Proof of Study 8

Qianlang Chen

U1172983

Problem 1

Part (a)

Since we can ignore the time for control overhead, the latency for reading is made up of 4 major parts: rotational delay, seek-time, reading time, and communication time.

For drive A, the time it would take to complete the above 4 major parts are as follows:

Rotational delay: secs/revolution / 2 = (1 / (7,500 / 60) rev/s) / 2 = **.004** s
Seek-time: **.007** secs
Reading time: secs/byte × bytes = (1 / 40e6 B/s) × 1,024 bytes = **.000 025 6** s
Communication time: secs/byte × bytes = (1 / 100e6 B/s) × 1,024 bytes = **.000 010 24** s
Total: .011 035 84 seconds

For drive B:

Rotational delay: secs/revolution / 2 = (1 / (10,000 / 60) rev/s) / 2 = **.003** s
Seek-time: **.004** secs
Reading time: secs/byte × bytes = (1 / 90e6 B/s) × 1,024 bytes = **.000 011 377 78** s
Communication time: secs/byte × bytes = (1 / 100e6 B/s) × 1,024 bytes = **.000 005 12** s
Total: .007 016 497 78 seconds

Part (b)

The minimum latency for either drive occurs when the data to read is already directly under the read-write head, meaning that the rotational delay and the seek-time are both zero. The reading time and the communication time are the only two major parts contributing in the final time.

For drive A:

Reading time: secs/byte × bytes = (1 / 40e6 B/s) × 2,048 bytes = **.000 05 12** s
Communication time: secs/byte × bytes = (1 / 100e6 B/s) × 2,048 bytes = **.000 020 48** s
Total: .000 071 68 seconds

For drive B:

Reading time: secs/byte × bytes = (1 / 90e6 B/s) × 2,048 bytes = **.000 022 755 56** s
Communication time: secs/byte × bytes = (1 / 200e6 B/s) × 2,048 bytes = **.000 010 24** s
Total: .000 032 995 56 seconds

Part (c)

The dominant factor that affects latency for both drive A and drive B is the seek-time, which contributes the most amount in the final time for both drives. Therefore, an improvement in the seek-time for either drive would yield (or at least, have the most potential to yield) the maximum decrease in latency.

Problem 2

Part (a)

Since SSDs do not have seek-times or rotational delays, the reading time and the communication time make up the total run time.

Therefore, for drive A:

Reading time: secs/byte × bytes = (1 / 1200e6 B/s) × 1,024 bytes = **.000 000 853 33** s
Communication time: secs/byte × bytes = (1 / 600e6 B/s) × 1,024 bytes = **.000 001 706 67** s
Total: .000 002 56 seconds

For drive B:

Reading time: secs/byte × bytes = (1 / 200e6 B/s) × 2,048 bytes = **.000 005 12** s
Communication time: secs/byte × bytes = (1 / 180e6 B/s) × 2,048 bytes = **.000 005 688 89** s
Total: .000 010 808 89 seconds

Part (b)

Since there is no seek-time or rotation delay for either drive, the minimum latency is calculated precisely the same way as the average latency, because it is also made up of reading time and communication time only.

For drive A:

Reading time: secs/byte × bytes = (1 / 1200e6 B/s) × 2,048 bytes = **.000 001 706 67** s
Communication time: secs/byte × bytes = (1 / 600e6 B/s) × 2,048 bytes = **.000 003 413 33** s
Total: .000 005 12 seconds

For drive B:

Reading time: secs/byte × bytes = (1 / 200e6 B/s) × 2,048 bytes = **.000 010 24** s
Communication time: secs/byte × bytes = (1 / 180e6 B/s) × 2,048 bytes = **.000 011 377 78** s
Total: .000 021 617 78 seconds

Part (c)

I would expect the latency to increase as the memory gets larger. A larger memory would require more effort to find the requested data, which would also increase the time for control overhead since a bigger memory address would need to be compared. However, since a flash ram does not have a spinning nature, this increase is insignificant.

Problem 3

Part (a)

Bits in-flight = transmission time * bit rate = (20 m / 2e8 m/s) * 100e6 bits/s = **10 bits**

Part (b)

Bits in-flight = transmission time * bit rate = (2,000 m / 2e8 m/s) * 10e9 bits/s = **100,000 bits**

Part (c)

For cable A:

Request sending delay: **.000 02** secs
Request sending time: (bits in request - 1) / bit rate = (100 bytes × 10.2 bits/byte - 1) / 100e6 bits/s = **.000 010 19** secs
Request transmission time: 20 m / 2e8 m/s = **.000 000 1** secs
Request receiving delay: **.000 02** secs
Data sending delay: **.000 02** secs
Data sending time: (100,000 bytes × 10.2 bits/byte - 1) / 100e6 bits/s = **.010 199 99** secs
Data transmission time: **.000 000 1** secs
Data receiving delay: **.000 02** secs
Total: .010 290 38 seconds

Note that the sending time is the time needed to push all bits into the wire. Between *N* bits, there are only *N-1* gaps, and that is why I subtracted one from the number of bits to send in when calculating the sending time for both sending the request and the data.

Similarly, for cable B:

Request sending delay: **.000 02** secs
Request sending time: (bits in request - 1) / bit rate = (100 bytes × 10.2 bits/byte - 1) / 10e9 bits/s = **.000 000 101 9** secs
Request transmission time: 2,000 m / 2e8 m/s = **.000 01** secs
Request receiving delay: **.000 02** secs
Data sending delay: **.000 02** secs
Data sending time: (100,000 bytes × 10.2 bits/byte - 1) / 10e9 bits/s = **.000 101 9** secs
Data transmission time: **.000 01** secs
Data receiving delay: **.000 02** secs
Total: .000 202 001 9 seconds

Part (d)

For both scenarios, the time it takes to push all bits of the data (response) into the wire takes the most amount of time. Therefore, the data sending time is the dominant factor of the latency in both cases, and decreasing it can provide the maximum potential to also decrease the overall latency.

Problem 4

Problem 6.3

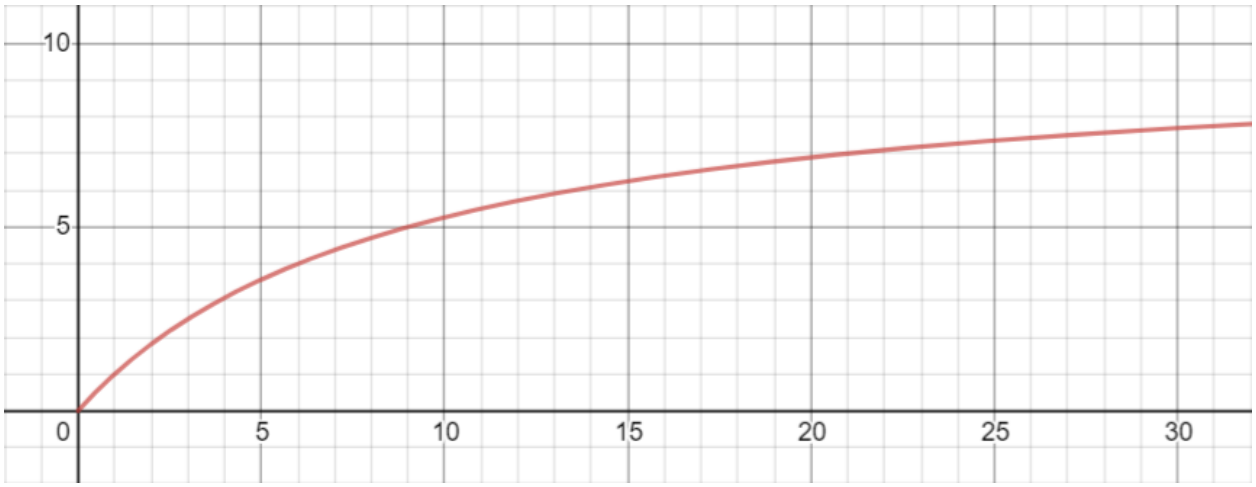
For a binary search, it is not effective to use many cores to speed up the algorithm at all. The strategy to use Y cores for a binary search would be to divide the array into Y equal portions and to let each core search for the queried element in the portion it is responsible for. Since the general behavior for a binary search in $log(N)$, using Y cores would only cut it down to $log(N/Y)$, which could be simplified down to $log(N) - log(Y)$. If Y is always smaller than N, the runtime saved by using such a strategy would be insignificant, since $log(Y)$ would ever only be a small amount. In this sense, the ideal number of cores for a binary search would be just *one* since using more cores would only be a waste and is never worth the slight improvement in the runtime, not to mention that dividing the array up and assigning them to multiple cores themselves would inevitably cost some runtime too.

Problem 6.5

Since merge-sort is a divide-and-conquer algorithm, and its dividing part can only be done on one core and is not parallelizable, let us assume that the conquering part, which is parallelizable, contributes to 90% of the total runtime. Also, assuming that dividing up the job for new cores takes negligible runtime, and assuming that the runtime of a merge-sort with a single core is T, we can calculate the speed-up factor when using Y cores like so:

$$\begin{aligned}\text{Execution time after improvement} &= \text{Exe. time affected} / \text{Improvement} + \text{Exe. time unaffected} \\ &= 90\%T / Y + 10\%T \\ &= (0.9 / Y + 0.1) \times T\end{aligned}$$

$$\begin{aligned}\text{Speed-up factor} &= T / \text{Execution time after improvement} \\ &= 1 / (0.9 / Y + 0.1)\end{aligned}$$



Note that, in the graph, the horizontal axis represents the number of cores and the verticle axis represents the speed-up factor. According to the graph, using 8 cores might be around the best (making the algorithm run about 5 times faster, with assumptions) since the runtime speed gain by using more than 8 cores does not justify a large number of cores.

When the number of cores used, *Y*, was equal to the length of the array to sort, even though most of the divide-and-conquer part of the algorithm would take a very short amount of time, the merging part at the end would still take a linear runtime for each stage, since only one core of the two can be doing the job. Since there are *logN* of such stages, the entire algorithm would still run in *NlogN* runtime. Therefore, it is not effective at all to use too many cores.