Generation of Titanic Prime Numbers through High Performance Computing Infrastructure

Mentor: Mr. Je'aime Powell (*ECSU*)
Elizabeth City State University
Elizabeth City, NC 27909

Authors:
John Bell (*MVSU*), Matravia Seymore (*ECSU*), Joseph Jackson (*MVSU*)

Abstract - The focus of the project was to generate a titanic prime number by using high performance computing resources. What makes prime numbers significant is their use in modern computers for the encryption of data. The generation of primes are particularly computing intensive the larger the prime. This makes titanic primes (thousand digit prime numbers) a perfect candidate for distribution through grid infrastructure. In order for the demands of the project to be met, a prime number generator had to be created. Multiple computer languages such as Javascript, Java, and C++ were tested for use in an attempt to develop a generator. Functionality was verified first through the terminal and then by job submission to Elizabeth City State University's VikeGrid (a Condor based computer cluster). Overall, the results indicated the successes of the project and the improvements needed for continued work.

Keywords **- titanic prime, Mersenne prime, cluster, VikeGrid, condor, Sieve of Eratosthenes, terminal**

I. Introduction

Samuel Yates first discovered titanic primes in the 1980s. Yates defined a titanic prime to be a prime number with at least 1000 decimal digits. Mersenne prime was named after Marin Mersenne, a French monk who began the study of these numbers in the early 1600's [1]. The largest known prime is over 2 million digits. Mersenne Primes was applied to this research project because they were used to find complex form of prime numbers.

The process of seeking and discovering new prime numbers became very complicated as the numbers grew larger. Unique patterns found in smaller primes were harder to detect as well. This complication led to the development of numerous theories, algorithms, and formulas proposed that could verify the primality of a number. In order to discover an unknown prime number, some used visualization techniques. There are many visualization resources available such as Benford's Law, Prime Number Theorem, Euclid of Alexandria, Pierre de Fermat, and Eratosthenes. Currently, there is no official formula for all prime numbers.

Prime number generators are critical because they find primes in an efficient manner through the use of computing resources. Due to the wide variety of programming languages available, each generator had an unique source code. A majority of them are based on the same common theorems and algorithms. As mentioned earlier, there is no definite formula for finding all prime numbers. As a result, all generators have limitations. Limitations include the number of primes that can be found as well as the size of primes found.

Primes that are found by generators are able to contribute to the encryption of the data. Encryption refers to algorithmic

schemes that encode plain text into non-readable form providing privacy. The RSA (Rivest, Shamir, and Adleman) system is the most popular cryptography method. The security of RSA primarily relies on the difficulty of factoring large composite numbers. Two very large prime numbers are found and multiplied to create a composite number where the primes are the only factors. Data is secured because hackers have to defeat the problem of factoring the resulting composite number — given two sufficiently large prime numbers. This task is believed to be extremely difficult to factor in a finite amount of time.

A. This project helped answer the following questions

1. Was a titanic prime number generated?

2. How to create a prime number generator?

3. How to run a java and jar file from terminal?

4.Was the job able to be submitted to the VikeGrid?

II. Literature Review

Mersenne primes are named after Marin Mersenne, a French monk who began the study of these numbers in the early 1600's. A Mersenne number (M) is a positive integer (P) that is one less than a power of two, $M_P = 2^P - 1$ [1] .The largest known prime is the Mersenne prime $2^{6,972,593} - 1$, a number with over 2 million digits. This is only the 38th in the sequence of Mersenne primes.

It is known that the fastest way to discover a Mersenne (Titanic) Prime is to use the Sieve of Eratosthenes method. First is the creation of a list of consecutive integers from two to *n* (2, 3, 4... *n*). Then *p* is set equal to 2, which is the first prime number. Canceling from the list all the multiples of *p* less than or equal to *n* (*2p, 3p, 4p, etc.*). Find the first number remaining on the list after *p*. That number will be the next prime and will replace *p*. The steps 3 and 4 are then repeated until 2*p* is greater than *n*. All the remaining numbers in the list after cancelling are considered Mersenne primes [2].

The Great Internet Mersenne Prime Search (GIMPS) was developed in 1996 as a cooperative venture in which people volunteered unused time on their personal computers to search for prime numbers. A central server on the Internet coordinated the efforts of those participating, and recorded the results. The Electronic Frontier Foundation (EFF) offered a $100,000 prize to the first person or group to find a prime number with at least ten million digits. The current record-holding Mersenne prime garnered a $50,000 prize from the EFF [1].

On April 12th 2009, the 47th known Mersenne prime, 242,643,801-1, a 12,837,064 digit number was found by Odd Magnar Strindmo from Melhus, Norway. It was the second largest known prime number [2]. Odd was an IT professional whose computers worked with GIMPS since 1996 during which time over 1400 candidates were tested. The calculation took 29 days on a 3.0 GHz Intel Core2 processor. The prime was first verified on June 12th of the same year by Tony Reix of Bull Serial Attached SCSI (SAS) in Grenoble, France used the Glucas program that ran on Bull NovaScale HPC servers. One featured Itanium2 CPUs and another featured Nehalem CPUs. SCSI is a

Small Computer System Interface and Serial Attached Storage(SAS) itself is a data transferring technology that moves data from various computer devices such as hard drives. The Glucas program is a free program to test the primality of Mersenne numbers. The prime was later independently verified by Rob Giltrap of Sun Microsystems using Ernst Mayer's Mlucas program running on a Sun SPARC Enterprise M9000 Server. The Mlucas program was similar to the Glucas program except it used a FORTRAN compiler [2].

Mersenne Primes are applied to this project because they are used to find complex forms of prime numbers. It is possible for them to be found through the use of a prime number generator [2]. Generators have the ability to perform the task of finding prime numbers more efficiently and effectively than humans. This is possible when computing resources are available and multiple algorithms are used for verification.

Kerry Soileau's generator (Figure 1) proved to be useful for this project [3]. It was created in javascript, a web-based programming language. An appealing feature of this program was that the number of primes could be chosen. However, the most appealing feature was that the program eliminated the idea of starting with an undesired number. The program gave the option of choosing a lower bound.

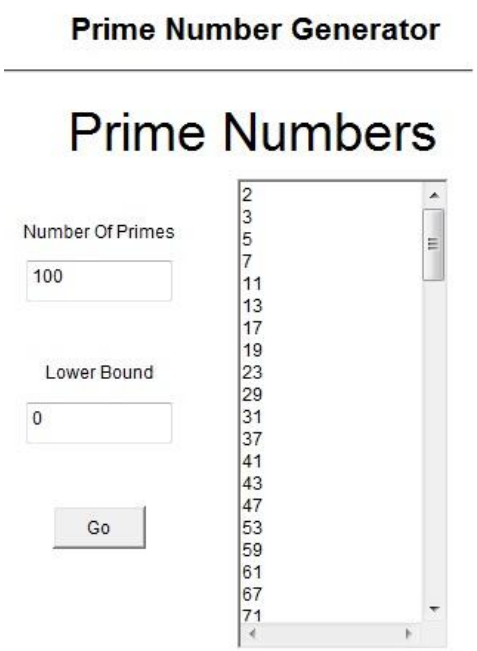

Figure 1

The aspects from these articles that were used and analyzed for this project were the basic knowledge of the use of prime numbers, different methods and formulas for finding primes, and the prime number generator. Also, the use of a lower bound in the generator was one of the ideas implemented in the program's design.

III. Methodology

Multiple programming languages were used for the project. The first language used was javascript. The source code from Kerry Soileau's generator was downloaded as a .class files. The class files were then saved as Java files so that they could be opened in an integrated development environment (IDE). It was then placed in the NetBeans IDE to be compiled and tested to see if it was usable for the long-term program that was to be created. After much experimenting, it did not work because javascript is a web-based language. This meant it could not be re-compiled into a

standalone program which is needed to use with Condor.

```
1  package primenumbergenerator;
2  import static java.lang.Math.pow;
3  import java.text.DecimalFormat;
4  import java.text.NumberFormat;
5
6  class Main
7  {
8    public static void main(String args[])
9    {
10         int a=13;
11         int b=31;
12         double lowerbound=pow(2.0,a)-1;
13         double upperbound=pow(2.0,b)-1;
14         double num = lowerbound;
15
16         NumberFormat formatter = new DecimalFormat();
17
18         while(num <= upperbound)
19         {
20      if (num % 2 !=0 && num % 3 !=0 && num % 5 !=0 && num % 7 != 0
21         && num % 11 != 0)
22         //Does not print 2,3,5,7,or 11
23          {
24          formatter = new DecimalFormat("0");
25         System.out.println(formatter.format(num));
26          num++;
27          }
28   else
29       num++;
30         }
31   }
32 }
33
```

Figure 2

The second programming language used was Java. A prime number generator based on probability testing was constructed in the language. Probability tests state that if a number is likely prime or not. Sometimes composite numbers are mistakenly reported as prime numbers. These numbers are called pseudoprimes. However, pseudoprimes have rarely occurred in good probability tests [4]. In Figure 2, primality testing began on line 18 with a “while loop”, a command performed if the statement within parentheses is true. Therefore, the loop was executed as long as the number was less than or equal to the upper bound. An “if/else” statement was inserted in the loop in order for the process to be continued. The command used the modulus operator (%) and the AND operator (&&) multiple times in the first section of the statement. Once the number had been divided by the first five prime numbers, the modulus operator retrieves the remainder. If the remainder was not zero for all five cases then the number was formatted in decimal notation to show all digits and printed out as a prime number. Otherwise, the number was incremented by one and the new value would be tested until the value was greater than the upper bound.

After the source code was written, it was then tested to see if it could be compiled and executed in terminal.

A. Terminal/Command Line Steps for Java File

1. Opened terminal in Mac OS X by clicking the application icon on the sidebar

2. Changed the directory to the desktop with the following command: *cd*\Desktop

3. Created a directory to hold the java file with the following command: *mkdir* Generator

4. Opened the terminal text editor and named the file the following command: *nano* PrimeNumberGenerator.java

5. Copied and pasted the source code from NetBeans into the text editor and saved it.

6. Compiled the PrimeNumberGenerator with the following command: *javac* PrimeNumberGenerator.java

7. Attempted to execute the file with the following command: *java* PrimeNumberGenerator

The failed attempt at execution suggested that another route be taken. The NetBeans IDE was opened and the source code was recompiled into a jar file, an alternative java format with multiple java and class files compiled. Upon completion, the IDE provided the single line command for opening the jar file in terminal. The file executed successfully in terminal after the jar command was entered. This led to preparation for file submission into condor.

For Condor to run the program, a submission file was created. The submission file was text file containing the values of certain parameters, such as the name of the program, the universe, location, and files to be created that held the log, output, and error information.

```
Universe   = java
Executable = PrimeNumberGenerator.jar
Arguments  =
Log        = Generator.log
Output     = Generator.out
Error      = Generator.error
Queue 1
```

Table 1

Table 1 shows the information typed inside the "condorsub" submission file. According the submission script, three files were created and the generator was to execute only one time. The command *condor_submit condorsub* was entered in for the submission of the prime number generator. Unfortunately, it failed because the parameters for arguments could not be found.

```
#include <iostream>
#include <cmath>
#include <iomanip>
#include <fstream>
using namespace std;

int main()
{
	double aye = 13;
	double bee = 61;
	double lowerbound = pow(2.0, aye)-1;
	double upperbound = pow(2.0, bee)-1;
	double num = lowerbound;
	ofstream prime;
	double number;

	prime.open("primenumberlist");

	while (num <= upperbound)
	{
		cout << fixed << setprecision(0);

		if (fmod(num,2) != 0 && fmod(num,3) != 0 &&
			fmod(num,5) != 0 && fmod(num,7) != 0 &&
			fmod(num,11) != 0)
		{
			cout << num << endl;
			number = num;
			prime << fixed << setprecision(0);
			prime << number << endl;
			num++;
		}
		else
		num++;
	}
	prime.close();
	return 0;
}
```

Figure 3

A C++ project was then opened in NetBeans so that the source code from the original generator could be converted. In Figure **3**, the source code did not differ much from the original. The same actions were performed except for one addition to the program. The generator created a text file titled primenumberlist containing the list of numbers found. There two ways of trying to compile the file that was used. First, the command *condor_compile* was used in terminal. Second, the C++ source code was compiled through the terminal and automatically opened

which eliminated the need for manual execution.

```
Universe   = vanilla
Executable = PrimeNumberGenerator
Arguments  =
Log        = Generator.log
Output     = Generator.out
Error      = Generator.error
Queue 1
```

Table 2

Table 2 is the information contained in the submission script for the generator written in C++. The universes changed from java to vanilla which supports execution of standalone programs. Also the file extension .jar was removed because it was no longer a jar file. Once the file was submitted, the command *condor_q* was used to check the status of the program. Two more prime number generator would be submitted in condor. However, all of them would have different lower and upper bounds. The first generator would use 107 as the lower bound and 127 for the upper bound. The second generator used 607 as the lower bound and 1279 for the upper bound. *Condor_q* was used throughout the execution of all three generators to check their statuses.

IV. Results

A. *Was a titanic prime number generated?*

A titanic prime number was not generated. The lack of memory and being unable to go over the data type limit were the primary reasons for this task failing.

B. *How to create a prime number generator?*

- It was learned how to create a prime number generator. Below are results for the generators:
- The only generator to execute and finish completely was the original generator.
- The first two generators were reported as running when the condor_q command was used in the terminal. However, the second generator was reported as idle. This was not unexpected because it was known that the generator was bugged to produce some kind of error. The error could be found with the bounds that were chosen. The double data type was only able to store a number that contained up to 308 digits. Obviously, the value of the number exceeded 308 digits before it was even equal to the upper bound.
- As mentioned earlier, the first two generators were reported as running by condor. However, the generator with the lower bound of 107 and the upper bound 127 did not have enough memory to finish executing the program.

C. *How to run a java and jar file from terminal?*

The ability to how to run a jar file through terminal was acquired. It uses the command *java –jar "filedirectory/file.extension".* The command Unfortunately, it was still unknown why the java file did not execute in terminal even thought it could be compiled in NetBeans.

D. *Was the job able to be submitted to the VikeGrid?*

The original generator in C++ could be submitted along with the two tests.. It could be compiled in the IDE and terminal. Also, it executed successfully in both as well. The final step was also completed when the generator could be submitted to the VikeGrid cluster.

E. *Additional Results*

It was found that condor_compile could not be used in the command line. The following error message was given when the command was entered: "This *version of Condor does not support checkpointing and remote system calls on this platform. You may only submit "vanilla" jobs. Therefore, you do not need to use condor_compile. In fact, condor_compile can't work, since there are no libraries to re-link your job with. Please see the Condor Manual for details, which can b efound at http://www.cs.wisc.edu/condor/manual"*

V. Future Work

A titanic prime was not found in this project. In order to find a titanic prime, there were several things that should be considered:

1. The prime number generator was able to execute successfully in terminal as a jar file but it would not run on condor. The correct argument parameters should be found if jar files are to be used in future projects involving condor.

2. The double data type was used numerous times in the source code of the generator. It was the double data type that could hold the most digits. However, the double data type has a limit of 308 digits. To find a titanic prime, there must be a data type to exceed this limit or an alternative solution to hold more than a thousand digits.

3. Although the generators were able to be uploaded and executed on condor, there was not enough memory for the program to finish. It recommended that computers with above-average memory be used. A second method would be to find a way to erase numbers out of memory after they are stored. A final method would be to utilize parallel computing over distribution (condor). Parallel computing allows for a task to be evenly divided among a cluster.

4. Condor_compile is a command in condor that would have been useful to "re-link [the] program with the condor libraries for submissions into Condor Standard Universe"[5]. It offered a feature called checkpointing, which allowed for the status of job to be displayed and restarted at a certain point if the job was accidently terminated. Full installation of condor_compile is needed for better job management.

References

[1] "Mersenne Prime," *MathemicalVignettes*, [Online]. Available: http://www.jcu.edu/math/vignettes/mersenne.htm. [Accessed Jun. 23, 2008].

[2] "47th Known Mersenne Prime Found!" *GIMPS*, para 1., Oct. 29, 2009. [Online]. Available: http://www.mersenne.org/. [Accessed: Jun. 24,2008].

[3] K. Soileau, "Prime Number Generator," *Java(TM) Boutique - Prime Number Generator*, [Online]. Available: http://javaboutique.internet.com/prime_numb/2010. [Accessed: Jun. 8, 2010].

[4] E. Weisstein, "Primality Test," *MathWorld—A Wolfram Web Resource*, para. 3, Apr. 30, 2002. [Online]. Available: http://mathworld.wolfram.com/PrimalityTest.html. [Accessed: Jul. 3, 2010].

[5] "condor_compile," [Online]. Available: http://www.cs.wisc.edu/condor/manual/v7.1/condor_compile.html.[Accessed: Jul. 5, 2010].