# An Analysis of Multithreading Computation Execution time

**John Christopher Egnatis**

University of Texas at Dallas
Eric Johnson School of Engineering and Computer Science

**Abstract**- This study was conducted to analyze the speed of execution time considering different numbers of threads. A program was developed to test the effectiveness of multithreading given different amounts of threads. The data collected suggests some rules that explain multithreading computations. One observation is that as the number of threads increase, the execution time will increase to a certain point, but eventually bottleneck. Another observation was that the size and nature of the input will affect the optimal number of threads to result in the fastest execution time. From the findings, it can also be assumed that there exists a general best number of threads for any given algorithm.

## I. INTRODUCTION

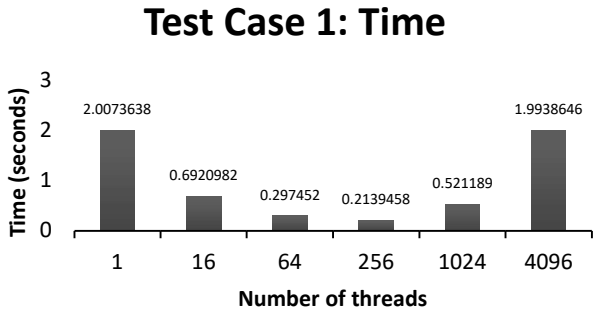
Multithreading is a technique which allows for large commutations or functions to divide the work between multiple threads in attempts to optimize the speed of execution. In systems programming where large files and directories are being accessed, multithreading is essential to creating a time efficient application. This study will attempt to determine the optimal number of threads to speed up a hashing function. It is hypothesized that there will be an initial spike in effectiveness for each new thread added, but eventually the multithreading will reach an upper limit and drop off in effectiveness. Using an application, this hypothesis will be thoroughly tested across varying number of threads and file sizes, recording the execution time.

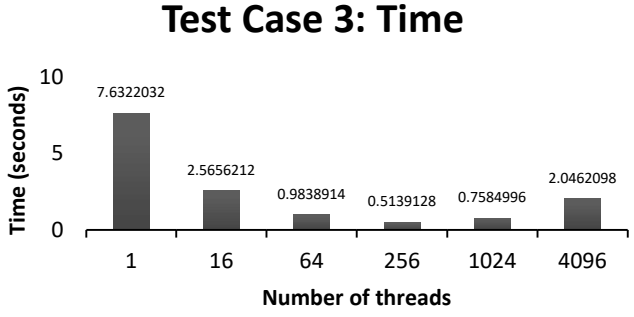
## II. RESEARCH

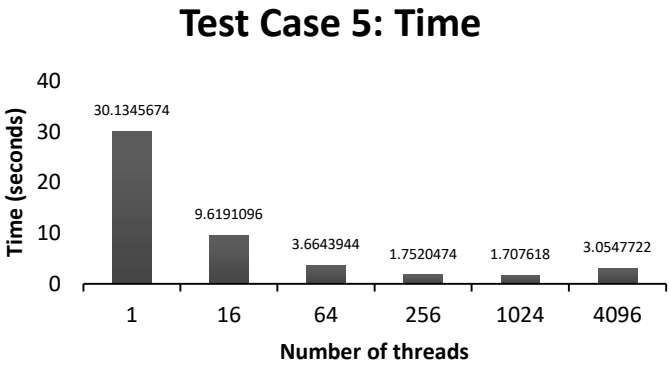
To test and record the speed of generating the hash value based on a file, an application was developed using the C programming language. This application receives parameters as input, which indicates the file to test and the number of threads to create in concurrence. When executed, the program will divide the file into **n** equally sized parts (n being the number of threads), so that each thread can compute the hash value for its section of the file. The time is recorded from the start to the end of the computation.

The application was developed and tested on the Linux servers at The University of Texas at Dallas. The software limits each user to at most 4200 concurrent threads running at a time, so the upper bound for number of threads cannot exceed that limit. Five test cases with different sizes have been created to test the hypothesis. The size specifications of these files can be found in the Appendix.

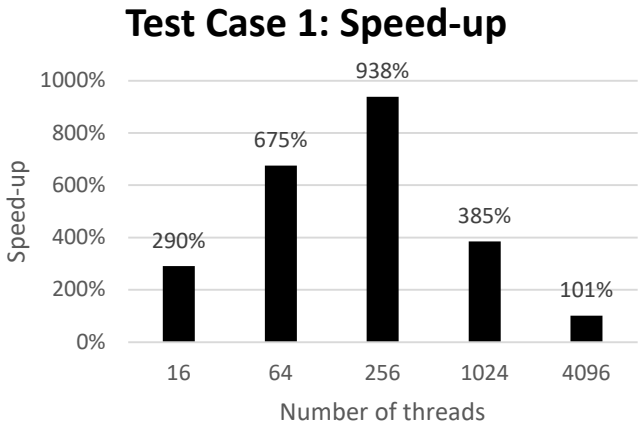
The procedure is as follows. Five files were created for testing, each of different sizes. These will be named as Test Case one through five for simplicity, and they increase in size with the number associated. Each file will be tested 5 times and average the resulting execution times, which will be compiled into the graphs below. Each test will compute the times for one, sixteen, sixty-four, two-hundred fifty-six, one-thousand twenty-four, and four-thousand ninety-six threads. *Test cases 2 and 4 will be included in the Appendix.*

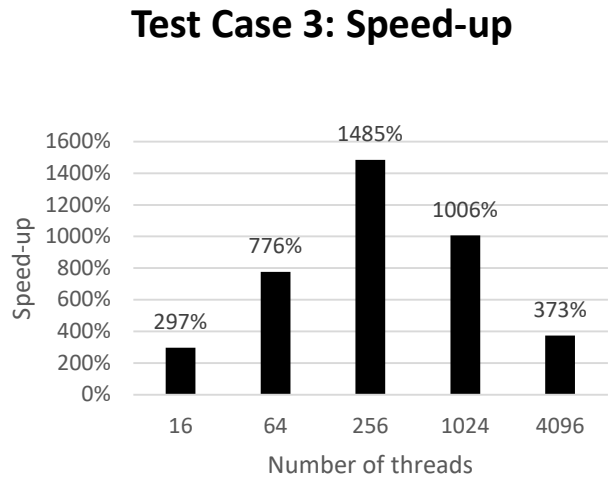


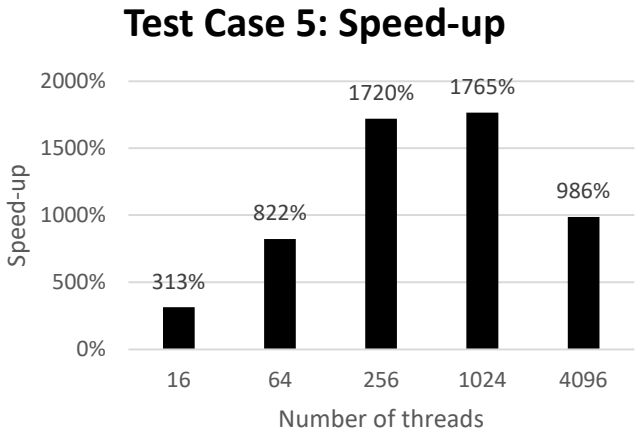






To assist in analysis, speed-up can be defined execution time for one thread to hash divided by the execution time for any **n** number of threads to hash. This will produce a percentage, which shows how much improvement the multithreaded execution time gives compared to single threading.

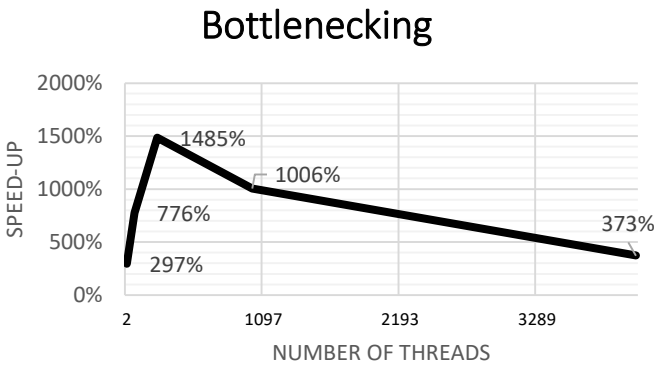



## III. ANALYSIS

From the data and the speed-up analysis, a pattern is formed. Clearly, single threading is the worst option to choose for computing the hash value of a large file, as it has performed the worst in every case in the range. This was the expected observation. Likewise, it was expected that increasing the number of threads would increase the speed of the execution. However, at a certain point the speed-up of increasing the number of threads bottlenecks. This is because creating a thread takes time, and that time for large number of threads slows the execution down faster than the extra threads speeds it up. Therefore, there exists a point in any speed-up graph where the number of threads is optimal.




*The above graph demonstrates the bottlenecking effect as a function of speed-up and threads in test case three. After thread 256, a gradual decline in any additional threads is observed.*

Additionally, it can be observed that the bottlenecking is dependent on the size of the file. In test case one, the file used was relatively small. According to the data, the bottleneck started well before one-thousand threads, and by four-thousand threads the program runs about as fast as if it were being single-threaded, defeating the purpose of multithreading to begin with. However, with test case five

where the file is relatively very large, the speed-up shows that the bottlenecking starts much later than it did in test case one. Test case five was more efficient with one-thousand thread; and at four-thousand threads, the speed-up from single threading is still greater by a factor of ten. However, the data indicates that the bottlenecking effect is inevitable, even for a large file.

One observation that was not expected was the deviation of the speed-up graphs. Before bottlenecking, it could be assumed that the speed-up graph would have a linear growth. However, the slope of the graph suggests a parabolic growth, as speedup is not proportional to the number of threads. Regardless, the data does confirm the initial hypothesis that the initial threads would more effective. The graph does not, however, affirm how the number of threads before bottlenecking seemed to be the most effective. Another unexpected observation was how multithreading with two-hundred fifty-six threads is consistently a strong option. Given that the input file size is unknown, any number of threads approximately 256 could be considered close to optimal in this study.

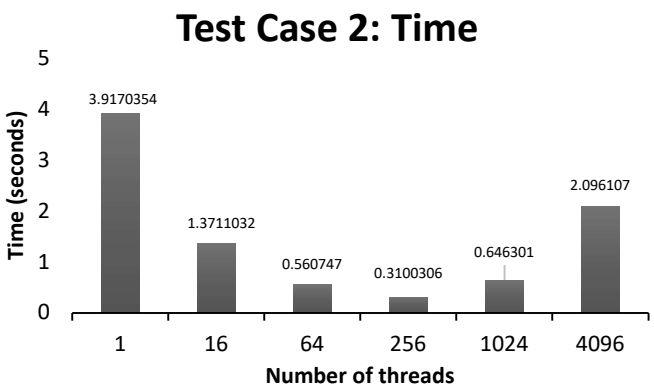
## IV. CONCLUSION

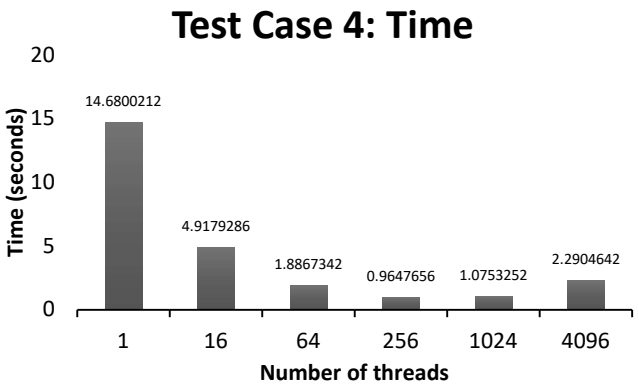
All the data indicates a couple undeniable truths concerning threading. The first truth is that all threading will eventually bottleneck. As with test case five, it may be after the creation of over one-thousand threads that the execution begins to bottleneck, but eventually it will fall off. No algorithm will be an exception to this rule. The second truth states that the number of threads which optimize the execution time is dependent on the algorithm and the input provided. In this study, the linear nature of the algorithm makes the file size the dominate the complexity. This will not be the case for different algorithm.

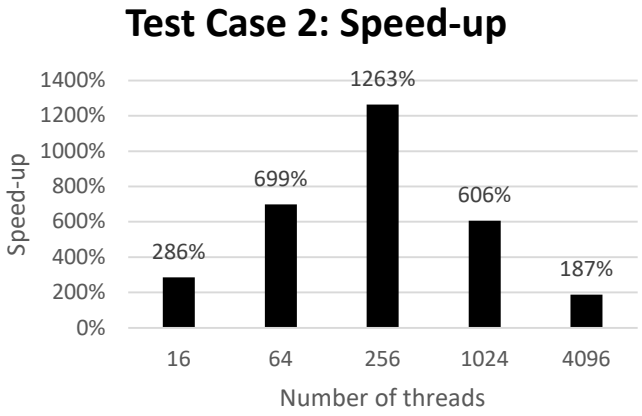
A final observation from the data is the possibility that every algorithm may have an exact number of threads that is close to optimal. This could be called a magic number. In this study, 256 threads were recorded to have a strong or best speed-up percentage across all five studies. What this indicates, is that there may be a specific number of concurrent running threads which guaranties a strong execution speed. It will vary between algorithms, but this magic number should be the same for all inputs except for very small input sizes. Further studies should be conducted to see if there is truth to this observation across many algorithms and applications.

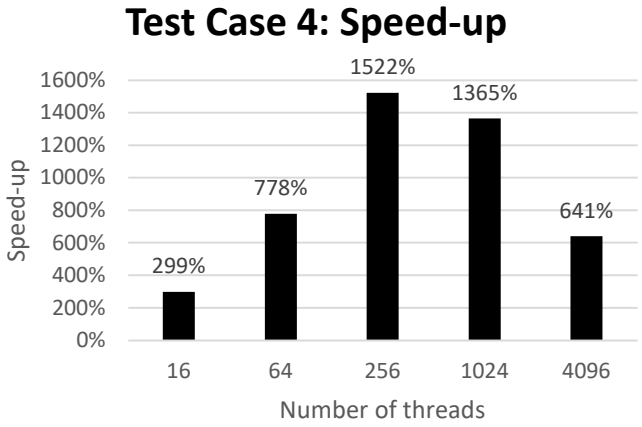
# APENDIX

- Test case 1: 268435456 bits
- Test case 2: 536870912 bits
- Test case 3: 1073741824 bits
- Test case 4: 2147483648 bits
- Test case 5: 4294967296 bits




## Test Case 1

| Number of Threads | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average | Speed-up |
|---|---|---|---|---|---|---|---|
| 1 | 2.009065 | 1.955731 | 1.890946 | 2.208254 | 1.972823 | 2.0073638 | 100% |
| 16 | 0.66148 | 0.690466 | 0.699981 | 0.678352 | 0.730212 | 0.6920982 | 290% |
| 64 | 0.310941 | 0.306994 | 0.340215 | 0.257408 | 0.271702 | 0.297452 | 675% |
| 256 | 0.206844 | 0.212272 | 0.207733 | 0.232173 | 0.210707 | 0.2139458 | 938% |
| 1024 | 0.55859 | 0.466047 | 0.430298 | 0.610108 | 0.540902 | 0.521189 | 385% |
| 4096 | 1.997081 | 2.001298 | 1.988687 | 1.995892 | 1.986365 | 1.9938646 | 101% |

## Test Case 2

| Number of Threads | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average | Speed-up |
|---|---|---|---|---|---|---|---|
| 1 | 3.651506 | 3.891588 | 3.832233 | 4.550808 | 3.659042 | 3.9170354 | 100% |
| 16 | 1.403014 | 1.359645 | 1.390408 | 1.385688 | 1.316761 | 1.3711032 | 286% |
| 64 | 0.584394 | 0.536834 | 0.545787 | 0.531302 | 0.605418 | 0.560747 | 699% |
| 256 | 0.284237 | 0.334546 | 0.308667 | 0.295933 | 0.32677 | 0.3100306 | 1263% |
| 1024 | 0.62102 | 0.711831 | 0.682404 | 0.595934 | 0.620316 | 0.646301 | 606% |
| 4096 | 2.058892 | 2.160968 | 2.09772 | 2.082769 | 2.080186 | 2.096107 | 187% |

## Test Case 3

| Number of Threads | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average | Speed-up |
|---|---|---|---|---|---|---|---|
| 1 | 7.283828 | 7.349698 | 8.045198 | 8.206324 | 7.275968 | 7.6322032 | 100% |
| 16 | 2.567531 | 2.470176 | 2.488315 | 2.804963 | 2.497121 | 2.5656212 | 297% |
| 64 | 1.054777 | 0.930669 | 1.030883 | 0.937771 | 0.965357 | 0.9838914 | 776% |
| 256 | 0.493577 | 0.491964 | 0.486036 | 0.588449 | 0.509538 | 0.5139128 | 1485% |
| 1024 | 0.764493 | 0.810055 | 0.776464 | 0.774635 | 0.666851 | 0.7584996 | 1006% |
| 4096 | 1.639006 | 2.224408 | 2.30231 | 1.824321 | 2.241004 | 2.0462098 | 373% |

## Test Case 4

| Number of Threads | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average | Speed-up |
|---|---|---|---|---|---|---|---|
| 1 | 14.627721 | 14.957258 | 14.584012 | 14.678585 | 14.55253 | 14.6800212 | 100% |
| 16 | 4.757558 | 4.861378 | 5.126909 | 5.009581 | 4.834217 | 4.9179286 | 299% |
| 64 | 1.901218 | 1.883135 | 1.892101 | 1.891502 | 1.865715 | 1.8867342 | 778% |
| 256 | 0.972002 | 0.934858 | 0.934378 | 0.997137 | 0.985453 | 0.9647656 | 1522% |
| 1024 | 1.108289 | 1.066301 | 1.062879 | 1.068787 | 1.07037 | 1.0753252 | 1365% |
| 4096 | 2.432942 | 2.030511 | 2.642473 | 1.720311 | 2.626084 | 2.2904642 | 641% |

## Test Case 5

| Number of Threads | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average | Speed-up |
|---|---|---|---|---|---|---|---|
| 1 | 29.863094 | 29.508118 | 31.60389 | 30.774349 | 28.923386 | 30.1345674 | 100% |
| 16 | 9.622835 | 9.779208 | 9.600723 | 9.558567 | 9.534215 | 9.6191096 | 313% |
| 64 | 3.641232 | 3.684672 | 3.621456 | 3.590991 | 3.783621 | 3.6643944 | 822% |
| 256 | 1.740224 | 1.803089 | 1.785589 | 1.707072 | 1.724263 | 1.7520474 | 1720% |
| 1024 | 1.678493 | 1.678535 | 1.703274 | 1.731075 | 1.746713 | 1.707618 | 1765% |
| 4096 | 3.285495 | 2.69874 | 3.245519 | 2.854351 | 3.189756 | 3.0547722 | 986% |