COMP 460, Spring 2018 — Final Project

Repo:

https://github.com/ConwayJ18/comp460s18final/

Contents:

File Name	File Description
Driver.java	Used to run all aspects of the program at once.
src/dixon/Dixon.java	Contains all necessary elements for the Dixon algorithm
src/logicalmatrix/LogicalMatrixMultiply.java	Contains all necessary elements for Logical Matrix Multiplication
src/millerrabin/MillerRabin.java	Contains all necessary elements for Primality Testing
src/pollardrho/PollardRho.java	Contains all necessary elements for the Pollard's Rho algorithm
src/semiprime/SemiPrime.java	Contains all necessary elements for Semi-Prime Testing
src/singlethread/DixonSingleThread.java	An archival file to run Dixon on a single thread
src/singlethread/MillerRabinSingleThread.java	An archival file to run Primality Testing on a single thread
src/singlethread/SemiPrimeSingleThread.java	An archival file to run Semi-Prime Testing on a single thread

How to run:

1. Compile and run Driver.java
2. Enter the number of cores (1-8) that you have on your computer
    - This will determine the number of threads used by the program
3. Enter a number to run the non-matrix algorithms on
4. Enter a number to use as the logical matrix dimension
5. Repeat 3 & 4 as many times as desired, typing "0" when finished

Introduction

This is the documentation for the final project created by Jess Conway and Pranjali Mishra for the Spring 2018 edition of COMP 460. The subject of this project is “Implementation of Number Theoretic Algorithms on GPU” and was packaged for submission in April 2018. All files necessary to run the project can be found in the repo listed above and a description of the project can be found in this documentation and within the comments of the code itself.

Project Motivation

The motivation of this project is to create programs allowing us to implement a number of mathematical algorithms on a computer and analyze the performance of these algorithms. The faster these programs run, the more useful they become, particularly when dealing with things like factoring algorithms and their applications in cryptography. Factoring algorithms are not the only useful mathematical algorithms, though, so we want to implement as many algorithms as possible in the time allotted. This would allow us to gain a better understanding of the mathematical and algorithmic processes underlying computer science and utilize these processes to perform useful tasks at an extremely fast rate of computation.

Project Goals

The original goal of the project was to implement a set of Number Theoretic Algorithms and analyze their performance on a GPU. The set of algorithms that we had in mind were algorithms dealing with primality testing, semi-primality testing, and factoring. Furthermore, we wanted to potentially implement efficient algorithms for finding the Greatest Common Divisor (GCD) of two numbers, algorithms that solve 0/1 Matrix Problems, and algorithms that deal with elliptic curves. Once these algorithms were implemented, we would then proceed to multithread these programs to run on multiple cores on a CPU, and from there we would modify these programs to instead run on a GPU. These multithreaded GPU versions would theoretically be the most efficient versions of our programs, and we could then measure the speedup of these programs versus the originals which ran on a single thread on a CPU.

Project Description:

This is a project designed to implement a set of Number Theoretic Algorithms on one or more threads.

The selected algorithms include the following:

1. The Miller-Rabin Primality Testing Algorithm
2. A standard Semi-Prime Testing Algorithm
3. The Pollard's Rho Factoring Algorithm
4. The Dixon Factoring Algorithm
5. A Logical (0,1) Matrix Multiplication Algorithm

Each of these programs is run by the Driver.java file and can be run using multiple threads if the user so desires.

Project Design & Development:

The development of each of these programs began with an intensive study of each algorithm to understand how they work and what would need to be implemented to make them run on a computer. Jess is a mathematics major and has a particular interest in number theory, so this part of the project was allocated to him. Once sufficient understanding of the algorithms was attained, the coding was able to begin which, upon completion, led to testing of each algorithm on a single thread. Once each algorithm was completed and functional using a single thread, multithreading and parallelization could begin. This is where Pranjali largely took over as a result of her experience with programming, experience which she has more of than Jess. She was then able to take each program, identify the most labor-intensive part of the algorithm, and modify each program to parallelize these parts to run on multiple cores using static scheduling. This led to speedup of each program up to a theoretical limit proportional to the number of cores being used. Once this was completed, everything was tested again to make sure things were still functional and each program actually was more efficient. Once all this work was completed, the project was ready for submission.

Project Roles:

Jess Conway — Project lead. Wrote the base code for each algorithm.
Pranjali Mishra — Multithreaded each algorithm to run on multiple cores.

The Miller-Rabin Primality Test

The Miller-Rabin primality test is an algorithm which determines whether a given number is prime. The original algorithm was discovered by Gary L. Miller and was a deterministic algorithm and its correctness relies on the extended. Michael O. Rabin then modified that algorithm to obtain an unconditional probabilistic algorithm. The Miller-Rabin test, like many other primality tests, relies on an equality or set of equalities that hold true for prime values, then checks whether or not they hold for a number we want to test.

The algorithm can be written in pseudocode as follows (from Wikipedia):

```
Input #1: n > 3, an odd integer to be tested for primality;
Input #2: k, a parameter that determines the accuracy of the test
Output: composite if n is composite, otherwise probably prime
write n − 1 as 2^r·d with d odd by factoring powers of 2 from n − 1
WitnessLoop: repeat k times:
   pick a random integer a in the range [2, n − 2]
   x ← a^d mod n
   if x = 1 or x = n − 1 then
      continue WitnessLoop
   repeat r − 1 times:
      x ← x^2 mod n
      if x = 1 then
         return composite
      if x = n − 1 then
         continue WitnessLoop
   return composite
return probably prime
```

Semi-Prime Testing

In mathematics, a semi-prime number (also known as a bi-prime number) is a natural number that is the product of two (not necessarily distinct) prime numbers. By definition, these numbers have no composite factors other than themselves. Semi-primes are highly useful in areas such as cryptography and number theory, most notably in public key cryptography where they are used by RSA. This algorithm and it's relatives rely on the fact that finding two large numbers and multiplying them together is computationally simple, while finding the original factors of the larger product is very hard.

Our test attempts to check whether a number is semi-prime. The program first checks if the number is composite. If so, it checks for factors between 2 and cbrt(n). If it finds one such factor, it checks if that factor is prime. If so, it divides that number out from n and checks if the is prime. Only if all three of these tests pass is a number semi-prime.

The algorithm can be written in pseudocode as follows:

```
Input #1: n > 3, to be tested for semi-primality;
Output: true if n is semi-prime, otherwise false
if n is composite then
   for i=2; i < ∛n; i++:
      if n%i = 0 and i is prime then
         if n / i is prime then
               return true
return false
```

Pollard's Rho Factorization

Pollard's Rho Algorithm is an integer factorization algorithm that was invented by John Pollard in 1975. It uses a small amount of space and has an expected run time proportional to the square root of the size of the smallest prime factor of the composite number being factored. The algorithm takes the integer being factored as input and uses the polynomial $g(x) = (x^2 + 1)\ mod\ n$ to generate a pseudo-random sequence of numbers used in factorization.

The algorithm can be written in pseudocode as follows (from Wikipedia):

```
x ← 2; y ← 2; d ← 1
while d = 1:
    x ← g(x)
    y ← g(g(y))
    d ← gcd(|x - y|, n)
if d = n:
    return failure
else:
    return d
```

Dixon Factorization

Dixon's Factorization Algorithm is a general-purpose factorization algorithm, and the first to use a factor base method. It is based on finding a congruence of squares modulo the integer N which we intend to factor. Unlike other factor base methods, its run-time bound comes with a rigorous proof that does not rely on conjectures about the smoothness properties of the values taken by polynomial. The algorithm was designed by John D. Dixon and was first published in 1981.

The algorithm takes as input an integer *n* to be factored and a smoothness bound *k* before progressing as follows:

1. Find all the primes, $\{p_1, \dots, p_k\}$ such that $p_i < k$
2. If $p_i | n$ for any $1 \leq i \leq k$, return p_i.
3. Set $i = 1$.
4. while $i < k + 1$ do
    a. Choose integer α_i randomly from $[1, n-1]$.
    b. If $\gcd(\alpha_i, n) \neq 1$, return it. Else, compute $\gamma_i = \alpha_i^2 \ mod \ n$.
    c. If γ_i is y-smooth, let $v_{\gamma_i} = (e_{i_1} \ mod \ 2, \dots, e_{i_k} \ mod \ 2)$ where $\gamma_i = p_1^{e_{i_1}} \dots p_k^{e_{i_k}}$. Set $i = i + 1$.
5. Find $I \subset \{1, \dots, k+1\}$ such that $\sum_{i \in I} v_{\gamma_i} = 0$ over $\mathbb{F}_2$.
6. Let $\gamma = \sum_{i \in I} \gamma_i = p_1^{e_{i_1}} \dots p_k^{e_{i_k}}$, $\beta = p_1^{e_{i_1}/2} \dots p_k^{e_{i_k}/2} \ mod \ n$, and $\alpha = \sum_{i \in I} \alpha_i \ mod \ n$.
7. If $\gcd(\alpha + \beta, n) \neq 1$ return the factor. Else, go to step 3.

For the sake of efficiency, our smoothness bound *k* is log(N) * log(log(N)).

Logical Matrix Multiplication

A logical matrix (also known as a binary matrix, relation matrix, Boolean matrix, or (0,1) matrix) is a matric with entries from the Boolean domain B = {0, 1}. Such a matrix can be used to represent a binary relation between a pair of finite sets. The multiplication of such matrixes proceeds as with ordinary matrix multiplication, but with “x” replaced by “AND” and with “+” replaced with “OR”.

The algorithm can be written in pseudocode as follows (from Wikipedia):

```
Input #1: matrix A of dimension n;
Input #2: matrix B of dimension n;
Output: the resulting matrix C = A*B;
For i from 1 to n:
   For j from 1 to n:
      let result = 0
      for k from 1 to n:
         result ← result OR (A_{i,k} AND B_{k,j})
      C_{i,j} ← result
return C
```

Multithreading

Four of our algorithms (Miller-Rabin, Semi-Prime, Dixon, and Logical Matrix Multiplication) have been multithreaded to run on multiple cores and therefore complete their work faster than they would on a single core running a single thread. We elected to complete this multithreading using static scheduling because that seemed most appropriate given the type of work each program completed and also because we are more comfortable with static scheduling, which made it easier for us to implement.

Parallelization of programs using static scheduling generally decreases the execution time proportionally to the number of additional cores executing within the program. For example, if a program normally takes 8 seconds to run on a single core, and we were able to parallelize the entire program, we would now expect this program to run in 1 second on 8 cores.

In an effort to maximize this speedup, we made an effort to always set our parallelization to take effect during the most labor-intensive parts of each program. For Miller-Rabin this was during the repeated trial (up to the degree of certainty) of random integers to determine primality. For Semi-Prime testing this was during the brute-force division of the integers between 2 and cbrt(n). For Dixon this was during the loading of the primes into the factor base. Finally, for Logical Matrix Multiplication this was during the multiplication itself. While each of these implementations were only able to speedup specific parts of each program, we were able to record significant speedup nonetheless because of the importance of these aspects to the overall program. While we were not able to reach ideal speedup, our benchmark testing showed significant improvement nonetheless.

Conclusion

The original goal of the project was to implement a set of Number Theoretic Algorithms and analyze their performance on a GPU. While we were not able to fully realize this project by extending the programs to the point where they could run on GPU, we were able to get to the point immediately before that in which we parallelized our code to run on a CPU. The algorithms that were successfully implemented included a primality test, a semi-primality test, two factoring algorithms, and logical matrix multiplication. Each of these algorithms can run on any number of available cores. This work was divided between Jess and Pranjali according to their respective specialties. Jess has significant mathematical experience and thus took care of the implementation of the single-threaded version of these algorithms. Pranjali, on the other hand, is a strong programmer and thus took care of the parallelization of these programs. Any additional work that didn't fall into one of these categories was either delegated or completed by Jess as team leader. Furthermore, he completion of this project did, in fact, allow for a better understanding of the mathematical and algorithmic processes underlying computer science and utilize these processes to perform useful tasks at an extremely fast rate of computation. We are very happy with the work that was completed and may seek to complete the final step of GPU implementation at some point in the future.