Theory Assignment 1

Tushita Sharva

August 30, 2024

1 Question 1

You are given a program that includes a method M that executes sequentially. Use Amdahl's law to resolve the following questions.

1. **Suppose M accounts for 30% of the program's execution time. What is the limit for the overall speedup that can be achieved on an n-processor machine?**
2. **Suppose M accounts for 40% of the program's execution time. You hire a programmer to replace M with M', which has a k-fold speedup over M. What value of k yeilds an overall speedup of 2 for the whole program?**
3. **Suppose M', the parallel replacement for M, has a four-fold speedup. What fraction of the overall execution time must M account for if replacing it with M' doubles the program's speedup? You may assume that the program when executed sequentially, takes unit time.**

Answer: Amdahl's law states that the **maximum speed up that can be achieved is**

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

In the problem statement it is given that one of the method which is sequential is M. There might be more methods that are sequential. For now, since we have some knowledge about one of the sequential methods, we know that having more sequential procedures would only lessen the speed up. Therefore, having this knowledge of M provides the maximum speedup that can be achieved if everything else is being executed in parallel.

1. Here it is not clear if 30% of the code which executed sequentially is going to be parallelised or the 30% is sequential and let is parallelised. If it is first case,

$$\begin{aligned}
&sequential = 30\% \\
&\implies \text{(maximum) } parallel = 70\% \\
&\implies S = \frac{1}{1 - 0.7 + \frac{0.7}{n}} \\
&\implies S = \frac{1}{0.3 + \dfrac{0.7}{n}} \\
&\implies \text{(simplifying) } S = \frac{10n}{7 + 3n}
\end{aligned}$$

Upper limit is $\lim_{n\to\infty} \dfrac{10n}{7+3n} = \dfrac{10}{3} = \mathbf{3.33}$

If it is second case,

$$\begin{aligned}
&parallel = 30\% \\
&\implies \text{(maximum) } sequential = 30\% \\
&\implies S = \frac{1}{1-0.3+\frac{0.3}{n}} \\
&\implies S = \frac{1}{0.7+\dfrac{0.3}{n}} \\
&\implies \text{(simplifying) } S = \frac{10n}{3+7n}
\end{aligned}$$

Upper limit is $\lim_{n\to\infty} \dfrac{10n}{3+7n} = \dfrac{10}{7} = \mathbf{1.43}$

2. Here it is not clear what is their intent. Do they want to say that 40% code is now parallelised with k-speed up while the other code remains the one that **can be** parallelised (with any speed up)? Or is it that 40% code is only one that could be executed in parallel and the rest 60% will be sequential? I am assuming that it is second case.

$$P = 0.40 \quad S = 2$$

Using Amdahl's Law:

$$2 = \frac{1}{(1-0.40)+\frac{0.40}{k}}$$

Solving for k:

$$2 = \frac{1}{0.60+\frac{0.40}{k}}$$

$$0.60 + \frac{0.40}{k} = 0.50$$

$$\frac{0.40}{k} = 0.50 - 0.60$$

$$\frac{0.40}{k} = -0.10$$

This equation is not possible because speedup k cannot be negative, because we cant have -ve speed up. So, no value of k can yield an overall speedup of 2 if only 40% of the execution is parallel.

3. Given: Method M' has a four-fold speedup. - We need to find the fraction of the overall execution time M must account for if replacing it with M' doubles the program's speedup. Let P be the fraction of the program that M accounts for.

$$k = 4 \quad S = 2$$

Using Amdahl's Law:

$$2 = \frac{1}{(1-P)+\frac{P}{4}}$$

Solving for P:

$$2 = \frac{1}{1 - P + \frac{P}{4}}$$

$$1 - P + \frac{P}{4} = 0.50$$

$$1 - 0.75P = 0.50$$

$$0.75P = 0.50$$

$$P = \frac{0.50}{0.75}$$

$$P = \frac{2}{3} = 0.678$$

So, M must account for approximately 68% of the overall execution time for replacing it with M' to double the program's speedup.

2 Question 2

Consider a variant of Peterson's algorithm, where we change the unlock method to be as shown in Fig. 2.17. Does the modified algorithm satisfy deadlock-freedom? What about starvation-freedom? Sketch a proof showing why it satisfies both properties, or display an execution where it fails

Answer: This is how the lock algorithm would look like in the mentioned setting.

```
class Peterson implements Lock {
    // thread-local index, 0 or 1
    private boolean[] flag = new boolean[2];
    private int victim;

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        victim = i;
        while (flag[j] && victim == i) {}
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
        int j = 1 - i;
        while (flag[j] == true) {}
    }
}
```

Listing 1: Peterson's Lock Implementation

Deadlock freedom: In this context, it means that when both the threads try to acquire the lock, at least one of them should succeed. The given program is **deadlock free** because:

1. If both of them are competing, after both write their values for victim, both of their flags are True, but only one would be the the victim at the end. The non-victim thread would acquire the lock.
2. Only if one of them is requesting, the flag of the other would be false. (Either because it is unlocked, or because it didn't even start yet). So the loop would break and requesting thread will acquire the lock.

Starvation freedom: In this context, it means that if a thread wants to acquire a lock, eventually it will succeed. The given program is **starvation free** because eventually all calls would succeed.

1. If both A, B are competing, the only way A's unlock would never return is when B's flag never turns off. For that to happen, B should not be able to call the `unlock()` method itself, because unlock ensures that B's flag becomes false. For that to happen, the `lock()` should not return, which means that B should be stuck in loop either because flag[A] is true and B is the victim at same time. But since A is already waiting at line 18, it already crossed line 16, where flag[A] = false. Therefore no chance of starvation.

   To explain in simple terms, the condition would not occur because the mutually exclusive conditions are being set just before entering the loop, and they are designed to exit at some time.

3 Question 3

```
1  public class HWQueue<T> {
2      AtomicReference<T>[] items;
3      AtomicInteger tail;
4      static final int CAPACITY = Integer.MAX_VALUE;
5
6      public HWQueue() {
7          items = (AtomicReference<T>[]) Array.newInstance(AtomicReference.class, CAPACITY);
8          for (int i = 0; i < items.length; i++) {
9              items[i] = new AtomicReference<T>(null);
10         }
11         tail = new AtomicInteger(0);
12     }
13
14     public void enq(T x) {
15         int i = tail.getAndIncrement();
16         items[i].set(x);
17     }
18
19     public T deq() {
20         while (true) {
21             int range = tail.get();
22             for (int i = 0; i < range; i++) {
23                 T value = items[i].getAndSet(null);
24                 if (value != null) {
25                     return value;
26                 }
27             }
28         }
29     }
30 }
```

Listing 2: HWQueue Implementation

Linearization Point at Line 15 The linearization point could be at line 15, where `tail` is incremented. However, consider the following execution:

1. **Thread A** calls `enq(x)`. It executes line 15, increments `tail`, and gets slot `i`.
2. **Thread B** also calls `enq(y)`. It executes line 15, increments `tail`, and gets slot `i+1`.
3. **Thread B** proceeds to line 16 and sets `items[i+1]` to `y`.
4. **Thread A** then sets `items[i]` to `x`.

If the linearization point were at line 15, the order of `enq()` calls implies that `x` should be enqueued before `y`. If `y` is observed in the queue before `x`, it contradicts this order. Therefore, the linearization point cannot be at line 15.

Linearization Point at Line 16 The linearization point could also be at line 16, where the item is stored in the array. Consider the following execution:

1. **Thread A** calls `enq(x)` and increments `tail` at line 15.
2. **Thread B** calls `enq(y)` and increments `tail` at line 15.
3. **Thread B** then stores `y` in `items[i+1]` at line 16.
4. **Thread A** finally stores `x` in `items[i]` at line 16.

If the linearization point were at line 16, it would imply that `y` was enqueued before `x` if the array is observed. This contradicts the order established by the `tail` values. Thus, the linearization point cannot be at line 16 either.

Conclusion Since the only memory accesses in `enq()` are at lines 15 and 16, and neither can serve as a definitive linearization point, we must conclude that `enq()` has no single linearization point.

Linearizability Even though `enq()` lacks a single linearization point, it can still be linearizable if every `enq()` operation appears to take effect in a way that respects the real-time order of execution. Linearizability does not require a single point in the code to act as the linearization point. It only requires that there exists some point during the operation (not necessarily a single instruction) where the operation appears to take effect instantaneously.

In this case, the entire `enq()` operation, including both lines 15 and 16, could together define the linearization interval. If the queue operations respect this interval, then the queue can still be considered linearizable. Therefore, the `enq()` method can still be linearizable even though it has no single linearization point.