CMDA 3634 Spring 2018 Homework 04

Pim Silpacharn

April 12, 2018

You must complete the following task by 5pm on Tuesday 04/10/18.

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 HW04.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. `firstnameLastnameHW04.tex` LaTeX file.
2. Any figure files to be included by `firstnameLastnameHW04.tex` file.
3. `firstnameLastnameHW04.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 HW04 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.

130 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 April 3rd and submitting "Adding CUDA 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

Using the work done in the previous assignments, we now can perform the three main operations of an ElGamal cryptographic system; we can select a suitable prime and generator of $\mathbb{Z}_p$, encrypt elements of $\mathbb{Z}_p$, and also decrypt them. We then used MPI to crack an ElGamal encryption in parallel. In this assignment we will add some additional functionality to our ElGamal codes, and use parallelism given by OpenMP to improve performance.

Encrypting Strings: In this assignment we'll deal with the question we have been avoiding since Homework 1: if we can only encrypt messages m when $m \in \mathbb{Z}_p$, how do we encrypt 'real' messages, i.e. strings?

In our programs we have been using the unsigned integer data type, `unsigned int`, which holds an integer number in memory as its 32-bit binary representation. The usual `int` type is similar, but instead uses the first bit to hold the sign of the number (0 for positive, 1 for negative). Somewhat surprisingly, the 8-bit `char` data type also has a 'sign' bit. This means if you were to cast a single `char` into an integer it could take values from $[-127, 127]$. It will therefore be more convenient for us to use the `unsigned char` data type to hold our strings, as casting a single `unsigned char` to an `unsigned int` can take values from $[0, 255]$.

The goal of our program will be to consider a string as an array of `unsigned char` variables and cast this string into an array of `unsigned int` variables so that each entry is an element of $\mathbb{Z}_p$. Once this is done, we should be able to perform our usual encryption and decryption functions and then cast back to an `unsigned char` array to recover our original message.

Q1.1(5 points) To begin, we need to decide how many characters we can encode with a single element of $\mathbb{Z}_p$. Notice that, in order to encode a single character $\mathbb{Z}_p$ must contain at least 256 elements, and hence p must be at least 9 bits in size. At the beginning of the `main` function, use the user's input bit length `n` and your knowledge of the size of the prime p to set the variable `charsPerInt` to be the number of `unsigned char` variables we can represent with a single element of $\mathbb{Z}_p$.

Q1.2(15 points) In the `main` function we have set up an `unsigned char` string message. To cast this string into an array of elements of $\mathbb{Z}_p$ we need to divide it into pieces of `charsPerInt` characters. However, if the length of the string isn't divisible by `charsPerInt` we will run into issues. We therefore first pad the end of the string with `' '` characters so that its length is divisible by `charsPerInt`. We have called this function `padString`. Complete the implementation of this function in `functions.c`. (Hint: Remember that a string's termination character is `'\0'`. A useful function may be `strlen` which will return the length of an input string.).

Q1.3(15 points) Once the string is padded we must cast the string into an array of `unsigned int` values. We have called this function `convertCharToZ`. Complete the implementation of this function in `functions.c`.

Q1.4(15 points) The conversion of the integer array back to a string is done in the function `convertZToChar`. Complete the implementation of this function in `functions.c`.

Acceleration with OpenMP: The strength of OpenMP is the ease at which we can parallelize sections of our program. Here we quickly identify and parallelize some embarrassingly parallel sections.

Q2.1(5 points) The process of converting a string to an integer array and vise versa would appear to be embarrassingly parallel. Use OpenMP to parallelize these functions.

Q2.2(5 points) The encryption and decryption of the array of integers in $\mathbb{Z}_p$ is also embarrassingly parallel.

Use OpenMP to parallelize these functions.

Q2.3(20 points) Searching for the secret key at the end of the `main` function is certainly embarrassingly parallel. Use OpenMP to parallelize this loop. Afterwards, alter the loop so that if one of the OpenMP threads finds the secret key, all the threads will exit the loop. (Hint: With MPI, you did this using an Allreduce, but with OpenMP we don't need a reduction. We just need a way to share the information among the threads is a safe way).

Q2.4(20 points) Using New River, run your program several times for $1, 2, 4, 8, 12, 16$, and 20 OpenMP threads. Recreate the runtime and throughput plots from HW03 to investigate the performance of your program with OpenMP threads. Is OpenMP's parallel performance comparable to MPI? Why/why not?

Answer: Each MPI process allocates the same amount of memory, whereas with OpenMP uses a shared memory model. I personally think that OpenMP is the easiest one to get started with. In the beginning you'll only need very few directives to parallelize some of your for-loops. It is hard to understand the full OpenMP standard and it is also hard to get the best performance out of this.

Bonus:(20 points) It is sometimes more convenient to use only strings when doing cryptographic operations, only casting the characters to integer when we need to do operations in $\mathbb{Z}_p$. While you could rewrite our encryption and decryption functions to do something like this, keep them for now and instead write another function which can convert all of the cyphertext pairs $(\hat{m}, a)$ into a single string, where the first several characters of the string correspond to the first cyphertext $(\hat{m}, a)$ and so on. You will have to determine how many characters are required to encode a pair $(\hat{m}, a)$, and be careful to remember that both $\hat{m}$ and a can be any element of $\mathbb{Z}_p$ now. Print out the cyphertext as a string once it is converted.