CMDA 3634 Spring 2018 Homework 03

N. Chalmers

March 29, 2018

You must complete the following task by 5pm on Tuesday 03/27/18.

Your write up for this homework should be presented in a LaTeX formatted PDF document. You may copy the LaTeX used to prepare this report as follows

1. Click on this link
2. Click on Menu/Copy Project.
3. Modify the HW03.tex document to respond to the following questions.
4. Remember: click the Recompile button to rebuild the document when you have made edits.
5. Remember: Change the author.

Each student must individually upload the following files to the CMDA 3634 Canvas page at https://canvas.vt.edu

1. `firstnameLastnameHW03.tex` LaTeX file.
2. Any figure files to be included by `firstnameLastnameHW03.tex` file.
3. `firstnameLastnameHW03.pdf` PDF file.

In addition, all source code must be submitted to an online git repository as follows:

1. While on the webpage for you git repository, go to Settings → Collaborators.
2. Add nchalmer@vt.edu and zjiaqi@vt.edu as collaborators.
3. Make a folder named HW03 in you repository and store all relevant source files to this assignment in this folder. Ensure your assignment files compile with `make`.

You must complete this assignment on your own.

160 points will be awarded for a successful completion.
Extra credit will be awarded as appropriate.

Q0.1(30 points) 30 points is awarded for attending class on March 13th and submitting "Computing pi in parallel using OpenMP" code to GitHub. All code and any plotting results should be saved to your GitHub repository. Any additional comments can be included here in this homework.

Q0.2(30 points) 30 points is awarded for attending class on March 15th and submitting "adding threading to a Mandelbrot set generator" code to GitHub. All code and any plotting results should be saved to your GitHub repository. Any additional comments can be included here in this homework.

ElGamal Public-key Cyptography

In the previous assignment we added some safe modular products and exponentiation functions, and used these to program functions which can identify very large prime numbers p and even easily find generators of $\mathbb{Z}_p$ for some specific primes. With these tools, we can now reliably setup an ElGamal cryptographic system. In this assignment, we will implement the ElGamal encryption and decryption algorithms and begin exploring how to use parallelism if we wish to determine the secret key used in an ElGamal cryptographic system.

Part 1

In the `HW03` folder you will find two folders, `Part1` and `Part2`. This section concerns the contents of the `Part1` folder.

Setting up an ElGamal Cryptographic System: Recall that in the description of the ElGamal cryptographic system in Homework 1 we considered three different people, Alice, Bob, and Eve. In the system, Alice sets up an ElGamal cryptographic system and shares her *public-key* which consists of the prime number p, generator g, and number h. She also secretly selects a number x, keeping this number private. Using the public-key, Bob is able to send messages that only Alice can read by encrypting his messages m using p, g, and h in such a way that only Alice is able to decrypt the resulting cyphertext.

Let us consider a single program which can emulate this process. In the source file `main.c` we have begun a program which must be run with MPI using at least 2 ranks. We will label rank 0 as Alice and rank 1 as Bob.

Q1.1(5 points) The program should begin with Alice setting up her ElGamal cryptographic system. Since this the same process each time, we have added the function `setupElGamal` in the `functions.c` file. Write this function so it calculates and returns the necessary values for an ElGamal cryptographic system.

Q1.2(5 points) Alter the `main` function so that only Alice inputs a bit length from the user and sets up an ElGamal cryptographic system.

Q1.3(5 points) Use MPI to broadcast the values which are part of Alice's public-key to all MPI processes.

Encryption and Decryption: Once the ElGamal system is set up, the crucial steps that remain are the encryption and decryption functions. The `main` function currently creates an array of `Nmessages=5` messages, encrypts and then decrypts them, outputting their values along the way.

Q2.1(10 points) In the file `functions.c` we have begun implementing the `ElGamalEncrypt` function. Use the description of the ElGamal encryption process from Homework 1 to complete this function.

Q2.2(10 points) In the file `functions.c` we have begun implementing the `ElGamalDecrypt` function. Use the description of the ElGamal decryption process from Homework 1 to complete this function.

Q2.3(15 points) Alter the `main` function so that only Bob populates the message array and encrypts it. After encryption, use MPI to send the encrypted messages to Alice. Alice should then receive the messages and decrypt them. Run your program and confirm that Alice obtains Bob's original messages after decryption.

Part 2

In the `HW03` folder you will find two folders, `Part1` and `Part2`. This section concerns the contents of the `Part2` folder. Before completing this section, copy the contents of your `setupElGamal`, `ElGamalEncrypt`, and `ElGamalDecrypt` functions completed above to `functions.c` in this folder.

Cracking the ElGamal Cryptographic System: Once the messages m are encrypted, is very difficult to decrypt them without knowledge of the random number y used in the encryption algorithm, or the secret key x. In the case where we have many messages, finding each y is not feasible and we must instead determine the secret key. This amounts to finding the number x such that

$$h = g^x \bmod p.$$

Trying to solve this equation by brute force (that is, literally trying every value x) is, luckily, embarrassingly parallel. In this part we will use MPI to accelerate the process of cracking the ElGamal cryptographic system by distributing the job of trying each value of x amoungst many MPI processes.

Q3.1(5 points) Alter the `main` function so that only rank 0 sets up the ElGamal cryptographic system and then broadcast the public key information to all MPI processes.

Q3.2(15 points) In the `main` function, all MPI processes currently loop through all values of x. Partition this loop between all MPI processes by setting values for `start` and `end` so that all values of x are tested, and the amount of work each MPI process performs is roughly equal.

Parallel performance:
Q4.1(10 points) What would be a good measure of the work performed inside the loop in your program's `main` function? Use `MPI_Wtime()` to compute the total time this loop takes to run inside your program and use your measurement of work to compute your program's throughput. Output both runtime and throughput quantities at the end of your program.

Q4.2(20 points) Using New River, run your program several times for $N = 1, 2, 4, 8, 12, 16$, and 20 MPI processes. You may need to alter your main file to fix the number of bits n, rather than inputting it from the user. Plot the average runtime of your program vs N for a variety of bit lengths. In a separate plot, show the average throughput numbers reported by your program. Give some comments on how this program's performance scales with the number of MPI processes.

MY ANSWER IS HERE:
The program seems to scale the opposite of what I expected. Instead of running faster with added MPI processes, it instead runs slower with a lower throughput. I believe that this is due to the way I set up the timers within the main method.
I expected the program to become faster as more MPI processes were used, because the program splits up the calculation of the x array based on the number of total ranks. Therefore, theoretically, it should be x times faster when x number of MPI processes are used.

Bonus:(20 points) In practice, we would like to stop searching for the secret key x once it is found, rather than continuing the loop. Alter the loop in your `main` function so that at regular intervals, say every `Ninterval` iterations, you determine if the secret key has been found, and stop iterating if so. Remake your runtime and throughput plots in Q3.3. If you averaged over a large number of runs, how much faster would you expect this updated version of the code to run? Does the frequency at which you check if the secret has been found, i.e. the size of `Ninterval` affect the performance?