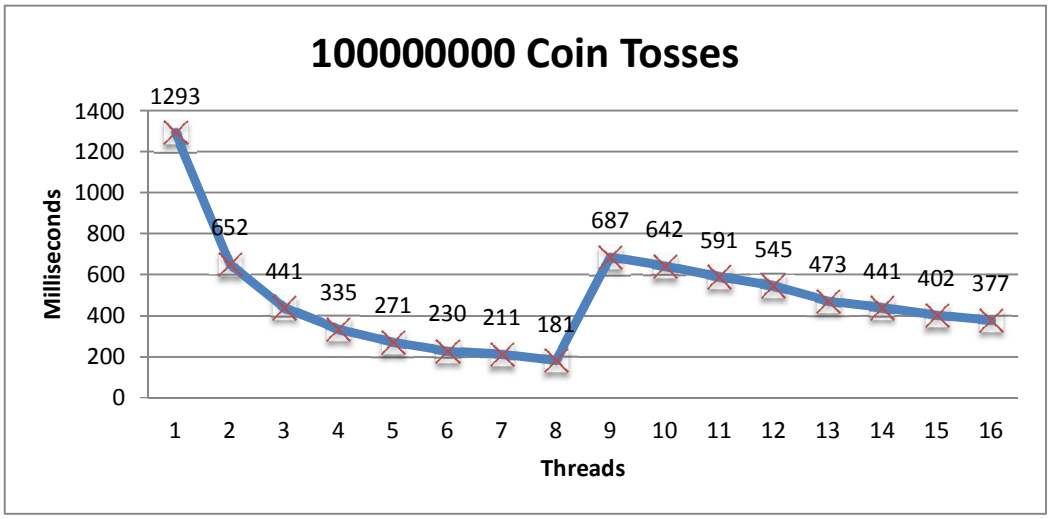
Parallel Programming
HW2

CHANG LIU
chang.liu@jhu.edu

Coin Flip

All the experiments are running on the **Basic Amazon Linux AMI 2011.09** with m2.4xlarge type (8 cores with 64GB memory). The result of 100000000 coin tosses is showed as below:



Time of coin tosses

Scaleup and speedup

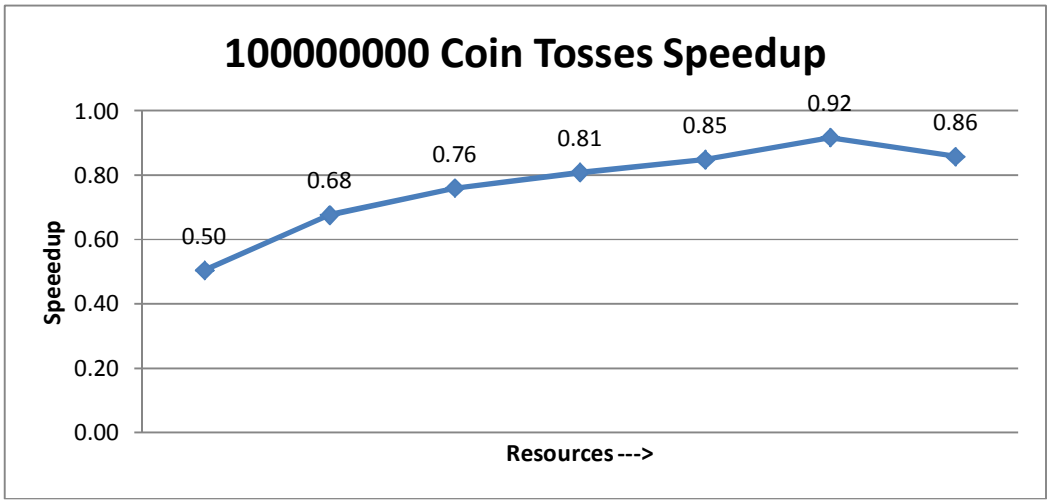
I. Produce charts that show the scaleup and speedup of your program.

1. Speedup

T_S=time to execute task on small machine

T_L=time to execute task on large machine

Thus we have speedup = T_S / T_L



Speedup Chart: From threads 1 to 8

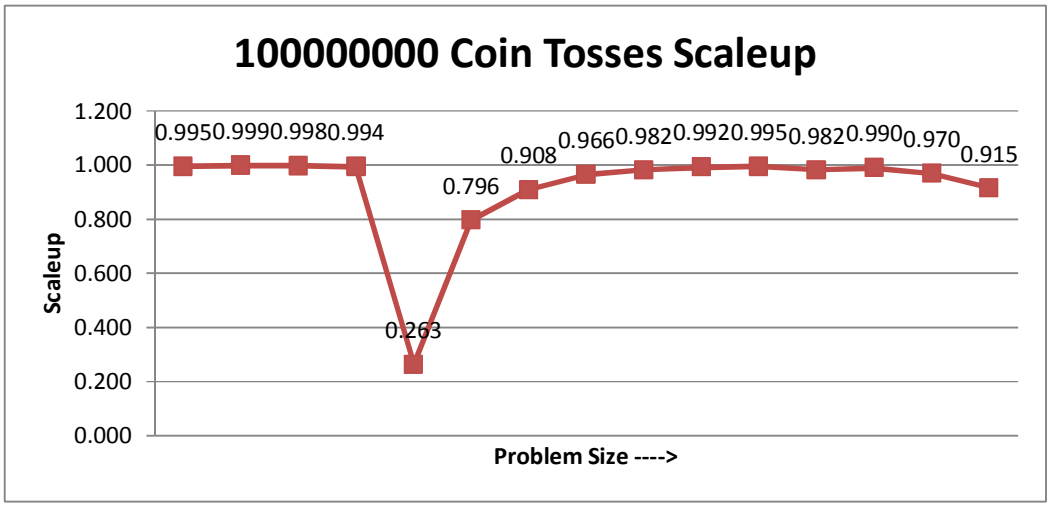
Speedup Chart: From threads 9 to 16

2. Scaleup

T_S=time to execute task on small machine

T_L=time to execute N times bigger task on N times larger machine

Thus we have scaledup = T_S / T_L



Scaleup Chart: From threads 1 to 16

II. 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.

From the chart we could know that the speedup/scaleup is sub-linear. The potential source of loss may be the following: 1) there's startup cost in the serial part of the code including fixed startup cost shared by all the threads and startup cost per threads. 2) For the parallel part of the program, the time it cost to executing the coin slipping may not be linear when the threads are more than the number of cores, for instance, when there are 16 threads on a 8 cores computer, half of the threads have to wait the free resource being released.

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

The speedup of the program does not always increase and even degrades when past the number of cores (in this case 8 cores). This is because when the threads are more than 8, the additional threads have to wait for the computer resources to execute as there're only 8 cores on the computer, which means only 8 threads could execute at the same time. In this case, the additional threads have to wait until at least one of the first 8 threads complete processing and release the available resources. That's why we have a "gap" in the running time (see the first figure) when thread shift from 8 to 9, the running time for 9 threads on 8 cores machine is much longer than 8 threads.

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

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

The total running time of the program may be divided into the following 3 parts:

$$TotalTime = FixedStartupCost + ThreadsStartupCost(threads) + ProcesingCost(input)$$

Thus for the same number of threads, as the FixedStartupCost remains the same, we could apply different input (total number of tossing), thus get the ProcessingCost for the same thread but different inputs:
Given the ThreadsStartupCost(threads) is the same,

$$\Delta TotalTime = ProcesingCost(input2) - ProcesingCost(input1)$$

In addition, as the processing time takes input as the only factor, we can guess the ProcessingCost has a linear relation with the input size.

To verify our guess, we can make an experiment that run the program for 1, 2, 3 threads at 10000000, 20000000, 30000000, 40000000, 50000000 and 100000000, 200000000, 300000000, 400000000, 500000000. The result is showed as following:

Milliseconds		**Tosses**									
		$1x10^7$	$2x10^7$	$3x10^7$	$4x10^7$	$5x10^7$	$1x10^8$	$2x10^8$	$3x10^8$	$4x10^8$	$5x10^8$
Threads	1	138	266	395	523	651	1293	2576	3863	5145	6426
	2	77	142	206	267	335	652	1297	1940	2584	3224
	3	55	98	141	184	227	441	869	1297	1728	2160

The above table proves our guess is true: there's a linear relationship between total number of input and processing. For example, when there is only 1 (or 2 or 3) thread, the running time difference between $1x10^7$ and $2x10^7$ is the same to the difference between $1x10^8$ and $2x10^8$, that means:

$$\Delta TotalTime$$
$$= ProcesingCost(2\times10^7) - ProcesingCost(1\times10^7)$$
$$= ProcesingCost(3\times10^7) - ProcesingCost(2\times10^7)$$

Thus we know the

$$\Delta TotalTime$$
$$= ProcesingCost(input2) - ProcesingCost(input1)$$
$$= ProcesingCost(input2 - input1)$$

Use the above formula we have:

$$TotalStartupCost = FixedStartupCost + ThreadStartupCost$$
$$= Time - ProcessingCost$$

For only 1 thread and input1 = $1x10^7$, input2 = $2x10^7$, we have:

$$TotalStartupCost = FixedStartupCost + ThreadStartupCost$$
$$= Time - ProcessingCost$$
$$= 138 - (266 - 138)$$
$$= 10$$

Thus, we could calculate the average startup cost when there are 1, 2, 3 threads. The result is showed as following:

Threads	1	2	3
Avg. Startup Cost	7.89	11.19	14.54

We can further calculate the **per thread startup cost** by using the Avg. Startup Cost of 2 - Avg. Startup Cost of 1 = 11.19 – 7.89 = 14.54 – 11.19 = 3.3 milliseconds

Then we can further calculate the **fixed startup costs** = 7.89 – 3.3 = 4.59 = 11.19 – 3.3*3 = 4.59

II. Estimate startup cost. Justify your answer.

From above problem we had already calculate and justified the startup cost:
Per Thread Startup Cost: 3.3 milliseconds
Fixed Startup Cost: 4.59 milliseconds

III. 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.)

From the Amdahl's law, we have $\frac{1}{1-P-\frac{P}{S}}$.

Also from the 2 tables we just created above, we have:

When the total tosses is $1x10^8$, total time = 1293, startup cost = 7.89, thus:

$$P = \frac{(1293 - 7.89)}{1293} = 0.9938$$

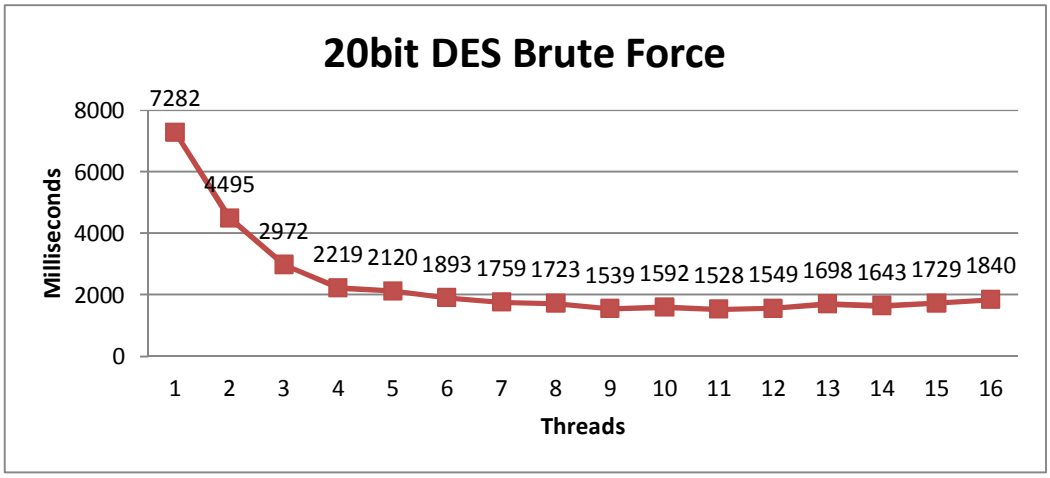
100 threads: $Speedup = \frac{1}{1 - 0.9938 + \frac{0.9938}{100}} = 61.966$

500 threads: $Speedup = \frac{1}{1 - 0.9938 + \frac{0.9938}{500}} = 122.136$

1000 threads: $Speedup = \frac{1}{1 - 0.9938 + \frac{0.9938}{1000}} = 139.008$

Brute Force a DES Key

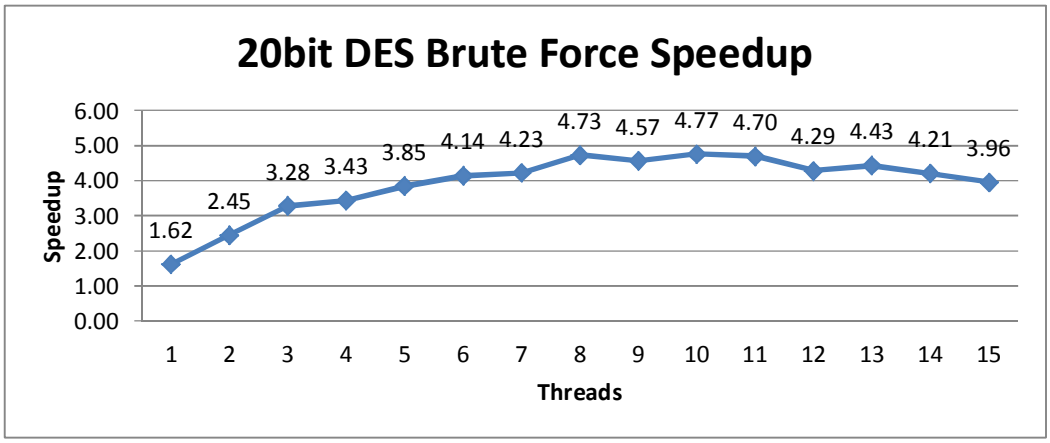
For reasonable parameters and for however many cores you have on the system, measure the scaleup and speedup of this program on the cluster.



Brute Force 20bit DES on 8 cores system with threads from 1 to 16

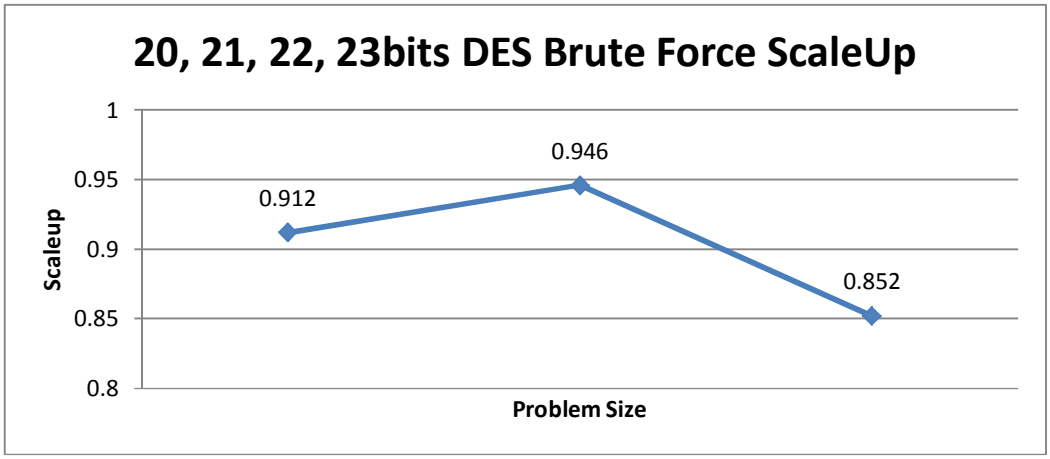
I. **Produce charts and interpret/describe the results. Is the speedup linear?**

The speed up is sublinear, as showed in the chart below:



The speedup of 20bit DES brute force from thread 1 to 16

The scale up is sublinear, as showed in the chart below:



The scaleup of 20, 21, 22, 23 bits DES brute force from thread 1, 2, 4, 8

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

From the chart we could know that the speedup/scaleup is sub-linear. The potential source of loss may be the following: 1) there's startup cost in the serial part of the code including fixed startup cost shared by all the threads and startup cost per threads. 2) For the parallel part of the program, the time it cost to executing the brute force for each thread may not be linear when the threads are more than the number of cores, for instance, when there are 16 threads on a 8 cores computer, half of the threads have to wait the free resource being released. Thus less efficiency.

III. 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.

For 56 bit DES key and a machine with 64 cores, the workload for each core is:

$2^{56} \div 64 = 2^{50}$

Thus we know each core need to take a 50 bit DES work.

Also, when we apply 20 bit DES brute force on a single thread (core), the time it takes to find the key is 7282s.

Thus for a 50 bit DES key, the number we need to test is $2^{50} \div 2^{20} = 2^{30}$ larger than the 20 bit DES key. Thus the average time it may take to find the key is also 2^{30} longer than the 20 bit key.

So we can measure the total time it takes to brute force a 56 bit DES key:

$2^{30} \times 7282$ Millisecond.