A Parallel Computing Application:

The Prime Numbers & The Sieve of Eratosthenes

José Ramón Romero Chávez (A01700318)
Tecnológico de Monterrey, Campus Querétaro
A01700318@itesm.mx

20th November 2018

Abstract

The Prime Numbers plays a key role in several routines in information technology, such as public-key cryptography, which relies on the difficulty of factoring large numbers into their prime factors.

Keeping in mind these important applications, is necessary that every person related to these fields to know at least one algorithm to get a set of them.

The sieve of Eratosthenes is a ancient algorithm for finding all prime numbers up to any given limit, it relies on the on the idea of constant difference between an integer and its multiples.

Along to get know this algorithm, the main purpose of this project is to expose the advantages of the parallel computing compared versus a sequential approach and all the applications and implications that it has.

Introduction

About Parallel Computing

"The number of transistors incorporated in a chip will approximately double every 24 months."—Gordon Moore, Intel co-founder

Is probably that back in the 1960' when Gordon Moore said this by first time, he could never imagine all the artifact that have today and with them also the finish of his prediction due to the physics implications that the resize of the elements inside of a chip had, especially talking about the amount heat produced and the energy used.

As it is described by Herb Sutter (2005), another idea will come out to revolution the computing industry, “the multicore systems” and along with them software tool that will aim to get the best of them by the use of multi thread tools.

As it is described by Marowka (2007), for many years, only small sectors of society were the only ones capable of exploiting the capabilities of multiprocessor systems, due to their complexity and cost, it was not until 2005 when AMD and Intel introduced the with the dual-core architecture processors and caused a dramatic drop in the price of desktop computers and laptops, as well as of multicore processors.

Dual-core processors were only the beginning of the multicore-processor era.

The primary consequence is now that applications will increasingly need to be parallelized to fully exploit processor throughput gains now becoming available.

Unfortunately, writing parallel code is more complex than writing serial code. This is where the apis such as OpenMP, Intel TBB, and others enter the parallel computing picture.

All of them helps developers to create multithreaded applications more easily while retaining the look and feel of serial programming.

Fortunately, this has not remained only in desktop computers, finally the jump has been made to mobile devices of similar dimensions.

All this has been a beautiful process of continuous improvement of both software and hardware, and the search for better alternatives has not stopped yet.

Now the bet relies on what has been called as "Hardware As A Service", offered by many companies such as Microsoft, Google and Amazon.

Herb Sutter in "Welcome to the Jungle" (2008) explains the opportunity offered by these services to access massive clusters of specialized supercomputers to our needs without having to invest the real cost of each of them from the comfort of our homes.

Truly this moment of history gives us the opportunity to practically make any software without serious limitations of time, difficulty and costs.

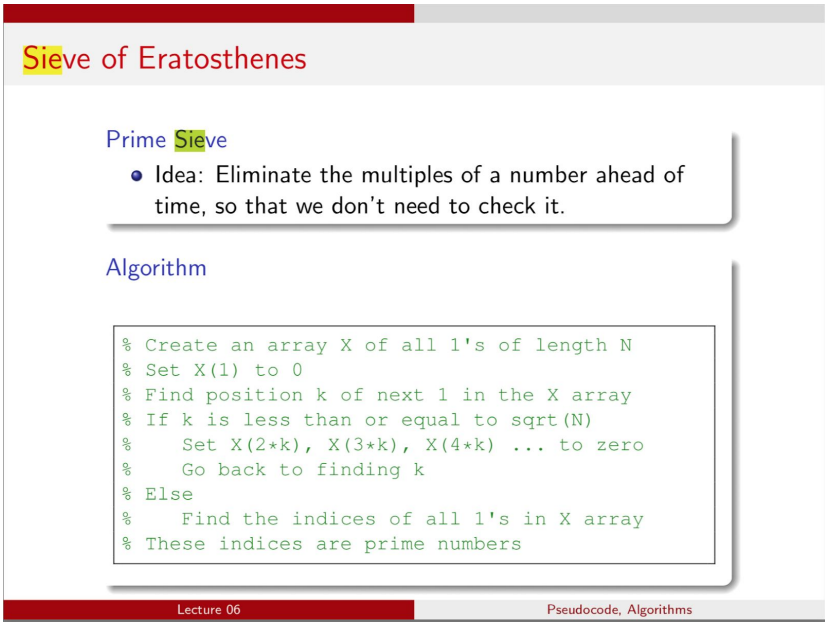
About the It is always necessary to consider that not all algorithms and programs are able to parallelize, for this it is necessary that the programmer is aware of the relationship that exists between the logical threads and their abstraction related to the cores and processors, their communication and synchronization, especially consider dependencies, race conditions, memory inconsistencies, deadlocks and other conflicts, in order to evaluate if the parallelization is an optimization option

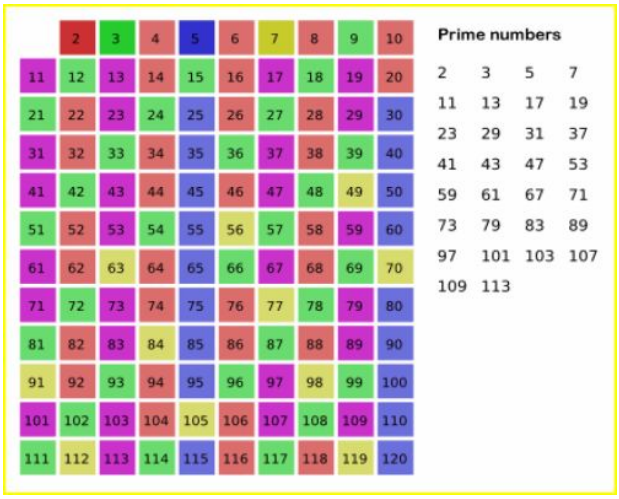
About the The Prime Numbers & The Sieve of Eratosthenes.

As it is described by Sierpiński (1988), any number bigger than 1 ,which has no natural (integer) divisors except itself and 1 is called a prime number. They are fundamental in number theory because of the fundamental theorem of arithmetic: every natural number greater than 1 is either a prime itself or can be factorized as a product of primes that is unique up to their order. Thanks to this theorem, is possible to use the in multiple algorithms in computer science such as public-key cryptography, which relies on the difficulty of factoring large numbers into their prime factors.

The sieve of Eratosthenes is an ancient algorithm for finding all prime numbers up to any given limit. It relies on the on the idea of constant difference between an integer and its multiples.

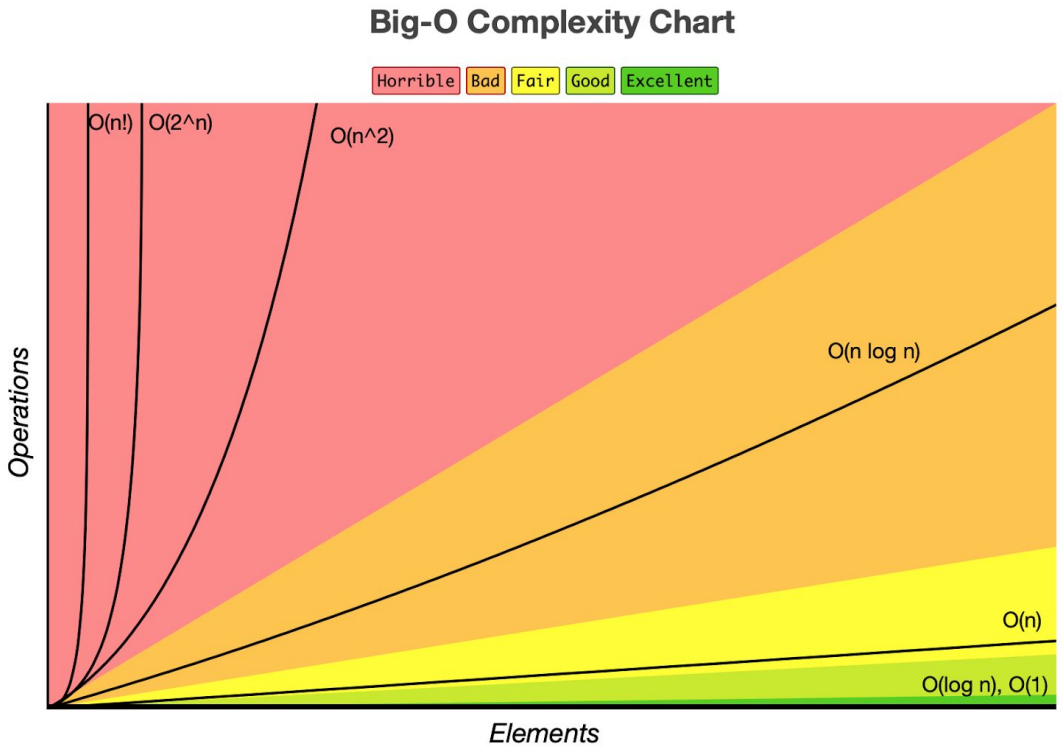
This is the algorithm/pseudocode of the sieve of Eratosthenes described by Erdal Yılmaz from Cornell University (2013) and a graphic representation of the same.





This is the sieve implies a great improve in the time complexity of the algorithm compared with a trial division method AKA “BruteForce”

Reminding that the Sieve has a time complexity of O(N*log(log(N))) and a trial division method has a time complexity of O(N*M) both of them with an space complexity of O(N), this implies a significant less time consumption.



As has been said previously, one of the main uses of prime numbers is in the current world is the asymmetric cryptography, a cryptographic system that uses pairs of keys:

- Public keys: which may be disseminated widely
- Private keys: which are known only to the owner.

According to Zaldivar (2012) this application has its origins back to 1976, when Whitfield Diffie and Martin Hellman published the asymmetric key cryptosystem.

This method of key exchange, which uses exponentiation in a finite field, came to be known as Diffie–Hellman key exchange. This was the first published practical method for establishing a shared secret-key over an authenticated (but not confidential) communications channel without using a prior shared secret.

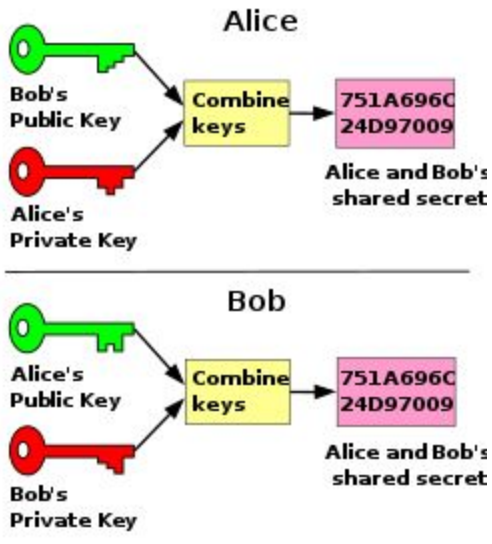
In 1977, an scheme was invented by Ron Rivest, Adi Shamir and Leonard Adleman. The latter authors published their work in 1978, and the algorithm came to be known as RSA, from their initials.

RSA uses exponentiation modulo a product of two very large primes, to encrypt and decrypt, performing both public key encryption and public key digital signature.

Its security is connected to the extreme difficulty of factoring large integers, a problem for which there is no known efficient general technique.

This accomplishes two functions: authentication, where the public key verifies that a holder of the paired private key sent the message, and encryption, where only the paired private key holder can decrypt the message encrypted with the public key

A simple application using The Sieve of Eratosthenes and the RSA algorithm will be described by the end of the note.



Methods and Development

This project has been splitted in two parts, the first of them is focused on the Sieve of Eratosthenes algorithm, its sequential programming versus different approaches of parallel computing technologies such as:

- C: OpenMP
- Java: Threads
- C++: Threading Building Blocks (TBB)
- Java: Fork Join Framework
- NVIDIA: CUDA: GPU Computing

The second one is a short implementation of the the RSA algorithm of public and private key generation, encryption and decription using The Sieve of Eratosthenes.

Main Hardware

As it was described on the introduction, the performance depends on the hardware where the program is running, the following implementation has been tested on:

- MacBook Pro (Retina, 13-inch, Early 2015)
    - macOS Mojave 10.14 (18A391)
    - Processor Name: Intel Core i5
    - Processor Speed: 2.7 GHz
    - Number of Processors: 1
    - Total Number of Cores: 2 with hyperthreading
    - L2 Cache (per Core): 256 KB
    - L3 Cache: 3 MB
    - Memory: 8 GB 1867 MHz DDR3
    - clang --version
        - Apple LLVM version 10.0.0 (clang-1000.11.45.2)
        - Target: x86_64-apple-darwin18.0.0
        - Thread model: posix
    - javac -version: javac 9
    - java -version:
        - java version "9"
        - Java(TM) SE Runtime Environment (build 9+181)
        - Java HotSpot(TM) 64-Bit Server VM (build 9+181, mixed mode)

The trial division AKA "brute force" approach

A few months ago I was doing some coding challenges online, some of then asked me to validate prime numbers in an range, back to that day the only way that I can get it done was to iterate over the range and look for those numbers that can only be divided by 1 and itselfs.

This idea seems to work right, but the fact is that iterate over a number and all those below itself that may be its divisor may work fine for an "small" range, but when it comes to a "big" ranges it simply last too much.

The reason of this, relies on its time complexity, which in Big O Notation is O(N*M), because for the worst case scenario (a prime number) is it necessary to iterate for all the integer numbers below it, it can be improved to O(N*Log(N)) if we only iterate until the square root of each element to validate its status as a prime number.

C/C++ (Appendix 1)

BruteForce Not Optimized C 10 times Average	
Limit	Time (ms)
10	0,001700
100	0,007100
1000	0,332000
10000	19,405500
100000	1510,602000
1000000	129228,512200

Java (Appendix 2)

Sequential Sieve Not Optimized Java 10 times Average	
Limit	Time (ms)
10	0,000000
100	0,000000
1000	0,700000
10000	25,800000
100000	1923,900000
1000000	156861,700000

On the other hand a simple improvement on the limit for the inner for, implies an incredible improvement of O(N*Log(N)), making more test cases posible

C/C++ (Appendix 3)

BruteForce Optimized Java 10 times Average	
Limit	Time (ms)
10	0,001600
100	0,004800
1000	0,063200
10000	1,148500
100000	23,145000
1000000	554,788400
10000000	14047,1854

Java (Appendix 4)

BruteForce Optimized Java 10 times Average	
Limit	Time (ms)
10	0,00
100	0,10
1000	0,20
10000	1,50
100000	22,90
1000000	519,70
10000000	13818,00

After some weeks of reading some programming lectures, I found the algorithm of Sieve of Eratosthenes, from the very first time I read it I was fascinated because of the possibilities, the improvements and the simplicity that it had, so I decided to implement it.

The Sieve of Eratosthenes, a sequential approach

As it was described on the introduction, the algorithm relies on the constant difference between a number and its multiples, that implies an important reduction of the times that we iterate over the numbers, also it includes the other optimization, the use of the square root of the limit to reduce the general number of iterations.

Keeping that in mind the resultant time complexity is now a decent O(N*log(log(N))).

C/C++ (Appendix 5)

Sequential Sieve C 10 times Average	
Limit	Time (ms)
10	0,001400
100	0,003200
1000	0,047100
10000	1,104900
100000	22,764300
1000000	567,267800
10000000	15141,640800

Java (Appendix 6)

Sequential Sieve Java 10 times Average	
Limit	Time (ms)
10	0,000000
100	0,000000
1000	0,100000
10000	1,900000
100000	25,400000
1000000	630,400000
10000000	16927,200000

The point of interest for the following sections relies directly in one section of the algorithm, the search of the numbers which are multiple of the prime, as it if a search and validation over the rest of the array, it is a task that can be parallelized.

The Sieve of Eratosthenes, a parallel approach

OpenMP

Retrieved from its website, The OpenMP is and Application Program Interface (API) that supports multi-platform shared-memory parallel programming in C/C++ and Fortran on all architectures, including Unix platforms and Windows NT platforms. Jointly defined by a group of major computer hardware and software vendors, OpenMP is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer.

It is probably the easiest API to use for parallel programing, its is widely used because it is compatible with C, C++ and Erlang.

For the implementation of The Sieve of Eratosthenes, it was only necessary to specify the proper constraints for our shared variables with the use of the sentence:

```
#pragma omp parallel for shared(a, limit) private(c)
```

That will allow us to start a parallel for capable of search through the array searching for the multiples of the prime numbers

The result were the following:

C: OpenMP (Appendix 7)

Parallel Sieve OpenMP 10 times Average	
Limit	Time (ms)
10	0,938400
100	0,392800
1000	0,413600
10000	1,807700
100000	18,612200
1000000	367,919000
10000000	10775,373300

Java: Threads

Retrieved from the Java Official Documentation, Threads are sometimes called lightweight processes. Both processes and threads provide an execution environment, but creating a new thread requires fewer resources than creating a new process.

Threads exist within a process — every process has at least one. Threads share the process's resources, including memory and open files. This makes for efficient, but potentially problematic, communication.

Multithreaded execution is an essential feature of the Java platform. Every application has at least one thread — or several, if you count "system" threads that do things like memory management and signal handling. But from the application programmer's point of view, you start with just one thread, called the main thread. This thread has the ability to create additional threads, this last point is fundamental for the proper understanding of the implementation of The Sieve of Eratosthenes using threads.

For this implementation is necessary two classes:

- Sieve Threads: Is an extension of the superclass Thread, that will share the sieve and an interval, defined by (Limit/#THREADS), that will allow the thread to iterate over the sieve without causing any memory inconsistency ehi it is looking for the multiple numbers
- Main: Where is defined the limit, and is created and executed a pool of Sieve Threads, and by the very end will show us the complete sieve.

The results of the executions were the following:

Java: Threads: (Appendix 8)

Parallel Sieve Threads Java 10 times Average	
Limit	Time (ms)
10	0,100000
100	0,200000
1000	0,300000
10000	1,500000
100000	19,300000
1000000	558,100000
10000000	14911,900000

C++: Intel® Threading Building Blocks (TBB)

Retrieved from the Intel® Threading Building Blocks website, Intel® TBB is a popular software C++ template library that simplifies the development of software applications running in parallel (key to any multicore computer). Intel® TBB extends C++ for parallelism in an easy to use and efficient manner. It is designed to work with any C++ compiler thus simplifying development of applications for multi-core systems. Intel® TBB is a C++ template library that adds parallel programming for C++ programmers. It uses generic programming to be efficient. Threading Building Blocks includes algorithms, highly concurrent containers, locks and atomic operations, a task scheduler and a scalable memory allocator. These components in Intel® TBB can be used individually or all together to ease C++ development for multi-core. Intel® TBB provides an abstraction for parallelism that avoids the low level programming inherent in the direct use of threading packages such as p-threads or Windows threads. It has programmers express tasks instead of threads. Intel® TBB facilitates scalable performance in a way that works across a variety of machines for today, and readies programs for tomorrow. It detects the number of cores on the hardware platform and makes the necessary adjustments as more cores are added to allow software to adapt. Thus, Intel® TBB more effectively takes advantage of multi-core hardware.

The implementation for the Sieve of Eratosthenes is similar in someway to the one that was done for Java Thread, as was described previously Intel® TBB template library, keeping that in mind we can use them to create our clases and define the actions to be paralyzed. For the Sieve, it was only necessary to define a Parallel Sieve, that will receive the sieve and according to the grain it will keep splitting it, and by the end join the inspected sections.

Results:

C++: Intel® TBB: (Appendix 9)

Parallel Sieve Threads Java 10 times Average	
Limit	Time (ms)
10	0,615400
100	0,193800
1000	0,263800
10000	2,110300
100000	38,966700
1000000	1133,330000
10000000	30988,400000

Java: Fork Join Framework

As it is described by the Java Official Documentation, The fork/join framework is an implementation of the ExecutorService interface that helps you take advantage of multiple processors. It is designed for work that can be broken into smaller pieces recursively. The goal is to use all the available processing power to enhance the performance of your application.

For the implementation of the Sieve of Eratosthenes using this framework, in a similar way that the previously described methods, it is necessary to create a class (ForkSieve) that is an extension from the class RecursiveAction. In this case, the class will recibe the sieve, and according to a limit specified, the sieve will be splitted in smaller pieces to iterate over them looking for the multiples of a number, similar to the previous techniques it is focused on the idea of divide and conquer.

Results:

Java Fork/Join Framework: (Appendix 10):

Limit	Time (ms)
10	1,700000
100	2,700000
1000	5,600000
10000	19,600000
100000	167,400000
1000000	1262,000000
10000000	24197,600000

NVIDIA: C/CUDA: GPU/CLOUD Computing

As final implementation of the Sieve of Eratosthenes, we will take a jump to the cloud, for this implementation is being used and instance of Amazon Elastic Compute Cloud (Amazon EC2) from Amazon Web Services because as Herb Sutter (2008) said, suggest us it allowed us to have incredible computing power without having to spend a lot of resources. *(Hardware Description (Appendix 11))*

Retrieved from the NVIDIA Developer Site, CUDA® is a parallel computing platform and programming model developed by NVIDIA for general computing on graphical processing units (GPUs). With CUDA, developers are able to dramatically speed up computing applications by harnessing the power of GPUs.

In GPU-accelerated applications, the sequential part of the workload runs on the CPU – which is optimized for single-threaded performance – while the compute intensive portion of the application runs on thousands of GPU cores in parallel. When using CUDA, developers program in popular languages such as C, C++, Fortran, Python and MATLAB and express parallelism through extensions in the form of a few basic keywords.

For the parallel implementation the Sieve of Eratosthenes, the main functionality will rely on a single parallel function, that thanks to the architecture of a GPU will have an exceptional performance, this function will recibe the array and depending on number of thread, block and dimension, will be access to the exact location of the array to evaluate it.

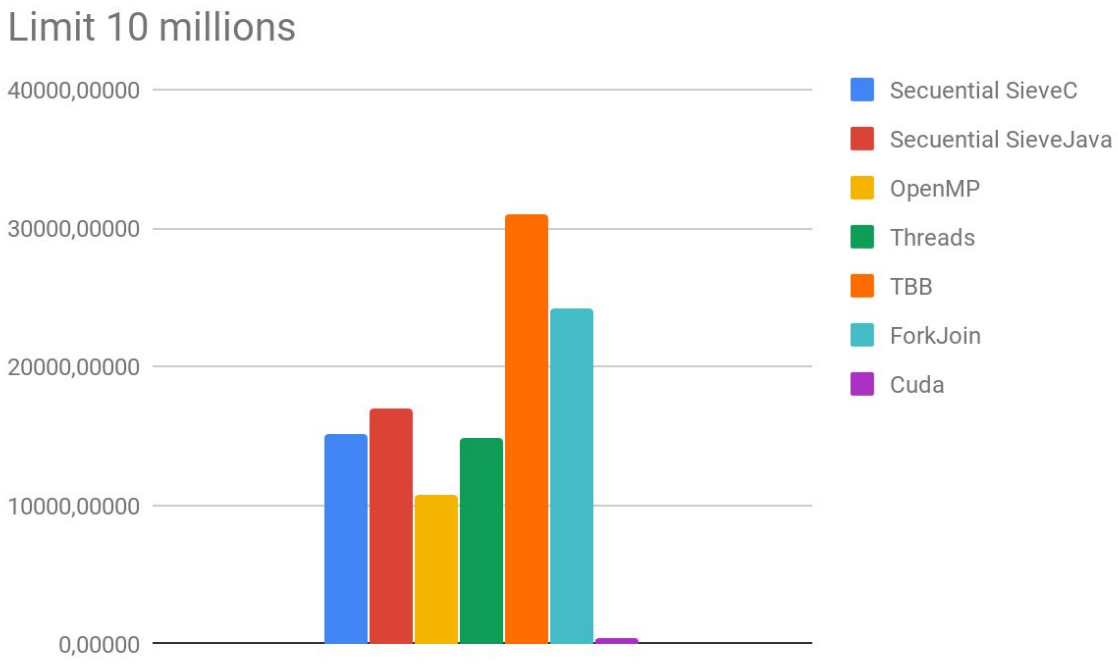
Results:

NVIDIA : C/CUDA: GPU/CLOUD COMPUTING (Appendix 12)

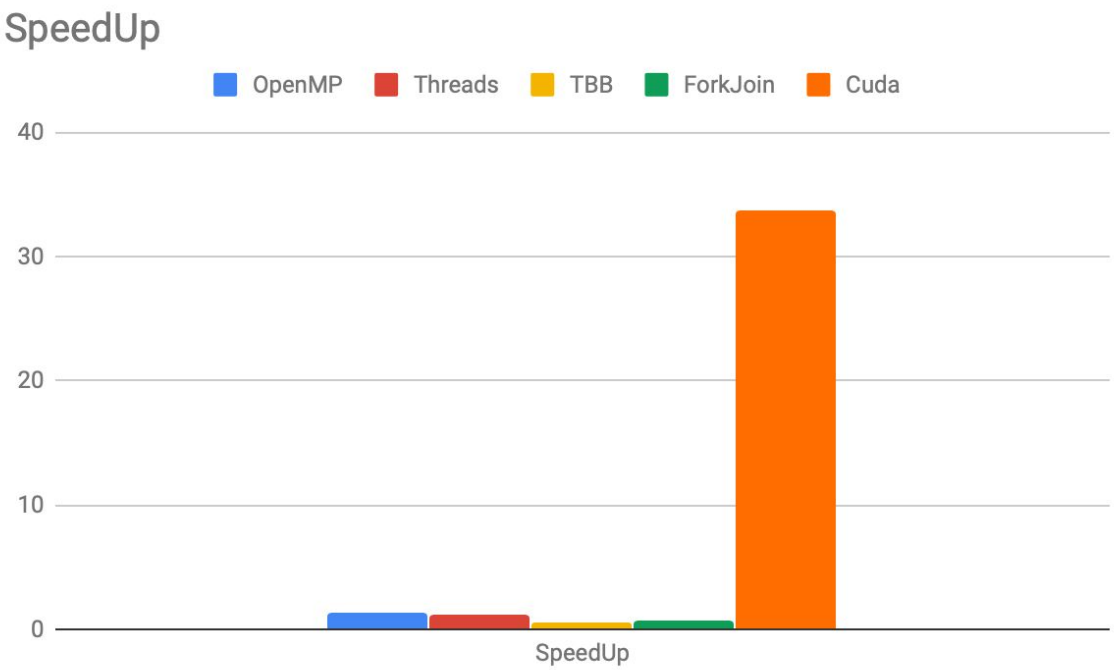
Parallel Sieve Threads Java 10 times Average	
Limit	Time (ms)
10	6,689600
100	5,507600
1000	6,674100
10000	6,756000
100000	7,952100
1000000	30,255500
10000000	448,502800

Benchmarking:

Limit	Sequential Sieve C	Sequential Sieve Java	OpenMP	Threads	TBB	ForkJoin	Cuda
10	0,00140	0,00000	0,93840	0,10000	0,61540	1,70000	6,68960
100	0,00320	0,00000	0,39280	0,20000	0,19380	2,70000	5,50760
1000	0,04710	0,20000	0,41360	0,30000	0,26380	5,60000	6,67410
10000	1,10490	1,60000	1,80770	1,50000	2,11030	19,60000	6,75600
100000	22,76430	26,70000	18,61220	19,30000	38,96670	167,40000	7,95210
1000000	567,26780	650,60000	367,91900	558,10000	1133,33000	1262,00000	30,25550
10000000	15141,64080	16927,20000	10775,37330	14911,90000	30988,40000	24197,60000	448,50280
Limit	Sequential Sieve C	Sequential Sieve Java	OpenMP	Threads	TBB	ForkJoin	Cuda
10000000	15141,64080	16927,20000	10775,37330	14911,90000	30988,40000	24197,60000	448,50280



	OpenMP	Threads	TBB	ForkJoin	Cuda
SpeedUp	1,405208003	1,135147097	0,4886228653	0,6995404503	33,76041532



Results and Conclusions

Parallel computing is here to stay, as it was observed with the different implementations the most of the time a parallel implementation will improve the performance of algorithm. Keeping this last in mind is important to remember that not all the problems are able to parallelize, we have to take in considerations things such as race conditions, deadlocks, memory inconsistencies, and the proper tool to do use.

By the implementation of the the Sieve of Eratosthenes in these 5 frameworks, we have another point to support the parallel computing, especially now the cloud computing, as was CUDA running on an instance of Amazon Elastic Compute Cloud (Amazon EC2) from Amazon Web Services, the implementation that had a better performance compared to the sequential implementation and the other frameworks due to the architecture of a GPU, followed by C with OpenMP and surprisingly Threads on Java on local deployment.

An Application of the Sieve of Eratosthenes: the RSA algorithm (Appendix 13)

As was said on the beginning of the text, one of the main applications of the prime numbers is in the generation of the keys for Asymmetric cryptography.

For this short and real application I have modified the RSA implementation of Jon Cooper to receive the 10 million limit generated sieve, using an already generated Sieve of Eratosthenes the computational work for the main program is reduced, additionally the program has been improved by the upgrading to Python 3 and the implementation of Binary Search inside of the sieve, reducing this section of the algorithm to a time complexity of O(log(N)).

RSA uses exponentiation modulo a product of two very large primes (asked P and Q), to encrypt and decrypt, performing both public key encryption and public key digital signature.

References

Zaldívar, F. (2012). Introducción a la teoría de números. Retrieved from https://0-ebookcentral.proquest.com.millenium.itesm.mx

Sierpiński, Wacław (1988). Elementary Theory of Numbers. North-Holland Mathematical Library. 31 (2nd ed.). Elsevier.

Moore's Law and Intel Innovation. (2018). Retrieved from https://www.intel.com/content/www/us/en/history/museum-gordon-moore-law.html

The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. (2018). Retrieved from http://www.gotw.ca/publications/concurrency-ddj.htm

Welcome To The Jungle (2008). Herb Sutter. Retrieved from https://herbsutter.com/welcome-to-the-jungle/

Marowka, A. (2007). Parallel Computing on any Desktop. Retrieved from Communications of the ACM

Yılmaz, E. (2013). Pseudocode, Algorithms. Retrieved from http://www.cs.cornell.edu/courses/cs1109/2013su/lecs/lec06.pdf

OpenMP.org. (2013). About the OpenMP. Retrieved from https://www.openmp.org/about/about-us/

CUDA Zone | NVIDIA Developer. (2018) Retrived from https://developer.nvidia.com/cuda-zone

Java Fork/Join. (2018). Retrieved from https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html

Java Threads. (2018). Retrieved from https://docs.oracle.com/javase/tutorial/essential/concurrency/runthread.html

Threading Building Blocks. (2018). Retrieved from https://www.threadingbuildingblocks.org

Cooper, J. (2018). A simple RSA implementation in Python. Retrieved from https://gist.github.com/JonCooperWorks/5314103

Appendices

Appendix 1: C: Brute Force Not Optimized

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <sys/time.h>
#include <sys/types.h>

struct timeval startTime, stopTime;
int started = 0;

void start_timer() {
 started = 1;
 gettimeofday(&startTime, NULL);
}

double stop_timer() {
 long seconds, useconds;
 double duration = -1;

 if (started) {
   gettimeofday(&stopTime, NULL);
   seconds  = stopTime.tv_sec  - startTime.tv_sec;
   useconds = stopTime.tv_usec - startTime.tv_usec;
   duration = (seconds * 1000.0) + (useconds / 1000.0);
```

```
   started = 0;
 }
 return duration;
}


void initarr(int limit,int *a) {
  int c;
  //#pragma omp parallel for shared(a, limit) private(c)
  for(c = 2; c < limit; c++) {
      a[c] = 0;
  }
}



void  printprimes(int limit, int *arr) {
  int c;
  //#pragma omp parallel for shared(arr, limit) private(c)
  for(c = 2; c <limit; c++) {
      if(arr[c] == 0) {
          fprintf(stdout,"%d ", c);
      }
  }
  fprintf(stdout,"\n");
  /* code */
}
```

```
int main(int argc, char **argv) {
  int N=10;
  int limit ;
  if (argc>3){
                    fprintf(stderr,     "Error:     uso:     %s
[limite_superior_positivo]\n", argv[0]);
    return -1;

  }else if (argc==2 || argc==3) {
    int parsed=atoi(argv[1]);
    if (parsed<0){
                        fprintf(stderr,    "Error:    uso:    %s
[limite_superior_positivo]\n", argv[0]);
      return -1;
    }else{
      limit=parsed;
    }
    if (argc==3) {
      N=1;
    }
  }else {
    limit=16;
  }



    int *arr;
```

```
double ms;
ms = 0;
int i;
for (i = 0; i < N; i++) {
  start_timer();
  //->
  arr = (int*)malloc(limit * sizeof(int));
  memset(arr,0,limit*sizeof(int));
  int c;

  for(c = 2; c < limit; c++) {
      int m;
      for ( m = 1; m < c; m++) {
        if (c%m==0 && m!=1 && m!=c ){
          arr[c]=1;
          break;
        }else{
          arr[c]=0;
        }
      }

  }


  //->
  ms += stop_timer();
}
```

```
    if (argc==2){
      printf("times %i - avg time = %.5lf ms\n",N,(ms / N));
    }


    printprimes(limit, arr);




    free(arr);

    return 0;
}
```

Appendix 2: Java : BruteForce Not Optimized

```
/*-------------------------------------------------------------
-----

* Autor: A01700318 Jose Ramon Romero Chavez
*-------------------------------------------------------------
--*/

public class Main {


     public static void  printprimes(int limit, int []arr) {
       int c;
```

```
       for(c = 2; c <limit; c++) {
           if(arr[c] == 0) {
                        System.out.print(""+c+" ");
           }
       }
          System.out.print("\n");
     }
     private     static     final     int     MAXTHREADS     =
Runtime.getRuntime().availableProcessors();

    public static void main(String[] args) {


                      //System.out.println("IO: "+args[0]+" "+
args.length);


                      int N = 10;
               int limit=16;
               if (args.length>2){
                        System.err.println("Error:  uso:  Main
[limite_superior_positivo]\n");
                 System.exit(-1);
               }else if (args.length==1 || args.length==2) {
                 int parsed=Integer.parseInt(args[0]);
                 //System.err.println(parsed);


                 if (parsed<0){
                          System.err.println("Error:  uso:  %s
[limite_superior_positivo]\n");
                   System.exit(-1);
```

```
      }else{
        limit=parsed;
      }
      if (args.length==2){
        N=1;

      }

    }


        //initarr(limit, arr);

        //
        long startTime, stopTime;
        double acum = 0;
        acum = 0;
        int [] arr = new int[limit];

        for (int j = 1; j <= N; j++) {
        startTime = System.currentTimeMillis();
        arr = new int[limit];
int c;
        int m;

for(c = 2; c < limit; c++) {
    for ( m = 1; m <  c; m++) {
      if (c%m==0 && m!=1 && m!=c ){
```

```
                    arr[c]=1;
                    break;
                  }else{
                    arr[c]=0;
                  }
                }

            }



                        stopTime                              =
System.currentTimeMillis();
                        acum +=  (stopTime - startTime);
                    }
                    if (args.length==1){
                        System.out.printf("times  %d  -  avg
time = %.5f ms\n", N, (acum / N));
                    }
                    //


                printprimes(limit, arr);









        System.exit(0);
```

```
    }
}
```

Appendix 3: C :Brute Force Optimized

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <sys/time.h>
#include <sys/types.h>



struct timeval startTime, stopTime;
int started = 0;

void start_timer() {
 started = 1;
 gettimeofday(&startTime, NULL);
}

double stop_timer() {
 long seconds, useconds;
 double duration = -1;

 if (started) {
```

```
   gettimeofday(&stopTime, NULL);
   seconds  = stopTime.tv_sec  - startTime.tv_sec;
   useconds = stopTime.tv_usec - startTime.tv_usec;
   duration = (seconds * 1000.0) + (useconds / 1000.0);
   started = 0;
 }
 return duration;
}
```

```
void initarr(int limit,int *a) {
  int c;
  //#pragma omp parallel for shared(a, limit) private(c)
  for(c = 2; c < limit; c++) {
      a[c] = 0;
  }
}
```

```
void  printprimes(int limit, int *arr) {
  int c;
```

```
  //#pragma omp parallel for shared(arr, limit) private(c)
  for(c = 2; c <limit; c++) {
      if(arr[c] == 0) {
          fprintf(stdout,"%d ", c);
      }
  }
  fprintf(stdout,"\n");
  /* code */
}



int main(int argc, char **argv) {
  int N=10;
  int limit ;
  if (argc>3){
                   fprintf(stderr,     "Error:     uso:     %s
[limite_superior_positivo]\n", argv[0]);
    return -1;


  }else if (argc==2 || argc==3) {
    int parsed=atoi(argv[1]);
    if (parsed<0){
                        fprintf(stderr,    "Error:    uso:    %s
[limite_superior_positivo]\n", argv[0]);
      return -1;
    }else{
      limit=parsed;
    }
```

```
  if (argc==3) {
    N=1;
  }
}else {
  limit=16;
}
```

```
  int *arr;

  double ms;
  ms = 0;
  int i;
  for (i = 0; i < N; i++) {
    start_timer();
    //->
    arr = (int*)malloc(limit * sizeof(int));
    memset(arr,0,limit*sizeof(int));
    int c;

    for(c = 2; c < limit; c++) {
        int m;
        for ( m = 1; m < (int)sqrt(c); m++) {
          if (c%m==0 && m!=1 && m!=c ){
            arr[c]=1;
            break;
```

```
        }else{
          arr[c]=0;
        }
      }

  }









  //->
  ms += stop_timer();
}

if (argc==2){
  printf("times %i - avg time = %.5lf ms\n",N,(ms / N));
}




//
```

```
    printprimes(limit, arr);



    free(arr);

    return 0;
}
```

Appendix 4: Java: Brute Force Optimized

```
/*-------------------------------------------------------------
-----

* Autor: A01700318 Jose Ramon Romero Chavez
*-------------------------------------------------------------
--*/

public class Main {



     public static void  printprimes(int limit, int []arr) {
       int c;

       for(c = 2; c <limit; c++) {
           if(arr[c] == 0) {
```

```
                        System.out.print(""+c+" ");
            }
        }
           System.out.print("\n");
     }
     private     static     final     int     MAXTHREADS     =
Runtime.getRuntime().availableProcessors();

    public static void main(String[] args) {



                    //System.out.println("IO: "+args[0]+" "+
args.length);


                    int N = 10;
               int limit=16;
               if (args.length>2){
                        System.err.println("Error:  uso:  Main
[limite_superior_positivo]\n");
                 System.exit(-1);
               }else if (args.length==1 || args.length==2) {
                 int parsed=Integer.parseInt(args[0]);
                 //System.err.println(parsed);


                 if (parsed<0){
                          System.err.println("Error:  uso:  %s
[limite_superior_positivo]\n");
                   System.exit(-1);
                 }else{
                   limit=parsed;
```

```
      }
      if (args.length==2){
        N=1;

      }

    }


        //initarr(limit, arr);

        //
        long startTime, stopTime;
        double acum = 0;
        acum = 0;
        int [] arr = new int[limit];

        for (int j = 1; j <= N; j++) {
        startTime = System.currentTimeMillis();
        arr = new int[limit];
int c;
        int m;

for(c = 2; c < limit; c++) {
    for ( m = 1; m <  (int)Math.sqrt(c); m++) {
      if (c%m==0 && m!=1 && m!=c ){
        arr[c]=1;
        break;
```

```
                }else{
                  arr[c]=0;
                }
              }

          }


                        stopTime                        =
System.currentTimeMillis();
                        acum +=  (stopTime - startTime);
                    }
                    if (args.length==1){
                        System.out.printf("times  %d  -  avg
time = %.5f ms\n", N, (acum / N));
                    }
                    //

              printprimes(limit, arr);







        System.exit(0);
    }
}
```

Appendix 5: C: The Sieve of Eratosthenes Sequential

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <time.h>
#include <sys/time.h>
#include <sys/types.h>



struct timeval startTime, stopTime;
int started = 0;

void start_timer() {
 started = 1;
 gettimeofday(&startTime, NULL);
}

double stop_timer() {
 long seconds, useconds;
 double duration = -1;

 if (started) {
   gettimeofday(&stopTime, NULL);
   seconds  = stopTime.tv_sec  - startTime.tv_sec;
   useconds = stopTime.tv_usec - startTime.tv_usec;
```

```
   duration = (seconds * 1000.0) + (useconds / 1000.0);
   started = 0;
 }
 return duration;
}
```

```
void initarr(int limit,int *a) {
  int c;
  //#pragma omp parallel for shared(a, limit) private(c)
  for(c = 2; c < limit; c++) {
      a[c] = 0;
  }
}
```

```
void  printprimes(int limit, int *arr) {
  int c;
  //#pragma omp parallel for shared(arr, limit) private(c)
  for(c = 2; c <limit; c++) {
      if(arr[c] == 0) {
```

```
          fprintf(stdout,"%d ", c);
      }
  }
  fprintf(stdout,"\n");
  /* code */
}



int main(int argc, char **argv) {
  int N=10;
  int limit ;
  if (argc>3){
                    fprintf(stderr,     "Error:     uso:     %s
[limite_superior_positivo]\n", argv[0]);
    return -1;

  }else if (argc==2 || argc==3) {
    int parsed=atoi(argv[1]);
    if (parsed<0){
                        fprintf(stderr,    "Error:    uso:    %s
[limite_superior_positivo]\n", argv[0]);
      return -1;
    }else{
      limit=parsed;
    }
    if (argc==3) {
      N=1;
    }
```

```
  }else {
    limit=16;
  }




    int *arr;

    double ms;
    ms = 0;
    int i;
    for (i = 0; i < N; i++) {
      start_timer();
      //->
      int sqroot = (int)sqrt(limit);
      arr = (int*)malloc(limit * sizeof(int));
      //initarr(limit, arr);
      memset(arr,0,limit*sizeof(int));
      int c;
      int m;
      for(c = 2; c <= sqroot; c++) {
          if(arr[c] == 0) {
             //#pragma omp parallel for shared(arr, limit, c)
private(m)
              for(m = c+1; m < limit; m++) {
                  if(m%c == 0) {
                      arr[m] = 1;
                  }
```

```
            }

        }
    }
```

```
    //->
    ms += stop_timer();
}

if (argc==2){
    printf("times %i - avg time = %.5lf ms\n",N,(ms / N));
}
```

```
printprimes(limit, arr);
```

```
    free(arr);

    return 0;
}
```

Appendix 6: Java: The Sieve of Eratosthenes Sequential

```
/*-------------------------------------------------------------
-----

* Autor: A01700318 Jose Ramon Romero Chavez
*-------------------------------------------------------------
--*/

public class Main {


     public static void  printprimes(int limit, int []arr) {
       int c;

       for(c = 2; c <limit; c++) {
           if(arr[c] == 0) {
                        System.out.print(""+c+" ");
           }
       }
          System.out.print("\n");
     }
```

```
    private static final int MAXTHREADS =
Runtime.getRuntime().availableProcessors();


    public static void main(String[] args) {



                    //System.out.println("IO: "+args[0]+" "+
args.length);


                    int N = 10;
               int limit=16;
               if (args.length>2){
                       System.err.println("Error: uso: Main
[limite_superior_positivo]\n");
                 System.exit(-1);
               }else if (args.length==1 || args.length==2) {
                 int parsed=Integer.parseInt(args[0]);
                 //System.err.println(parsed);


                 if (parsed<0){
                         System.err.println("Error: uso: %s
[limite_superior_positivo]\n");
                   System.exit(-1);
                 }else{
                   limit=parsed;
                 }
                 if (args.length==2){
                   N=1;


                 }
```

```
                }


                    //initarr(limit, arr);

                    //
                    long startTime, stopTime;
                    double acum = 0;
                    acum = 0;
                    int [] arr = new int[limit];

                    for (int j = 1; j <= N; j++) {
                    startTime = System.currentTimeMillis();
                    int sqroot = (int)Math.sqrt(limit);
                    arr = new int[limit];
        int c;
        int m;
      for(c = 2; c <= sqroot; c++) {
          if(arr[c] == 0) {
             //#pragma omp parallel for shared(arr, limit, c)
private(m)
              for(m = c+1; m < limit; m++) {
                  if(m%c == 0) {
                      arr[m] = 1;
                  }
              }

          }
```

```
        }


                            stopTime                                =
System.currentTimeMillis();
                            acum +=  (stopTime - startTime);
                        }
                        if (args.length==1){
                            System.out.printf("times  %d  -  avg
time = %.5f ms\n", N, (acum / N));
                        }
                        //


                printprimes(limit, arr);









          System.exit(0);
      }
}
```

Appendix 7: C: OpenMP : The Sieve of Eratosthenes Parallel

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>
#include <sys/types.h>
#include <string.h>

struct timeval startTime, stopTime;
int started = 0;

void start_timer() {
 started = 1;
 gettimeofday(&startTime, NULL);
}

double stop_timer() {
 long seconds, useconds;
 double duration = -1;

 if (started) {
   gettimeofday(&stopTime, NULL);
   seconds  = stopTime.tv_sec  - startTime.tv_sec;
   useconds = stopTime.tv_usec - startTime.tv_usec;
   duration = (seconds * 1000.0) + (useconds / 1000.0);
   started = 0;
 }
 return duration;
```

```
}
```

```
void initarr(int limit,int *a) {
  int c;
  #pragma omp parallel for shared(a, limit) private(c)
  for(c = 2; c < limit; c++) {
      a[c] = 0;
  }
}
```

```
void  printprimes(int limit, int *arr) {
  int c;
  //#pragma omp parallel for shared(arr, limit) private(c)
  for(c = 2; c <limit; c++) {
      if(arr[c] == 0) {
          fprintf(stdout,"%d ", c);
      }
  }
  fprintf(stdout,"\n");
```

```
  /* code */
}



int main(int argc, char **argv) {
  int N=10;
  int limit ;
  if (argc>3){
                    fprintf(stderr,     "Error:     uso:     %s
[limite_superior_positivo]\n", argv[0]);
    return -1;


  }else if (argc==2 || argc==3) {
    int parsed=atoi(argv[1]);
    if (parsed<0){
                        fprintf(stderr,    "Error:    uso:    %s
[limite_superior_positivo]\n", argv[0]);
      return -1;
    }else{
      limit=parsed;
    }
    if (argc==3) {
      N=1;
    }
  }else {
    limit=16;
  }
```

```
    int *arr;

    double ms;
    ms = 0;
    int i;
    for (i = 0; i < N; i++) {
      start_timer();
      //->
      int sqroot = (int)sqrt(limit);
      arr = (int*)malloc(limit * sizeof(int));
      //initarr(limit, arr);
      memset(arr,0,limit*sizeof(int));

      int c;
      int m;
      for(c = 2; c <= sqroot; c++) {
          if(arr[c] == 0) {
               #pragma omp parallel for shared(arr, limit, c)
private(m)
              for(m = c+1; m < limit; m++) {
                  if(m%c == 0) {
                      arr[m] = 1;
                  }
              }

          }
```

```
  }
  //->
  ms += stop_timer();
}
if (argc==2){
  printf("times %i - avg time = %.5lf ms\n",N,(ms / N));
}
//
printprimes(limit, arr);
```

```
    free(arr);

    return 0;
}
```

Appendix 8: Java: Threads : The Sieve of Eratosthenes Parallel

```
/*-------------------------------------------------------------
-----

* Autor: A01700318 Jose Ramon Romero Chavez
*-------------------------------------------------------------
--*/

public class Main {



     public static void initarr(int limit,int []a) {
       int c;
       for(c = 2; c < limit; c++) {
           a[c] = 0;
       }
     }

     public static void  printprimes(int limit, int []arr) {
```

```
        int c;


        for(c = 2; c <limit; c++) {
            if(arr[c] == 0) {
                        System.out.print(""+c+" ");
            }
        }
           System.out.print("\n");
        /* code */
      }
     private     static     final     int     MAXTHREADS     =
Runtime.getRuntime().availableProcessors();


    public static void main(String[] args) {



                      //System.out.println("IO: "+args[0]+" "+
args.length);


                      int N = 10;
                int limit=16;
                if (args.length>2){
                         System.err.println("Error:  uso:  Main
[limite_superior_positivo]\n");
                  System.exit(-1);
                }else if (args.length==1 || args.length==2) {
                  int parsed=Integer.parseInt(args[0]);
                  //System.err.println(parsed);


                  if (parsed<0){
```

```
                        System.err.println("Error: uso: %s
[limite_superior_positivo]\n");
                    System.exit(-1);
                  }else{
                    limit=parsed;
                  }
                  if (args.length==2){
                    N=1;

                  }

                }


                    //initarr(limit, arr);

                    //
                    long startTime, stopTime;
                    double acum = 0;
                    acum = 0;
                    int [] arr = new int[limit];

                    for (int j = 1; j <= N; j++) {
                    startTime = System.currentTimeMillis();
                    int a=(int)(limit/MAXTHREADS);
                    int sqroot = (int)Math.sqrt(limit);
                    arr = new int[limit];
```

```
                    SieveThread          []          hilos=new
SieveThread[MAXTHREADS];

        for (int i = 0; i < MAXTHREADS; i++) {
                if (i != MAXTHREADS - 1) {
                                                hilos[i]=  new
SieveThread((a*i),(a*(i+1)), arr);
                } else {
                          hilos[i] = new SieveThread((i * a),
limit, arr);
                }
        }

                    int c;
                    int m;



                    for(c = 2; c <= sqroot; c++) {
                              if(arr[c] == 0) {

                                   for  (int  i  =  0;  i  <
MAXTHREADS; i++) {

     hilos[i].ct=c;
                                   }


                                   for  (int  i  =  0;  i  <
MAXTHREADS ; i++) {
                                             hilos[i].run();
```

```
                                    }


                                }
                        }
                        try {
                               for (int i = 0; i < MAXTHREADS ;
i++) {
                                        hilos[i].join();
                               }
                        } catch (InterruptedException e) {
                               e.printStackTrace();
                        }




                                stopTime                                =
System.currentTimeMillis();
                                acum +=  (stopTime - startTime);
                        }
                        if (args.length==1){
                                System.out.printf("times  %d  -  avg
time = %.5f ms\n", N, (acum / N));
                        }
                        //


                printprimes(limit, arr);
```

```
        System.exit(0);
    }
}



class SieveThread extends Thread {
     public int [] arr;
     public int ct=-1;
           public int begin;
           public int end;


    public SieveThread( int b, int e, int [] a ) {
        this.arr     = a;
        this.begin  = b;
        this.end   = e;
    }

    public int[] getArr() {
        return arr;
    }
```

```
    public void run() {
                for (int i = this.begin; i != this.end; i++) {

                    this.arr[ct] = 0;
                    //System.err.println(ct+" "+i);

                    if(i%ct == 0) {
                              this.arr[i] = 1;

                    }

                }

    }

}
```

Appendix 9: C++: Intel® Threading Building Blocks (TBB) : The Sieve of Eratosthenes Parallel

```
#include <math.h>
#include <iostream>
#include <tbb/parallel_for.h>
#include <tbb/parallel_reduce.h>
```

```
using namespace std;
using namespace tbb;

const int GRAIN = 100000;



const int N = 10;
const int DISPLAY = 100;
const int MAX_VALUE = 10000;

class Timer {
private:
    timeval startTime;
    bool  started;

public:
    Timer() :started(false) {}

    void start(){
     started = true;
        gettimeofday(&startTime, NULL);
    }

    double stop(){
        timeval endTime;
        long seconds, useconds;
```

```
        double duration = -1;

        if (started) {
               gettimeofday(&endTime, NULL);

               seconds  = endTime.tv_sec  - startTime.tv_sec;
               useconds       =       endTime.tv_usec       -
startTime.tv_usec;

               duration  =  (seconds  *  1000.0)  +  (useconds  /
1000.0);
               started = false;
        }
          return duration;
    }
};

/*
State Pattern for Initialization



class ParallelSieve {

public:
     int *myArray;
     int c;
     int state;
     ParallelSieve(int   *array,   int   cm,   int   s)   :
myArray(array), c(cm), state(s) {}
```

```
    ParallelSieve(ParallelSieve &x, split)
        : myArray(x.myArray), c(x.c) , state(x.state) {}
    void operator() (const blocked_range<int> &r) {
        if (state==0){
            for (int i = r.begin(); i < r.end(); i++) {
                    myArray[i] = 0;
        }
    }   else{
            for (int i = r.begin()+1; i != r.end(); i++) {
                myArray[c] = 0;
                if(i%c == 0) {
                    myArray[i] = 1;

                }
        }
    }
}




    void join(const ParallelSieve &x) {
        myArray = x.myArray;
    }
};


*/
```

```
class ParallelSieve {

public:
     int *myArray;
     int c;
     ParallelSieve(int  *array,  int  cm)  :  myArray(array),
c(cm)  {}

     ParallelSieve(ParallelSieve &x, split)
          : myArray(x.myArray), c(x.c) {}
     void operator() (const blocked_range<int> &r) {

               for (int i = r.begin()+1; i != r.end(); i++) {
                    myArray[c] = 0;
                    if(i%c == 0) {
                              myArray[i] = 1;

                    }
          }

}

     void join(const ParallelSieve &x) {
          myArray = x.myArray;
     }
};
```

```
void  printprimes(int limit, int *arr) {
  int c;

  for(c = 2; c <limit; c++) {
      if(arr[c] == 0) {
          fprintf(stdout,"%d ", c);
      }
  }
  fprintf(stdout,"\n");
  /* code */
}



int main(int argc, char **argv) {

     int N=10;
  int limit ;
  if (argc>3){
                    fprintf(stderr,     "Error:     uso:     %s
[limite_superior_positivo]\n", argv[0]);
    return -1;

  }else if (argc==2 || argc==3) {
    int parsed=atoi(argv[1]);
    if (parsed<0){
```

```
                        fprintf(stderr,    "Error:    uso:    %s
[limite_superior_positivo]\n", argv[0]);
      return -1;
    }else{
      limit=parsed;
    }
    if (argc==3) {
      N=1;
    }
  }else {
    limit=16;
  }

     Timer t;
     double ms;
     double result;
     ms = 0;
     int *arr ;

     for (int i = 0; i < N; i++) {
     t.start();




//

int sqroot = (int)sqrt(limit);
arr = (int*)malloc(limit * sizeof(int));
```

```
if(arr == NULL) {
          fprintf(stderr, "Error: Failed to allocate memory
for arr.\n");
          return -1;
}

//initarr(limit, arr);

int c=2;
int m;

for(c = 2; c <= sqroot; c++) {
          if(arr[c] == 0) {

      ParallelSieve  init(arr, c);
               parallel_reduce( blocked_range<int>(0, limit,
GRAIN),init );
               arr = init.myArray;

          }
}

//
```

```
            ms += t.stop();
            }

            if (argc==2){
                 cout << "times "<< N <<" - avg time = " <<
(ms/N) << " ms" << endl;
    }

    printprimes(limit, arr);



    free(arr);

    return 0;
}
```

Appendix 10: Java: Fork/Join Framework : The Sieve of Eratosthenes Parallel

```
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveAction;

public class Main {


     public static void initarr(int limit,int []a) {
       int c;
```

```
        for(c = 2; c < limit; c++) {
            a[c] = 0;
        }
      }

      public static void  printprimes(int limit, int []arr) {
        int c;

        for(c = 2; c <limit; c++) {
            if(arr[c] == 0) {
                        System.out.print(""+c+" ");
            }
        }
           System.out.print("\n");
        /* code */
      }

              private   static   final   int   MAXTHREADS   =
Runtime.getRuntime().availableProcessors();

      public static void main(String args[]) throws Exception
{

    int N = 10;
    int limit=16;
    if (args.length>2){
                      System.err.println("Error:   uso:   Main
[limite_superior_positivo]\n");
       System.exit(-1);
    }else if (args.length==1 || args.length==2) {
```

```
      int parsed=Integer.parseInt(args[0]);
      //System.err.println(parsed);

      if (parsed<0){
                          System.err.println("Error:   uso:   %s
[limite_superior_positivo]\n");
        System.exit(-1);
      }else{
        limit=parsed;
      }
      if (args.length==2){
        N=1;

      }

    }

    long startTime, stopTime;
    double acum = 0;
    acum = 0;
    int [] arr = new int[limit];


    for (int j = 1; j <= N; j++) {
    startTime = System.currentTimeMillis();
    arr = new int[limit];
```

```
//

int sqroot = (int)Math.sqrt(limit);
ForkJoinPool pool = new ForkJoinPool(MAXTHREADS);;
//

//
//initarr(limit, arr);
int c;
int m;


for(c = 2; c <= sqroot; c++) {
    if(arr[c] == 0) {

      //
        pool.invoke(new ForkSieve(0, limit, arr, c));

      //


    }
}
//
  stopTime = System.currentTimeMillis();
  acum +=  (stopTime - startTime);
}
```

```
    if (args.length==1){
      System.out.printf("times %d - avg time = %.5f ms\n", N,
(acum / N));
    }

  printprimes(limit, arr);

    System.exit(0);

     }
```

```
}
/*-------------------------------------------------------------
-----

*

* Actividad de programación: Fork-join framework

* Fecha: 23-Sep-2018

* Autor: A01700318 Ramon Romero

*

*-------------------------------------------------------------
--*/

class ForkSieve extends RecursiveAction {
     private static final long MIN = 10;
  public int [] arr;
  public int ct=-1;
  public int begin;
  public int end;


  public ForkSieve( int b, int e, int [] a, int c ) {
     this.arr   = a;
     this.begin = b;
     this.end   = e;
```

```
    this.ct    = c;
  }


    public void computeDirectly() {

   for (int i = this.begin; i < this.end; i++) {
     //System.err.println(ct+" "+i);

     this.arr[ct] = 0;
     if(i%ct == 0) {
         this.arr[i] = 1;

     }

   }
    }

    @Override
    protected void compute() {
         if ( (this.end - this.begin) <= ForkSieve.MIN ) {
              computeDirectly();

         } else {
              int middle = (end + begin) / 2;

              invokeAll(new  ForkSieve(  begin,  middle,  arr,
ct),
```

```
                              new ForkSieve( middle, end, arr,
ct));
          }
     }
}
```

Appendix 11: NVIDIA: C/CUDA: GPU / Cloud Computing: Hardware

```
H/W path    Device  Class      Description
=========================================
                    system     HVM domU
/0                  bus        Motherboard
/0/0                memory     96KiB BIOS
/0/401                processor  Intel(R) Xeon(R) CPU E5-2686
v4 @ 2.30GHz
/0/402              processor  CPU
/0/403              processor  CPU
/0/404              processor  CPU
/0/1000             memory     61GiB System Memory
/0/1000/0           memory     16GiB DIMM RAM
/0/1000/1           memory     16GiB DIMM RAM
/0/1000/2           memory     16GiB DIMM RAM
/0/1000/3           memory     13GiB DIMM RAM
/0/100              bridge     440FX - 82441FX PMC [Natoma]
/0/100/1                    bridge        82371SB  PIIX3  ISA
[Natoma/Triton II]
/0/100/1.1                  storage       82371SB  PIIX3  IDE
[Natoma/Triton II]
/0/100/1.3          bridge     82371AB/EB/MB PIIX4 ACPI
```

```
/0/100/2            display    GD 5446
/0/100/3    ens3    network    Ethernet interface
/0/100/1e           display    GK210GL [Tesla K80]
/0/100/1f           generic    Xen Platform Device

ubuntu@ip-172-31-41-114:~/ramoncuda$ sudo lshw
ip-172-31-41-114
    description: Computer
    product: HVM domU
    vendor: Xen
    version: 4.2.amazon
    serial: ec211522-4aa4-9cf2-d830-86869fcf74b0
    width: 64 bits
    capabilities: smbios-2.7 dmi-2.7 vsyscall32
                              configuration:       boot=normal
uuid=221521EC-A44A-F29C-D830-86869FCF74B0
  *-core
       description: Motherboard
       physical id: 0
     *-firmware
          description: BIOS
          vendor: Xen
          physical id: 0
          version: 4.2.amazon
          date: 08/24/2006
          size: 96KiB
          capabilities: pci edd
     *-cpu:0
          description: CPU
```

```
          product: Intel(R) Xeon(R) CPU E5-2686 v4 @ 2.30GHz
          vendor: Intel Corp.
          physical id: 401
          bus info: cpu@0
          slot: CPU 1
          size: 2701MHz
          capacity: 3GHz
          width: 64 bits
          capabilities: fpu fpu_exception wp vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ht syscall nx pdpe1gb rdtscp x86-64 constant_tsc rep_good nopl xtopology nonstop_tsc aperfmperf pni pclmulqdq ssse3 fma cx16 pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand hypervisor lahf_lm abm 3dnowprefetch invpcid_single kaiser fsgsbase bmi1 hle avx2 smep bmi2 erms invpcid rtm rdseed adx xsaveopt cpufreq
     *-cpu:1
          description: CPU
          vendor: Intel
          physical id: 402
          bus info: cpu@1
          slot: CPU 2
          size: 2697MHz
          capacity: 3GHz
          capabilities: cpufreq
     *-cpu:2
          description: CPU
          vendor: Intel
          physical id: 403
          bus info: cpu@2
```

```
     slot: CPU 3
     size: 2241MHz
     capacity: 3GHz
     capabilities: cpufreq
*-cpu:3
     description: CPU
     vendor: Intel
     physical id: 404
     bus info: cpu@3
     slot: CPU 4
     size: 2699MHz
     capacity: 3GHz
     capabilities: cpufreq
*-memory
     description: System Memory
     physical id: 1000
     size: 61GiB
   *-bank:0
        description: DIMM RAM
        physical id: 0
        slot: DIMM 0
        size: 16GiB
        width: 64 bits
   *-bank:1
        description: DIMM RAM
        physical id: 1
        slot: DIMM 1
        size: 16GiB
```

```
          width: 64 bits
     *-bank:2
          description: DIMM RAM
          physical id: 2
          slot: DIMM 2
          size: 16GiB
          width: 64 bits
     *-bank:3
          description: DIMM RAM
          physical id: 3
          slot: DIMM 3
          size: 13GiB
          width: 64 bits
*-pci
     description: Host bridge
     product: 440FX - 82441FX PMC [Natoma]
     vendor: Intel Corporation
     physical id: 100
     bus info: pci@0000:00:00.0
     version: 02
     width: 32 bits
     clock: 33MHz
   *-isa
          description: ISA bridge
          product: 82371SB PIIX3 ISA [Natoma/Triton II]
          vendor: Intel Corporation
          physical id: 1
          bus info: pci@0000:00:01.0
```

```
             version: 00
             width: 32 bits
             clock: 33MHz
             capabilities: isa bus_master
             configuration: latency=0
        *-ide
             description: IDE interface
             product: 82371SB PIIX3 IDE [Natoma/Triton II]
             vendor: Intel Corporation
             physical id: 1.1
             bus info: pci@0000:00:01.1
             version: 00
             width: 32 bits
             clock: 33MHz
             capabilities: ide bus_master
             configuration: driver=ata_piix latency=64
               resources: irq:0 ioport:1f0(size=8) ioport:3f6
ioport:170(size=8) ioport:376 ioport:c100(size=16)
        *-bridge UNCLAIMED
             description: Bridge
             product: 82371AB/EB/MB PIIX4 ACPI
             vendor: Intel Corporation
             physical id: 1.3
             bus info: pci@0000:00:01.3
             version: 01
             width: 32 bits
             clock: 33MHz
             capabilities: bridge bus_master
             configuration: latency=0
```

```
        *-display:0 UNCLAIMED
             description: VGA compatible controller
             product: GD 5446
             vendor: Cirrus Logic
             physical id: 2
             bus info: pci@0000:00:02.0
             version: 00
             width: 32 bits
             clock: 33MHz
             capabilities: vga_controller bus_master
             configuration: latency=0
                         resources:  memory:80000000-81ffffff
memory:86004000-86004fff
        *-network
             description: Ethernet interface
             physical id: 3
             bus info: pci@0000:00:03.0
             logical name: ens3
             version: 00
             serial: 0a:ca:af:93:f8:44
             width: 32 bits
             clock: 33MHz
                    capabilities:  pciexpress  msix  bus_master
cap_list ethernet physical
                     configuration:  broadcast=yes  driver=ena
driverversion=1.5.0K   ip=172.31.41.114   latency=0   link=yes
multicast=yes
             resources: irq:0 memory:86000000-86003fff
        *-display:1
             description: 3D controller
```

```
             product: GK210GL [Tesla K80]
             vendor: NVIDIA Corporation
             physical id: 1e
             bus info: pci@0000:00:1e.0
             version: a1
             width: 64 bits
             clock: 33MHz
                  capabilities: pm msi pciexpress bus_master
cap_list
             configuration: driver=nvidia latency=248
                           resources: iomemory:100-ff irq:79
memory:84000000-84ffffff         memory:1000000000-13fffffff
memory:82000000-83ffffff
        *-generic
             description: Unassigned class
             product: Xen Platform Device
             vendor: XenSource, Inc.
             physical id: 1f
             bus info: pci@0000:00:1f.0
             version: 01
             width: 32 bits
             clock: 33MHz
             capabilities: bus_master
             configuration: driver=xen-platform-pci latency=0
                     resources: irq:47 ioport:c000(size=256)
memory:85000000-85ffffff


CUDA Device Query...
There are 1 CUDA devices.
```

```
CUDA Device #0
Major revision number:         3
Minor revision number:         7
Name:                          Tesla K80
Total global memory:           11996954624
Total shared memory per block: 49152
Total registers per block:     65536
Warp size:                     32
Maximum memory pitch:          2147483647
Maximum threads per block:     1024
Maximum dimension 0 of block:  1024
Maximum dimension 1 of block:  1024
Maximum dimension 2 of block:  64
Maximum dimension 0 of grid:   2147483647
Maximum dimension 1 of grid:   65535
Maximum dimension 2 of grid:   65535
Clock rate:                    823500
Total constant memory:         65536
Texture alignment:             512
Concurrent copy and execution: Yes
Number of multiprocessors:     13
Kernel execution timeout:      No
```

Appendix 12: NVIDIA: C/CUDA: GPU / Cloud Computing: The Sieve of Eratosthenes Parallel

```
#include <math.h>
#include <stdio.h>
```

```
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>
#include <sys/types.h>


struct timeval startTime, stopTime;
int started = 0;

void start_timer() {
 started = 1;
 gettimeofday(&startTime, NULL);
}

double stop_timer() {
 long seconds, useconds;
 double duration = -1;

 if (started) {
   gettimeofday(&stopTime, NULL);
   seconds  = stopTime.tv_sec  - startTime.tv_sec;
   useconds = stopTime.tv_usec - startTime.tv_usec;
   duration = (seconds * 1000.0) + (useconds / 1000.0);
   started = 0;
 }
 return duration;
}
```

```
void  printprimes(int limit, int *arr) {
  int c;
  //#pragma omp parallel for shared(arr, limit) private(c)
  for(c = 2; c <limit; c++) {
      if(arr[c] == 0) {
          fprintf(stdout,"%d ", c);
      }
  }
  fprintf(stdout,"\n");
  /* code */
}
```

```
__global__ void init(int *arr, int sqroot, int limit) {
  int c;
  for(c = 2; c <= sqroot; c++) {
      if(arr[c] == 0) {
          /*
              #pragma omp parallel for shared(arr, limit, c)
private(m)
          for(m = c+1; m < limit; m++) {
              if(m%c == 0) {
                  arr[m] = 1;
              }
          }
```

```
          */
                int tid = c+1+ threadIdx.x + (blockIdx.x *
blockDim.x);
          if (tid<limit){
            if (tid % c ==0) {
                    arr[tid] = 1;
                }
          }



      }
  }
}


double getThreadAndInfo(cudaDeviceProp devProp)
{
  /*
       printf("Major revision number:                %d\n",
devProp.major);
       printf("Minor revision number:                %d\n",
devProp.minor);
     printf("Name:                                   %s\n",
devProp.name);
      printf("Total global memory:                  %lu\n",
devProp.totalGlobalMem);
        printf("Total shared memory per block:  %lu\n",
devProp.sharedMemPerBlock);
       printf("Total registers per block:          %d\n",
devProp.regsPerBlock);
     printf("Warp size:                              %d\n",
devProp.warpSize);
```

```
       printf("Maximum memory pitch:                  %lu\n",
devProp.memPitch);

        printf("Maximum threads per block:           %i\n",
devProp.maxThreadsPerBlock);

    for (int i = 0; i < 3; ++i)

       printf("Maximum dimension %d of block:   %d\n", i,
devProp.maxThreadsDim[i]);

    for (int i = 0; i < 3; ++i)

       printf("Maximum dimension %d of grid:    %d\n", i,
devProp.maxGridSize[i]);

      printf("Clock rate:                             %d\n",
devProp.clockRate);

       printf("Total constant memory:                %lu\n",
devProp.totalConstMem);

      printf("Texture alignment:                     %lu\n",
devProp.textureAlignment);

         printf("Concurrent copy and execution:  %s\n",
(devProp.deviceOverlap ? "Yes" : "No"));

        printf("Number of multiprocessors:           %d\n",
devProp.multiProcessorCount);

        printf("Kernel execution timeout:            %s\n",
(devProp.kernelExecTimeoutEnabled ? "Yes" : "No"));

    */

    return devProp.maxThreadsPerBlock;

}


int main(int argc, char **argv) {

    // Number of CUDA devices

    int threads=1000000;

    int devCount;

    cudaGetDeviceCount(&devCount);

    //printf("CUDA Device Query...\n");
```

```
    //printf("There are %d CUDA devices.\n", devCount);

    // Iterate through devices
    for (int i = 0; i < devCount; ++i)
    {
        // Get device properties
        //printf("\nCUDA Device #%d\n", i);
        cudaDeviceProp devProp;
       cudaGetDeviceProperties(&devProp, i);
       if (getThreadAndInfo(devProp)<threads){
         threads=getThreadAndInfo(devProp);
       }

    }

    int N=10;
    int limit ;
    if (argc>3){
                        fprintf(stderr,    "Error:    uso:    %s
[limite_superior_positivo]\n", argv[0]);
      return -1;

    }else if (argc==2 || argc==3) {
      int parsed=atoi(argv[1]);
      if (parsed<0){
                          fprintf(stderr,   "Error:   uso:   %s
[limite_superior_positivo]\n", argv[0]);
        return -1;
      }else{
```

```
    limit=parsed;
  }
  if (argc==3) {
    N=1;
  }
}else {
  limit=16;
}
```

```
  int *arr;

  double ms;
  ms = 0;
  int i;
  int *p_array;
  for (i = 0; i < N; i++) {
    start_timer();
    //->
    int sqroot = (int)sqrt(limit);
    arr = (int*)malloc(limit * sizeof(int));
    cudaMalloc((void**) &p_array, limit * sizeof(int));
    cudaMemset(p_array, 0, limit*sizeof(int));
    if (limit<=threads){
      threads=limit;
      init<<<1, threads>>>(p_array, sqroot, limit);
```

```
        }else{
             init<<<int(limit/threads)+1, threads>>>(p_array,
sqroot, limit);
        }


              cudaMemcpy(arr, p_array, limit * sizeof(int),
cudaMemcpyDeviceToHost);


        //->
        ms += stop_timer();
      }


      if (argc==2){
              printf("times %i - avg time = %.5lf ms, %i
threads\n",N,(ms / N), threads);
      }









          printprimes(limit, arr);




      free(arr);


  cudaFree(p_array);
```

```
    return 0;
}
```

Appendix 13: An Application of the Sieve of Eratosthenes: the RSA algorithm

```
import random


def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a


def multiplicative_inverse(e, phi):
    d = 0
    x1 = 0
    x2 = 1
    y1 = 1
    temp_phi = phi

    while e > 0:
```

```
        temp1 = temp_phi//e

        temp2 = temp_phi - temp1 * e
        temp_phi = e
        e = temp2

        x = x2- temp1* x1
        y = d - temp1 * y1

        x2 = x1
        x1 = x
        d = y1
        y1 = y

    if temp_phi == 1:
        return d + phi

def binarySearch(alist, item):
    first = 0
    last = len(alist)-1
    found = False
    while first<=last and not found:
        midpoint = (first + last)//2
        if alist[midpoint] == item:
            found = True
        else:
            if item < alist[midpoint]:
                last = midpoint-1
```

```
            else:
                first = midpoint+1
    return found


def getlist():
    with open('./sieve10m.txt') as f:
        content = f.readline()

    content=content[:-2]

    return list(map(int,content.split(" ")))

def is_prime(num):
    return binarySearch(getlist(),num)

def generate_keypair(p, q):
    if p<=10 or not is_prime(p):
          raise ValueError('P  must be prime and bigger than
10.')
    elif q<=10 or not is_prime(q) :
          raise ValueError('Q  must be prime and bigger than
10')
    elif p == q:
        raise ValueError('p and q cannot be equal')
    n = p * q

    phi = (p-1) * (q-1)
```

```
    e = random.randrange(1, phi)

    g = gcd(e, phi)
    while g != 1:
        e = random.randrange(1, phi)
        g = gcd(e, phi)

    d = multiplicative_inverse(e, phi)

    return ((e, n), (d, n))

def encrypt(pk, plaintext):
    key, n = pk
    cipher = [(ord(char) ** key) % n for char in plaintext]
    return cipher

def decrypt(pk, ciphertext):
    key, n = pk
    plain = [chr((char ** key) % n) for char in ciphertext]
    return ''.join(plain)



if __name__ == '__main__':

    print ("RSA Encrypter/ Decrypter")
    #
    p = int(input("Enter a prime number: "))
```

```
     q = int(input("Enter another prime number (Not one you
entered above): "))

    #p=int(4995889)

    #

    #q=int(4995911)

     print ("Generating your public/private keypairs now . .
.")

    public, private = generate_keypair(p, q)

     print ("Your public key is ", public ," and your private
key is ", private)

    message = str(input("Enter a message to encrypt with your
private key: "))

    encrypted_msg = encrypt(private, message)

    print ("Your encrypted message is: ")

    print (''.join(map(lambda x: str(x), encrypted_msg)))

    print ("Decrypting message with public key ", public ," .
. .")

    print ("Your message is:")

    print (decrypt(public, encrypted_msg))
```