Final Write Up

Changes Made to the Initial Design

Parallel Module

The big change I made here was to have the thread function iterate over k, and have the threads wait on the barrier defined in the thread function.

I had initially thought that I could synchronize all threads from the main thread. Eg. I could keep the iteration on k in the main thread, and then at every k tell the threads to compute their row or else wait on the barrier. However, this was me misunderstanding the threads API. That previous approach would've required me to (from the main thread) put worker threads to sleep once they've completed their job, then re-awake them once all workers have completed their task. This would've required condition variables and signaling, so I found restructuring my code easier.

Now, the thread function itself iterates on k, waiting to next iterate on k until all other threads have reached the barrier as well. (Note: this was in the file fw_parallel_module.c).

Output Module

I changed this module so that the names are more organized for the experiments. Before, I would name the input like "1_sample_input.txt" and output something like "1_sample_output.txt". Now, I've autogenerated th einput so it's "N_16_T_1_input.txt" and outputs "N_16_T_1_parallel_output.txt". This made writing a script to automatically compare outputs easier.

The issue with this was that I could no longer process the test cases I specified out in the design doc that I had manually verified (those test cases have since been removed from the directory). I think this is OK though, because I was able to verify the correctness of the serial version using my hand-verified test cases while using the old input/output format, so I feel confident that switching to the new output module for renaming's sake and better comparison at a large scale still preserves correctness.

Testing

I compared the output of my serial code with my parallel code to verify correctness. To do this at scale, I wrote a script to compare the outputs for a given graph and a given set of processings, eg. If N=16, I compared the output of the serial version or parallel versions with T=1,2,..., 64.

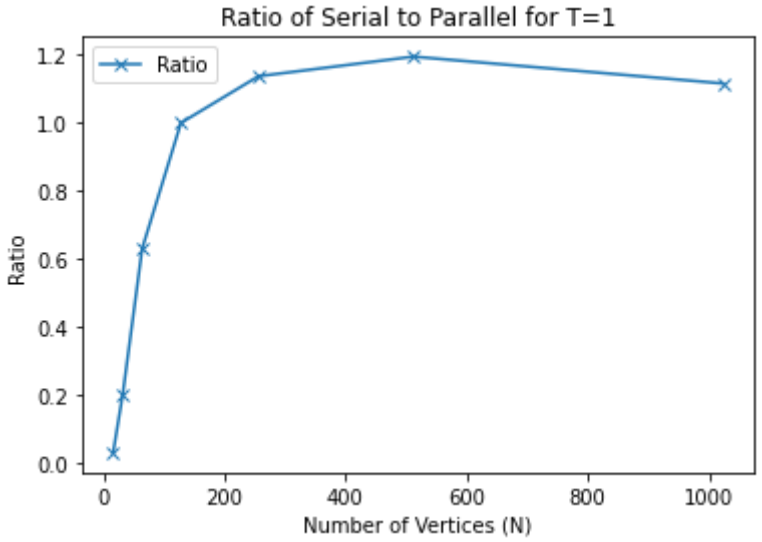
You could have done comparisons for all inputs (i.e. all N=16,32,64,...) graphs, but I chose to do a comparison on only N=16, figuring that that would be sufficient, especially since I am randomly generating the input graphs in test_module.py. There are many ways to suggest correctness, but no way to fully guarantee it, and I figured I felt safe enough with this approach.

Hypotheses

* Note: all the data + graph generating logic is in graphing.ipynb.

Parallel Overhead

N	16	32	64	128	256	512	1024
Serial Time (ms)	0.012	0.074	0.523	3.785	28.662	217.977	1581.96
Parallel Time with T=1 (ms)	0.381	0.363	0.831	3.797	25.288	183.037	1422.43
Ratio	0.0314	0.2038	0.629	0.996	1.1334	1.190	1.112



This graph is somewhat inline with my hypothesis for parallel overhead. As the amount of compute work increases (with N), the fixed cost of spawning a thread pales more and more in comparison as a fraction of the total work, thus parallel times approach serial time. This trend is observed up to N=128.

Past this point, things get weird. I would've expected it to stay steady at around 1.0. Theoretically, the parallel execution should always be slower because it has thread spawn work in addition to the same compute work.

I was initially inclined to attribute this to hardware reasons, something like different runs of the executable on the SLURM machines would lead to different times (because of physics, hardware, etc). This would mean that I was simply unlucky on this run, and that if you had instead run these tests 100 times and averaged them out, you would see the expected graph which asymptotically approaches a ratio of 1.0.

However, I submitted the same run of a parallel and serial execution multiple times (eg. run parallel with T=1 and N=1024 for 5 times), and the variance I got was less than 5% of the average runtime. Eg. I tried running the parallel execution with T=1 for N=1024 10 times, and the results all ranged from 1516 to 1643ms. This suggests that the hardware is consistent, and there is something off about my code that makes the parallel execution lead to less time than the serial execution.

I have looked at my code and honestly do not know why this is the case. I moved my timer to right before and after the executions of both versions of Floyd-Warshall. And I made sure that both serial and parallel executions worked on the same slurm cluster.

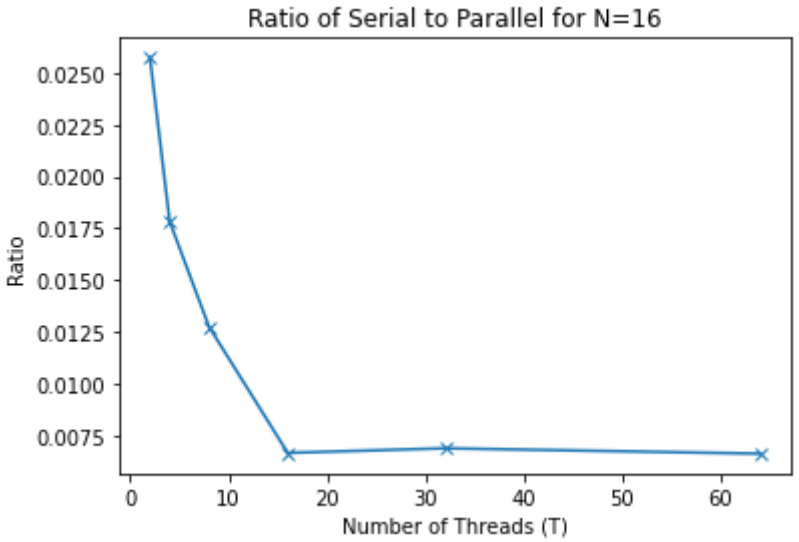
The only reason I can think of is that at some N, here past 128, the cost of spawning a thread is outweighed by the benefit that spawning a separate thread has on executing the program. What's the benefit? The benefit could be that the thread is focusing on the specific computation task. Compared to the serial version of code which requires the main thread to store a lot of variables and memory to do with I/O and setup, the spawned thread already has everything allocated for it already and its local stack only needs to track the minimal number of variables needed to execute its computation. I personally find this argument unconvincing, but after running my code multiple times and getting reliable results, I am trying to rationalize it as best I can.

I will say however that a sanity check which my parallel overhead numbers pass is that the parallel and serial executions are more or less the same time. It would be concerning to see the difference in time multiplicatively increase for larger N, since we expect the executions to only differ by a constant amount C of spawning the threads. The time difference in the table does not clearly double, so that is a slight reassurance about my code.

Parallel Speedup

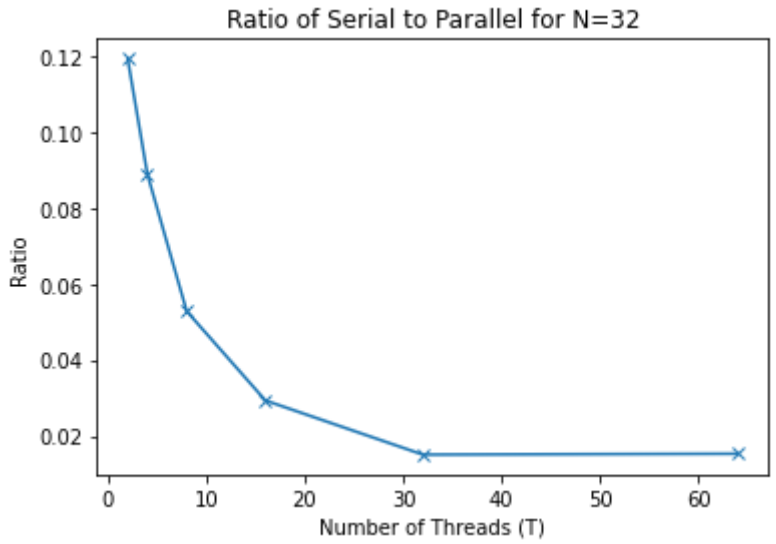
Time in ms of each execution

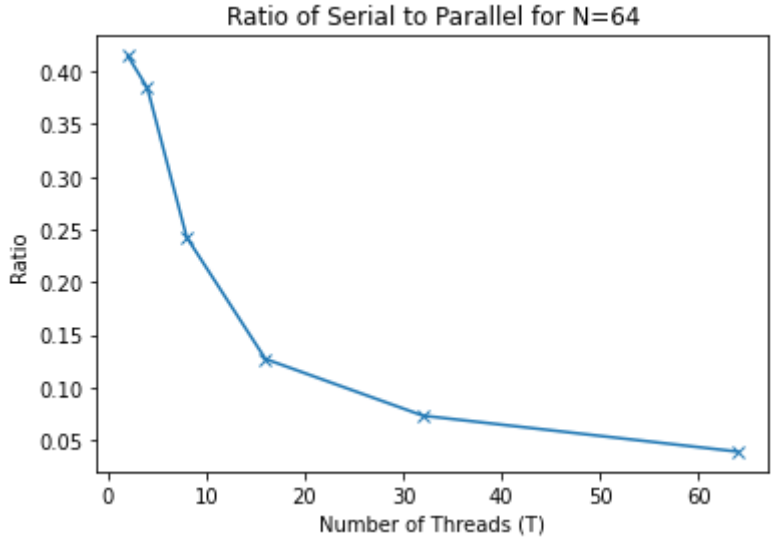
N \ T	2	4	8	16	32	64
16	0.466	0.673	0.942	1.807	1.748	1.817
32	0.62	0.832	1.401	2.524	4.912	4.834
64	1.048	1.319	2.177	4.199	8.324	15.998
128	3.086	2.913	4.169	8.051	15.104	31.223
256	14.878	9.88	10.205	19.646	33.48	62.290
512	97.35	53.69	37.40	53.069	77.764	125.46
1024	714.794	368.00	206.871	258.282	305.96	379.35



The first graph, seeing how the speedup changes for a N=16 graph for thread counts T=2,4,8,16,32,64, lines up with what I expect. For this small of a graph, spawning an additional thread is not worth the thread overhead, even if you are just spawning an additional thread. Thus, the more threads you spawn the slower your code becomes relative to the serial implementation. You can see this for data points T=2,4,8,16.

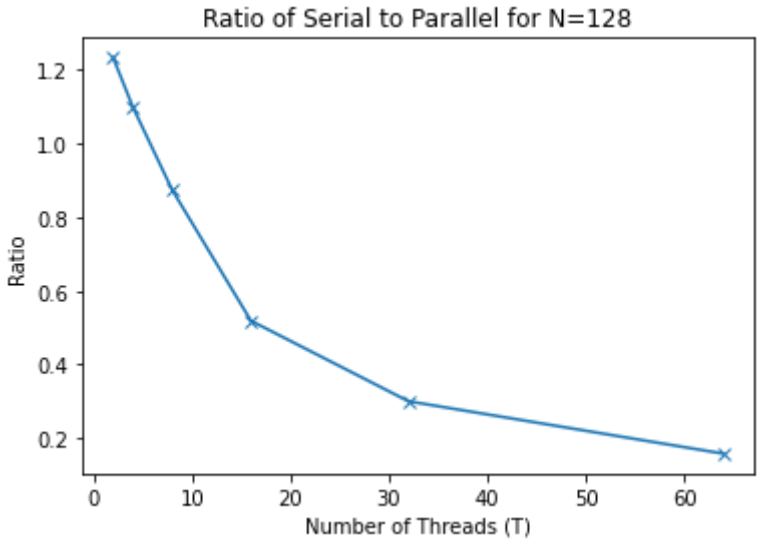
What's interesting is that for T=16,32,64, the program should run identically because I don't spawn additional threads. There is a slight variation, but here I think we really are OK attributing it to a variety of reasons like cache misses / OS scheduling conflicts, the temperature of the machine. Concretely, the numbers were 1.807, 1.748, 1.817ms for T=16,32,64. These differences are 2 or 3% of the total execution time, so this isn't alarming.





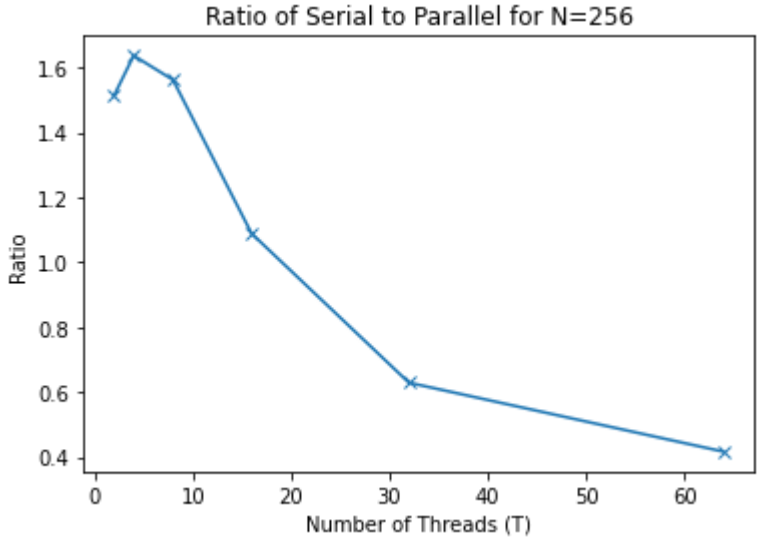
We observe a similar trend as for N=16 in N=32, 64. Here, the overhead of spawning additional threads is not worth it, so additional threads just drag on performance. However, notice how the ratio gets better as N gets bigger. Eg. for T=8 threads spawned, you can see from each group that the ratio goes up from 0.0125, 0.05, 0.25 for N=16,32,64.

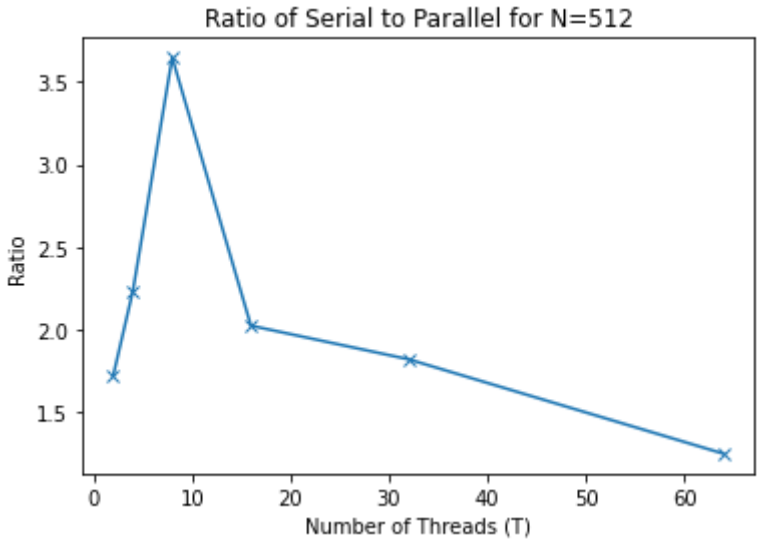
This makes sense, because as N gets bigger, threads have a better chance of being worth their spawn cost.

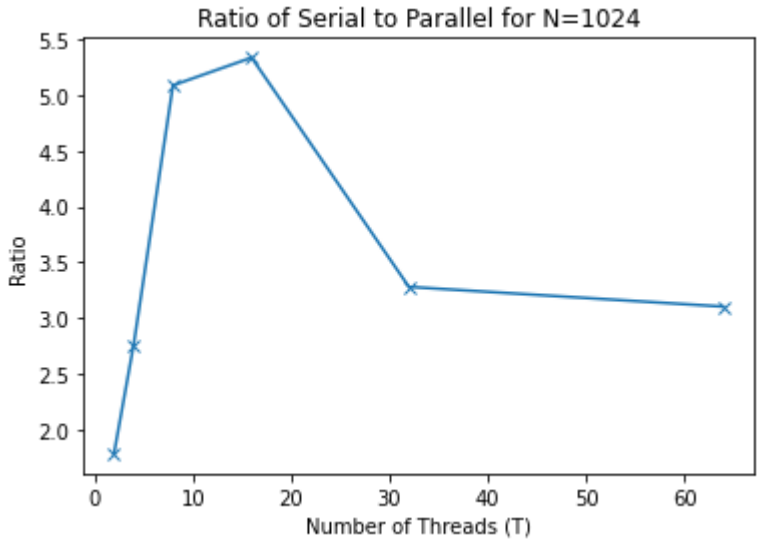


Here, for N=128, we again observe the odd behavior mentioned in the parallel overhead. Parallel with T=1 should never outperform the Serial version, yet here it does. Aside from that, a ratio above 1 for T=2 shouldn't be unexpected, since N is starting to get large to the point that the additional work a thread can do outweighs the cost of spawning that thread. However, that does not hold for T=8 and larger.

We would expect now that the peak should shift to the right as N goes up. This is because as N gets larger, each thread has a better shot of being worth its spawn time. If you spawn too few threads, you lose out on a potential speedup that additional threads could contribute to. Still, if you spawn too much, the overhead is not worth it. Thus, as N gets larger, we should expect the peak to keep moving to the right. And in fact, this is what we see in the following N = 256,512, 1024 graphs.







This later trend of a shifting peak somewhat lines up with my initial hypothesis. My initial hypothesis was that "I do wonder if the overhead cost may outweigh spawning additional threads." This is not predictive, but it says that there are two sides of the coin. Thinking about it more, it is clear that the side favoring spawning additional threads gets more favored as N gets larger, and this is what we observed, so that's good to see. A cool experiment would be to run this on finer increments of N, and to see if the peak smoothly shifts to the right.

HW1 Theory Problems

Don Assamongkol

October 12, 2022

* Discussed with Paul, James, Jack.

1 Ex 2

Safety is nothing bad ever happens. Liveness is something good eventually happens.

1. safety
2. liveness
3. liveness
4. liveness
5. liveness
6. safety
7. liveness
8. safety

2 Ex 3

We make a modification to the producer-consumer fable given on pg 10 so that Bob is still able to get information about if Alice's pets are in the pond (so he can go leave food out without getting eaten), without seeing Alice's tin can himself. We do this by having Alice send a signal to Bob. In particular, both Alice and Bob have 1 can each, and can pull the other person's can down via a string. Start off by having Alice's can down, and Bob's can up. There is food in the pond.

For Alice:

1. Wait until Alice's can is down
2. Release pets to go eat
3. When pets return, Alice checks if they have finished the food. If they have not, do nothing. If they have finished, then Alice keeps her pets inside, puts her can back up and pulls Bob's can down.

For Bob:

1. If Bob's can is up, do nothing.
2. If it is down, then Bob puts food in the yard, then resets his own can up, then pulls the string and knocks Alice's can down.

   In this way, we have encoded the same information as the original fable, only now we are mirroring that information to two cans.

   If Bob's can is up and Alice's can is down, it means that there is food in the pond, and the pets may or may not be there, in which case Bob does not need to go out.

   If Bob's can is down and Alice's can is up, it means the pets are locked inside and there is no food in the pond, which means Bob is safe to put food out.

3 Ex 4

*Note: went to OH to clarify assumptions about this problem.

Assume that the situation is starvation free. I.e. when the warden selects one prisoners at random, we are guaranteed that he doesn't just pick the same guy over and over again. I.e. it doesn't really matter what N you pick, because you can assume that all prisoners eventually visit an arbitrary number of times, in a starvation-free way. I.e. I am guaranteed that a prisoner p_i will visit before/after another prisoner p_j an arbitrary number of times eventually.

Fix the number of prisoners P.

1. Suppose the light begins in the OFF state. Let p_P be the counter. His job is to turn the light off whenever he sees it on. If turns the light off $P-1$ times, then he alerts the warden.

   All other prisoners do the following: when they enter the switch room, they turn the light on if it is not already on. They do this exactly once. If the light is already on, they wait, and try again later, trying to turn the switch on when it is off for one time.

   In this way, the light only becomes on if a prisoner who hasn't turned the switch on turns it on. It only turns off if the counter turns it off. Thus, if the counter counts $P-1$ times that he turned it off, it means that $P-1$ unique prisoners turned it on. Adding 1, since the counter also visits the room, he can know that P prisoners visited.

2. Now, if we do not know the initial state of the light switch, we change our strategy. Let us have a counter prisoner like before who turns the light off whenever he sees it on. All other prisoners will turn the light on from an off state exactly twice. The counter will report to the warden when he has counted $2P-2$ times that he switched off.

   Case 1: Light begins off. In this case, if the warden has turned the switch off $2P-2$ times, then the $P-1$ other prisoners must have come in 2 times each. This is correct.

   Case 2: Light begins on. Then in this case, the counter may be the first one in the room to turn off the light, even though no other prisoner has yet visited the room. This means we have one false count to turn the light off that did not actually come from another prisoner. However, if the counter counts $2P-2$ visits, then that means the other prisoners must have visited a total of $2P-3$ times (one less than $2P-2$). Each prisoner visits the room at most 2 times to turn the light on, so that means at least $\frac{2P-3}{2} = P-1.5$ other prisoners have visited. Since we cannot have fractional visitors, we round up to get that $P-1$ other prisoners have been there. Counting the counter himself, this leads to all prisoners.

4 Ex 5

Note: I heard about this problem before on a TedEd video, I don't claim credit for coming up with the solution originally.

First, the idea behind this is that if prisoners blindly guess red/blue, then their hit rate is going to be 50%. We need to encode information (which can be one of two states) into what we are saying, and this starts off with the first prisoner.

Let the first prisoner say red if he sees an odd number of red hats, and say blue if he sees an even number of red hats. This prisoner may or may not get their own hat right, but the rest will following this strategy.

If I am the second prisoner and I hear red, then that means the previous prisoner saw an odd number of red hats in front of him.

Thus, if I look ahead and also see an odd number of red hats, then I know I must not have a red hat, therefore I must have a blue hat. I say "blue!".

If instead I look ahead and see an even number of red hats, then I know I must have a red hat. I say "red!".

The idea here is that each prisoner (after the first) will know how the parity of the number of red hats for the hats of themselves plus the hats in front of them by counting the calls to red. Given P total prisoners, we know if prisoner p_i hears an odd number of calls to red, they know that prisoners $p_i, p_{i+1}, \ldots, p_P$ have an odd number of red hats.

Prisoner p_i knows whether $p_{i+1}, \ldots, p_P$ is odd or even. We can do case analysis.

Case 1: $p_i, p_{i+1}, \ldots, p_P$ odd red hats, with $p_{i+1}, \ldots, p_P$ even red hats implies p_i is a red hat. Thus, prisoner p_i saying "red!" correctly indicates the hat of p_i while also correctly signalling that $p_{i+1}, \ldots, p_P$ is even red hats.

Case 2: $p_i, p_{i+1}, \ldots, p_P$ odd red hats, with $p_{i+1}, \ldots, p_P$ odd red hats implies p_i is not a red hat. Thus, prisoner p_i saying "blue!" correctly indicates the hat of p_i while also correctly signalling that $p_{i+1}, \ldots, p_P$ is odd red hats (we didn't call red, therefore the oddity of red hats remains the same as before.

You can do case analysis similarly for case 3, 4, and this strategy checks out.

5 Ex 7

First,

$$S_2 = \frac{1}{1 - p + \frac{p}{2}}$$

By rearranging, we get

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

Now, we have

$$S_n = \frac{1}{1 - p + \frac{p}{n}}$$

Or,

$$S_n = \frac{1}{1 - p\frac{n-1}{n}}$$

Thus we can substitute p and get

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

6 Ex 8

I would only pick the ten-processor multiprocessor for a program when the program's parallel portion exceeds $\frac{8}{9}$.

Proof. Let us say a program costs 1 unit of work to run, p of which is parallelizable.

Then here are the runtimes on the Single processor versus multiprocessor.

$$T_S = \frac{1 workunit}{5 zil/sec}$$

$$T_M = \frac{1 - p}{1 zil/sec} + \frac{p}{10 zil/sec}$$

T_M is as above because the serial code can only be executed on one core, and the speed of the core for the multiprocessor chip is $1 zil/sec$.

Now, you just solve for when $T_M > T_S$ and get the result that multiprocessing is better for programs where $p \geq \frac{8}{9}$.

7 Ex 11

Claim: First, I argue that the Flaky Computer Corporation's Lock is mutually exclusive.

Proof. Suppose by contradiction that the Lock is not mutually exclusive. Then it is possible to have multiple threads enter the critical section after having called the lock.

Let T_1, T_2 be the two threads that enter the critical section after calling Flaky Lock.

In order to have entered the critical section, both threads must have had that $turn == me$, else they would not have passed the while loop on line 11. This means that they must have executed line

8, setting $turn = me$, and have then have proceeded to line 11 without any other thread overwriting the private Lock variable $turn$.

Therefore, one thread, say T_1, must have executed its lines 6 to 11 before another thread T_2 was able to execute its line 8.

However, since T_1 fully completed lines 6 to 11, it must have set the private Lock variable busy to true on line 10. Thus, when T_2 then begins calling the lock which happens at the earliest on line 8, then T_2 must be stuck on $while(busy)$ loop on line 9. This is a contradiction that T_2 was able to go through. □

Claim: Now, I argue that their lock is not deadlock free. Since starvation-free implies deadlock freedom, the contra positive implies that not deadlock free leads to not starvation-free.

Proof. We prove by showing an execution where there is deadlock. Suppose I have two threads T_1, T_2. Let this execution have both threads make it to the end of line 10. This is clearly possible, and results to the Lock's private variable turn being set to either 1 or 2. WLOG say it is set to 1.

Now, T_1 keeps executing and goes to line 11. Since $turn = 1$ and $me = 1$ for T_1, it goes through to the critical section. Now suppose T_1 unlocks the lock. Now, before T_2 even continues executing on line 10, T_1 calls the lock again.

Because $busy$ is set to true, T_1 is now stuck on line 9. Now, if T_2 continues from line 10, it is stuck on line 11, since $turn == 1$ but $me == 2$.

Thus, both threads are stuck. This lock is not deadlock free, therefore it is not starvation free either. □

8 Ex 14

We can create an l-exclusion algorithm by chopping the bottoms of the Filter algorithm. Now, instead of having the bottom layer allow for only one thread, we have it allow for l threads.

This satisfies l-exclusion. In the original algorithm, only a fixed number of threads were allowed at each level/waiting room. Thus, setting the lth waiting room as the final destination (i.e. the critical section, CS) allows for at most l threads to be here at the same time.

This modification also satisfies l-starvation-freedom. Because the original filter algorithm was starvation free, then it means that each level must have been starvation free too. If not, one of the threads would clearly have starved out. It follows then that the new level l which we set to be the final destination must be l-starvation-free.

9 Ex 15

Claim: This modified lock is not mutually exclusive

Proof. We will show via a counter example. Suppose I have two threads T_1, T_2. Let both threads get to the beginning of line 9. Clearly, this is possible because there have not been block conditions since y has not yet been set.

Now, if both threads are at this point, the variable x must have been set. Let's say it was set to 1. In other words, T_2 executed line 7 before T_1.

Now, let us say that T_1 executes line 9 followed by T_2 executing that same line. Then, y is set to 2.

Now, let us execute the following lines. T_2 continues execution from line 9 and since $x = 1$, T_2 locks the given lock.

However, T_1 now picks up its own execution from line 9. Since we have x equals 1, i equals 1, then T_1 does not lock the lock on line 11. It simply goes through to the critical section, even though T_2 should have been given exclusive access.

Thus, this lock fails at mutual exclusion and the scheme does not work. □

10 Ex 16

Claim: At most one thread gets the value STOP.

Proof. Suppose not. Then $p \geq 2$ threads must execute line 15 to return stop.

This means both threads must have each executed lines 10 to 15 consecutively before another thread overwrote the lock's private *last* variable. This is because *last* gets assigned to $i = myIndex.get()$ on line 10, and in order to return STOP we must get through the conditional on line 14 requiring $last == i$.

Thus, we have that a thread T_1 executes lines 10 to 15, returning STOP, while another thread T_2 must be at the beginning of line 10. Since T_1 did a full execution of lines 10 to 15 before T_2 starts, the lock's *goRight* variable is set to true. Therefore, when T_2 attempts to execute lines 10 to 15, it returns RIGHT on line 11. This contradicts that T_2 returns STOP as well. □

Claim: At most n 1 threads get the value DOWN.

Proof. Suppose by contradiction that all n threads return DOWN.

Then we must have that all threads execute lines 16-17, which means that for every thread's execution, we have $last! = i$ (otherwise we go into the logic of lines 14-15)

How does this happen? Say we have a thread T_1. On line 10 it assigns $last = i$. In order for that $last! = i$ before line 14, then another thread must have overwritten *last*, call that thread T_2. Now, T_1 returns DOWN, but T_2 is in trouble. We need to save T_2 by overwriting *last* using T_3, but this only passes the problem on. Eventually, there is no thread to overwrite *last* for a thread T_p's execution, so thread T_p must have $last == i$ and return STOP instead of DOWN. □

Claim: At most n 1 threads get the value RIGHT.

Proof. Suppose n threads return RIGHT. The only way for a thread to return RIGHT is if *goRight* is set to true. However, *goRight* is initialized to false. Thus, at least one thread must get to line 13 to change *goRight* to true, in which case that thread can no longer hit the return RIGHT statement on line 12. Thus, at most one less than the total number of threads can return RIGHT. □