PARALLELIZING SHAMIR'S SECRET SHARING ALGORITHM

1 ABSTRACT

This paper describes how Shamir's Secret Sharing algorithm can be parallelized for sharing large secrets with a large group of participants. Using an open source C implementation of Shamir's algorithm and the OpenMP API, we explored regions of the algorithm where running in parallel reduced the time for both key generation and joining the shares. We were able to see near linear strong scaling results, both with generating the keys and combining them to reproduce the secret. We also observed weak scaling when generating the key shares. By parallelizing Shamir's algorithm, a secret can be divided among many participants in a relatively short amount of time.

2 INTRODUCTION

Shamir's Secret Sharing scheme is a method for dividing a secret within 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. 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 parallel algorithms is broken into two categories: weak and strong scaling. Weak scaling is defined as when the problem size and number processes/threads increases proportionally, the execution time stays relatively the same. Strong scaling is defined as increasing the number of processes/threads and keeping a constant problem size, the execution time decreases each time another process/thread is added.

2.1 Background

Shamir's Secret Sharing algorithm, explained in [1], works by taking the number 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 to the degree t-1. The secret becomes the constant value in the polynomial equation.

Example 2-1:

$$4x^3 + 9x^2 + 3x + secret$$

The required threshold to reproduce the secret of the function listed in Example 2-1 would be 4 keys (t - 1 = 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 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 set out to explore the possible benefits of parallelizing Shamir's Secret Sharing algorithm. The overall focus of our project described and explained in this paper, was to speed up the process of generating key shares for large files between a large number of parties.

3 METHODS

Utilizing a single Dell PowerEdge R430 with a Xeon E5-2630v3 processor with 8 Cores, 2.4Ghz, Hyper Threaded, we approached this problem using OpenMP to take advantage of the API's parallel for loop feature. Specifically, we identified 3 regions of the code where parallelism could exist. All of our experiments and testing where conducted on the Dell PowerEdge, 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 test, consist of doubling the character count of the input file, while double 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. Specifically, 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 our parallel results, by the fact that the key joining function could reassemble the keys into the original text file that was input. After, implementing parallelism in the key generation stage of Shamir's algorithm, we switched our focus to implementing parallelism with the key joining function. The challenge with parallelizing the key joining stage of Shamir's secret sharing is correctly identifying variable scope and critical regions. The most important region to consider, is when each thread updates the secret after computing the Lagrange Interpolating Polynomials. Access to this region has to be synchronized. By correctly identifying these, we were able to parallelize the for loop that computes the Lagrange Interpolating polynomial of the function used to generate the keys.

4 RESULTS

We were able to significantly speed up the secret splitting and joining using OpenMP. Our approach shows substantial strong and weak scaling. Here are some of our results:

4.1 Strong Scaling

In Figure 1 and Table 1, we see that the times to create the key shares gets nearly cut in half every time we double the number of threads. So we can say the speedup is "near linear".

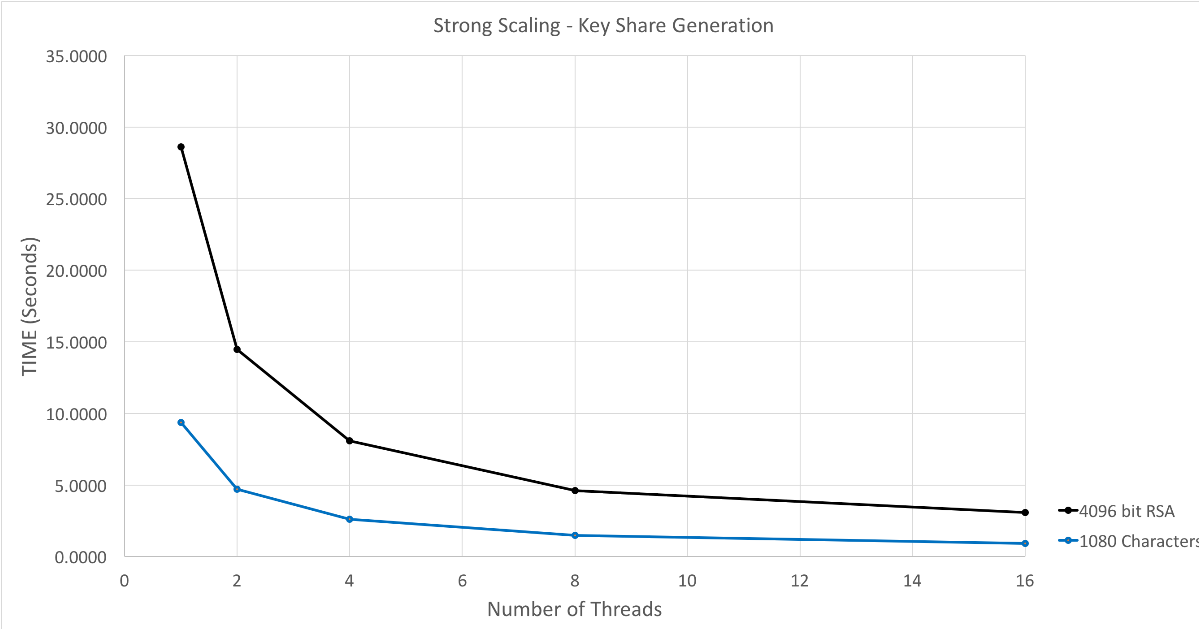

Figure 1: Results of Generating 255 keys with a required unlock threshold of 255 for a 4096 bit RSA key and 1080 character input file.

Key Generation Results

Threads	4096 Bit RSA Key	1080 Character File
1	28.6211	9.3661
2	14.4674	4.7160
4	8.0794	2.6117
8	4.6039	1.4698
16	3.0933	0.9234

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. It's import to note here, that we start to see increases in time at 16 threads when joining the keys back together. This is still an open research question, but we suspect that the cost of spawning 16 threads is greater than the problem size, therefore an increase in time occurs. Also, the critical region has to synchronize more often with 16 threads entering the critical region, which is likely another culprit for the increases.

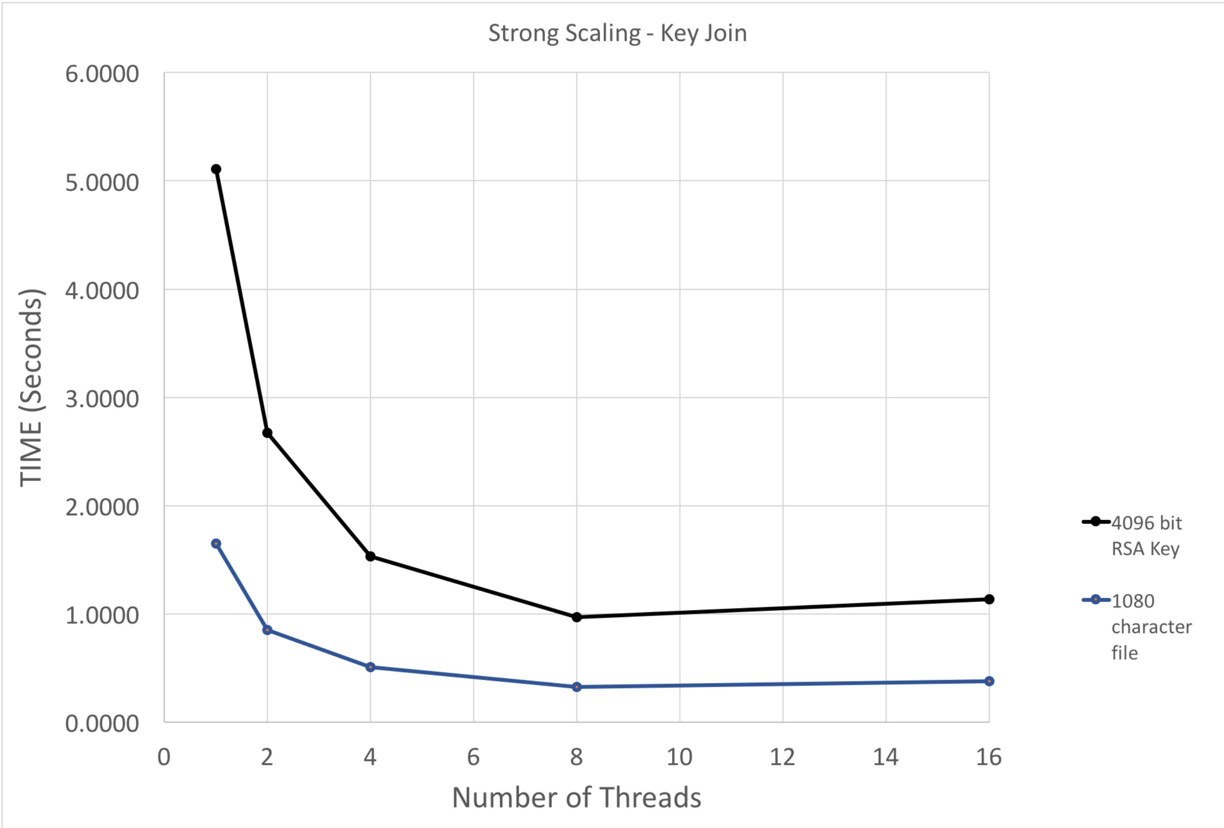

Figure 2: Results of joining all 255 keys to reproduce the secret from the RSA key and the 1080 character file.

Key Join Results

Threads	4096 Bit RSA Key	1080 Character File
1	5.1092	1.6499
2	2.6740	0.8492
4	1.5339	0.5051
8	0.9670	0.3247
16	1.1331	0.3786

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

4.2 Weak Scaling

We discovered second inner loop in the split_string function that provided beneficial results in our weak scaling test. Table 3 shows our time results for our Weak scaling test. We had positive results with our weak scaling tests. The times stay relatively the same as we double the input and number of threads. It's not perfect weak scaling because there are increases, but the increases are less than the increase in problem size.

Weak Scaling Test Results

Threads	Character Count	Time
1	540	4.6673
2	1080	4.7118
4	2160	5.2831
8	4320	5.8683
16	8640	7.4179

Table 3: This table shows our results of our weak scaling tests after parallelizing the second loop in split_string function.

5 CONCLUSION

It was very satisfying to get the amount of speedup and scaling that we did using only OpenMP. In our project, we did not make any attempt to decide whether or not it is useful to parallelize Shamir's in real world applications. We simply set out to study the parallelization of an existing implementation, and we were successful. Future work could be on using Shamir's with distributed memory parallelism using tools like MPI. It would be interesting to discover if it would continue to scale or if the communication between nodes would be a limiting factor on speedup. We concluded that the best approach to using MPI with Shamir's would be to write an implementation from scratch with the intent of having MPI compatibility. Rabin's secret sharing algorithm [3] would also be a good option for parallelism in future research. We hope our work can serve as a stepping stone for future projects, as there is still a lot that can be done with Shamir's as well as other secret sharing algorithms, in regards to parallel and distributed systems.

6 References

[1] D. Bogdanov, "Foundations and properties of shamirs 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.