Advanced Multiprocessor Programming // Exercise 1

Leon Schwarzäugl

April 24, 2023

1 Amdahl's Law

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

1.1 1.1)

1.1.1 a)

$$p = 0,98 \quad n = 96 \Rightarrow S = 33,1$$

1.1.2 b)

$$\lim_{n \to \infty} \frac{1}{1 - p + \frac{p}{n}} = \frac{1}{1 - p} = 50$$

1.2 1.2)

$$
\begin{aligned}
SU_{rel} &= \frac{T_1}{T_p} \\
SU_{rel} &= \frac{1}{1-p+\frac{p}{n}} \\
SU'_{rel} &= \frac{\frac{1-p}{k}+p}{\frac{1-p}{k}+\frac{p}{n}} \\
SU'_{rel} &\overset{!}{=} 3SU_{rel} \\
\frac{3}{1-p+\frac{p}{n}} &= \frac{\frac{1-p}{k}+p}{\frac{1-p}{k}+\frac{p}{n}} \\
3(\frac{1-p}{k}+\frac{p}{n}) &= (1-p+\frac{p}{n})(\frac{1-p}{k}+p) \\
\frac{3(1-p)}{k}+3\frac{p}{n} &= \frac{(1-p)^2}{k}+p(1-p)+\frac{p(1-p)}{nk}+\frac{p^2}{n} \\
\frac{3(1-p)}{k}-\frac{p(1-p)}{nk}-\frac{(1-p)^2}{k} &= p(1-p)\frac{p^2}{n}-3\frac{p}{n} \\
k &= \frac{3(1-p)-p(1-p)-(1-p)^2}{p(1-p)-p(3-p)n^{-1}} \\
k &= \frac{3n-(1-p)n-p}{pn-p\frac{3-p}{1-p}} \\
k &= \frac{2n+p(n-1)}{pn-p\frac{3-p}{1-p}} \\
k &\overset{p=0,8}{=} \frac{7n-2}{2n-22}
\end{aligned}
$$

1.3 1.3)

1.3.1 a)

$$
\begin{aligned}
S(n) &= \frac{1}{1-p+\frac{p}{n}} \overset{p=0,7}{=} \frac{1}{0,3+\frac{0,7}{n}} \\
S^{(I)}(n) &= \frac{1-p+\frac{p}{7}}{1-p+\frac{p}{7n}} \overset{p=0,7}{=} \frac{0,3+0,1}{0,3+\frac{0,7}{n}} \\
S^{(II)}(n) &= \frac{\frac{1-p}{3}+p}{\frac{1-p}{3}+\frac{p}{n}} \overset{p=0,7}{=} \frac{0,1+0,7}{0,1+\frac{0,7}{n}} \\
S(10) &= \frac{1}{0,3+0,07} = 2,703 \\
S^{(I)}(10) &= \frac{0,3+0,1}{0,3+0,07} = 1,29
\end{aligned}
$$

$$S^{(II)}(10) = \frac{0,1 + 0,7}{0,1 + 0,07} = 4,706$$

$$T^{(I)}(10) = \frac{T^{(I)}(1)}{S^{(I)}(10)} = 0,31$$

$$T^{(II)}(10) = \frac{T^{(II)}(1)}{S^{(II)}(10)} = 0,17$$

1.3.2 b)

$$\begin{aligned}
S^{II}(n') &= \frac{T^{(II)}(1)}{T^{(II)}(n')} \\
T^{(II)}(n') &< T^{(I)}(n') \\
\frac{T^{(II)}(n')}{S^{(II)}(n')} &< \frac{T^{(I)}(n')}{S^{(I)}(n')} \\
0,1 + \frac{0,7}{n'} &< 0.3 + \frac{0,1}{n'} \\
0.2n' &> 0.6 \\
n' &> 3
\end{aligned}$$

Optimization (II) results in lower execution time starting from $n = 4$.

1.3.3 c)

This is because the original program had some relative speedup depending on processor core count, whereas the speedup for optimization (I) only measures the speedup that can here be achieved by more processor cores. So this simply means that this optimization does not benefit so much from more cores, whereas the original program benefitted more from it (this speedup is already accounted for in optimization (I)). As such, this is the potential pitfall in relative speedup, as we are vulnerable to drawing false conclusions such as "Optimization (I) must make the program worse than originally, since the relative speedup is lower", when we do not think carefully about what the term describes.

1.4 1.4

$$
\begin{aligned}
T_1(20) &\overset{?}{>} T_{10}(1) \\
S(20) = \frac{T_1(20)}{T_1(1)} &= \frac{1}{1-p+\frac{p}{20}} \\
T_1(1) = \frac{T_{10}(1)}{10} &\Rightarrow \\
T_1(20) &= \frac{T_{10}(1)}{10(1-p+\frac{p}{20})} \\
T_1(20) &= \frac{T_{10}(1)}{10-10p+\frac{p}{2})} \\
T_1(20) &= \frac{2T_{10}(1)}{20-19p} \\
\frac{2T_{10}(1)}{20-19p} &> T_{10}(1) \\
2 &> 20-19p \\
p &> \frac{18}{19}
\end{aligned}
$$

Hence, we should buy the multiprocessor machine is the parallelization of our application is at least $\frac{18}{19}$.

2 2.1

The doorway section ensures that the computations - which take a bounded amount of time - are done before waiting. With this we can make sure that every thread that calls lock() will eventually enter the critical section, competing fairly for entry to the critical section.
Problems with FCFS being defined in a mutual exclusion algorithm by the first step taken arise from it being hard to decide who took the first step in most cases:

- **Read to different locations**: If multiple threads read concurrently, it is impossible to tell which went first
- **Read to same location**: Again, we cannot tell which process went first
- **Write to different locations**: Again, we cannot tell which process went first
- **Write to same location**: Here, a thread cannot distinguish if it alone writes or if another thread writes first, since it's write will destroy what the potential other thread wrote.

3 2.2

If we have two threads in the critical section at the same time and thread 1 goes beyond line 11, turn must have been equal to 1. If now thread 2 wants to go beyond that line, turn must be 2 - however, this can only happen once thread 2 has set it in line 8 and then made it out of the inner loop. However, since thread 1 has already set busy to true, thread 2 cannot leave the inner loop until thread 1 calls unlock(). Hence this protocol satisfies mutual exclusion.
This protocol can deadlock by one thread setting busy to true while another thread sets turn to itself. One thread will then be stuck in the inner loop (since busy is true) and the other will be stuck in the outer loop (since turn is not set to itself).
Since this protocol can deadlock, it cannot be starvation-free, as this is a stronger condition.

4 2.3

Mutual exclusion: Since the Peterson lock provides mutual exclusion, we know that a tree of depth 1 (=a Peterson lock) must also provide mutual exclusion. For a tree of greater depth however, a problem can arise in the following scenario:
Imagine a tree of depth two, and 4 threads A, B, C and D. At the left leaf, A spins while B acquires the leaf, and at the right leaf C spins and D acquires the leaf. At the root node B spins and D acquires the leaf. D now unlocks. Which makes C acquire the right leaf. At the root though, it can now happen that C sets victim to itself, making B entering CS, and then D setting the flag to false, which makes C also enter CS. Hence this construction does not provide mutual exclusion.
We can fix this by changing the unlocking order: by unlocking from the root node down to the leaves, we can guarantee that no more than two threads can ever be involved at one node at the same time. As such, each node now provides the mutual exclusion as known from a Peterson lock, and thus each subtree must also provides mutual exclusion. This fix as such leads to the treelock providing mutual exclusion.
Starvation freedom: Again, the Peterson lock provides starvation freedom, and as such, at every leaf, a thread cannot be blocked forever, meaning it must eventually succeed at acquiring the lock.
Deadlock freedom: Starvation freedom implies deadlock freedom.
Upper bound between acquiring and success: There is no upper bound to be given, as any number of threads may overtake another thread during a delay between acquiring one node and proceeding to the next one. Going back to our previous example, let us presume that B has just acquired the left leaf. If thread B is now delayed, any number of threads may acquire the treelock from the right side.

5 2.4

In line 13, we should change $i < n$ to $i < n - L$ - this decreases the number of filter levels by $l - 1$, which in turn allows l threads into the critical section. Also we need to adjust the spin to account for up to n-l threads at levels lower than the thread checked, and allowing a new thread in if a spot is free.

```
1  class Filter implements Lock {
2      private int[] level;
3      private int[] victim;
4
5  public Filter(int n) {
6      level = new int[n];
7      victim = new int[n-L+1];
8      for (i=0; i<n; i++) level[i] = 0;
9      }
10
11 public void lock() {
12     int i = ThreadID.get();
13     for (int j=1; j<=n-L; j++) {
14         level[i] = j;
15         victim[j] = i;
16         for (k=0; k<n; k++) {
17             if (k==i) continue;
18             while (count(level[k]>j) >= L && victim[j]==i);
19         }
20     }
21 }
22
23 public unlock() {
24     int i = ThreadID.get();
25     level[i] = 0;
26     }
27 }
```

[h]

6 2.5

If thread 0 sets x and gets to line 8, reading y as -1, thread 1 can overtake, setting x to 1, also reading y as -1 and then setting y to 1. x is 1 and thread 1 enters CS. If now thread 0 gets to line 10, x is 1 and thread 0 enters the if path, calling lock() and also entering CS. Hence FastPath does not offer mutual exclusion and their claim is wrong.

7 3.1

The construction does not yield an atomic Boolean MRSW register:

Imagine s_table is of great length N, and a thread T writes x to s_table. If a thread with a low index (such as 0) now reads, it can read the new value written by thread T. However, it takes time for s_table to fill up with the updated values, so thread with index close to N has a greater chance to read the old value. For an atomic register, that would be not allowed.

8 3.2

The construction does not yield an atomic MRSW register:

Imagine thread 0 writing x at time t_0 and writing y at time t_1 such that $t_0 \rightarrow t_1$. If thread 1 now reads overlapping with the second write it returns y and writes y in the second row of a_table. If thread 2 now initiates a read at a later time than thread 1, but still overlapping with the second write, it can now happen that it reads the outdated value x in its corresponding third column. For an atomic register, this should not be able to happen.

9 3.3

The replacement with safe registers leads to a scenario of consideration in which read and write of flag overlap, as the read might then return either the new or the old value. In case the read returns true, thread 0 will wait in the loop until thread 1 writes victim = 1 and enters CS after, while thread 1 waits for thread 0 to call unlock(). If the read returns false, thread 0 will enter CS, while thread 1 waits for thread 0 to call unlock().
In either case, the threads do not enter CS concurrently, and as such we can still assert mutual exclusion.