CMSC 23010: Homework 1 Final Writeup and Theory

Will Thomas

October 12, 2022

Introduction

The goal of the following documentation is to discuss design changes, implementation challenges, correctness outcomes, and performance outcomes for a program that computes the Floyd-Warshall algorithm using explicit programming.

Design

Shared Functions

time handler

Basically, given elapsed time, a method of computation, a thread count, and an input file, this function writes to that file stating how long computation took. It also specifies if the computation was serial or concurrent, and if it was concurrent how many threads were used.

Input

My input function takes a file name as well as a pointer to a vertex magnitude variable that I declare in main. The matrix is read from a textfile. The very first line of every text file is the vertex magnitude, so once this is read the out-parameter pointer to the vertex magnitude variable is set to the correct value. From there, the function dynamically allocates a $|V|$ by $|V|$ matrix. This plan differs from my design documentation. Initially I wanted to allocate and free within main, but I was unable to do so because I did not know the vertex magnitude of a graph until I open the file. From there, the function reads one row of the textfile and then inputs the same row into the matrix. It repeats this until the entire textfile has been traversed. The matrix is returned.

Conversion

This function simply takes in a completed matrix, the vertex magnitude, and our value for infinity ($1000|V|$), and replaces every 0 value that is not on the diagonal with our infinity value.

Output

The output function takes a file name, the matrix holding the minimum distances, the vertex magnitude, and the constant infinity value. The output file is opened, and then each row of the matrix is read, converted to a string, and written into the file. Every instance of ($1000|V|$) is written as "inf".

Serial Functions

Serial

The serial function takes as input the matrix and the vertex magnitude, and very simply traverses the matrix and computes the recurrence relation at every cell for every value of k.

Concurrent Functions

Concurrent

The concurrent function takes as input a void pointer that, when called, is a pointer to the thread data struct. This struct contains the thread id, the memoization matrix, the vertex magnitude, the thread count (which is capped by the vertex magnitude so as not to incur excess overhead), and a pointer to a pthread barrier object.

There is a pthread barrier after the outermost loop but before the second loop. This way, no threads move on to the next value of k before all of the other ones. From there, the function works the same as the serial function, apart from the conditional statements that allocate rows to each thread. That is, each thread is allocated $\frac{v}{t}$ (rounded down). The last thread is allocated the difference between the number of rows and the amount of other threads multiplied by this base allocation. This is so that if the number of rows is not evenly divisible by the number of threads then the last thread will pick up the remainder.

This design follows that in the original design document, except now the barrier is at the beginning of each iteration of k as opposed to the end. Also, the design document did not define when each thread will be created and subsequently joined. I ended up creating and ending them both in main.

Pthread Barrier Functions

When trying to implement a pthread barrier, I found some code online that I used. The link is as follows:

http://byronlai.com/jekyll/update/2015/12/26/barrier.html

Main

The program itself takes the following arguments:

1. An input file
2. An output file
3. A method (either S or C for serial and concurrent, respectively)
4. A threadcount (only read if the previous argument is C)

These arguments are parsed and used where they are needed in main. For the purpose of timing, for serial the clock is started right before the function call and stopped right afterward. For concurrent the clock is started before creating all the threads and stopped after they all finish.

Testing

The testing plan follows directly from the design document. Each test is handled by a python script. test1.py runs the serial version of the program on 9 hand-solvable graphs, and then checks them against the solutions. These hand-solvable graphs hit the edge cases of sparsity, one node, 0 nodes, and full density. The solutions are 'correct' because our invariant, the recurrence relation:

$$M[i][j] = \min(M[i][j], M[i][k] + M[k][j])$$

holds for every cell in the matrix.

Once that passes, there is a python function graph_generate.py that creates 20 graphs each of sizes $N \in \{16, 32, 64, 128, 256, 512, 1024\}$. Each graph has a different sparsity rating, referring to what percentage of the vertices are not empty. As there are 20 graphs per node size, the sparsity ratings increase by 5% for each graph, so we hit the whole array of graph density. From there, we re-run test1.py, but this time we test it with our concurrent function. We test it for the following thread values $T \in \{1, 2, 3, 4, 8\}$. If those pass, we have a python script test_concurrent.py that runs the concurrent function on the random problems generated by the graph_generate.py script and checks the outputs against the serial outputs. It, once again, does this for values $T \in \{1, 2, 3, 4, 8\}$. At this point, we have proved beyond a reasonable doubt the the concurrent method is accurate, so we move on to performance testing.

Performance

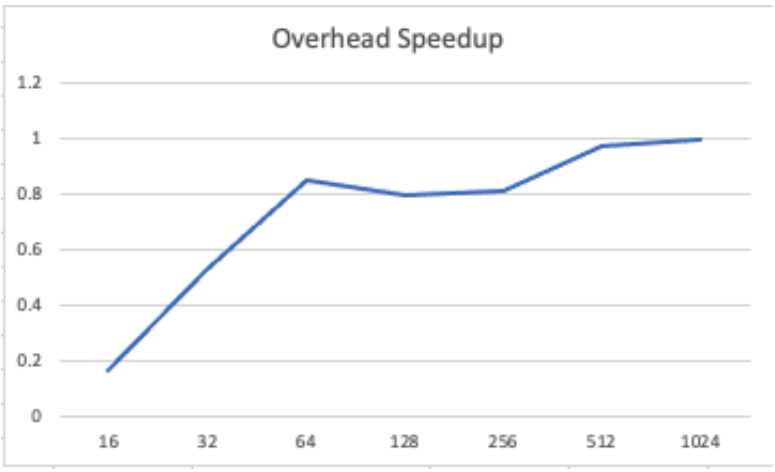
To test performance, we scp'd the following things to the remote machine: a directory to contain the time information, a directory of inputs (one graph of each specified size, all at sparsity 75%), a directory of sinks so that the program would run, a Makefile, the c program itself, and a python script perform_script.py. The python script has two functions within it, once called overhead and one called speedup. The overhead function runs the serial version of the program on all inputs as well as the 1-thread concurrent version. the speedup version runs on all the input graphs as well but once for each threadcount specified in the assignment. All the timing data is written in the data_sink files. The stopwatch code is also passed to the remote machine.

Parallel Overhead:

Table:

Nodes	Serial	1-Thread Concurrent	Ratio
16	0.038	0.227	0.16740088
32	0.256	0.489	0.52351738
64	1.846	2.177	0.8479559
128	6.803	8.553	0.79539343
256	36.528	44.983002	0.81204007
512	269.84201	278.018005	0.97059185
1024	2094.58398	2102.917969	0.99603694

Speedup Plot:



Parallel Speedup:

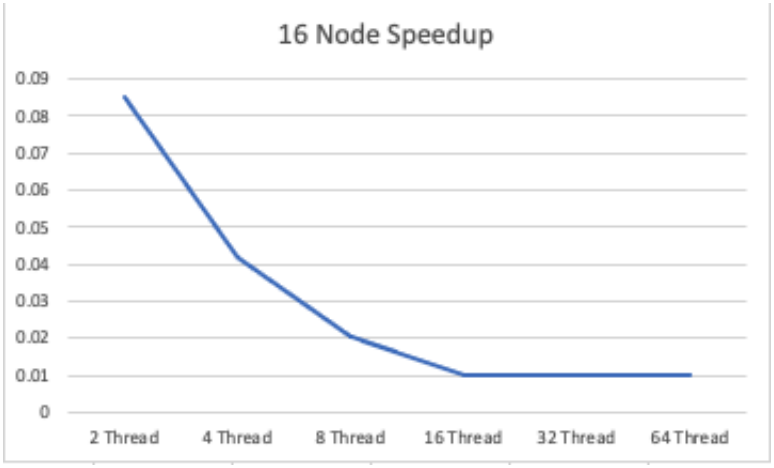
Raw Value Table:

Nodes	2 Thread	4 Thread	8 Thread	16 Thread	32 Thread	64 Thread
16	0.446	0.906	1.868	3.894	3.72	3.87
32	0.767	1.402	2.938	6.214	12.472	10.833
64	2.159	2.831	5.375	11.526	21.667	32.938
128	6.347	7.15	9.942	22.589001	31.764999	59.797001
256	32.912998	31.02	24.871	31.472	70.236	122.666
512	214.102005	126.781998	93.445999	105.927002	161.822006	252.729996
1024	1472.73303	885.302979	521.22699	358.700012	628.739014	771.247986

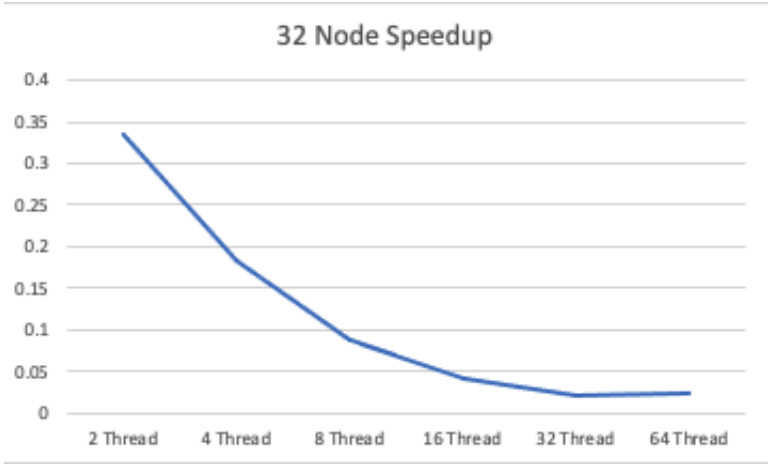
Speedup Table:

Nodes	2 Thread	4 Thread	8 Thread	16 Thread	32 Thread	64 Thread
16	0.08520179	0.0419426	0.02034261	0.0097586	0.01021505	0.00981912
32	0.33376793	0.18259629	0.0871341	0.0411973	0.02052598	0.0236315
64	0.85502547	0.65206641	0.34344186	0.16015964	0.08519869	0.05604469
128	1.07184497	0.95146853	0.68426876	0.30116427	0.21416654	0.11376825
256	1.10983509	1.17756286	1.46869848	1.16065074	0.52007518	0.29778423
512	1.26034322	2.12839373	2.88767858	2.54743366	1.66752358	1.06770868
1024	1.42224282	2.36595158	4.01856393	5.83937527	3.3314045	2.71583721

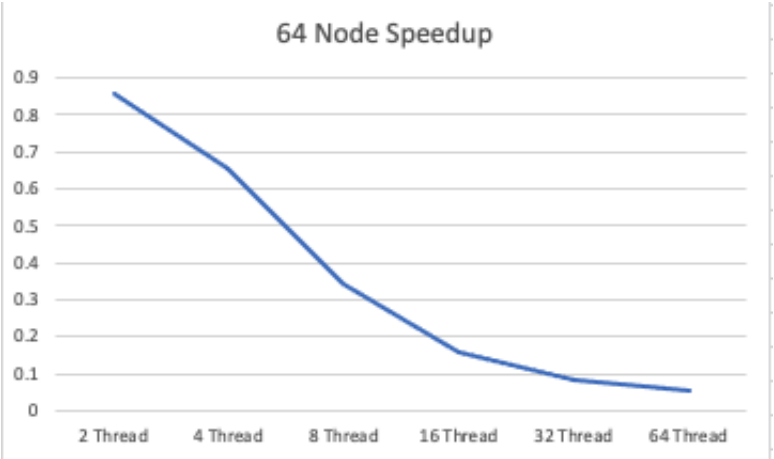
16 Node Speedup:



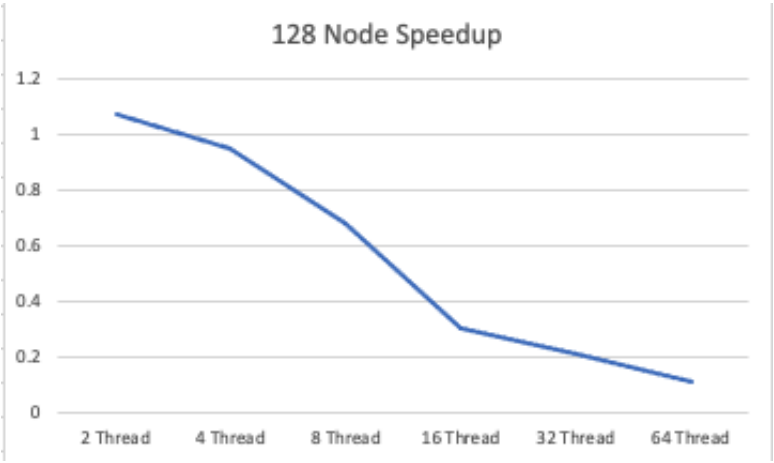
32 Node Speedup:



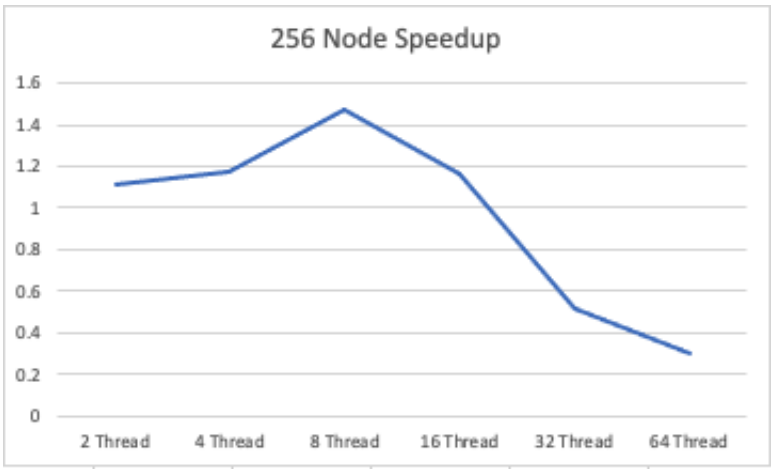
64 Node Speedup:



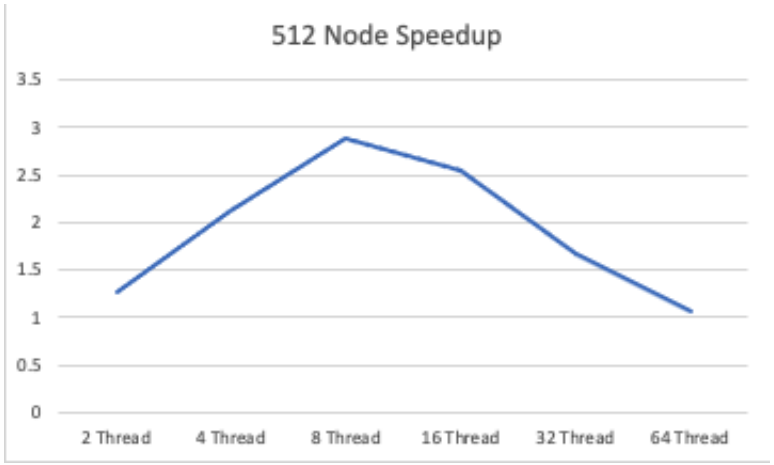
128 Node Speedup:



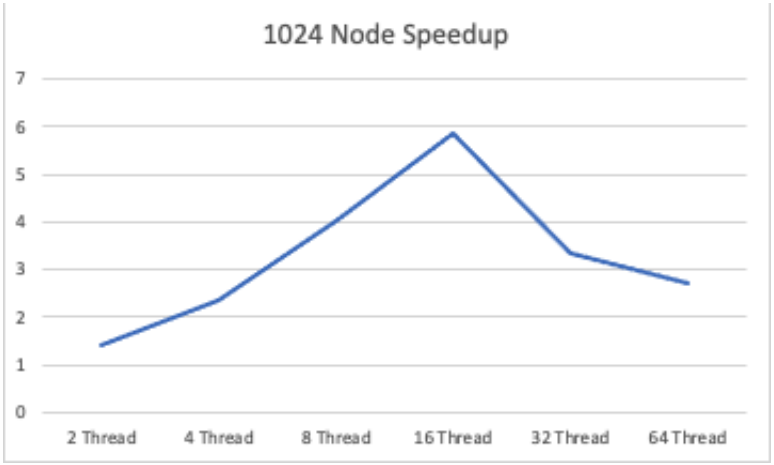
256 Node Speedup:



512 Node Speedup:



1024 Node Speedup:



Analysis

1. **Parallel Overhead**:

   My hypothesis was as such:

   When independently running Serial and Parallel with $T = 1$ and $N \in \{16, 32, 64, 128, 256, 512, 1024\}$, we will see, for each value of N, that Parallel is a constant amount of time slower than Serial and that the ratio of Serial to Parallel will approach 1 as N grows larger (the constant overhead to create and join 1 thread is getting smaller in reference to the overall runtime as the inputs get larger).

   Looking at the data, we can see that my hypothesis was semi-correct. Clearly, the ratio approaches 1 as N grows large. However, the overhead only appears to be a constant for $N = 16, 32, 64, 128$. After that point, there is some odd behavior where the overhead grows at a faster rate. By $N = 1024$, the ratio is 1.

2. **Parallel Speedup**

   My original hypothesis is as follows:

   I hypothesized that for small values of N such as 16 or 32 then the speedup will be very small and that it will plateau once the number of threads becomes larger than the number of nodes. However, I believe that the speedup will become considerably more pronounced on larger input sizes. For the largest example of $N = 1024$, I suspect that the speedup will resemble a straight line.

   This hypothesis is, also, semi-correct. Obviously, in the case of there being more threads than vertices the speedup will plateau. However, I did not account for how large the number of vertices has to be to receive any real benefit from parallelism. Even then, the benefit is not monotonically increasing with the number of threads. We only see real speed up benefits with 256 Nodes, 512 Nodes, and 1024 nodes. In those cases, the speedup maxes out at 8 threads, 8 threads, and 16 threads, respectively. After the maximal point, the process actually takes even more time, often computing in more time than the 2 thread computation, as seen in 256 and 512 Nodes. As such, my final hypothesis regarding $N = 1024$ resembling a straight line is only true for 2 through 16 threads, after which point the performance drops drastically. There seems to be some sort of critical mass, or an optimal number of threads given a certain amount of nodes, that is not an intuitive maxing of the number of threads given the number of nodes.

Theory

- Exercise 2:

  1. This is a liveness property. The good thing that is guaranteed to happen is that the patron will be served.
  2. This is a liveness property. The good thing that is gravity is assured.
  3. This is a safety property. The bad thing that is avoided is a deadlock.
  4. This is a liveness property. The good thing is that the message will arrive in a knowable amount of time.
  5. This is a safety property. There will never be an interrupt with no corresponding message.
  6. This is a safety property. In this example, the cost of living decreasing is bad, and will never happen.
  7. This is a liveness property. The good things of death and taxes are assured to happen.
  8. This is a liveness property. Harvard men being able to be told is a good thing that is assured to be true.

- Exercise 3:

  In this protocol, there is a can blocking Bob's door. The can is tied to a string, the other end of which is held by Alice. When the can is blocking the door, Alice can let her pets out to eat. Once they have finished their food, she can bring them back inside and then pull the string, unblocking (and opening) the door. Bob, seeing that the door is open, will go and refill the food, and then re-block the door on his way back inside.

- Exercise 4:

  (a) Winning strategy if initial state of the switch is off:
  Assuming that the prisoners can see the switch room, the strategy here would

  (b) Winning strategy if initial state of the switch is unknown:

- Exercise 5:

- Exercise 7:

  Consider Amdahl's law: $S_n = \frac{1}{(1-p)\frac{p}{n}}$, as well as Amdahl's law for the case of $S_2 = \frac{1}{(1-p)\frac{p}{2}}$.

  From here, we can say that:

  $$\frac{1}{S_2} = (1-p)(\frac{p}{2}) \implies (1-p) = \frac{2}{pS_2}.$$

  Plugging in for $(1-p)$,

  $$S_n = \frac{1}{(\frac{2}{pS_2})(\frac{p}{n})} \implies S_n = \frac{S_2 n}{2}.$$

  As such, we have our formula for S_n.

- Exercise 8:

  Amdahl's Law provides upper bounds on the speedup of a program via parallelization. If a program has been parallelized to its fullest extent, the less powerful processor makes sense, but if the program has a large enough parallel component such that it has not been fully tapped, the more powerful processor makes sense.

- Exercise 11:
- Exercise 14:
- Exercise 15:
- Exercise 16: