Writeup Document

Alex Miller

April 13, 2021

NEED TO INCLUDE BIT ABOUT no NEGATIVE edge weights

Changes

Changes to the design

My final design ended up including three distinct modules as opposed to the previously proposed two; these included a graph generation module (graph.py), a Floyd-Warshall module (graph.c, floyd.c), as well as a testing module (fw_serial.c, fw_parallel.c, test_script.sh). Describing these pieces in the order in which they are salient:

- graph.py describes a simple python script that randomly generates an adjacency matrix of specified length and stores it in an appropriately named text file. These files are used to test the performance of the functions described in floyd.c. I added this module in order to cut down on the time that I would need to spend making example graphs.
- floyd.c describes serial and parallelized implementations of Floyd-Warshall. We will discuss the changes to this module later. For now just keep in mind that the serial implementation is called fw_serial and the parallel implementation fw_parallel. graph.c describes the I/O methods by which preliminary adjacency matrices are loaded into memory and results are written to text files.
- fw_serial.c describes a simple program that loads an adjacency matrix from a text file, performs an operation of fw_serial, and writes the result to a text file (the text file name describes the performance of the operation). fw_parallel.c is similar, but utilizes fw_parallel (using a specified number of threads) instead. test_script.sh performs all necessary testing by running the following:
  - For $n \in$ [16, 32, 64, 128, 256, 512, 1024]:
    - ./fw_serial n.txt 1
    - For $t \in$ [2, 4, 8, 16, 32, 64]:
      - ./fw_parallel n.txt t

  I chose to make this change to my testing structure because my original design tested fw_serial and fw_parallel in the same file, which I thought would account for unnecessarily long testing periods; there is no need to retest my serial version every time I want to test my parallel version. Furthermore, putting everything in a bash file made testing easy to do with slurm.The basic logic of my test plan did not change; I verified correctness of each module before moving on the next and checked the results of fw_parallel using those of fw_serial

As per the specific modifications I made to my design of floyd.c: My conception of fw_serial remained unchanged throughout this process. However, my design for fw_parallel changed a lot. Our discussion last Tuesday made me realize that by statically partitioning the matrix into equal sized blocks, I could utilize all the threads that I am given access to in a given experiment, rather than being bounded by the number of vertices in a given graph. More correctly, I should say that the new bound of how many threads can be employed for a given graph is the number of indices in its adjacency matrix (the number of vertices squared) rather than the number of vertices in the graph. However, this increase in our ability to utilize thread brought certain drawbacks:

- Since such blocks aren't guaranteed to be squares, this method would require more overhead in generating the geometry of the partitioning of the matrix. However, the dimensions themselves could be stored in the graph_t data type I designed before, allowing all threads to know the area of the blocks they are being asked to compute.

```
typedef struct graph {
    int num_v; //NUmber of vertices in the graph
    int b_i; //A block's i dimension
    int b_j; //A block's j dimension
    int M[MAX_V][MAX_V]; //Easiest to implement
} graph_t;
```

- Threads would have to know one more information in order to operate; instead of just needing to know what row index they were working on, they would need to know the corner indices from which to begin computing their assigned blocks.

- If I wanted to synchronize the operation of these threads I would need to pass them all a pointer to a thread barrier

However, by allowing for the use of more threads, it was my thought that I could increase performance, at the very least for larger adjacency matrices where the larger amount of overhead would pay off in improved run times. I maintain the same invariants as before; threads still wait for a the matrix to be computed for a given value of k before computing the matrix for $k+1$; moreover, the block structure still reflects the independence of the order in which indices are computed. AS stated in my previous document: As an invariant, we know that we will always have generated the shortest path lengths of all (i, j) pairs for a value of k before we generate such lengths for a value of $k + 1$, maintaining data dependencies within the algorithm.

Changes in hypotheses

I did not implement my past design before attempting my current, so I cannot speak to improvements in performance of one in relation to the other. I maintain the general same hypotheses as I did before, with the following caveats:

- I do not expect to see similar cache performance between my serial and parallel implementations of Floyd-Warshall; considering that we are reading my blocks as opposed to rows, there is no reason to think that fw_serial's cache will look like that of fw_parallel

- I expect to see minimal speedup, or even slowdown for small graphs with few vertices; the increased overhead of using as many threads as possible should detract from any gains in performance we may see in graphs with 16, 32, or 64 vertices

- For large enough graphs however, I expect to see speedup that is linear in proportion to the amount of threads used.

Results

Both fw_serial and fw_parallel print their results to a file named "hw1/res/¡n¿,¡p¿,¡time¿" where:

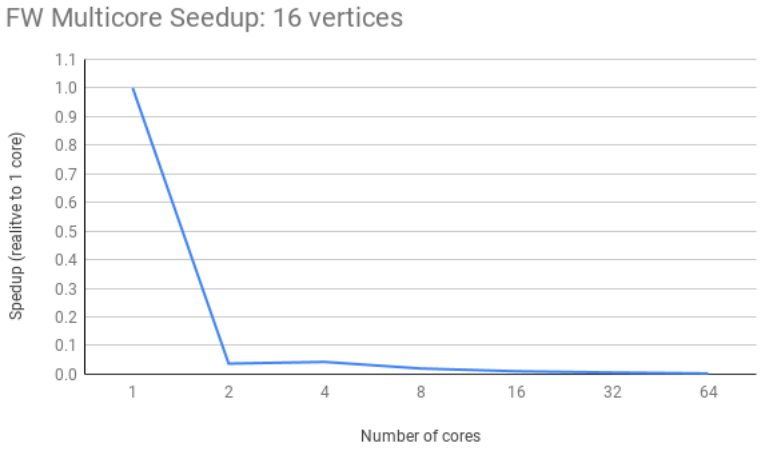
- ¡n¿ denotes the number of vertices in the graph for which a result was computed

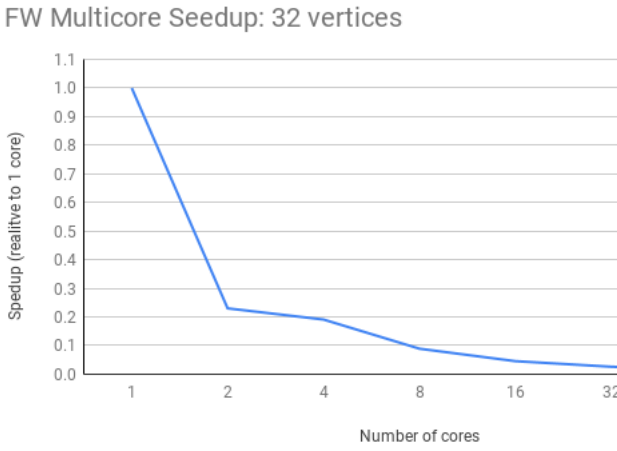
- ¡p¿ denotes the number of threads used in computing that result

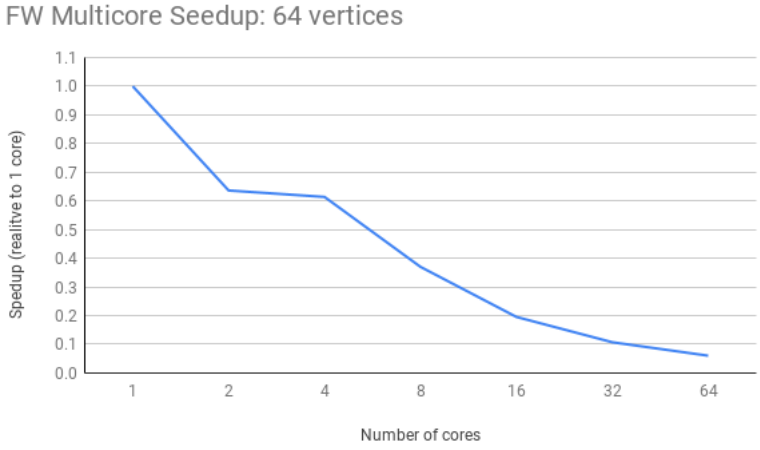
- ¡time¿ the amount of time it took to perform such an operation of fw_serial or fw_parallel (depending on which implementation is being tested)

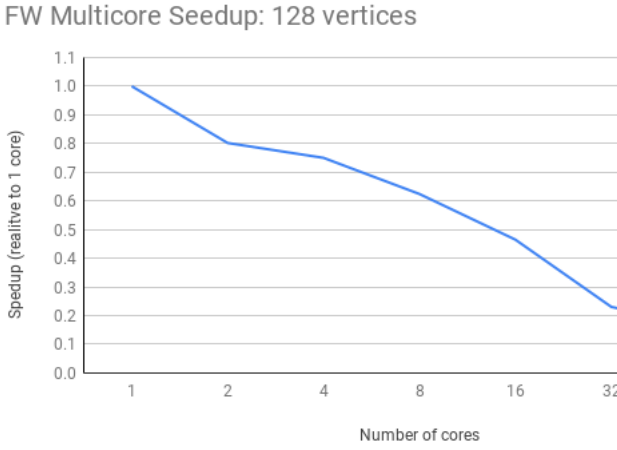
This raw performance data is reflected in a file called "hw1/results.csv". It can also be found (in a nice format with speedup data) in this Google sheet or "hw1/hw1_result_data.pdf" The combined data set can be found in a file called "hw1/results.csv."

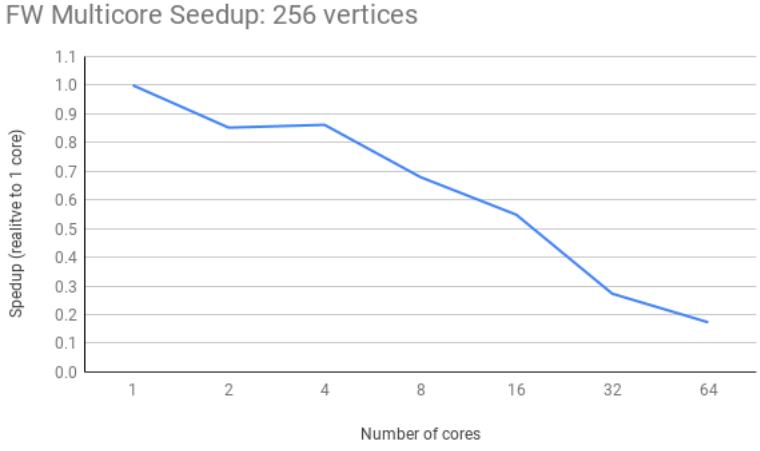
The following graphs represent this data as well:

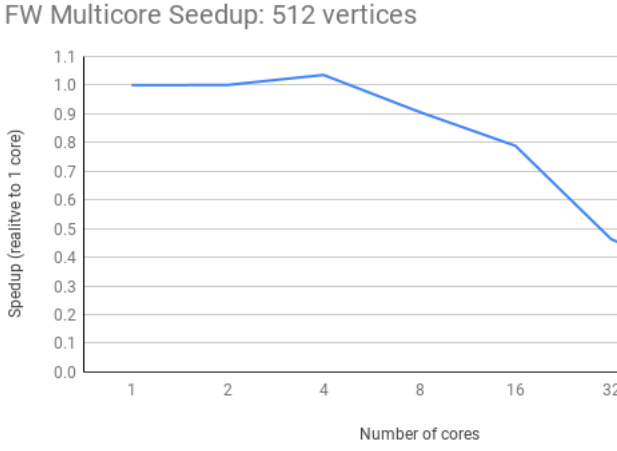


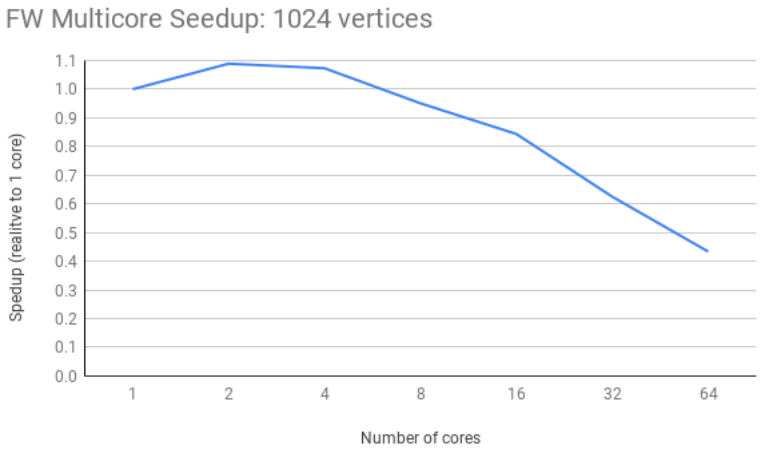












Analysis

As the data shows, my parallel implementation of Floyd-Warshall failed to deliver noticeable improvement over my serial implementation. If anything, the extra overhead of parallelizing Floyd-Warshall seemed to heavily outweigh any potential benefit from having multiple concurrent processes computing results for distinct blocks of vertices.
That being said, it is notable that for larger graphs with more vertices, my implementation of fw_parallel did deliver slight speedup. However, this observed speedup did not confirm my hypothesis that for larger graphs, speedup would appear linearly proportional to the number of threads utilized. Rather, for the trials testing performance on graphs containing 512 and 1024 vertices, maximum possible speedup (given my implementation) was achieved using 2 or 4 threads, with drop offs in performances being observed for larger thread counts.
Overall, I seem supported in my hypothesis that observed speedup would increase with the size of the input space. However, it also seems that I require a new model by which to predict speedup since I did not observe linear speedup. This lack of support for my hypothesis indicates to me that the overhead of managing and utilizing as many threads as possible does not pay off in increased performance necessarily. I do wonder, however, whether or not I would observe something closer to linear speedup if I were to test a large enough input space; though I did not observe linear speedup, the curves of my data display a tendency to flatten for larger input space. Perhaps I might observe something close to linear speedup proportional to the number of threads used for large enough graphs

Theory Questions

Provide solutions to questions 2, 3, 4, 5, 7, 8, 11, 14, 15, 16.

Problem 2

1. Liveliness; the good thing that eventually happens is that all patrons will be served
2. Safety; the bad thing that never happens is that things float away into the air
3. Safety; the bad thing that never happens is that the program gets stuck
4. Liveliness; the good thing that eventually happens is that the user is informed of the event
5. Liveliness; the good thing that eventually happens is that the user is informed of the event
6. Safety; the bad thing that never happens is that the cost of living decreases
7. Liveliness; the good thing that eventually happens is that I will die and have to pay taxes

8. Safety; the bad thing that never happens is that I never get snuck up on by a Harvard man.

Problem 3

Alice and Bob set cans up on each other's window sills; each one's can has a string attached to it that the other can use to knock it down. Alice does the following:

1. She waits until the can on her window sill is down.
2. She releases the pets.
3. When the pets return, Alice checks whether they finished the food. If so, she resets the can on her windowsill and then knocks down the can on Bob's window sill.

Bob does the following:

1. He waits until the can on his window sill is down.
2. He puts food in the yard.
3. He resets the can on his window sill and knocks down the can on Alice's window sill.

Problem 4

The following assume $P \geq 2$ In the case that we know the initial state of the switch is off:

- We designate one of the P prisoners to be the *counter*; the counter maintains an internal count c. Initially, $c = 0$.
- When the counter enters the switch room:
  - If the switch is down, they do nothing and leave
  - If the switch is up, the counter increments c
- When $c = P - 1$, the counter may declare that all P prisoners have visited the switch room at least once
- For all other prisoners; when they enter the switch room:
  - If the switch is down, and they have never flipped the switch before, they should switch it to the "on" position. Otherwise, they should do nothing and leave.
  - If the switch is up, they should do nothing and leave.

In the case that we don't know the initial state of the switch:

- We designate one of the P prisoners to be the *counter*; the counter maintains an internal count c. Initially, $c = 0$.
- When the counter enters the switch room:
  - If the switch is down, they do nothing and leave
  - If the switch is up, the counter increments c
- When $c = (2 * P) - 1$, the counter may declare that all P prisoners have visited the switch room at least once
- For all other prisoners; when they enter the switch room:
  - If the switch is down, and they have not flipped the switch more than once, they should switch it to the "on" position. Otherwise, they should do nothing and leave.
  - If the switch is up, they should do nothing and leave.

Problem 5

Starting from the back of the line, the first prisoner does the following:

- If they see an odd number of red hats in front of them, they call out "red!"
- Otherwise, they call out "blue!"

The next prisoner does the following:

- If the previous prisoner called out "red!":
  - If there are an odd number of red hats in front of them, they know they must not be wearing a red hat. They should call out "blue"
  - Otherwise, they know they are wearing a red hat. They should call out "red"
- If the previous prisoner called out "blue!":
  - If there are an odd number of red hats in front of them, they know they must be wearing a red hat. They should call out "red"
  - Otherwise, they know they are wearing a blue hat. They should call out "blue"

Likewise, every prisoner following the first can know what hat they are wearing by:

1. Knowing whether or not there were initially an even number of red hats or not by remembering the first prisoner's answer
2. Counting how many red and blue hats were claimed by each prisoner after the first
3. Deducing the color of their hat from how many red and blue hats have been claimed (excluding the first prisoner), the initial eveness in the number of red hats, as well as the evenness in the number of red and blue hats in front of them.

Problem 7

$S_n = \frac{1}{1-p+\frac{p}{n}}$
$S_2 = \frac{1}{1-p+\frac{p}{2}}$
$S_2 = \frac{1}{1-\frac{p}{2}}$
$1 - \frac{p}{2} = \frac{1}{S_2}$
$p = 2 - \frac{2}{S_2}$
$S_n = \frac{1}{1-(2-\frac{2}{S_2})+\frac{(2-\frac{2}{S_2})}{n}}$

Problem 8

- Let $proc - a$ denote a uni-processor that executes five instructions per second
- Let $proc - b$ denote a ten processor multi-processor, each core of which executes 1 instruction per second
- Let P be a program we are considering to run on either $proc - a$ or $proc - b$. The fraction of P that is parallelizable is p.
- Let A be the time it takes for $proc - a$ to run P
- Let $B_i, 0 \leq i \leq 10$ be the time it takes for $proc - b$ to run P using i cores
- $5 * A = B_1$
- $B_{10} = \frac{1}{1-p+\frac{p}{10}} = \frac{1}{1-\frac{9p}{10}}$
- Assume that P runs faster on $proc - b$ than $proc - a$ while using all ten cores:

- That must mean that B_{10} achieves a speedup ratio of 5 or more in relation to B_1
- Therefore $B_{10} \geq 5$
- $\frac{1}{1-\frac{9p}{10}} \geq 5$
- $\frac{9p}{10} \geq 1 - \frac{1}{5}$
- $\frac{9p}{10} \geq \frac{4}{5}$
- $9p \geq 8$
- $p \geq \frac{8}{9}$

- Therefore, if P runs better on $proc - b$ than $proc - a$, it must be that $p \geq \frac{8}{9}$.
- Therefore, if $p > \frac{8}{9}$, we should buy $proc - b$, and $proc - a$ if otherwise.

Problem 11

The protocol satisfies mutual exclusion:

- Assume that it does not and consider the steps of two threads, A and B
- Therefore the following relationships must hold:
  1. $write_A(turn = A) \rightarrow write_B(turn = B) \rightarrow CS_A$
  2. $write_B(turn = B) \rightarrow write_A(turn = A) \rightarrow CS_B$
- Consider these relationships assuming a specified ordering by which the threads enter the critical section. WLOG, that B is the last thread to write a value to $turn$ before entering the critical section; therefore
  1. $write_A(turn = A) \leftarrow write_B(turn = B)$

  If the lock is not mutually exclusive, as we assume it is, this means that B must enter it's critical section before A leaves its. However, B cannot enter its critical section until A leaves its critical section and $write_A(turn = A)$. Therefore B waits for A to leave the CS_A before it enters CS_B. Therefore for some $i, j, CS_A^i \rightarrow CS_B^j$
- By contradiction, the lock is mutually exclusive.

The protocol is not deadlock free. Consider the following execution: $write_A(turn = A) \rightarrow read_A(busy == false) \rightarrow write_A(busy = true) \rightarrow write_B(turn = B)$ After this point, $busy$ never stops being true, so B never stops setting $turn$ to B, so A never exits its outer loop and never enters its critical section Because the protocol suffers from deadlock, it also suffers from starvation.

Problem 14

Assuming that $n > l$, we can get the desired behavior out of the existing filter lock design by reducing that sizes of the $level$ and $victim$ arrays from n to $n - (l - a)$ and reflecting that change in our loop structure so as not not iterate outside the bounds of our (now smaller) data structures. This should be enough.

Problem 15

The scientists are wrong in their claim.

1. Consider the execution of two threads, A and B, acquiring the FastPath lock at the same time:
   - $write_A(x = A) \rightarrow read_A(y == -1) \rightarrow write_A(y = A) \rightarrow read_A(x == B)$
   - $write_B(x = B) \rightarrow read_B(y == -1) \rightarrow write_B(y = B) \rightarrow read_B(x == A)$
2. WLOG, assume that B acquires the lock shortly after A and that $write_A(x = A) \rightarrow write_B(x = B)$
3. Assuming that $y = -1$ initially, it can happen that A and B both read $y == -1$ before either have the chance to $write_{AorB}(y = AorB)$

4. Therefore it is possible for neither thread to wait in the while loop in line 8 of the declaration of FastPath.

5. Because $write_A(x = A) \rightarrow write_B(x = B)$, $write_A(x = A) \rightarrow write_B(x = B) \rightarrow read_A(x == B)$; A can proceed to call the base lock(). But because $write_B(x = B) \rightarrow read_B(x == A)$, B never calls the base lock(); instead it just proceeds to CS_B.

6. Therefore both threads can enter the Critical Section at the same time, because one does not even need to call the base lock in order to proceed past the FastPath wrapper lock. This violates any mutually exclusive property the base Lock may have.

7. Therefore it is not the case that our scientists' FastPath wrapper is mutually exclusive and starvation free $\Leftrightarrow$ the base Lock is.

Problem 16

At most one thread gets the value STOP:

- Consider two threads, A and B
- Assume that A returned STOP
- Therefore the ordering $write_A(last = A) \leftarrow read_A(goRight == False) \leftarrow write_A(goRight = True) \leftarrow read_A(last == A)$ occurs
- Assume that B also returns STOP, but after A
- If this is the case then $read_A(last == A) \leftarrow write_B(last = B) \leftarrow read_B(goRight == False) \leftarrow write_B(goRight = True) \leftarrow read_B(last == B)$
- However, $read_B(goRight == False)$ never occurs because $goRight$ is set to true by A and is never set to false again anywhere in the program.
- Therefore B would return RIGHT, and not STOP.
- WLOG, assume that thread B is some thread that comes after some thread A which returns STOP. It can't be the case that B also returns STOP, so at most one thread does.
- **QED**

At most $n - 1$ threads get the value DOWN.

- Consider two cases:
    1. One or more of n threads return RIGHT; the amount of threads that could return DOWN is therefore $n - 1$.
    2. No threads return RIGHT. If this is the case then $\forall i, j \in [n], read_i(goRight == false) \leftarrow write_j(goRight = true)$. This implies that there was some thread k for which $write_k(last = k)$ occurred after all other n threads. Therefore, if no threads return RIGHT, such a thread k will evaluate $read_k(last == k)$ as $true$, in which case it would return STOP
- In either case, at most $n - 1$ threads get the value DOWN
- **QED**

At most $n - 1$ threads get the value RIGHT.

- FSOC, Assume that no thread returns STOP or DOWN
- Therefore $\forall i \in [n], read_i(goRight == true)$
- However, for this to be true, $\exists j \in [n], write_j(goRight = true)$

- Such a thread however would never return RIGHT, as $write_j(goRight = true)$ never proceeds a thread returning RIGHT
- Therefore, for all threads to return RIGHT, some thread must return either STOP or DOWN
- Therefore not every thread can return RIGHT
- Therefore, at most $n - 1$ threads get the value RIGHT
- **QED**