

Contents

1	Introduction & Instructions	1
2	Project Possibilities	1
2.1	Ethical Hacking: Dictionary Attack on a Linux Shadow File	1
2.2	LU Decomposition	4
2.3	QR Algorithm	4
2.4	Quadratic Sieve Factoring	5
2.5	Elliptic Curve Factoring	8
3	Presentation & Report	10
3.1	Report	10
3.2	Presentation	10

1 Introduction & Instructions

When you are finished submit all your work through the MyClasses page for this class. Create a directory called FinalProject, put all of your materials into this directory, zip the entire FinalProject directory up into the file FinalProject.zip, and then submit this zip file to the Final Project assignment.

2 Project Possibilities

2.1 Ethical Hacking: Dictionary Attack on a Linux Shadow File

When you log onto a Linux system (or other activity requiring you to input your password) the OS searches the shadow file (located at `/etc/shadow`) for the username of the login, extracts information from the line it is on, specifically the encryption algorithm and salt. It takes the password that was input, the encryption algorithm, and salt and uses the `crypt` function from the `crypt` library which is built into `g++` to encrypt the password. This result is then compared to the one in the shadow file to authenticate the user and allow access to the system. If you have `sudo` privileges you can view the shadow file on your system with

```
sudo cat /etc/shadow
```

You don't want to edit this file or you will run the risk of not being able to log into your own computer. Each line of this file is of the form

```
<username>:$<algorithm id>$<salt string>$<encrypted password>
```

For example,

```
despickler:$1$ab$FPyWVGc2x83IsQ7.q775k1
```

The username here is `despickler`, the number between the first set of \$ symbols is the encryption/hashing algorithm that is being used, depending on the system this can be any number of strings. You can find a list of the ones supported by your system in the man pages. The next string in the set of \$ symbols is the salt string. The salt string is used to hinder a dictionary attack, which is what you will be doing in this exercise.

The salt string is simply set of random characters chosen by the OS that is appended to the user's password before encryption. For example, say the user is peter, his password is "fluffybunny", and the system chose "abc" as the salt, the system will encrypt the password as "abcfluffybunny" to get a shadow file entry of

```
peter:$5$abc$A9rdTaiz4GZYERYE22q6P6XqGZ5o9buuxrzaXIrLA/0
```

Here it used 5 as the encryption/hashing algorithm which is SHA-256.

A dictionary attack is literally that, a brute force attack trying words from the dictionary as passwords. Most users will not simply use English words as their passwords, although many do. They usually append numbers as prefixes or suffixes, such as "bunny123" or "55attack". They may combine words, such as "fluffybunny". They may do a combination of both, "fluffy123bunny" or "55fluffy123bunny987", They will sometimes change the cases of letters (which is sometimes insisted upon by the system) such as "FluffyBunny" or "fluffyBunny". And of course the inclusion of special symbols like '.', '%', '*', etc.

As a side note, this is where the salt comes in to hinder this type of attack. If there were no salt added to the password you could run through the dictionary using various encryption schemes and store all the encrypted passwords in a file, then simply search the file for a user's encrypted password from the shadow file. Now if a simple three character salt is added using just upper and lower case letters then the number of salts that could be added to each password is $52^3 = 140608$. So the work you would need to do to preprocess a set of encrypted passwords has increased 140,608 times. Even if you just had access to the password portion and the algorithm, for example knowing that the system used SHA-256 and the encrypted password was

```
A9rdTaiz4GZYERYE22q6P6XqGZ5o9buuxrzaXIrLA/0
```

you would still need to do your attack with each possible salt.

In this exercise you will play the role of the "ethical hacker". You have obtained access to a shadow file, I will provide a fake one. You also have a dictionary of words, I will provide one of these as well, it will simply be a dictionary file used by Linux systems for spell checkers for programs like LibreWriter. You will use the dictionary along with the possible manipulations of passwords discussed above to break as many passwords as you can. Since you have access to the shadow file you will know the algorithm that was used and the salt that was applied

to the password, significantly simplifying the attack.

To use the `crypt` function you need to add in a `#define _XOPEN_SOURCE` flag before your includes, you also need to include `crypt.h`. When you compile the code (really the linking stage) you will need to include the `crypt` library in the linking. For example, if the code file is `makeshadow.cpp` then the following command will compile the program.

```
g++ makeshadow.cpp -lcrypt
```

The program itself that created the `peter` and `fluffybunny` entry is below. Note that there is a thread-safe version of this command as well.

```
1 #define _XOPEN_SOURCE
2 #include <crypt.h>
3 #include <stdio.h>
4
5 int main(int argc, char **argv) {
6
7     char *pw;
8     FILE *fp;
9
10    fp = fopen("shadow", "w+");
11
12    pw = crypt("fluffybunny", "$5$abc$");
13    fprintf(fp, "%s:%s\n", "peter", pw);
14
15    fclose(fp);
16
17    return 0;
18 }
```

A few specific requirements.

1. Code this dictionary attack in
 - (a) Serial.
 - (b) Parallel using shared memory and OpenMP.
 - (c) Parallel using distributed memory and MPI.
2. Do the standard timing analysis to test speedup and efficiency of the code on both a single machine in the HPCL. and on your cluster.
3. I will give you a fake shadow file to attack and a dictionary list of words. You will probably want to make your own shadow file for testing as well.
4. Get as many passwords from this file as you can. Also report on ones that were not cracked.
5. There are many ways to break this work up between processors. Consider several approaches and combinations of these.
6. You will probably want to save your cracked passwords to a file, there are many ways to do this on a parallel system. You may want to investigate different ways of accomplishing this to see what best fits your needs.

7. Include a short paper summarizing the timing analysis, the approaches that worked, the approaches that did not work as well, a list of cracked passwords and those that were not cracked.

2.2 LU Decomposition

Simply stated, *LU* Decomposition is the process of taking a square matrix A and writing it as $A = LU$, the product of two matrices L and U , where L is lower triangular and U is upper triangular. Hence this is sometimes called *LU* Factorization. This is used extensively in the solutions to large linear systems and has numerous applications in science and engineering. In fact, it is heavily incorporated into the Linpack benchmarking software used for benchmarking supercomputers. There are numerous resources on this process. Most introductory linear algebra textbooks discuss this method and many other resources can be found online. Research the different methods for calculating this and parallelizing this method. Try several approaches and see which works best.

A few specific requirements.

1. Code this in
 - (a) Serial.
 - (b) Parallel using shared memory and OpenMP.
 - (c) Parallel using distributed memory and MPI.
2. Do the standard timing analysis to test speedup and efficiency of the code on both a single machine in the HPCL. and on your cluster.
3. There are many ways to break this work up between processors. Consider several approaches and combinations of these.
4. Include a short paper summarizing the timing analysis, the approaches that worked, the approaches that did not work as well, and applications for this process.

2.3 QR Algorithm

The QR Algorithm is a way to find all the eigenvalues of a square matrix. The method was developed in the 1960's and is considered one of the top-ten developments in computation in the 20th century.

Unlike the power method that finds only the dominant real eigenvalue, the QR Algorithm will find all the eigenvalues, both real and complex, of a square matrix with real number entries. The algorithm works just as well with matrices that have complex number entries and much of the literature on this will probably be using matrices over \mathbb{C} as opposed to \mathbb{R} , but you can concentrate on real matrices.

The algorithm relies on the QR-factorization of a matrix. The QR-factorization is where we take a square matrix A and write it as, $A = QR$ where $Q^*Q = I$ and R is upper triangular. A sequence of matrices is constructed that converge to an upper block triangular matrix. The values below the main diagonal converge to 0. This sequence of matrices are all similar to the original matrix A , that is if a matrix M is in this sequence then $M = BAB^{-1}$ for some invertible matrix B . Hence A and M have the same eigenvalues. Since this sequence of M matrices converges to an upper block triangular matrix the eigenvalues are simply the entries on the diagonal. A little bit of a lie there, more specifically, if the block is 1×1 then this entry is an eigenvalue. If the block is 2×2 then the eigenvalues are complex (conjugate pairs) and can be found by solving the quadratic characteristic matrix on that 2×2 , which is a very simple task. This algorithm is fairly expensive in computation time, on the order of $O(n^3)$, and hence could benefit from parallelization.

There are many ways to do these computations and many options to consider. This is primarily a technique in numerical linear algebra and you will probably not find this method in a standard introductory linear algebra textbook. On the other hand, if you look in a numerical linear algebra text it will most likely be there and of course there are many online resources for this method.

A few specific requirements.

1. Code this in
 - (a) Serial.
 - (b) Parallel using shared memory and OpenMP.
 - (c) Parallel using distributed memory and MPI.
2. Do the standard timing analysis to test speedup and efficiency of the code on both a single machine in the HPCL. and on your cluster.
3. There are many ways to break this work up between processors. Consider several approaches and combinations of these.
4. Include a short paper summarizing the timing analysis, the approaches that worked, the approaches that did not work as well, and applications for this process.

2.4 Quadratic Sieve Factoring

The RSA algorithm was developed by three professors at MIT in 1977, Ron Rivest, Adi Shamir, and Leonard Adleman, their initials give the algorithm its name. This was one of the first algorithms to implement the concept of public-key cryptography. In 1976, when Whitfield Diffie and Martin Hellman came up with the idea of public-key cryptography, they did not have a method for implementing the concept. Rivest, Shamir and Adleman took the concept of Diffie and Hellman and devised a method that uses the one-way function of integer multiplication to devise a digital implementation.

As a historical note, Rivest, Shamir and Adleman were not the first mathematicians to discover this technique. In 1973, Clifford Cocks, a British mathematician and cryptographer at the Government Communications Headquarters (GCHQ), had developed an equivalent system. The Government Communications Headquarters is a British intelligence agency responsible for providing signals intelligence and information assurance to the UK government and armed forces. GCHQ was originally established after the First World War as the Government Code and Cypher School (GC&CS or GCCS). During the Second World War it was located at Bletchley Park, which is where the German Enigma machine was cracked. The GCHQ is the British equivalent of the NSA (National Security Agency) in the United States. Hence anything that was discovered at GCHQ had to remain classified and Clifford Cocks did not get the recognition, or the money, from the discovery of the method. In fact, it was not until 1997 that GCHQ declassified his work.

Currently the RSA algorithm is heavily used in monetary transactions from simple online ordering to intra-bank transactions. In fact, every time you make an online purchase your credit card information is encrypted with this algorithms before it leaves your computer and is transmitted to the company you are ordering from. This is currently a very secure public-key system but if a method was devised to quickly crack this algorithm (e.g. the development of more powerful quantum computers) then the entire e-commerce system as we know it would collapse and we would need to completely alter the way these transmissions are made to compensate, for instance, using post-quantum cryptographic methods, a currently active research area.

The RSA algorithm relies on the one-way function of integer multiplication of large integers. So one way to break this code is simply to take the large integer “key” (that is a product of two large prime numbers, i.e. a semiprime number) and factor it. In fact, if you can factor this number then you have broken the code. Seems simple, and mathematically it is, but all of the current factoring algorithms, that we know about, run in exponential time. As an example, you can find this key and all the other parameters used for the encryption algorithm by looking at the certificate of an ordering page say from Amazon. Most web browsers will show this to you by looking at the properties of the page/certificate, probably just right click on the little lock icon and view the properties. In this you will see the big number denoted as n or modulus. It is usually written in hexadecimal format but is easily convertible to decimal. These numbers are usually around 600 digits in length (for 2048 bit encryption) but can be much larger. So all the information you need to break this code is publicly available. The catch is that even if you were to use the fastest factoring algorithms available and combine all of the computers on the planet it would still take billions of years to factor the number. This is why the method is so secure.

One of the fastest factoring methods to date is the Quadratic Sieve Factoring method. It has many offshoots and ways to increase its speed. It is also very parallelizable. In fact, one of the first projects that utilized the internet for distributed computation was to break one of the RSA challenges. In this, people donated the idle time of their computers to help factor the challenge number. Their computers would process parts of the Quadratic Sieve algorithm that could be done in parallel (embarrassingly) and these results were transmitted back via the web to the control machine that would finish the computation. While this

is a very common practice today (see the SETI and GIMPS projects for example), it was groundbreaking at the time.

Basically, you search for numbers n such that when you take n^2 modulo the number you are trying to factor, that is the residue, it factors into a product of small primes. These are called B -smooth numbers. If you get enough of these B -smooth numbers and their small prime factorizations you can do some (modulo 2) matrix row reductions to solve a matrix that will then give you a set of dependencies. These dependencies when combined will give you two numbers x and y such that $x^2 \equiv y^2 \pmod{n}$. From this you trivially get, $x^2 - y^2 \equiv 0 \pmod{n}$, that is $(x+y)(x-y) \equiv 0 \pmod{n}$. So if $x+y \not\equiv 0 \pmod{n}$ and $x-y \not\equiv 0 \pmod{n}$ you have a non-trivial factorization of n .

You can find discussions of this method in most books on cryptography and of course there are numerous resources online.

A few specific requirements.

1. Code this in
 - (a) Serial.
 - (b) Parallel using shared memory and OpenMP.
 - (c) Parallel using distributed memory and MPI.
2. You will be using very large exact integers for this project, much larger than is supported by C++. I will give you the InfInt: Arbitrary-Precision Integer Arithmetic Library by Sercan Tutar that has all the functionality you will need. It has many overloaded operators and conversion functions, in fact, even a conversion from a string representation of a number into its framework. You can also find this online. It is one of the fastest and easiest to use arbitrary precision integer libraries I have found and is simply a single .h file, so incorporating it into your code is trivial.
3. I will give you a file of numbers to factor but you will want to do some testing of this on your own numbers. You should write this program to completely factor a number, no matter how many factors it has but you should first concentrate on semiprime numbers, that is a number that is the product of two primes. Most computer algebra systems have a way to get the next prime number given a number. For example in the cross-platform open-source package Maxima (wxMaxima) the process to create a semi-prime number (as well as use Maxima to factor it) is below.

```
(% i1) p:next_prime(430527575045);
```

```
430527575077                                (% o1)
```

```
(% i2) q:next_prime(12947619890321);
```

```
12947619890323                             (% o2)
```

```
(% i3) n:p*q;
```

```
5574307394399493888279871 (% o3)
```

```
(% i4) factor(n);
```

```
430527575077 12947619890323 (% o4)
```

4. Do the standard timing analysis to test speedup and efficiency of the code on both a single machine in the HPCL. and on your cluster.
5. There are many ways to break this work up between processors. Consider several approaches and combinations of these.
6. Include a short paper summarizing the timing analysis, the approaches that worked, the approaches that did not work as well, and applications for this process. Also report on the numbers you were able to factor and those that were not, as well as execution times for them.

2.5 Elliptic Curve Factoring

Read the first four paragraphs of the Quadratic Sieve Factoring section above.

Another very fast factoring algorithm was developed by H. W. Lenstra and uses the concept of an Elliptic Curve in its calculations. You really do not need to know anything about elliptic curves to code this and frankly the amount of Elliptic Curve theory that is involved is fairly trivial. Elliptic Curves have become widely used in cryptography since Lenstra's discovery in fact there is an entire field of Elliptic Curve Cryptography. Lenstra's algorithm is asymptotically the third fastest integer factorization algorithm known, the faster ones are the Quadratic Sieve and the General Number Field Sieve.

The algorithm, in a nutshell, is to take a random elliptic curve modulo the number you intend to factor and a random point on the curve and calculate large multiples of that point until there is an error in the computation, which corresponds to a non-trivial factorization of the modulus. If one curve does not give a result then usually another random elliptic curve and point are chosen for a repeat of the process. This general description gives an immediate first approach to parallelization.

You can find discussions of this method in most books on cryptography and of course there are numerous resources online.

A few specific requirements.

1. Code this in
 - (a) Serial.
 - (b) Parallel using shared memory and OpenMP.

- (c) Parallel using distributed memory and MPI.
2. You will be using very large exact integers for this project, much larger than is supported by C++. I will give you the `InfInt: Arbitrary-Precision Integer Arithmetic Library` by Sercan Tutar that has all the functionality you will need. It has many overloaded operators and conversion functions, in fact, even a conversion from a string representation of a number into its framework. You can also find this online. It is one of the fastest and easiest to use arbitrary precision integer libraries I have found and is simply a single .h file, so incorporating it into your code is trivial.
 3. I will give you a file of numbers to factor but you will want to do some testing of this on your own numbers. You should write this program to completely factor a number, no matter how many factors it has but you should first concentrate on semiprime numbers, that is a number that is the product of two primes. Most computer algebra systems have a way to get the next prime number given a number. For example in the cross-platform open-source package Maxima (wxMaxima) the process to create a semi-prime number (as well as use Maxima to factor it) is below.

```
(% i1)  p:next_prime(430527575045);
```

```
430527575077                                (% o1)
```

```
(% i2)  q:next_prime(12947619890321);
```

```
12947619890323                             (% o2)
```

```
(% i3)  n:p*q;
```

```
5574307394399493888279871                (% o3)
```

```
(% i4)  factor(n);
```

```
430527575077 12947619890323               (% o4)
```

4. Do the standard timing analysis to test speedup and efficiency of the code on both a single machine in the HPCL. and on your cluster.
5. There are many ways to break this work up between processors. Consider several approaches and combinations of these.
6. Include a short paper summarizing the timing analysis, the approaches that worked, the approaches that did not work as well, and applications for this process. Also report on the numbers you were able to factor and those that were not, as well as execution times for them.

3 Presentation & Report

Each group will submit a report and do a presentation during the final exam time for the class.

3.1 Report

The report should be a summary of the work done on the project and the analysis of the results. To be included,

1. Each member's contribution to the final project.
2. Approaches discussed and tried to the parallelism of the task.
3. What worked, what didn't work, what worked better, and why.
4. Results of the project.
5. Timing analysis of the code on different platforms, single HPCL machine, single cluster machine, entire cluster using just cores and using hardware threads.
6. Discussions and even predictions of how well the final code will scale on different platforms and how well the load was balanced during the computation.
7. Include a bibliography of all the resources used.

3.2 Presentation

Each group will give a 30 minute presentation on their project during the final exam period for this class. The presentation should be 30 minutes in length with 5 to 10 minutes for questions and shifting to the next group.

The members of the group should have an equal portion of the presentation. The presentation should,

1. Summarize the project.
2. Give some history of the methods.
3. Give some applications of the methods.
4. Give the significance to the computational landscape.
5. Methods of parallelization you tried.
6. Methods that might have been tried but were abandoned, and why they were abandoned.

7. Results of the running and timing of the runs on the different platforms.
8. Discussions and even predictions of how well the final code will scale on different platforms and how well the load was balanced during the computation.