AuthenticAMD
CPU family:                      23
Model:                           104
Model name:                      AMD Ryzen 7 5700U with Radeon Graphics
Stepping:                        1
CPU MHz:                         1796.624
BogoMIPS:                        3593.24
Hypervisor vendor:               KVM
Virtualization type:             full
L1d cache:                       96 KiB
L1i cache:                       96 KiB
L2 cache:                        1.5 MiB
L3 cache:                        8 MiB
NUMA node0 CPU(s):               0-2
Vulnerability Itlb multihit:     Not affected
Vulnerability L1tf:              Not affected
Vulnerability Mds:               Not affected
Vulnerability Meltdown:          Not affected
Vulnerability Spec store bypass: Mitigation; Speculative Store Bypass disabled v
                                 ia prctl and seccomp
Vulnerability Spectre v1:        Mitigation; usercopy/swapgs barriers and __user
                                  pointer sanitization
Vulnerability Spectre v2:        Mitigation; LFENCE, STIBP disabled, RSB filling
Vulnerability Srbds:             Not affected
Vulnerability Tsx async abort:   Not affected
Flags:                           fpu vme de pse tsc msr pae mce cx8 apic sep mtr
                                 r pge mca cmov pat pse36 clflush mmx fxsr sse s
                                 se2 ht syscall nx mmxext fxsr_opt rdtscp lm con
                                 stant_tsc rep_good nopl nonstop_tsc cpuid extd_
                                 apicid tsc_known_freq pni pclmulqdq ssse3 cx16 
                                 sse4_1 sse4_2 x2apic movbe popcnt aes xsave avx
                                  rdrand hypervisor lahf_lm cmp_legacy cr8_legac
                                 y abm sse4a misalignsse 3dnowprefetch ssbd vmmc
                                 all fsgsbase avx2 rdseed clflushopt arat
```



Memory	7.8 GiB
Processor	AMD® Ryzen 7 5700u with radeon graphics × 3
Graphics	llvmpipe (LLVM 12.0.0, 256 bits)
Disk Capacity	14.0 GB
OS Name	Ubuntu 20.04.4 LTS
OS Type	64-bit
GNOME Version	3.36.8
Windowing System	X11
Virtualization	Oracle

Supportive Function

These are common functions to both implementation which are used to run the program.

Input:

Getting the input from user, in the format below

num_nodes num_edges

$$\left.\begin{array}{l}\text{node1 node2 weight}\\\text{node1 node2 weight}\\\text{node1 node2 weight}\end{array}\right\}\text{num\_edges}$$

```
int readInput()
{
    int num_nodes, num_edges;
    int edge[3];
    printf("Input:\n");
    scanf("%d %d", &num_nodes, &num_edges);
    // ensure the number of nodes for the matrix is at least 3
    while (num_nodes < 3)
    {
        printf("There must be at least 3 nodes for the algorithm to make sense\n");
        printf("Re-enter the number of nodes: ");
        scanf("%d", &num_nodes);
    }
    // ensure that the number of nodes requested to assign weight to is sensible
    while (num_edges < 0 || num_edges > num_nodes * num_nodes)
    {
        printf("Illegal number of edges to be inserted\n");
        printf("Re-enter the number of edges to add weight to: ");
        scanf("%d", &num_edges);
    }
    // initialize matrices
    graph = malloc(num_nodes * sizeof(int *));
    dist = malloc(num_nodes * sizeof(int *));

    for (int i = 0; i < num_nodes; i++)
    {
        graph[i] = malloc(num_nodes * sizeof(int));
        dist[i] = malloc(num_nodes * sizeof(int));
        for (int j = 0; j < num_nodes; j++)
        {
            graph[i][j] = 0;                    // set the empty edge to 0
            dist[i][j] = (i == j) ? 0 : MAX_VAL; // set the empty edge to infinity (0, when same node pointed
by i and j)
        }
    }

    for (int i = 0; i < num_edges; i++)
    {
        scanf("%d %d %d", &edge[0], &edge[1], &edge[2]);
        while (!validateEdgeInput(edge, num_nodes))
        {
            printf("Previous line is ignored due to incorrect value, please enter valid inputs!\n");
            scanf("%d %d %d", &edge[0], &edge[1], &edge[2]);
        }

        // assign weight to edges in dist matrix and set graph matrix to 1
        dist[edge[0] - 1][edge[1] - 1] = edge[2];
        graph[edge[0] - 1][edge[1] - 1] = 1;
        dist[edge[1] - 1][edge[0] - 1] = edge[2];
        graph[edge[1] - 1][edge[0] - 1] = 1;
    }

    return num_nodes;
}
```

Validate input for edges

- ensures matrix values where i = j, meaning node to themselves stays 0
- ensure user requests to enter weigh for a correct node
- ensure weight assign is within range of 0 to MAX_VAL (INF)

```
int validateEdgeInput(int *edge, int num_nodes)
{
    int valid = 1;

    if (edge[0] == edge[1])
    {
        printf("weight of nodes to themselves is always 0!\n");
        valid = 0;
    }

    for (int i = 0; i < 2; i++)
    {
        if ((edge[i] < 1 || edge[i] > num_nodes))
        {
            printf("%s node %d does not exist!\n", i == 0 ? "Starting" : "Destination", edge[i]);
            valid = 0;
        }
    }

    if (edge[2] < 0 || edge[2] > MAX_VAL)
    {
        printf("Weight %d is too %s!\n", edge[2], edge[2] < 0 ? "small" : "large");
        valid = 0;
    }

    return valid;
}
```

Display matrix

Takes the matrix to be displayed and the number of nodes in the matrix. The outer loop changes the row to which to output to and the inner loop changes the column to write to. the “if condition” ensures that INF is displayed if there is no edge connecting the nodes.

```
void displayMatrix(int **matrix, int num_nodes)
{
    printf("\nOutput:\n");
    for (int i = 0 : num_nodes)
    {
        for (int j = 0 : num_nodes)
        {
            if (matrix[i][j] >= MAX_VAL)
            {
                printf("INF ");
            }
            else
            {
                printf("%d ", matrix[i][j]);
            }
        }
        printf("\n"); // next line
    }
}
```

Appendix A

To get the time taken by the algorithm, the code below is used (reference: https://stackoverflow.com/questions/16275444/how-to-print-time-difference-in-accuracy-of-milliseconds-and-nanoseconds-from-c)

The header file time.h is required for the timer implementation

```
#include <time.h>
```

It contains “timespec” struct which is used to store the value obtained from the clock.

```
struct timespec {
    time_t   tv_sec;        /* seconds */
    long     tv_nsec;       /* nanoseconds */
};
```

Thus, we create two instances to record the time at the start of the algorithm and one to record the time at the end of the algorithm.

```
struct timespec start, end;
```

then using the following line, we “start” the timer, by getting the current time at that point in time

```
clock_gettime(CLOCK_MONOTONIC, &start);
```

Then at the end of the block of code which we are interested in recording the time for place the following line to “stop” the timer, by getting the current time at that point in time.

```
clock_gettime(CLOCK_MONOTONIC, &end);
```

using these 2 timestamps, we use the following calculation to get the amount of time taken by the algorithm in terms of seconds. (Encapsulate the formula so that it can be used multiple times)

```
double timeTaken(struct timespec start, struct timespec end)
{
    return ((double)end.tv_sec + 1.0e-9 * end.tv_nsec) - ((double)start.tv_sec + 1.0e-9 *
start.tv_nsec);
}
```

tv_sec is the number of seconds at that time, and tv_nsec is the number of nanoseconds at that time, thus we convert the nanosecond into seconds by multiplying the number of nanoseconds by 1e-9.

Appendix B

The table below contains the time for the multiple runs for both the single threaded and multithreaded implementation to go through different number of nodes (see Appendix C for how matrices were created). The table contains the time taken by the single threaded, multithreaded without the time taken for thread creation and the same with the amount of time taken by thread creation included. The previous last row is the average time taken and the last is for the ratio/speed up (time taken single threaded/ time taken multithreaded).

#nodes	10			20			…
threads	single	w/-creation	multithread	single	w/-creation	multithread	…
1	0.000004	0.001407	0.003578	0.00003	0.017648	0.028096	…
2	0.000004	0.005041	0.007709	0.00003	0.021964	0.033266	…
3	0.000004	0.00844	0.011096	0.00003	0.015694	0.024782	…
average	0.000004	0.004962667	0.007461	0.00003	0.018435333	0.028714667	…
ratio	0.000536121			0.00104			…

…	30			40			…
…	single	w/-creation	multithread	single	w/-creation	multithread	…
…	0.000086	0.019296	0.036658	0.000191	0.025379	0.061413	…
…	0.000096	0.021586	0.044636	0.000445	0.026872	0.066709	…
…	0.000085	0.019294	0.043092	0.000212	0.030064	0.072736	…
…	0.000089	0.020058667	0.041462	0.000282667	0.027438333	0.066952667	…
…	0.002147			0.004221888			…

…	50			60			…
…	single	w/-creation	multithread	single	w/-creation	multithread	…
…	0.000379	0.031249	0.086771	0.000662	0.050747	0.132726	…
…	0.000426	0.038142	0.093737	0.000634	0.042672	0.134899	…
…	0.000385	0.03749	0.095449	0.000662	0.050735	0.128876	…
…	0.000396667	0.035627	0.091985667	0.000652667	0.048051333	0.132167	…
…	0.004312266			0.004938197			…

…	70			80			…
…	single	w/-creation	multithread	single	w/-creation	multithread	…
…	0.001041	0.069251	0.173229	0.001579	0.105398	0.311792	…
…	0.001002	0.094376	0.228839	0.001574	0.087559	0.244146	…
…	0.001065	0.055985	0.161364	0.001499	0.094858	0.229843	…
…	0.001036	0.073204	0.187810667	0.001550667	0.095938333	0.261927	…
…	0.005516			0.005920225			…

…	90			100		
…	single	w/-creation	multithread	single	w/-creation	multithread
…	0.002055	0.103345	0.277158	0.002825	0.137396	0.386788
…	0.002047	0.10052	0.27703	0.00284	0.127736	0.358215
…	0.002188	0.116599	0.300922	0.00362	0.123789	0.34978
…	0.002096667	0.106821333	0.285036667	0.003095	0.129640333	0.364927667
…	0.007355779			0.008481		

Appendix C

Complete graphs were created to test the algorithm on different number of nodes, to do so the following program was used.
The “testing” program get the number of nodes from the user (**n**) and create a file (named “RandomNodes_**n**.txt”) that contain the inputs for the Floyd-Warshall all pairs shortest path algorithm.
Since the algorithm it is intended for an undirected graph, only half the matrix is populated, the other half is populated by the algorithm itself as it implements undirected graph from the inputs, only the values in the blue cells are populated with a value between 1 and MAXVAL.

0	🟦	🟦	🟦
	0	🟦	🟦
		0	🟦
			0

Input structure:

```
num_nodes num_edges
node1 node2 weight
node1 node2 weight      num_edges
node1 node2 weight
```

```
const MAXVAL 100
int main()
{
   FILE *fp;
   String filename;

   int numNodes;
   input(“Please, enter the number of nodes: “, numNodes);

   filename = "RandomNodes_${numNodes}.txt";

   fp = fopen(filename, "w");

   fprintf(fp, "%d %d\n", numNodes, ((numNodes * numNodes) / 2 - (numNodes / 2)));

   for (int i = 0 : numNodes – 1)
   {
      for (int j = i + 1 : numNodes)
      {
         fprintf(fp, "%d %d %d\n", i + 1, j + 1, (rand() % MAXVAL) + 1);
      }
   }

   fclose(fp);
}
```

Appendix D

htop reference: (https://www.linuxfordevices.com/tutorials/linux/htop-command-in-linux#:~:text=htop%20command%20is%20a%20Linux,switching%20between%20values%20and%20tabs)

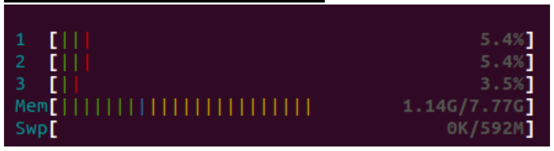
For further monitoring, we may use the command "htop" to monitor the CPU during execution.

NOTE: If the system is ubuntu then it needs to be downloaded by using the command:

sudo apt-get install htop

Key details:

CPU and Memory Usage



```
1  [|||                                  5.4%]
2  [|||                                  5.4%]
3  [||                                   3.5%]
Mem[|||||||||||||||||||||||      1.14G/7.77G]
Swp[                                0K/592M]
```

The count 1-3, represents the cores/CPUs of the system. The bar describes the amount and type of processes using each core. The value against the bar denotes the percentage each core is being consumed.

Color coding for CPU

Green – Amount of CPU consumed by the user's processes.

Red – Amount of CPU used by system processes.

Grey – Amount of CPU used for Input/Output based processes.

'mem' bar represents the main memory (or RAM), whereas 'swp' refers to the swap memory.

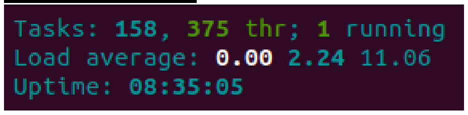
Color coding for Memory

Green – Percentage of RAM being used for running processes in the system.

Blue – Percentage of RAM being consumed by buffer pages.

Orange – Percentage of RAM being used for cache memory.

Task statistics



```
Tasks: 158, 375 thr; 1 running
Load average: 0.00 2.24 11.06
Uptime: 08:35:05
```

Task: 158 the number of current tasks/processes in the system

These 158 processes are handled by '375' number of threads (thr).

Among 158 tasks, only a single task is in the state of running.

Load Average – Since this is a triple-core system, the maximum amount of load is 3.0. The values mentioned are moving averages over different periods of time.

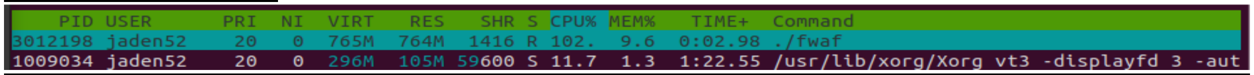
'0.00' – Average load for the last minute.

'2.24' – Average load for the last 5 minutes.

'11.06' – Average load for the last 15 minutes.

Uptime – Amount of time since the last system reboot.

Process information



```
    PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
3012198 jaden52    20   0  765M  764M  1416 R 102.  9.6  0:02.98 ./fwaf
1009034 jaden52    20   0  296M  105M 59600 S 11.7  1.3  1:22.55 /usr/lib/xorg/Xorg vt3 -displayfd 3 -aut
```

CPU% – The percentage of CPU used by the process.

MEM% – The percentage of Memory consumed by the process.

TIME+ – The period of time since the process initiated.

Command – The complete command for the process with program name and arguments

Appendix E

The table below shows the values for the large number of nodes matrices for the single threaded, multithreaded, and multithreaded without thread creation time. it contains empirical values obtained from running the program and the average time taken.

	single				…
#nodes	1	2	3	average	…
250	0.045801	0.043189	0.042368	0.043786	…
500	0.3999902	0.380904	0.348715	0.3765364	…
750	1.154514	1.124111	1.122172	1.133599	…
1000	2.775693	2.782502	2.779841	2.779345333	…
2000	22.189104	22.368273	22.693369	22.41691533	…
3000	74.48304	74.22493	73.277649	73.99520633	…

multithreaded				…
1	2	3	average	…
11.974744	11.722907	11.935972	11.87787433	…
51.960449	50.740753	51.343304	51.34816867	…
120.477516	122.557213	122.541924	121.8588843	…
247.060974	252.792694	257.478729	252.4441323	…
1295.697754	1280.381714	1232.755249	1269.611572	…
3809.787598	3815.140137	3822.301514	3815.743083	…

without thread creation			
1	2	3	average
0.252457	0.238367	0.246991	0.245938333
1.019936	1.047836	1.030792	1.032854667
2.374565	2.398415	2.394943	2.389307667
4.294861	4.461426	4.466766	4.407684333
18.992554	18.820312	17.917725	18.57686367
74.924072	80.226562	70.905029	75.35188767

References

- Measuring time: https://stackoverflow.com/questions/16275444/how-to-print-time-difference-in-accuracy-of-milliseconds-and-nanoseconds-from-c

- htop explanation:https://www.linuxfordevices.com/tutorials/linux/htop-command-in-linux#:~:text=htop%20command%20is%20a%20Linux,switching%20between%20values%20and%20tabs