

CRACKING DES WITH MPI

A research paper by Hunter E. Rick

Final program questions and answers

1.

Final program description

This program is a test of what I can do with the processing power of 240 cores using mpi in order to crack the 56 bit DES key. I will attempt to iterate through each key size and track speed up at each size until this isn't viable because of time constraints.

The value of your solution

"Because the Key length for DES is only 56-bit, it is considered that even unspecialized computer hardware can break DES-encrypted content in less than two days. Hence, you are recommended to remove this setting if present" -

<https://docs.microsoft.com/en-us/services-hub/health/remediation-steps-ad/remove-the-highly-insecure-des-encryption-from-user-accounts>

As stated in the above article by microsoft, 56 bits is no longer a secure key length well since the 90's. If a 24 year old undergrad can get anywhere near cracking this with just 240 cores and MPI, then this is as good as proof that this problem will only become exacerbated every year with larger amounts of processing power becoming more feasible. In short, I am exposing a vulnerability on out of date encryption methods.

What numerical methods and algorithms are used and what type of math is required?

I used knowledge of permutations and bit representation from my discrete math class. Permutations especially helped me understand how to go about implementing my parallel solution as I had to think carefully about how I could split up the permutations without having a race condition and skipping combinations while still achieving optimum speed up. As a side note I also taught myself how to do extrapolation on google sheets, as I had never learned this math technique before.

What parallel programming methods are used?

MPI

2.

Example of Sequential program running and time being averaged. All runs are contained in the appendix as they are too many to include in a 2-3 page summary.

Key length = 2 bytes / 16 bits

```
hunter@hunter-desktop:~/CSCI551/finalProject$ ./main 2
?? = ??
program took 0.002952 seconds
hunter@hunter-desktop:~/CSCI551/finalProject$ ./main 2
?? = ??
program took 0.002949 seconds
hunter@hunter-desktop:~/CSCI551/finalProject$ ./main 2
?? = ??
program took 0.002873 seconds
hunter@hunter-desktop:~/CSCI551/finalProject$ ./main 2
?? = ??
program took 0.002859 seconds
```

Average runtime:

```
⌚ (0.002952 + 0.002949 + 0.002873 + 0.002859) / 4 =
0.00290825
```

Example of Parallel program runs and time being averaged. All runs are contained in the appendix as they are too many to include in a 2-3 page summary.

Key length = 16 bits/2 bytes

```
hunter@hunter-desktop:~/CSCI551/finalProject/finalAssignmentMPI/worstCase$ cat worst2-*
?? = ??
program took 0.000203 seconds
?? = ??
program took 0.000222 seconds
?? = ??
program took 0.000215 seconds
?? = ??
program took 0.000079 seconds
```

Average runtime:

```
⌚ (0.000203 + 0.000222 + 0.000215 + 0.000079) / 4 =
0.00017975
```

3.

Note: All calculations shown here are for 4 bytes worst case, for all bytes data look in appendix.

Sequential Portion

1.247024918% Sequential. I found this by subtracting $100 - 98.752975082$.

Key length 4 bytes Worst Case:

$$33.0447355 / .5480455 = 60.2956059305$$

Parallel Portion

$$(S * (1 - (1 / SU))) / (S - 1) = p$$

$$(240 * (1 - (1 / 60.2956059305))) / (240 - 1) = \mathbf{98.752975082\% \text{ Parallel}}$$

Number of shared Cores used

240. This is the number of total cores the 60 working cluster nodes provide. There are 4 cores on each for a total of $60 * 40 = 240$.

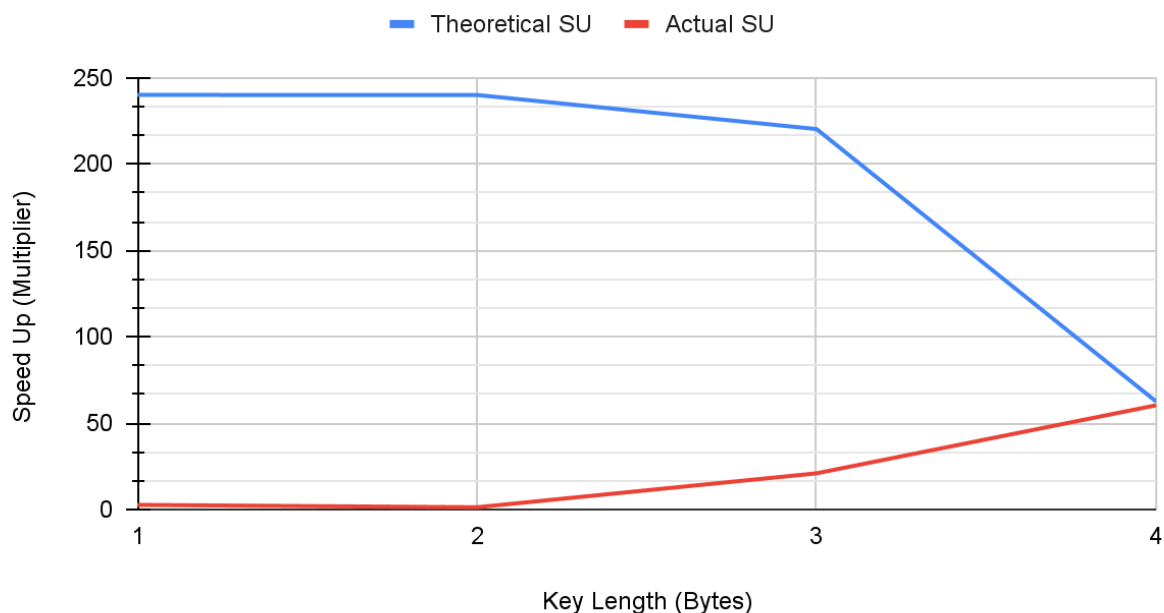
Number of nodes used

60. This is simply the total number of working nodes I could use for my project.

Final Value used for S

240. I used 240 because it is simply the amount of cores I used to split up the work.

Theoretical SU vs Actual SU



As I suspected earlier, Gustafson's law is at work here. Only once the data and runtime became big enough did the expected speed up and actual speed up start to converge. I suspect if I were to continue charting the key sizes they would remain pretty similar in speed up.

APPENDIX

DESIGN APPROACH

For my program I looked into various ways of generating all possible combinations of a set. There were many interesting tricks with bit masking algorithms, however they all relied on the idea that the user wanted to find a subset of a larger set.

To crack DES, the entire set of ascii characters had to be considered, and so I found no quicker algorithm than to simply just use nested for loops. Brute strength was my best route. It is important to note that the machine nicknamed Deep Crack was brute strength in nature although its architecture was more complicated than my software. Utilizing 1856 custom built ASIC DES chips, they were able to decipher and label interesting plain text from decryptions before even passing any keys to the software. This meant they could cut down on the amount of keys to try by billions.

This approach is simply out of my reach so I decided to do the next best thing and focus all 240 cores available to me into testing the keys based on a known randomly generated key, as well as a worst case scenario key of all 256 ASCII value characters (in binary that'd be 11111111 11111111 11111111 11111111 11111111 11111111 11111111). I would then see if I could create speed up in these two possible situations. Actually deciphering each key and checking for recognizable plain text would simply be too much overhead for the amount of processing power I have, and would make even a 32 bit key very slow.

There are 256 possible symbols that can be represented by an 8 bit binary combination. These make up the 256 different ascii characters a DES key can be made up of. I verified this with a bit of math. A DES key is 56 bits (ignoring the 8 bit parity). This is a total combination of 2^{56} or $7.2057594e+16$. If we create a 7 byte long string from any possible ascii combination, that is 256^7 or $7.2057594e+16$ as well. It is important to note that not all ascii characters are printable, and control code's like ctrl+z and NULL terminators are part of this representation. These need to be included as a DES key is technically in the form of bits, but can represent these unprintable ascii values as well. Without including control codes, not all possible bit combinations would be represented.

I also played with the idea of using infinite while (while(1)) loops with a break statement, but after reading over information on run time speed of the two loops, I found that for my situation, there would be no difference in speed. Both loops would have to do an evaluation each loop, and a second evaluation inside the loop.

I also had to make a design choice of what kind of values to use to represent my ASCII characters. The options being decimal, binary, octal, and hexadecimal. I ended up taking the hexadecimal approach for two reasons. The first one is that the C programming language can naturally store hexadecimal values as their associated characters, and the second is readability. I found decimal and octal too hard to keep track of for what I was doing. In binary's case, the C programming language must

convert binary representations of ascii to hex or decimal first before it recognizes it as a character. This would create unnecessary overhead, which is extremely important to me to avoid because runtime speed will make or break this program.

Another important factor was I had to look into which data type ran faster for my program, signed or unsigned char. After reading up a bit, I found that unsigned was significantly quicker because it allows the compiler to assume that overflows won't happen and it does not need any bit shifting. When a signed data type passes its upper bound, it must shift bits to begin representing negatives.

Finally, I needed to decide on my method of comparison between the brute force guess and the actual key. I decided on C's `strcmp()` function. This was a quick and easy choice as I looked at the runtime complexity of several approaches to comparing strings in C, and none came close to the speed of `strcmp` which is bounded $O(n)$ to the smaller strings length in constant time. Not bad.

Now that I had finished researching DES and decided on a basic algorithm, I had to worry about how I was going to go about generating my DES key's. For this I chose openssl, and generated a key inside the program with a system call to "`openssl rand <key length> > DES.key`". This command creates a DES key of specified length and pipes it into a file called DES.key. My program would then read from that file and store the key as a c string. In many implementations of C programming with openssl, I saw people using the openssl C library, but unfortunately this module is not available to students on ECC-linux and so I chose the command line route as a work around. Due to this added overhead, I decided not to have my timer start until after the key generation as it's not what we are interested in speeding up.

Everything necessary for a sequential program had fallen into place at this point. Now to integrate parallelism with mpi. At first I could not decide which loops to make parallel in my program, only to realize that no matter how you split the nested for loops, the run time is exactly the same. This led me to decide on the simplest readable approach, completely parallelizing the outer loop with 240 cores. This meant that the original 256^7 total keys could now be seen as $256^6 * 1.06666666667$. This is because you take the outer loop of 256, divide it by 240 and chop off one exponential power to get your new loop total. Essentially, if a loop ran in parallel, it could be counted as 1 loop instead, and I changed the outer loop to 1.06666666667 loops. That decimal is a representation of a residual of 16 loops.

At first I was handing the 16 residual loops off to the last rank. This was a mistake on my part because I realized that meant a single core had to run an extra $256^6 * 16$ loops than every other core. I realized that if I took that 16 and split it further into 16 different ranks at the end, then even in worst case runtimes where the key reaches the end of the iterations, that's only $256^6 * 2$ extra loops now. (The last 16 cores each are running an extra outer loop that the other previous cores were not).

After I had optimized my mpi's split and algorithm the only thing left was to use the -o3 optimization tag when compiling.

One final thing to note is that all sequential programs were run natively on my pc and so the single core comparison here is based on my desktop's type of core and not ecc-linux's.

IMPORTANT CONTEXT TO HOW MY TESTING WORKED

To reiterate and expand on some of my earlier statements, testing was done with two scenarios. The first scenario is a key with entirely 1 binary values or 256 ASCII values. This is the Worst Case scenario.

The second scenario is a randomly generated key, using openssl. The second scenario is more practical and comparable to a real life example. This is the Average Case scenario. We can expect to see worse times from both the sequential and parallel programs in the worst case scenario.


The Average case scenario will have much larger variation due to its random nature. Worsening this variation in results is the fact that with my time constraints I decided to only use a sample size of 4 runs per key length to get an average runtime. This sample size would have to be taken into account when trying to observe confidence intervals for whether or not my answers are within a significant range from the true mean.

SEQUENTIAL PROGRAM WORST CASE RUNTIME SPEED

Key length = 1 byte / 8 bits

```
hunter@hunter-desktop:~/CSCI551/finalProject$ ./main 1
? = ?
program took 0.000230 seconds
hunter@hunter-desktop:~/CSCI551/finalProject$ ./main 1
? = ?
program took 0.000226 seconds
hunter@hunter-desktop:~/CSCI551/finalProject$ ./main 1
? = ?
program took 0.000219 seconds
hunter@hunter-desktop:~/CSCI551/finalProject$ ./main 1
? = ?
program took 0.000223 seconds
```

Average runtime:

 $(0.000230 + 0.000226 + 0.000219 + 0.000223) / 4 =$
0.0002245

Key length = 2 bytes / 16 bits

```
hunter@hunter-desktop:~/CSCI551/finalProject$ ./main 2
?? = ??
program took 0.002952 seconds
hunter@hunter-desktop:~/CSCI551/finalProject$ ./main 2
?? = ??
program took 0.002949 seconds
hunter@hunter-desktop:~/CSCI551/finalProject$ ./main 2
?? = ??
program took 0.002873 seconds
hunter@hunter-desktop:~/CSCI551/finalProject$ ./main 2
?? = ??
program took 0.002859 seconds
```

Average runtime:



$(0.002952 + 0.002949 + 0.002873 + 0.002859) / 4 =$

0.00290825

Key length = 3 bytes / 24 bits

```
hunter@hunter-desktop:~/CSCI551/finalProject$ ./main 3
??? = ???
program took 0.131241 seconds
hunter@hunter-desktop:~/CSCI551/finalProject$ ./main 3
??? = ???
program took 0.131390 seconds
hunter@hunter-desktop:~/CSCI551/finalProject$ ./main 3
??? = ???
program took 0.134760 seconds
hunter@hunter-desktop:~/CSCI551/finalProject$ ./main 3
??? = ???
program took 0.131438 seconds
```

Average runtime:



$(0.131241 + 0.131390 + 0.134760 + 0.131438) / 4 =$

0.13220725

Key length = 4 bytes / 32 bits

```
hunter@hunter-desktop:~/CSCI551/finalProject$ ./main 4
0000 = 0000
program took 32.823398 seconds
hunter@hunter-desktop:~/CSCI551/finalProject$ ./main 4
0000 = 0000
program took 32.791241 seconds
hunter@hunter-desktop:~/CSCI551/finalProject$ ./main 4
0000 = 0000
program took 32.780472 seconds
hunter@hunter-desktop:~/CSCI551/finalProject$ ./main 4
0000 = 0000
program took 33.783831 seconds
```

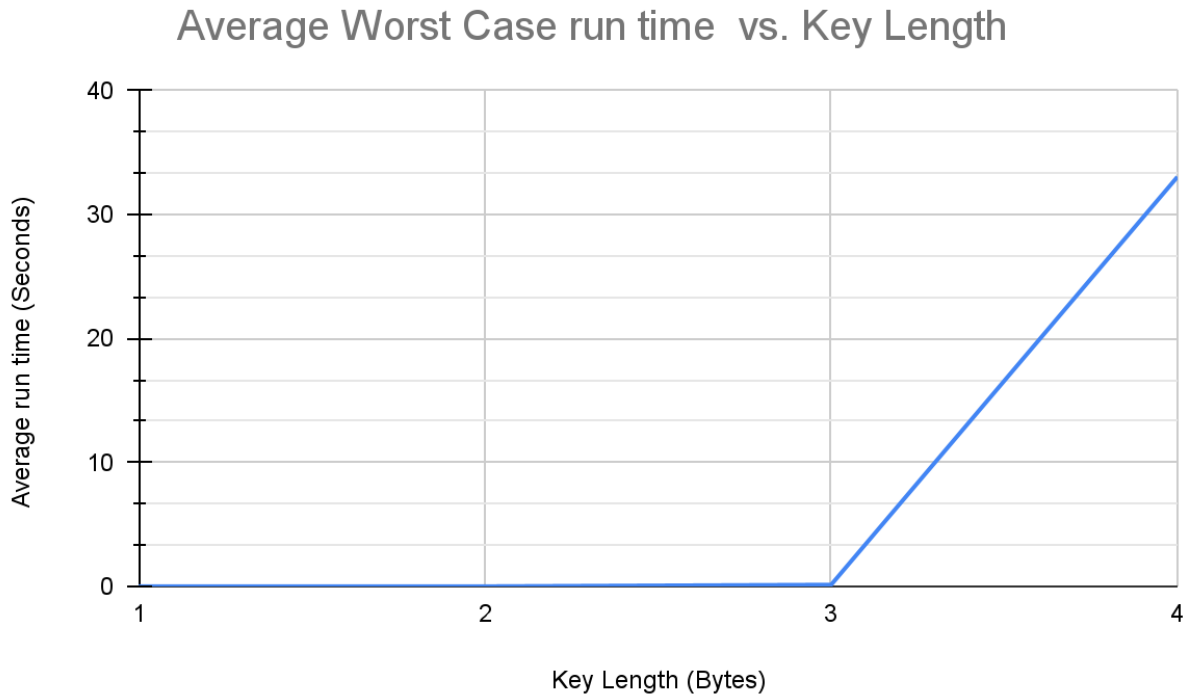
Average runtime:



$(32.823398 + 32.791241 + 32.780472 + 33.783831) / 4 =$

33.0447355

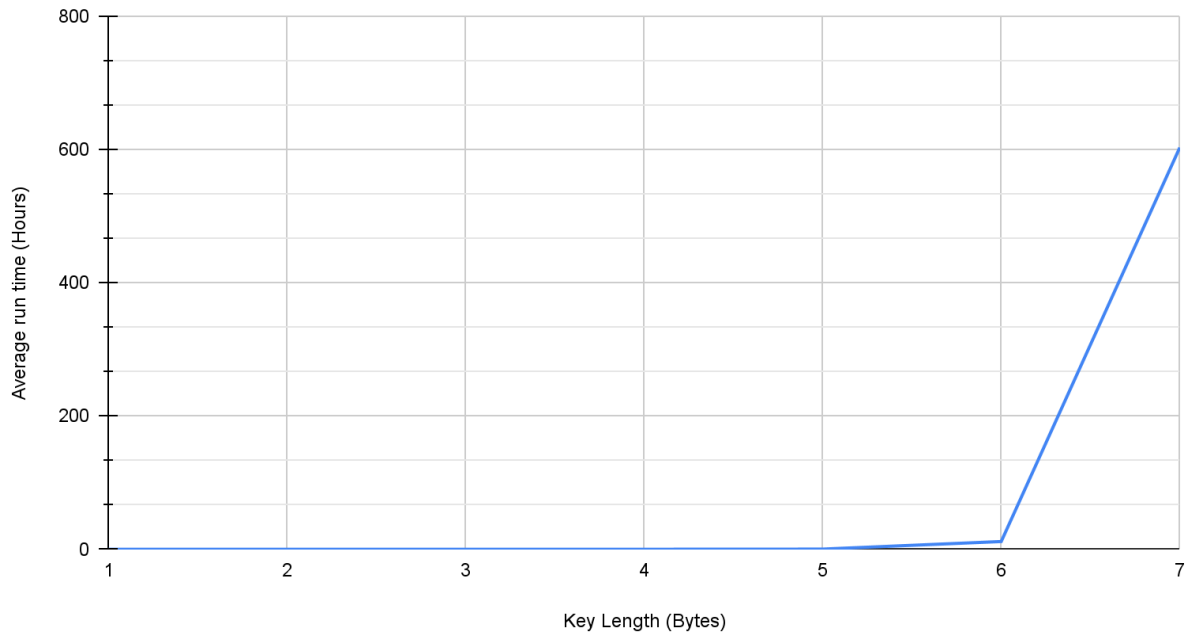
KNOWN VALUES WORST CASE RUNTIME CHART



As you can see, the sequential runtime of each key length increase could get bad pretty fast. It was bad enough that I only opted to do 4 runs of each key length up to 4 and find their average, because past 4 bytes, the runtime starts taking upwards of 20 minutes. Instead of running tests on 5 and up, I extrapolated from the previous 4 average run times, to get 5, 6 and 7's average run time in the next chart.

EXTRAPOLATED WORST CASE RUNTIME CHART

Average worst case run time vs. Key Length (5 and up extrapolated)



Extrapolation for values of 5 bytes and up was done using google sheet's GROWTH function, which takes known x and y values, and creates an output for a new given x value.

As the huge gap in the chart makes it hard to read, the values for 5, 6, and 7, were 0.2227507846, 11.58636955, and 602.6643614 hours respectively. The last value comes out to be about 25.11 days. This further confirms my analyses of the known values in the chart before this one. The runtime sequentially of a worst case scenario is technically viable, but a bit insane. Not many people would be willing to wait 25 days for the results. Note that the previous chart was in seconds, but hours was a better representation of the higher byte sizes.

SEQUENTIAL PROGRAM AVERAGE CASE RUNTIME SPEED

Key length = 1 byte / 8 bits

```
hunter@hunter-desktop:~/CSCI551/finalProject/avgKeyTime$ cat key1*
@ = @
program took 0.000004 seconds
8 = 8
program took 0.000003 seconds
=
program took 0.000003 seconds
e = e
program took 0.000005 seconds
```

AVERAGE RUNTIME



$(0.000004 + 0.000003 + 0.000003 + 0.000005) / 4 =$

0.00000375

Key length = 2 bytes / 16 bits

```
hunter@hunter-desktop:~/CSCI551/finalProject/avgKeyTime$ cat key2*
@/ = @/
program took 0.000400 seconds
-e = -e
program took 0.000090 seconds
mw = mw
program took 0.000217 seconds
@ = @
program took 0.000494 seconds
```

AVERAGE RUNTIME



$(0.000400 + 0.000090 + 0.000217 + 0.000494) / 4 =$

0.00030025

Key length = 3 bytes / 24 bits

```
hunter@hunter-desktop:~/CSCI551/finalProject/avgKeyTime$ cat key3*  
gg = gg  
program took 0.086827 seconds  
< = <  
program took 0.031124 seconds  
@` = @`  
program took 0.100121 seconds  
s/ = s/  
program took 0.089093 seconds
```

AVERAGE RUNTIME



$(0.086827 + 0.031124 + 0.100121 + 0.089093) / 4 =$

0.07679125

Key length = 4 bytes / 32 bits

```
hunter@hunter-desktop:~/CSCI551/finalProject/avgKeyTime$ cat key4*  
0iD = 0iD  
program took 24.984895 seconds  
T0 = T0  
program took 26.171384 seconds  
T = T  
program took 10.747340 seconds  
u0 = u0  
program took 26.723816 seconds
```

AVERAGE RUNTIME



$(24.984895 + 26.171384 + 10.747340 + 26.723816) / 4 =$

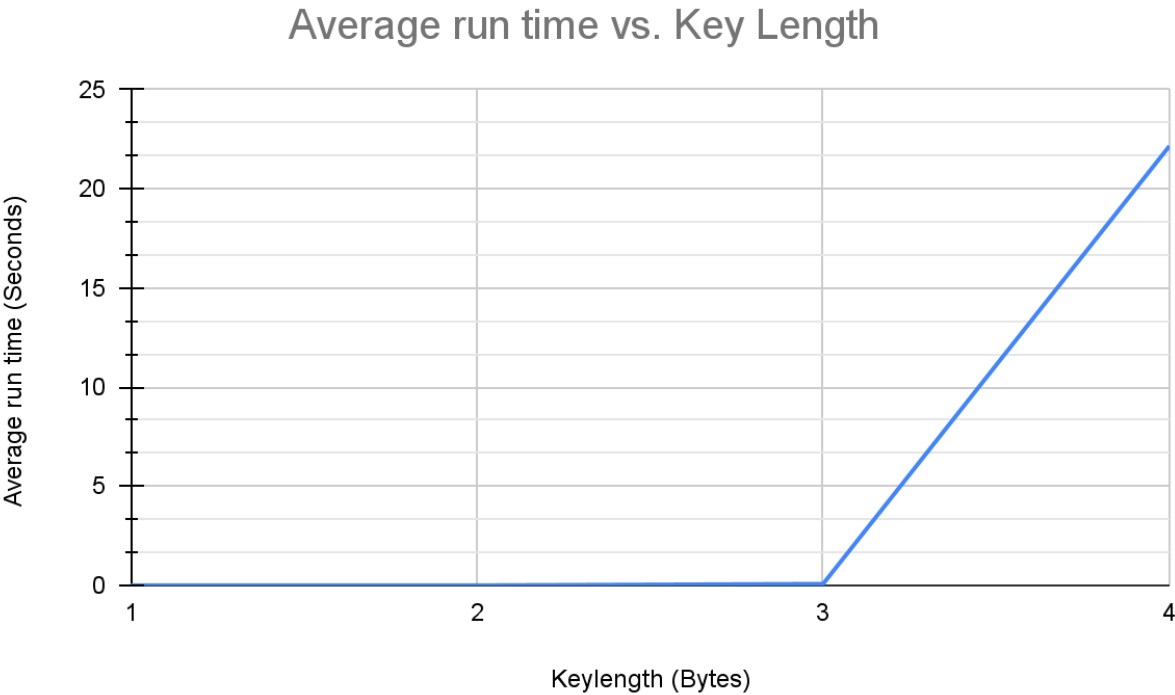
22.15685875

Key length = 5 bytes / 40 bits

```
hunter@hunter-desktop:~/CSCI551/finalProject/avgKeyTime$ cat key5-1.txt  
c#L0| = c#L0|  
program took 3352.590851 seconds
```

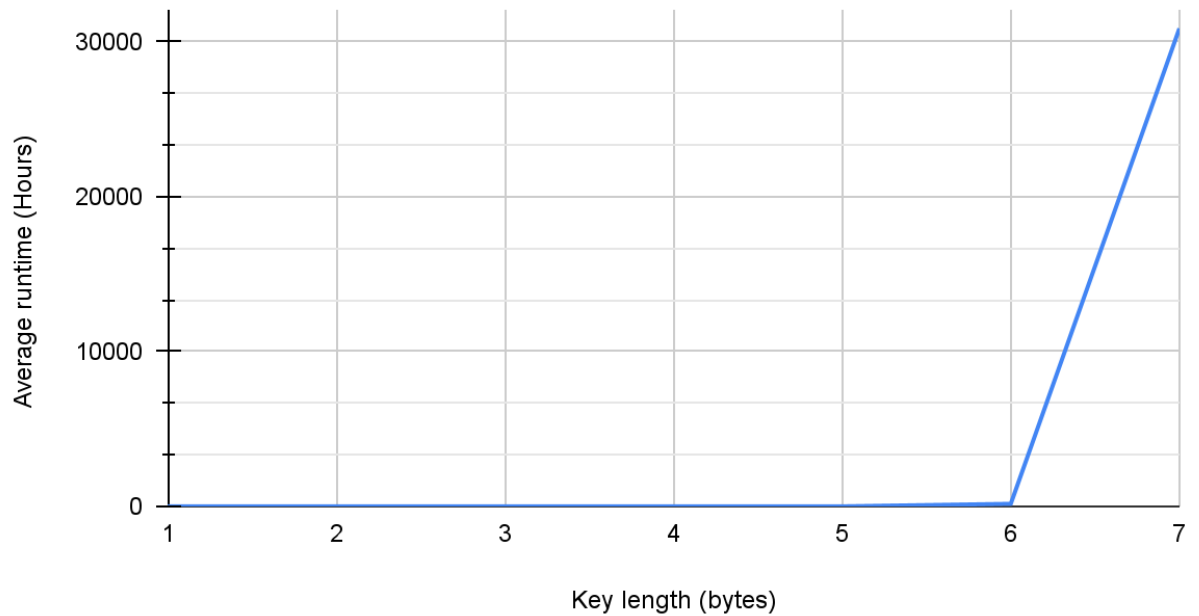
Note: This value will not be included in the chart analysis as I was under time constraints that wouldn't let me get a sample size of 4 to average from. This is simply to display that my program was able to crack a 40 bit key within reasonable time given the key was random.

KNOWN VALUES AVERAGE CASE RUNTIME CHART



EXTRAPOLATED AVERAGE CASE RUNTIME CHART

Average runtime vs key length (5 and up extrapolated)



The values 5, 6, and 7, are 0.8807363539, 164.8435169, and 30853.02989 hours respectively. These values were extrapolated using excel's GROWTH function like in the previous extrapolated worst case chart. As you can see, extrapolation is prone to error in the case of an abnormal jump in size between two known y values. In this case that would be the jump from 3 bytes to 4 bytes going from 0.07679125 seconds to 22.15685875 seconds. The reason I don't trust 7's value is because it comes out to a much higher value than the worst case scenario. This is most likely due to small sample sizes creating larger variation. My worst scenario did not experience as large of a jump in time between bytes. At its greatest leap in time it went from about .1 to 33 seconds. In any case 30853.02989 hours is about 1285.54 days. A huge jump from the 25 days predicted in the worst case scenario. A somewhat interesting find is that in my single run of 5 bytes at 3352.590851 seconds or about 0.93 hours, which is not far off from the extrapolated value of 0.8807363539. However because it was only a single run, it'd be dangerous to draw conclusions that there's any real correlation between the two. Just something to keep in mind. It is possible that some accuracy remained until 7 bytes was calculated.

PARALLEL PROGRAM WORST CASE RUNTIME SPEED

Key length = 8 bits/1 byte

```
hunter@hunter-desktop:~/CSCI551/finalProject/finalAssignmentMPI/worstCase$ cat worst1-*.log
00 = 00
program took 0.000052 seconds
00 = 00
program took 0.000045 seconds
00 = 00
program took 0.000188 seconds
00 = 00
program took 0.000058 seconds
```

Average runtime:



$(0.000052 + 0.000045 + 0.000188 + 0.000058) / 4 =$

0.00008575

Key length = 16 bits/2 bytes

```
hunter@hunter-desktop:~/CSCI551/finalProject/finalAssignmentMPI/worstCase$ cat worst2-*.log
000 = 000
program took 0.000203 seconds
000 = 000
program took 0.000222 seconds
000 = 000
program took 0.000215 seconds
000 = 000
program took 0.000079 seconds
```

Average runtime:




$(0.000203 + 0.000222 + 0.000215 + 0.000079) / 4 =$

0.00017975

Key length = 24 bits/3 bytes

```
hunter@hunter-desktop:~/CSCI551/finalProject/finalAssignmentMPI/worstCase$ cat worst3-*  
000 = 000  
program took 0.007138 seconds  
000 = 000  
program took 0.006326 seconds  
000 = 000  
program took 0.005624 seconds  
000 = 000  
program took 0.006217 seconds
```


Average runtime:

 $(0.007138 + 0.006326 + 0.005624 + 0.006217) / 4 =$
0.00632625

Key length = 32 bits/4 bytes

```
hunter@hunter-desktop:~/CSCI551/finalProject/finalAssignmentMPI/worstCase$ cat worst4-*  
0000 = 0000  
program took 0.590458 seconds  
0000 = 0000  
program took 0.504664 seconds  
0000 = 0000  
program took 0.582153 seconds  
0000 = 0000  
program took 0.514907 seconds
```

Average runtime:

 $(0.590458 + 0.504664 + 0.582153 + 0.514907) / 4 =$
0.5480455

Key length = 40 bits/5 bytes

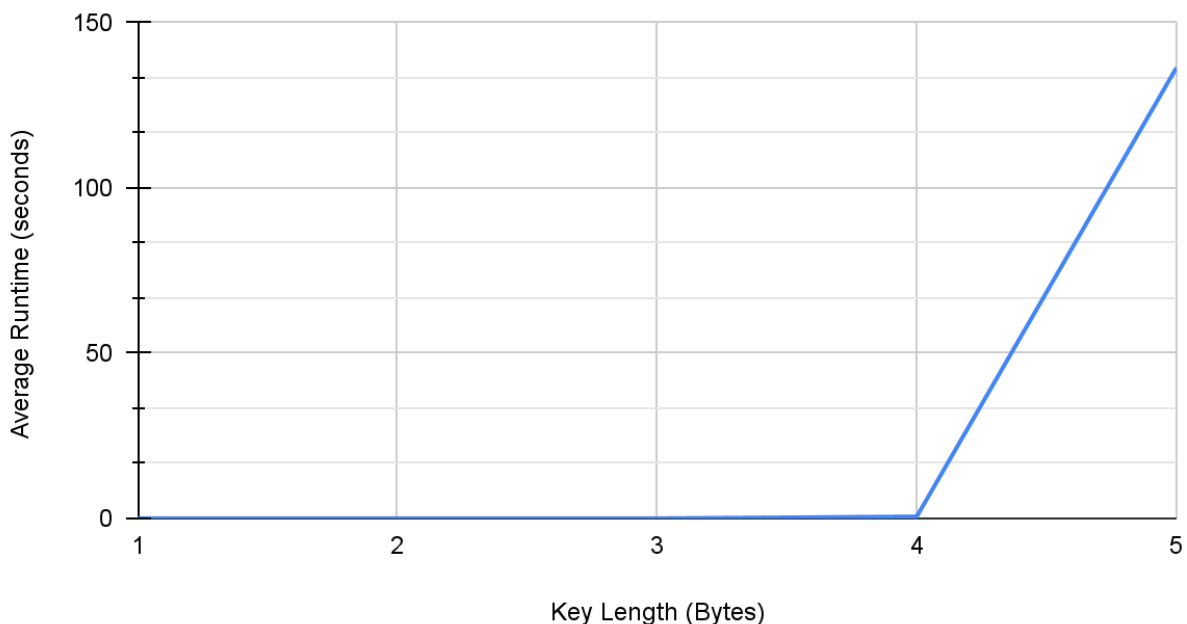
```
hunter@hunter-desktop:~/CSCI551/finalProject/finalAssignmentMPI/worstCase$ cat worst5-*.txt
00000 = 00000
program took 145.836209 seconds
00000 = 00000
program took 126.526160 seconds
00000 = 00000
program took 145.274121 seconds
00000 = 00000
program took 127.771637 seconds
```

Average runtime:

```
(145.836209 + 126.526160 + 145.274121 + 127.771637) / 4 =
136.35203175
```

KNOWN VALUES WORST CASE RUNTIME CHART

Average Worst Case Runtime vs Key Length

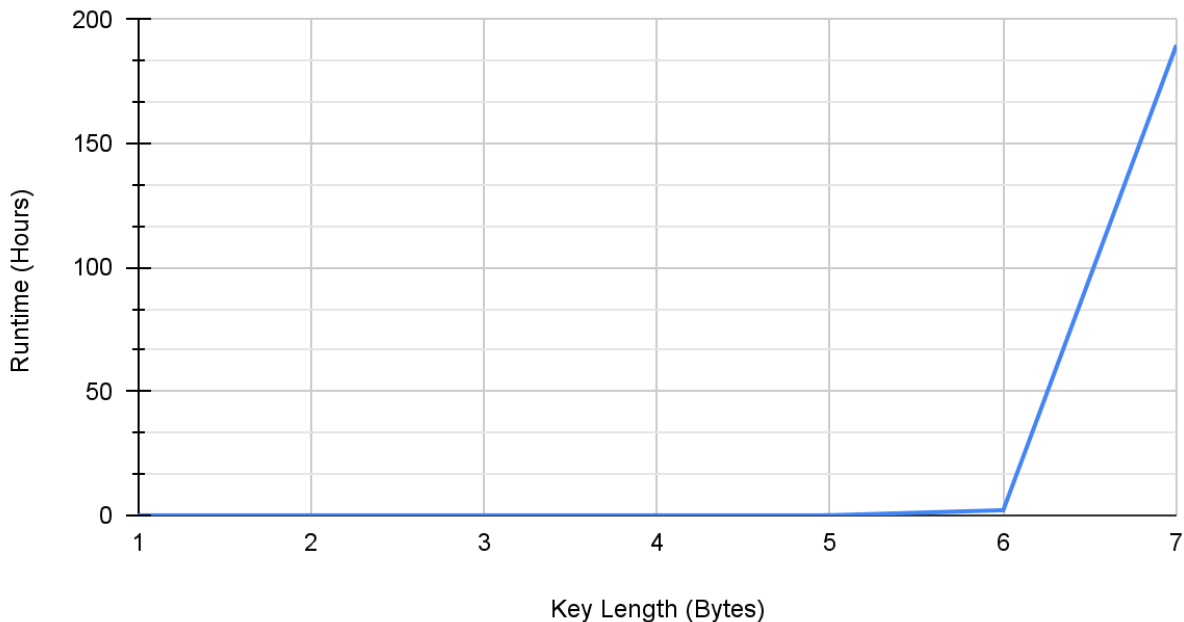


Already notable speedup has been achieved in the lower values. This only becomes more apparent as the byte size increases. The time difference starts to really become significant at byte size 4. Sequential ran at about a 33 second average. Here we are getting about a 0.5 second average. The gap only widens at 5 bytes with a

runtime of 0.2227507846 HOURS sequentially (this was extrapolated), while here it only took a meager average of 136-ish seconds.

EXTRAPOLATED WORST CASE RUNTIME CHART

Worst case runtime vs Key Length (6 and up extrapolated)



The values of 6 and 7 are 2.091380101 and 189.7570029 hours respectively. The mere fact that we no longer have to extrapolate 5 bytes as it's become viable to actually run and test 4 times speaks volumes for the amount of speed up being created. Although I am wary of comparing two extrapolated values to each other on a sample size of 4, it is notable that byte 6 has decreased by $11.58636955 - 2.091380101 = 9.494989449$ hours, while 7's decrease is even larger at $602.6643614 - 189.7570029 = 412.9073585$ hours. Although these numbers are probably prone to a large percentage of error, I still think they are helpful in getting a general sense of the type of speed we could be getting at the higher untestable byte sizes.

PARALLEL PROGRAM AVERAGE CASE RUNTIME SPEED

Key length = 8 bits/1 byte:

```
hunter@hunter-desktop:~/CSCI551/finalProject/finalAssignmentMPI/keylengthAvg$ cat keylength1-*.log
N = N
program took 0.000056 seconds
< = <
program took 0.000062 seconds
0 = 0
program took 0.000072 seconds
1 = 1
program took 0.000076 seconds
```

Average runtime:



$(0.000056 + 0.000062 + 0.000072 + 0.000076) / 4 =$

0.0000665

Key length = 16 bits/2 bytes:

```
hunter@hunter-desktop:~/CSCI551/finalProject/finalAssignmentMPI/keylengthAvg$ cat keylength2-*.log
>F = >F
program took 0.000060 seconds
0 = 0
program took 0.000017 seconds
C = C
program took 0.000007 seconds
0] = 0]
program took 0.000069 seconds
```

Average runtime:



$(0.000060 + 0.000017 + 0.000007 + 0.000069) / 4 =$

0.00003825

Key length = 24 bits/3 bytes:

```
hunter@hunter-desktop:~/CSCI551/finalProject/finalAssignmentMPI/keylengthAvg$ cat keylength3-*
0T0 = 0T0
program took 0.000432 seconds
0R = 0R
program took 0.000371 seconds
0@0 = 0@0
program took 0.000298 seconds
f0r = f0r
program took 0.000574 seconds
```

Average runtime:

```

(0.000432 + 0.000371 + 0.000298 + 0.000574) / 4 =
0.00041875
```

Key length = 32 bits/4 bytes:

```
hunter@hunter-desktop:~/CSCI551/finalProject/finalAssignmentMPI/keylengthAvg$ cat keylength4-*
0>V0 = 0>V0
program took 0.307424 seconds
|$F, = |$F,
program took 0.040888 seconds
Dv00 = Dv00
program took 0.114700 seconds
000j = 000j
program took 0.257820 seconds
```

Average runtime:

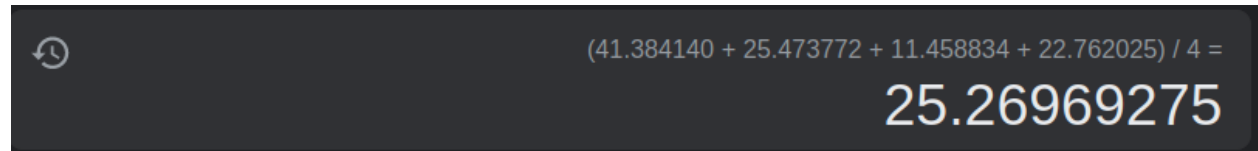
```

(0.307424 + 0.040888 + 0.114700 + 0.257820) / 4 =
0.180208
```

Key length = 40 bits/5 bytes:

```
hunter@hunter-desktop:~/CSCI551/finalProject/finalAssignmentMPI/keylengthAvg$ cat keylength5-*.txt
00AZ = 00AZ
program took 41.384140 seconds
0m[o = 0m[o
program took 25.473772 seconds
0-:} = 0-:}
program took 11.458834 seconds
00H = 00H
program took 22.762025 seconds
```

Average runtime:



$(41.384140 + 25.473772 + 11.458834 + 22.762025) / 4 =$
25.26969275

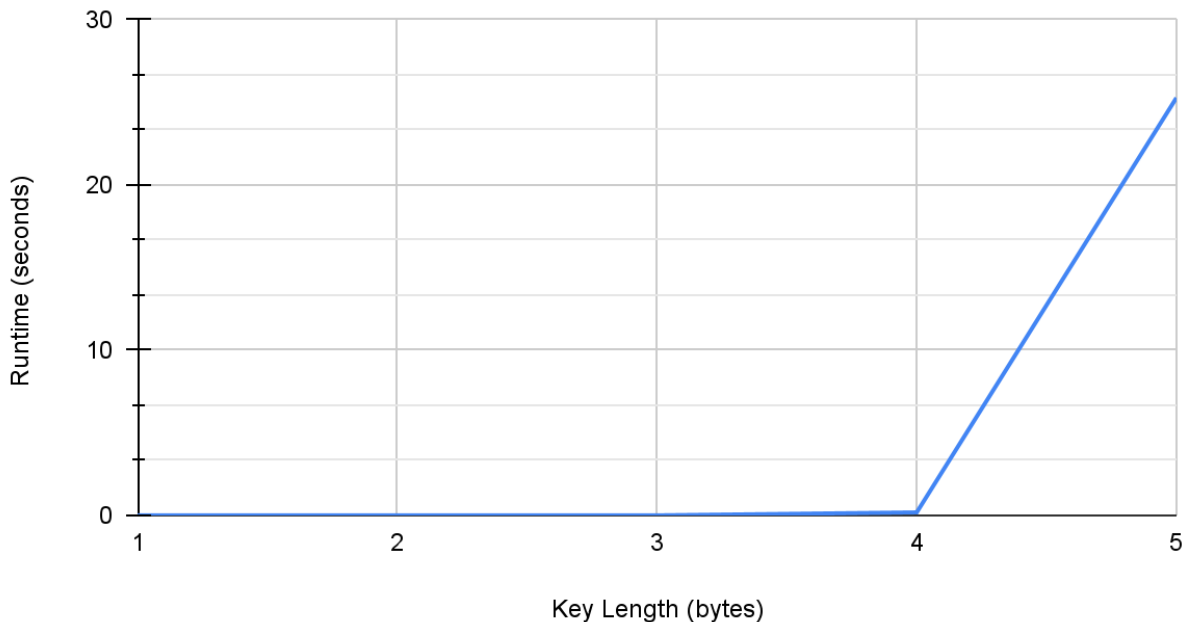
Key length = 48 bits/6 bytes:

```
hunter@hunter-desktop:~/CSCI551/finalProject/finalAssignmentMPI/keylengthAvg$ cat keylength6-*.txt
000F4o = 000F4o
program took 12222.206046 seconds
```

Note: This value will not be included in the chart analysis as I was under time constraints that wouldn't let me get a sample size of 4 to average from. This is simply to display that my program was able to crack a 48 bit key within reasonable time given the key was random. This is about 3.40 hours.

KNOWN VALUES AVERAGE CASE RUNTIME CHART

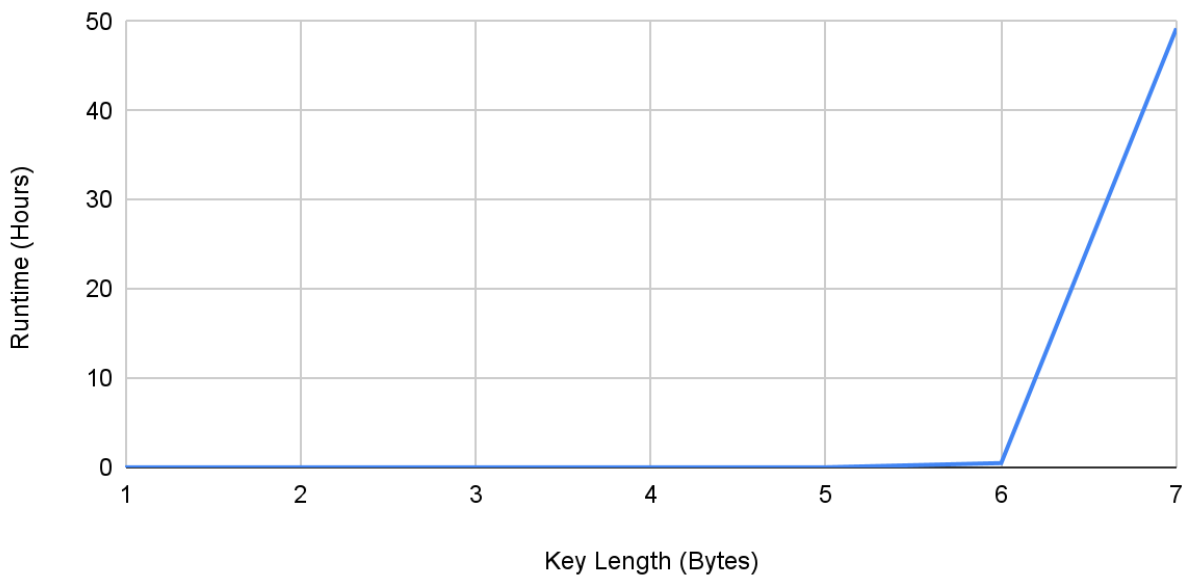
Average Case Runtime vs. Key Length



This is where my most notable speed up occurred as you'll see in the later section with amdahl's law calculations. The smallest size of 1 however did not benefit from mpi in an average case. This can easily be concluded as an issue of mpi's overhead in sending messages across nodes. The sequential program finishes quicker simply because it does not have this issue. Fortunately, no one is trying to crack keys of a single byte, so this is irrelevant really, but I felt it necessary to include. The best speed up occurred with 3 bytes. 3 bytes the average case sequentially took 0.07679125 seconds. With MPI I got it down to 0.00041875 seconds. If you are looking for a more humanly noticeable change in time, then 5 bytes is my second largest with an original time of about 3170.65 seconds or about 0.88 hours and a parallel time of about 25.27 seconds or about 0.007 hours.

EXTRAPOLATED AVERAGE CASE RUNTIME CHART

Average Case Runtime vs Key Length (Extrapolated values 6 and up)



The main takeaway here is that 6 becomes a viable key length to crack coming in at 0.4818021951 hours. I ran one instance of this just to test it and it took about 3.40 hours. This is much higher than the extrapolated result, but due to the random generation of keys, it'd be dangerous to make an assumption off a single run. This run was mainly done just to show it could be done in a reasonable amount of time but I simply don't have the time to test it 4 times over to match the rest of my data. Next, if we are to trust the extrapolation 7 bytes went from taking about 1285.54 days to 49.23129452 days for a difference of about 1236.31 days. Although 49.23 days is a far cry from the 22 hours and 15 minutes record of deep crack in the 90's, it is a significant speed up in the context of my tests.

SPEED UP WITH AMDAHL'S LAW

In this section I will be providing calculations of speed up for each key size for worst case and average case results. Without further ado.

Key length 1 byte Worst Case:

Task proportion (p)	62.06 %▼
Speedup factor (s)	240
Speedup (S)	2.618
Original time	0.0002245 sec ▼
Improved time	0.00008575 sec ▼
Time percentage change	61.8 %

$$0.0002245 / 0.00008575 = 2.61807580175$$

$$(S * (1 - (1 / SU))) / (S - 1)$$

$$(240 * (1 - (1 / 2.61807580175))) / (240 - 1) = 62.062603088\% \text{ Parallel}$$

Key length 1 Average Case:

SLOWDOWN

Task proportion (p)	-1,680.3 %▼
Speedup factor (s)	240
Enter proportion of task in the range 0-1 (0-100%).	
Speedup (S)	0.05639
Original time	0.00000375 sec▼
Improved time	0.0000665 sec▼
Time percentage change	-1,673.3 %

Note: This is most likely due to the small size of the key i.e. gustafson's law, as well as the fact that there is overhead from the mpi process calls. If for some reason you were trying to crack an 8 bit key, parallel wouldn't really be anyone's go to answer for obvious reasons. Still wanted to include this data though.

$$0.00000375/0.0000665 = 0.05639097744$$

$$(S * (1 - (1 / SU))) / (S - 1)$$

$$(240 * (1 - (1 / 0.05639097744))) / (240 - 1) = -1680.33472815\% \text{ Parallel}$$

Key length 2 bytes Worst Case:

Task proportion (p)	20.017 %▼
Speedup factor (s)	240
Speedup (S)	1.249
Original time	0.0002245 sec▼
Improved time	0.00017975 sec▼
Time percentage change	19.933 %

$$0.0002245/0.00017975 = 1.24895688456$$

$$(S * (1 - (1 / SU))) / (S - 1)$$

$$(240 * (1 - (1 / 1.24895688456))) / (240 - 1) = 20.016587302\% \text{ Parallel}$$

Key length 2 Average Case:

Task proportion (p)	87.63 %▼
Speedup factor (s)	240
Speedup (S)	7.85
Original time	0.00030025 sec▼
Improved time	0.00003825 sec▼
Time percentage change	87.26 %

$$0.00030025 / 0.00003825 = 7.84967320261$$

$$(S * (1 - (1 / SU))) / (S - 1) = p$$

$$(240 * (1 - (1 / 7.84967320261))) / (240 - 1) = 87.625723333\% \text{ Parallel}$$

Key length 3 bytes Worst Case:

Task proportion (p)	95.61 % ▼
Speedup factor (s)	240
Speedup (S)	20.9
Original time	0.13220725 sec ▼
Improved time	0.00632625 sec ▼
Time percentage change	95.21 %

0.13220725/0.00632625 =

$(S * (1 - (1 / SU))) / (S - 1) = p$

$(240 * (1 - (1 / 20.8982019364))) / (240 - 1) = 95.613288034 \% \text{ Parallel}$

Key length 3 bytes Average Case:

<u>Task proportion (p)</u>	99.87 %▼
<u>Speedup factor (s)</u>	240
<u>Speedup (S)</u>	183.4
<u>Original time</u>	0.07679125 sec ▼
<u>Improved time</u>	0.00041875 sec ▼
<u>Time percentage change</u>	99.45 %

$$.07679125/.00041875 = 183.382089552$$

$$(S * (1 - (1 / SU))) / (S - 1) = p$$

$$(240 * (1 - (1 / 183.382089552))) / (240 - 1) = 99.870818887\% \text{ Parallel}$$

Key length 4 bytes Worst Case:

Task proportion (p)	98.75 %▼
Speedup factor (s)	240
Speedup (S)	60.3
Original time	33.0447355 sec▼
Improved time	0.5480455 sec▼
Time percentage change	98.34 %

$$33.0447355 / 0.5480455 = 60.2956059305$$

$$(S * (1 - (1 / SU))) / (S - 1) = p$$

$$(240 * (1 - (1 / 60.2956059305))) / (240 - 1) = 98.752975082\% \text{ Parallel}$$

Key length 4 bytes Average Case:

Task proportion (p)	99.6 % ▼
Speedup factor (s)	240
Speedup (S)	122.95
Original time	22.15685875 sec ▼
Improved time	0.180208 sec ▼
Time percentage change	99.19 %

$$22.15685875 / 0.180208 = 122.951582338$$

$$(S * (1 - (1 / SU))) / (S - 1) = p$$

$$(240 * (1 - (1 / 122.951582338))) / (240 - 1) = 99.601678706\% \text{ Parallel}$$

FROM HERE ON OUT EXTRAPOLATED SEQUENTIAL VALUES ARE USED

Key length 5 bytes Worst Case:

Task proportion (p)	83.34 %▼
Speedup factor (s)	240
Speedup (S)	5.881
Original time	801.9028245 sec▼
Improved time	136.3520318 sec▼
Time percentage change	83 %

$$801.9028245/136.3520318 = 5.8811211972$$

$$(S * (1 - (1 / SU))) / (S - 1) = p$$

$$(240 * (1 - (1 / 5.8811211972))) / (240 - 1) = 83.343704951\% \text{ Parallel}$$

Key length 5 bytes Average Case:

Task proportion (p)	99.62 % ▼
Speedup factor (s)	240
Speedup (S)	125.47
Original time	3,170.650874 sec ▼
Improved time	25.26969275 sec ▼
Time percentage change	99.2 %

$$3170.650874 / 25.26969275 = 5.8811211972$$

$$(S * (1 - (1 / SU))) / (S - 1) = p$$

$$(240 * (1 - (1 / 125.472474294))) / (240 - 1) = 99.618087814\% \text{ Parallel}$$

AMDAHL'S LAW PLOT FOR WORST CASE

1 byte

0.000236 (Entire program) - .0002245 (proportion to make parallel) = 0.0000115

100 - 0.0000115 = 99.9999885% parallel

$1/(1-p) + (p/s) = \text{SU}$

$1/(1-.999999885) + (.999999885 / 240) = 239.993403781$

2 bytes

0.002849 (Entire program) - 0.00290825 (proportion to make parallel) = 0.00005925

100 - 0.00005925 = 99.99994075% parallel

$1/(1-p) + (p/s) = \text{SU}$

$1/(1-.9999994075) + (.9999994075 / 240) = 239.966019012$

3 bytes

0.169765 (Entire program) - 0.13220725 (proportion to make parallel) = 0.03755775

100 - 0.03755775 = 99.96244225

$1/(1-p) + (p/s) = \text{SU}$

$1/(1-.9996244225) + (.9996244225 / 240) = 220.231366861$

4 bytes

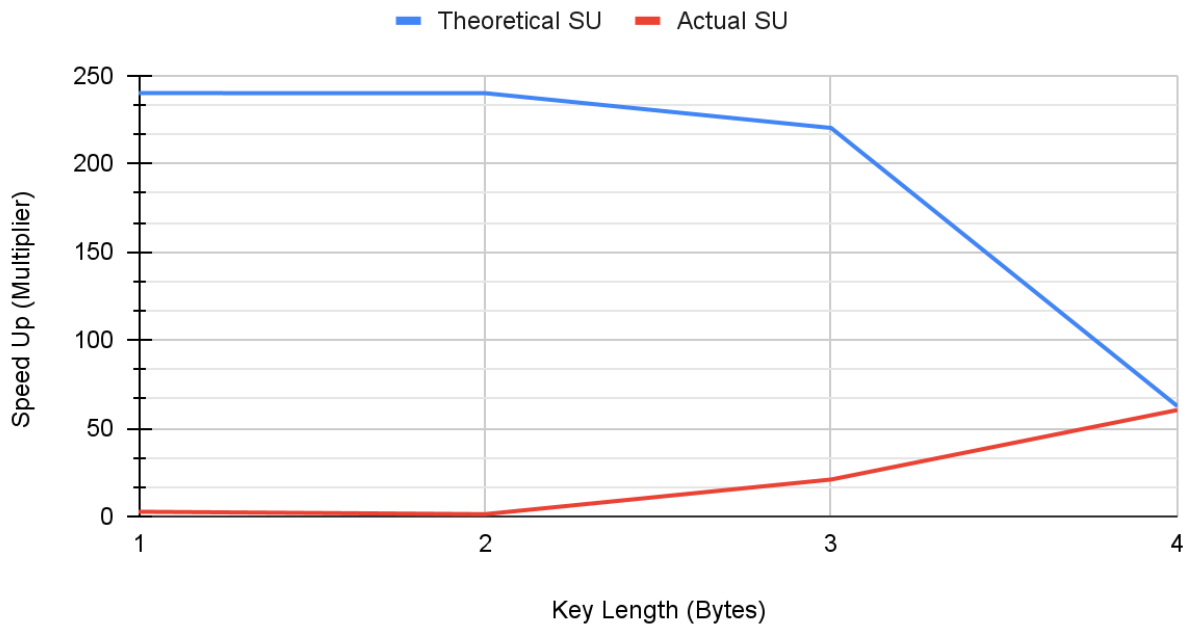
34.233518 (Entire program) - 33.0447355 (proportion to make parallel) = 1.1887825 SU

100 - 1.1887825 = 98.8112175% parallel

$1/(1-p) + (p/s) = \text{SU}$

$1/(1-.988112175) + (.988112175 / 240) = 62.4806346642$

Theoretical SU vs Actual SU



As I suspected earlier, Gustafson's law is at work here. Only once the data and runtime became big enough did the expected speed up and actual speed up start to converge. I suspect if I were to continue charting the key sizes they would remain pretty similar in speed up.

Conclusion

This problem was embarrassingly parallel, and in fact I would go as far to say that there's no point in even trying to crack an encryption key with sequential code. The problem practically begs for a parallel answer or none at all. I was unable to achieve a realistic time for cracking a 56 bit DES key unfortunately, but was still happy to see massive speed up theoretically at that size. Less theoretically, I actually did get to crack a 48 bit key in a practical amount of time, so all in all not too bad. My limitations were simply a result of not enough processing power. Although deep crack did it in the 90's and their chips were definitely weaker than the cores we have today, they were custom built and there were 1856 of them running assembly. Assembly is notably faster than C. I am also confident that they had more processing power than me, based on the fact alone that they used a brute force algorithm but ALSO decrypted the first 64 bits of every message to check for interesting plain text. This is more overhead than my program and yet they got it to run significantly faster.

INSTRUCTIONS TO COMPILE AND RUN

TO COMPILE main.c:

“gcc main.c -o main”

TO RUN main.c:

“./main <key length in bytes> (must be 1-7)”

TO COMPILE mpiDEScracker.c and mpiWorstCase.c:

“make”

TO RUN mpiDEScracker.c:

**mpirun -np 240 -ppn 4 -f c1_hosts ./mpiDEScracker <key length in bytes>
(must be 1-7, but note 7 was not implemented. It's not viable)**

TO RUN mpiWorstCase.c:

**mpirun -np 240 -ppn 4 -f c1_hosts ./mpiWorstCase <key length in bytes>
(must be 1-7, but note 7 was not implemented. It's not viable)**