Assignment 10

Alex Clemmer
Student number: u0458675

Problem 1:

6.3.1 The book's example [pg. 577] contains a constant for controller overhead, while the question contains a "controller transfer rate". I will calculate *exactly* how I was told to by Tyler, the TA.

That said, I think that this is wrong, and the assignment tells us not to worry about the controller overhead anyway. The reason I think this is wrong is because in data storage, we all know that speed is the premium. I suspect that it is highly unlikely that we transfer all data from disk to the controller and then, after waiting for all of it, from the controller to main memory. Generally the idea is to get everything back to main as swiftly as possible, and it makes no sense to incur linear controller overhead.

Besides that, I've researched this as best I can, and as far as I can tell, there is a controller overhead, but it tends to be the time it takes to get the controller activated and sending information out, rather than a linear stack on top of the disk transfer time.

If you think that this reasoning is correct, then I would appreciate it if you took this into account as you evaluated the credit I get on this answer.

$$\begin{aligned} \textbf{(a)} \quad & 11.0 \text{ ms} + \frac{0.5 \text{ rotation}}{7,200 \text{ RPM}} + \frac{1 \text{ KB}}{34 \text{ MBytes/sec}} + \frac{1\text{K B}}{480\text{Mbits/sec}} \approx 15.1975 \text{ ms} \\ \textbf{(b)} \quad & 9.0 \text{ ms} + \frac{0.5 \text{ rotation}}{7,200 \text{ RPM}} + \frac{1 \text{ KB}}{30 \text{ MBytes/sec}} + \frac{1\text{KB}}{500 \text{ Mbits/sec}} \approx 13.2013 \text{ ms} \end{aligned} \tag{1}$$

6.3.2 Our best time would contain no seek overhead and no rotational latency. Thus:

$$\begin{aligned} \textbf{(a)} \quad & \frac{2 \text{ KB}}{34 \text{ MBytes/sec}} + \frac{2 \text{ KB}}{480\text{Mbits/sec}} \approx 0.061711519607 \text{ ms} \\ \textbf{(b)} \quad & \frac{2 \text{ KB}}{30 \text{ MBytes/sec}} + \frac{2 \text{ KB}}{500\text{Mbits/sec}} \approx 0.06920016666 \text{ ms} \end{aligned} \tag{2}$$

6.3.3 There is a clear and dominant factor here, and it is seek time. Whether or not we get data fast or slow is pretty much contingent upon how long it takes to seek. After that, the dominant factor is the rotational latency. Obviously we don't want to downplay how much time this takes compared to the other two factors, but even worst-case it still only takes around half the total time that an average-case seek would take.

Also worth noting is that increasing block size doesn't really change the read or write time by that much. In other words, for reasonably-sized blocks, the size does not appear to matter that much.

Problem 2:

6.6.1

$$\begin{aligned} \textbf{(a)} \quad & \frac{1 \text{ KB}}{34 \text{ MBytes/sec}} + \frac{1 \text{ KB}}{480 \text{ Mbits/sec}} \approx 0.0308557598 \text{ ms} \\ \textbf{(b)} \quad & \frac{1 \text{ KB}}{30 \text{ MBytes/sec}} + \frac{1\text{KB}}{500\text{MBits/sec}} \approx 0.03460008333 \text{ ms} \end{aligned} \tag{3}$$

6.6.2

$$\begin{aligned} \textbf{(a)} \quad & \frac{0.5 \text{ KB}}{34 \text{ MBytes/sec}} + \frac{0.5 \text{ KB}}{480\text{Mbits/sec}} \approx 0.0154279 \text{ ms} \\ \textbf{(b)} \quad & \frac{0.5 \text{ KB}}{30 \text{ MBytes/sec}} + \frac{0.5 \text{ KB}}{500\text{Mbits/sec}} \approx 0.0173000 \text{ ms} \end{aligned} \tag{4}$$

6.6.3 Obviously, the larger the memory, the more decode logic you have to wade through, not to mention the fact that propagating addresses around and through the Flash "array" obviously costs time. What may be surprising is that the dropoff appears to be so precipitous. Remember though, that in this example, the size increases by powers of two, while the dropoff does not decline at this rate. So the effects are not as severe as one might think.

Problem 3:

3-A So, first, the assumption I'm going to make is that one bit is sent per cycle. To compute this, we start by seeing how far a signal will travel in a single cycle:

$$\frac{2 \cdot 10^8 \text{ m/s}}{100 \cdot 10^6 \text{ MHz}} = 2 \text{ meters} \tag{5}$$

Then, we evaluate the "capacity" of the 10 meter wire. Assuming the rate of signal propagation across the wire is constant, that means that immediately as one is introduced at one end, another will disappear at the other end. Thus at any point there should be **5 bits** moving across the wire.

3-B The first thing we look at is how much wire the signal covers in one cycle:

$$\frac{2 \cdot 10^8 \text{ m/s}}{10 \cdot 10^9 \text{ MHz}} = 0.02 \text{ meters} \tag{6}$$

Using similar logic to above we get $\dfrac{1500 \text{ meters}}{0.02 \text{ meters/bit}} = 75,000 \text{ bits}$

3-C This is tricky. First we'll do **(a)**. Let's start with what we know. There is 1 request of 100 bytes that is sent out. Since we are writing, there is an overhead of 0.03 ms. The other end must read, which also incurs this overhead. In other words, the signal gets sent to the receiver, and then the server takes 0.03 ms to process this on top of transmission time. At this point, our total should look like this:

$$0.03 \text{ ms} + \frac{100 \cdot 10.2 \text{ bits}}{100 \cdot 10^6 \text{ bits/sec}} + \frac{100 \cdot 2^{10} \text{ bits}}{100 \cdot 10^6} + 0.03 \text{ ms} \tag{7}$$

Since I do not see anything about the amount of time it takes to find the data on disk or respond, I will not account for it here, although you should note that this is DEFINITELY NOT negligible. So, assuming that there is no data retrieval wait, we have another 0.03 ms that we must wait before we can write back in response.

Now we have to send the actual message. That's 100,000 bytes plus the overhead for each byte. And then at the end of the line, there's another read overhead. So, to our previous equation, we should add this:

$$\ldots + 0.03 \text{ ms} + \frac{100,000 \cdot 10.2 \text{ bits}}{100 \cdot 10^6 \text{ bits/sec}} + \frac{100,000 \cdot 2^{10} \text{ bits}}{100 \cdot 10^6} + 0.03 \text{ ms} \tag{8}$$

For **(a)**, this gives us 1.0352342 seconds $+4 \cdot 0.03$ ms, for a **total of 1.0353542**.

For **(b)**, the same sort of derivation follows, except we changed the clock speed from 10^6 to 10^9. This gives us a total of **total of 0.010472342**. It's a lot smaller because the clock cycle is smaller.

3-D It's important to note that between the two examples, the *only* difference is how fast we were sending the message out. This is pretty strong evidence that the main bottleneck is the frequency of our signal. If you want to go faster, increase how many cycles a second are going out.

I should also say that, while it's not a problem here, as you drop your cycle time, other bottlenecks may appear. If the R/W overhead of a real machine is more like a few milliseconds, or, say, you actually have to access memory, then you will start optimizing in other places.

Problem 4:

7.6.1 So there are $(m \times p \times n)$ multiplications, which leaves us with $(m \times p \times (n-1))$ additions. We will be adding up the products, which means that we need to wait until at least two are available per addition. So, all things considered, in this case, the speedup should be in the ballpark of 4 times.

7.6.2 In this case, we are basically mapping different elements to the same cache line. Thus, each update ends up causing a cache miss, and so any speedup gained is diminished by a factor of 3 multiplied by the costs associated with a cache miss.

7.6.3 The simplest way is to traverse the matrix via columns instead of rows. That should pretty much eliminate the problem we were having because then the elements will end up mapped to different cache lines.

The only other thing I can think of – and this is minor in comparison – would be to process the matrix index (i, j) and $(i+1, j)$ on the same core. Remember that we are dealing with false sharing, and thus this is important.

Problem 5:

First, the constraints. Once again, the additions depend on pairs of products, so in some ways we will be dependent on that factor. We obviously have overhead of $mx + mx$ to begin with.

That said, there are $(m \times p \times n)$ multiplications, and $(m \times p \times (n-1))$ additions. The multiplications are parallelizable, and after a few of them are done, so are the additions.

At some point, we can expect the number of multiplications to strictly limit how much more we can speed this all up by parallelization. In other words, this is in lot of ways this is the ideally parallel problem, and the "hard" limitations on speedup will come from limiting the number of operations there are to perform.

That is, apart from the distinct and well-defined constraints above, there will probably not be that many, unless you count hardware constraints (*e.g.*, you have to run one OS per core on Intel's 80-core Polaris chip, because neither support that many cores; also there are issues with memory and things like that).

Because the task is so blatantly parallelizable, the speed up limitation should be roughly the number of operations that can possibly be parallelized divided by the number of cores. My guess would be $\frac{mpn + t \cdot mp(n-1)}{x} - 2mx$, where t is some variable that accounts for the "problem" of addition.

What does that mean? Note that since addition is usually faster than multiplication (except in cases where the multiplication operation is strength-reduced, for example), the speedup afforded by parallelizing the addition will usually dependent on how fast the multiplication makes its operands available. That is, addition will execute as multiplication operations produce products to add together, and even then, the addition will finish generally before the next multiplication has, meaning that most of the speedup should be from how fast you can do the multiplication.

That of course is not to say that you can't speed up by parallelizing the addition! Certainly computing all of them at once is faster than doing them sequentially, but it is not freely parallelizable as the multiplication is.