Distributed Systems Final Project: RSA Factorization

THESIS

Mark Demore

Brennen Garland

Chad Willis

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

Abstract

In this paper, we will explain the background concepts necessary to understand the architecture and algorithm used in this project. The two main concepts to become familiar with are grid computing, and Rho Pollards Algorithm. We created a computational grid which is able to perform prime factorization on very large numbers, targeted specifically at finding the two primes of RSA numbers. The program uses a single, command line interface that creates three servers that try to reduce the calculation of a large RSA number. Each node is able to work concurrently to solve to problem more efficiently than if each node was only capable of doing work with a single thread. To prove its effectiveness, we ran the program through a series of tests with varying parameters to show that it not only works, but also works in an efficient manner. We recorded the subsequent data and have displayed it in a table and graph for easy consumption. We conclude that, with a large enough number, our implementation overpowers a single threaded implementation in terms of speed.

Table of Contents

Distributed Systems Final Project: RSA Factorization

I. Introduction

1.1 RSA

RSA encryption works by creating large semi-primes, numbers that are made by multiplying together two large prime numbers. These large semi-primes therefore take a lot of time and computational power to process. A distributed, multi-threaded system would greatly reduce the time it takes to factor these semi-primes using Pollard's rho algorithm. Our system uses this method and successfully does so when compared to the time it takes a single process implementation of this algorithm.

1.2 Grid System to Factor Large Numbers

We created a a computational grid which is able to perform prime factorization on very large RSA numbers. The program is a single interface which spins up 3 network nodes in order to solve the problem. The single threaded code provided was modified to allow each node to do work with multiple threads. With that, each node is able to work concurrently to solve to problem more efficiently than if each node was only capable of doing work with a single thread. To prove its effectiveness, we ran the program through a series of tests with varying parameters to show that it not only works, but also works in an efficient manner. We recorded the subsequent data and have displayed it in a table and graph for easy consumption.

II. Background Concepts

2.1 Grid Computing

A grid computing system was set up to solve the factorization problem. Grid computation is ideal when trying to solve very data intensive, difficult problems. The word grid comes form the idea of a power grid which has access to a source of electricity [6]. But when speaking about grid computing, a grid is a type of infrastructure that provides high computational capacity to a distributed system by making use of shared, but geographically distributed resources [2]. In this case, our grid is not geographically separated. In grid computing, a problem such as factorization, is broken up into smaller pieces and scheduled to the available resources in the network system to be worked on. The network consists of a cluster of connected homogeneous computers, meaning they are similar computers - running the same operating system and hardware, for example [2]. This network of computers essentially becomes a supercomputer, with every computer having access to the processing power and resources to be able to solve one large problem. Grid systems are designed with scalabiltiy in mind because the amount of resources needed could increase. For fault tolerance, a grid system should be able to reallocate resources and resuming the task at hand. Grid systems are meant to be transparent, meaning from the end users perspective, the system is one computer.Grid computing also supports parallel programming, which is necessary to apply to Pollards Rho Algorithm. With these concepts in mind, we created our grid system.

Grid computing also comes with some drawbacks. Application of a grid system is narrow because it is not ideal to use when a problem is smaller in scope. For example, regular cluster computing would be quicker for medium sized problems. Grid computing is not the easiest system to set up. Open source grid software development

tools are limited, and no common standard exists when building a grid system.

2.2 Pollard Rho Algorithm

Pollards Rho algorithm is the workhorse behind our system. It was developed in 1975 to use a small amount of space with its run time being proportional to the square root of the size of the smallest prime factor of the of the composite number being factored. The algorithm chosen to be implemented over the Sieve of Erathosthenes when one wants to factor very large numbers because numbers within the 50-100 range would give the Sieve problems. The downside of Pollard Rhos is that with an increasing number size, the number of cores needed to factor the number increases [1]. Also, the algorithm is slower for numbers that have larger factors. The algorithm, however, is very parallelizable, which we exploit using our multiple nodes. According to Anjan Koundinya, the author of *Performance Analysis of Parallel Pollard's Rho Algorithm*, one can expect a parallel algorithm speed up of three times when the number of cores used is increased cores increased by 2 fold [1].

2.3 RSA Encryption

RSA was developed as a means of public key encryption that takes advantage of the computational difficulty of prime factorization, emphasizing the benefits of a potential distributed system solution. RSA is an asymmetric method of encryption, where the large semi-prime is used to encrypt the message and the two prime factors need to be known for decryption. It is one of the most commonly used encryption methods in the world today.

III. Design and Methodology

3.1 Language and Operating System Choice

The choice to use of a compiled language like C++ was an easy one because it is much faster than an interpreted language such as Python. An important goal is making sure that we have an efficient end product, and C++ can deliver that. Also, our team has been working with C++ exclusively in recent classes so it only made sense to stick with what we are comfortable with. We also knew that the OpenMP tools existed for C++, which we will discuss later, and that OpenMP would be straightforward to set up multithreading for our system. We also knew that from Homework 3 that OpenMP is clearly the choice for ease of use, and solid speed.

In regards to operating system, we chose Linux for development. Linux is superior to Windows in our teams opinion because of the ease of programming, and the ease of installing libraries and tools to help us program. For example, it was simple to install boost in the Linux terminal with the sudo apt get command. It also much easier to jump right into networking in a Linux system compared to Window. Linux is also more secure than Windows because most hackers tend to attack Windows users.

3.2 Grid Computing Design

A grid computing system was chosen because it is the perfect architecture to solve the factorization problem. We took into account what concepts are needed to achieve grid computing. We spin up multiple nodes acting as our grid. These nodes are connected and have access to shared resources. And when used together, have more processing power to do more work. We want to take advantage of the shared resources from each node to have a quick performing program. Also, it makes sense to use a grid computing system compared to a regular cluster system because all of

our nodes are homogeneous. In regards to fault tolerance, we have implemented error checking for server errors and user input errors. If our communication fails, a message is displayed. For scalability, we could implement the addition of more nodes. Also, our system is transparent because from the end users perspective, it is all a singular system.

3.3 Pollard Rho Algorithm Design

To enhance the provided code to become multithreaded, we used the OpenMP library for C++. Specifically, to create threads we used pragma omp parallel. This powerful tool splits up loop iterations, and forms a team of threads to be assigned to each of these iterations to do work concurrently. This team of threads executes the same task which is the following statement block [5]. So, when one of the threads completes the task, the rest of the threads are halted.

In regards to design of our algorithm to calculate factorization, we stayed true to the Pollard Rho algorithm. As input, the algorithm takes an integer n to be factorized, and a pseudo-random number function f modulo n. The psuedo-random number function is in the form $x^2 + c$ [1]. The c value is the number of iterations to be performed to find a factor. When our nodes are spun up, they are each given the number n that is going to be factored and a different random function. Each node tries to compute a factor, and because each value of c is different, each node requires a different amount of time to produce a result. Once, a node has a result, the problem solving on the rest of the nodes halts. The number n is then divided by the resulting factor to create another number n to be worked on [1]. Yash Varyani of Geeks for Geeks simply explains the whole algorithm design as follows [3]:

Start with random x and c. Take y equal to x and f(x) = x2 + c. 1. While a divisor isn't obtained 1. Update x to f(x) (modulo n) 2. Update y to f(f(y)) (modulo

n) 3. Calculate GCD of —x-y— and n 4. If GCD is not 1 1. If GCD is n, repeat from step 2 with another set of x, y and c 2. Else GCD is our answer

3.4 Architecture Choice

Our system can be described as a hybrid between being centralized and decentralized architecture. The system is partly centralized because there is a single entry point that spins up all the nodes. But at the same time, the nodes that are spun up work independently and can be considered decentralized. With that, we can describe the system as push-based because updates are propagated to all the nodes without those nodes even asking for the updates. The nodes are actively listening for messages from other nodes but not asking for updates, so it is not pull-based. The nodes can be considered asynchronous because as they listen, they are also checking for primes. We also have middleware in our prime server. We constructed the middleware so that when one node finishes problem solving, it interrupts the algorithm that is occurring and returns whether or not the number is prime.

3.5 Combining Both Designs Together

The combination of these designs required us to work around a few problems. First, we had to fix the threaded algorithm. We decided to use a sort of 'interrupt'. This was just a boolean that would stop the algorithm if the server had received information that another server had found the factorization. This way the nodes would not continue running and wasting resources but instead would finish all in the same time. The second fix was working around our traditional server architecture where the 'select' function did not have a timeout. Without the timeout, the server would block until a new message was received and so no servers were talking to each other. By setting a timeout value of one second, the servers started talking to each

other. The server communication is fairly limited, only one server sends out a message if it has found a prime. If the number is indeed prime, then the servers will never talk to each other.

3.6 Testing Methodology

The testing was fairly straight-forward. We started out with lower bit RSA numbers and built up to higher ones. In this way, we could see any speed difference. We tested each on the same machine, a four core desktop computer running headless Ubuntu server. The runtime of the entire program was not measured, but instead the runtime of the calculation. Due to each node being seeded randomly, the hope is that one node will finish faster than the others and let them know they can stop.

IV. Results and Analysis

4.1 Results

We gathered our results using RSA numbers generated by bigprimes.org and running our factorization test using the provided single-process code as well as our distributed server, multi-threaded code. We tested the RSA factorization on 64-bit, 86-bit, 90-bit, 96-bit, and 100-bit RSA numbers. Our program is capable of processing up to 256-bit RSA numbers using define directives, but on our 4-core machines the processing time was far too long to be conducive for testing. Our results for these tests are shown below in fig. 2.

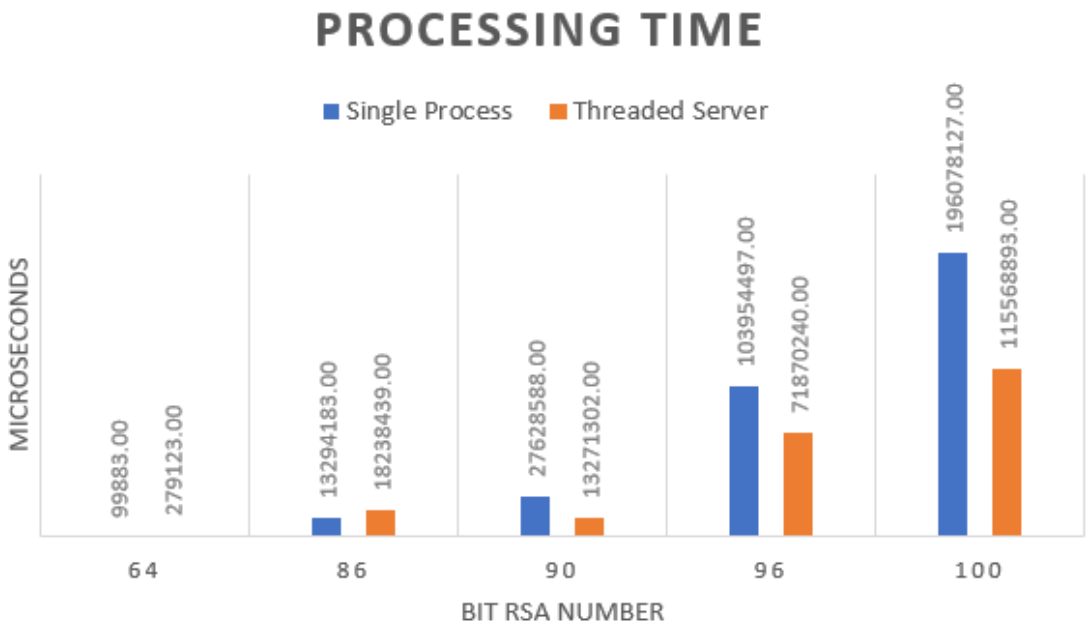

Figure 1: Comparative Processing Time

4.2 Analysis

An important thing to note is that the single process is faster than the threaded server for 64-bit and 86-bit RSA numbers. This is a result of the overhead for spawning servers and threads to tackle the factorization. For these smaller numbers, it is more effective to run the algorithm than spend time spinning up more resources to seed the algorithm differently in an attempt to find it sooner.

However, this changes at the 90-bit RSA number, and the threaded server gets much faster, compared to the single process method, with much larger numbers. It is worth noting that these results will vary, given the inherent randomness of the algorithm. In some cases the threaded server could be even quicker, and in a rare situation, the single-process may start the algorithm much closer to the prime and finish sooner.

V. Conclusions

Without a doubt, grid computation has allowed us to solve the problem of prime factorization faster than if we had used a single threaded approach. With computationally extensive problems, grid computation is ideal. But, as noted in the results, with smaller numbers, specifically 64-bit and 86-bit numbers, it is quicker to use a single threaded approach.

5.1 Future Work

The distributed system presented could definitely be expanded on. For one, we could make the program able to spin up a variable number of nodes, as opposed to fixed at 3 nodes. The program could also be extended to work on different difficult problems with the architecture that we have developed. To find the full factorization of a prime number could be implemented, as opposed to just finding the factors of an RSA number.

Appendix A. Software Dependencies and Compile/Install Instructions

Our codebase is available on GitHub (https://github.com/mdemore2/CSCE689_FinalProject).

Our software was developed for linux and is dependent upon the BOOST and OpenMP libraries, which can be easily installed by running 'sudo apt-get install libboost-dev' and 'sudo apt-get install libomp-dev' from the linux terminal.

Once the dependencies are installed, we use cmake as our build tool. We found this to be the easiest tool and better for cross-platform development in the future, if needed.

To build and run this program follow the below steps:

1. Install the above mentioned dependencies.
2. Move to the projects root directory, *CSCE689_Final Project*.
3. Run *cmake CMakeLists.txt*, this will build the Makefile.
4. Run *make* to create the executable.
5. To run the executable use *./prime_server [RSA number to factor]*

Appendix B. Extra Data

Bits	Factor 1	Factor 2	Single Process	Threaded Server	Ratio
64	1688169221.00	6617805671.00	99883.00	279123.00	2.7945
86	1540256347239.00	3665869318777.00	13294183.00	18238439.00	1.371911
90	13866199818883.00	62146804635659.00	27628588.00	13271302.00	0.480347
96	440430596527111.00	130108466228801.00	103954497.00	71870240.00	0.691362
100	2038479802313130.00	42484677222223.00	196078127.00	115568893.00	0.589402

Figure 2: Comparative Processing Time - Table

Bibliography

1. Koundinya, Anjan K., et al. *"Performance Analysis of Parallel Pollard's Rho Algorithm."* 2013. EBSCOhost, doi:10.5121/ijcsit.2013.5214.

2. Singh, Manjeet. *"An Overview of Grid Computing."* 2019 International Conference on Computing, Communication, and Intelligent Systems (ICCCIS), International Conference On, Oct. 2019, pp. 194–198. EBSCOhost, doi:10.1109/ICCCIS48478.2019.8974490.

3. Varyani, Yash. *"Pollard's Rho Algorithm for Prime Factorization."* GeeksforGeeks, 10 Feb. 2018, www.geeksforgeeks.org/pollards-rho-algorithm-prime-factorization/.

4. Pollard, J.M. *"A Monte Carlo Method for Factorization."* 1975.

5. Yliluoma, Joel. *"Guide into OpenMP: Easy Multithreading Programming for C"* , June 2016, bisqwit.iki.fi/story/howto/openmp/.

6. Zhang, Shuai et al. *"The Comparison between Cloud Computing and Grid Computing."* ICCASM 2010 - 2010 International Conference on Computer Application and System Modeling, Proceedings. Vol. 11. N.p., 2010. ICCASM 2010 - 2010

7. Rivest, R. L., A. Shamir, and L. Adleman. *"A Method for Obtaining Digital Signatures and Public-Key Cryptosystems."* Communications of the ACM 21.2 (1978): 120–126. Communications of the ACM. Web.

8. https://en.wikipedia.org/wiki/Pollard%27s_rho_algorithm