An Improved Parallel Implementation of Rainbowcrack Using MPI

Edward R. Sykes, Wesley Skoczen

Faculty of Applied Science and Technology
Sheridan College Institute of Technology and Advanced Learning
1430 Trafalgar Road, Oakville, Ontario, Canada
{ed.sykes, wesley.skoczen}@sheridanc.on.ca

Abstract. Nearly three decades ago, Martin Hellman introduced a cryptanalytic time-memory trade-off algorithm which reduced the time of cryptanalysis by using precalculated data. Rivest shortly improved this technique by introducing distinguished points which significantly reduced the frequency of memory lookups during cryptanalysis. In 2003, Philip Oechslin proposed a new and improved algorithm which reduces the number of calculations during cryptanalysis by a factor of two. In this research we present the results of a parallel implementation of Oechslin’s faster time-memory trade-off algorithm using MPI on a SHARCNET supercomputer. We use MS-Windows hashes (alphanumeric sequences of length 14 characters or less). We can construct 119GB of data in 6 days and crack 99.9% of all passwords hashes in 6 seconds or less. On a standard desktop machine, the same task requires 2,354 days (6 years!) to construct the data and 3-15 minutes to crack the password.

Keywords: Rainbow tables, MPI, time-memory trade-off, password cracking, parallel processing, cryptanalysis.

1 Introduction

To be successful, cryptanalytic attacks using exhaustive search techniques require significant computational power or a lot of time. If a previous attack has been executed several times, it may be possible to perform the exhaustive search in advance and store the results in memory. Storing this precomputation in memory allows for an attack to be performed very quickly. However, in current systems, including supercomputers, this method is not yet feasible because of the huge amount of memory needed (i.e., hundreds of gigabytes). However, there are methods available that trade memory against attack time [1,2,3]. For example, Hellman proposed a method by which a cryptosystem having \( n \) keys could recover a key in \( n^{2/3} \) operations using an equal number of words of memory [1]. Typical applications of this method include the recovery of a key when both the plaintext and ciphertext are known (e.g., password hashes).

Many popular operating systems generate password hashes by encrypting a fixed plaintext with the user’s password as the key and store the result as the password hash. If the password hashing scheme is not well-designed, the plaintext and the
encryption method will be the same for all passwords [1]. Depending on the design of
the hashing algorithm, the plaintext and the encryption method may be the same for
all the passwords [1]. If this is the case, then password hashes can be calculated in
advance and an efficient attack can be performed using the time-memory trade-off
technique [2].

RainbowCrack is a general purpose implementation of Oechslin’s faster time-
memory trade-off technique [3]. The precomputation is done in advance and stored in
a collection of files called rainbow tables. It takes a substantial amount of time to
construct rainbow tables, however by using the tables the time required to crack a
password is hundreds of times faster when compared to a brute-force hash cracker [3].

A rainbow table consists of many pairs of plaintext strings in the following form:
(startpoint, endpoint). Each pair is constructed by an iterative hashing process
which applies the hash function, H to the plaintext followed by a reduction function
(R₁ through Rₖ) in an alternating fashion as shown in the following example:

aaaaaa —H→ 281DAF40 —R₁→ sgfnyd —H→ 920ECF10 —R₂→ kiebgt

Figure 1 shows a section of a Rainbow table consisting of three rows. In the
example above “aaaaaa” is a startpoint and “kiebgt” is an endpoint. Each startpoint
and endpoint is 8 bytes in size.

```
00 00 00 00 00 00 00 00 00 00 30 02 3C 61 01 00 00 00
01 00 00 00 00 00 00 00 00 00 77 09 F0 98 06 00 00 00
02 00 00 00 00 00 00 00 00 00 14 49 40 CB 0A 00 00 00
```

![startpoint](startpoint) ![endpoint](endpoint)

**Fig. 1.** Sample contents of a Rainbow table.

Suppose we are given a hash “A2A63B”; the following steps illustrate how
Rainbow tables are used in the password cracking process (please refer to Figure 2).

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Using the given hash, compute the last reduction used in the table (in this case R₃) and check if the password is in the last column of the Rainbow table.</td>
</tr>
<tr>
<td>2</td>
<td>If it does not (i.e., <em>odyssey</em> does not appear in the table), then compute the chain with the last two reductions¹</td>
</tr>
<tr>
<td>3</td>
<td>If the previous step succeeds (i.e., <em>mako</em> appears at the end of the chain and as an endpoint in the table), then the startpoint (i.e., <em>whale</em>) is used as the starting point for the computation of a new chain—the chain that contains our hash.</td>
</tr>
<tr>
<td>4</td>
<td>A new chain is generated using the startpoint <em>whale</em> and during each</td>
</tr>
</tbody>
</table>

¹ This process of chain computations continues from Rₖ down to R₁ until the chain contains the password or the attack fails.
iteration (i.e., alternating $H$ and $R_i$ where $0 < i \leq k$) compares the calculated hash with the initial hash ($A2A63B$). When we find a match, the plaintext password is the string prior to the $H$ that matched the hash values (i.e., *this_is_it*).

RainbowCrack consists of the following components:

- **rtgen**: program to generate rainbow tables.
- **rtsort**: program to sort rainbow tables generated by *rtgen*.
- **rcrack**: program to lookup rainbow tables sorted by *rtsort*.

The purpose of this research is to explore Oechslin’s faster time-memory trade-off technique in the context of determining the degree of potential parallelism that may be applied and to design efficient parallel techniques to optimize the execution time for this algorithm. Specifically, we focus on:

i) reducing the precomputation time (i.e., minimize the time required to construct the rainbow tables) by using MPI (Message Passing Interface) parallel programming techniques;

ii) reducing the time to crack that hash by using MPI and parallel programming optimizations on a supercomputer; and

iii) performing a series of tests to reveal empirical findings as to the efficiency of our parallel implementation.

The motivation for this research is to explore parallel programming techniques using supercomputers to reduce the overall computational time required by Oechslin’s algorithm. Currently, there are no parallel RainbowCrack implementations. We aimed...
to increase the computational efficiency and decrease the execution time by investigating the parallelization of the precomputation and the cracking algorithm.

2 RainbowCrack – An Overview

In this research we designed and implemented a parallel implementation of Philip Oechslin’s faster time-memory trade-off algorithm for cracking hash passwords using a SHARCNET supercomputer. The Shared Hierarchical Academic Research Computing Network (SHARCNET) is a world-class consortium of 17 Ontario colleges, universities and research institutes in a “cluster of clusters” of high performance computers linked by advanced fiber optics. Its unique infrastructure is designed to meet the computational needs of researchers in diverse research areas and to facilitate the development of leading-edge tools for high performance computing grids. Established in 2001, SHARCNET provides advanced computational equipment to accelerate the production of research results for academic and industry partners. Its members seek linkages between academic researchers and corporate partners in new business opportunities; to attract and retain the best students, researchers and companies; and to create new opportunities for further developing Canada’s knowledge based economy.

Our implementation of RainbowCrack started on the SHARCNET supercomputer called “Bruce.” Bruce has 128 CPUs, 8 GB memory and runs on HP Linux XC 3.1 with a total storage capacity of 2800 GB. However, the shared environment and job queue wait times were quite prohibitive. To accelerate the process we moved to two other SHARCNET supercomputers to continue our research and development—“Whale” with 3072 processors, 4 GB memory; and “Requin” with 1536 processors and 8 GB memory.

Our primary task in this project was to design and implement a parallel algorithm for RainbowCrack. This was done by exploring the current RainbowCrack code developed by Zhu Shuanglei [5]. Shuanglei developed this code in C++; however, it is designed for a single CPU machine [5]. MPI allows multiple CPUs to communicate with one another using a set of APIs that are callable in C++ and C. By using MPI efficiently we were able to use \( n \) number of processors to generate rainbow tables.

In ability and efficiency for RainbowCrack to crack a password is based on the probability that a sequence (or partial sequence) of the password is resident in a rainbow table. The probability that a given table will contain a match for the password is given by the following MATLAB script [5]:

```matlab
function ret = calc_success_probability(N, t, m)
    arr = zeros(1, t - 1);
    arr(1) = N * (1 - exp(-m/N));
    for i = 2 : t - 1
        arr(i) = N * (1 - exp(-arr(i-1)/N));
    end;
    ret = 1;
    for i = 1 : t - 1
        ret = ret * (1 - arr(i)/N);
    end;
```
\[ ret = 1 - ret; \]

where \( N \) represents the character set, \( t \) is the length of the chains and \( m \) is the total chain count. During our tests we worked with all four categories of passwords:

\[
\begin{align*}
\text{alpha} &= [ABCDEFGHIJKLMNOPQRSTUVWXYZ] \\
\text{alpha-numeric} &= [ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789] \\
\text{alpha-numeric-symbol14} &= [ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789!@#$%^&*()-_+=] \\
\text{all} &= [ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789!@#$%^&*()-_+=\~\_|\[\]\{\}\|\:\;\"\'<\>,.?/]
\end{align*}
\]

As an example, consider the first category (i.e., alpha). In this case, the keyspace would be \( 8353082582 \). Therefore, a table with dimensions 2,100 by 8,000,000 would have about a 75% chance of containing any given password in that range. Five such tables would give an overall probability of 99.9%. Larger character sets require upwards of 30 tables with more and longer chains to achieve the same level of success [3].

3 \ RainbowCrack – Table generation

During the design of this work, we examined Zhu Shuanglei’s code and reviewed it for potential parallelization. We analyzed the code and looked for places where a bottleneck was occurring and what exactly was taking the most time. From the information we collected, we noticed that the generation of the tables was significantly longer than any other process of RainbowCrack. The second slowest process was the sorting of the tables and then finally searching.

Shuanglei’s code consists of four binaries which share some classes in common: rtgen, rtsort, rcrack and rtdump. rtdump is used to display various statistics about a given table and other maintenance purposes and was not included in our parallel implementation.

We decided to tackle the generation process first. We found that most of this work was generated within two nested for loops, the outer loop being responsible for writing the total number of chains and the inner one responsible for each transformation in the chain. Since the initial starting value of each chain is randomly generated, each chain can be computed independently. This scenario is perfectly parallel since no inter-process communication is necessary. The modifications we made were the initialization and finalization of MPI, removing the nice(19) statement to maintain a high priority, as well as a new variable to hold the number of chains that each process should calculate, expressed by:

\[
\text{int chainsPerProc} = (\text{int} \text{ceil}(n\text{RainbowChains}/n\text{Processes})); \quad (1)
\]

and using that variable as the exit condition for the outer for loop. Also, in the original code, the first value of the chain was written to disk as soon as it was generated, the transformations were calculated and the final value of the chain is written. We altered
this so that the first value is saved in a buffer and written to disk with the last value in one operation, cutting I/O operations in half.

In this research, three different MPI design approaches for table construction were implemented in an effort to determine the most efficient algorithm from a runtime perspective. These implementation approaches are shown in Figure 3. In the first approach our MPI rtgen version is executed on multiple processors and each process is writing to separate files (chunks collectively representing one Rainbow table). This approach used standard native file library functions. The second approach used multiple processors where each process writes to their own file using MPI file library functions. The third technique implemented involved multiple processors using MPI file library functions where all MPI rtgen processes write to the same file.

4 RainbowCrack – Sorting

The next component that we focused on was the rtsort algorithm. This program sorts the rainbow tables generated by rtgen. Sorting in general from a parallel programming perspective is a problem that is classified as synchronous parallelism since each process can work more or less independently but significant communication overhead is required. The original code of rtsort from Shuanglei checks the available physical memory and if the entire table can fit, it is sorted via the quicksort algorithm. However, if the table is too big, it is broken up into chunks; each chunk is then sorted (by quicksort) and written to a new temporary file and then mergesorted with the other chunks.
We discovered that the \texttt{rtsort} was fast (less than a minute for a 611 MB file) when memory was available and a little bit longer when the system didn’t have enough memory to fit the entire table and temporary files were used. Our implementation of the \texttt{rtsort} assigns to each process to read an equal portion of a file (i.e., Rainbow table), quicksort it and then write it back to the same file. In the next iteration however, only half of the processors would mergesort the file reducing it finally down to process zero which would do the final sort (see Figure 4). The algorithm is presented below.

Algorithm MPI\_rtsort()

\begin{verbatim}
while (nProcesses >= 1) {  
    if (rank < nProcesses) {  
        do all (read sort write) (Note 1)  
        nProcesses = nProcesses/2;  
    }  
    (synchronize) (Note 2)  
}
\end{verbatim}

\textbf{Note 1:} (read sort write):

\begin{verbatim}
MPI\_File\_seek(fh, rank * nFileLenOrg/nProcs, MPI\_SEEK\_SET);
MPI\_File\_read(fh, pChain, nFileLenOrg/nProcs, MPI\_INT, &status);
\end{verbatim}

\textbf{Note 2:} (synchronize)

A barrier is required for synchronization:

\begin{verbatim}
MPI\_Barrier(MPI\_COMM\_WORLD);
\end{verbatim}

We discovered that in the single process \texttt{rtsort} and our parallel MPI implementation the bottleneck was the size of the file passed to memory. When the system memory was low the chunks were sorted and written to a new temporary file immediately for single process and in the last pass for MPI implementation which resulted in additional delay.

5 RainbowCrack – Searching

Searching was the last component that we attempted to optimize using parallel techniques. The searching component of Oechslin’s algorithm is handled by \texttt{rcrack} of RainbowCrack. This program seeks to find potential matches to the password within the rainbow tables that were constructed in the previous steps (i.e., \texttt{rrtgen} and \texttt{rtsort}). The \texttt{rcrack} program uses a binary search algorithm which is $O(\log_2 n)$ complexity where $n$ is the number of rows in the Rainbow table.
The rcrack program provides two input options for searching--search for a single hash or for a list of hashes contained in a file. Our first approach for parallelization rcrack involved searching the file of hashes. This design involved dividing the number of lines in the file so each process reads an equal number of lines and searches for those values. This process, while parallel, was not too beneficial as each process need to read multiple tables and in the worse case scenario the match could be located in the last table.

In a second approach we decided to pass the value of the hash (or the list of hashes) to each process and then divide the tables between processes. Each process is searching for all values in its own set of tables, and then information from all processes is collected and displayed.

### 6 Findings

The result of this research is presented in two forms. Table 1 shows various tests (table creation vs. cracking time) comparing RainbowCrack run on a single processor machine against a SHARCNET supercomputer. Figure 2 shows a graphical comparison of the speed difference in execution the MPI parallel implementation offers over the standard sequential version of RainbowCrack.
For table creation, our MPI implementation performed nearly 400 times faster, reducing a problem that could take 6 years to perform to a problem space that requires only a few days to complete. For various technical and resource limitation reasons the majority of our research focused on using 32 processors. We were able to complete the entire process of table creation for alpha-numeric-all-characters within 6 days by running multiple sets of our MPI rtgen for 32 processors on the whale SHARCNET system (3072 processors).

We also achieved substantial gains in the cracking time for a list of hashes. Using our parallel implementation of rcrack, we can crack a list of 10 hashes 4 times faster than using the standard rcrack version. Table 1 summarizes the various timings of our MPI RainbowCrack implementation.

Table 1. Results of test runs of RainbowCrack on an AMD athlon 2.0 GHz desktop machine and our parallel implementation of RainbowCrack on a SHARCNET supercomputer using 16, 32, 64, and 90 processors.

<table>
<thead>
<tr>
<th>Table time creation</th>
<th>Single Processor</th>
<th>MPI</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Table creation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Number of processors</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>2 days 18 hours</td>
<td>3-30 sec</td>
<td>01:07:05</td>
</tr>
<tr>
<td>alpha</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15 days 17 hours</td>
<td>6-65 sec</td>
<td>06:38:30</td>
</tr>
<tr>
<td>alpha-numeric</td>
<td></td>
<td></td>
</tr>
<tr>
<td>224 days (7 months)</td>
<td>28-287 sec</td>
<td>92.38:30</td>
</tr>
<tr>
<td>alpha-numeric-all</td>
<td></td>
<td></td>
</tr>
<tr>
<td>characters</td>
<td></td>
<td></td>
</tr>
<tr>
<td>alpha-numeric-all</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5 depicts a graphical view of the comparison between the sequential RainbowCrack implementation and our parallel implementation for table generation.

---

2 Timings for Single Processor were extracted from the documentation of the source files (i.e., RainbowCrack from [5]). Timing for our MPI multiprocessor implementation were calculated from output log files generated on SHARCNET supercomputers.
Fig. 5. Results of tests runs of RainbowCrack on an AMD athlon 2.0 GHz desktop machine and our parallel implementation of RainbowCrack on a SHARCNET supercomputer using 16, 32, 64, and 90 processors.

7 Discussion

This section discusses some of the problems and issues we experienced as we explored parallelization of RainbowCrack. Two areas in particular are presented here relating to the precomputation (rainbow table construction) and the cracking implementations.

7.1 RainbowCrack – Table generation

One area where we had difficulty was the MPI_File_write(). We were able to get each process to write to a shared file with MPI_File_write() but during our testing we found the performance to be very poor. The following section of code illustrates how one file is being manipulated by several participating processes [6, 7].

```c
MPI_File fh;
MPI_File_open(MPI_COMM_WORLD, szFileName,
MPI_MODE_CREATE | MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, rank * 16 * chainsPerProc, MPI_INT,
```
Initially we suspected that there was a problem with MPI_File_write() library function. So, we decided to use fwrite() instead by modifying our implementation so each process is writing to its own file. This resulted in good timings. We are not sure why performance degradations occurred in the case of multiple processors writing to a single file. We suspect that the issue is with implementation of the distributed file system while communicating with multiple processes.

7.2 RainbowCrack – Searching

Our parallel implementation of rcrack has a restriction—one or more rainbow tables are assigned to one process. It is not that important in case of large data sets, but it reduces the benefits of parallel implementation in a case of small set of tables (i.e., 5). Searching is a part in which we can envision significant gains especially in the case of large file sets; rewriting the major parts of the code could be beneficial [8]. With a few easy steps we can improve performance (for example by doubling the number of processors). With deeper changes in the code we expect to see significant improvements.

8 Conclusion

The purpose of this research was to i) reduce the precomputation time (i.e., minimize the time required to construct the rainbow tables) by using MPI (Message Passing Interface) parallel algorithms; ii) to reduce the time to crack a given hash by using MPI and parallel programming optimization on a supercomputer; and iii) perform a series of tests to determine the degree of success.

We believe we have accomplished modest gains in performance, particularly in the parallelization of table creation of RainbowCrack. Since generation by far takes the most amount of time we are ecstatic to report significant time improvements. What that takes single core machines years to do can be performed in a matter of days using our parallel implementation.

9 Future Research

We have accomplished some of our research goals but there are many other areas of improvement. The discussion section revealed some areas for improvement. Future
research will explore the scalability to a larger number of processors on SHARCNET; improve the parallel precomputation algorithm; and explore ways to improve the parallel cracking algorithm. In the upcoming months we plan on continuing and refining our parallel implementations and conducting more tests using SHARCNET.

Acknowledgements

This work was made possible by the facilities of the Shared Hierarchical Academic Research Computing Network (SHARCNET:www.sharcnet.ca) and the contributions of my students: Alan Bell, Jeffrey Bates, Jesse Gitonga, and Kyle Williams. Special thanks to Zhu Shuanglei who graciously provided the serial version of RainbowCrack from which we conducted this parallel implementation and research.

References

Appendix

This section describes some of the relevant parts of output files to illustrate the table creation process.

This output shows typical time each process needs to create multiple files on saw.sharcnet.ca (new system in SHARCNET). Processor rank is the number on the left margin. Each processor is responsible for the construction of a chunk of a rainbow table consisting of 250,000 entries (i.e., [startpoint, endpoint] rows) of the total 8,000,000 required for this entire table.

generating...
4: 250000 of 8000000 rainbow chains generated (4 m 20 s)
23: 250000 of 8000000 rainbow chains generated (4 m 21 s)
20: 250000 of 8000000 rainbow chains generated (4 m 21 s)
10: 250000 of 8000000 rainbow chains generated (4 m 21 s)
1: 250000 of 8000000 rainbow chains generated (4 m 21 s)
15: 250000 of 8000000 rainbow chains generated (4 m 21 s)
19: 250000 of 8000000 rainbow chains generated (4 m 20 s)
22: 250000 of 8000000 rainbow chains generated (4 m 21 s)
21: 250000 of 8000000 rainbow chains generated (4 m 21 s)
0: 250000 of 8000000 rainbow chains generated (4 m 21 s)
24: 250000 of 8000000 rainbow chains generated (4 m 21 s)
14: 250000 of 8000000 rainbow chains generated (4 m 21 s)
12: 250000 of 8000000 rainbow chains generated (4 m 21 s)
26: 250000 of 8000000 rainbow chains generated (4 m 21 s)
2: 250000 of 8000000 rainbow chains generated (4 m 21 s)
31: 250000 of 8000000 rainbow chains generated (4 m 21 s)
17: 250000 of 8000000 rainbow chains generated (4 m 21 s)
25: 250000 of 8000000 rainbow chains generated (4 m 21 s)
29: 250000 of 8000000 rainbow chains generated (4 m 21 s)
9: 250000 of 8000000 rainbow chains generated (4 m 21 s)
5: 250000 of 8000000 rainbow chains generated (4 m 21 s)
7: 250000 of 8000000 rainbow chains generated (4 m 21 s)
30: 250000 of 8000000 rainbow chains generated (4 m 21 s)
27: 250000 of 8000000 rainbow chains generated (4 m 21 s)
3: 250000 of 8000000 rainbow chains generated (4 m 21 s)
6: 250000 of 8000000 rainbow chains generated (4 m 21 s)
28: 250000 of 8000000 rainbow chains generated (4 m 21 s)
13: 250000 of 8000000 rainbow chains generated (4 m 21 s)
16: 250000 of 8000000 rainbow chains generated (4 m 21 s)
18: 250000 of 8000000 rainbow chains generated (4 m 22 s)
11: 250000 of 8000000 rainbow chains generated (4 m 22 s)
8: 250000 of 8000000 rainbow chains generated (4 m 25 s)