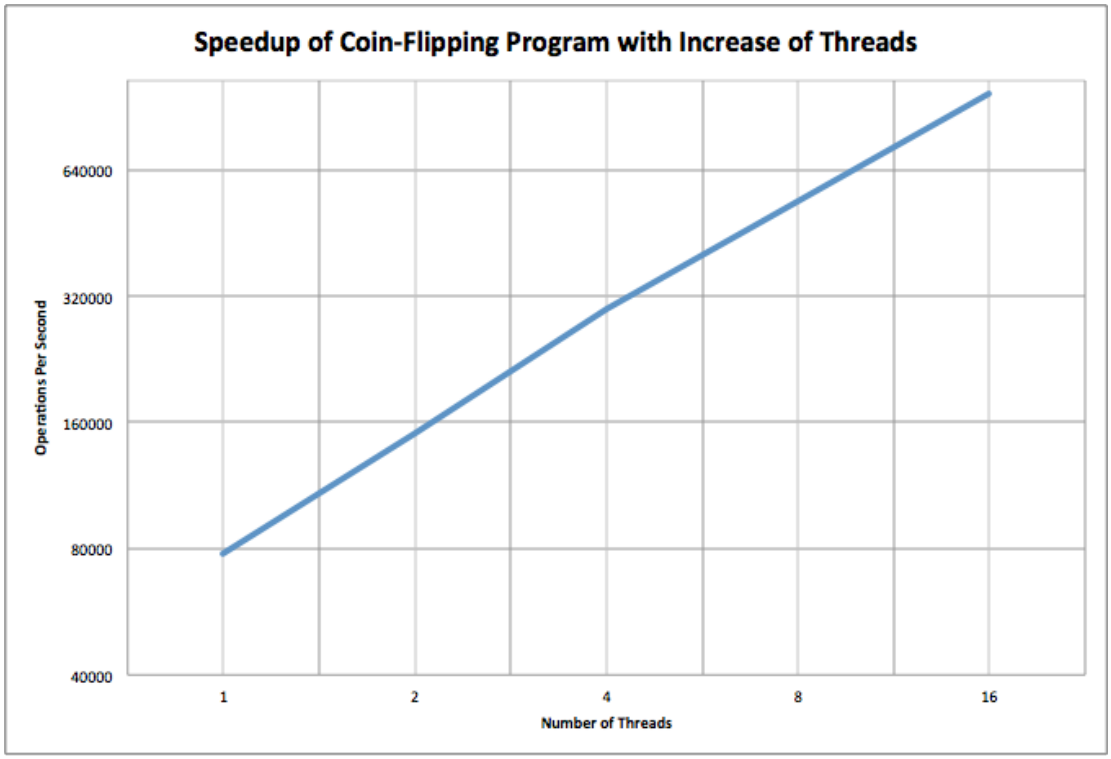
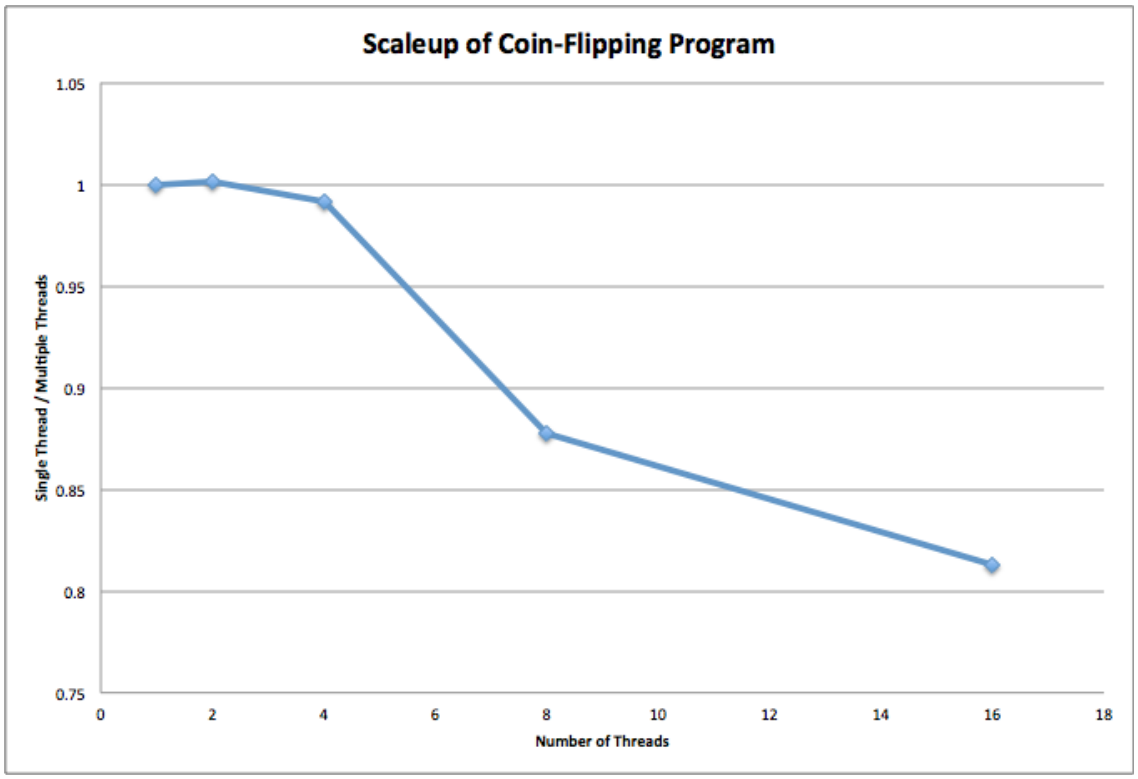
Scaleup and Speed

1.



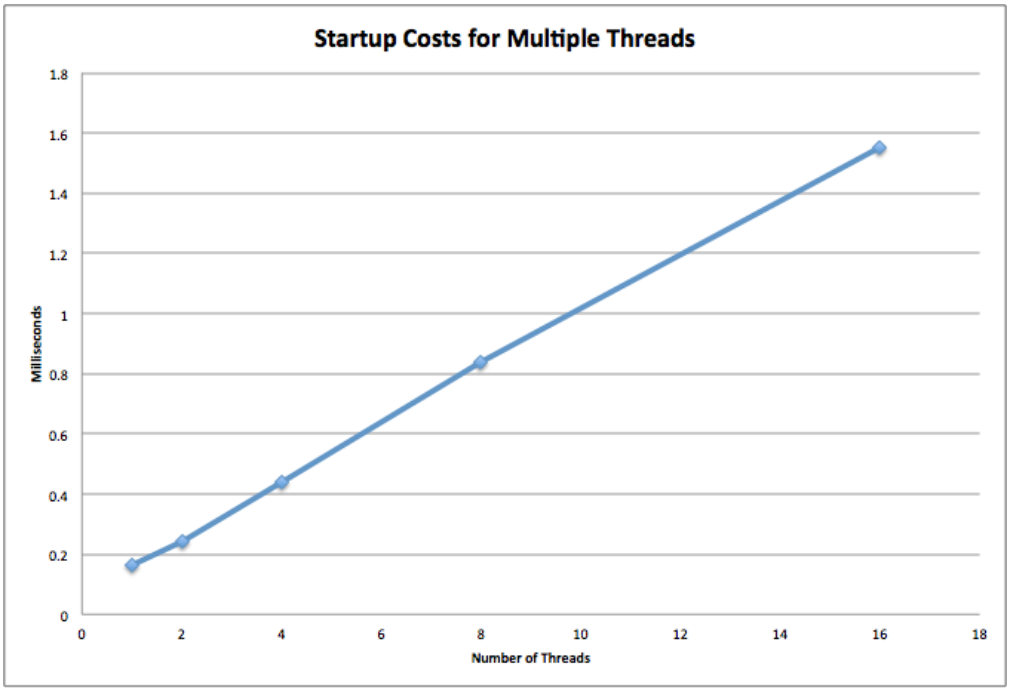


2. The speedup, as seen by the graph above, is slightly sublinear. This small lapse in runtime (which starts around the 8 thread mark) is probably due to not having the full potential of the virtual cores of the CPU as well as startup costs. The Amazon c4.4xlarge instance uses 8 real cores and 8 virtual cores for a total of 16 cores at our disposal. We see the slight drop-off from 8 cores to 16 cores because the virtual cores are performing less efficiently than the first 8 physical cores thus the program runs slightly slower at more than 8 threads. Secondly, the sublinear speedup could be due to the startup costs of having additional threads. The increase in time to create additional threads could be hurting the overall runtime of programs with multiple threads. Third, there may be some interference from the synchronized *increment* method that accesses the global static variable for "number of heads" in the entire program. This could be considered a shared resource. Each thread calculates the number of heads for its portion of the total calculations. At this point, the static *increment* method is called to add this total to the overall total. Because this method is synchronized to access the static overall total, each thread has to wait for the one before it to finish calling this method before it can call it. As the threads were increased, this lock could cause the program to have sublinear runtime. Looking at the graph for scaleup, we see a somewhat constant scaleup until 4 threads. From there we see a dramatic dropoff to 8 threads and a slightly less dramatic dropoff in scaleup at 16 threads. This is sublinear scaleup. This can be explained with similar reasoning as above. As the data set gets bigger, the number of threads are able to function at a linear scaleup initially because the physical cores of the CPU are able to handle the increase in data with the increase in threads. But as we get to 8 and 16 threads we see dropoff because the cores are no longer performing at their full efficiency. This correlates to the speedup graph where we begin to see sublinear performance around the 8 thread mark. This could be because of the virtual cores performing at a lesser efficiency than the physical cores or because of other factors including the interference mentioned above or startup costs that cause the programs with more threads to spend more time creating the initial threads. This would slow down the entire program even if the parallel portion performed faster.

3. Speedup does not continue past the number of cores because the CPU does not hold enough physical and virtual cores to handle the increase in threads. Thus, the program tries to compensate with the cores it does have by using the maximum number of cores available. For example, if the CPU had 8 physical cores and 8 virtual, it can run a maximum of 16 threads. An addition of threads to 32 would not lead to a speedup because the 16-core CPU could not run the extra threads. Thus, the CPU handles the program on 16 cores and instead of speedup we see a plateau. It plateaus because more threads are being designated than can be handled by the CPU and the CPU uses the maximum number of cores that it has every time, no matter the number of threads designated. Thus the runtime (speedup) would be constant or degrade for these calls. As sublinear factors like startup costs begin accumulating (for example in creating more threads that don't exist), the speedup could begin to degrade over time.

Design and run an experiment that measures the startup costs of this code

1. The measure startup costs, I ran the program for zero coin flips while changing the number of threads from 1 to 16. Because we designate that zero flips will take place the resulting time indicates how much time it takes to create the threads and do nothing with them. This is the startup cost.



2. From running the experiment, I can see a linearly proportional increase in startup cost versus number of threads. Looking at the graph, I would estimate that the each thread costs a startup time of approximately .097 milliseconds per thread created. For 16 threads, it takes a total of 1.55 milliseconds to create the threads with zero input. With that said, my experiment includes all costs of running the program, not just the creating of the threads. My estimation includes these costs in estimating the startup costs for creating extra threads. That is why the program with a single thread still takes .16 milliseconds.

3.

For 100 threads:

$$Startup\ Costs\ for\ 1\ thread\ = .097\ ms$$

$$P\ = \frac{total\ time - startup\ costs}{total\ time}$$

$$P\ = \frac{77664.47394 - .097}{77664.47394} = 0.9999987510377$$

$$S\ = factor\ by\ which\ parallel\ portion\ improves\ =\ 100$$

$$Amdahl's\ Law:\ Speedup\ = \frac{1}{1 - P + \frac{P}{S}}$$

$$Speedup\ = \frac{1}{1 - 0.9999987510377 + \frac{0.9999987510377}{100}}$$

$$Speedup\ = 99.98763680187567$$

For 500 threads:

$$S\ = factor\ by\ which\ parallel\ portion\ improves\ =\ 500$$

$$Amdahl's\ Law:\ Speedup\ = \frac{1}{1 - P + \frac{P}{S}}$$

$$Speedup\ = \frac{1}{1 - 0.9999987510377 + \frac{0.9999987510377}{500}}$$

$$Speedup\ = 499.6885779932193$$

For 1000 threads:

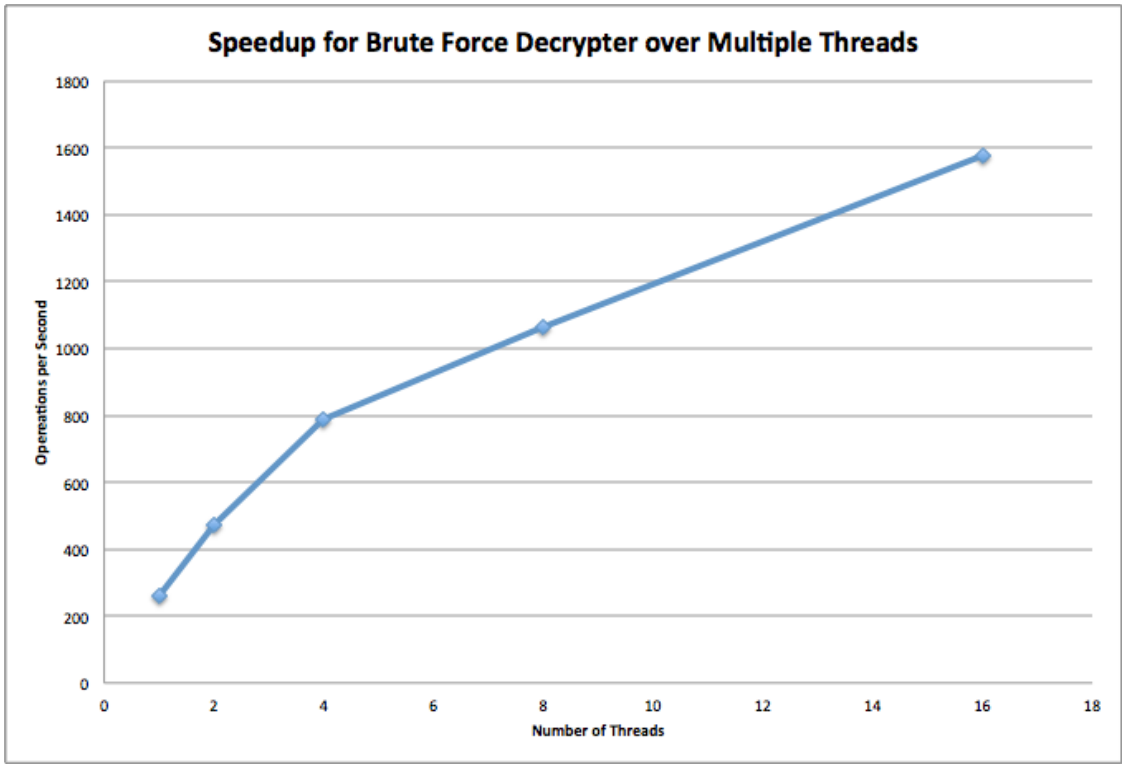
$$S\ = factor\ by\ which\ parallel\ portion\ improves\ =\ 1000$$

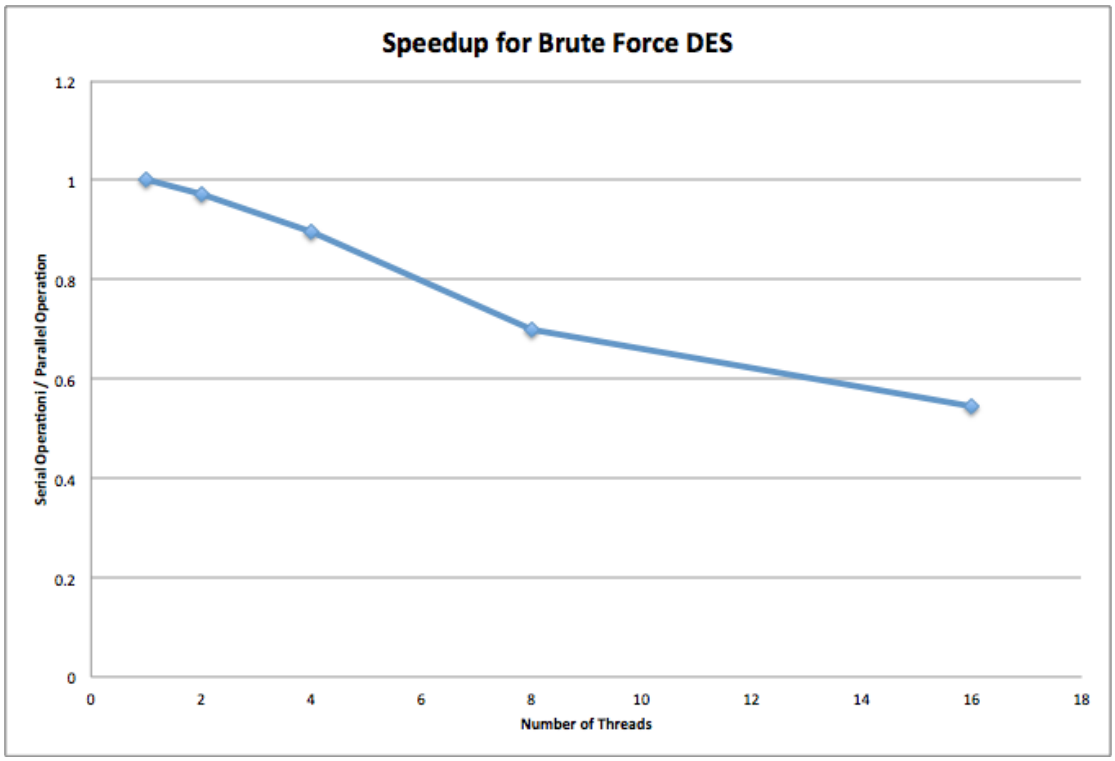
$$Amdahl's\ Law:\ Speedup\ = \frac{1}{1 - P + \frac{P}{S}}$$

$$Speedup\ = \frac{1}{1 - 0.9999987510377 + \frac{0.9999987510377}{1000}}$$

$$Speedup\ = 998.75384150857351$$

Analysis for Part 2





1. According to the speedup graph, the brute force program runs at a sublinear speed. Up to 4 cores, we see an almost linear speedup. From there, we see the drop-off in speed and sublinear performance. We might expect that the 16 core performance be slightly lower than it appears on the graph such that the curves begins to plateau as is common with sublinear performance. The sublinear speedup indicates the program and the multi-threads are not perfectly efficient in their parallelism and speeding up the overall program. I will discuss why in the next section. For scaleup, we also see sublinear performance. We see the serial performance of the brute force program as a baseline and increase the number of a keys and the number of threads by a factor of 2 for each data plot. The first three plots (1 thread through 4 threads) show a somewhat linear decrease in performance as it correlates to the the bigger data inputs. We see a drop at 8 threads that seems to be lower than the previous trend. From there we see the performance slightly level off at 16 threads. The drop indicates that as larger data sets are used with more amounts of threads, the performance of the brute force program will not be maintain perfect efficiency with the increased data load.

2. The speedup and scaleup are less than linear most likely due to some interference in the parallel section of the code. The starting costs were measured to be approximately 2 milliseconds. Thus, we can rule starting costs out as a reason for the noticeable sublinearity of both the speedup and scaleup graphs. We can also rule out skew as a major factor because in each data point was tested for twenty iterations and in analyzing each iteration, we do not see particular cases where one iteration takes markedly longer than others. This means that in no cases is the decryption method takes an uncommonly longer amount of time to determine a key. Thus, this assumption was proven false and interference must be the source of the sublinear behavior of the program. As I was conscious not to include shared variables or objects with individual threads, the primary culprit could be the java SealedObject object that holds the data to be decrypted. It is read in every thread to decrypt the code. Because it is a native java object, it is possible that java does not allow asynchronous calls to access the object and this is what causes a dropoff in performance as the program becomes more parallel. This is the only major piece that could case the sublinearity we see with the speedup and scaleup of the program. A slight performance decrease could be attributed to the minor startup costs mentioned above or the 8 virtual cores of the CPU running less efficiently than the first 8 physical ones but the SealedObject is likely the main culprit.

3.

a. In the scenario provided where we are running 2^{56} keys on 64 cores, each thread is responsible for 2^{50} keys because $2^{56} / 2^6 = 2^{50}$
    i. $2^6 = 64$
    ii. In other words, if we were running 2^{56} keys on 64 cores, each core would be looking at 2^{50} keys
b. According to our scaleup data, it took 1 thread 1425.6 ms to process 2^{18} keys
    i. 2^{18} was chosen arbitrarily by me as a test value
c. For 1 thread to iterate over 2^{50} keys, it would take approximately $\mathbf{2^{32}}$ times as long
    i. this is because $2^{50} = 2^{32} * 2^{18}$
d. $1425.6 * 2^{32} = 6122905377177.6$ms which is the time it takes for one thread to iterate over 2^{50} keys
e. If all 64 threads are acting in parallel (with no interference and linear scaleup), 6122905377177.6 ms would be the time it takes for each thread (and thus the whole program) to complete → this would be if we had perfect linearity (straight horizontal line) for scaleup
    i. 6122905377177.6 = 194.027307259826244 years to deal with a 56 bit key with 64 cores in an ideal situation
f. Including interference and the sublinear scaleup behavior of the program:
    i. We know that: 194 years / actual time = ‘y value’ on the scaleup graph
    ii. We know that at 16 threads, the ‘y value’ is .5449, thus the ‘y value’ for 64 threads MUST be less this number.
    iii. By estimating that y = .50, then the actual amount of time = 194 years/ real time = .5 ⇒ 388 years

Thus, it would take approximately **388 years** including interference (as estimated by scaleup) to crack a 56-bit key with a 64 core CPU.