James Madison University

Department of Computer Science

CS 470 Parallel & Distributed Systems

CS 470: Parallel Shamir's Secret Sharing

Author(s):
Joey Arbogast
Isaac Sumner

Submitted to:
Dr. Michael Lam

April 27, 2017

Honor Pledge: I have neither given nor received help on this project that violates the JMU Honor Code.

Joey Arbogast	Isaac Sumner
____________________	____________________
Signature	*Signature*
____________________	____________________
Date	*Date*

Contents

List of Figures

List of Tables

1 Background

1.1 Background

When we originally started this project and began looking into Shamir's Secret Sharing, we were unsure if there would be any benefit to parallelizing or distributing the algorithm. We discovered after experimenting with the original serial implementation that we found on Github [1], that the algorithm was very slow at generating key shares using large text files.

1.1.0.1 How Shamir's Secret Sharing Algorithm Works

The implementation of Shamir's Secret Sharing algorithm that we used works by taking a specified number of keys you would like to generate and the required number of those keys that need to be combined in order to reproduce the secret. The secret can be a password, but we decided to test it's limits using text files and RSA keys as input.

The algorithm is best explained on Wikipedia [2]. The simplest explanation is that you provide the number keys you want to share (n) and the threshold that is required to unlock the secret (t). The algorithm generates the keys by generating a polynomial equation to the degree t-1. It does this by looping t-1 times and generating random numbers as coefficients. The secret becomes the constant value in the polynomial equation.

Example:

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

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. These XY pairs become the keys.

To reproduce the secret the function join_strings and join_shares is called. The shares contain a few ingredients, which is why Shamir's Secret Sharing is not technically considered encryption. Each key contains at the beginning of the key, the number of shares and the required unlock threshold in hexadecimal. The secret is reproduced from a mathematical equation called Lagrange Interpolating Polynomial [3]. This equation is implemented in the join_shares function in our program and is used to determine the equation that produced the key shares, given a certain number of XY points. The join_shares function was a region we explored parallelism and were successful.

2 Methods, Experiments, Results & Conclusion

2.0.1 Methods

Our methods for approaching this problem were to study firstly what the serial version of the program was doing. After we felt we had a good grasp on what was going on we decided that OpenMP would be a good fit for parallelism of the for loops.

2.0.1.1 Split_number Function

Our focus at first was studying the function split_number, which dealt will computing the key shares. We looked for loop dependencies in the split_number function, but could not find any dependencies. We began experimenting with OpenMP parallel pragmas with the first for loop in the split_number function which computes the random coefficients to be used with the polynomial function.

The first for loop was relatively easy to parallelize, the main difficulty was understanding that the coefficients array and shares needed to be shared among the threads. We got some speedup just by parallelizing this first loop. After that loop was parallelized we experimented with the second for loop. We attempted to parallelize the inner loop, but this seemed to break joining the secret back up later, because we were getting random characters. We found that only the outer loop could be parallelized. Figure 1 shows our final OpenMP regions in the split_number function.

```
int * split_number(int number, int n, int t) {
	int *shares;
	int coef[t];
	int x,i;
	shares = malloc(sizeof(int)*n);
	coef[0] = number;

#	pragma omp parallel default(none) shared(num_threads, prime,n,t,coef,shares) private(number,x,i)
	{
		num_threads = omp_get_num_threads();
#	pragma omp for schedule(static, (t - 1))
		for (i = 1; i < t; ++i)
		{
		/* Generate random coefficients */
			coef[i] = rand() % (prime - 1);
		}

#	pragma omp for schedule(static,2)
		for (x = 0; x < n; ++x)
		{
			int y = coef[0];
			/* Calculate the shares */
			for (i = 1; i < t; ++i)
			{
				int temp = modular_exponentiation(x+1, i, prime);
				y = (y + (coef[i] * temp % prime)) % prime;
			}

			/* Sometimes we're getting negative numbers, and need to fix that */
			y = (y + prime) % prime;
			shares[x] = y;
		}
	}

```

Figure 1: Our final OpenMP parallel regions in split_number function.

2.0.1.2 Split_string Function

The split_string function, was not extremely valuable in speeding up the key share generation time originally. The first for loop shown in Figure 2, shows the first parallel loop. The loop simply allocates space for the shares array, and then puts the hex values of the number of shares, threshold and some fake digits which are explained by the original developer of the program. This loop only gave us slight increases in speed. The major development was parallelizing the inner loop of the second for loop, which gave us better speed up in our weak scaling test. We experimented with the outer loop as well, but all attempts at parallelizing this loop proved that it was not working when we tried to join the secret, meaning the keys computed were wrong, which was verifiable by the fact that the reproduced secret was random ASCII characters and symbols.

```c
char ** split_string(char * secret, int n, int t) {

        char **shares = malloc(sizeof(char *) * n);
        int len = strlen(secret);
        int i;

#       pragma omp parallel for schedule(static, t -1) default(none) shared(n,t,len,secret,shares) private(i)
        for (i = 0; i < n; ++i)
        {
                /* need two characters to encode each character */
                /* Need 4 character overhead for share # and quorum # */
                /* Additional 2 characters are for compatibility with:

                        http://www.christophedavid.org/w/c/w.php/Calculators/ShamirSecretSharing
                */

                shares[i] = (char *) malloc(2*len + 6 + 1);

                sprintf(shares[i], "%02X%02XAA",(i+1),t);
        }


        /* Now, handle the secret */
    // This doesn't work I couldn't remember if you tried this
//#   pragma omp parallel for default(none) shared(len, shares, secret, n, t)
        for (i = 0; i < len; ++i)
        {
                // fprintf(stderr, "char %c: %d\n", secret[i], (unsigned char) secret[i]);
                int letter = secret[i]; // - '0';
                if (letter < 0)
                        letter = 256 + letter;

                //fprintf(stderr, "char: '%c' int: '%d'\n", secret[i], letter);
                int * chunks = split_number(letter, n, t);
                int j;
#       pragma omp parallel for schedule(static, 2)
                for (j = 0; j < n; ++j)
                {
                        if (chunks[j] == 256) {
                                sprintf(shares[j] + 6 + i * 2, "G0");   /* Fake code */
                        } else {
                                sprintf(shares[j] + 6 + i * 2, "%02X", chunks[j]);
                        }
                }

                free(chunks);
```

Figure 2: Our final OpenMP parallel regions in split_string function.

2.0.1.3 Join_shares Function

The join_shares function is the main function that deals with reproducing the secret from the key shares. This function was the most challenging function to parallelize, as you can see in Figure 3, the variable scope was incredibly hard to figure out. After identifying the variable scope of the parallel region, we still had issues with the function producing garbage output instead of the actual secret. We discovered that the threads were all trying to update the secret at once, so we protected the region with an `omp critical` statement shown in Figure 3, which solved the problem.

```c
int join_shares(int *xy_pairs, int n) {
	int secret = 0;
	long numerator;
	long denominator;
	long startposition;
	long nextposition;
	long value;
	int i;
	int j;

#	pragma omp parallel default(none) shared(num_threads, secret, n, prime, xy_pairs) \
		private(numerator, denominator, value, startposition, nextposition, i, j)
	{
		num_threads=omp_get_num_threads();
#	pragma omp for schedule(static, 2)
		for (i = 0; i < n; ++i)
		{
			numerator = 1;
			denominator = 1;
			for (j = 0; j < n; ++j)
			{
				if(i != j) {
					startposition = xy_pairs[i*2];
					nextposition = xy_pairs[j*2];
					numerator = (numerator * -nextposition) % prime;
					denominator = (denominator * (startposition - nextposition)) % prime;
					//fprintf(stderr, "Num: %lli\nDen: %lli\n", numerator, denominator);
				}
			}

			value = xy_pairs[i *2 + 1];
#	pragma omp critical
			secret = (secret + (value * numerator * modInverse(denominator))) % prime;
		}
	}

	/* Sometimes we're getting negative numbers, and need to fix that */
	secret = (secret + prime) % prime;

	return secret;
```

Figure 3: Our final OpenMP parallel regions in join_shares function.

We experimented with some of the other for loops found in other functions of the par_shamir.c file, but were unable to parallelize any of them with OpenMP. Particularly, we were able to parallelize the join_strings function in several places in the for loop, but this actually slowed the secret joining down by seconds every time we double the thread count, so it was removed.

2.0.2 Experiments

2.0.2.1 OpenMP For Loop Scheduling

We also experimented with scheduling of the for loops that we were able to parallelize. Table **??** shows experimentation of scheduling the for loops in the split_number function of the par_shamir.c file. It is not all inclusive, as we conducted numerous combinations of scheduling, but gives you an idea of a few that we tried. The highlighted scheduling was the best one we found, where the first for loop is schedule static, t - 1 chunk size, and the second for loop is schedule static, 2.

				split_number Scheduling		
Threads	none	static,1	static,2	dynamic,2	guided	(static, t-1) and (static,2)
1	9.3375	9.3358	9.341	9.3806	9.3631	9.3601
2	4.7731	4.7715	4.7713	4.7892	4.783	4.7102
4	2.7715	2.8485	2.8686	2.8564	2.9104	2.6125
8	1.5565	1.554	1.5599	1.5418	1.5388	1.4701
16	0.9973	1.0134	1.0038	0.9977	1.0028	0.9255

Table 1: This table shows some of our experimentation times with for loop schedules in the split_number function, it is not all inclusive

Table **??** shows our final results from scheduling split_string and Table **??** shows the final results of scheduling the join_shares function.

	split_string
Threads	(dynamic,2) and 2nd Loop (static,2)
1	9.3522
2	4.7158
4	2.6101
8	1.4696
16	0.9214

Table 2: This table shows the final scheduling results of split_string

	join_shares Scheduling	
Threads	none	static,1
1	1.6668	1.6659
2	0.8743	0.8682
4	0.5952	0.5071
8	0.3273	0.3283
16	0.3713	0.2782

Table 3: This table shows the final scheduling results of join_shares

2.0.2.2 MPICH

One of our goals was to implement parallelism in a distributed fashion using MPI. We wanted to see if the secret splitting and joining would continue to scale beyond the results we had already achieved with OpenMP. We started by trying to parallelize areas in the code that we already had success parallelizing with OpenMP. When we started using MPI, we ran into major issues, because of things like I/O. We discovered that when one process would open the file to write the shares to, the other processes would block because, they were also trying to open the file. We decided that only one process needed to write the shares to a file. Our plan was to have each process compute part of the shares being generated for each byte, since we were already able to parallelize this with OpenMP. See Figure: 4

```c
/*
    split_number() -- Split a number into shares
    n = the number of shares
    t = threshold shares to recreate the number
*/
int * split_number(int number, int n, int t) {
    int *shares;
    int coef[t];
    int local_coef[t/nprocs];
    int x,i;
    shares = malloc(sizeof(int)*n);
    coef[0] = number;
    int *loc_shares = malloc(sizeof(int)*n/nprocs);

//# pragma omp parallel shared(nprocs,prime,t,coef,shares) private(number,x,i)
{
    num_threads = omp_get_num_threads();
//# pragma omp for schedule(static, (t - 1))
    for (i = 1; i < t; ++i)
    {
    /* Generate random coefficients */
        coef[i] = rand() % (prime - 1);
    }
//# pragma omp for schedule(static, 2)
    for (x = 0; x < n/nprocs; ++x)
    {
        int y = coef[0];
        /* Calculate the shares */
        for (i = 1; i < t; ++i)
        {
            int temp = modular_exponentiation(x+1, i, prime);
            y = (y + (coef[i] * temp % prime)) % prime;
        }

        /* Sometimes we're getting negative numbers, and need to fix that */
        y = (y + prime) % prime;
        loc_shares[x] = y;
    }
}
    MPI_Gather(loc_shares, n/nprocs, MPI_INT, shares, n/nprocs, MPI_INT, 0, MPI_COMM_WORLD);
    return shares;
}
```

Figure 4: Using MPI to Gather the shares on Rank 0

Unfortunately, this approach was unsuccessful. When we were finally able to stop getting deadlock and segmentation faults, the output was incorrect. Valid shares were generated, but, whenever we joined the shares again, we got back nonsense. This was tested using a value of `n` divisible by the number of processes. We ran into similar (if not worse) problems when using MPI with other parts of the implementation.

2.0.3 Results

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

2.0.3.1 Strong Scaling

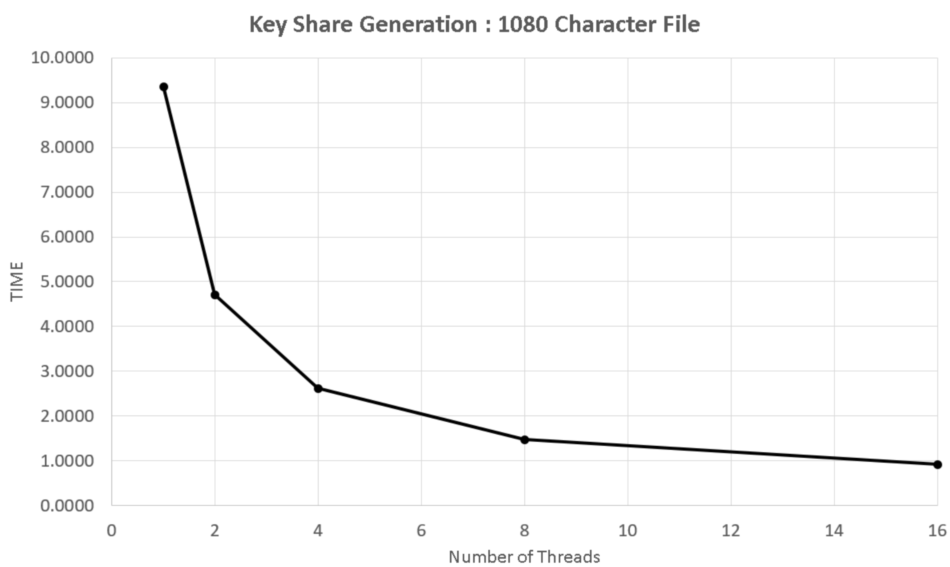

Figure 5: Results of Generating 255 keys with a required unlock number of 255 for a 1080 character input file.

Threads	Time
1	9.3661
2	4.7160
4	2.6117
8	1.4698
16	0.9234

Table 4: This table shows the times for creating the shares with a 1080 character input

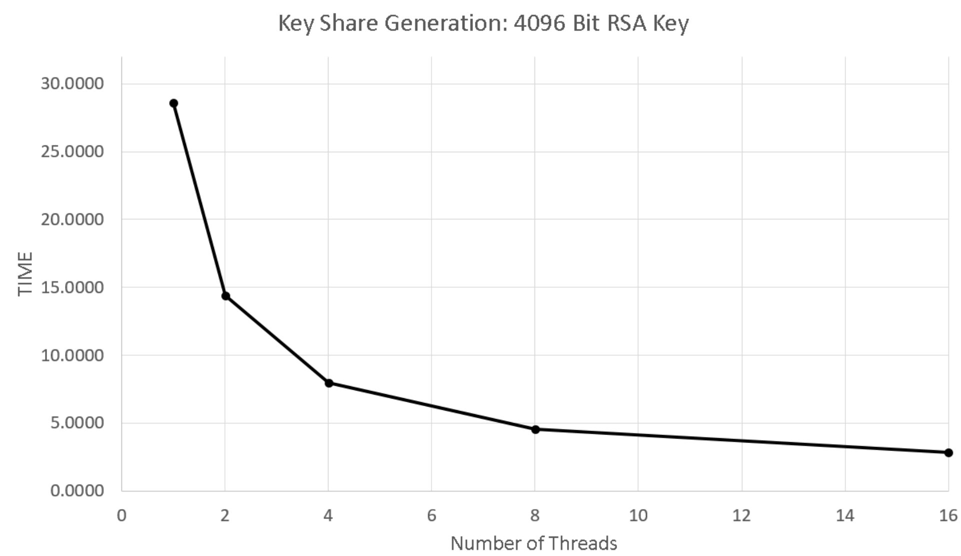

Figure 6: Results of Generating 255 keys with a required unlock number of 255 for a 4096 Bit RSA key(The ASCII Character Representation of the key)

Threads	Time
1	28.5968
2	14.4024
4	7.9756
8	4.5282
16	2.8208

Table 5: This table shows the times for creating the shares with a 4096-bit RSA key

Here we see that the times to create the shares get nearly cut in half every time we double the number of threads. So we can say the speedup is "near linear".

Figure 7: Results of reproducing the secret from the keys generated from the 1080 character input file.

Threads	Time
1	1.6659
2	0.8700
4	0.5788
8	0.4194
16	0.3680

Table 6: This table shows the times for joining the shares and outputting a 1080 character file

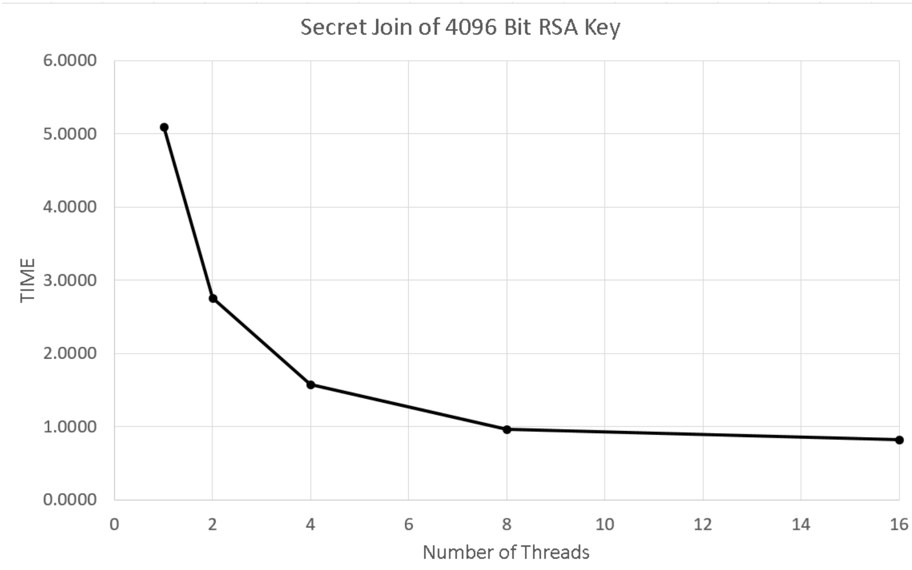

Figure 8: Results of reproducing the secret from the keys generated from the RSA key input file.

Threads	Time
1	5.0986
2	2.7584
4	1.5730
8	0.9620
16	0.8216

Table 7: This table shows the times for joining the shares and outputting a 4096-bit RSA key

We get similar scaling results when joining the shares back together to get the original secret. (It takes much less time to join the shares than to split the secret in general).

2.0.3.2 Weak Scaling

Our initial weak scaling results are shown in Figure 9. These results were before we found out that the second inner loop in the split_string function could be parallelized and provide beneficial results in our weak scaling test. Table 8 shows the corresponding time illustrated in Figure 9.

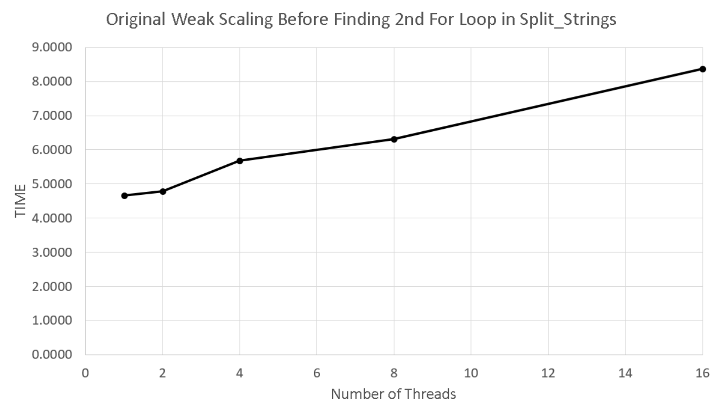

Figure 9: Results of our original weak scaling test before finding an inner for loop that improved the weak scaling

Threads	Time
1	4.6713
2	4.7812
4	5.6829
8	6.3220
16	8.3774

Table 8: This table shows our original results of our weak scaling tests

When comparing our original weak scaling results shown in Figure 9 and Table 8, with the results in Figure 10 and Table 9, you can see that we were able to improve the scaling results of 16 threads by almost a second.

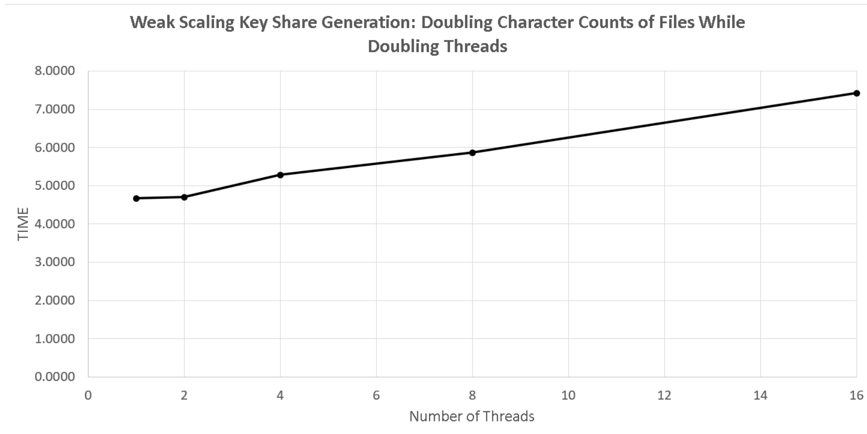

Figure 10: Results of our final weak scaling test after parallelizing the inner for loop in split_string

Threads	Time
1	4.6673
2	4.7118
4	5.2831
8	5.8683
16	7.4179

Table 9: This table shows the results of our weak scaling tests

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.

2.0.4 Future Work

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. Future work could be on using Shamir's with MPI and parallelizing Rabin's secret sharing method.

2.0.4.1 MPI

An extension of our work could be to find a way to parallelize Shamir's using MPI. It would be interesting to discover if it would continue to scale or if the communication between nodes 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. Separating the shares generation between processes seems to be possible, because the coefficients and y values are generated independently using pseudo-random numbers and modular exponentiation respectively.

2.0.4.2 Rabin's Secret Sharing

Verifiable Secret Sharing, as created by T. Rabin and M. Ben-Or, can be implemented under the assumption that each participant can broadcast a message, and each pair of participants can communicate secretly [4]. This may be a more reasonable secret sharing approach to parallelize. Unfortunately, it is much less well known, and the reading material is limited. From what we have heard, it could be worth pursuing a parallel implementation.

2.0.5 Conclusion

We had a lot of fun learning about Shamir's Secret Sharing and trying to find ways to improve the implementation through parallelization. It was hard to understand the mathematics and underlying algorithms at first, but, after working with Shamir's for a while, we were able to understand what was actually going on. 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.

3 References

[1] F. T. Penney. (). Original c implementation of shamir's secret sharing algorithm. original source code, [Online]. Available: https://github.com/fletcher/c-sss.

[2] Wikipedia. (). Shamir's secret sharing, [Online]. Available: https://en.wikipedia.org/wiki/Shamir\%27s_Secret_Sharing.

[3] Wolfram. (). Lagrange interpolating polynomial, [Online]. Available: http://mathworld.wolfram.com/LagrangeInterpolatingPolynomial.html.

[4] 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.