PARALLELIZING SHAMIR’S SECRET   
SHARING ALGORITHM

### 1 ABSTRACT

This paper describes how Shamir’s secret sharing algorithm can be parallelized, decreasing the time required to generate keys for secrets shared among a large group of participants. Using an open source C implementation of Shamir’s algorithm and OpenMP, we found regions of the algorithm where running in parallel reduced execution time significantly. We were able to see near-linear speedup in both the key share generation phase and the re-combining phase. We also observed weak scaling when generating the key shares. Our work enables more efficient secret-sharing using Shamir’s algorithm.

### 2 INTRODUCTION

Shamir’s secret sharing scheme is a method for dividing a secret among a group, where a certain threshold of the keys must be combined in order to reproduce the secret. With large secrets, such as an entire text document, this process can be slow and expensive, especially if the number of participants and the threshold are high. Motivation for our work described in this paper comes from an interest in security and a semester project in a parallel and distributed systems course. In our initial research for the project, we discovered several citations referring to Shamir’s secret sharing and other secret sharing schemes based off of Shamir’s, but no real effort to use concurrent programming to speed up the original Shamir’s secret-sharing algorithm.

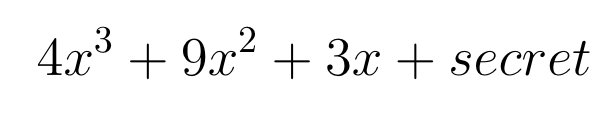
In this paper, we explore opportunities for parallelism in the algorithm, with a goal of reducing the amount of time taken to generate keys and join keys in a scalable manner.

Scaling in high-performance computing is broken into two categories: strong and weak scaling. Strong scaling describes how performance changes as the number of processes or threads increases for a fixed problem size, while weak scaling examines how performance changes as both the processor/thread count and problem size increase.

#### 2.1 Background

Shamir’s secret sharing algorithm, explained in [1], works by taking the number of keys desired (n) and the threshold that is required to unlock the secret (t). The algorithm computes the keys by generating a random polynomial equation of degree t-1. The secret becomes the constant value in the polynomial equation.

Example 2-1:



The required threshold to reproduce the secret of the function listed in Example 2-1 would be 4 keys, t = 4 yields a degree 3 polynomial. After generating the random polynomial, the algorithm computes X and Y coordinate pairs by generating a random X value and plugging it into the polynomial function to get a corresponding Y value. This is repeated n times for each character in the input file. The XY pairs become the keys that are distributed to each individual in the group.

The problem with this approach (when computed serially) is that an XY ordered pair for each character of the input data must be computed n number of times. This process is slow and provides opportunities for data parallelism.

#### 2.2 Project goals

Using an open source C implementation of Shamir’s secret sharing algorithm [2], we explored the benefits of parallelizing Shamir’s secret sharing algorithm. The overall focus of our project was to speed up the process of generating key shares for large files between a large number of parties.

### 3 METHODS

We used OpenMP in order to take advantage of its parallel for loop construct. Specifically, we identified 3 regions of the code where parallelism could be exposed and implemented parallel versions of these functions (Source code available at: <https://github.com/arbogajk/470_SP>). All of our experiments and testing were conducted on a Dell server with an 8-core (2.4Ghz w/ hyperthreading) Xeon E5-2630v3 processor with 32GB RAM using the maximum number of shares and threshold the original program was capable of generating, which was 255 key shares. For our test data sets, we used text files containing 540, 1080, 2160, 4320, and 8640 characters. We also used a 4096 bit RSA private key, containing 3,272 characters including the RSA header details as an input file into the program. Our weak scaling tests consist of doubling the character count of the input file while simultaneously doubling the thread count.

Our focus at first was studying the functions that dealt with generating the key shares. We identified two functions in the implementation that allowed for substantial decreases in time for computing the keys. Firstly, we were able to parallelize the for loop that generates the random coefficients used in the polynomial function. Secondly, we were able to parallelize the for loop that handles computing the key shares. By leaving the key joining function untouched, we were able to verify the correctness of this parallelization, because the key joining function could reassemble the keys into the original text file. After implementing parallelism in the key generation stage of Shamir’s secret sharing scheme, we switched our focus to implementing parallelism in the key joining function.

The challenge with parallelizing the key joining stage of Shamir’s secret sharing is correctly identifying OpenMP variable scope (i.e., selecting the variables that should be visible to all threads) as well as identifying and annotating critical regions to prevent race conditions. The most important region to consider is where each thread updates the secret after computing Lagrange interpolating polynomials. Access to this region had to be synchronized using the OpenMP “critical” construct, enabling us to parallelize the for loop that computes the Lagrange interpolating polynomial of the function used in joining the keys back together.

### 4 RESULTS

We were able to significantly speed up the secret splitting and joining using OpenMP. Our approach shows excellent strong and weak scaling. This section details some of our results.

#### 4.1 Strong Scaling

In Figure 1 and Table 1, we show that the times to create the key shares is nearly halved every time we double the number of threads, which means the new implementation achieves close to linear speedup.

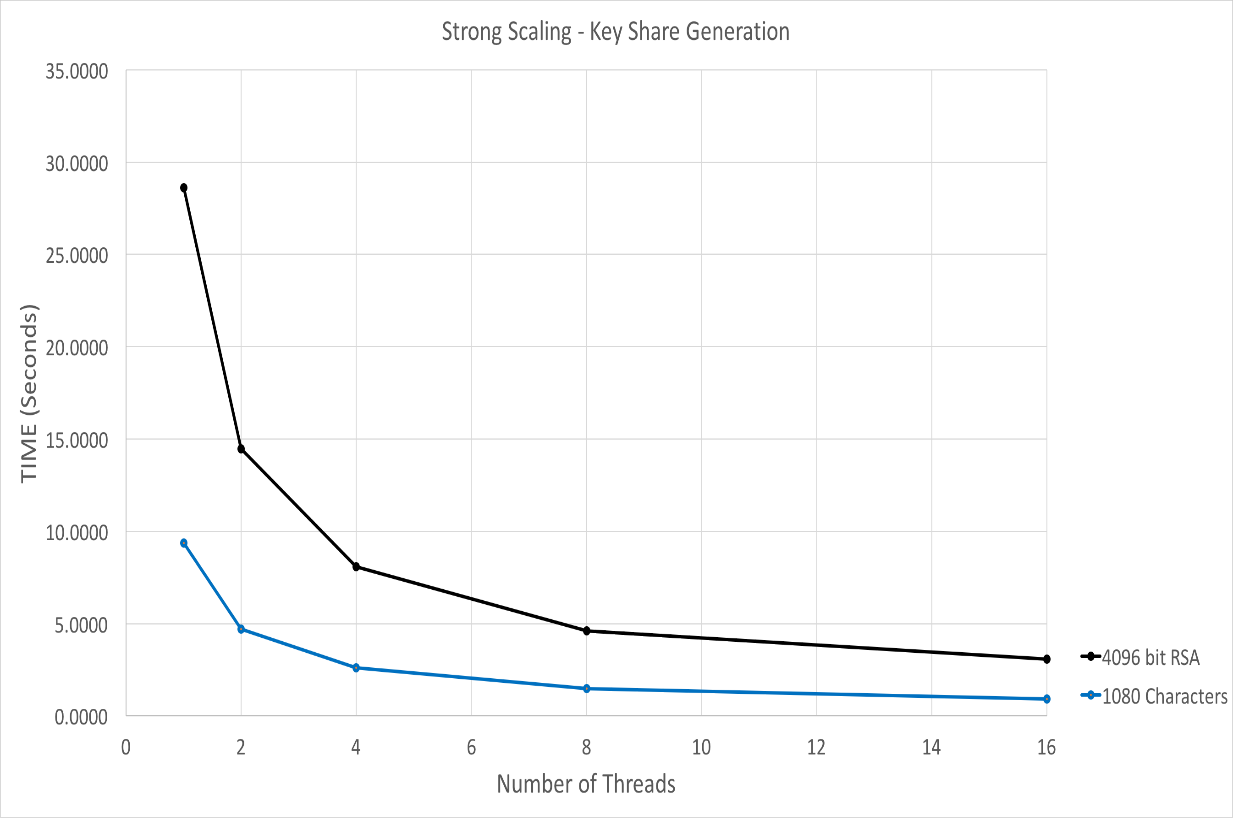


Figure 1: Results of Generating 255 keys with a required unlock threshold of 255 for a 4096 bit RSA key and 1,080-character input file.

Key Generation Results

|  |  |  |
| --- | --- | --- |
| Threads | 4096 Bit RSA Key | 1080 Character File |
| 1 | 28.62 sec | 9.36 sec |
| 2 | 14.46 sec | 4.71 sec |
| 4 | 8.07 sec | 2.61 sec |
| 8 | 4.60 sec | 1.46 sec |
| 16 | 3.09 sec | 0.92 sec |

Table 1: Shows the times taken to generate 255 keys with a threshold of 255

We get similar scaling results when joining the shares back together to get the original secret, shown in Figure 2 and Table [2](#x1-7004r2). It’s important to note here that we start to see increases in time at 16 threads when joining the keys back together. This corresponds to the number of physical cores our test machine (eight, with sixteen hyperthreads). At this point, increasing the number of threads becomes counterproductive because they cannot all run in parallel on the hardware.

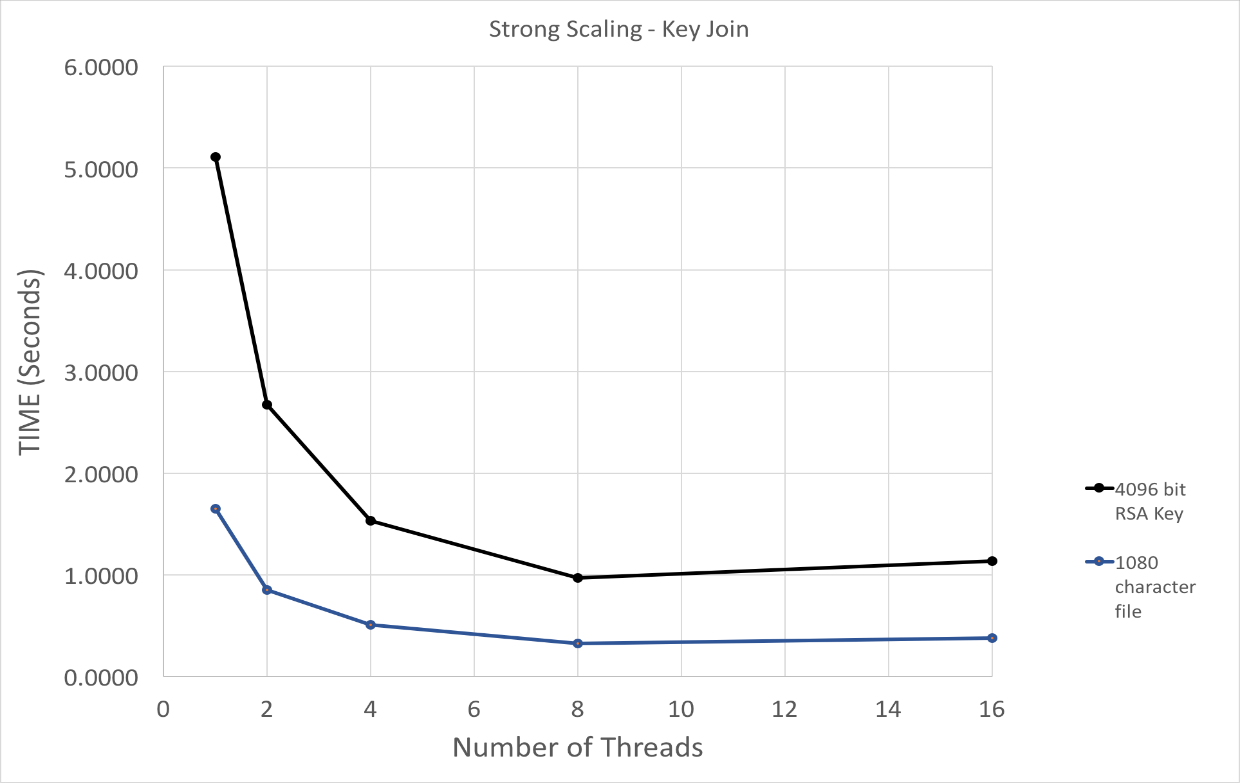
****

Figure 2: Results of joining all 255 keys to reproduce the secret from the RSA key and the 1,080-character file.

Key Join Results

|  |  |  |
| --- | --- | --- |
| **Threads** | **4096 Bit RSA Key** | **1080 Character File** |
| 1 | 5.10 sec | 1.64 sec |
| 2 | 2.67 sec | 0.84 sec |
| 4 | 1.53 sec | 0.50 sec |
| 8 | 0.96 sec | 0.32 sec |
| 16 | 1.13 sec | 0.37 sec |

Table 2: Times taken to reassemble the 255 keys to reproduce the secret

#### 4.2 Weak Scaling

We discovered a second inner loop in the split\_string function that provided beneficial results in our weak scaling tests. Table [3](#x1-8001r3) shows that the times do not increase proportionally as we doubled both the input size and the number of threads.

Weak Scaling Test Results

|  |  |  |
| --- | --- | --- |
| Threads | Character Count | Time |
| 1 | 540 | 4.66 sec |
| 2 | 1080 | 4.71 sec |
| 4 | 2160 | 5.28 sec |
| 8 | 4320 | 5.86 sec |
| 16 | 8640 | 7.41 sec |

Table 3: Times taken after parallelizing the second loop in *split\_string* function.

### 5 CONCLUSION

We were successful in parallelizing Shamir’s secret sharing algorithm, achieving near linear speedup using OpenMP. Future work could extend this work to distributed memory architectures using message-passing frameworks like MPI, exploring whether the communication between nodes would be a limiting factor on speedup. Rabin’s secret sharing algorithm [3] would also be a good candidate for parallelism in future research. This sharing scheme operates under the assumption that each participant can broadcast a message, and each pair of participants can communicate secretly. We hope our work can serve as a stepping stone for future projects, as there is still a lot that can be done with secret sharing algorithms in the context of parallel and distributed systems.

### 6 References

[1] D. Bogdanov, “Foundations and properties of Shamir’s secret sharing scheme,”

Research Seminar in Cryptography, pp. 1–10, 2007. [Online]. Available: https:/ /pdfs.semanticscholar.org/540b/faa26cfafde5be79aadde37cb79f9d2daf76.pdf

.

[2] F. T. Penney. (). Original c implementation of Shamir’s secret sharing algorithm. original source code, [Online]. Available: <https://github.com/fletcher/c-sss>

[3] T. Rabin and M. Ben-Or, “Verifiable secret sharing and multiparty protocols with honest majority,” in Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing, ser. STOC ’89, Seattle, Washington, USA: ACM, 1989, pp. 73–85,

isbn: 0-89791-307-8. [Online]. Available:

http ://doi.acm.org/10.1145/73007.73014

[4] “OpenMP Application Programming Interface”. OpenMP Architecture Review Board, 2015. OpenMP [Online]. Available: http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf