DRS: Project Submission

Lt Luis Sepulveda

Lt. Ricky Anderson

TSgt Daniel (Dare) Oke

Air Force Institute of Technology

CS689-01 Distributed Software Systems

Lt Col George E. Noel, Ph.D.

March 17,2020

**Introduction**

Our team’s code submission can be found at <https://github.com/DRS-CS689-Project/DRS2.0>

In today’s data driven world, high performance computing (HPC) is emerging as an interesting platform for those who look to gain deep insight into hot topics such as genomics, computational chemistry, seismic imaging etc. Initially adopted by research scientists who needed to perform complex mathematical calculations, HPC’s popularity has expanded to a wider number of fields of study. “Environments that thrive on collection of analysis and distribution of data -and depend on reliable systems to support streamlines workflow with immense computational power – need HPC, “says Dale Brantly, director of systems engineering at Panasas, an HPC data storage-system provider (Edwards 2019). This paper elaborates on our team project, in which we designed a grid based high performance computing system which solves prime factorization of very large numbers. While accuracy is assumed, prime performance measurement is speed. Now let us examine some key concepts.

**Prime Factorization**

Prime numbers are a set of all numbers that can only be equally divided by one and themselves. Examples of prime numbers include 2, 3,5 and 7. What very little people know is the importance of prime numbers and how the mathematical logic behind them has resulted in vital applications in our modern history today. Mathematician have been able to show that any whole number can be expressed as a product of primes. Only primes and nothing else. This rule is called the prime factorization rule (GeeksforGeeks 2016).

The task of prime factorization may seem like a cool mathematical oddity at first but as the number to be factorized gets bigger, it gets more challenging. The best mathematicians and scientist have been able to determine that it is totally impossible to find a completely efficient algorithm for factorizing large numbers into primes. There is some limit to the size of number we can factorize into prime numbers. This fact is absolutely key to modern computer security. To put this in another way, anything that computers can easily do without being able to easily undo will be of interest to computer security. Modern encryption algorithms exploit the fact we can easily take two large primes and multiply them together to get a much larger number but that no computer has yet been created that can take the much larger number and very quickly figure out which of the two primes went into making it (GeeksforGeeks 2016).

**Pollards Rho Algorithm of Factorization**

The Pollard’s Rho prime factorization is very fast for large composite numbers with small factors. This paper will describe our distributed system which performs prime factorization on large numbers using Pollard’s Rho algorithm. Let us look at some of the key concepts employed.

* Two numbers are said to congruent modulo n (x = y modulo n) if
  + Each leave the same reminder when divided by n
  + The absolute of their difference is a multiple of n
* Birthday Paradox: When assessing a set of people, the probability of two persons having the same birthday is high.
* Floyd’s Cycling Algorithm. If the tortoise and hare start at the same point and move in a cycle, as such as the hare goes twice the speed of the tortoise, they are bound/sure to meet at some point.

Our Algorithm is shown below

Start with random x and c. Take y equal to x and f(x) = x2 + c.

While a divisor isn’t obtained

Update x to f(x) (modulo n) [Tortoise Move]

Update y to f(f(y)) (modulo n) [Hare Move]

Calculate GCD of |x-y| and n

If GCD is not 1

If GCD is n, repeat from step 2 with another set of x, y and c

Else GCD is our answer

Once we are given a number N, our system generates two sets of random numbers using a random number generator. The two random Numbers are set to variables x and c. Variable x is copied into a variable named y. While a divisor is not yet found, the tortoise move is performed on x and the hare moves is performed on y. The greatest common denominator of the difference of x-y and n is computed. If the GCD is equal to n repeat steps (see above). If GCD is not n and it is not 1, it must be our answer!

**Testing Performed**

Our application mainly comprises of a main server and which allocates jobs to other servers. The other servers perform all the computation involved in obtaining the prime factors and return as output to the server a prime factor or a set of prime factors. To test our system, we multiplied all the numbers from our output to ensure that we arrived back at N. More detailed explanation will be given in subsequent chapters. See diagram below to illustrate (we assume N = 12)

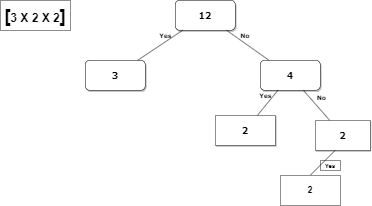


Figure 1: Hierarchical representation of testing approach

**Concepts (Background)**

An iterative Server is a server that handles request and returns a response to the requesting client. It iterates through each client, handling it one at a time. The main server (DivFinder.cpp) in our project was modeled as an iterative server. A concurrent Server handles multiple clients request at the same time by passing them to a thread or a process after which it waits for next incoming request. The server may call a fork function, creating one child process for each child. An alternative is also to pass client request to threads. Our design utilized an arbitrary number of concurrent servers which were objects of DivFinderSP.cpp, each of which spun several threads based on messages passed to them from the main server (Tanenbaum & Steen 2018).

What do we actually mean by the term concurrency? When we talk about concurrency when it relates to computers (servers/clients), we are referring to single systems performing multiple independent activities in parallel rather than sequentially. Historically, most computers had just one processor with a single unit processing core; although some these computers still exist today. Such machines, although may have appeared mildly fast, could really only perform one task at a time. The trick was that it could switch between tasks many times per seconds. This is called task switching. The task switch provides an illusion to both the user and the application itself. Because the illusion of concurrency behavior maybe subtly different when executing in a single process task switching environment compared to when executing in an environment with true concurrency. Computers containing multiple processors (cores) are now mostly being used for servers and high-performance computing tasks. The PCs are capable of running more than one task in parallel (Tanenbaum & Steen 2018).

**Threads in distributed systems**

A useful property of threads in distributed systems is that it can provide a convenient means of allowing blocking system calls with blocking the entire process in which the thread is running. This makes threads commonly used in distributed systems because it is easier to express communication by maintaining multiple logical connections at the same time. Our concurrent servers in our project utilizes this thread concept (Tanenbaum & Steen 2018).

**Inter-process Communication with Sockets**

Sockets are an abstract endpoint of communication between a pair of processes. Developed by Berkley Software Distribution as a part of BSD UNIX, sockets are part of the I/O of an operating system. There are two types of sockets; datagram sockets and stream sockets. TCP/IP sockets are stream sockets while UDP/IP sockets are datagram sockets. The diagram below briefly describes the basic client-server communication.

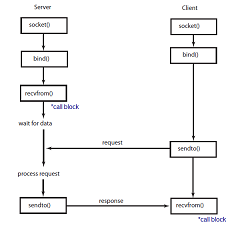


Figure 2: Basic client-server

The structural design of our application (see Design and Methodology) involves a main server assigning prime factorization computation jobs to a set of servers. These servers return values or are reassigned new jobs by the main server until all required computation is done. All of the communication is done through TCP connections.

**Security in Concurrent Systems**

Security has become an important issue in distributed systems with the growth of networking and internet base applications. If one can assume that one’s computer is a trustworthy box containing only software then the security concern must be the data in transit. Most communication today takes place on the public network which is accessible to anyone. How do prevent an eavesdropper from stealing information on the network? How do we prevent secrecy of a sensitive conversation between two agencies over a public network? The concern relates to data security. A different type of security is the prevention from attacks on a computer system by viruses, worms, etc. which intrude systems and compromise software and operating systems. The concerns are related to system security. Our paper will focus on data security. There are six major requirements in security (Tanenbaum & Steen 2018).

* Confidentiality: Secure data must not be accessible to unauthorized persons.
* Integrity: Data consistency should not be compromised. Modifications must be approved.
* Authentication: The identity of a person performing secure transaction must be confirmed beyond doubt.
* Authorization: the user’s action must be consistent with the user permissions.
* Non-repudiation: The originator of a communication must be made accountable.
* Availability: Authorized users have access to data when they need them.

A different way of looking at security in a computer system is by attempting to protect the data and services it offers against security threats.

There are four security threats to consider- interception, interruption, modification, fabrication.

A security policy highlights what actions entities are allowed or not allowed to take. Once the security policy has been set, we need to shift focus to security mechanism (Tanenbaum & Steen 2018).

Important security mechanisms are:

* Encryption
* Authentication
* Authorization
* Auditing

For this project we chose not to implement encryption or authorization into our application/system simply because it would increase (substantially) the runtime – since we are designing a high-performance computing system. Our system however implements authentication. Our main server authenticates all incoming connection by checking its whitelist. If a system trying to connect does not have its IP address listed on the main server’s whitelist, it is will be denied connection.

**Design and Methodology**

**System Architecture**

The system organization is based off a client/server model. A main server (pfserver) manages all the system level logic and coordinates computing tasks to the clients (pfclient). The system was designed to be scalable. It is very easy to add additional clients to system by specifying to the server the number of clients that will be connecting. This is done with the ‘-n’ parameter, and the server will wait for the requested number of clients to join before beginning its computation. The clients are computing resource nodes that find a prime factor of a given number. The clients implement a multithreaded approach which allows the Pollard Rho algorithm to execute on a background thread while the main thread monitors the connection with the main server for commands.

**Communications**

Communication between the main server and nodes is done through TCP Connection Based sockets which implements a layered protocol model. The TCP protocol is a connection-oriented communication method which guarantees the data sent arrives in order. This is important to the methodology used to obtain all the prime factors of a given number, as explained in methodology section. The established system socket connections form message-oriented communications through a request-reply pattern.

**Methodology**

The system level algorithm which finds all the prime factors of a given number was based off the “Performance Analysis of Parallel Pollard’s Rho Factoring Algorithm” (A.K Koundinya et al, 2013) paper. At a high level, each node (pfclient) is an instance of the Pollard’s Rho algorithm. The server provides the same number (N) to each node but each node randomizes their selection of the “c” value in the function . Due to the different “c” values between the nodes, each node will take a different amount of time to execute the full Pollard’s Rho algorithm. The first node to finish sends the prime factor (primeFactor) to the main server. The main server then sends a stop command to all the nodes in order to halt their process. N is then recomputed as N = N / primeFactor and sent back out to the nodes restarting the cycle. The system level algorithm end when all prime factors have been found.

**Results**

To initially test our system, we started with increasing the number of digits while using two nodes/servers for computation (see Table 1 below). The time of completion was measured. We noticed that although our completion was inconsistent in some cases, there was a general uptick as we increased the size of our input number for prime factorization.



Table 1: Performance test by increasing number of digits

Another evaluation was performed to see how the number of nodes in the system affected the execution time for the prime factorization of different sized numbers. These tests were performed on an AMD Ryzen 5 3600 6-Core desktop machine. The testing consisted of 100, 110, 115 and 120-bit semi-prime numbers. For each number we adjusted the number of nodes used in the system from 1 to 3 and timed how long it took to compute the factorials. The results are shown below.

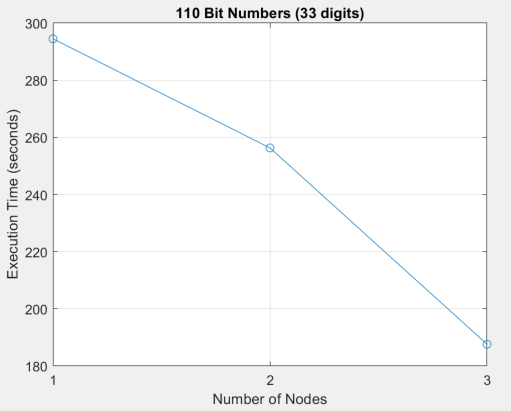
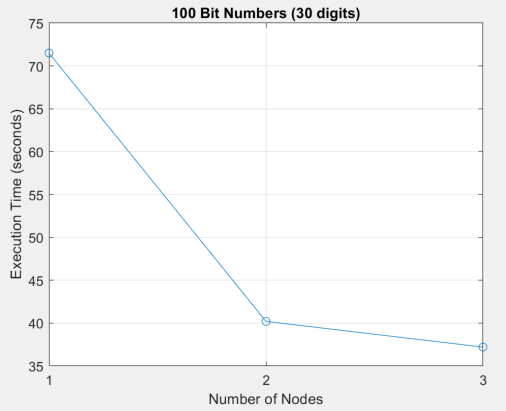


Figure 3: 100 Bit Number Execution Times Figure 4: 110 Bit Number Execution Times

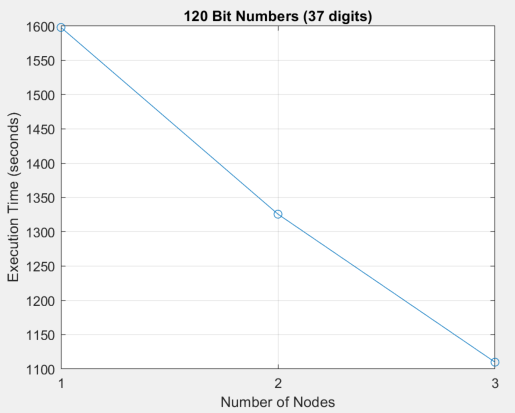
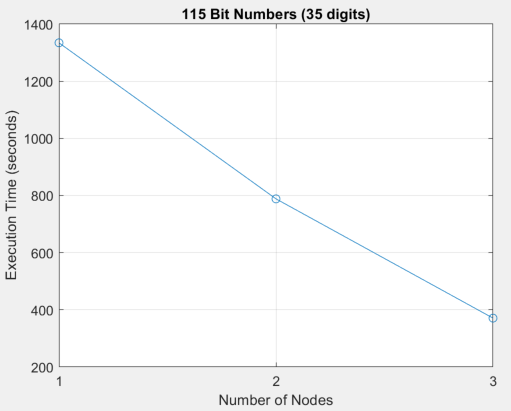


Figure 5: 115 Bit Number Execution Times Figure 6: 120 Bit Number Execution Times

By incorporating additional nodes into the system, we were able to achieve a faster execution time in these examples. Testing was also performed on numbers less than 100 bits, but no speed up was observed. It seems in those situations where the numbers were small enough, a single threaded application was able to compute the prime factorials faster. This is due to the overhead associated with our system, in terms of network communications and the time associated with spawning new threads. An attempt was also made to compute a 128-bit number. We were able to compute the prime factorials after 6,387 seconds, approximately 107 minutes, using our algorithm and 4 nodes. Attempts were not made with less nodes due to the anticipated time needed to perform the calculations.

The last test performed was checking how the number of Pollard’s Rho iteration affected the execution of the system. The number of iterations was declared in DivFinderServer.h, which can be changed by the developer. The system showed an increase of execution time when dealing with large prime numbers (larger than 8 digits) as shown in Figure 7. This is due to the Pollard’s Rho algorithm running until a collision occurs. Surprisingly, the execution time stay the same when dealing with small prime number (no larger than 3 digits) as seen in Figure 8. The number of iterations has no effect on the total run time as the Pollard’s Rho algorithm would quickly get to a collision.

A close up of a map

Description automatically generated

Figure 7: System Execution time when N is comprised of large primes

A screenshot of a cell phone

Description automatically generated

Figure 8: System Execution time when N is comprised of small primes

**Conclusion**

In conclusion, the system achieves the desired result faster than a single process procedure, although the code base could potentially be improved in the depending on the application in two ways. First, the system could dynamically change the algorithm procedure and what type of computations are done with the node depending on the preference of the user. For example, currently the algorithm always runs the Pollard’s Rho algorithm then checks the output of the function to see if it’s a prime. Unfortunately, but expected, if given a large enough (over 8 digits) prime number, Pollard’s Rho algorithm will continue for a relatively long time until a collision occurs between the “tortoise” and “hare” functions. Instead of always going straight to Pollard’s Rho algorithm, if the size of the number to be factor is over a designated threshold (over 8 digits) then it can first be checked by a probabilistic primality test like Miller-Rabin to check if it is a prime, which runs relatively fast. The drawback is that the test result will not 100% accurate given the probabilistic nature, so depends on the application if this approach would make sense. For this final project, we decided to balance speed and accuracy using a single process procedure as a threshold.

Secondly, given the limited timeline of the project, the team was not able to perform conclusive testing on alternate methods of performing computations in order the find the most efficient. For example, given that an application’s highest concern prime factor accuracy, node could have been designated to run a program like homework 3, which would build a table of prime number. The system could then just make a call to the node to check whether a number is prime. This would take O(1) to lookup assuming the node had the table built up to the specified number. This approach would again depend on the application as storage space require for the system would depend on the size of the number being worked with.

In closing, this project enabled the team to implement a distributed system with the knowledge gathered through the course. This allowed the team to cement literature concepts through practical application. The streamlined design approach taken by the team leveraged lessons learned previous assignment and minimized issued cumulating into a functional distributed system.

**Appendix A**

**Compile instructions**

Our application utilizes the boost library and was tested on Linux environment. The compilation and install instruction are as follow:

* Configure Linux environment as done in previous homework (‘autoreconf -i’ followed by ‘configure’)
* Once the environment is configured, run ‘make’ in the DRS2.0 folder
* In the project ‘src’ folder, start the server (see picture below) in the format ./pfserver -f <number to factorize> -n <number of nodes>



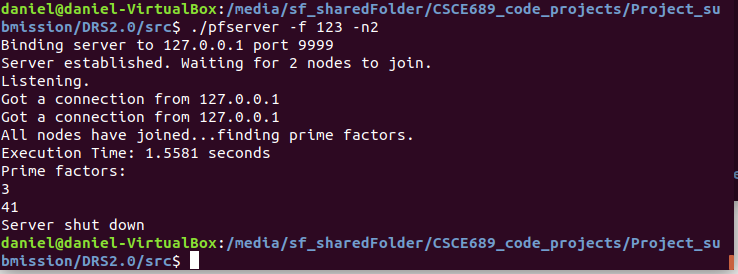
* While in the project ‘src' folder start (see picture) each node (one terminal per node) in the format ./pfclient <ip address> <node>



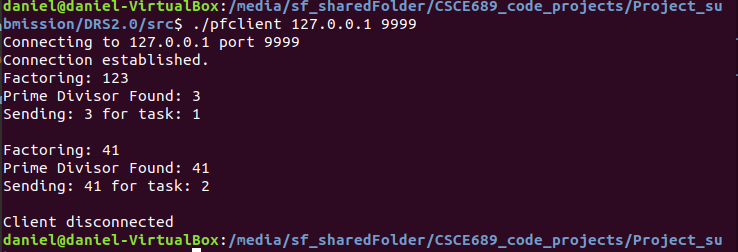
* Done!

***Screen shot of display***

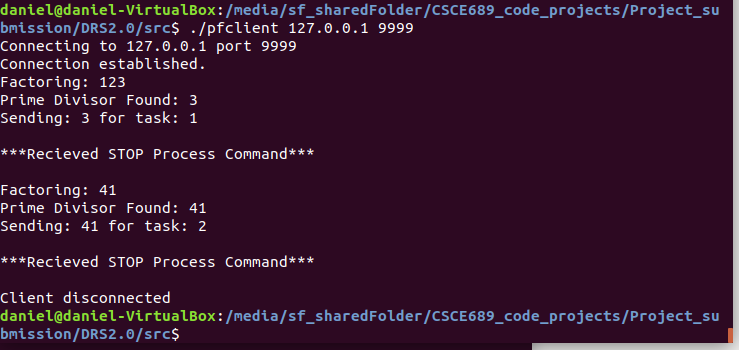
Server



Node 1



Node 2



**References**

Anthony, W. (2012). *C++ Concurrency in Action: Practical Multithreading*. Manning Publications

Sukumar, G. (2014).   *Distributed Systems, 2nd Edition.* Chapman and Hall/CRC publications

Steen, M. V., & Tanenbaum, A, S. (2018). *Distributed Systems, 3rd Edition.* Maarten Van Steen

Templeton, G. (2015, December 18). *What are prime numbers, and why are they so vital to modern life?*

https://www.extremetech.com/extreme/219570-what-are-prime-numbers-and-why-are-they-so-vital-to-modern-life

GeeksforGeeks. (2016, March 14). *Pollard’s Rho Algorithm for Prime Factorization.*

https://www.geeksforgeeks.org/pollards-rho-algorithm-prime-factorization/

Edwards, J. (2019, October 4). *High performance computing: Do you need it?*

https://www.networkworld.com/article/3444399/high-performance-computing-do-you-need-it.html