Operating Systems
Assignment 2 - Multithreaded Floyd-Warshall Algorithm
Ben Jollymore A00400128
Jonathan Power A00405363

Introduction

The Floyd Warshall Algorithm is used to find the shortest distance between each node in a matrix. This project investigates single and multithreaded implementations of this algorithm, and compares the efficiency between the two.

General Approach

This is the pseudocode for the overall approach that we took in developing this program.

```
//struct to pack data into for thread access
struct {
	number of nodes
	i value
	k value
	pointer to shortest distance array
} thread_data;

//global shortest distance matrix
shortest distance matrix;

//function for each thread to execute
void runner(thread_data struct){
	unpack data in struct into local variables;

	//do final block of FW algorithm
	for (j = 1 to number of nodes; j++){

	if (
		//make sure read is protected by mutex
		lock read mutex;
		dist[i][k] + dist[k][j] < dist[i][j]
		unlock read mutex;
	 )
		//lock mutex prior to writing
		lock write mutex;
		dist[i][j] = dist[i][k] + dist[k][j]
		unlock write mutex;
	}
}

int main(){

//recieve number of nodes and edges from user
scanf(number of nodes);
scanf(number of edges);

//initialize matrix of nodes and matrix of weighted edges between nodes
shortest distance matrix[num nodes][num nodes];
```

```
weighted edge matrix[3][num edges];

//place values into weighted edge matrix
for (i = 0 to number of edges; i+=) {
	scanf(node1, node2, edge weight)
	weighted edge matrix[0][i] = node 1;
	weighted edge matrix[1][i] = node 2;
	weighted edge matrix[2][i] = weighted edge;
}

//place initial values in shortest distance matrix
for (i = 1; to number of nodes; i++){
		for (j = 1; to number of nodes; j++) {
			if (i == j)
				distance matrix[i][j] = 0;
			else
				distance matrix[i][j] = Infinity
		}
	}

//start FW algorithm
for (k = 1 to number of nodes; k++){
	for (i = 1 to number of nodes; i++){
	//create thread for each instance of i
	assemble data into instance thread_data struct;
	create thread(thread_data, runner, i-1);
	}
	//join threads before next iteration of k
	join threads(number of nodes);
}

printf(final distance matrix);

return 0;

}
```

Troubleshooting

1. **Creating threads too fast**

   When creating threads within the i loop, data to be passed through to each new thread was “packed” into a single thread_data struct, and each thread was given the address of that struct to retrieve the data from when it started running. On high end CPUs, this worked perfectly fine, and the algorithm functioned as it should.

   However, on lower end CPUs, the algorithm would not properly execute, as some nodes were never calculated. After some rudimentary debugging using print statements, it was uncovered that on some CPUs, multiple threads were executing with the same data “packet”, although every thread is supposed to have a distinct packet.

   The issue turned out to be due to the fact that on lesser CPUs, the threads did not have time to initialize and pull data from the address due to scheduling, such that by the time it was able to initialize and access the data, it had been overwritten for use by another

thread. This was solved by adding a nanosecond sleep after each thread was created, such that it has time to access the data before it is overwritten.

2. **Going out of bounds in memory allocation**
   Our algorithm worked perfectly fine for up to 5 nodes, however, after 5 nodes, the program would crash at runtime due to a Segmentation Fault, as the heap could not be resized in malloc.c. I passed the program through Valgrind, and was notified of severe memory leaks on several lines in the main Assignment2.c file. Some of those lines contained malloc calls to initialize the shortest distance array. Simply changing (sizeof(int)*nodes) to (sizeof(int)*nodes+1) solved the segmentation fault as it was no longer going out of bounds, however I was perplexed as to why it would work for up to 5 nodes and then crash at 6. After reading through malloc.c, it seemed that the heap would allocate data in an initial malloc call, and then resize if it needed more data. I theorized that it was able to initially allocate enough data to handle up to 5 nodes, but would fail when attempting to resize to handle 6 nodes.

3. **Memory Leaks**
   As mentioned in the previous segment, Valgrind exposed a plethora of memory leaks, mainly associated with malloc statements and calls to potentially uninitialized variables in our helper functions.
   By reducing the number of unnecessary malloc calls and changing how thread_data structs were both passed and initialized by the runner() function to not use mallocs and instead void pointers, we were able to not fully eliminate the number of memory leaks, but at least reduce the quantity.

4. **Read lock helper function**
   When trying to implement the mutex to solve the reader/writer problem in regard to this particular implementation, I did a decent amount of research and came upon the build in pthread_rwlock, which creates a single mutex to satisfy the reader/writer problem. However, the read statement is nested within an if statement, making it difficult to isolate the read statement such that it can be surrounded by the read mutex. To solve this, I implemented a small helper function readHelper(), which returns the result of the statement evaluated, however surrounds the statement with a read mutex to prevent data corruption. That function is then called from within the if statement.

5. **Trying to reduce overheads**
   Helper functions and nanosleep were reducing performance by a marginal amount. Pthreads require system resources to execute and by using a function such as createThreads(...) we require additional system resources. In order for the program To function correctly, these helper functions were required, requiring the system to acquire additional overhead. As mentioned earlier about the speed of pthread creation, we required the need of a sleep to act as a temporary interrupt after every creation to allow time for a thread to be created and execute. This in turn adds an additional

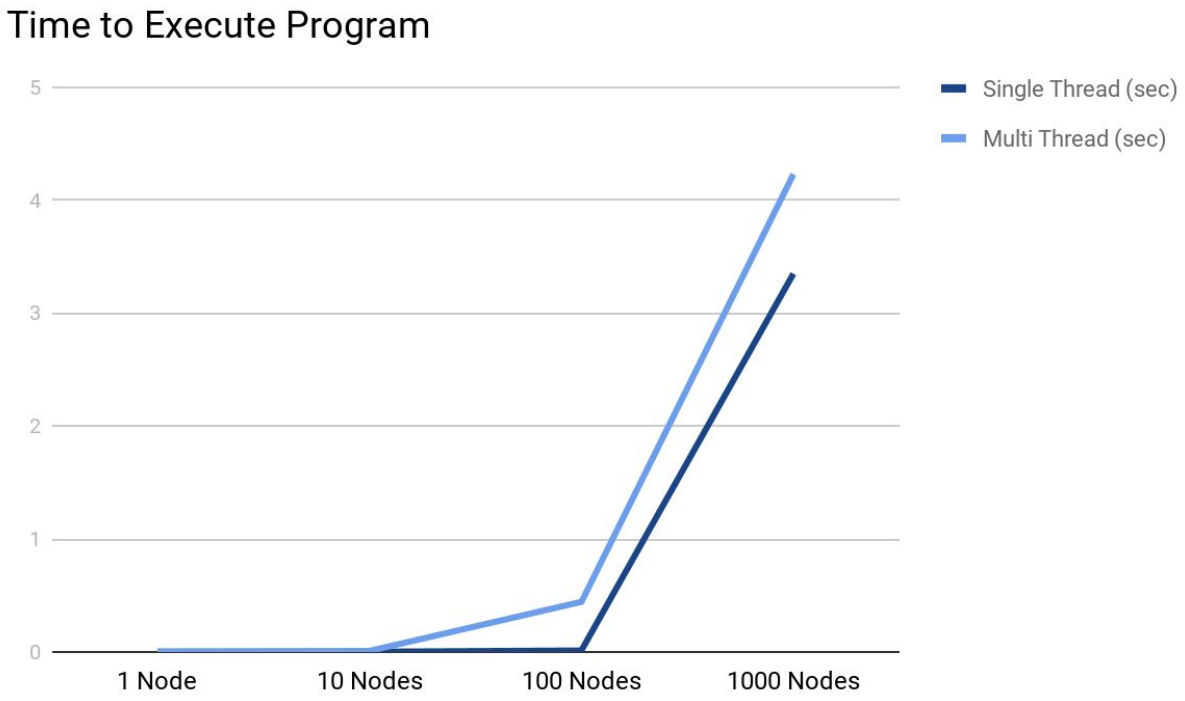
overheard by calling the time.h library but also halts the system adding an extra microsecond per created thread. Although this did fix the majority of issues regarding the data structure being overwritten by another thread before the current has time to access, the data was still at the mercy of scheduling. Poor scheduling could lead to a thread not being fully created as a sequential thread is created and iterates the data structure.

Time Complexity

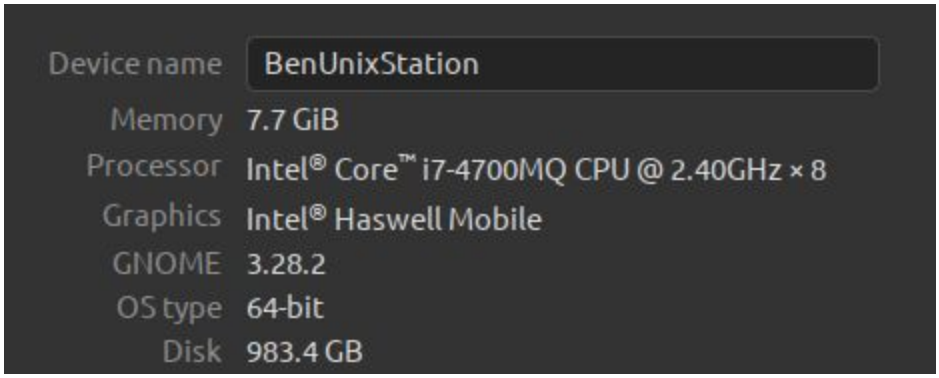
This FW implementation is O(n^3).

Extra overhead for every thread created greater than cpu core/thread.

	Single Thread (sec)	Multi Thread (sec)
1 Node	0.002	0.003
10 Nodes	0.002	0.009
100 Nodes	0.013	0.444
1000 Nodes	3.35	4.231

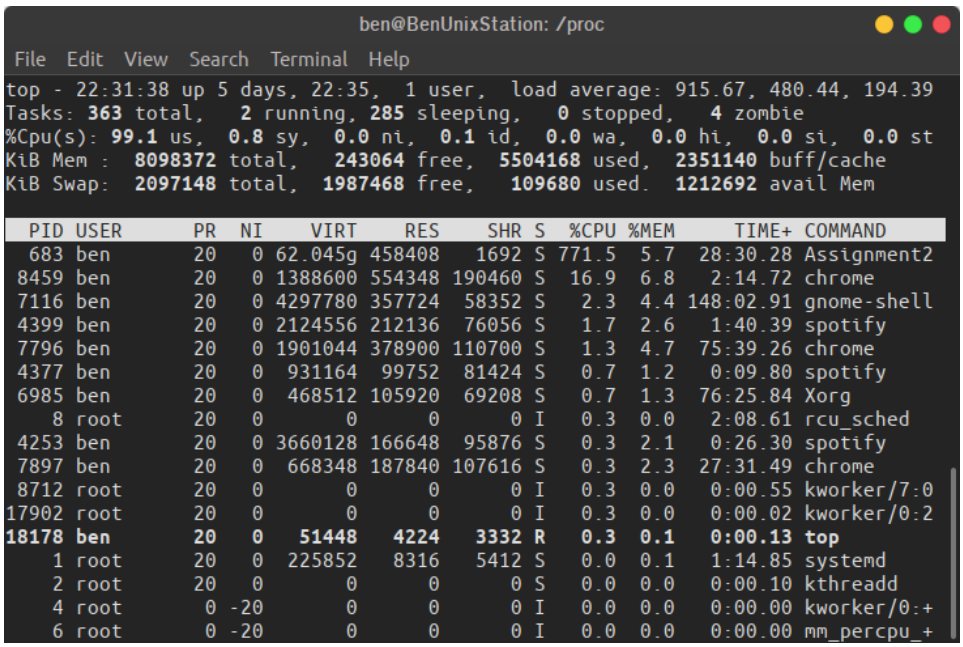


This data was collected as the average of 5 trials using the time command in Ubuntu. Time is calculated as summation of user and system time. The machine tested on using Ubuntu 18.04 has the following specifications.



It is clear that when run in the multithreaded program, calculations take longer than in the single threaded implementation. This can be attributed to the fact that the multithreaded implementation generated *n*^2 threads. On a quad core i7 processor, the maximum threads that can be handled at a time is 8, or two per core, thus when *n* nodes enters the thousands, the quantity of threads overwhelms the CPU.

Testing with higher numbers of nodes caused significant system slowdown. Attempting 10 thousand nodes ran for two hours without completion, and testing with 100 thousand nodes caused the test machine's CPU usage to hit 771.5% before crashing the entire machine.



CUDA Attempt

As we were creating too many threads for the CPU to handle in a timely fashion, we considered the idea of using NVIDIA's CUDA library and compiler for C to pass off the computations to the GPU, where the exponentially greater number of cores would be able to process the data significantly faster.

However, CUDA does not seem to integrate well with 2D arrays when allocating memory on the GPU. We attempted to convert the 2d array to a 1d array and then pass it off to the GPU, but with little success. This attempt was abandoned due to lack of time, but would be worth investigating further in future.

Conclusion

Creating more threads than your CPU can run at once is more detrimental to the system then pushing all the data through a single thread. Creating a thread for every iteration of 'i' to handle a line in the matrix will require more system resources as the matrix grows in size. Locks in this case are optimal as all those threads will be accessing different data in the structure at the same time. Not using a lock will cause threads to access the structure at the same time, leading the conflicts in data usage (ie thread1 not having access to the correct data while thread5 is accessing the structure). The ideal fix in this case would be instead of creating a thread for every iteration of 'i', we instead create a limited amount of threads (equal to system core amount) that will take an equal share of the work load from 'i' then join the threads. To optimize further, the use of a thread pool would let us save on system resources as they're pre-instantiated and remain idle unless called on to perform a task.

References

Synchronization between posix threads - https://www.youtube.com/watch?v=GXXE42bkqQk
This was a great troubleshooting resource

User and System time
https://stackoverflow.com/questions/556405/what-do-real-user-and-sys-mean-in-the-output-of-time1

2D arrays in CUDA
https://stackoverflow.com/questions/5029920/how-to-use-2d-arrays-in-cuda

Passing Structs to threads
https://stackoverflow.com/questions/8223742/how-to-pass-multiple-parameters-to-a-thread-in-c

Read-Write locks in C
https://docs.oracle.com/cd/E19455-01/806-5257/6je9h032u/index.html

Relationship between threads and cores
https://askubuntu.com/questions/668538/cores-vs-threads-how-many-threads-should-i-run-on-this-machine

Pthread_create manual
http://man7.org/linux/man-pages/man3/pthread_create.3.html

Segmentation handling
http://pubs.opengroup.org/onlinepubs/009604499/functions/sigaction.html