Parallel Programming Assignment 1 Writeup

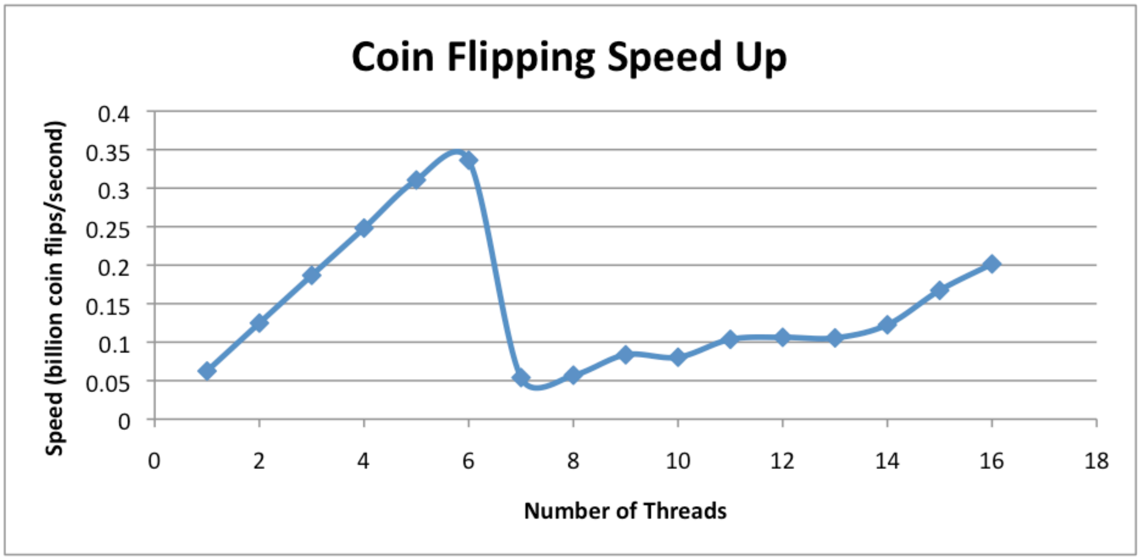
Daniel Crankshaw
9/22/2011

Part 1:

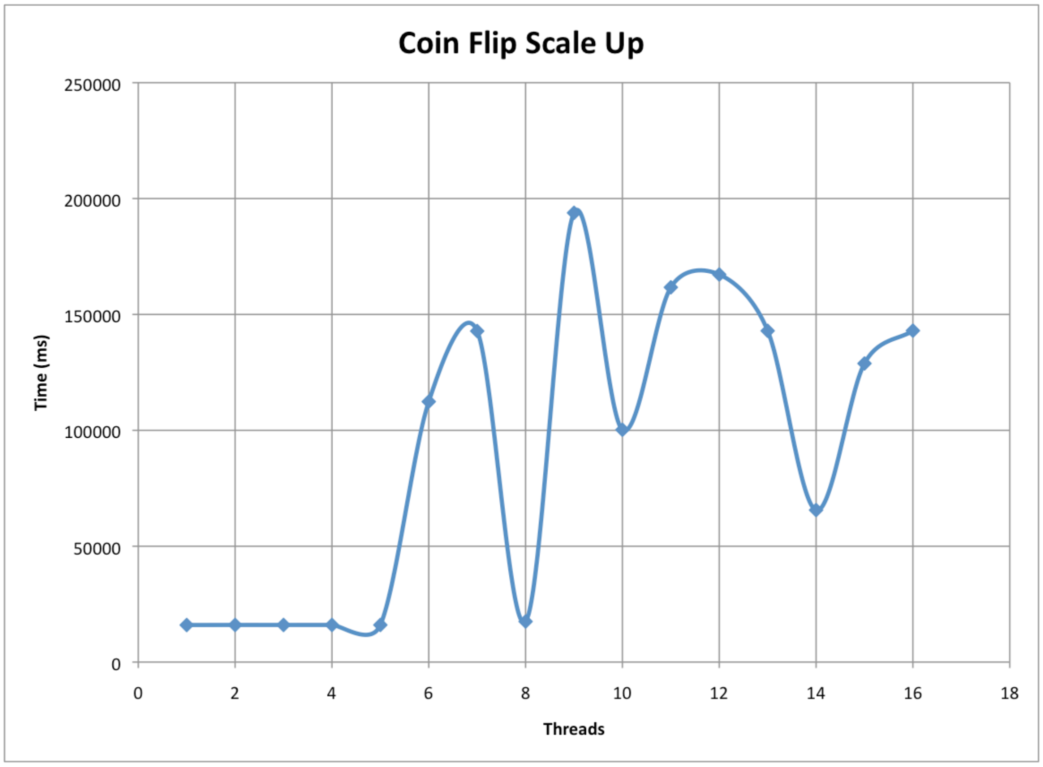
My implementation of part 1 was very simple. I wrote a FlipperRunner class which extends Runnable. The FlipperRunner is given a number of times to flip and coin, and when it's run() method is called, it generates a random int that is either 0 or 1 the specified number of times. The main method creates the number of FlipperRunners the user has specified, and divides the total number of times the coin must be flipped evenly between the FlipperRunners. Once all of the runners have finished flipping, it totals the results and prints them out.

Analysis:

1)
To measure the speedup of the program, I recorded the total execution time to flip 1,000,000,000 coins using one through sixteen threads. The results are graphed below.



To measure the scaleup of the program, I recorded the total execution time to flip 1,000,000,000 per thread (so 1,000,000,000 for 1 thread, 2,000,000,000, for 2 threads, etc). The results are graphed bel



2)

Speed up:

- The speed up of the coin flipping algorithm is relatively linear up through six threads with a best fit line of S = 0.0568N+0.0128 (where N = number of threads and S = speedup). However, the seventh thread seems to produce a lot of extra overhead, and ends up being slower than running the program with just one thread. However, the speed up with threads 7-16 is also pretty linear, although with a much lower slope.

Scale up:

- The scale up of the program is very good up through 5 threads (5 billion flips). However, after that point, there is no clear trend, with the time of execution increasing significantly with 6 and 7 threads, then dropping back down with 8 threads, and basically oscillating after that.

One potential source of performance loss would be that that JVM is running. This basically counts as an extra thread of execution, which would help explain why the speed up doesn't see linear performance up through eight threads. In addition, there are other processes running on the machine (e.g. an ssh session, emacs, bash, a bash script, just to know the ones I as a user initiated), and so if one of the coin flipping thread gets scheduled to a core that is also running one (or several) of these processes, this could slow down that threads execution. And because the time recorded is basically the time for the longest running thread to finish, this would affect my results.

3) Speed up will never exceed the speed up realized by running a single coin flipping thread on each core. Any less threads than that, and there is unused processing power lying around that could be used to be flipping coins. But if we use more threads than there are cores, than multiple threads will have to share a single hardware processing unit. Whether two threads switch off flipping coins every clock cycle, or one thread flips twice as many coins, the processor cannot flip more coins in the same amount of time. However, there is additional overhead associated with an additional core. This overhead is acceptable when the thread will be taking advantage of unused processing power, but not when it has to share a processor with another thread. In fact, this additional overhead causes a degradation in performance because it takes time for a processor to switch between executing instructions between multiple threads (it has to switch register contents, etc.) and there is additional startup and teardown time (serial portions of the code) associated with more threads.

Startup Costs:

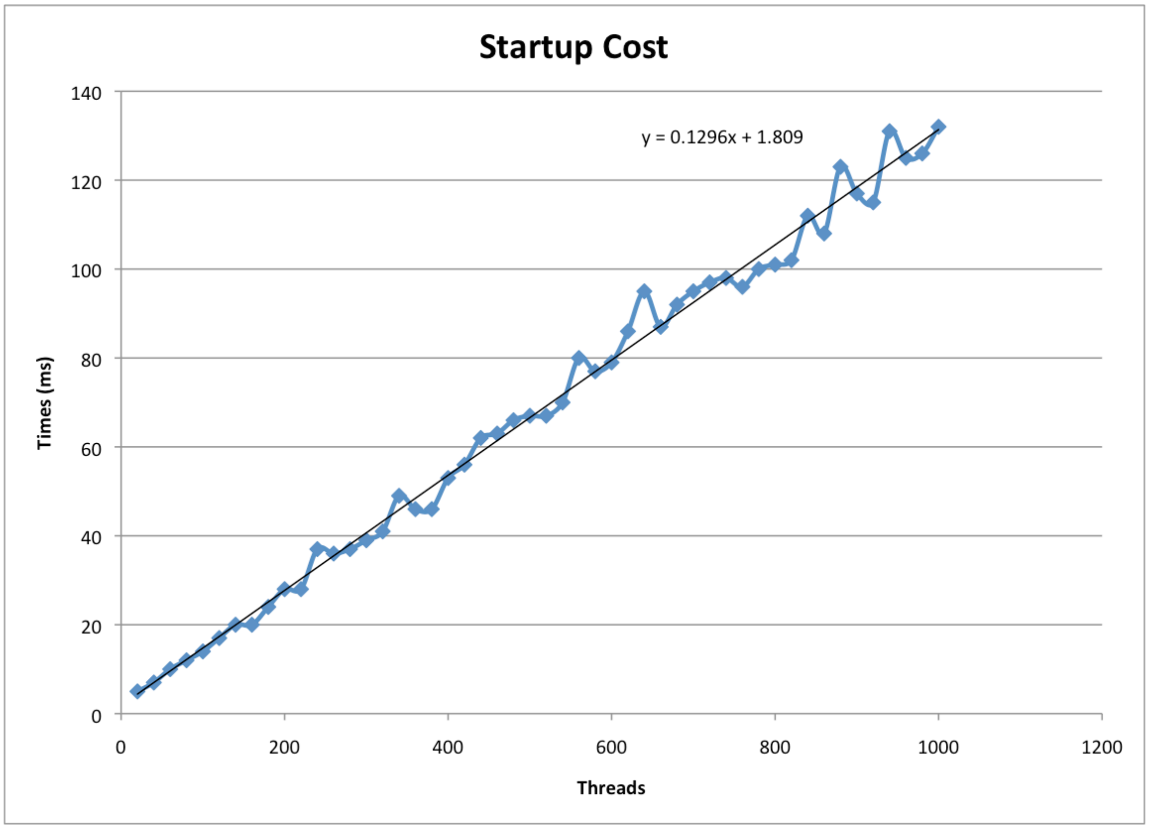
- My experiment runs the coin flipping code but flipping 0 coins for a varying number of threads. The actual coin flipping portion of the code is the parallel part. In essence, the program is generating more than one random number at a time (at least while there are still cores available), which makes the generation of a set total of random numbers go faster. Everything else, such as creating new Thread objects, reading and processing command line arguments, and fetching the number of heads flipped from each Runnable object after the thread has been joined, is all done in serial. Therefore, none of the actual parallel code runs, and I am timing just the serial portion.

- I estimate the startup cost to be 1.81ms + 0.13ms per thread. This is based on the graph of time vs. number of threads when flipping 0 coins. It makes sense that the y intercept of this equation is greater than 0. Even if no threads were being created, the program would still not execute instantaneously, because thread creation is not the only serial portion of the code. It must also parse command line arguments, allocate two arrays, and perform several other operations that are independent of the number of threads being created. The 0.13ms associated with thread creation is the part of the code that contains operations that must be done for each thread created.

- Assuming that each thread has it's own core to run on (e.g. the 500 thread version runs on a 500 core machine), I derived my estimates in the following way. Amdahl's law says that speedup for a program where P percent of the program is parallelizable running on N cores is:

$$S = 1 / [1 - P + P/N]$$

From my measurements of speedup (where each thread has it's own core) $S = 0.0568N+0.0128$. However, the situation is a little more complicated because the serial portion of the code increases as a function of N. So to estimate parallel portion, I am going to take the average value of P for N = 1 through 16 by subtracting the startup costs from the total execution time and dividing that by the total execution time. This yields P = 0.999. From Amdahl's law, the expected speed up of running this on 100 cores would be S = 91. For 500 cores S 333.5 For 1000 cores S = 500.25.

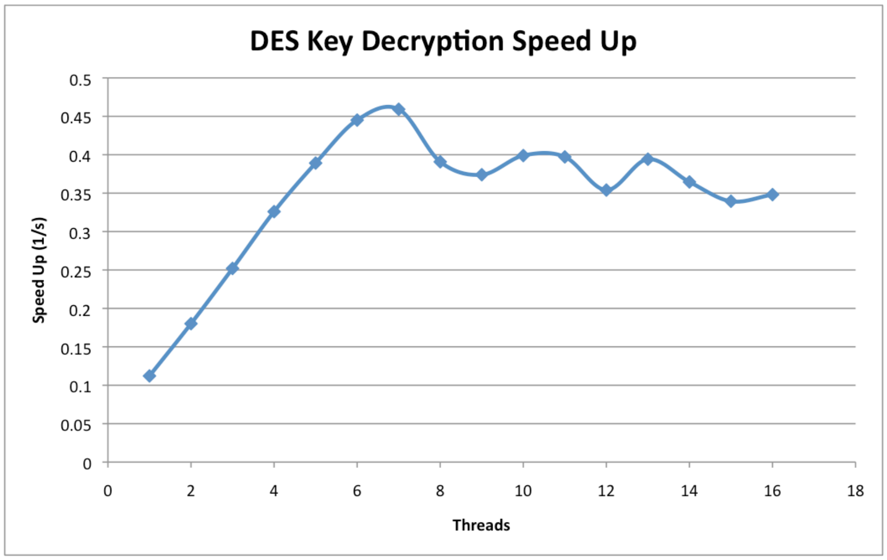


Part 2:

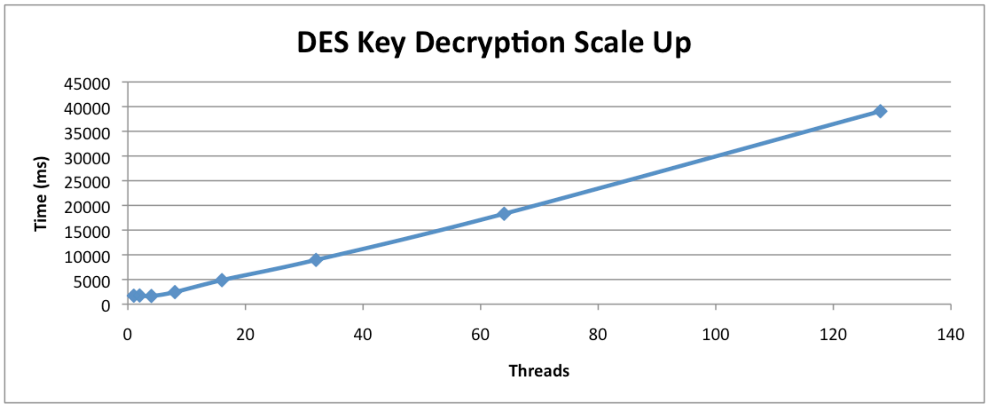
The code for part 2 very closely mirrors the serial implementation of the brute force decryption. All the code does is divide the key space among the number of threads and pass each thread a SealedDES object and a copy of the encrypted message. Each thread then runs through its set of keys, checking to see if any of them have successfully decrypted the message.

Analysis:

1)
Below is the graph of DES key decryption speed up vs number of threads for a key 20 bits long. The graph is linear for threads 1-7, but then the speed up basically levels off and starts to actually degrade a little bit.



Below is a graph of DES key decryption time vs number of threads such that every thread is always decrypting a key space of the same size. So when there are 4 times as many thread, the key space is 2 bits longer. Each thread is responsible for a keyspace of size 2^17. To see linear scale up, this line would have to be horizontal. This graph implies that additional threads actually cause an increase in the amount of time a single thread can process a key space of a fixed size. However, the scale up is better with a smaller numbers of threads, 1-4 threads have a roughly linear scale up (their times are all within 100ms of each other).



2) As I discussed in part 1, the losses of parallel efficiency when there are more threads than cores have to do with the startup costs of more threads, and the costs of a processor switching processing between multiple threads. The reasons for the loss of efficiency with the DES key decryption are similar to those of the coin flipping program.

3) If we only examine the portion of the scale up graph where there is less than or equal to one thread per core, we find that one thread can process a keyspace of size 2^17 in about 1700ms. This comes out to 2^17/(1.7) keys per second per core. So 64 cores can process 64*2^17/1.7 keys/s or 2^23/1.7 keys/s. The key space of a 56 bit key is 2^56, so the time to brute force a 56 bit key:

T = 2^56/(2^23/1.7) s
   = 1.7(2^(56-23)) s
   = 1.7*(2^33) s
   = 1.46*10^10 seconds
   = 462.75 years or 462 years and 9 months.