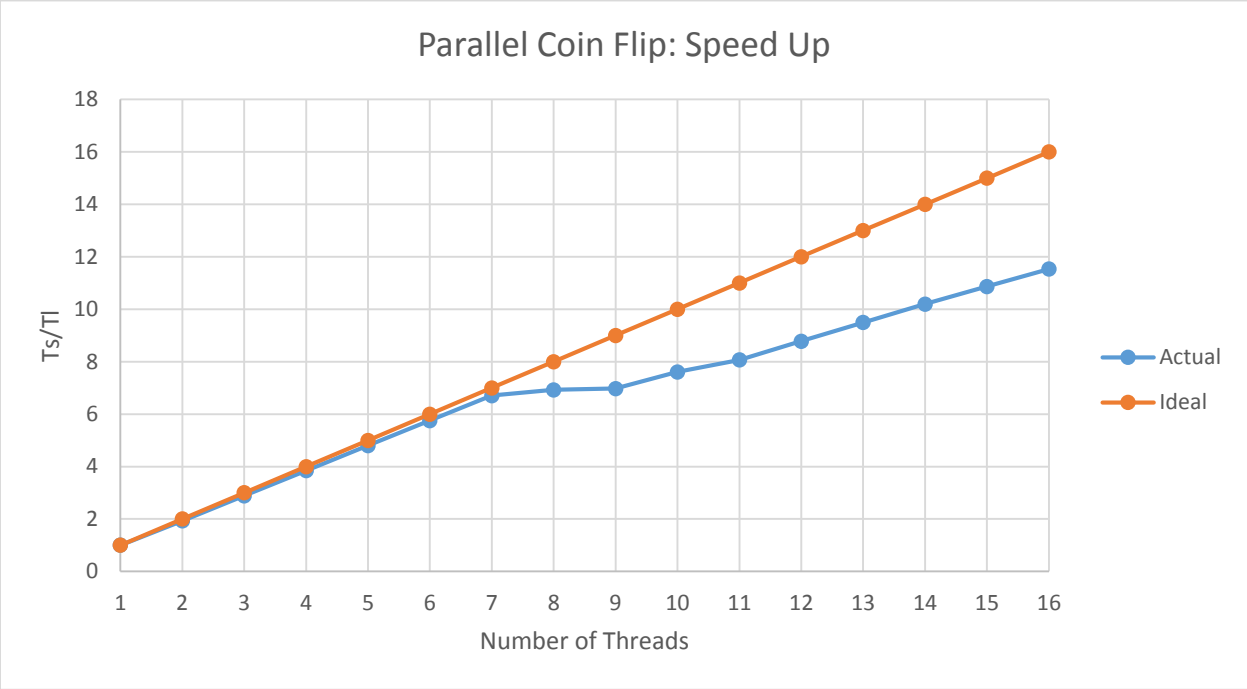
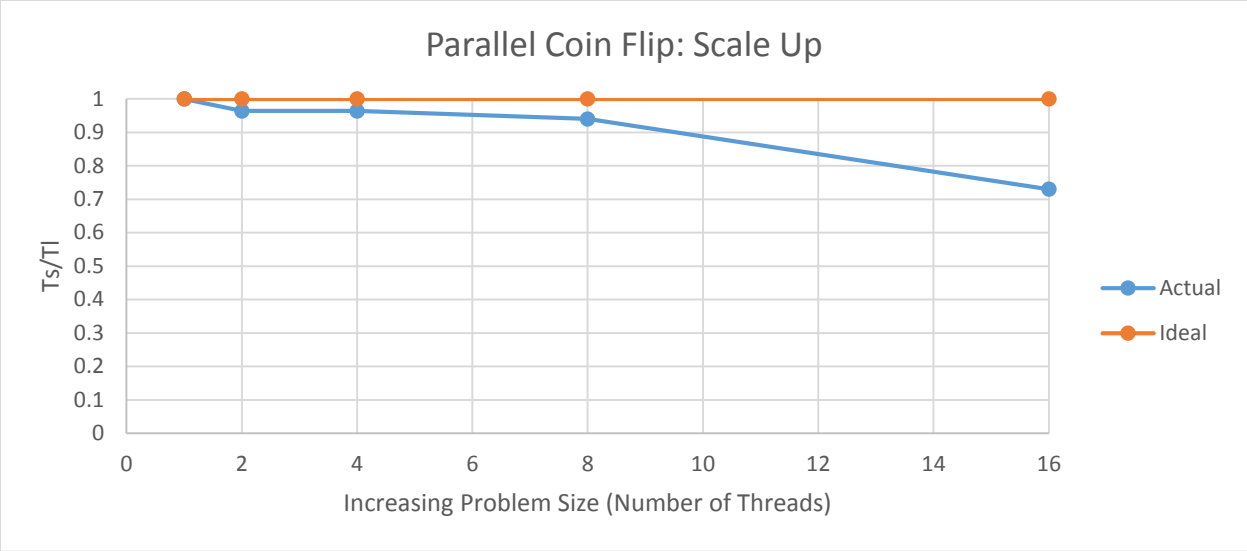
Stephanie Chew
schew@jhu.edu
schew4

In the implementation of CoinFlip, I implemented Runnable and so each of the threads that was created in the main method will go through the run method and using a random number generator, simulate a coin flip and count up the number of flips the coin made. This is made without using shared variables, so after each of the threads is finished and joined, the main method gets the number of heads from the CoinFlip object and adds it to its own running total of heads.

In the implementation of BruteForceDES, we take a similar approach in implementing Runnable and have each of the threads go through the run method to brute force through each of the keys. However, there is the issue of knowing which threads are supposed to search which particular key, so we specify and pass through a particular interval that the thread is suppposed to search. If the key is found, then the thread will say so. No shared variables are necessary. However, the startup costs are greater as we have to pass through a different SealedObject for each of the BruteForceDES objects and make a new BruteForceDES object for each thread.





Algorithm (true) speedup/scaleup measures the scaling performance of the algorithm as a function of processing elements. In this case, from 1…8. Characterize the algorithmic speedup/scaleup. If it is sub-linear, describe the potential sources of loss.

We observe a slightly sublinear speed up and scale up, but as the number of threads increases past 16, it will taper off and as the problem size increases, the performance time is hindered due to startup costs. There is a slight dip in performance due to hyperthreading around 7 threads since having two threads on one core will still not perform as well as two threads on two different cores. This is because there are times in the program where the available resources in the core will be split between the two threads. Though we would expect the dip in performance to occur at 8 threads, there is actually a dip in performance at 7 threads because you have to take into account the main original thread that's creating the other threads.

Why does the speedup not continue to increase past the number of cores? Does it degrade? Why?

In this case, there are 16 CPU (8 virtual cores and 8 physical cores) with 2 threads per core, 8 cores per socket, and 1 socket. Due to hyperthreading, the instance can support 16 threads, but cannot support anything past that because no more than two threads can run on a core at a time. Otherwise if there were more than 16 threads, then the threads would have to start waiting on each other and wait for a core to be available to run.

There is a slight degrade once the number of threads reaches past 16. One reason for this includes startup costs as more time is taken to create threads. More threads also leads to more memory usage for each thread stack. In addition, if there were more than 16, then the additional threads would have to wait for an available core to execute. As a result, the additional threads would have to wait for the first threads to finish executing and free up a core so that they can start their tasks. The total running time would increase as the threads would have to wait even longer before they can all be joined.

If the program were to be created by using a shared variable, there would be synchronized sections in the code where the local counter for the number of threads is added to the running total of number of heads for each of the threads in the overridden run method. Since the block is synchronized, the hold up will increase as the number of threads increases as each of the threads

would have to wait on each other to finish executing that piece of code before it can run that piece.

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

a. Describe your experiment. Why does it measure startup?

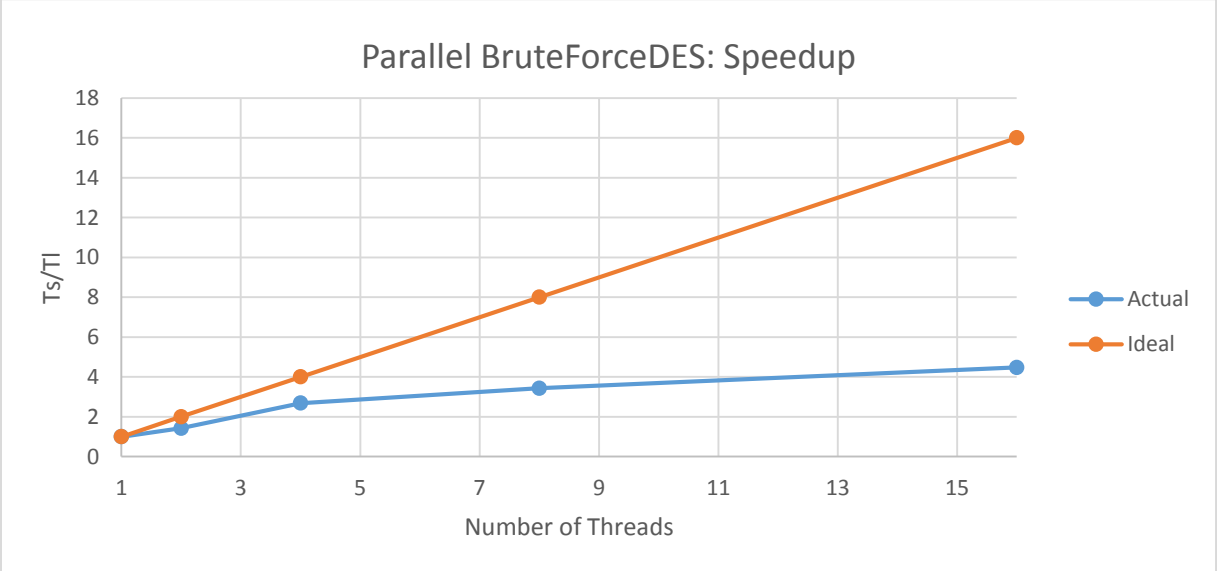
We would want to measure the amount of time the program takes to read in the parameters, convert the parameters into integers and create the lists of Thread and CoinFlip objects, and then start the threads, so all the time through when the program starts the threads. This can be measured by measuring the time (from the start of the main method) the time it takes for 0 flips to be performed. For accuracy, we would run this 20 times and take the average time. Startup costs include all the time that cannot be parallelized, meaning there is no way to break that part of the code into sections and run it concurrently.

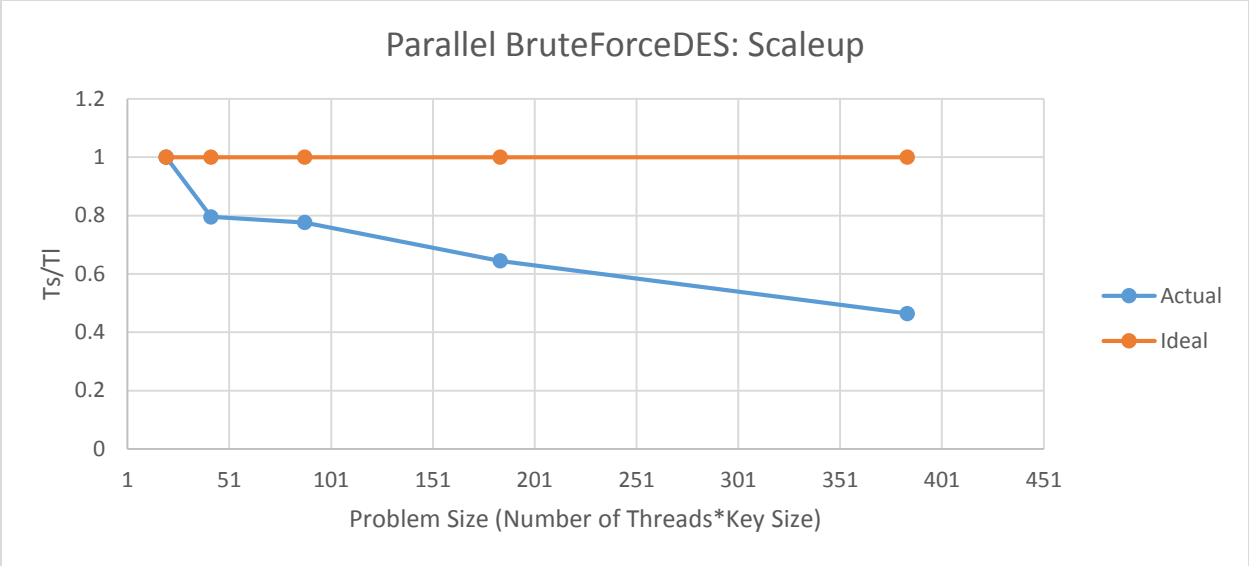
b. Estimate startup cost. Justify your answer.

The startup cost for 1 thread is about 0.156823 ms, 2 threads is about .228633 ms, 4 threads is about 1.907286 ms, 8 threads is about 3.195828 ms, and about 27.070934 ms for 16 threads. This is the average of about 20 trials. We expect a greater startup cost as the number of threads increases, because more time needs to be taken to create and allocate out memory for more threads.

c. Assuming that the startup costs are the serial portion of the code and the remaining time is the parallel portion of the code,what speedup would you expect to realize on 100 threads? 500 threads? 1000 threads? (Use Amdahl's law.)

We see a speed up of 41.85 for 100 threads, 62.48 for 500 threads, and 66.579 for 1000 threads. Using Amdahl's law, we have Speedup = 1/(1-P+P/S), where S is the number of threads and P is the proportion of time that is parallelized. We find the proportion of the time parallelized by taking (runtime for 16 threads – startup time for 16 threads)/runtime for 16 threads is about 0.986, because we know that no matter how many more threads are added, the amount that can be parallelized cannot exceed 16 threads. We use 16 threads because it is the highest number of threads that can run efficiently, otherwise performance lags as the number of threads increases beyond 16 due to the lack of cores.





Produce charts and interpret/describe the results. **Please Use line charts with correctly labelled axes and ticks. Do not use bar graphs or scatter plots.** Is the speedup linear?

We observe a sub-linear speedup for both speedup and scaleup. Up until 4 threads, there is a close trend to linear speedup, but it tapers off and then becomes sublinear after 4. In addition, there is a big decrease in performance as the number of threads increases and the problem size increases.

Why do you think that your scaleup/speedup are less than linear? What are the causes for the loss of parallel efficiency?

We see sublinear scaleup and speedup due to startup costs and hyperthreading. This is because the system does not keep track of which virtual cores share the physical cores and as more virtual cores are used, the higher the chance of sharing with the physical cores will occur. The nonparallel portions of the code has a higher impact as the number of threads and problem size increases and from Amdahl's law, we can see scaleup and speedup have sublinear performance. In addition, thread creation that goes into startup costs plays a larger part in this program than it did for CoinFlip because we are not only creating a new BruteForcesDES object, but also a SealedObject (which also creates a cipher) to pass through to BruteForceDES and in turn, the new thread. Thus, as more threads are created, we do see more parallelism but that is reduced by a higher startup cost. The program creates a new SealedObject to pass onto the BruteForceDES to prevent sharing between the threads. This addresses the issue of interference. The BruteForceDES objects are each given different intervals so that their tasks do not overlap, so skew is reduced.

Extrapolating from your scaleup analysis, how long would it take to brute force a 56 bit DES key on a machine with 64 cores? Explain your answer.

From our scaleup analysis, we can look at the scaleup in 64 cores. We consider the scaleup at 8 threads, where all cores are used and hyperthreading is not an issue. We extrapolate the scaleup to be proportional to where all 64 cores are used, so $S_{64} = S_8$, and we also know that speedup is given by $S_{64} = T_1/T_{64}$. Taking the average change in each scaleup yields a 0.867 change, so the scaleup for 64 threads is extrapolated to be about 0.4203, so the runtime would be about

10332.55 ms for a 26 bit key. This means that 2^{26} keys are checked at a rate of $2^{26}/10332.55$ = 6494.899 keys/ms. Using this same rate, we scale up the problem size to 2^{56} keys (because we're looking for how long it takes to brute force check through a 56 bit key) to result in $(6494.899^{-1}$ ms/keys)*2^{56} keys = 1.109e13 = 352.77 years.