10-1-2008

Distributed pre-computation for a cryptanalytic time-memory trade-off

Michael S. Taber

Distributed Pre-computation for a

Cryptanalytic Time-Memory Trade-Off

By

Michael S. Taber

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Engineering

Supervised by

Dr. Muhammad Shaaban
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, NY
October 2008

Approved By:

Dr. Muhammad Shaaban
Primary Advisor – R.I.T. Dept. of Computer Engineering

Dr. Roy Czernikowski
Secondary Advisor – R.I.T. Dept. of Computer Engineering

Dr. Roy Melton
Secondary Advisor – R.I.T. Dept. of Computer Engineering

Thesis Release Permission Form

Rochester Institute of Technology
Kate Gleason College of Engineering

Title: Distributed Pre-computation for a Cryptanalytic Time-Memory Trade-Off

I, Michael S. Taber, hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or part.

_____
Michael S. Taber

_____
Date

Dedication

To my wife and children.

Acknowledgements

I would like to thank Dr. Shaaban for his guidance in the direction of my research, for reviewing multiple drafts of my thesis, and for accepting my request to head up my thesis committee. I would like to thank Dr. Czernikowski for his continued support of my education, dating back to the freshman year of my undergraduate degree. I also wish to thank Dr. Melton for various conference calls to ensure my research was progressing as planned and for providing additional assurances that I was on the right track. Additionally, I would like to express my gratitude to the entire committee for their support over the past six months. I can't thank you enough for your support and assistance.

Thanks to the Computer Engineering department, including Dr. Savakis and Pam Steinkirchner for their help in addressing various RIT requirements and helping sort out what needed to be done to certify my Masters degree requirements.

I would also like to express my thanks to Dr. Matthew MacLean for reviewing the first draft of my thesis and for his insight into the thesis defense process. Thank you for the late night conversations and your various bits of advice.

Next, I wish to thank all of my family and friends who have supported me in this endeavor. They are too numerous to list individually, but have my thanks, nonetheless. Finally, I wish to thank the brotherhood of Phi Delta Theta for rounding out my non-technical education throughout my college career and beyond.

Abstract

Cryptanalytic tables often play a critical role in decryption efforts for ciphers where the key is not known. Using a cryptanalytic table allows a time-memory tradeoff attack in which disk space or physical memory is traded for a shorter decryption time.

For any N key cryptosystem, potential keys are generated and stored in a lookup table, thus reducing the time it takes to perform cryptanalysis of future keys and the space required to store them. The success rate of these lookup tables varies with the size of the key space, but can be calculated based on the number of keys and the length of the chains used within the table.

The up-front cost of generating the tables is typically ignored when calculating cryptanalysis time, as the work is assumed to have already been performed. As computers move from 32 bit to 64 bit architectures and as key lengths increase, the time it takes to pre-compute these tables rises exponentially. In some cases, the pre-computation time can no longer be ignored because it becomes infeasible to pre-compute the tables due to the sheer size of the key space.

This thesis focuses on parallel techniques for generating pre-computed cryptanalytic tables in a heterogeneous environment and presents a working parallel application that makes use of the Message Passing Interface (MPI). The parallel implementation is designed to divide the workload for pre-computing a single table across multiple heterogeneous nodes with minimal overhead incurred from message passing. The result is an increase in pre-computational speed that is close to that which can be achieved by adding the computational ability of all processors together.

Table of Contents

Thesis Release Permission Form ii

Dedication iii

Acknowledgements iv

Abstract v

List of Charts x

List of Figures xi

List of Tables xii

Glossary xiii

Chapter 1 Introduction 1

Chapter 2 Previous Works 5

Chapter 3 Developments leading to Rainbow Tables 7

3.1. Martin Hellman's Original Method 8

3.2. Ronald Rivest's use of Distinguished Points 15

3.3. Philippe Oechslin's Improved Method 16

3.4. Rainbow Table Example 19

3.4.1 Example Cryptographic Algorithm 19

3.4.2 The Reduction Function 20

3.4.3 Rainbow Table Parameters 21

3.4.4 Building the Rainbow Table 22

3.4.5 Using a Rainbow Table for Decryption 27

3.5. Summary 28

Chapter 4 Parallel analysis and design 30

4.1. Introduction to Parallel Computing 30

4.2. Parallel Granularity 33

4.3. Granularity of Rainbow Table Generation Tasks 35

4.4. Disk I/O Considerations 37

4.5. Differences in Processor Hashing Speeds 40

4.6. Network Topology 43

4.7. Final Architecture 46

4.7.1 Work assignment using a master and slave architecture 46
4.7.2 Determining the ideal master node 47
4.7.3 Dividing the problem into tasks 48
4.7.4 Task Assignment 53
4.7.5 Network Considerations 54

4.8. Parallel Analysis & Design Summary 55

Chapter 5 Implementation 58

5.1. MPI Setup Parameters 59

5.2. PRTGen parameters 62

5.2.1 hash_algorithm 62
5.2.2 charset 63
5.2.3 minlen 64
5.2.4 maxlen 64
5.2.5 table_index 64
5.2.6 chain_len 65
5.2.7 chain_count 65
5.2.8 file_suffix 66
5.2.9 timeslice 66
5.2.10 bench 67

5.3. PRTGen Communications Architecture 68

5.4. Error Handling 72

5.5. Platform Specifics 74

5.6. Conclusion 75

Chapter 6 Results and Analysis 76

6.1. Reference Nodes 76

6.2. Test Scenario 1 79

6.3. Test Scenario 2 81

6.4. Test Scenario 3 84

6.5. Test Scenario 4 88

6.6. Non-Parallel vs. Parallel Scenarios 93

6.7. Conclusion 99

Chapter 7 Conclusions 100

7.1. Features of PRTGen 100

7.2. Contributions to the Field 101

7.3. Areas for Future Work 101

7.4. Closing Remarks 104

Bibliography 106

Appendix A 108

Appendix B 114

Appendix C: PRTGen Source Code 120

Benchmark.h file contents 120

Benchmark.cpp file contents 121

ChainWalkContext.h file contents 122

ChainWalkContext.cpp file contents 124

HashAlgorithm.h file contents 135

HashAlgorithm.cpp file contents 136

HashRoutine.h file contents 138

HashRoutine.cpp file contents 139

Public.h file contents 141

Public.cpp file contents 142

RainbowTableGenerate.cpp file contents 147

List of Charts

Chart 1: Bytes/second vs. Chain Length 39

Chart 2: Actual vs. Estimated Time of different time slices in Scenario 3 85

Chart 3: Total Idle time of processes in Scenario 3 86

Chart 4: Total waiting time of processes in Scenario 3 86

Chart 5: Time not spent working for processes in Scenario 3 87

Chart 6: Actual vs. Estimated Time of different time slices in Scenario 4 90

Chart 7: Total Idle time of processes in Scenario 4 91

Chart 8: Total waiting time of processes in Scenario 4 92

Chart 9: Time not spent working for processes in Scenario 4 92

Chart 10: Non-Parallel vs. Parallel application speed 94

Chart 11: Efficiency of the Parallel Implementation 97

List of Figures

Figure 1: Construction of the function f. [10] 10

Figure 2: Matrix of images under f. [10] 11

Figure 3: Generic distributed-memory MIMD system 44

Figure 4: A “star network” 45

List of Tables

Table 1: Plaintext to Ciphertext for Rainbow table example 20
Table 2: Intermediate Values for Sample Rainbow Table 22
Table 3: Value of the Least Significant Cipher Text Character 23
Table 4: Value of the Most Significant Cipher Text Character 24
Table 5: Starting Point through Chain Position X_2 25
Table 6: Chain Position X_2 through Chain Position X_5 26
Table 7: Chain Position X_5 through Chain Position X_8 26
Table 8: Chain Position X_8 through Ending Point 26
Table 9: Final Example Rainbow Table 27
Table 10: Sample test machine hashing speeds 38
Table 11: Processor Comparison 41
Table 12: Hypothetical hashing speeds 49
Table 13: Master/Slave workflow 71
Table 14: Reference hardware specifications 77
Table 15: System benchmarks 78
Table 16: Application Parameters for Scenario 1 80
Table 17: Scenario 1 speed test results 80
Table 18: Application Parameters for Scenario 2 82
Table 19: Scenario 2 speed test results 82
Table 20: Scenario 3 speed test results 84
Table 21: Scenario 4 speed test results 89
Table 22: Scenario 3 vs. Scenario 4 speed comparison 96

Table A1: Scenario 3 Results for Time Slice = 1 108
Table A2: Scenario 3 Results for Time Slice = 5 109
Table A3: Scenario 3 Results for Time Slice = 10 110
Table A4: Scenario 3 Results for Time Slice = 15 111
Table A5: Scenario 3 Results for Time Slice = 30 112
Table A6: Scenario 3 Results for Time Slice = 60 113

Table B1: Scenario 4 Results for Time Slice = 1 114
Table B2: Scenario 4 Results for Time Slice = 5 115
Table B3: Scenario 4 Results for Time Slice = 10 116
Table B4: Scenario 4 Results for Time Slice = 15 117
Table B5: Scenario 4 Results for Time Slice = 30 118
Table B6: Scenario 4 Results for Time Slice = 60 119

Glossary

brute force attack

A method for decrypting encrypted information in which a large number of keys are tried, in an attempt to find the unencrypted message.

chain

When a set of plaintext is encrypted, a reduction function is applied, and the process is repeated, the result is a chain. Typically, only the starting point and ending point are stored.

cipher text

Cipher text is the encrypted form of a set of data.

collision

A collision occurs during table computation when values in two different chains are reduced to the same value. The reduction function results in a mapping of a larger set of values into a smaller set. The more mappings there are, the higher the probability of a collision.

cryptanalysis

The study of methods for obtaining encrypted information without knowing the secret keys needed to decrypt that information.

DES

DES is the Data Encryption Standard. It is a 56 bit cipher developed in the early 1970's and selected as the official Federal Information Processing Standard for the United States in 1976.

distinguished point

A data point for which a set of criteria must hold true

Distributed-memory MIMD system

A system in which each processor has its own memory that is considered separate from the others and connected by an arbitrary network

efficiency

An approximation of the amount of time spent actually doing work versus the amount of time that could be spent doing work.

EP

End point of a chain.

false alarm

A situation that arises when an endpoint in a chain matches the output of a reduction function, but the key found in the previous column of the matrix does not decrypt the cipher text.

granularity

Granularity is a reference to a task size. It takes into account the ratio of computation to communication.

idle time

Idle time is defined as the duration during which work is still being assigned to slave nodes, but in the case of a particular slave, no work is being done. This is typically a result of communication overhead between the master and slave. The slave is considered to be idle while it is requesting additional work.

merge

A merge typically occurs after a collision due to the fact that from that point on, two points have the same value and are using the same reduction functions to generate the rest of the chain.

MPI

Message Passing Interface

MPIFL

Fault Tolerant Message Passing Interface Farm Library

OpenSSL

OpenSSL is an open source toolkit that implements the SSL protocol.

plaintext

A string which has either not yet been encrypted or is the human readable string which has been decrypted.

PRTGen, or PRTGen.exe

PRTGen is the parallel application that is the implementation of this thesis. It stands for Parallel Rainbow Table Generator.

rainbow table

A rainbow table is another name for a pre-computed cryptanalysis table that is based on Oechslin's work.

RTGen or RTGen.exe

This is the reference application, which is built to run on a single node and does not use MPI. It stands for Rainbow Table Generator.

SP

Starting point of a chain

time slice

A user specified period of time which helps determine task size.

waiting time

During the process of generating chains in a parallel environment, this refers to the time period during which the slave node has completed its work unit, but cannot be assigned another because no more tasks are available. The time spent waiting for the other nodes to complete their work is referred to as the waiting time.

work slice

A discrete unit of work, otherwise defined as a task. A work slice is calculated using the hashing speed of the slowest computer in the cluster, the user specified time slice, and the chain length.

working time

The working time is the time during which a slave node is generating chains for a rainbow table.

Chapter 1 Introduction

Cryptanalysis is the study of methods for obtaining encrypted information without knowing the secret keys that are normally required to decrypt that information. The algorithms used to encrypt information may be simple or complex, but are judged primarily on how well a brute force attack can be executed against that algorithm. A brute force attack is an attempt to decrypt a message by generating a large number of possible keys until the decrypted message has been determined. If no methods exist for decrypting an algorithm in less time than it takes to use brute force, the algorithm is considered to be reasonably secure.

When the total number of keys is relatively small, it is possible to store every key and the corresponding cipher on disk. When a decryption is required, the plaintext is looked up in this table using the cipher text. As the size of the key space increases, the storage space required also increases, eventually trending towards a point where storing all possible keys on disk is no longer feasible.

One of the first time-memory trade-off techniques for cryptanalysis was published in 1980 by Martin Hellman. In his research, he described a technique for reducing the time required for recovering a key in any N key cryptosystem [1] using pre-computed tables. The number of operations required to generate the lookup tables was approximately equivalent to that of a brute force attack, but the goal of reducing the disk space required to store the tables had been achieved. In addition, once the tables had been generated, it was much faster than a brute force attack. Two years later, Rivest introduced the concept of distinguished points [2] based on Hellmans' technique. A distinguished point is a data point for which a set of criteria must hold true. Rivest proposed that only

distinguished points are stored in memory. For years these algorithms had been studied, but no further improvements had been published.

Finally in 2003, Philippe Oechslin published a technique to improve upon Hellman's original work [3]. Instead of using distinguished points, Oechslin used unique reduction functions in each element of the chains of alternating keys and cipher texts.

The focal point of Oechslin's work was an implementation to attack a Microsoft Windows password hash. Using 1.4GB of data, he demonstrated the ability to crack 99.9% of all alphanumerical password hashes (2^37 hashes) in 13.6 seconds, where it had previously taken 101 seconds using distinguished points via Hellman’s method.

What Oechslin fails to discuss at any length is the pre-computation time required for his experiment, or how long it takes to generate any of these tables. A time-memory trade-off is only feasible if the time to pre-compute the tables can be achieved in a reasonable time frame. A reasonable time frame is subjective, but time is clearly important. Were it not, then there would be no purpose to researching time-memory trade-offs.

Oechslin's research used chain lengths of 4,666, a chain count of 38,223,872 and a single table. This table resulted in a success probability of approximately 77% and was proven to be more efficient and successful than the original Hellman tables. On a 2.6GHz Athlon processor with 3GB of RAM, this table could be generated in approximately 20 hours.

To achieve a success rate of 99.9%, Oechslin used 5 tables, which can be generated in approximately 4 days. While seemingly reasonable, this experiment does not provide the whole story.

The character set used by Oechslin in his experiment was the alpha numeric set consisting of uppercase letters and numbers only. This excludes all lowercase characters and 33 special characters. Oechslin's experiment only took into account 36 possible characters, but a standard Windows password has the potential to contain 95 different characters, assuming we are not using Unicode characters.

To achieve a 99.9% success rate using 95 possible characters, using a chain length of 4,666 and a chain count of 38,223,872 would require 2,736 tables. The additional tables are required to maintain the success rate while increasing the number of characters in the keys as will be explained in Chapter 3. Increasing the number of characters in the key from 36 to 95 increases the number of possible keys from approximately 2^37 to 2^46.

This would require more than 1.5 TB of disk space. Single hard drives are available today which can hold 1TB each. However the ultimate problem is the time it would take to generate these rainbow tables. Simple benchmarking indicates that it would require approximately 7.6 years on a 2.6GHz Athlon XP processor to generate these tables.

This pre-computation time is arguably no longer feasible or realistic. While it is technically possible to generate all of these tables in less than a lifetime, the fact remains that 7 years to pre-compute the tables is not generally acceptable.

The main motivation behind this research is to examine parallel methods which will reduce the time required to generate rainbow tables in a heterogeneous environment. That is to say, that the goal is to implement a system that allows for the use of processors of different speeds, rather than a cluster of computers which share the same hardware

characteristics, such as CPU speed, RAM, disk size, and operating system. As explained in Chapter 4.7.3, simply dividing the tasks up among the processors in the system according to their relative speeds does not work in practice due to events that may occur on the computers that are outside the scope or control of the parallel program.

This paper is organized in the following manner. Chapter 2 provides an overview of previous works that are related to using rainbow tables and the parallel techniques that are implemented in this work. Chapter 3 concentrates on explaining in detail the mathematics behind the algorithms developed by Hellman, Rivest and Oechslin leading up to this work. In Chapter 4, Chapter 5, and Chapter 6 the fundamentals of this research are detailed.

Chapter 4 discusses parallel techniques in general and how they may be applied to the problem of generating rainbow tables in parallel. Chapter 5 details the MPI application that was created to demonstrate an implementation of this research. It includes all command line parameters and descriptions of what each of them is used for. Then in Chapter 6, four different scenarios are examined using the parallel implementation and compared to one another to determine the efficiency of the parallel implementation.

Chapter 7 summarizes the results of the implementation and describes areas for further research. Appendix A contains the data tables for the results of Scenario 3 that is discussed in Chapter 6. Appendix B contains the data tables for the results of Scenario 4 that is discussed in Chapter 6. Finally, Appendix C contains a source code listing for the MPI application that was developed to demonstrate this research.

Chapter 2 Previous Works

In 1999, Quisquater and Desmedt proposed a massively-parallel hardware approach to attack DES or similar ciphers [4]. The basic concept was that if someone were to embed specialized decryption hardware into consumer electronics, thus enabling them to perform decryption on units of work for part of a larger problem, the decryption would take place in a massively parallel manner and a solution would be found very quickly. The hardware unit that was determined to have cracked the code would be declared the “winner”. The suggestion of deploying this in China came about due to the large population, hence this proposal was dubbed the "Chinese Lottery".

Quisquater and Desmedt suggested that a massively-parallel software approach could be feasible, but did not explore this any further [4]. RFC 3607 [5] illustrates the potential for a massively-parallel software approach to decryption. In this RFC, an example is provided that assumes approximately 500,000 hosts connected to the internet could be infected with a specific form of an internet worm or virus that is designed to aid in distributed cracking attempts.

With this assumption and an estimated aggregate performance of 9.79e+11/sec, an 8 character MD5 password could be cracked by brute force in 4.79 minutes. A 64-bit MD5 key could be cracked in 218 days. Neither of these time periods is completely unreasonable and each assumes that a complete brute-force attempt is made. Were this to be combined with Oechslin's rainbow table method, the average time to crack a hash key would drop dramatically.

In 2005, Quisquater expanded upon his earlier research by updating the cost estimates associated with parallel hardware decryption [6]. These new cost estimates

showed that a $12,000 machine could break DES encryption in a mere 3 hours. In contrast, a hardware implementation from 1998 cost nearly $200,000.

He pointed to Oechslin's research as a viable optimization of Hellman's research and a clear path to reducing the number of lookups required for resolving a cryptographic attack. His new research indicated that it would be financially feasible to implement a massively-parallel hardware attack against DES using FPGA's. This was an expansion of his previous work [20] where he merely proved that it was possible to do, rather than financially feasible.

This thesis consists of original research for generating rainbow tables in parallel in a heterogeneous environment on dissimilar hardware. There are no publications since 2003 that explore the use of parallel programming for the generation of Oechslin’s rainbow tables, nor has a parallel software implementation in a heterogeneous environment using commodity hardware been examined.

The implementation in this thesis shows that using a small cluster of dissimilar hardware, it is possible to increase the hashing speed in a linear fashion by adding more processors to the task with a performance overhead cost of less than 2% overall. The average hashing speeds of each processor may be added together minus the 2% performance overhead to provide an estimate of the total hashing speed of the system. If the number of processors dedicated to the task is doubled, the time to complete the rainbow tables is approximately halved. It would not otherwise be feasible to generate rainbow tables using all 95 ASCII characters for decrypting Windows passwords.

Chapter 3 Developments leading to Rainbow Tables

When referring to cryptanalysis, a time-memory trade-off is a method of trading the time to decipher an encrypted message for memory. In this sense, memory can refer to either physical memory or to disk space, however in the case of rainbow tables it is more common to refer to disk space. Memory or time can either be increased or decreased, but improving one will adversely affect the other, hence the trade-off. To achieve decreased memory requirements takes more time to decrypt a cryptogram. Decreasing the time to decrypt a message requires more disk space. To improve both of these resource requirements would be an algorithmic improvement, rather than a trade-off.

Rainbow tables trade an increased decryption time to achieve an exponentially lower disk space requirement. The key difference is that the reduced disk space is exponentially lower. It is not enough to simply trade one resource for another. Examine the following example of the md5 hash. Assume that the intent is to store all character combinations which are 1-8 characters in length using a set of all lowercase letters and numbers. This provides a plaintext space of 36 characters. The corresponding 128 bit md5 checksum for each combination will also be stored as a 32 character string.

This results in 2,901,713,047,668 potential combinations of “passwords”, or approximately 2.9 trillion combinations. To store only the potential password combinations requires approximately 21.5 TB of disk space. Storing the md5 “passwords” would require an additional 84.4 TB of disk space. While not impossible, this is clearly a difficult requirement.

Next, examine the situation if the plaintext character set were expanded from 36 characters to 95 characters, which include all upper and lowercase characters, all numbers, and virtually every symbol on a standard keyboard. Again, assume that the intent is to store all character combinations which are 1-8 characters in length. The number of possible "password" combinations climbs from 2.9 trillion to 6.7 quadrillion. This is more than 2,000 times greater. The required storage space climbs accordingly to nearly 50,000 TB required to store the plaintext passwords, plus another 200,000 TB to store the md5 hashes.

It is clear from these disk requirements that it is not feasible to store this information without some sort of compromise. This chapter describes several of the methods that have been proposed to address this problem in the past. Chapter 3.1 focuses on the original trade-off algorithm introduced by Martin Hellman in 1980. The improvements suggested by Ronald Rivest in 1982 are detailed in Chapter 3.2. In Chapter 3.3, the improvements discovered by Philippe Oechslin are discussed, followed by a working example of Oechslin's algorithm in Chapter 3.4. Finally, a summary of the advantages and drawbacks is provided in Chapter 3.5.

3.1. Martin Hellman's Original Method

The original time-memory trade-off [1] that was proposed in 1980 by Martin Hellman was a significant breakthrough in cryptanalysis. Previously, there had not been any generalized time-memory trade-off algorithms published.

Allowing C to be an arbitrary cipher text, P the corresponding plaintext, and S_k to be the enciphering operation using the key k, we have the following generic equation:

$$C = S_k(P)$$

Given an arbitrary cipher text C_0, there exists a key and a plaintext P_0 such that the following holds true:

$$C_0 = S_k(P_0)$$

The key itself within a particular cryptography algorithm does not change, thus for every plaintext there exists one and only one cipher text. It is possible to have a cipher text that can be created by multiple plaintexts but this has no impact on the algorithm or its accuracy.

Applying a reduction function R to this encryption gives us the following equation, also demonstrated in Figure 1:

$$f(K) = R[\ S_k(P_0)\ ]$$

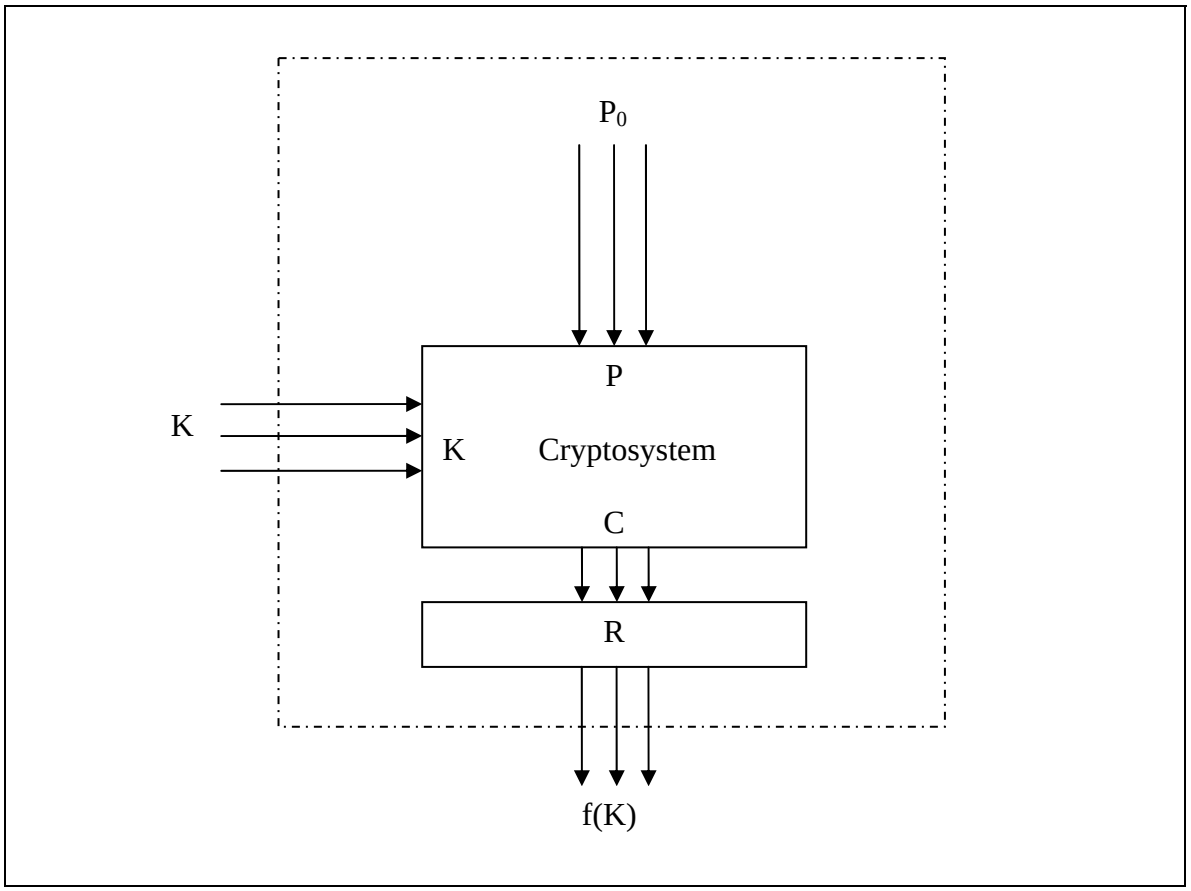


Figure 1: Construction of the function f. [10]

The reduction function is an arbitrary function that reduces the size and complexity of the cipher text. A simplistic example of a valid reduction function would be a case where the resulting cipher text were a 64 bit number and our reduction function were to simply truncate it to 32 bits. The calculation of f(K) is a simple, one way function. However, calculating the key K when f(K) is known is essentially the same as performing a cryptanalysis. The function f is demonstrably a one way function [7] and the time-memory tradeoff may be applied to any one way function [1].

Using *m* points randomly chosen from the key space N, and an arbitrarily chosen chain length of *i*, we use the encryption function S_k to encrypt the plaintext, apply the

reduction function, and map the result back into the key space to obtain a new plaintext. The process is repeated for *i* iterations until the desired chain length has been reached. All intermediate points are discarded to save memory and only the starting points (SP) and ending points (EP) are stored. It follows that for $1 < i < m$:

$$X_{i0} = SP_i$$

and that

$$X_{ij} = f(X_{i,j-1}) \text{ for } 1 < j < t$$

results in a matrix of operations shown by Figure 2. Hellman refers to this as a "Matrix of images under *f*."

$$SP_1 = X_{10} \xrightarrow{f} X_{11} \xrightarrow{f} X_{12} \xrightarrow{f} \dots \xrightarrow{f} X_{1t} = EP_1$$

$$SP_2 = X_{20} \xrightarrow{f} X_{21} \xrightarrow{f} X_{22} \xrightarrow{f} \dots \xrightarrow{f} X_{2t} = EP_2$$

$$\dots$$

$$SP_m = X_{m0} \xrightarrow{f} X_{m1} \xrightarrow{f} X_{m2} \xrightarrow{f} \dots \xrightarrow{f} X_{mt} = EP_m$$

Figure 2: Matrix of images under f. [10]

After *m* starting points and ending points have been calculated and stored, a plaintext P_0 is encrypted and the cipher text C_0 is made known or discovered by the cryptographer.

$$C_0 = S_k(P_0)$$

Applying the reduction function R, Y1 can be calculated as follows:

$$Y_1 = R(C_0) = f(K)$$

Should $Y_1 = EP_i$, it follows that the key can be found at $X_{i,t-1}$ or EP_i has more than one inverse, a case which is referred to as a false alarm. If $Y_1 = EP_i$, then the cryptographer must compute $X_{i,t-1}$. This is done by starting at SP_i and computing $X_{i,1}$, $X_{i,2}$, $X_{i,3}$, etc until $X_{i,t-1}$ is reached. This is required because all of the intermediate columns from Figure 1 had been previously discarded during the pre-calculation to save memory.

It is quite possible that Y_1 does not match any of the endpoints. Should this be the case, then Y_2 must be calculated, as follows:

$$Y_2 = f(Y_1)$$

The process is repeated and each endpoint is tested to see if it matches Y_2. If it does, then $X_{i,t-2}$ is calculated in the manner described above to find the key. Should no endpoints match, the iteration process starts again until Y_i has been reached. If Y_i is reached and no valid matches to an end point have been found, then the key to decrypt the plaintext is not in the table.

The performance gains inherent in this algorithm are such that the probability of success is $P(S) = \frac{mt}{N}$, assuming that no elements in any of the columns of Figure 1 overlap with any other element. The probability of success of an exhaustive search with *t* operations results in $P(S) = \frac{t}{N}$. A table lookup with m elements in memory results in $P(S) = \frac{m}{N}$.

If the columns in Figure 1 do have overlap, there is a reduction in the success probability that is directly proportional to the number of overlapping elements. Thus, some overlap is tolerable due to the gains that can be achieved. The actual probability of success can be calculated as follows:

$$P(S) = Pr \sum_{i=1}^{m} \sum_{j=0}^{t-1} I \frac{\{ X_{ij} \; is \; new \}}{N}$$

Using $Pr(X_{ij}$ is new), where being "new" means that it has not occurred in a previous row, or thus far in its row:

$$1 \geq \Pr\left(X_{i0}, X_{i1}, X_{i2}, \ldots, X_{ij} \; are \; all \; new\right)$$

$$= \Pr(X_{i0} \; is \; new) \Pr(X_{i1} \; is \; new \mid X_{i0} \; is \; new) \ldots \Pr\left(X_{ij} \; is \; new \mid X_{i0}, X_{i1}, \ldots, X_{i.j-1} \; are \; new\right)$$

This is in essence, a conditional probability equation, where we are trying to verify the probability of A, given that B has occurred [8].

If we assume that every element in each chain is never the same as any other element in any other chain (ie: all elements are unique), then we have a maximum probability that the chains will produce a successful hit, and that probability is bound by this equation:

$$\Pr\left(X_{ij} is \; new\right) \geq \left[\frac{N - it}{N}\right]^{j+1}$$

As there are at most *t* elements in each row. The final probability equation is as follows:

$$P(S) \geq \frac{1}{N} \sum_{i=1}^{m} \sum_{j=0}^{t-1} \left[\frac{N - it}{N}\right]^{j+1}$$

Or

$$P(S) \geq \frac{1}{N} \sum_{i=1}^{m} \sum_{j=0}^{t-1} \left[1 - \frac{it}{N}\right]^{j+1}$$

Hellman realized that with a fixed value of N, there is little to be gained in the success rate by increasing m or t beyond a certain point. That point occurs when $mt^2 = N$.

Hellman also realized that the downfall of this algorithm was that P(S) = $\frac{mt}{N}$, assumed that no intermediate elements overlapped with one another.

With a limited key space governed by the reduction function, the more intermediate elements that exist, the higher the probability for overlap between any two given elements produced by the reduction function. This situation is also known as a collision and is highly similar to the "birthday problem" [9], which states that with a limited space of elements, the more random elements that are chosen, the higher the probability of two elements being identical to one another. As the table size increases, the efficiency of the table decreases. Hellman recognized this problem and proposed that using multiple tables with different reduction functions was an effective method of addressing the potential for collisions.

The probability of success of multiple tables is given by the following equation, with *l* representing the number of tables.

$$P(S) \geq 1 - (1 - \frac{1}{N} \sum_{i=1}^{m} \sum_{j=0}^{t-1} \left[1 - \frac{it}{N}\right]^{j+1})^{l}$$

This is a simple probability equation derived from the above probability of success of a single table. When a different reduction function is applied to each table, some collisions are still likely. A merge will not occur because the reduction function is different for each table.

3.2. Ronald Rivest's use of Distinguished Points

In 1982, Rivest introduced the concept of distinguished points [2] which addressed the collision problem more effectively. His rationale was that disk access accounted for the majority of the time spent performing the decryption and that the use of distinguished points would reduce the number of disk accesses to approximately $\sqrt{T}$.

A distinguished point is a data point for which a set of criteria must hold true. An example of defining a distinguished point might be stating that the first 10 bits of a key must be a specific binary value, such as all zeros. Rivest proposed that only distinguished points are stored in memory as the endpoints. To decrypt a cipher text, simply generate chains according to Hellman's method until a distinguished point is found. Once a distinguished point has been found, look it up in the table. This greatly speeds up the performance of the algorithm, assuming that it is trivial in terms of time and complexity to calculate whether a distinguished point has been found.

The use of distinguished points has been extensively analyzed since it was introduced and the majority of research in this field between 1982 and 2003 is based on the use of distinguished points. Adjusting the table parameters properly can result in lower memory consumption, a higher probability of success, or faster decryption time, but always results in a trade-off between them. This has been demonstrated in research done by Koji Kusuda and Tsutomu Matsumoto[11] who specifically examined how to achieve a higher success probability.

It was also studied by Johan Borst, Bart Preneel, and Joos Vandewalle[12] who introduced a hybrid approach for distributed key searches. The research of Borst, Preneel and Vandewalle concentrated on the assumption made by Hellman that memory accesses

were negligible. They demonstrated that this assumption was no longer valid when performing a distributed key search. They also introduced a trade-off method that reduced the number of memory accesses by a large factor, thus reducing the problems associated with a distributed key search. However, their research was performed in 1998 and made use of distinguished points. This unfortunately means that none of their work is relevant to Oechslin's work, nor is the distributed key search relevant to the actual generation of the tables.

3.3. Philippe Oechslin's Improved Method

Oechslin's rainbow tables are a relatively simple modification of the original methods introduced by Hellman, but with better results. The fundamental difference that Oechslin makes is to use a different reduction function for each column in a chain, rather than the same reduction function for all of the chains. The net result is that collisions may still occur between chains, but unless they occur in the same column, the chains will not merge, thus increasing the probability of success, and decreasing the number of chains that must be thrown out and recalculated due to chains that merge. Chains are assumed to be of length t, resulting in reduction functions 1 through t-1.

In a chain of length t, the probability of a collision remains the same as in Hellman's method. However, for any arbitrary collision, the probability of a chain also being a merge is $\frac{1}{t}$. This is far less than the 100% chance of a collision being a merge in Hellman's method. The effect of this is that using Oechslin's method, chain lengths can be dramatically longer, and more chains can be used in a single table rather than using separate tables to achieve the same probability of success.

The probability of success of Oechslin's method is as follows:

$$P_{table} = 1 - \prod_{i=1}^{t}\left(1 - \frac{m_i}{N}\right)$$

where $m_1 = m$ and $m_{n+1} = N\ (1 - e^{-\frac{m_n}{N}})$

Oechslin's method offers several notable improvements over Hellman's method. The longer chain length means that a greater number of chains can be put into a single table. This is a direct result of the use of what Oechslin refers to as a "successive reduction function". In addition, the total number of calculations required to search for a matching key using rainbow tables is roughly half of the classic method.

This claim of half the total number of calculations is disputed by Barkan, Biham and Shamier [13]. They point out that Oechslin ignores the number of bits used to represent the starting and ending points and only considers the actual number of starting and ending points. They contend that by doubling *m* in Hellman's scheme, the same amount of data is stored and *t* is reduced by a factor of 4 in the time-memory tradeoff. Reducing *t* by a factor of 4 outweighs the benefits garnered by Oechslin's method. However, they do acknowledge that they themselves ignore the benefits of Oechslin's method in reducing the number of operations required by identifying false alarms. Oechslin's test cases show measurable improvements in this area, which they declined to quantify.

Another benefit of the rainbow tables is that merges can be easily identified because they will have the same endpoints, just as they would if we were using distinguished points. Any endpoint that matches another can be removed from the table

and a new starting point can be selected, thus generating a replacement chain. This allows for the creation of tables that are guaranteed to be merge-free.

Oechslin identifies two other advantages of rainbow tables over the use of distinguished points. Rainbow chains inherently do not have loops. A loop is a condition under which a reduction function may be applied to a data element to generate a reduction element of X. Further down the same chain, the reduction function is applied to a different number that also results in a reduction element of X. This leads to an infinite loop because a distinguished point has not been discovered in the chain, and the chain will simply repeat itself because the reduction function never changes.

In rainbow table chains, the reduction function is different in every single column, which means that we are guaranteed that there cannot be a loop. The benefit of this is that we never need to attempt to detect the existence of loops, nor do we need to spend time pursuing and rejecting loops. The existence of loops in algorithms using distinguished points also reduces coverage, as the data elements in that chain must be completely discarded. Rainbow table chains do not suffer from this problem.

The second advantage that rainbow tables have over distinguished points is that they are a constant length. When trying to determine whether or not a potential match is a false alarm for a table using distinguished points, the entire chain must be regenerated. However, using a rainbow table, only a subset of the chain must be regenerated. Oechslin also provides statistics for several tests that show that classical tables encounter more false alarms per endpoint found, and require more keys generated to verify whether an endpoint is a false alarm.

3.4. Rainbow Table Example

Philippe Oechslin's rainbow tables are somewhat difficult to understand without an example. For this reason, a working example of how a rainbow table are built and an explanation of how it would operate is provided in this section.

Assume that the plaintext to be encrypted consists only of lowercase alphabetical characters plus the numbers zero through 9, giving a total of 36 potential characters in the plaintext space. The cryptographic algorithm used shall be extremely simplistic, as it is for demonstration purposes only. While this encryption algorithm can be easily cracked at a glance, it is helpful to use a simplistic encryption scheme to make the underlying algorithm that governs creation of rainbow tables easier to understand. It also makes verification easier.

3.4.1 Example Cryptographic Algorithm

The cryptographic algorithm to be used works as follows. To "encode" an arbitrary character, create a character string with a length of 2 consisting of the plaintext character to be encoded in both character positions of the new string. Next, increment the second character by 3 plaintext character positions. Thus, the character 'a' is initially expanded to 'aa', and then the second 'a' is incremented by 3 character positions, translating from an 'a' to a 'd'. This is encoded as 'ad'. Similarly, the character 'b' would be encoded as 'be' and so on.

Incrementing a 'z' will enter into the numeric portion of the plaintext, starting at zero. Thus, a plaintext of 'z' is encoded as 'z2'. For purposes of the encryption algorithm, incrementing a character past the value of '9' will wrap around the plaintext space to start at 'a' again. This can be seen in Table 1 on the following page.

Plaintext	Cipher	Plaintext	Cipher	Plaintext	Cipher	Plaintext	Cipher
a	ad	j	jm	s	sv	1	14
b	be	k	kn	t	tw	2	25
c	cf	l	lo	u	ux	3	36
d	dg	m	mp	v	vy	4	47
e	eh	n	nq	w	wz	5	58
f	fi	o	or	x	x0	6	69
g	gj	p	ps	y	y1	7	7a
h	hk	q	qt	z	z2	8	8b
i	il	r	ru	0	03	9	9c

Table 1: Plaintext to Ciphertext for Rainbow table example

An additional restriction on the example will be that all plaintext strings to be encrypted will be only one character long. The process for creating a rainbow table would be the same with longer strings, but is simplified here for demonstration purposes.

3.4.2 The Reduction Function

The reduction function to be used will also be simplistic to make it easier to understand. Recall that the purpose of the reduction function is to map an enciphered character string onto the set of plaintext. The simplest method for doing this is to translate the encrypted text into a numeric value, and then use the modulo function to find the remainder.

In addition, each position in a chain must use a different reduction function, so this algorithm must be modified based on the position in the chain the reduction function is being applied to. Accomplishing this is very straightforward. Prior to performing the modulo operation, add the chain position to the numeric value of the cipher text. For the first chain position, add 1. For the second chain position, add two, etc.

Translating the cipher text to a numeric value is done by counting the characters as if it were a base 36 numeric value. The character ‘a’ is considered to be at position

zero, while the character ‘9’ is considered to be at position 35. If a different sized plaintext set is used, the numeric base will be different as well.

3.4.3 Rainbow Table Parameters

The first step to building a rainbow table is determining the chain length and the number of starting points to use. This is typically decided upon by selecting a desired success rate. Recall that the probability of success of an arbitrary rainbow table is governed by the following equation.

$$P_{table} = 1 - \prod_{i=1}^{t} \left(1 - \frac{m_i}{N}\right)$$

where $m_1 = m$ and $m_{n+1} = N\ (1 - e^{-\frac{m_n}{N}})$

The number of starting points is represented by m and the chain length is represented by i. Using a number of starting points of m=8 and a chain length of t=11 gives us an approximate probability of success of 91.3%, which should be acceptable for the purposes of an example. These numbers are often obtained through some trial and error. However, there do exist upper bounds on the success rate that can be calculated. The math behind these upper bounds is beyond the scope of this discussion.

The success rate can also be increased using multiple tables. The reduction function for an arbitrary column is also different from one table to another. This can be accomplished by multiplying the table number by the chain length position prior to applying the reduction function. To simplify this example, only one table will be used.

3.4.4 Building the Rainbow Table

After the number of starting points and the chain length has been determined, *m* random starting points must be selected. The starting point must be encrypted, have the reduction function applied, and converted back to plaintext to get each intermediate point. There exist *t* intermediate points. The final intermediate point is commonly referred to as the endpoint. Only the starting point and the endpoint are stored in memory. The data table on the following page provides eight randomly selected starting points, all chain intermediate points, and the ending point for each chain.

SP	**X_1**	**X_2**	**X_3**	**X_4**	**X_5**	**X_6**	**X_7**	**X_8**	**X_9**	**X_{10}**	**EP = X_{11}**
a	e	j	p	w	4	d	n	y	a	n	1
d	h	m	s	z	7	g	q	1	d	q	4
l	p	u	0	7	f	o	y	9	l	y	c
r	v	0	6	d	l	u	4	f	r	4	i
w	0	5	b	i	q	z	9	k	w	9	n
4	8	d	j	q	y	7	h	s	4	h	v
5	9	e	k	r	z	8	i	t	5	i	w
9	d	i	o	v	3	c	m	x	9	m	0

Table 2: Intermediate Values for Sample Rainbow Table

Recall that each plaintext value in the column labeled “SP” has been randomly selected from the plaintext space of a-z and 0-9 with a length of 1. Again, this example is simplified and the plaintext strings could be longer than 1, but for this example, only a single character is being encrypted. To generate the value of the first chain column labeled X_1, the starting point must be encrypted, have the reduction function applied, and then mapped back to the plaintext space.

Table 1 shows that for the starting point of ‘a’, the corresponding cipher text is ‘ad’. Due to the plaintext space size of 36, the cipher text must be translated to a numeric value that is base 36. Starting at the rightmost character in the cipher text, the letter ‘d’ is

found. If the plaintext space is numbered from 0 – 35, the letter ‘a’ would be valued at 0, ‘b’ would be valued at 1, ‘c’ would be valued at 2, the letter ‘d’ would be valued at 3, etc. The values of the least significant Cipher Text character can be seen in Table 3.

Cipher Text	Numeric Value	Cipher Text	Numeric Value	Cipher Text	Numeric Value	Cipher Text	Numeric Value
a	0	j	9	s	18	1	27
b	1	k	10	t	19	2	28
c	2	l	11	u	20	3	29
d	3	m	12	v	21	4	30
e	4	n	13	w	22	5	31
f	5	o	14	x	23	6	32
g	6	p	15	y	24	7	33
h	7	q	16	z	25	8	34
i	8	r	17	0	26	9	35

Table 3: Value of the Least Significant Cipher Text Character

The next most significant character in the cipher text is the letter ‘a’. As seen in Table 4, the value of ‘a’ is also zero when it appears in the most significant position. Adding the values of the least and most significant positions of the cipher text provides us with a numeric equivalent of that cipher text. Thus, the cipher text of ‘ad’ maps to the numeric equivalent value of 4.

Note that a typical cryptographic system would have cipher texts that were more than two characters long. In these cases, the value of each position must be added to get the numeric value of the cipher text. It would also be possible to have plaintext of ‘a’ and ‘aa’, which would be required to correspond to different numeric values. Again, this example is simplified for ease of understanding.

Cipher Text	Numeric Value	Cipher Text	Numeric Value	Cipher Text	Numeric Value	Cipher Text	Numeric Value
a	0	j	324	s	648	1	972
b	36	k	360	t	684	2	1,008
c	72	l	396	u	720	3	1,044
d	108	m	432	v	756	4	1,080
e	144	n	468	w	792	5	1,116
f	180	o	504	x	828	6	1,152
g	216	p	540	y	864	7	1,188
h	252	q	576	z	900	8	1,224
i	288	r	612	0	936	9	1,260

Table 4: Value of the Most Significant Cipher Text Character

Examining the starting point of ‘9’ from Table 1, it is known that the cipher text is ‘9c’. From Table 3 and Table 4, we find that the character ‘c’ for the least significant position is valued at 2 and the character ‘9’ in the most significant position is valued at 1,260. Adding them together, we get 1,262. This is the numeric value of the cipher text of ‘9c’.

Once this numeric value is determined, it must be mapped back onto the plaintext. This mapping is implemented by applying the reduction function. Per section 3.4.2, the reduction function we are using is quite simplistic. To apply the reduction function, add the column position to the numeric value of the cipher text, and then use the modulo function with a divisor of the size of the plaintext space, which in this case is 36. This guarantees that the resulting value will be between 0-35, so as to provide a direct mapping to a new plaintext value. Table 5, Table 6, Table 7, and Table 8 show the starting points, the cipher text, the corresponding cipher text numeric value, the reduction function result, and the chain position for all of the intermediate steps.

SP	Cipher Text	Cipher Text Value	R_0	X_1	Cipher Text	Cipher Text Value	R_1	X_2	Cipher Text	Cipher Text Value	R_2	X_3
a	ad	3	4	**e**	eh	151	9	**j**	jm	336	15	**p**
d	dg	114	7	**h**	hk	262	12	**m**	mp	447	18	**s**
l	lo	410	15	**p**	ps	558	20	**u**	ux	743	26	**0**
r	ru	632	21	**v**	vy	780	26	**0**	03	965	32	**6**
w	wz	817	26	**0**	03	965	31	**5**	58	1,150	1	**b**
4	47	1,113	34	**8**	8b	1,225	3	**d**	dg	114	9	**j**
5	58	1,150	35	**9**	9c	1,262	4	**e**	eh	151	10	**k**
9	9c	1,298	3	**d**	dg	114	8	**i**	il	299	14	**o**

Table 5: Starting Point through Chain Position X_3

Now look at a specific example for generating chains in the rainbow table. For the starting point ‘a’, the cipher text value is ‘ad’, and the numeric value for that cipher text is 3. Add 1 to the value of 3 (because this is the first ‘column’ in the chain) and apply the modulo function. This shows that 4 mod 36 = 4. Looking up the plaintext from Table 3 of the value 4, it is found to be the plaintext character ‘e’. This is the first chain position, which is labeled X_1.

Next, the plaintext at X_1 is encrypted and becomes ‘eh’. Per the previous tables, this cipher text as a numeric value is found to be 151. Now add 2 to that number (because this is the second ‘column’ and the reduction function for the second column must be applied) prior to performing the modulo arithmetic. Thus, we have 153 modulo 36 = 9.

The value of 9 is mapped back to the plaintext character ‘j’, which is the second column in the chain and is labeled X_2. This process of encrypting, reducing, and mapping is repeated until the desired chain length has been reached. When the desired chain length has been reached, the starting point and endpoint of the chain are stored in memory.

X_3	Cipher Text	Cipher Text Value	R_3	X_4	Cipher Text	Cipher Text Value	R_4	X_5	Cipher Text	Cipher Text Value	R_5	X_6
p	ps	558	22	**w**	wz	817	30	**4**	47	1,113	3	**d**
s	sv	669	25	**z**	z2	928	33	**7**	7a	1,188	6	**g**
0	03	965	33	**7**	7a	1,188	5	**f**	fi	188	14	**o**
6	69	1,187	3	**d**	dg	114	11	**l**	lo	410	20	**u**
b	be	40	8	**i**	il	299	16	**q**	qt	595	25	**z**
j	jm	336	16	**q**	qt	595	24	**y**	y1	891	33	**7**
k	kn	373	17	**r**	ru	632	25	**z**	z2	928	34	**8**
o	or	521	21	**v**	vy	780	29	**3**	36	1,076	2	**c**

Table 6: Chain Position X_3 through Chain Position X_6

X_6	Cipher Text	Cipher Text Value	R_6	X_7	Cipher Text	Cipher Text Value	R_7	X_8	Cipher Text	Cipher Text Value	R_8	X_9
d	dg	114	13	**n**	nq	484	24	**y**	y1	891	0	**a**
g	gj	225	16	**q**	qt	595	27	**1**	14	1,002	3	**d**
o	or	521	24	**y**	y1	891	35	**9**	9c	1,262	11	**l**
u	ux	743	30	**4**	47	1,113	5	**f**	fi	188	17	**r**
z	z2	928	35	**9**	9c	1,262	10	**k**	kn	373	22	**w**
7	7a	1,188	7	**h**	hk	262	18	**s**	sv	669	30	**4**
8	8b	1,225	8	**i**	il	299	19	**t**	tw	706	31	**5**
c	cf	77	12	**m**	mp	447	23	**x**	x0	854	35	**9**

Table 7: Chain Position X_6 through Chain Position X_9

X_9	Cipher Text	Cipher Text Value	R_9	X_{10}	Cipher Text	Cipher Text Value	R_{10}	**EP = X_{11}**
a	ad	3	13	**n**	nq	484	27	**1**
d	dg	114	16	**q**	qt	595	30	**4**
l	lo	410	24	**y**	y1	891	2	**c**
r	ru	632	30	**4**	47	1,113	8	**i**
w	wz	817	35	**9**	9c	1,262	13	**n**
4	47	1,113	7	**h**	hk	262	21	**v**
5	58	1,150	8	**i**	il	299	22	**w**
9	9c	1,262	12	**m**	mp	447	26	**0**

Table 8: Chain Position X_9 through the Endpoint

3.4.5 Using a Rainbow Table for Decryption

Typically, the resulting rainbow table would be sorted by the endpoint, so as to allow fast binary searching. This optimization shall be ignored in this example. The final rainbow table in this example is shown in Table 9. Note that to store this rainbow table requires only 2 bytes of storage per chain for a total of 16 bytes. To store an entire lookup table for the entire plaintext space would require 3 bytes for every plaintext value, thus would require 108 bytes total. Note that the rainbow table is less than one-sixth of the size requirement for storing the full table, thus the space savings are considerable.

SP	EP
a	1
d	4
l	c
r	i
w	n
4	v
5	w
9	0

Table 9: Final Example Rainbow Table

To use a rainbow table for decrypting information, the underlying assumption is that we have access to the encrypted data. Assume that the encrypted data we have been provided with is ‘hk’. The first step in decrypting the data is to see if this matches with any of the endpoints by applying the reduction function R_{n-1} where *n* is a counter that begins with a value of the chain length.

The cipher text ‘hk’ maps to a numeric value of 262. If the reduction function R_{10} is applied to our cipher text, we end up with the equation (262+11) mod 36 = 21. The value of 21 maps onto the plaintext space and becomes the letter ‘v’. Checking the Endpoints in Table 9, we see that a match has been found.

Since a match has been identified, the corresponding starting point for the chain is identified as the plaintext of ‘4’. To find the plaintext for which ‘hk’ is the cipher text, start at the plaintext of ‘4’, then encipher, map, and reduce the plaintext for $n - 1$ times. Thus, we apply this sequence of instructions 10 times’.

From Table 8, we see that the plaintext at X_{10} is ‘h’. This is considered to be a potential match. If encrypting the plaintext of ‘h’ results in ‘hk’, then the plaintext has been found and the hash has been decrypted. If encrypting the plaintext found at this position in the chain results in anything other than ‘hk’, then a false positive has been encountered. False positives can occur due to the reduction function that maps a larger set of encrypted values back onto a smaller set of plaintext values. This can cause multiple hashes to map back to the same plaintext, which in turn can cause false positives.

If applying the reduction function R_{n-1} to the cipher text does not yield a match to an endpoint, the result is thrown away and R_{n-2} is applied instead. If a potential match is found, then the process of enciphering, mapping, and reducing the plaintext is applied $n - 2$ times. If this does not yield a matching result, then the reduction function R_{n-3} is applied. This continues until all reduction functions have been applied or a valid plaintext match has been found.

3.5. Summary

It is clear from Chapters 3.1 – 3.3 that the development of Oechslin’s rainbow tables was a significant advancement in cryptanalysis. They provide solid, quantifiable advantages over Hellman’s methods. Even when compared to Ronald Rivest’s advancements, the rainbow tables prove to be more efficient and better in general. However they are not without their problems.

The single biggest problem with any of these methods is the sheer time required to pre-compute the lookup tables. Neglecting disk write times, a reasonably fast computer can achieve a hashing rate of 2 million md5 hashes per second. The examples provided in the introduction of this chapter would take approximately 16 days and 38,733 days, respectively. While the first timeframe is tolerable, the second is not.

The goal of this thesis is to demonstrate a parallel methodology that can be used to reduce the time it takes to build rainbow tables, regardless of the encryption algorithm. While it will not improve the success rate or change how rainbow tables fundamentally operate, it will address the underlying shortcoming of rainbow tables, which is the time investment required to create the tables before they can be used.

Chapter 4 Parallel analysis and design

Parallel computing is a methodology used to decrease the amount of time required to solve a large problem by dividing it into smaller tasks which are solved concurrently. This parallelism can be achieved using either hardware, software, or a combination of the two.

This chapter focuses on analyzing how to address the parallelization of rainbow table generation from the software perspective. First, an introduction to parallel computing is provided. Next in Chapter 4.2 task granularity is explored, followed by an analysis of rainbow table task granularity in Chapter 4.3. In Chapters 4.4 – 4.6, the effect of hardware resources on task granularity is detailed for the md5 algorithm on the test hardware being used.

In Chapter 4.7, the resulting software architecture is described, including the rationale for various design choices, such as network latencies, processor speeds, and work assignment. Finally, Chapter 4.8 summarizes the analysis and design.

4.1. Introduction to Parallel Computing

Amdahl's Law [14] states that the potential speedup of any application when moved from a single processor to multiple processors is governed by the following equation, where P is the fraction of the application that is parallelizable and N is the number of processors used.

$$Speedup = \frac{1}{(1-P) + \frac{P}{N}}$$

Any sufficiently large problem will consist of parts that are parallelizable and parts that must be evaluated sequentially. For example, in all of the time-memory tradeoff algorithms discussed in this thesis, building of an individual chain is considered to be a sequential procedure that may not be parallelized, due to the inherent dependencies upon previous results in the chain.

As *N* approaches infinity, the maximum speedup factor that can be achieved is:

$$Speedup = \frac{1}{(1 - P)}$$

Amdahl forwards this theory based on a fixed problem size that has been misused over the years to argue against parallelization. In essence, Amdahl's law states that if a problem has a serial component that accounts for 10% of the application runtime, then we can achieve no more than a 10x speedup. For more than 30 years, this was used to argue against parallel computing because it could not be refuted.

While Amdahl's Law is still regarded as being technically accurate, Gustafson formulated a new method [15] of calculating the speedup and expanded the apparent usefulness of parallel computing. Gustafson points out that Amdahl's Law is only directly applicable when the problem is a fixed size, thus the problem has already been explored in an entirely measurable sequential manner.

Gustafson suggests that by scaling the problem size itself, we can achieve a different speedup factor from the base problem. Amdahl's approach would be to measure the sequential timing of the scaled problem and then calculating the speedup that could be achieved by parallelizing it. Gustafson's Law removes the fixed problem size limitation to provide a new perspective. It can be described mathematically as follows:

Let *n* represent the size of the problem. Let the function a represent the sequential fraction of the application and the function b represent the parallel fraction of the application. Thus:

$$a(n) + b(n) = 1$$

To generalize the time equation, we have:

$$a(n) + pb(n) = 1$$

where p represents the number of processors. The speedup achieved by increasing the problem size is thus:

$$S = a(n) + p\big(1 - a(n)\big)$$

Research done by Yuan Shi [16] in 1996 shows that Amdahl's Law is mathematically equivalent to Gustafson's Law. He points out that there are several prerequisites to applying Amdahl's Law and that they are often neglected. The primary prerequisite to the application of the law is that "the serial and parallel programs must compute the same total number of steps for the same input". He suggests that only time-based formulations should be used for evaluating the performance of parallel applications.

Yuan Shi also notes that in Amdahl's Law, it is not practical to obtain the serial percentage of any given application. Deriving the serial percentage from computational experiments leads to the inclusion of overhead in the experimental application, including communication, I/O, and memory access. However, counting the total number of serial and parallel instructions would exclude the overhead, thus preventing speedup predictions from being accurate. A hybrid approach may yield a more accurate answer,

but the work associated with doing so is only going to be applicable to a specific problem and may not be generalized.

4.2. Parallel Granularity

In parallel computing, granularity is a relative measurement of the ratio of computation to communication. This often results in classifying something as either a course grained task or a fine grained task. Course grained tasks tend to have very little communication, while fine grained tasks tend to be more communications intensive. The classification can be applied to discrete portions of the application, or to the application as a whole. It is not uncommon to have a course grained parallel application with some fine grained components.

A third measure of granularity is not often used, but does exist. It is called “embarrassingly parallel” and refers to any tasks which exhibit massively inherent parallelism. An embarrassingly parallel task is any task for which the problem size can be scaled up dramatically to N processors, and achieve a speedup of approximately N, due to the lack of communications required between tasks.

A simple example would be a dataset where an XOR function must be applied to every byte in the data set. There is no cause for processes to communicate with one another, and there are no dependencies between the tasks which would require communications. At the end of the processing, it might be required to recombine the dataset, but this is not typically a part of the parallelization process or considered in the speedup factor.

Analyzing the granularity of a problem is an important part of determining how easily the solution can be parallelized and can provide a good approximation of what the anticipated speedup would be. Based on a machine size that is measured in the number of processors, the optimal granularity can change [17]. Hammond, Loidl and Partridge described a tool for analyzing task granularity and attempting to quantify the optimal grain size for parallel applications.

The inherent difficulty in quantifying the optimal grain size has led to studies that assist programmers in visualizing communication patterns [18]. However, additional research [19] illustrates that simply identifying the communication patterns between objects in memory and using those objects as independent tasks is not enough for two important reasons. The first relates to the fact that creating new objects in memory in a serial application is done many times per second, yet is necessary to do so. While the performance costs of doing so tends to be high, the costs become prohibitive in a parallel environment due to additional overhead.

The second reason is that communication costs become prohibitive when every object in memory becomes a separate task and thus is required to communicate separately with every other task. It makes no difference whether the messages are sent using shared memory, or some sort of message passing library. The sheer number of additional data objects in memory that must be created and the overhead incurred by either shared memory or network library messages makes this prohibitive. Even shared memory message passing is significantly slower than cache accesses in a single processor machine running a sequential program.

4.3. Granularity of Rainbow Table Generation Tasks

The generation of rainbow tables is an inherently massively parallel operation. Virtually no communication is required between the processors that are generating the chains. In theory, the speedup that can be achieved by parallelizing the table generation can be as high as the number of chains, where each processor is assigned a single chain to generate from a data table. This neglects the time required to combine the results. Of course, housing each result on a separate processor would result in the capability to perform a massively parallel decryption effort.

It is not particularly realistic to use a separate processor for every chain, as a single rainbow table may consist of over a hundred million chains and the network transmission costs associated with assigning extremely tiny workloads would greatly hinder the performance of the system. Instead, number of chains must be divided such that they may be assigned to different processors in the system with the goal being that all processors finish their work at approximately the same time. In a cluster configuration or any homogeneous environment, this is a very straightforward task and falls under the category of “embarrassingly parallel”.

To divide the work in a homogeneous environment, simply divide the number of chains by the number of processors, and instruct each processor to generate the chains it has been assigned. Assuming that the processors involved are approximately equivalent, each processor should finish its task at approximately the same time. This is an important distinction between this thesis and a typical parallel problem running on a homogeneous cluster. The focus of this thesis is generating the rainbow tables on heterogeneous

hardware environments as opposed to homogeneous hardware. This has a direct impact on the granularity of the tasks that can be assigned.

In a homogeneous environment, the task size would likely be $\frac{mt}{N}$, where N is the number of processors and *mt* is the total number of hashes to be produced. In a heterogeneous environment, this is no longer the most efficient division of tasks. The efficiency of the system can be modeled by the following equation:

$$Efficiency = 1 - \sum_{p=1}^{N} \frac{t_{idle\ percentage}}{N}$$

For every processor in the system, we sum the idle time percentage and divide by the total number of processors. The resulting percentage is subtracted from 1 to give us our final efficiency, which is the amount of time that processors in the system are idle in relation to the time that they are doing work.

Take, for example, a system with five processors, four of which are 0% idle and the fifth is 50% idle. This results in an Efficiency of the system of:

$$Efficiency = 1 - \left(\frac{0}{5} + \frac{0}{5} + \frac{0}{5} + \frac{0}{5} + \frac{0.5}{5}\right) = 90\%$$

Intuitively, this makes sense because each processor is expected to do about 20% of the work and four are 100% busy, resulting in 80% efficiency. The fifth processor is only working half the time, thus only contributing half of its available CPU cycles to the task at hand doing meaningful work, thus the addition of an additional 10% efficiency for a total of 90%.

Note that this equation is time based and has absolutely no relationship to the processing capacity of the processors. The efficiency is measured based on the percentage of time doing work vs. the percentage of time not doing work. If every

processor were fully loaded throughout the life of the parallel program, then efficiency would be 100%. In practice, this would not be likely. To divide the work in a meaningful way that is more efficient, we need to account for differences in disk I/O, processor hashing speeds, fluctuating workloads, and network messaging overhead.

4.4. Disk I/O Considerations

The number of disk accesses is relative, from one processor to another. The faster a processor is able to process units of work, the more disk accesses it will do. The proportion of disk accesses will grow in direct proportion to the speed at which an individual processor completes chains, as the starting and ending points of a chain are written to disk at the same time immediately after the ending point has been calculated.

In relation to the number of hashes generated, the amount of data stored on disk is minimal, thus the occasional disk accesses that must be made to store the starting and ending points will have very little impact on the final system. All processors are expected to make some disk accesses, and the amount of data written to disk in any given disk access is only 4 bytes for the starting point, and 4 bytes for the ending point. Some sample hashing times for three of the test machines are provided in Table 10.

Computer Specifications	Approximate MD5 hashing speed (hashes/second per processing unit)	Approximate Total hashing speed across all processors
1) Quad Core E5405 2.0GHz Xeon 4.0 GB RAM 10,000 RPM SCSI hard disk	1,183,000 hashes/second	4,732,000 hashes/second
2) Dual Core Athlon XP 2.6GHz 3.0 GB RAM 7,200 RPM SATA hard disk	2,623,000 hashes/second	5,246,000 hashes/second
3) Dual processor 2.4GHz Xeon 2.0 GB of RAM 10,000 RPM SCSI hard disk	695,000 hashes/second	1,390,000 hashes/second

Table 10: Sample test machine hashing speeds

Based on the approximate hashing speeds seen in Table 10, we can calculate approximately how much data will be written to the hard disk each second of processing. The amount of data written is a function of the chain length and is illustrated by both the following equation and the following chart.

$$\frac{Bytes}{Second} = \frac{Hashing\ Speed}{Chain\ Length} * 8\ bytes$$

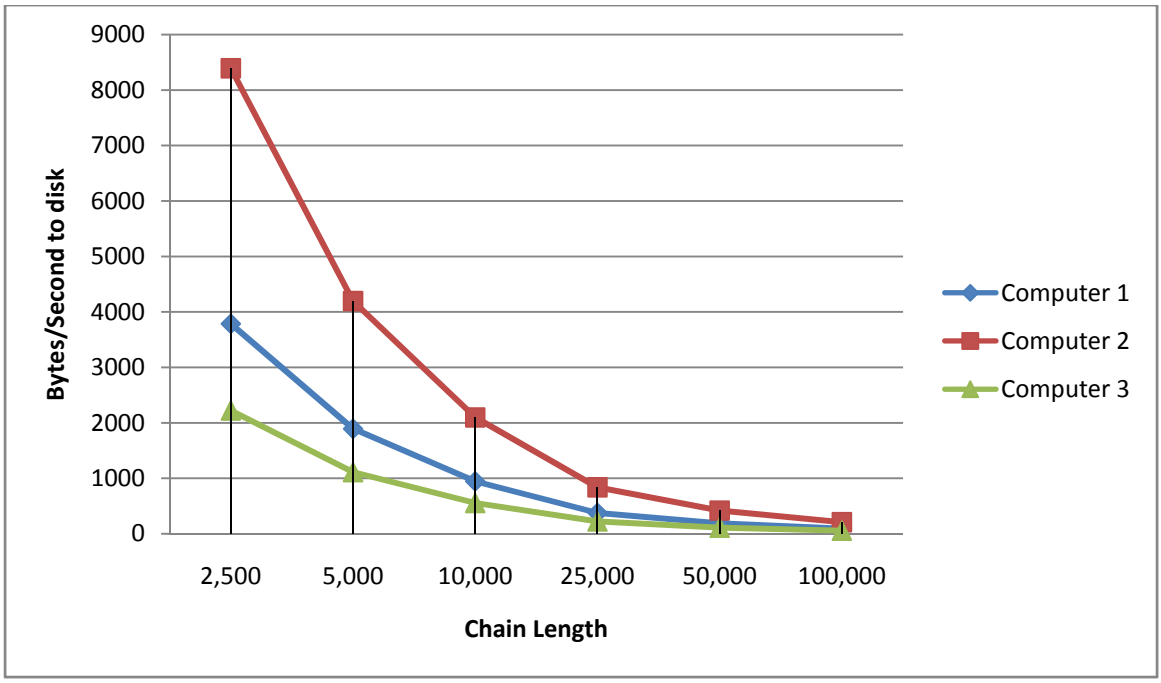


Chart 1: Bytes/second vs. Chain Length

The number of bytes per second written to disk is quite small, as indicated by the above graph. The maximum bytes/second being written to disk is no more than 9,000 in a worst case scenario, which is on Computer #2. We must consider that the worst case scenario on this computer is that both processors are writing to the disk at the same time, thus generating 18,000 bytes/second of disk I/O. The hard disk for this computer is rated at a theoretical 3Gbps, thus providing 402,653,184 bytes/second of bandwidth. The actual disk I/O is a tiny fraction of the I/O that the disk is capable of.

One might also consider that the number of disk accesses per second could influence the results. During testing on Computer #1, it was found that decreasing the chain length to 1 and generating 10 million chains resulted in approximately 156 MB of data written to disk. The anticipated time for completion was only 7.1 seconds, but the measured speed was approximately 21 seconds. The available bandwidth to disk would

indicate that the system can write 156 MB of data to disk in less than 2 seconds, which does not account for the additional 14 seconds of processing time.

A second test using a chain length of 1,000 and 10,000 chains requires a total of 10 million hashes, just as the first test does. In this test, the calculated optimal time was between 7.15 – 7.18 seconds and the measured time was 7.17 seconds. These tests show that disk accesses per second can play an important role in table generation speed.

In practice, disk access has minimal impact on the application because chain lengths are typically in the thousands, or tens of thousands. When chain lengths of more than 1,000 are used, there are significantly fewer disk accesses per second. Modern disk caching techniques tend to reduce or eliminate delays associated with writing data to disk in 8 byte blocks. A common programming technique that takes advantage of head location on the disk is to cache many values to memory and write a single large block of data all at once. This is especially useful for older hard disks or ones that do not support command queuing architecture. Chain lengths of less than 1,000 are not practical in any case.

4.5. Differences in Processor Hashing Speeds

It is important to realize that the differences in hashing speeds have nothing to do with the amount of RAM in the computers or the speed of the hard disks. The hashing speeds are a factor of the processor architectures. Computer #1 is the newest of the three test computers and is a server class machine using an Intel chip. Computer #2 is a high end workstation using an AMD Athlon processor that was released approximately 3 years

ago. Finally, Computer #3 is using much older processors which are approximately 6-7 years old. Table 11 outlines the differences between these processors.

Computer	Hash speed/processor	CPU Speed	Bus Speed	L2 Cache size
1) Intel E5405	1,183,000 / s	2.0 GHz	1333 MHz	2 x 6MB
2) Athlon XP 5200+	2,623,000 / s	2.6 GHz	2 x 1 GHz	2 x 1024KB
3) Intel Xeon 2.4GHz	695,000 / s	2.4 GHz	533 MHz	512 KB

Table 11: Processor Comparison

It is to be expected that Computer #3 is measurably slower, primarily due to the age of the computer architecture. The processors are located in separate sockets which reduce potential issues with shared cache, but the processors only have 512KB of L2 cache, and the Bus Speed is only 533MHz.

More interesting than Computer #3 is the comparison between Computer #1 and Computer #2. The hashing speed of Computer #2 is more than twice that of Computer #1. A variety of hardware differences could help to explain this, however as we pointed out previously, disk I/O is minimal so it really comes down to system memory, CPU speed, bus speed, cache size, and computer architecture.

Computer #1 has less physical RAM per CPU with 1.0GB/CPU while the AMD processor has 1.5GB/CPU. However, the application is a processor intensive application. Adding more system memory over a certain threshold is unlikely to substantially affect the hashing speed. Additional testing has demonstrated that benchmarking the hashing speed while memory intensive applications are running on Computer #1 reduces the hashing speed. This was measured at approximately 10% with a reduction in total memory of 1.5GB. Thus it would seem unlikely that the differences in system memory

have a great enough effect on the computer to account for a doubling or halving of the hashing speed.

The CPU speed is the most likely culprit that influences hashing speed. Computer #3 has a 20% greater raw CPU speed than Computer #1, yet benchmarks at only half the hashing speed of Computer #1. Computer #2 is 30% faster than Computer #1 in this regard, yet benchmarks at more than double Computer #1. CPU speed alone does not seem to be the most significant factor in the hashing speed, but it must be a contributing factor.

Bus speed, cache size and computer architecture are the only remaining factors. The significantly lower bus speed of Computer #3 would seem to account for the difference in hashing speed between Computer #3 and the other computers. However, the bus speed differences between Computer #1 and Computer #2 must be evaluated further. The cache is structured differently between the two types of processors so we must eliminate that first.

The Intel processor shares 6MB of L2 cache between 2 processors while the AMD processor provides 1MB of cache to each processor. It is possible that cache thrashing might be responsible for reduced hashing speeds, but a simple test using only one of the 4 CPU's on the Intel processor eliminated this as a possibility, thus the cache sharing does not impact the hashing speed, and because the Intel processor has more cache to begin with, this cannot be a factor.

Finally, we are left with bus speed and computer architecture as viable contenders for significantly influencing the hashing speed. The bus speed of Computer #3 is significantly lower than the other computers, and thus lends credibility to this theory. The

processor architectures are such that the bus speed is not a direct comparison. The Intel processor shares the bus across all four processors, while the AMD processor has a separate bus for each processor, each running at the same speed.

We can theorize that if the processor bus were shared across all four Intel processors, then the theoretical speed of the bus might be only 333MHz. Testing has shown that whether we use only one CPU on this processor or four at the same time, the hashing speed is not affected so this theory is not plausible. The bus speed of the Intel processor seems to be faster than that of the AMD Athlon processor.

The preceding information leads to only one possible conclusion. The processor architecture is the single most significant factor in hashing speed. While various other differences between the processors may be contributing factors, none can individually or collectively account for the significant differences in hashing speed from one processor to another more than the processor architecture.

4.6. Network Topology

Fundamentally, the network is the slowest component of the average computer. This statement holds true even with the introduction of gigabit Ethernet. High speed clusters may be built that use fiber optic network devices which dramatically reduce the network access times and transmission latencies between nodes of a system to less than that of disk access. These clusters are not in easy reach of the average user, and thus not the focus of this thesis.

In a distributed-memory MIMD system, every processor is considered a separate entity and is connected by an arbitrary network. This is generically shown in Figure 3.

There are numerous types of distributed-memory MIMD systems which are broadly categorized by their network connections into either static or dynamic networks. Static networks are those where the nodes are directly connected to one another, in whatever configuration has been deemed appropriate. In a dynamic network, some nodes are connected to switches, which dynamically determine where to route traffic based on routing information provided with the network traffic.

Modern computer architectures have given rise to multi-core processors which consist of multiple CPU's on a single silicon chip. In Figure 3, this could mean that in some cases, the network used to communicate between processors may actually exist within the CPU itself.

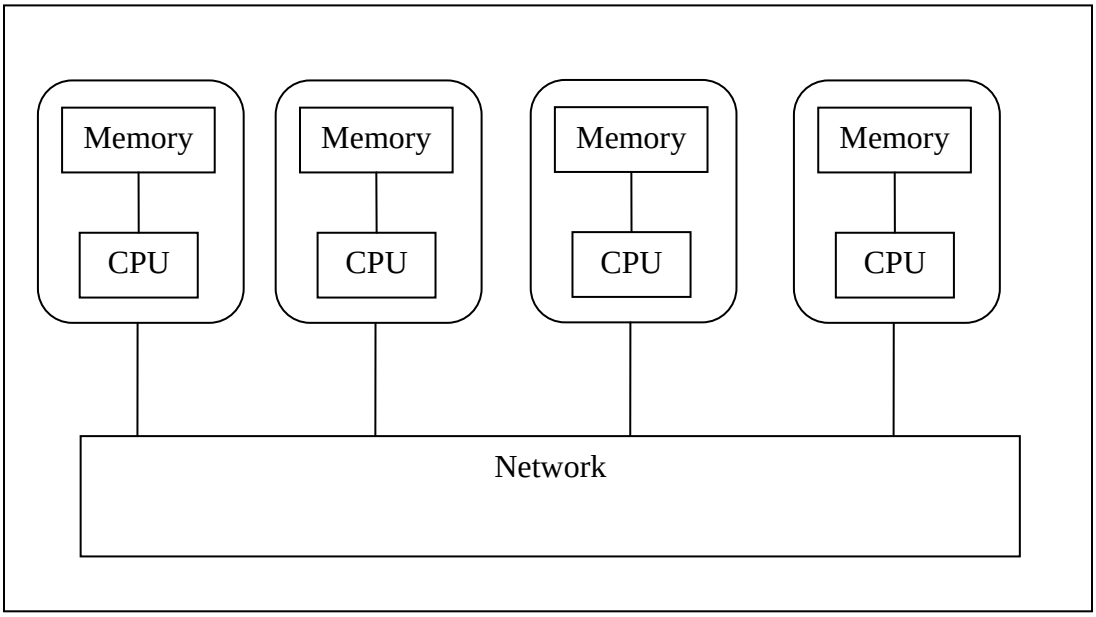


Figure 3: Generic distributed-memory MIMD system

The ideal network topology for a distributed-memory MIMD system is what is commonly referred to as a "star network", where every node is directly connected to every other node. This allows each node to speak directly to every other node, without

any of the problems associated with using a shared network. This helps to alleviate excessive network congestion, and network packet collisions, which lengthen the time between communications. This is shown in Figure 4.

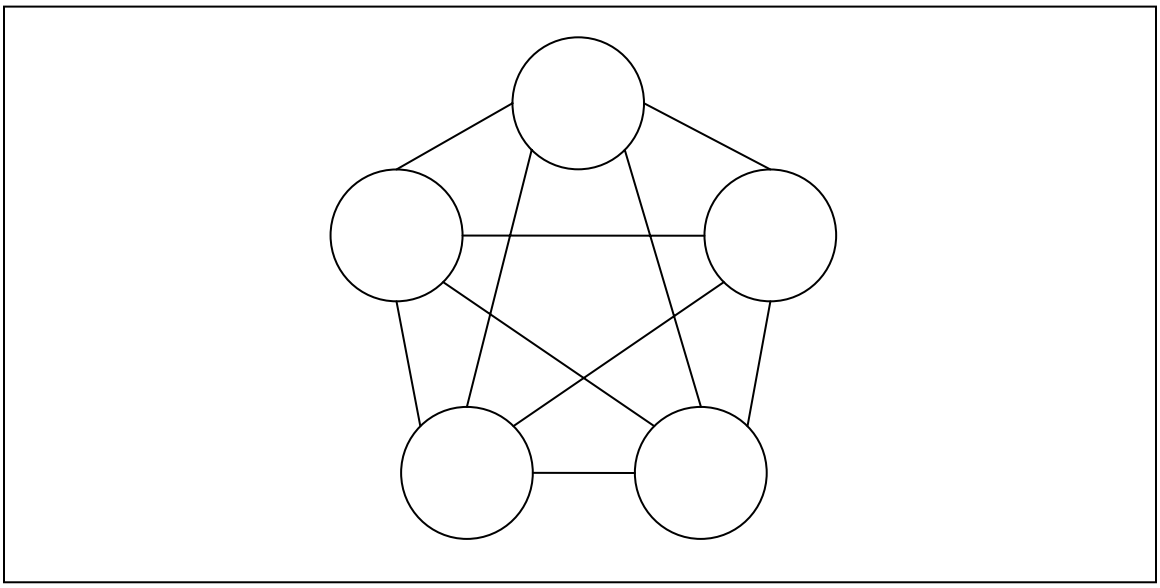


Figure 4: A "star network"

Using hardware that is typically already available on the network, a star configuration is not common. Instead, we concentrate on the network topology provided by the generic distributed-memory MIMD system represented by Figure 3, and commonly interpreted as a network that would commonly be found in a corporate environment. It consists of a number of nodes, each of which have one or more processors and are connected to a network backbone. This network backbone may stretch for as little as a few feet, or across large geographical distances.

In this thesis, we have assumed that computers are local and network latencies can be measured in less than 5 milliseconds between nodes. The attached network is a 1Gbps network and all nodes are on the same network switch.

4.7. Final Architecture

One of the primary motivations of this thesis is to identify ways to generate rainbow tables faster using nodes that are already in place on a typical network. The use of FPGA's for generating rainbow chains has been explored [20] and showed that FPGA's could be used to dramatically reduce the time required to decrypt DES. Generically, this could be applied to rainbow tables for other encryption types. However, a parallel implementation entirely in software using MPI or a heterogeneous network has not been researched, both of which are explored here.

4.7.1 Work assignment using a master and slave architecture

The first step in determining a suitable parallel software architecture is to identify a method of assigning work to the various nodes which results in the highest efficiency possible. From 4.3, this Efficiency is calculated as follows:

$$Efficiency = 1 - \sum_{p=1}^{N} \frac{t_{idle\ percentage}}{N}$$

The best case scenario is if all processors are busy 100% of the time, yielding a 100% Efficiency rating. There are very few ways this can be accomplished. One method might be to predetermine the speed that every processor is able to generate hashes and assign work such that all processors finish at precisely the same time. In this scenario, it is unclear whether this predetermination would be made prior to running the application, or as part of it.

If this predetermination were made during the application execution, then one or more worker nodes must evaluate the results. This typically results in a master/slave relationship between a single master node and all of the other nodes, referred to as slaves.

4.7.2 Determining the ideal master node

Shao, Berman and Wolski [21] suggest a method for determining the master and slaves on a grid of heterogeneous computers which is based on identifying the master, based on the work capacity of each node in the environment. Their research may not be directly applied to this thesis due to the way tasks are assigned because the master selection is arbitrary.

In their research, they explored how to identify the node that should act as a master that would result in the most efficient use of the nodes in a system. Interestingly, they illustrate in their research that as the number of slave processes increases, the efficiency drops off when the communication costs exceed the gain provided by adding more processors. The underlying assumption they make is that the total task size stays constant while the number of processors increases, thus doing little more than illustrating Amdahl's Law [14].

The fundamental assumption that Shao, Berman and Wolski make is that the master should be measured in tasks per second. In addition, it is assumed that this is a sustained measurement of tasks per second which continues for a significant portion of the overall application. The parallel generation of rainbow tables does not perform in this manner.

Instead, the parallel rainbow table generation described by this thesis can assign a variable number of tasks with a single message, without varying the size or number of messages. A single assignment can be thousands or millions of tasks rolled into a single message. This does not match the paradigm described by Shao, Berman and Wolski.

The result is that an arbitrary processor in the system can be chosen and based on the specified task granularity, has the potential to send out a million tasks with a single message. With the same message, it could instead send 10 million tasks, thus modifying the tasks per second with no change in hardware.

If a sufficiently large environment is deployed, the grain size is likely to be quite large, thus negating any issues they raise with determining the best master node and increasing the number of tasks assigned with a single message. As the problem scales up, so does the grain size. Their research holds true in a general sense, but falls short in cases where many tasks can be assigned with a single message and the number of messages per second is low. Therefore, an arbitrary processor may be chosen as the master with no noticeable impact on the efficiency or scalability of the problem, due in part by the nature of the problem and the required granularity.

4.7.3 Dividing the problem into tasks

In a heterogeneous environment, different processors perform work at different rates of speed. Working backwards from the Efficiency equation, we must still determine a way to keep all processors as busy as possible. If we predetermine the hashing speed of all processors prior to execution, it would be possible to dynamically calculate the ratios

of processor speeds. Using these ratios, we can calculate a number of chains that should be assigned to each processor in the following manner.

Processor	Hashing speed (hashes/second)	Relative speed
1	500,000 / s	0.08695
2	750,000 / s	0.13043
3	1,000,000 / s	0.173913
4	1,500,000 / s	0.260869
5	2,000,000 /s	0.347826
Total	5,750,000 / s	1.0

Table 12: Hypothetical hashing speeds

Using a hypothetical heterogeneous environment of five processors as shown in Table 12, we can examine these ratios. The relative speed of each processor is found by adding up the individual processing speeds to get the total processing speed. Then the hashing speed of that processor is divided by the total hashing speed. This is the relative speed of the processor in regards to the entire system.

The simplest method of dividing the work would be to take the number of chains and multiply by the relative speed of each processor in regards to the entire system. Theoretically, this should result in the most efficient system. Every processor would calculate the assigned number of hashes, and should finish at approximately the same time.

There are two reasons why this doesn’t work in practice. The first is that the benchmarking is not very accurate. Hashing speed is a function of the plaintext, and as the chain is generated, the plaintext changes. Benchmarking must be done over a significant period of time in order to be reasonably accurate, and this accuracy is merely

an average. Depending on the size of the total task, benchmarking for a significant period of time might not be very beneficial in determining an average hashing speed.

The second and more important reason that using the relative speed doesn't work in this scenario is that the conditions on the nodes doing work are subject to change and are outside of the control of the parallel application. As noted in section 4.5, system memory was not considered a significant enough factor in determining the differences between processor types, but it was measured at 10%, which is statistically significant for this scenario. In the case of this testing, the process using the additional memory was a database server. Over the course of 72 hours, this becomes 7.2 hours of additional processing that must take place on that processor, which creates a significant amount of idle time on the other processors. In a 10 node system, this alone reduces the efficiency from 100% to 90%, as per the following equation.

$$Efficiency = 1 - \left(\frac{.1}{9} + \frac{.1}{9} + \frac{.1}{9} + \frac{.1}{9} + \frac{.1}{9} + \frac{.1}{9} + \frac{.1}{9} + \frac{.1}{9} + \frac{.1}{9}\right) = 90\%$$

What is interesting about this efficiency problem is that it is virtually no different than doing an improper work assignment. If we used 9 processors that were the same and one processor that was 10% slower, dividing the work equally would result in the same efficiency. In virtually any case, variances in the hashing speed do not bode well for the efficiency of the system as a whole. It is unlikely that all of the hashing algorithms would be either more or less efficient by approximately the same percentage, thus resulting in a noticeable efficiency loss in either case.

A more appropriate division of work is to use the processor farm paradigm, but implement it with a much smaller task size. The above example used a task size of $\frac{mt}{N}$. For our purposes, this is insufficient because as we discussed, the large task size can

easily result in less efficient behavior. For reference, *mt* is the total number of hashes to be produced.

The important question that we need to answer is how to determine what an appropriate task size is. If we are measuring the task size as a number of chains to be generated, then we can either use a static number of chains or a dynamic number of chains. Using a dynamic number of chains allows for the application to adjust the amount of work sent to each node as a function of its hashing speed. It also introduces a level of complexity that is not easily managed.

Using a static number of chains as a task assignment is a more straightforward approach. The goal is to choose a number of chains that is large enough that prevents communication problems between the master and slaves, but small enough to help minimize the loss in efficiency for idle processor time.

We have chosen to implement a system where the grain size is selected based on time, rather than an arbitrary amount of work. To do this, the user specifies what is called a time slice, represented by t_{slice} and is measured in seconds. Let $S_{slowest}$ represent the hashing speed of the slowest computer and W_{slice} represent the number of chains that are in a single task assignment, also known as a work slice. If the chain length is specified as *t* and the total number of chains to be generated is *m*, the following equation governs the work slice:

$$W_{slice} \leq \frac{t_{slice} * S_{slowest}}{t}$$

Solving for t_{slice}, we have the following:

$$t_{slice} \geq \frac{W_{slice} * t}{S_{slowest}}$$

The work slice is measured by an absolute number of chains, but is based on parameters related to hashes per second and the length of each chain. The work slice size is rounded down to an integer for two reasons.

First, the individual task must be completed in a timeframe that does not exceed the time slice specified. This is also the reason that the hashing speed used in the calculation is taken from the slowest computer, rather than any other.

The second reason work slices are rounded down is that they are measured as a number of chains. A chain is made up of *t* hashes. It is unlikely for the work slice to be calculated to an integer on its own, due to the variable nature of the hashing speed of the slowest computer. Assigning tasks that are partial chains introduces a level of granularity in the task that is inappropriate for the problem. In the context of a rainbow table, a half chain is meaningless, so tasks must consist of full hashes.

All work slices are expected to be the same size with one exception. The last work slice that is assigned to a processor will likely be smaller than the rest. A work slice consists of a subset of the total number of chains and is constructed such that the size of that work slice is governed by a user specified time slice. This has no relation to the total number of chains and may not divide equally into it.

One way of dealing with this would be to ignore the implications of generating extra chains and simply include them in the final output. As outlined in sections 3.1 and 3.3, some overlap is expected due to the random starting points of each chain. It is possible that some of the chains generated must be thrown out due to either collisions or

the accidental reuse of a starting point. If extra chains are generated and they are not necessary because of a low collision or starting point overlap, then they may be discarded.

For the purposes of this research, we chose to implement the system in a manner that generates exactly what the user has requested, ignoring potential optimizations that would help to reduce the effects of collisions or starting point reuse. Addressing these issues, while certainly beneficial, are not the main focus of this thesis and must be dealt with similarly in a single node environment.

In addition, one of the post table generation steps is to sort the chains by endpoint, discarding duplicates. For this task, all of the rainbow table data must be collected on a single node. It is a standard practice to discard duplicates after sorting and only generate additional chains as needed to prevent doing unnecessary work. This standard practice is followed in our parallel implementation.

4.7.4 Task Assignment

The assignment of work slices as tasks in the system is best implemented on a first come, first served basis. The restrictions that are in place on the size of the work slice create a fixed size task that is based on a maximum time that each task is anticipated to take on the slowest processor. Faster processors will complete the work slice in less time and thus complete more work slices throughout the runtime of the application.

A byproduct of this is that a certain amount of load balancing is guaranteed. Faster processors will complete more work units than slower processors and using a first

come, first served basis guarantees that the maximum processor idle time will not exceed $N * t_{slice}$.

4.7.5 Network Considerations

Every system contains bottlenecks and due consideration must be given to each of them, including the network. Many parallel applications put a larger amount of stress on a network than a typical application due to communication messages and data transfer that are required to solve the problem.

The generation of rainbow tables creates an unbalanced load on the network because there are two distinct phases of the process. The first phase includes both the assignment of tasks, and the generation of the tables themselves. This is not excessively stressful on the network. The messages that carry the task assignment are only a few bytes in size. Even if we consider the additional overhead associated with using MPI for this task, the load on the network is considerably low in this phase of the problem.

The second phase includes aggregating all of the results onto a single master node. Results are sent to the master node upon request by the master with up to 1,024 chains at a time. Upon receiving the initial request, each node will send the starting and ending point of every chain it has generated back to the master node. The aggregation message is not sent until every task in the system is complete.

A standard rainbow table is no more than 2 GB in size. Thus, the total size of the data being aggregated at the master node will be no more than 2 GB, plus the typical overhead associated with sending network messages. Using a 1 gigabit network, which

can be typical in a corporate environment, each table can theoretically be transferred back to the master node in approximately 16 seconds.

However, a typical corporate network does not run at theoretical speeds. Running at only 20% of theoretical bandwidth is common, thus we use 200 Mbps as the measured network bandwidth capacity. Using this capacity, a 2 GB rainbow table may be aggregated on the master node in approximately 82 seconds. Assuming that the performance could be as much as twice as bad as that due to network latencies and collisions, aggregation of the results should take no more than 3 minutes.

This aggregation time would not typically be counted towards the performance of a parallel application, as it is commonly assumed to take place offline. It is important to note that an additional 3 minutes of time spent aggregating results from multiple nodes is far less of an impact on the total problem time than if the rainbow table chains were all generated on a single node. This is further validation of a parallel implementation.

4.8. Parallel Analysis & Design Summary

In this section, we have examined various components that are associated with generating rainbow tables on multiple nodes in parallel. It is obvious that simply dividing the total work equally among all of the nodes is not a viable option. This method forces the application performance to be no better than $\frac{Single\ Node\ Time}{N_{slowest}}$, where $N_{slowest}$ is the slowest node in the cluster.

Similarly, benchmarking the nodes in the cluster is not an ideal solution either. Due to circumstances outside of the control of the parallel application, it is entirely possible for the application to run slower and slower on any given node. Other

applications may be executing, or available memory may decrease over time. This can cause unforeseen fluctuations in hashing speed.

Ultimately, these fluctuations cause a loss of efficiency that is compounded by the total time the problem takes. A loss of 10% efficiency on a single processor results in an identical amount of idle time on all other processors. This inefficiency must be avoided. The best way to address this is to determine a reasonable task size that may be assigned to each node in the system such that the problem is solved with as little idle time on each processor as possible.

It must be pointed out that the generation of a rainbow table is only the first of three distinct components required to use rainbow tables. Rainbow table generation is the sole focus of this thesis, as it is the area that offers the most significant gain.

The second component is a tool for sorting the rainbow tables by the endpoint. Recall that in the example in Chapter 3.4.5, it was stated that sorting the rainbow table was ignored for the purposes of that example. In practice, the rainbow tables must be sorted by the endpoint to allow for fast binary searches of the endpoints for matches. An unsorted rainbow table is orders of magnitude slower than a sorted table.

The third and final component is a tool for searching the tables after they have been built. A rainbow table is useless without a tool that can search through the tables to find the plaintext values of cipher text. This is an area that has not yet been parallelized and would offer a high potential for performance improvements, especially when used on extremely large problem sets where the chain length is long or the number of tables used is relatively high.

Many critical components of parallel rainbow table generation have been examined in this chapter. It was important to establish a solid design so that the potential bottlenecks of each component could be minimized, yielding a better implementation. In the next chapter, the implementation of this design is detailed.

Chapter 5 Implementation

The main purpose of a parallel implementation is to solve a given problem faster than it could be normally solved on a single node. In this case, the problem is the generation of a rainbow table based on algorithms developed by Philippe Oechslin [3]. In some cases, it is not feasible to build rainbow tables on a single node, due to the sheer amount of processing required. The time it takes to build a single table can be measured in months, or even years.

This chapter provides an overview of the implementation of a parallel application for creating rainbow tables in a distributed heterogeneous environment called PRTGen. To help differentiate between an implementation built for a single node and the parallel implementation, the former will be referred to as RTGen and the latter as PRTGen.

PRTGen is based on an implementation of Oechslin's rainbow table algorithms called Rainbow Crack, written and published by Zhu Shuanglei. This tool and the source code are freely available for download on the internet. Rainbow Crack was heavily reengineered to accommodate a parallel implementation using MPI. The reference source code was not well documented, so an effort to document it was undertaken to allow future researchers the opportunity to make meaningful changes.

First, the MPI and PRTGen application parameters are discussed in detail. Next, the architecture of the application and the communications are examined. Then, potential problem areas are addressed. Finally, platform specifics are provided such that the implementation can be duplicated by others.

5.1. MPI Setup Parameters

The implementation of any parallel application requires the use of a messaging subsystem for the various nodes to communicate with one another. Several implementations of messaging subsystems exist and are readily available. It is certainly possible to design a parallel application without the use of such programming libraries. However, the use of such libraries offers an API that has been extensively tested and is, presumably, more reliable than an implementation written from the ground up.

API's such as the Message Passing Interface (MPI) or Parallel Virtual Machine (PVM) provide the programmer with a standard set of portable libraries that can be used on multiple platforms. This allows the utilization of the same function calls, regardless of the operating system, so long as an implementation of that particular API exists.

For the PRTGen application, version 1.0.5 of the MPICH 2 library from Argonne National Laboratory was used [22]. This API supports a wide variety of hardware platforms, including 32 and 64 bit Windows, Unix, and various Linux distributions such as Ubuntu, FreeBSD, Slackware, Fedora, and Debian. In a corporate environment, the standard desktop is a Windows computer, thus Windows is the focus of the implementation. There are some parts of the application that are likely to be platform specific, but these are often noted in the source code. It was not tested on other platforms.

There are two MPI application parameters that must be used when running PRTGen. The two parameters are "–hosts" and "–path". The "-hosts" parameter instructs the MPI subsystem which nodes the application is to be executed on. The MPICH 2 implementation runs as a service wherever it is installed and the MPI application calls are

passed through this service. The service is responsible for instantiating the PRTGen application on each of the nodes.

Application failures or exceptions are passed back to the master node for handling. The nature of MPI is such that failures or exceptions must be handled by the parallel application. There is no clearly defined standard for what MPI should do with any given exception, thus the typical way it is handled is by aborting the entire application.

The “-hosts” parameter has two different ways it can be used. The first, requires the user to enter the number of host systems, followed by *n*, which is the number of processes to run on each system, and then the list of hostnames. This is specified in the following format:

-hosts <n host1, host2 … hostn>

Optionally, the number of processes to start on each individual system can also be specified. This allows the user to have a different number of processes on each system, rather using the same number of processes on every system. The purpose of this is to allow the user to allow more powerful computers to act as more nodes within the system. A common place this is applicable is on computers with multi-core processors.

In the implementation of PRTGen, it is assumed that a computer with multi-core processors will treat each processor as an independent node in the system. Thus, a single core processor in a computer would count as one node, a quad core computer would

count as four nodes, and a dual quad core computer would count as eight nodes. This is specified in the following format on the command line:

-hosts <n host1 m1 host2 m2 ... hostn mn>

The second major command line parameter in MPI that must be used is “-path”. The “-path” parameter essentially modifies the environment path on the target node. This is necessary to assist MPI in knowing where the PRTGen application is located on each node in the system. The execution of PRTGen will fail if the application is not located in the same drive and directory on each computer.

Depending on the computers involved and the available disk space on each, this is not always going to be the case. The expectation is that temporary files that are generated for the rainbow tables are stored in the same location as the PRTGen executable so the available disk space may require moving the executable to a different drive.

In addition, PRTGen uses some configuration files during execution and these are expected to be located in the PRTGen directory. The “-path” parameter allows the user to specify one or more locations where PRTGen is located. Multiple paths can be entered by enclosing them in double quotes and are semi-colon delimited. The following format is used:

-path <search path for executable, ; separated>

After specifying the hosts and the potential paths for the executable, the PRTGen application and its configuration parameters are specified.

5.2. PRTGen parameters

The PRTGen application has nine required command line parameters, and one optional parameter. These parameters allow a user to influence the rainbow table that is being generated. The following line illustrates the usage of the prtgen.exe command.

**prtgen hash_algorithm charset minlen maxlen **

table_index chain_len chain_count file_suffix timeslice [-bench]

5.2.1 hash_algorithm

There are three different hashing algorithms that may be used with PRTGen. The three hashing types are NTLM from Microsoft, MD5, and SHA1. Additional hashing algorithms can easily be built into the application, but these are currently the only ones that are accepted.

5.2.2 charset

The “charset” is a set of characters that is specified in an external configuration file called “charset.txt”. This configuration file must be located on every node in the cluster and must be located in the same directory as prtgen.exe.

The format of charset.txt is a text string that identifies the character set, followed by an equals sign and the character set enclosed in square brackets. There are no escape characters for square brackets contained within the character set. Characters are read into the character set until the final square bracket on the line. The following lines are example character sets that might be defined in a charset.txt file and were used during testing.

```
alpha      = [ABCDEFGHIJKLMNOPQRSTUVWXYZ]
alpha-numeric  = [ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789]
alpha-numeric-symbol14 = [ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789!@#$%^&*()-_+=]
all  = [ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789!@#$%^&*()-_+=~`[]{}|\:;'"<>,.?/]
numeric                = [0123456789]
loweralpha             = [abcdefghijklmnopqrstuvwxyz]
loweralpha-numeric  = [abcdefghijklmnopqrstuvwxyz0123456789]
```

If desired, it is also possible to specify “byte” as the charset. Doing so will specify all 256 characters as the charset of the plaintext. This allows for Unicode password decryption in addition to ASCII decryption.

5.2.3 minlen

The minlen is the minimum length of the plaintext string. Specifying a minlen of zero or one that is less than the maxlen is not allowed.

5.2.4 maxlen

The maxlen is the maximum length of the plaintext string. Specifying a maxlen of zero or one that is less than the minlen is not allowed.

5.2.5 table_index

The table_index is used to differentiate between multiple tables that have been created using virtually the same initialization parameters. From Chapter 3, it is known that using multiple tables results in a higher success rate. The table index is used for two things.

First, it allows multiple tables to be stored in the same directory by providing each with a different index number. Second, the table index affects the reduction algorithm such that two identical starting points in different tables do not result in the same ending point. This helps to improve the success rate of a set of tables because collisions across tables are eliminated.

5.2.6 chain_len

The chain length is the numbers of times that the hashing and reduction function are applied to a given plaintext starting point to determine the ending point. The greater the chain length is, the higher the success rate [3].

5.2.7 chain_count

The chain count is the number of chains to generate. This number is limited to no more than 134,217,728 chains. Each starting and ending point is made up of 8 bytes, thus storing a pair requires 16 bytes on disk. If we attempt to store more than 134,217,728 chains results in a single file, it will exceed 2 GB in size. It is standard practice to not do this for a several reasons. First, physical memory on many computers is somewhat limited. Two gigabytes of RAM is not terribly uncommon, but more than that often requires 64 bit operating systems.

Windows computers are typically limited to 3 GB of RAM without a 64 bit operating system. If a rainbow table is into memory that is larger than the physical RAM in the computer, it causes a paging file to be used, thus significantly slowing down the system. Until 64 bit computers with more RAM are more prevalent in a corporate network, this limit should not be exceeded.

This limitation is hardcoded into the application. Should circumstances change, the software may be changed to accommodate larger rainbow tables.

5.2.8 file_suffix

The file suffix has nothing to do with generating rainbow tables. It is simply a mechanism for naming rainbow tables differently based on when they were generated, or the purpose behind generating them. This suffix is added to the end of the filename of the rainbow table and allows multiple rainbow tables to be generated in the same directory while avoiding overwriting one of them unintentionally.

5.2.9 timeslice

The time slice is one of the more interesting parameters of PRTGen. All of the previous parameters are determined by the user in order to build a solution to a particular problem. However, the time slice is the only parameter that can be modified to adjust the efficiency of the application.

The time slice is measured in seconds, and is the maximum time in seconds that a work slice should execute on the slowest processor. Using this parameter, PRTGen calculates the size of a work slice and assigns these work slices as tasks to the slave nodes. The total number of work slices is approximately the total time that it would take to create the rainbow table on the slowest computer, divided by the time slice.

Adjusting the time slice upwards results in less frequent communication between a slave node and the master node, but increases the potential for lower efficiency should the average hashing speed drift significantly from the calculated hashing speed found during the benchmarking process.

Decreasing the time slice will result in more communication between the master and slave nodes. This introduces more overhead and latencies into the application due to higher network traffic.

5.2.10 bench

The benchmark parameter exists so that the application can provide the user with an approximation of how long a particular rainbow table is expected to take based on some quick benchmarking of the nodes in the system. If the user determines that the task will take too long after running a benchmark, more nodes can be added, or the rainbow tables themselves can be resized to help accommodate for this.

The benchmark process used here is no different than the benchmarking process using in actually generating the rainbow tables. The difference is that this instructs the application to stop after the benchmarking process, allowing the user to determine the best course of action based on the information provided.

5.3. PRTGen Communications Architecture

PRTGen is structured in a master-slave configuration. A common implementation assumes that each processor in the cluster is considered to be one node. There exists only one master node, and all others are referred to as slaves. In smaller implementations, the processor running the master node may also be used as a slave node.

The master is responsible for coordinating tasks among the slaves. The selection of the master node is done by the user and may be considered an arbitrary selection. As discussed in section 4.7.2, the choice of the master node for this problem is immaterial to the results.

The general program flow is as follows. At the command line on the root node, the PRTGen application is executed with the desired parameters. The MPI subsystem then starts execution of the PRTGen application on each node in the cluster. The master node is automatically selected as the node where the command line program was executed.

The slave nodes immediately begin a benchmarking process by performing 5,000,000 hash and reduction functions. The time it takes to complete this process is measured and sent back to the master node. The master node aggregates all of the results and then calculates its own hashing speed, also known as the reference speed. Meanwhile, the slave nodes wait for the signal from the master node to start requesting tasks. The slave nodes wait for the master node to signal them to continue to prevent skewed results from the reference node.

Once the master node has calculated its own hashing speed, benchmark information is printed to the screen. This provides an overview of the hashing speed of

every processor in the system. It also calculates the total hashing speed of the system versus the hashing speed of the master node, and the optimal completion time of the reference node versus the optimal completion time of the entire system.

Finally, as part of the benchmarking process, the work slice size in chains and total number of work slices is printed. If the “-bench” command line option was entered, all processes exit the application. Otherwise, master and slave nodes continue with program execution.

Each slave node will wait until it is instructed to perform work. The master node immediately assigns one work unit to each slave node, indicating the number of chains it should generate, and the task ID associated with that task. The task ID is used only for logging purposes and ensures that all tasks have been assigned and completed.

Once the master node has assigned a task to each slave, the master listens for responses from the slaves indicating that they have finished their assigned task. This text message to the master is printed to the screen as a status update to the user, but could easily be sent to a log file instead. After the master receives a message from the slave, it verifies the completed chain count and determines whether there is any more work to complete.

If a full work slice of work is available, it is sent to the slave as if it were no different than the first assigned task. If a partial work slice is available, then this partial work slice is assigned to the slave that just finished and the master enters a synchronization phase. From this point on, the master accepts messages from the slaves indicating that each slave has finished its assigned work, but then the master assigns tasks

of size zero to each slave. If a full work slice is assigned and the total amount of work to be assigned is reduced to zero, then the master enters the synchronization phase anyway.

A task of size zero is a special case in which the master is instructing the slave to wait for further instructions. When the last node has finished its assigned work and all slave nodes are awaiting further instructions, the master node calculates the actual completion time, measured from just before the first task is assigned to each node in the system. The completion time is shown in seconds, minutes, hours, days and years as a decimal.

Finally, the master node signals the slaves to send all of the rainbow chains they have generated back to the master node so that they may be aggregated into a single rainbow table. The process of aggregating the rainbow chains is timed and measured separately from the generation process. For reference purposes, this time is broken down into seconds, minutes, hours, days and years so that it may be compared directly to the actual completion time of the rainbow table generation.

The majority of PRTGen is executed in parallel, but there are two primary synchronization points. The presence of these synchronization points can be clearly seen in Table 13, as shown by the absence of a task in the “Slave node tasks” column.

Neither of these synchronization points detracts significantly from the efficiency of the overall application. The first synchronization point is prior to task assignment and is part of the benchmarking process, so it is not counted in the total execution time. It is considered as part of the application setup.

Master node tasks	Slave node tasks
Initialization	Initialization
Wait for slave benchmark results	Measure hashing benchmarks
Receive slave benchmark results	Send benchmark results to master
Perform reference node benchmark	Wait for master to perform benchmark
Assign initial tasks	Receive initial task
Assign new task	Notify master of task completion
Assign final task	Continue to work on existing task and notify master of task completion
Wait for all slaves to complete	Wait for all other slaves to complete
Receive results	Send results
End program	End program

Table 13: Master/Slave workflow

While it might be desirable to include the benchmarking process in the total application time, hashing speeds on the slowest test node was measured at approximately 500,000 hashes per second in the test environment. The benchmarking process consists of 5 million hashes, thus this process is no more than 10 seconds of the total application runtime. In the context of this application, 10 seconds is not significant. It is considered a constant cost that must be paid, thus as the problem scales up, the initial cost as a percentage of the total application time goes down.

The second synchronization point occurs after all of the tasks have been assigned and the master is waiting for the remaining slaves to finish working. This synchronization point is more significant than the first, but not overly so. As discussed in Section 4.7.4, the use of work slices that are based on a user defined time slice offer a guarantee that the processor idle time at the second synchronization point shall not exceed $N * t_{slice}$.

In reality, this idle time is likely to be lower. Tasks are assigned as work slices, consisting of a number of chains to be generated. Recall that a work slice is governed by

the following equation where t_{slice} is the user supplied time slice $S_{slowest}$ is the hashing speed of the slowest computer and *t* is the chain length.

$$W_{slice} \leq \frac{t_{slice} * S_{slowest}}{t}$$

The variable nature of $S_{slowest}$ and the requirement that W_{slice} be an integer results in an interesting situation where the final work slice is more than likely a partial work slice which will take less time to generate than a full work slice. In the worst case, the idle time may approach, but not exceed N * t_{slice}. This could happen if the work slices were evenly divisible by the chain length, and N-1 nodes completed all of their work immediately after the last work slice were assigned to the slowest computer in the system.

5.4. Error Handling

As with any application, there exists the potential for errors during execution. In a dedicated cluster, the likelihood of system failures is greatly reduced, but it is not eliminated. In a distributed heterogeneous environment where the nodes are not tightly controlled, such failures can be very common.

Many errors that can occur in a parallel environment are duplicates of problems that exist in a non-parallel environment. These include problems such as power outages, disk failures, other hardware failures, or bugs in the source code. In a single node environment, any one of these failures would be catastrophic and has the potential to cause the loss of all progress up to that point.

Other events can cause premature termination of the application, which might or might not be recoverable. These include problems such as running out of disk space or an

unanticipated reboot. While these issues are not necessarily catastrophic, they are made more complicated in a parallel environment. The single node environment can typically survive them and pick up where it left off. It merely needs to retrieve the current size of the table, calculate how big it is expected to be after completion, and then performs a simple subtraction to determine the number of chains left to generate.

Parallel applications in a heterogeneous environment have similar types of problems, but are not necessarily as quick or as easy to address. The catastrophic failure of a single node will virtually always cause the application to halt. The MPI specification does not provide for error handling or the loss of a node during execution. While fault tolerant forms of MPI such as MpiFL [23] do exist, they are not mainstream implementations and add additional overhead.

Check pointing is a common strategy for maintaining the progress of an application and in the case of rainbow table generation is likely the most feasible course of action. With this strategy, the system state is saved such that if a failure occurs, not all progress is lost. This is not currently implemented with PRTGen.

To implement check pointing would require the following. During the benchmarking process, the master node should instruct a single processor on each system to check the file system for the existence of temporary files matching the filename format that is about to be generated. Only one processor on each system should be instructed to do this, or multiple copies of the same intermediate rainbow table would be returned to the master node, thus decreasing the total number of chains and the efficiency of the table.

Once the data has been returned from all of the temporary tables, the files are deleted on each slave node. The master node aggregates all of the data, and subtracts the number of chains it has collected from the total chains to be created. The work slices are then calculated and the task assignments take place normally.

At the end of the rainbow table generation process, the results are then aggregated on the master node, taking care to start writing at the end of the file, rather than at the beginning. These are relatively straightforward steps, but were not implemented as fault tolerance was neither the goal, nor the main focus of this thesis. Program failures in the current implementation of PRTGen require that the application be restarted and that all progress to the point of the failure will be lost.

5.5. Platform Specifics

The current implementation of PRTGen uses version 1.0.5 of the MPICH 2 library from Argonne National Laboratory [22]. It was compiled using Microsoft's C++ compiler contained within Visual Studio 2005. The software was not tested on 64 bit operating systems. While no problems with 64 bit operating systems are anticipated, they cannot be ruled out at this time.

Some assembly language is used in the code to calculate the plaintext string given an index into the plaintext space. This requires the installation of the Microsoft Assembler (MASM32), which is also required for compiling OpenSSL. The OpenSSL source code is required to provide access to some of the hashing algorithms. These are currently linked to the application using external libraries specified in the compiler after OpenSSL has been compiled.

This implementation is primarily Windows based. While most of the code is cross platform capable, and the MPI library is certainly capable of cross platform compatibility, the application was implemented using a Windows based compiler, Windows API calls for determining the computer name, and the Microsoft Assembler for the Assembly language code.

5.6. Conclusion

The PRTGen application closely follows the algorithm set forth by Philip Oechslin in 2003 for creating rainbow tables. Although it is a Windows only application, the majority of researchers are expected to have access to a heterogeneous Windows environment more readily than a homogeneous dedicated cluster of any given platform.

The communications architecture used by the application is a farming technique. This technique allows the user to guarantee minimal processor idle time, thus achieving very high application efficiency. The network overhead of using MPI is expected to be small in comparison to the size of the problems being solved. While some amount of overhead is required for the parallel implementation, it is an acceptable loss considering the potential gain of the entire system.

There exists room for future improvement in the area of fault tolerance. The current implementation is not fault tolerant in any way. The loss of any node in the system will cause a failure that may not be recovered from. With the potential gain, the application could be restarted several times and would still be finished faster than a single processor implementation.

Chapter 6 Results and Analysis

This chapter summarizes the results of the parallel implementation in three different scenarios. Several reference nodes are compared against each other and against the parallel system for each scenario. These comparisons illustrate the usefulness of a parallel implementation over using faster computers.

First, we examine a scenario of using extremely outdated hardware for generating the rainbow tables. Next, we examine a scenario of using modern hardware for generating a rainbow table that is four times more work to generate than that of the first scenario. Finally, we examine the results of using a parallel implementation to generate both tables to illustrate the effectiveness of a parallel implementation.

6.1. Reference Nodes

Twelve computers were used to test the implementation of PRTGen that is the focus of this thesis. These twelve computers consisted of a total of seven different hardware configurations and a total of twenty-five processors. The age of the hardware ranges from approximately eight years old to less than one year old. Many corporations perform what is called a “hardware refresh” every three years, so it is not typical to have hardware this old in anything but a development environment or testing lab.

The use of such hardware in this implementation demonstrates the value of older hardware and its viability to outperform a single node implementation on high end systems. A full cost analysis is not provided, but can be estimated based on the results.

The hardware specifications for the twelve nodes used in this implementation are detailed in Table 14. Benchmark information is provided for each of these nodes follows in Table 15.

Platform ID	Computer Name	CPU Manufacturer	CPU Type	# of CPU's	CPU speed (in GHz)	System RAM (in GB)	Disk Speed (RPM)
1	inspiron8100	Intel	Pentium III	1	1.0	0.5	4,200
2	lpjwandke	Intel	Core 2 Duo	2	2.2	3.0	7,200
3	mrsrack01	AMD	Athlon XP	2	2.2	4.0	7,200
3	mrsweb01	AMD	Athlon XP	2	2.2	4.0	7,200
4	svAltiris	Intel	Xeon	4	2.2	4.0	15,000
5	svgtserver	Intel	Pentium III	2	1.0	1.0	7,200
6	svsm5	Intel	Xeon	2	2.4	2.0	15,000
6	svsm6	Intel	Xeon	2	2.4	2.0	15,000
6	svsm7	Intel	Xeon	2	2.4	2.0	15,000
6	svsm8	Intel	Xeon	2	2.4	2.0	15,000
6	svSymantec	Intel	Xeon	2	2.4	2.0	15,000
7	wsmtaber	AMD	Athlon XP	2	2.6	2.0	7,200

Table 14: Reference hardware specifications

The Platform ID identifies unique hardware configurations. Computers that share a platform ID also share a hardware configuration. There are some slight differences in some cases. For example, platform #7 is almost identical to platform #3, but has a faster processor and less system memory. The two computers identified as platform #3 have slightly different RAID configurations. Similarly, the computers in platform #6 also have different RAID configurations.

In general, the differences between computers that have been classified with the same platform ID are minor. Based on the analysis of hashing speed in Chapter 4.5, the architecture of the processor is the single most significant factor in hashing speed. For this reason, minor variations are treated as if they were insignificant.

Computer	CPU #	Node Hashing Speed (per second)	System Hashing Speed
inspiron8100	1	453,338	453,338
lpjwandke	1	1,241,852	2,486,858
	2	1,245,006	
mrsrack01	1	976,924	1,980,844
	2	1,003,920	
mrsweb01	1	1,081,167	2,163,420
	2	1,082,253	
svAltiris	1	1,233,861	4,910,891
	2	1,226,074	
	3	1,221,623	
	4	1,229,333	
svgtserver	1	463,549	927,543
	2	463,994	
svsm5	1	626,593	1,253,631
	2	627,038	
svsm6	1	626,228	1,254,952
	2	628,724	
svsm7	1	628,724	1,244,328
	2	622,566	
svsm8	1	625,783	1,251,594
	2	625,812	
svSymantec	1	573,305	1,145,803
	2	572,498	
wsmtaber	1	2,571,459	5,156,475
	2	2,585,016	

Table 15: System benchmarks

The hashing speeds listed in Table 15 are based on the MD5 algorithm. The hashing speed shown is an average of three separate benchmark tests that were run on each computer. Computers with similar hardware platforms report similar results, but there is certainly some variation between them.

One notable example is the “svSymantec” computer, which has an average hashing speed that is below that of other computers with an identical hardware configuration by approximately 10%. This particular computer is running Enterprise

security software on the network. It is more heavily loaded than the other the other computers; therefore its performance is also lower. Recall from Chapter 4.5 that decreased available memory can reduce the performance of a computer by approximately 10%. In this case, the decreased available memory on the system also decreases the performance of the system in regards to hashing speed.

These computers are connected by a high speed 1 Gbps network, with 1 Gbps network cards. The network cards are of various models, but are all built into the motherboards of the computers. No special Ethernet cards were used. The network switch used was a Linksys SR2024 10/100/1000Mbps 24 port Gigabit switch.

6.2. Test Scenario 1

The first scenario that we consider directly evaluates the slowest computer in the cluster over an extended period of time. For the purpose of this thesis, an extended period of time is arbitrarily defined as 6 hours. This is a large enough sample size that it is useful for evaluating the effectiveness of the parallel implementation, yet small enough such that multiple tests may be executed in a reasonable amount of time.

From Table 15, “inspiron8100” is the slowest computer in the cluster. It is a laptop that is more than 8 years old with limited memory and a slow hard drive. This computer is chosen for this scenario to illustrate the effectiveness of a parallel implementation when outdated hardware is used for the task. The application parameters are shown in Table 16.

Parameter	Setting
Algorithm	MD5
Character set	Alpha
Minimum plaintext length	1
Maximum plaintext length	8
Chain Length	10,000
Number of Chains	1,000,000

Table 16: Application Parameters for Scenario 1

The benchmarking process estimates that this task will take approximately 22,112 seconds on Inspiron8100. Converting this to a more easily understood time format, the table will take approximately 6.04 hours to generate. The results of this test are contained in the following table.

Time	Estimated Time	Measured Time	Data Aggregation
Seconds	22,112.0062	22,061.0625	1.2120
Minutes	368.5334	367.6844	0.0202
Hours	6.0415	6.0276	0.0003

Table 17: Scenario 1 speed test results

These results show that the estimated time calculated by the benchmarking process is marginally different than the measured time by approximately 0.231%. The reference computer in this scenario was not being used for any other work at the time of this test. In addition, very little software was active on the computer. This test shows that with no outside interference, the benchmarking process is fairly accurate for a single computer.

This test was set up with a master node and a slave node running on the same processor. In a large parallel implementation, this is not ideal as the master node is likely to eventually become a bottleneck as the size of the problem scales up. In this scenario, it

is acceptable as the overhead associated with having another process provide 369 tasks over the course of 6 hours is minimal.

The Data Aggregation column shows the time it takes the slave node to transmit the results over the network to the master node. This uses the local loopback interface in this scenario because both the master and slave nodes are on the same computer. The total size of the rainbow table generated by this scenario is 16 million bytes, or a little less than 16 MB. The speed seems to be a little slower than gigabit Ethernet, but this is to be expected as it is transmitting the data to itself and must handle disk I/O overhead for all reads and writes, in addition to both sending and receiving the data.

6.3. Test Scenario 2

The second scenario that we consider directly evaluates the most modern computer in the cluster over an extended period of time. Again, for the purpose of this thesis, an extended period of time is arbitrarily defined as 6 hours.

From Table 15, “svAltiris” is the most modern computer in the cluster. It is a quad core rack mounted server with very fast hard drives and more than a reasonable amount of memory. This computer is chosen for this scenario to illustrate the effectiveness of a parallel implementation when modern hardware is used for the task. The application parameters are shown in Table 18: Application Parameters for Scenario 2. Only one slave process was used to perform the work.

Parameter	Setting
Algorithm	MD5
Character set	Alpha
Minimum plaintext length	1
Maximum plaintext length	8
Chain Length	40,000
Number of Chains	1,000,000

Table 18: Application Parameters for Scenario 2

Note that the main difference between this case and test Scenario 1 is the chain length is quadruple that of Scenario 1. It is otherwise identical. The chain length was quadrupled to increase the amount of work. The benchmarking process estimates that this task will take approximately 20,624 seconds on svAltiris. Converting this to a more easily understood time format, the table will take approximately 5.6 hours to generate. The results of this test are shown in Table 19.

Time	Estimated Time	Measured Time	Data Aggregation
Seconds	20,624.0104	20,632.0100	1.0427
Minutes	343.8668	343.8668	0.0174
Hours	5.6350	5.6372	0.0003

Table 19: Scenario 2 speed test results

The results of Scenario 2 are very similar to the results from Scenario 1. The main difference is that the measured time is slower than the estimated time by 0.04%. In the first scenario, the measured time was faster than the estimated time. This can be attributed to the variable requirements of other applications running on the computer during the benchmark testing. This was a production server running a database and various client management tools. It was expected that it might be slower than the

estimated time. Without any additional messaging overhead associated with a parallel scenario, this is expected to be fairly accurate.

The Data Aggregation time is slightly lower than in Scenario 1. The reference computer used in this scenario has a raw disk speed that is about 3 times faster than that of the reference computer from Scenario 1. In addition, the master and slave processes on this reference computer are likely running on different processors which would reduce context switching in the processor during the data transfer.

6.4. Test Scenario 3

This scenario uses the same application parameters as Scenario 1, but with the addition of multiple nodes in the system to speed up the implementation. The application parameters are the same as in Table 16. One additional setting required in a parallel scenario is a time slice.

Six different time slices are used to help formulate conclusions based on the data. All of the systems used in this scenario are outlined in Table 14. The measured application times are shown in Table 20 and illustrated in Chart 2. Processor working times, idle times, and wait times for each of the time slices are listed in Tables A1 through A6 which may be found in Appendix A.

	Measured Time					
	Time slices (in seconds)					
	1	**5**	**10**	**15**	**30**	**60**
Estimated Time (in seconds)	317.7212	315.9138	317.3912	320.1802	317.7586	324.4991
Actual Time (in seconds)	340.4060	322.7650	324.1560	330.2190	345.1090	336.8130
Actual Data Aggregation Time (in seconds)	0.125	0.1250	0.1250	0.1250	0.1250	0.1250
Number of tasks	19,231	3,803	1,909	1,268	636	335
Tasks/second	56.49	11.78	5.89	3.84	1.84	0.995

Table 20: Scenario 3 speed test results

The time to aggregate the rainbow tables back on the master node did not change, regardless of the time slice used. This is expected because the time slice is only used during generation of the tables, not in the aggregation phase.

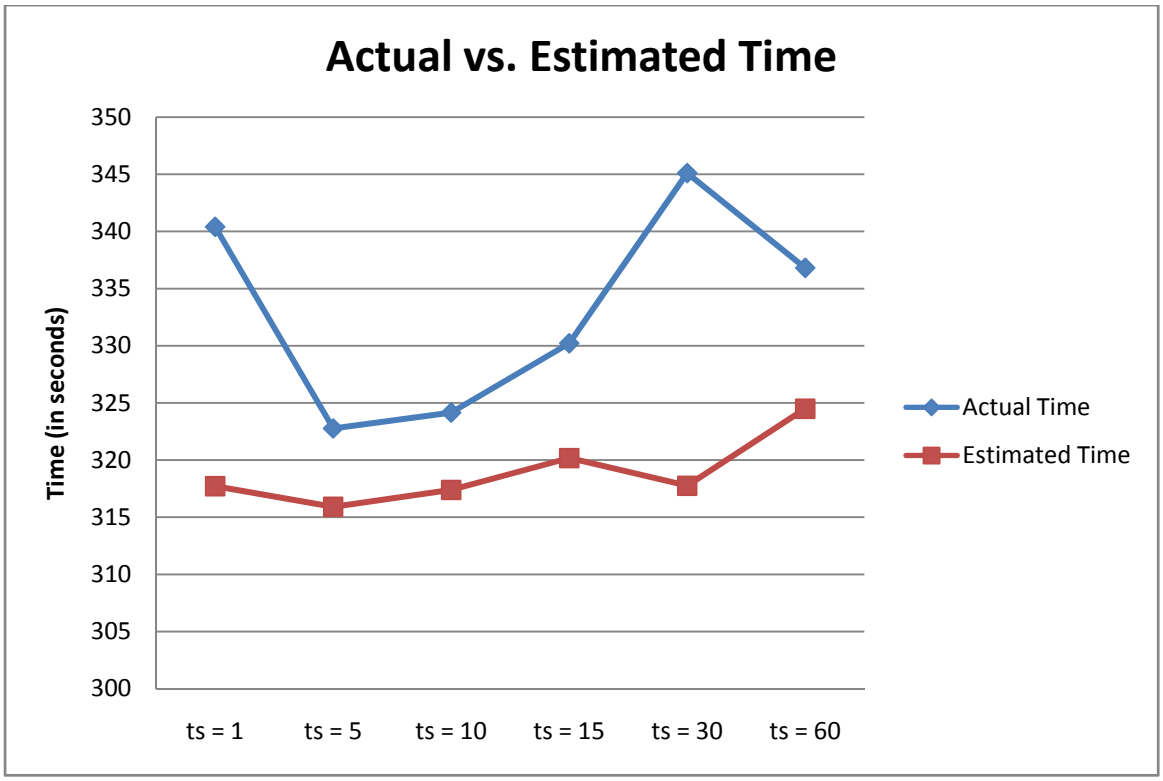


Chart 2: Actual vs. Estimated Time of different time slices in Scenario 3

The time measurements in Chart 2 show that a time slice of 1 is not the optimal solution with regards to solving the problem in the least amount of time. This is attributed to the number of messages being sent per second. In the case of time slice 1, there is an average of five times more messages per second than with a time slice of 5.

The additional messages sent per second cause the processor handling this load to become a bottleneck in the system when the time slice is too low. From Tables A1 – A6, we can review both the Total Idle Time and the Total Waiting Time against the time slice being used. This is seen in Chart 3, Chart 4, and Chart 5.

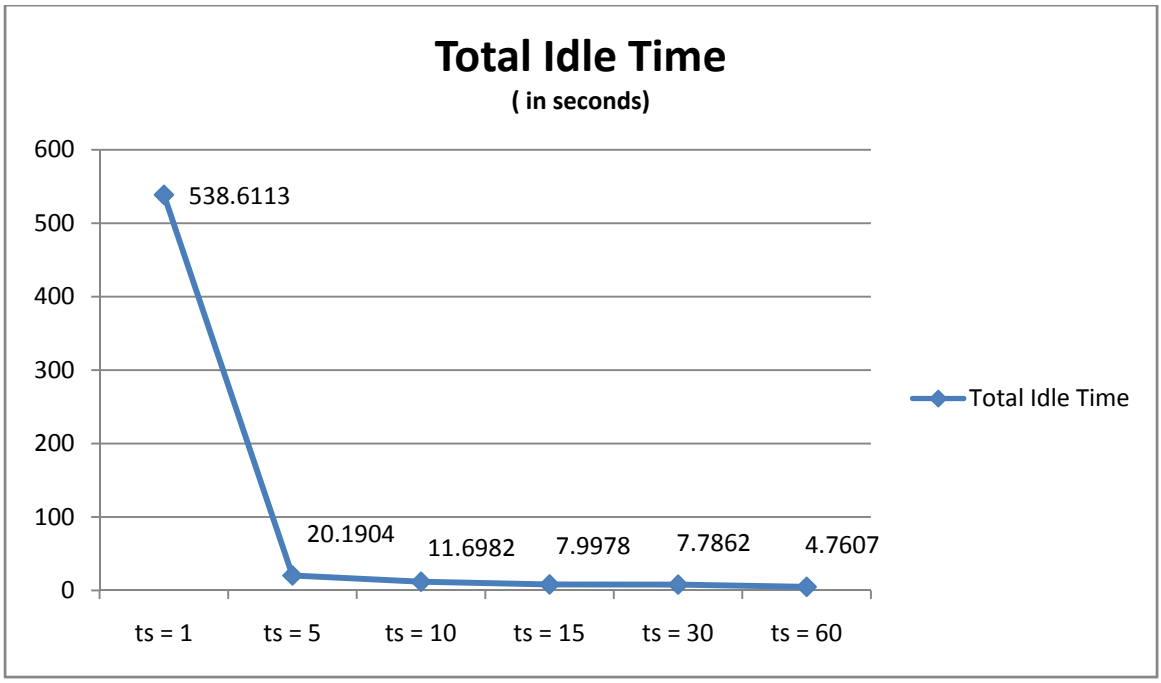


Chart 3: Total Idle time of processes in Scenario 3

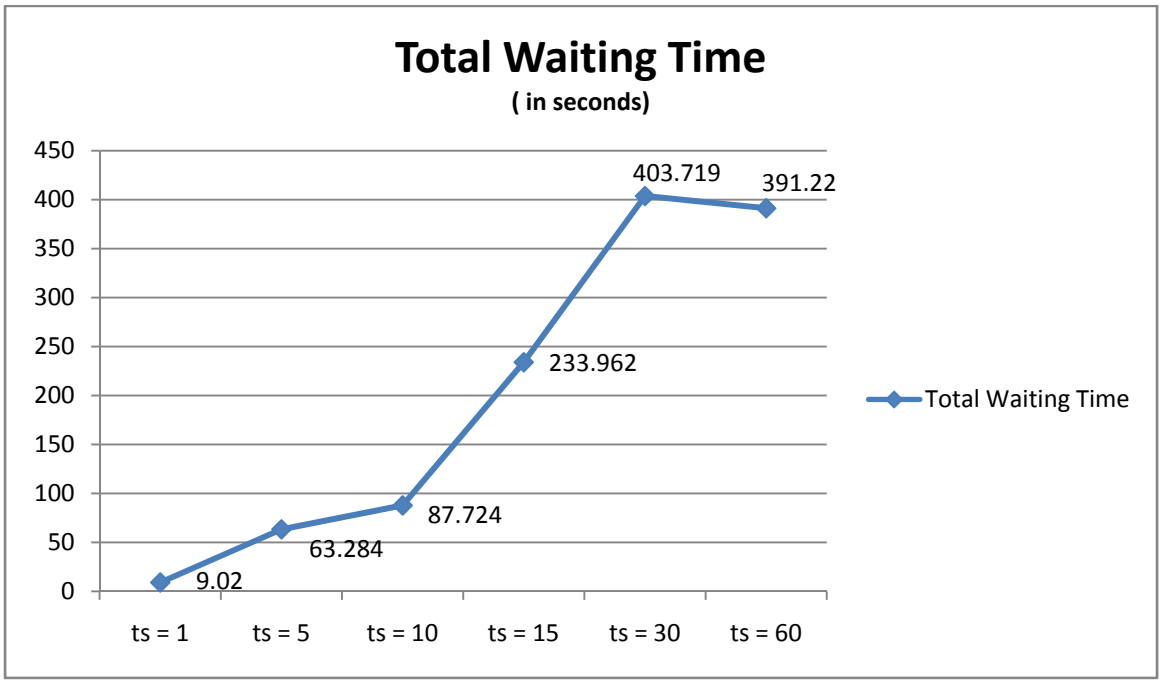


Chart 4: Total waiting time of processes in Scenario 3

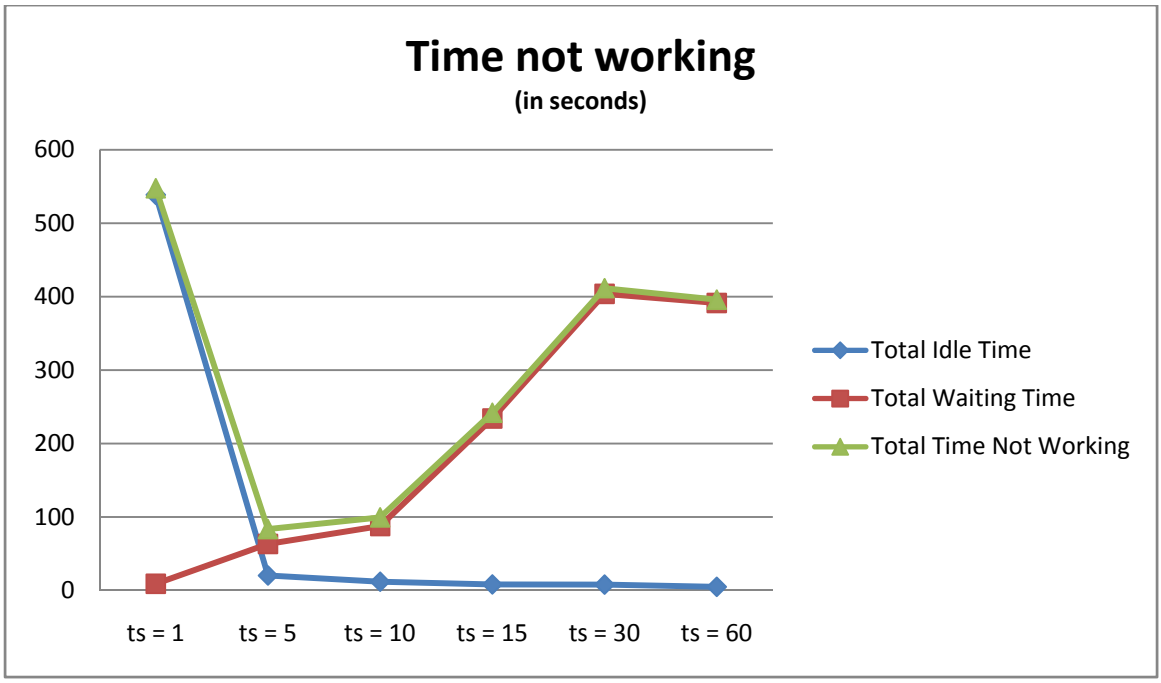


Chart 5: Time not spent working for processes in Scenario 3

The idle time for a process is defined as any duration that a node is not generating hashes as part of a task when there are still tasks to be assigned. Understandably, this includes time spent requesting new tasks from the master node over the network. Once there are no more tasks that may be assigned, a different counter is used to track the time that a process is waiting for the entire job to finish. This is called the waiting time.

The similarities between Chart 2 and Chart 5 are unmistakable. From these charts, we can see that with a time slice of 1 second, there are so many messages being sent that the slave nodes are spending a lot of time waiting for the master node to respond to their requests. The fastest computer is requesting tasks at a rate of more than 10 per second.

Clearly in this limited test case, the problems are not so severe that the application does not execute. The task runs approximately 18 seconds slower than if a time slice of 5 were used. This is a performance reduction of approximately 5%.

However, if we scale up the number of computers or decrease the time slice even further, the problem will be compounded in direct proportion to the scaling we apply. If the time slice is halved, the number of messages per second will double. If we double the hashing capabilities of the entire system without introducing a computer slower than the slowest node, then the number of messages will double.

It is obvious from these simple calculations that although the system still runs well enough with a time slice of 1 second, it is by no means ideal. Should the network be flooded with additional traffic outside of this application, the performance would suffer more.

6.5. Test Scenario 4

Scenario 3 was the parallel implementation of Scenario 1. Scenario 4 is the parallel implementation of Scenario 2. The application parameters are the same as Scenario 2 and can be found in Table 18: Application Parameters for Scenario 2. As with the previous parallel implementation, the time slice is examined as the primary factor in performance and efficiency.

The set of systems used in this scenario are the same as the previous parallel implementation and are outlined in Table 14. The measured application times are shown in Table 21 and illustrated in Chart 6. Processor working times, idle times, and wait times for each of the time slices are listed in Tables B1 through B6 which may be found in Appendix B.

	Measured Time					
	Time slices (in seconds)					
	1	**5**	**10**	**15**	**30**	**60**
Estimated Time (in seconds)	1,281.5616	1,303.1301	1,271.0019	1,325.4266	1,280.8504	1,305.1063
Actual Time (in seconds)	1,369.6870	1,298.4530	1,284.9690	1,294.1560	1,302.0780	1,319.0460
Actual Data Aggregation Time (in seconds)	0.1250	0.1250	0.1250	0.1100	0.1250	0.5310
Number of tasks	76,924	15,385	7,634	5,587	2,434	1,220
Tasks/second	56.16	11.89	5.94	4.32	1.87	0.93

Table 21: Scenario 4 speed test results

The data aggregation time includes some slight variances, especially in the last case. However, the data aggregation results are very similar to those seen in Table 20 for Scenario 3. As a percentage of the total time, this is a mere 0.04% and is relatively inconsequential.

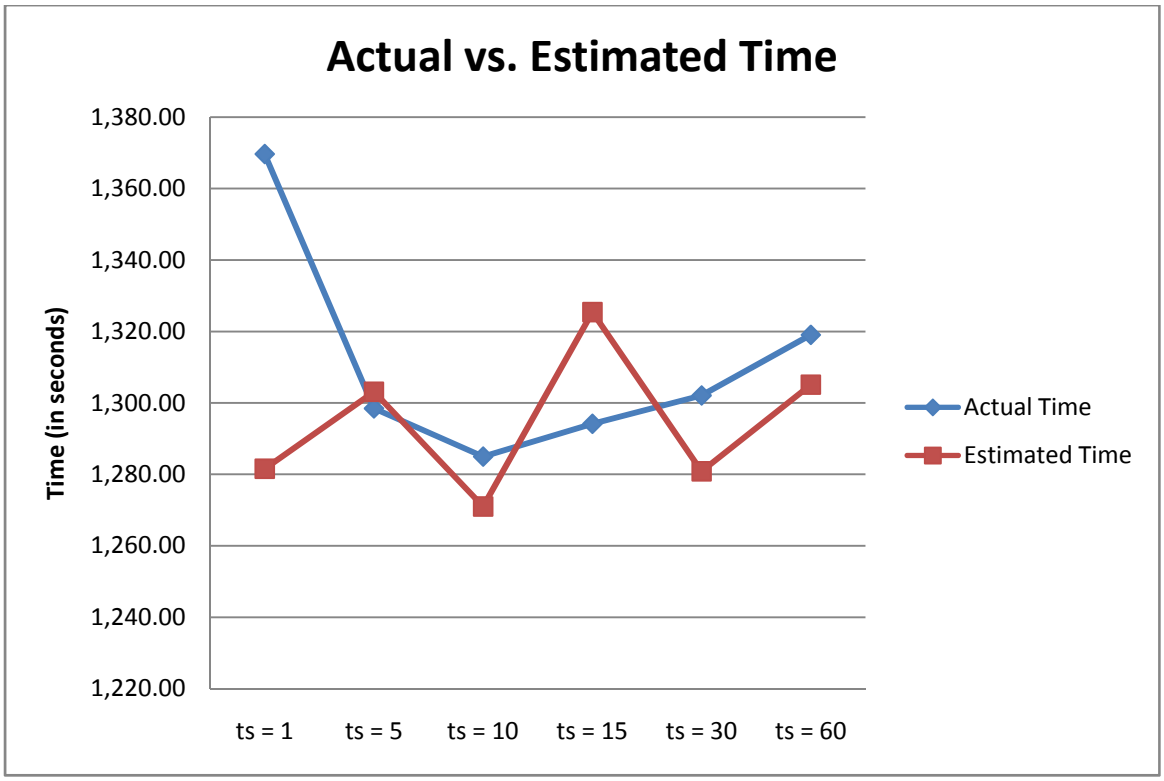


Chart 6: Actual vs. Estimated Time of different time slices in Scenario 4

The time measurements in Chart 6 illustrate that the estimated time is not always accurate. It bears little resemblance to the estimated results of Scenario 3. As this is merely an estimation based on an extremely short benchmarking process, this is not a cause for concern. The average of all six estimated times falls beneath the lowest point of the Actual Time on the graph.

The actual time follows a pattern that is identical to the parallel implementation of Scenario 3. As before, it shows that a time slice of 1 is not the optimal solution with regards to solving the problem in the least amount of time. This is directly caused by the number of messages being sent per second.

What is interesting is that in Scenario 4, we quadrupled the length of the chain to quadruple the amount of work. The total time to completion also roughly quadrupled, but

the number of tasks per second stayed relatively constant. This is a direct result of using a time slice methodology for determining task size.

Additional data points from each of these tests can be found in Tables B1 – B6. There, the Total Working Time, Total Idle Time and the Total Waiting Time are listed for each time slice being used. The aggregated information can be seen in Chart 7, Chart 8, and Chart 9.

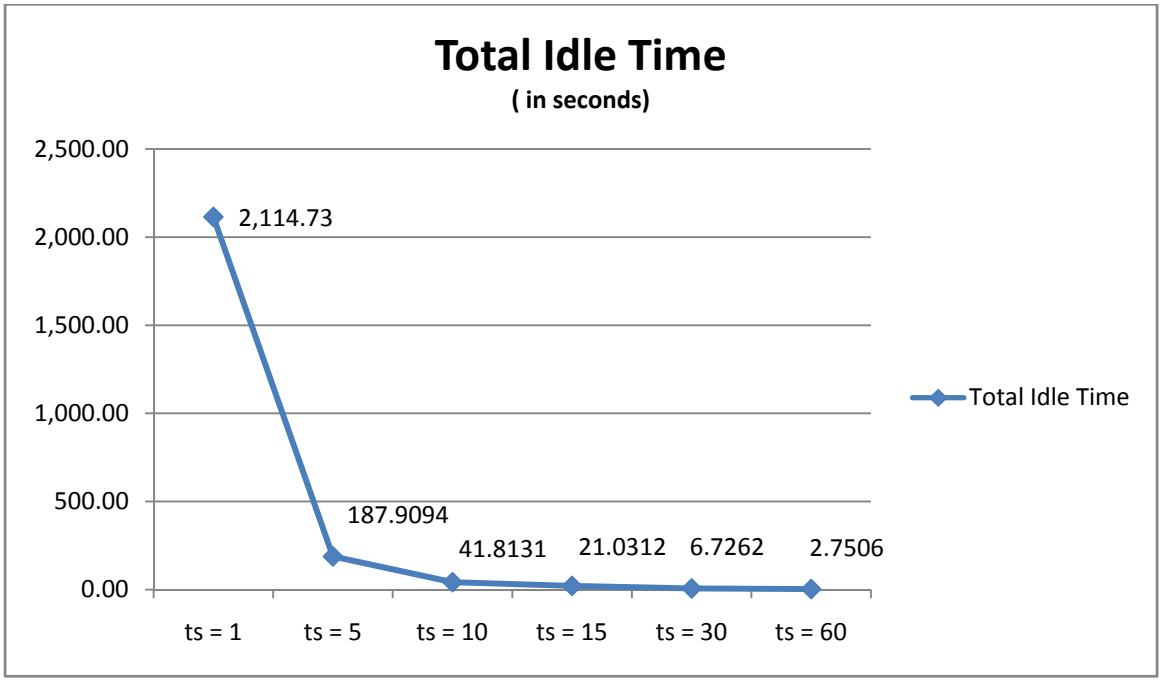


Chart 7: Total Idle time of processes in Scenario 4

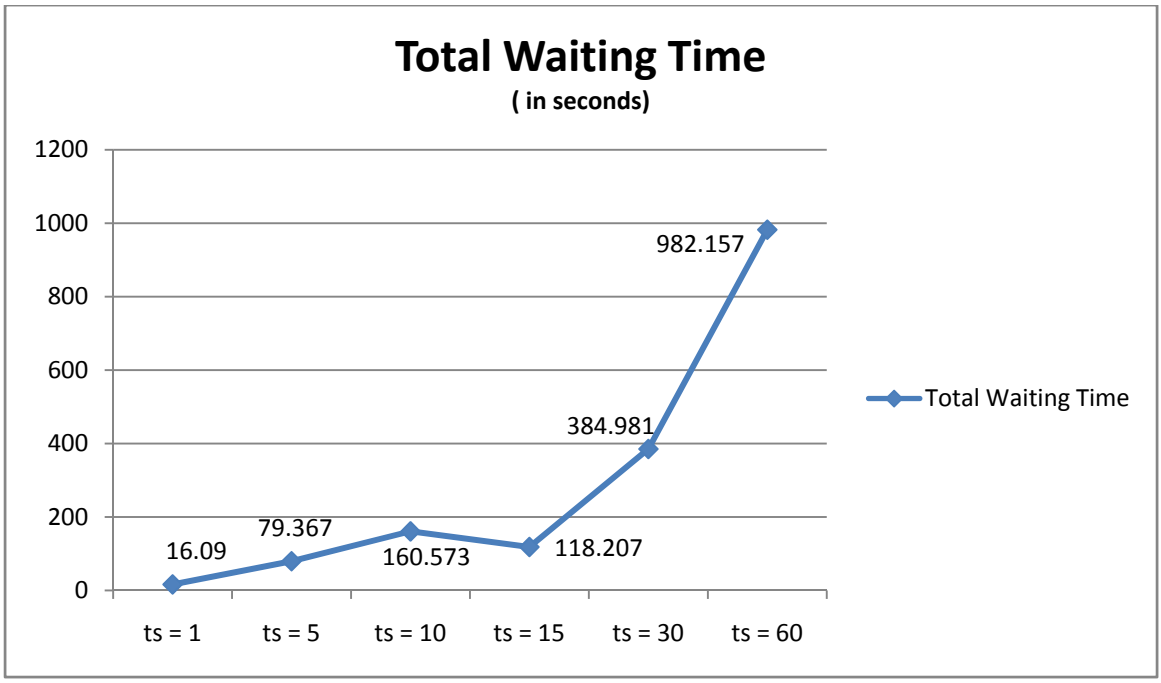


Chart 8: Total waiting time of processes in Scenario 4

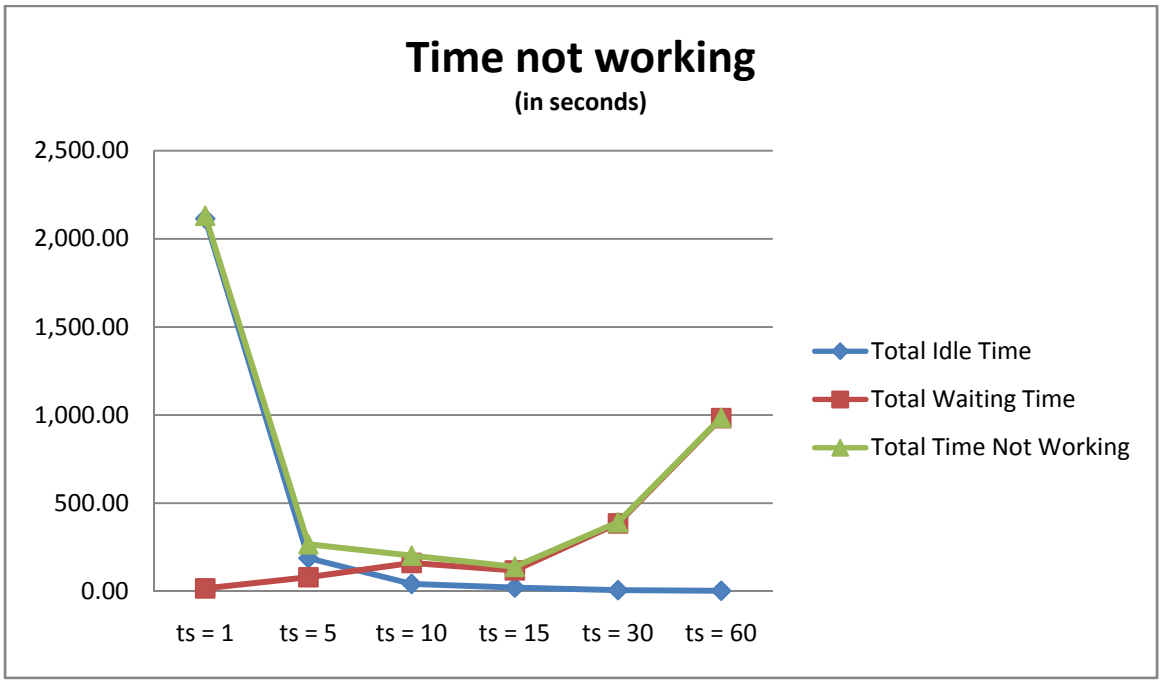


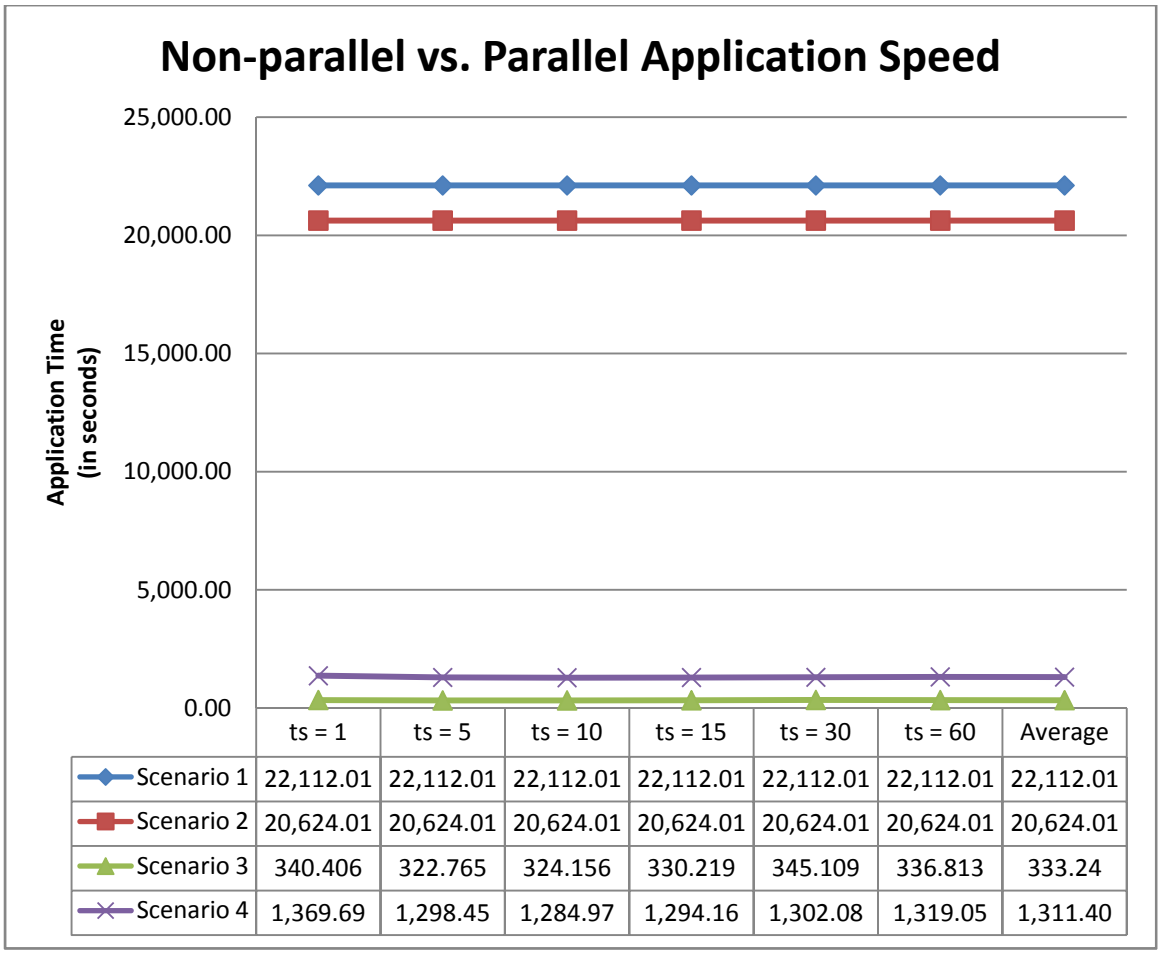
Chart 9: Time not spent working for processes in Scenario 4

The idle time for a process is defined as any duration of time that a node is not generating hashes as part of a task when there are still tasks to be assigned. Understandably, this includes time spent requesting new tasks from the master node over the network. Once there are no more tasks that may be assigned, a different counter is used to track the time that a process is waiting for the entire job to finish. This is called the waiting time.

Ignoring the Estimated Time from Chart 6, the similarities between Chart 6 and Chart 8 closely mimic the similarities between Chart 2 and Chart 6. Once again, it is clear that if the time slice is too small, there are so many messages being sent that the slave nodes are spending a significant amount of time waiting for the master node to respond to their requests.

6.6. Non-Parallel vs. Parallel Scenarios

Comparing the parallel implementations to the corresponding single node implementations, there is a clear advantage in terms of computational time to using a parallel application to perform the work. This is illustrated by Chart 10.



	ts = 1	ts = 5	ts = 10	ts = 15	ts = 30	ts = 60	Average
Scenario 1	22,112.01	22,112.01	22,112.01	22,112.01	22,112.01	22,112.01	22,112.01
Scenario 2	20,624.01	20,624.01	20,624.01	20,624.01	20,624.01	20,624.01	20,624.01
Scenario 3	340.406	322.765	324.156	330.219	345.109	336.813	333.24
Scenario 4	1,369.69	1,298.45	1,284.97	1,294.16	1,302.08	1,319.05	1,311.40

Chart 10: Non-Parallel vs. Parallel application speed

Time slices were not used in the non-parallel case, due to the nature of the non-parallel case and the fact that only one process will be performing the work, the application speed can be assumed to be constant regardless of the time slice.

The data in Chart 2 and Chart 6 show a clear advantage to decreasing the number of messages by using a time slice of 5 seconds within the parallel case, but the high level nature of Chart 10 makes these differences virtually invisible. We have clearly ascertained that the time slice of 5 seconds will yield the best time in both parallel cases, thus we use this time to calculate the speedup factor. The direct application of the

speedup calculation using Amdahl's Law is not possible, due to the prerequisites [15] placed on using it and the difficulty in making the required measurements.

However, it is possible to calculate an approximation of the speedup by comparing the parallel case against the non-parallel case. Using the time slice of 5 seconds, we find that Scenario 3 requires only 1.46% of the time of the non-parallel case, and Scenario 4 requires only 6.30% of the corresponding non-parallel case. These correspond to approximate speedup factors of 68.5 times faster for the first parallel case, and 16 times faster for the second parallel case.

The apparent discrepancy in speedup factors can be explained by reviewing the speed of the reference computer and the size of the problem in each scenario. Recall that to compensate for the speed differences between the computers in Scenario 1 versus Scenario 2 we quadrupled the size of the problem. If we divide the speedup factor achieved the first parallel case by four, we see a recognized speedup of approximately 17.1 times faster.

To verify this, additional tests were run using the settings from Scenario 1 on the reference computer from Scenario 2. This resulted in an application run time of 6,004.0029 seconds. The parallel case is only 5.4% of this time, which calculates to an approximate speedup factor of 18.5.

Both of these methods are approximations, and show that given a similar set of circumstances, the speedup will be virtually identical, as Amdahl's law would indicate. A parallel scenario is going to have additional messaging overhead and as a result, will skew the calculation to some degree. Variances in hashing speed from one minute to the

next due to any number of external factors cannot be reasonably accounted for prior to runtime.

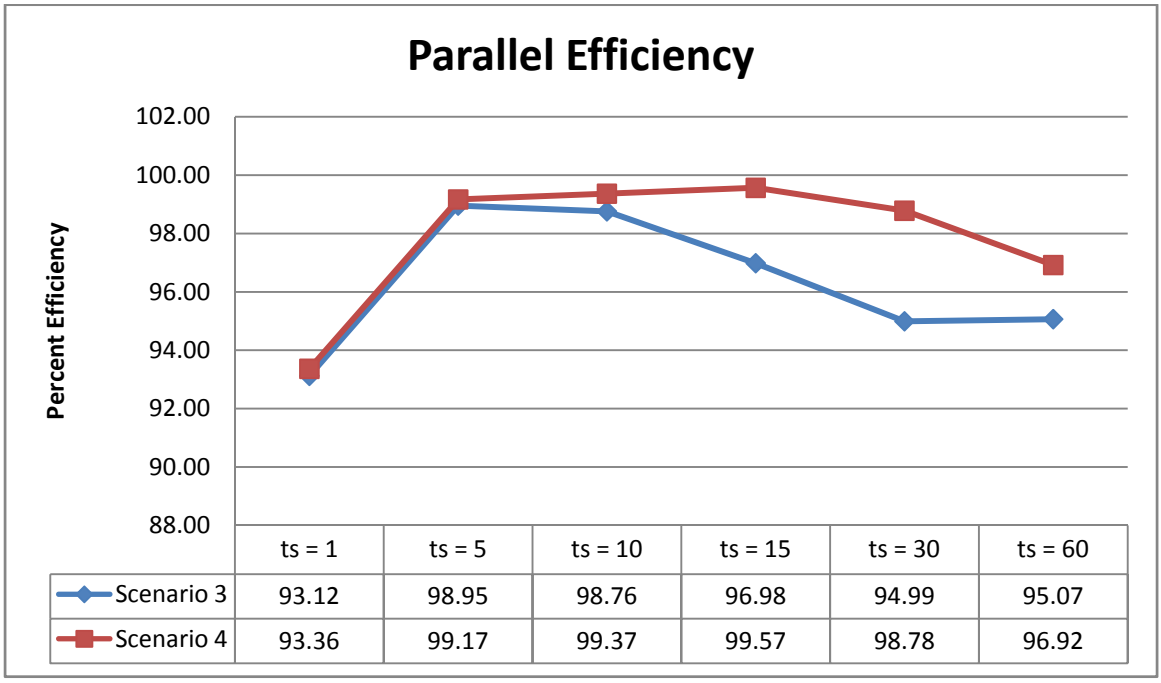
Another way of looking at it would be to compare the application times of Scenario 3 and Scenario 4. Given that Scenario 4 requires four times as many calculations as Scenario 3, we can quadruple all of the times from Scenario 3 and we should end up with approximately the same results from identical time slices as in Scenario 4. The table below shows this calculation, further illustrating the validity of the speedup factor.

	ts = 1	**ts = 5**	**ts = 10**	**ts = 15**	**ts = 30**	**ts = 60**
Scenario 3	340.406	322.765	324.156	330.219	354.109	336.813
Scenario 4	1,369.69	1,298.45	1,284.97	1,294.16	1,302.08	1,319.05
Scenario 3 (timings quadrupled)	1,361.624	1,291.06	1,296.624	1,320.876	1,404.436	1,347.252
Scenario 4 minus 4x(Scenario 3)	8.066	7.39	-11.654	-26.716	-102.356	-28.202

Table 22: Scenario 3 vs. Scenario 4 speed comparison

Reviewing the results reveals that the estimated time is far more accurate for the non-parallel implementation than for the parallel implementation. There are a few different reasons for this. Although both sets of tests used the same code base, the master nodes in the parallel implementation were constantly responding to requests. In the non-parallel cases, the master node was responding to the same slave node over and over. When the slave was ready to request more work, there was virtually no wait involved because the master had no other slaves to service requests for. This speeds up the implementation, and makes the benchmarking more accurate in the non-parallel scenario.

In the parallel scenario, additional idle time is introduced into each slave whenever overlap occurs between two slaves making a request to the master node. Chart 3 and Chart 7 show that the more messages per second are being sent through the master node, the higher the probability of overlap and the more it will impact the total application time.



	ts = 1	ts = 5	ts = 10	ts = 15	ts = 30	ts = 60
Scenario 3	93.12	98.95	98.76	96.98	94.99	95.07
Scenario 4	93.36	99.17	99.37	99.57	98.78	96.92

Chart 11: Efficiency of the Parallel Implementation

Overall, the implementation achieved a high efficiency in both parallel scenarios. From Chart 11, we can see the efficiency of the parallel implementation using various time slices. The efficiency is directly related to the application execution time. The lower the execution time is, the higher the efficiency rating. Note that the efficiency of the application in Scenario 3 reaches its peak when the time slice is set to 5 seconds. Table

21 and Chart 1 indicate that this is also the point at which the execution time is lowest for Scenario 3.

Similarly, the efficiency of the application in Scenario 4 reaches its peak when the time slice is set to 15 seconds. Table 21 and Chart 6 would be expected to indicate that the execution time would be lowest here, but it does not. If we review the results shown in Chart 8 and Chart 9, we see that although the execution time is slightly higher, the amount of time spent doing work is also higher. The lowest execution time is achieved with a time slice of 10 seconds, but the highest level of efficiency is achieved when the time slice is 15 seconds.

This appears to be a classic tradeoff of efficiency for execution time. A parallel implementation of virtually anything is less efficient, but the rate at which work can be done exceeds that of using a single node, thus making the tradeoff worthwhile. When generating rainbow tables, processors are generally running at very close to 100% utilization, making them virtually worthless for doing any other work. In the case of rainbow tables, it would be worthwhile to choose a larger time slice if it results in a shorter execution time. Unfortunately, due to processor scheduling inconsistencies, it would be impossible to reliably predict the optimal time slice without generating the tables, thus defeating the purpose of predicting them.

6.7. Conclusion

The results outlined in this Chapter show that a parallel implementation for generating rainbow tables is clearly worth the effort. The efficiency of the parallel implementation reaches as high as 99% in some cases. This was accomplished by doing a thorough analysis of messaging techniques and focusing on optimizations that result in a very high efficiency.

This allows rainbow tables to be built in hours that would have previously taken days, or even weeks. The ability to drastically reduce the time it takes to generate a rainbow table has broad implications for the security industry. Rainbow tables are typically used when many passwords need to be decrypted, because the process of building a rainbow table is little better in terms of performance than a brute force attack for a single hash decryption. As a byproduct of this research, implementing a parallel brute force attack becomes a trivial exercise.

Chapter 7 Conclusions

This chapter summarizes the Parallel Rainbow Table Generator application that was implemented and outlines possible future enhancements. This chapter also provides an overview of the available features in PRTGen and the contributions of this research to the field.

7.1. Features of PRTGen

The Parallel Rainbow Table Generator that was implemented to demonstrate the effectiveness of this research provides several important features to the user, such as the following:

- Provides the ability to use PRTGen for multiple hashing algorithms
- Provides the ability to configure rainbow tables for different sets of plaintext character sets.
- Provides user configurable settings for modifying the chain length and the chain count, thus directly affecting the success rate.
- Provides a benchmarking tool that can be used to estimate the amount of work and the time it would take to generate a set of tables.
- Provides a simple to use, command line interface that has an extremely high efficiency and very low setup costs.
- Provides a time slice option, allowing for modifying the amount of communication between the master and slave nodes.
- Provides multiple data points for measuring the effectiveness of the generation process

7.2. Contributions to the Field

The major contribution of this thesis is the implementation of tool to generate rainbow tables using parallel techniques on commodity hardware. No previous research has been published that explored techniques for doing this. The main reason for the lack of research is that most researchers have focused on using pre-computed tables for addressing the shortcomings of the DES algorithm.

Since the DES algorithm was invented, it has been proven to be insecure and better encryption methods have been developed. Little consideration has been given to using the same techniques for cracking the DES algorithm in a manner that are applicable to larger scale problems. There is an upper limit on the complexity of solving any given encryption algorithm, but the DES algorithm is no longer considered to be a large scale problem.

Some research exists for building rainbow tables using specialized FPGA hardware, but requires extensive knowledge of both specialized hardware and the ability to program that hardware. The tool implemented as part of this research uses commodity computer hardware that is widely available and not dependent upon specialized hardware knowledge.

7.3. Areas for Future Work

Due to the extremely high parallel efficiency that can be garnered by modifying the time slice, it would appear that there is little room for improvement. This is not the case. There are a number of areas where additional research would be warranted and the results of that research could prove useful.

First, the implementation uses only a single master node. As the number of nodes trends higher, the total amount of communication required between the master and the slaves increases. The time slice can be adjusted to compensate for the increased communication to help maintain a high level of efficiency. However, as the problem scales higher, so must the number of computers used to generate the rainbow tables.

This poses a problem with only one master. Even with a dedicated processor servicing all of the slaves, there comes a point where the slaves spend more time waiting to receive instructions than they spend doing real work. More communication occurs when the time slice is set too low, but is also a function of the number of nodes in the environment.

There should exist an ideal number of nodes per master, and using multiple masters could be an extension of this work. Exploring this aspect would allow this research to become the cornerstone of an extremely large implementation of hundreds, or even thousands of nodes, working together on commodity hardware to generate rainbow tables.

A related area to research is the synchronization and waiting that must occur when the slave communicates with the master. The current implementation uses blocking sends and receives when communicating between the master and slave nodes. During the rainbow table generation process, there is an implicit expectation that all tasks with the exception of the last one are exactly the same size, measured in a number of chains to generate as part of that task.

One optimization could be to send a non-blocking request to the master for a new instruction. In the meantime, it could begin work on a new set of chains with the

anticipation that the new task is going to be the same size as the previous task. Some arbitrary amount of progress on the anticipated task could be made, after which the node checks back for the reply from the master node. If a reply is not found, then the slave node continues working on the anticipated task. If a reply is found, it is mapped to the current work being done, thus eliminating processing delays due to network latencies or a master node that is too busy to respond in a timely manner.

This would help to eliminate additional overhead that is introduced by forcing every slave node to request a task from the master and wait until it has received one before continuing. If the master instructs a slave that it is to do no more work but it has attempted to get ahead, then the time spent waiting would have been lost anyway. However, if the master provides the slave with a new task that is the same size as the old one, the latency time involved in communicating with the master can be put to good use, thus increasing the efficiency overall.

To aid in these additional research efforts, the reference parallel code base has been heavily commented to provide clarity with regard to the communications architecture. Depending on the specifics of the communication, it's possible for a node to use a blocking send or receive in the wrong place and be unable to continue, resulting in a process that will never complete. Additionally, code comments are provided to assist with compiling the code and information on the platform specific nature of this implementation have been provided, should others deem it suitable for porting to a non-Windows environment.

Finally, this implementation is just one of the four different components of a toolset for using rainbow tables. The next tool that is the most likely candidate for a parallel implementation is “rtcrack.exe”.

There are a number of mathematical equations that govern the speed at which a password may be decrypted using a rainbow table. These are typically accounted for prior to generating a set of rainbow tables. A table that exhibits a 100% success rate is not very effective if it takes years to find the hash that is associated with it. A parallel implementation of “rtcrack.exe” would help to mitigate this problem and would be quite useful if it were used in conjunction with PRTGen.exe.

7.4. Closing Remarks

The implementation of the parallel rainbow table generator was based on a non-parallel implementation. While this should have made the process easier, it was not. The code was poorly documented and was not designed for a parallel implementation. This introduced a number of design problems, mostly revolving around information that needed to be passed to different objects within the code. This was made more difficult by the MPI API, which does not have a straightforward mechanism for specifying parameters that might be different on specific machines, such as the paths to configuration files, path to the executable, or even the host name of the computer.

The complexity of analyzing the results was also underestimated. In addition to verifying that the application was functionally equivalent to the non-parallel reference code, special attention needed to be paid to whether or not the performance was as high as should have been expected. The implementation uses a timing mechanism which is

slightly imprecise. While it works for our purposes, it has a tendency to introduce rounding errors which in some cases, cannot be easily discerned from outright mathematical errors.

The benefits of this implementation are quite impressive. This research shows that using a myriad of commodity hardware ranging from very old to very new yields speedup multipliers that could not be reasonably achieved on a single computer, regardless of its age. This keeps the costs associated with generating rainbow tables very low, even in situations where the size of a desired rainbow table are beyond the capabilities of a single computer.

Bibliography

1. Martin Hellman. “A cryptanalytic time-memory trade off”. *IEEE Transactions on Information Theory*, IT-26: p.401-406, 1980.
2. D.E. Denning. “Cryptography and Data Security”, page 100. Addison-Wesley, 1982.
3. Philippe Oechslin. “Making a faster cryptanalytic time-memory trade-off”. 2003.
4. J.-J. Quisquater, Y.G. Desmedt, “Chinese Lotto as an Exhaustive Code-Breaking Machine”. *Computer* Volume 24, Issue 11, pages 14-22. 1991.
5. RFC 3607. “Chinese Lottery Cryptanalysis Revisited: The Internet as a Code-Breaking Tool”
6. QUISQUATER, J., AND STANDAERT, F. “Exhaustive key search of the DES: Updates and refinements”. SHARCS 2005 (2005).
7. W. Diffie and M. Hellman, “New directions in cryptography”, IEEE Transactions on Information Theory, volume IT-22: p.644-654, November 1976.
8. Jay L. Devore, “Probability & Statistics for Engineering and the Sciences”, Duxbury Thomson Learning, 2000.
9. E. H. McKinney “Generalized Birthday Problem”, American Mathematical Monthly 73, 385-387, 1966.
10. Martin Hellman. “A cryptanalytic time-memory trade off”. *IEEE Transactions on Information Theory*, IT-26: p.402, 1980.
11. Koji Kusuda and Tsutomu Matsumoto. “Achieving higher success probability in time-memory trade-off cryptanalysis without increasing memory size”. *TIEICE: IEICE Transactions on Communications/Electronics/Information and Systems*, 1999.
12. Johan Borst, Bart Preneel, Joos Vandewalle, “On the Time-Memory Tradeoff Between Exhaustive Key Search and Table Pre-computation ”, Proceedings of the 19th Symposium on Information Theory in the Benelux, Veldhoven(NL), p 111-118, 1998.
13. Elad Barkan, Eli Biham, and Adi Shamir, “Rigorous Bounds on Cryptanalytic Time/Memory Tradeoffs”. 2006
14. Amdahl, G (April 1967) “The validity of the single processor approach to achieving large-scale computing capabilities”. *Proceedings of a FIPS Spring Joint Computer Conference*, Atlantic City, NJ. AFIPS Press, p: 483 – 485.
15. John L. Gustafson, “Reevaluating Amdahl’s Law”, 1988.
16. Yuan Shi, “Reevaluating Amdahl’s Law and Gustafson’s Law”, 1996.
17. Kevin Hammond, Hans Wolfgang Loidl, Andrew Partridge. “Visualizing Granularity in Parallel Programs: A Graphical Winnowing System for Haskell”, *High Performance Functional Programming Conference*, April 1995.
18. J. Waldo, G. Wyant, A. Wollrath and S. Kendall, "A Note on Distributed Computing", *Technical Report SMLI TR-94-29*, Sun Microsystems, November, 1994.
19. Williamn Hui, Steve MacDonald, Jonathan Schaeffer, and Duane Szafron. "Visualizing Object and Method Granularity for Program Parallelization", 2000.

20. Quisquater Jean-Jacques, Standaert Francois-Xavier, Rouvroy Gael, David Jean-Pierre, Legat Jean-Didier, “A Cryptanalytic Time-Memory Tradeoff: First FPGA Implementation, 2002.
21. Gary Shao, Francine Berman, Rich Wolski, “Master/Slave Computing on the Grid”, 2000.
22. MPICH 2, http://www.mcs.anl.gov/research/projects/mpich2/
23. Nuno Fonseca, Joao Gabriel Silva. “MPI Farm programs on non-dedicated clusters”, 2003.

Appendix A

Process ID	Hostname	Working Time (in seconds)	Idle Time (in seconds)	Waiting Time (in seconds)
1	svAltiris	337.5040	2.3080	0.5940
2	svAltiris	339.3630	0.5740	0.4690
3	svAltiris	338.2020	1.7510	0.4530
4	svAltiris	337.9540	2.1390	0.3130
5	Inspiron8100	328.8740	11.4220	0.1100
6	lpjwandke	319.1410	20.6870	0.5780
7	lpjwandke	329.6380	10.1740	0.5940
8	mrsrack01	311.0090	28.8340	0.5630
9	mrsrack01	309.6810	30.1940	0.5310
10	mrsweb01	309.8730	30.1270	0.4060
11	mrsweb01	309.6190	30.3810	0.4060
12	svgtserver	334.8450	5.5610	0.0000
13	svgtserver	334.3790	6.0270	0.0000
14	svsm5	310.5770	29.5320	0.2970
15	svsm5	310.8510	29.2580	0.2970
16	svsm6	312.2240	27.8850	0.2970
17	svsm6	311.7670	28.3420	0.2970
18	svsm7	321.5110	18.6450	0.2500
19	svsm7	320.7380	19.0580	0.6100
20	svsm8	312.4500	27.6590	0.2970
21	svsm8	311.5380	28.5710	0.2970
22	svSymantec	318.8010	21.4640	0.1410
23	svSymantec	319.6330	20.2730	0.5000
24	wsmtaber	285.2870	54.7590	0.3600
25	wsmtaber	287.0600	52.9860	0.3600
	Totals:	**7,962.5195**	**538.6113**	**9.0200**

Table A1: Scenario 3 Results for Time Slice = 1

Process ID	Hostname	Working Time (in seconds)	Idle Time (in seconds)	Waiting Time (in seconds)
1	svAltiris	320.0620	0.0780	2.6250
2	svAltiris	319.4540	0.0460	3.2650
3	svAltiris	319.5160	-0.0010	3.2500
4	svAltiris	319.1410	0.1560	3.4680
5	Inspiron8100	320.1440	0.4650	2.1560
6	lpjwandke	318.3540	0.4430	3.9680
7	lpjwandke	319.0930	0.3290	3.3430
8	mrsrack01	318.7490	0.8760	3.1400
9	mrsrack01	319.4060	1.0000	2.3590
10	mrsweb01	318.8440	0.8430	3.0780
11	mrsweb01	319.0340	0.9190	2.8120
12	svgtserver	322.3950	0.3700	0.0000
13	svgtserver	322.0990	0.5730	0.0930
14	svsm5	319.9700	0.7020	2.0930
15	svsm5	320.0340	0.8880	1.8430
16	svsm6	319.9530	0.7660	2.0460
17	svsm6	319.8450	0.8270	2.0930
18	svsm7	317.9850	2.1090	2.6710
19	svsm7	317.7160	2.3310	2.7180
20	svsm8	319.8840	0.7410	2.1400
21	svsm8	320.0700	0.5860	2.1090
22	svSymantec	319.4010	0.9270	2.4370
23	svSymantec	319.4010	0.9430	2.4210
24	wsmtaber	317.8330	1.4320	3.5000
25	wsmtaber	317.2680	1.8410	3.6560
	Totals:	**7,985.6509**	**20.1904**	**63.2840**

Table A2: Scenario 3 Results for Time Slice = 5

The negative idle time for Process ID 3 can be attributed to a rounding error in the calculation of the idle time.

Process ID	Hostname	Working Time (in seconds)	Idle Time (in seconds)	Waiting Time (in seconds)
1	svAltiris	320.3130	-0.0010	3.8440
2	svAltiris	321.1570	0.1240	2.8750
3	svAltiris	321.4060	0.2030	2.5470
4	svAltiris	320.2650	0.0470	3.8440
5	Inspiron8100	320.1840	0.1120	3.8600
6	lpjwandke	318.8890	0.3920	4.8750
7	lpjwandke	319.8720	0.1740	4.1100
8	mrsrack01	319.8740	0.6570	3.6250
9	mrsrack01	321.0600	0.4080	2.6880
10	mrsweb01	319.5620	0.8130	3.7810
11	mrsweb01	319.5170	0.3110	4.3280
12	svgtserver	321.1920	0.9330	2.0310
13	svgtserver	320.7540	0.3080	3.0940
14	svsm5	318.7970	0.4210	4.9380
15	svsm5	318.6870	0.5780	4.8910
16	svsm6	318.6250	0.5620	4.9690
17	svsm6	318.6880	0.4830	4.9850
18	svsm7	321.3430	0.8910	1.9220
19	svsm7	321.1110	1.0290	2.0160
20	svsm8	318.4900	0.6810	4.9850
21	svsm8	318.8800	0.5880	4.6880
22	svSymantec	323.5890	0.2700	0.2970
23	svSymantec	323.8560	0.3000	0.0000
24	wsmtaber	318.7850	0.7150	4.6560
25	wsmtaber	319.5820	0.6990	3.8750
	Totals:	**8,004.4771**	**11.6982**	**87.7240**

Table A3: Scenario 3 Results for Time Slice = 10

The negative idle time for Process ID 1 can be attributed to a rounding error in the calculation of the idle time.

Process ID	Hostname	Working Time (in seconds)	Idle Time (in seconds)	Waiting Time (in seconds)
1	svAltiris	321.3440	0.0000	8.8750
2	svAltiris	319.1410	0.0310	11.0470
3	svAltiris	318.1240	0.0320	12.0630
4	svAltiris	317.8910	-0.0010	12.3290
5	Inspiron8100	330.2080	0.0110	0.0000
6	lpjwandke	316.6700	1.3920	12.1570
7	lpjwandke	319.9350	0.1590	10.1250
8	mrsrack01	317.2480	0.3460	12.6250
9	mrsrack01	319.3260	0.5330	10.3600
10	mrsweb01	320.9240	0.2790	9.0160
11	mrsweb01	320.5970	0.3720	9.2500
12	svgtserver	327.2860	0.1980	2.7350
13	svgtserver	326.7870	0.1820	3.2500
14	svsm5	319.8270	0.2040	10.1880
15	svsm5	320.1090	0.1880	9.9220
16	svsm6	319.7650	0.3910	10.0630
17	svsm6	319.8120	0.4070	10.0000
18	svsm7	319.8900	0.5940	9.7350
19	svsm7	320.2490	0.5630	9.4070
20	svsm8	319.7860	0.2140	10.2190
21	svsm8	319.9260	0.1990	10.0940
22	svSymantec	322.6520	0.3630	7.2040
23	svSymantec	322.6680	0.3790	7.1720
24	wsmtaber	316.6440	0.5280	13.0470
25	wsmtaber	316.7060	0.4340	13.0790
	Totals:	**8,013.5151**	**7.9978**	**233.9620**

Table A4: Scenario 3 Results for Time Slice = 15

The negative idle time for Process ID 4 can be attributed to a rounding error in the calculation of the idle time.

Process ID	Hostname	Working Time (in seconds)	Idle Time (in seconds)	Waiting Time (in seconds)
1	svAltiris	323.4070	-0.0010	21.7030
2	svAltiris	320.3280	0.0150	24.7660
3	svAltiris	321.9840	0.0160	23.1090
4	svAltiris	325.5790	0.1080	19.4220
5	Inspiron8100	345.0000	0.1090	0.0000
6	lpjwandke	320.2790	0.3770	24.4530
7	lpjwandke	327.7650	0.6720	16.6720
8	mrsrack01	331.8420	0.2830	12.9840
9	mrsrack01	323.4510	0.2670	21.3910
10	mrsweb01	320.1580	0.4350	24.5160
11	mrsweb01	319.8460	0.3410	24.9220
12	svgtserver	340.3490	0.2910	4.4690
13	svgtserver	340.1460	0.2910	4.6720
14	svsm5	333.4210	0.3130	11.3750
15	svsm5	333.8120	0.2810	11.0160
16	svsm6	333.5780	0.4530	11.0780
17	svsm6	333.5940	0.4530	11.0620
18	svsm7	326.4380	0.4840	18.1870
19	svsm7	326.2350	0.6080	18.2660
20	svsm8	333.4420	0.3230	11.3440
21	svsm8	333.5660	0.3090	11.2340
22	svSymantec	325.5440	0.6120	18.9530
23	svSymantec	325.1530	0.4870	19.4690
24	wsmtaber	324.2840	0.1380	20.6870
25	wsmtaber	327.0190	0.1210	17.9690
	Totals:	**8,216.2197**	**7.7862**	**403.7190**

Table A5: Scenario 3 Results for Time Slice = 30

The negative idle time for Process ID 1 can be attributed to a rounding error in the calculation of the idle time.

Process ID	Hostname	Working Time (in seconds)	Idle Time (in seconds)	Waiting Time (in seconds)
1	svAltiris	309.9380	0.0160	26.8590
2	svAltiris	319.0150	0.0170	17.7810
3	svAltiris	309.8130	0.0160	26.9840
4	svAltiris	310.1560	0.0160	26.6410
5	Inspiron8100	336.7970	0.0160	0.0000
6	lpjwandke	309.3110	0.0020	27.5000
7	lpjwandke	312.9200	0.0180	23.8750
8	mrsrack01	313.3250	0.2070	23.2810
9	mrsrack01	322.0450	0.2050	14.5630
10	mrsweb01	315.0790	0.2030	21.5310
11	mrsweb01	314.6270	0.2020	21.9840
12	svgtserver	323.4120	0.2290	13.1720
13	svgtserver	323.3960	0.1980	13.2190
14	svsm5	331.1250	0.2040	5.4840
15	svsm5	330.6560	0.2040	5.9530
16	svsm6	330.7820	0.3750	5.6560
17	svsm6	330.8760	0.3740	5.5630
18	svsm7	323.7650	0.3760	12.6720
19	svsm7	323.5000	0.3750	12.9380
20	svsm8	330.7230	0.1990	5.8910
21	svsm8	330.7860	0.1990	5.8280
22	svSymantec	312.3710	0.3790	24.0630
23	svSymantec	333.6670	0.3800	2.7660
24	wsmtaber	315.5670	0.1680	21.0780
25	wsmtaber	310.6920	0.1830	25.9380
	Totals:	**8,024.3447**	**4.7607**	**391.2200**

Table A6: Scenario 3 Results for Time Slice = 60

Appendix B

Process ID	Hostname	Working Time (in seconds)	Idle Time (in seconds)	Waiting Time (in seconds)
1	svAltiris	1,353.2419	15.8361	0.6090
2	svAltiris	1,360.1560	8.7030	0.8280
3	svAltiris	1,360.2679	8.6541	0.7650
4	svAltiris	1,355.0909	13.8771	0.7190
5	Inspiron8100	1,325.6130	43.5120	0.5620
6	lpjwandke	1,291.1121	77.9500	0.6250
7	lpjwandke	1,314.6300	54.4170	0.6400
8	mrsrack01	1,259.0560	110.1000	0.5310
9	mrsrack01	1,258.0090	110.8030	0.8750
10	mrsweb01	1,252.1750	116.7930	0.7190
11	mrsweb01	1,253.3660	115.6020	0.7190
12	svgtserver	1,343.5640	26.1230	0.0000
13	svgtserver	1,342.9570	26.7300	0.0000
14	svsm5	1,254.1350	114.7710	0.7810
15	svsm5	1,257.0050	111.9010	0.7810
16	svsm6	1,253.8260	115.0800	0.7810
17	svsm6	1,255.8680	113.0380	0.7810
18	svsm7	1,292.8650	75.9630	0.8590
19	svsm7	1,294.1720	75.0930	0.4220
20	svsm8	1,256.1870	112.7190	0.7810
21	svsm8	1,256.0400	112.8660	0.7810
22	svSymantec	1,299.2800	69.9070	0.5000
23	svSymantec	1,297.8220	71.3650	0.5000
24	wsmtaber	1,162.2321	206.7520	0.7030
25	wsmtaber	1,162.6801	206.1790	0.8280
	Totals:	**32,111.3496**	**2,114.7339**	**16.0900**

Table B1: Scenario 4 Results for Time Slice = 1

Process ID	Hostname	Working Time (in seconds)	Idle Time (in seconds)	Waiting Time (in seconds)
1	svAltiris	1,294.2371	0.2789	3.9370
2	svAltiris	1,293.7050	0.4360	4.3120
3	svAltiris	1,294.2980	0.8740	3.2810
4	svAltiris	1,294.4530	0.2970	3.7030
5	Inspiron8100	1,294.1610	3.4950	0.7970
6	lpjwandke	1,290.4969	4.1441	3.8120
7	lpjwandke	1,290.3311	4.0599	4.0620
8	mrsrack01	1,282.4871	11.3259	4.6400
9	mrsrack01	1,286.3669	8.2271	3.8590
10	mrsweb01	1,282.1920	12.0890	4.1720
11	mrsweb01	1,283.4580	10.7920	4.2030
12	svgtserver	1,286.3190	12.1340	0.0000
13	svgtserver	1,286.2520	12.2010	0.0000
14	svsm5	1,289.3320	6.1840	2.9370
15	svsm5	1,289.8610	5.9670	2.6250
16	svsm6	1,289.2980	6.1080	3.0470
17	svsm6	1,289.4120	5.3850	3.6560
18	svsm7	1,282.4240	12.8731	3.1560
19	svsm7	1,284.9139	11.1961	2.3430
20	svsm8	1,289.5470	5.5630	3.3430
21	svsm8	1,289.6530	6.0350	2.7650
22	svSymantec	1,287.3149	7.5601	3.5780
23	svSymantec	1,287.4550	7.7330	3.2650
24	wsmtaber	1,277.4880	16.9810	3.9840
25	wsmtaber	1,278.5930	15.9700	3.8900
	Totals:	**32,194.0508**	**187.9094**	**79.3670**

Table B2: Scenario 4 Results for Time Slice = 5

Process ID	Hostname	Working Time (in seconds)	Idle Time (in seconds)	Waiting Time (in seconds)
1	svAltiris	1,278.0470	0.3120	6.6100
2	svAltiris	1,279.0000	0.1400	5.8290
3	svAltiris	1,276.4840	0.1410	8.3440
4	svAltiris	1,277.9360	0.2510	6.7820
5	Inspiron8100	1,280.2620	0.6910	4.0160
6	lpjwandke	1,276.8590	1.4220	6.6880
7	lpjwandke	1,276.5320	0.7960	7.6410
8	mrsrack01	1,278.0229	1.4460	5.5000
9	mrsrack01	1,275.9340	1.8780	7.1570
10	mrsweb01	1,274.4860	2.5920	7.8910
11	mrsweb01	1,275.6270	2.5920	6.7500
12	svgtserver	1,281.3051	2.0849	1.5790
13	svgtserver	1,282.9580	2.0110	0.0000
14	svsm5	1,275.4060	1.2190	8.3440
15	svsm5	1,275.6740	1.2480	8.0470
16	svsm6	1,275.1379	1.3931	8.4380
17	svsm6	1,277.8770	1.4040	5.6880
18	svsm7	1,274.9180	4.6910	5.3600
19	svsm7	1,274.8130	3.8590	6.2970
20	svsm8	1,275.7531	1.2309	7.9850
21	svsm8	1,279.6560	1.4690	3.8440
22	svSymantec	1,275.4530	1.3910	8.1250
23	svSymantec	1,275.3770	1.4820	8.1100
24	wsmtaber	1,274.4420	3.0890	7.4380
25	wsmtaber	1,273.8790	2.9800	8.1100
	Totals:	**31,921.8398**	**41.8131**	**160.5730**

Table B3: Scenario 4 Results for Time Slice = 10

Process ID	Hostname	Working Time (in seconds)	Idle Time (in seconds)	Waiting Time (in seconds)
1	svAltiris	1,290.2020	0.0480	3.9060
2	svAltiris	1,288.2321	0.2840	5.6400
3	svAltiris	1,291.0780	0.0320	3.0460
4	svAltiris	1,289.1870	0.0790	4.8900
5	Inspiron8100	1,293.8929	0.2631	0.0000
6	lpjwandke	1,286.5010	0.7340	6.9210
7	lpjwandke	1,288.0450	0.7050	5.4060
8	mrsrack01	1,286.1949	1.0551	6.9060
9	mrsrack01	1,289.0710	0.5230	4.5620
10	mrsweb01	1,289.5551	0.9760	3.6250
11	mrsweb01	1,285.6920	1.1680	7.2960
12	svgtserver	1,288.2560	0.5880	5.3120
13	svgtserver	1,288.1930	0.4480	5.5150
14	svsm5	1,287.5020	0.6701	5.9840
15	svsm5	1,287.0300	0.6730	6.4530
16	svsm6	1,286.3010	0.8240	7.0310
17	svsm6	1,287.2350	0.8120	6.1090
18	svsm7	1,289.9830	1.1420	3.0310
19	svsm7	1,290.6899	1.1701	2.2960
20	svsm8	1,286.3910	0.6720	7.0930
21	svsm8	1,287.0170	1.3740	5.7650
22	svSymantec	1,291.9540	1.1400	1.0620
23	svSymantec	1,291.8450	1.2490	1.0620
24	wsmtaber	1,287.8311	2.2940	4.0310
25	wsmtaber	1,286.7830	2.1080	5.2650
	Totals:	**32,214.6621**	**21.0312**	**118.2070**

Table B4: Scenario 4 Results for Time Slice = 15

Process ID	Hostname	Working Time (in seconds)	Idle Time (in seconds)	Waiting Time (in seconds)
1	svAltiris	1,281.2960	0.0170	20.7650
2	svAltiris	1,281.6090	0.0320	20.4370
3	svAltiris	1,282.0310	0.0160	20.0310
4	svAltiris	1,288.9840	0.0160	13.0780
5	Inspiron8100	1,292.3910	0.2180	9.4690
6	lpjwandke	1,285.0000	0.2810	16.7970
7	lpjwandke	1,288.3900	0.1410	13.5470
8	mrsrack01	1,282.5630	0.5150	19.0000
9	mrsrack01	1,281.6090	0.2350	20.2340
10	mrsweb01	1,280.3979	0.2581	21.4220
11	mrsweb01	1,280.1490	0.3980	21.5310
12	svgtserver	1,300.9740	0.1980	0.9060
13	svgtserver	1,301.8800	0.1980	0.0000
14	svsm5	1,287.4850	0.3900	14.2030
15	svsm5	1,286.4850	0.2180	15.3750
16	svsm6	1,286.8140	0.5300	14.7340
17	svsm6	1,286.5630	0.4210	15.0940
18	svsm7	1,288.5780	0.4690	13.0310
19	svsm7	1,287.2040	0.3590	14.5150
20	svsm8	1,286.4690	0.2500	15.3590
21	svsm8	1,287.2830	0.2480	14.5470
22	svSymantec	1,283.4980	0.3930	18.1870
23	svSymantec	1,283.4210	0.3760	18.2810
24	wsmtaber	1,281.9280	0.1810	19.9690
25	wsmtaber	1,287.2410	0.3680	14.4690
	Totals:	**32,160.2422**	**6.7262**	**384.9810**

Table B5: Scenario 4 Results for Time Slice = 30

Process ID	Hostname	Working Time (in seconds)	Idle Time (in seconds)	Waiting Time (in seconds)
1	svAltiris	1,271.7340	0.0010	47.2810
2	svAltiris	1,270.3120	0.0010	48.7030
3	svAltiris	1,283.4530	0.0000	35.5630
4	svAltiris	1,278.3900	0.0170	40.6090
5	Inspiron8100	1,319.0780	-0.0620	0.0000
6	lpjwandke	1,275.7810	0.2040	43.0310
7	lpjwandke	1,271.6410	0.1720	47.2030
8	mrsrack01	1,274.0970	0.1689	44.7500
9	mrsrack01	1,273.9091	0.1689	44.9380
10	mrsweb01	1,289.4680	-0.0300	29.5780
11	mrsweb01	1,276.7960	0.1570	42.0630
12	svgtserver	1,299.1160	0.0560	19.8440
13	svgtserver	1,297.1770	0.0260	21.8130
14	svsm5	1,269.0620	0.0630	49.8910
15	svsm5	1,298.4060	0.0630	20.5470
16	svsm6	1,268.3280	0.2040	50.4840
17	svsm6	1,268.5630	0.2190	50.2340
18	svsm7	1,269.7660	0.0470	49.2030
19	svsm7	1,270.6720	0.0470	48.2970
20	svsm8	1,283.9070	0.0460	35.0630
21	svsm8	1,268.4850	0.0470	50.4840
22	svSymantec	1,283.4060	0.1720	35.4380
23	svSymantec	1,283.4670	0.1430	35.4060
24	wsmtaber	1,272.2460	0.4890	46.2810
25	wsmtaber	1,273.2321	0.3309	45.4530
	Totals:	**31,990.4902**	**2.7506**	**982.1570**

Table B6: Scenario 4 Results for Time Slice = 60

The negative idle time for Process ID's 5 and 10 can be attributed to rounding errors in the calculation of the idle time.

Appendix C: PRTGen Source Code

Benchmark.h file contents

```
/*
   PRTGen - A Parallel implementation of RainbowCrack using MPI.
   Copyright (C) 2008 Mike Taber <mstaber@gmail.com>
*/
#pragma once

#include <iostream>
#include <list>
#include <time.h>

using namespace std;

class Benchmark
{
      friend ostream &operator<<(ostream &, const Benchmark &);

public:
      Benchmark(void);
      ~Benchmark(void);

      Benchmark &operator=(const Benchmark &rhs);
      int operator==(const Benchmark &rhs) const;
      int operator<(const Benchmark &rhs) const;

public:
      int processID;
      string hostname;
      long speed;
    clock_t waitingTimeStart; // used to keep track of when this
process first started waiting for other processes to complete

      double dWorkingTime;          // time spent doing "real work"
      double dWaitingTime;          // time spent not doing "real work"
      double dIdleTime;           // time the process spent waiting for
other nodes to finish their work
                                          // dWorkingTime +
dWaitingTime + dIdleTime = dTotalTime
};
```

Benchmark.cpp file contents

```
/*
   PRTGen - A Parallel implementation of RainbowCrack using MPI.
   Copyright (C) 2008 Mike Taber <mstaber@gmail.com>
*/
#include "Benchmark.h"

Benchmark::Benchmark(void)
{
      processID=0;
      hostname="";
      speed=0;
}

Benchmark::~Benchmark(void)
{
}

Benchmark& Benchmark::operator=(const Benchmark &rhs)
{
   this->processID = rhs.processID;
   this->hostname = rhs.hostname;
   this->speed = rhs.speed;
   this->waitingTimeStart = rhs.waitingTimeStart;
   this->dWorkingTime = rhs.dWorkingTime;
   this->dWaitingTime = rhs.dWaitingTime;
   this->dIdleTime = rhs.dIdleTime;
   return *this;
}

// equality is based on a process ID and a hostname. That's it.
int Benchmark::operator==(const Benchmark &rhs) const
{
   if( this->processID != rhs.processID) return 0;
   if( this->hostname.compare(rhs.hostname) != 0 ) return 0;
   return 1;
}

// This function is required for built-in STL list functions like sort
int Benchmark::operator<(const Benchmark &rhs) const
{
      // sort by speed
   if( this->speed < rhs.speed ) return 1;
   return 0;
}

ostream &operator<<(ostream &output, const Benchmark &b)
{
      output << b.processID << ' ' << b.hostname.c_str() << ' ' <<
b.speed << b.dWorkingTime << b.dWaitingTime << b.dIdleTime << endl;
   return output;
}
```

ChainWalkContext.h file contents

```
/*
   PRTGen - A Parallel implementation of RainbowCrack using MPI.
   Copyright (C) 2008 Mike Taber <mstaber@gmail.com>

   This source code is based on:
   RainbowCrack - a general propose implementation of Philippe
Oechslin's faster time-memory trade-off technique.
   Copyright (C) Zhu Shuanglei <shuanglei@hotmail.com>
*/
#ifndef _CHAINWALKCONTEXT_H
#define _CHAINWALKCONTEXT_H

#include "HashRoutine.h"
#include "Public.h"

class CChainWalkContext
{
public:
      CChainWalkContext();
      virtual ~CChainWalkContext();

private:
      static string m_sHashRoutineName;
      static HASHROUTINE m_pHashRoutine;
      // Configuration
      static int m_nHashLen;
            // Configuration

      static unsigned char m_PlainCharset[256];
      // Configuration: stores the charset being used as a character
array
      static int m_nPlainCharsetLen;
            // Configuration
      static int m_nPlainLenMin;
            // Configuration
      static int m_nPlainLenMax;
            // Configuration
      static string m_sPlainCharsetName;
      static string m_sPlainCharsetContent;
      // stores the charset being used as a string
      static uint64 m_nPlainSpaceUpToX[MAX_PLAIN_LEN + 1];        //
Performance consideration
      static uint64 m_nPlainSpaceTotal;
      // Performance consideration

      static int m_nRainbowTableIndex;
      // Configuration
      static uint64 m_nReduceOffset;
            // Performance consideration

      // Context
      uint64 m_nIndex;
      unsigned char m_Plain[MAX_PLAIN_LEN];
```

```
	int m_nPlainLen;
	unsigned char m_Hash[MAX_HASH_LEN];

private:
//	static bool LoadCharset(string sName);
	static bool LoadCharset(string exePath, string sName);

public:
	static bool SetHashRoutine(string sHashRoutineName);
                                                                //
Configuration
//	static bool SetPlainCharset(string sCharsetName, int
nPlainLenMin, int nPlainLenMax);                                //
Configuration
	static bool SetPlainCharset(string exePath, string sCharsetName,
int nPlainLenMin, int nPlainLenMax);	// Configuration
	static bool SetRainbowTableIndex(int nRainbowTableIndex);
                                                                //
Configuration
//	static bool SetupWithPathName(string sPathName, int&
nRainbowChainLen, int& nRainbowChainCount);		// Wrapper
	static bool SetupWithPathName(string exePath, string sPathName,
int& nRainbowChainLen, int& nRainbowChainCount);		// Wrapper
	static string GetHashRoutineName();
	static int GetHashLen();
	static string GetPlainCharsetName();
	static string GetPlainCharsetContent();
	static int GetPlainLenMin();
	static int GetPlainLenMax();
	static uint64 GetPlainSpaceTotal();
	static int GetRainbowTableIndex();
	static void Dump();

	void GenerateRandomIndex();
	void SetIndex(uint64 nIndex);
	void SetHash(unsigned char* pHash);	// The length should be
m_nHashLen

	void IndexToPlain();
	void PlainToHash();
	void HashToIndex(int nPos);

	uint64 GetIndex();
	string GetPlain();
	string GetBinary();
	string GetPlainBinary();
	string GetHash();
	bool CheckHash(unsigned char* pHash);	// The length should be
m_nHashLen
};

#endif
```

ChainWalkContext.cpp file contents

```
/*
   PRTGen - A Parallel implementation of RainbowCrack using MPI.
   Copyright (C) 2008 Mike Taber <mstaber@gmail.com>

   This source code is based on:
   RainbowCrack - a general propose implementation of Philippe
Oechslin's faster time-memory trade-off technique.
   Copyright (C) Zhu Shuanglei <shuanglei@hotmail.com>
*/
#ifdef _WIN32
      #pragma warning(disable : 4786)
#endif

#include "ChainWalkContext.h"

#include <ctype.h>
#include <openssl/rand.h>
#ifdef _WIN32
      #pragma comment(lib, "libeay32.lib")
#endif

////////////////////////////////////////////////////////////////////////

string CChainWalkContext::m_sHashRoutineName;
HASHROUTINE CChainWalkContext::m_pHashRoutine;
int CChainWalkContext::m_nHashLen;

unsigned char CChainWalkContext::m_PlainCharset[256];
int CChainWalkContext::m_nPlainCharsetLen;
int CChainWalkContext::m_nPlainLenMin;
int CChainWalkContext::m_nPlainLenMax;
string CChainWalkContext::m_sPlainCharsetName;
string CChainWalkContext::m_sPlainCharsetContent;
uint64 CChainWalkContext::m_nPlainSpaceUpToX[MAX_PLAIN_LEN + 1];
uint64 CChainWalkContext::m_nPlainSpaceTotal;

int CChainWalkContext::m_nRainbowTableIndex;
uint64 CChainWalkContext::m_nReduceOffset;

////////////////////////////////////////////////////////////////////////

CChainWalkContext::CChainWalkContext()
{
}

CChainWalkContext::~CChainWalkContext()
{
}

bool CChainWalkContext::LoadCharset(string exePath, string sName)
{
      if (sName == "byte")
      {
```

```
            int i;
            for (i = 0x00; i <= 0xff; i++)
                  m_PlainCharset[i] = i;
            m_nPlainCharsetLen = 256;
            m_sPlainCharsetName = sName;
            m_sPlainCharsetContent = "0x00, 0x01, ... 0xff";
            return true;
      }

      vector<string> vLine;
      string filePath = exePath + "\\charset.txt";
      if (ReadLinesFromFile(filePath.c_str(), vLine))
      {
            int i;
            for (i = 0; i < (int)vLine.size(); i++)
            {
                  // Filter comment lines
                  if (vLine[i][0] == '#')
                        continue;

                  vector<string> vPart;
                  if (SeperateString(vLine[i], "=", vPart))
                  {
                        // sCharsetName
                        string sCharsetName = TrimString(vPart[0]);
                        if (sCharsetName == "")
                              continue;

                        // sCharsetName charset check
                        // Valid characters in the sCharsetName are
alpha-numeric, and dashes ('-').
                        // Anything else that appears generates an
error
                        bool fCharsetNameCheckPass = true;
                        int j;
                        for (j = 0; j < (int)sCharsetName.size(); j++)
                        {
                              if (   !isalpha(sCharsetName[j])
                                    && !isdigit(sCharsetName[j])
                                    && (sCharsetName[j] != '-'))
                              {
                                    fCharsetNameCheckPass = false;
                                    break;
                              }
                        }
                        if (!fCharsetNameCheckPass)
                        {
                              printf("invalid charset name %s in
charset configuration file\n", sCharsetName.c_str());
                              continue;
                        }

                        // sCharsetContent
                        string sCharsetContent = TrimString(vPart[1]);
                        if (sCharsetContent == "" || sCharsetContent ==
"[]")
                        {
```

```
                              // skip empty character sets
                              continue;
                        }
                        if (sCharsetContent[0] != '[' ||
sCharsetContent[sCharsetContent.size() - 1] != ']')
                        {
                              // skip character sets that don't start
and end with square brackets
                              printf("invalid charset content %s in
charset configuration file\n", sCharsetContent.c_str());
                              continue;
                        }

                        // set the contents of the characterset
                        sCharsetContent = sCharsetContent.substr(1,
sCharsetContent.size() - 2);
                        if (sCharsetContent.size() > 256)
                        {
                              // charactersets are not allowed to be
more than 256 bytes long.
                              // skip these if they appear as well
                              printf("charset content %s too long\n",
sCharsetContent.c_str());
                              continue;
                        }

                        // Is it the wanted charset?
                        if (sCharsetName == sName)
                        {
                              m_nPlainCharsetLen =
(int)sCharsetContent.size();
                              memcpy(m_PlainCharset,
sCharsetContent.c_str(), m_nPlainCharsetLen);
                              m_sPlainCharsetName = sCharsetName;
                              m_sPlainCharsetContent = sCharsetContent;
                              return true;
                        }
                  }
            }
            printf("charset %s not found in charset.txt\n",
sName.c_str());
      }
      else
    {
            printf("can't open charset configuration
file\n\t%s\n",filePath.c_str());
    }

      return false;
}

////////////////////////////////////////////////////////////////////

bool CChainWalkContext::SetHashRoutine(string sHashRoutineName)
{
      CHashRoutine hr;
      hr.GetHashRoutine(sHashRoutineName, m_pHashRoutine, m_nHashLen);
```

```
	if (m_pHashRoutine != NULL)
	{
		m_sHashRoutineName = sHashRoutineName;
		return true;
	}
	else
		return false;
}

bool CChainWalkContext::SetPlainCharset(string exePath, string
sCharsetName, int nPlainLenMin, int nPlainLenMax)
{
	// m_PlainCharset, m_nPlainCharsetLen, m_sPlainCharsetName,
m_sPlainCharsetContent
	if (!LoadCharset(exePath, sCharsetName))
		return false;

	// m_nPlainLenMin, m_nPlainLenMax
	// perform error checking on the minimum and maximum plaintext
length
	if (nPlainLenMin < 1 || nPlainLenMax > MAX_PLAIN_LEN ||
nPlainLenMin > nPlainLenMax)
	{
		printf("invalid plaintext length range: %d - %d\n",
nPlainLenMin, nPlainLenMax);
		return false;
	}

	// set the class static variables
	m_nPlainLenMin = nPlainLenMin;
	m_nPlainLenMax = nPlainLenMax;

	// calculate the key space for this run, based on the size of the
character set, and the plaintext range.
	// each entry in the array "m_nPlainSpaceUpToX" stores the key
space calculated so far for each plaintext length,
	// and all previous plaintext lengths.
	// The purpose of this is to be able to help map an index number
to a specific plaintext value
	// the first element in m_nPlainSpaceUpToX is always ZERO
	m_nPlainSpaceUpToX[0] = 0;
	uint64 nTemp = 1;
	int i;
	for (i = 1; i <= m_nPlainLenMax; i++)
	{
		nTemp *= m_nPlainCharsetLen;
		if (i < m_nPlainLenMin)
			m_nPlainSpaceUpToX[i] = 0;
		else
			m_nPlainSpaceUpToX[i] = m_nPlainSpaceUpToX[i - 1] +
nTemp;
	}

	// m_nPlainSpaceTotal
	m_nPlainSpaceTotal = m_nPlainSpaceUpToX[m_nPlainLenMax];

	return true;
```

```
}

bool CChainWalkContext::SetRainbowTableIndex(int nRainbowTableIndex)
{
	// the RainbowTableIndex must be a number greater than or equal
to zero
	if (nRainbowTableIndex < 0)
		return false;
	m_nRainbowTableIndex = nRainbowTableIndex;
	// m_nReduceOffset is a mechanism for helping to ensure that the
reduction function used in each rainbow table file is unique,
	// even for the same position, or at least it is extremely likely
to be unique
	m_nReduceOffset = 65536 * nRainbowTableIndex;

	return true;
}

bool CChainWalkContext::SetupWithPathName(string exePath, string
sPathName, int& nRainbowChainLen, int& nRainbowChainCount)
{
	// something like lm_alpha#1-7_0_100x16_test.rt

#ifdef _WIN32
	int nIndex = (int)sPathName.find_last_of('\\');
#else
	int nIndex = sPathName.find_last_of('/');
#endif
	if (nIndex != -1)
		sPathName = sPathName.substr(nIndex + 1);

	if (sPathName.size() < 3)
	{
		printf("%s is not a rainbow table\n", sPathName.c_str());
		return false;
	}
	if (sPathName.substr(sPathName.size() - 3) != ".rt")
	{
		printf("%s is not a rainbow table\n", sPathName.c_str());
		return false;
	}

	// Parse
	vector<string> vPart;
	if (!SeperateString(sPathName, "___x_", vPart))
	{
		printf("filename %s not identified\n", sPathName.c_str());
		return false;
	}

	string sHashRoutineName   = vPart[0];
	int nRainbowTableIndex    = atoi(vPart[2].c_str());

	nRainbowChainLen          = atoi(vPart[3].c_str());
	nRainbowChainCount        = atoi(vPart[4].c_str());

	// Parse charset definition
```

```
	string sCharsetDefinition = vPart[1];
	string sCharsetName;
	int nPlainLenMin, nPlainLenMax;
	if (sCharsetDefinition.find('#') == -1)		// For backward
compatibility, "#1-7" is implied
	{
		sCharsetName = sCharsetDefinition;
		nPlainLenMin = 1;
		nPlainLenMax = 7;
	}
	else
	{
		vector<string> vCharsetDefinitionPart;
		if (!SeperateString(sCharsetDefinition, "#-",
vCharsetDefinitionPart))
		{
			printf("filename %s not identified\n",
sPathName.c_str());
			return false;
		}
		else
		{
			sCharsetName = vCharsetDefinitionPart[0];
			nPlainLenMin =
atoi(vCharsetDefinitionPart[1].c_str());
			nPlainLenMax =
atoi(vCharsetDefinitionPart[2].c_str());
		}
	}

	// Setup
	if (!SetHashRoutine(sHashRoutineName))
	{
		printf("hash routine %s not supported\n",
sHashRoutineName.c_str());
		return false;
	}
	if (!SetPlainCharset(exePath, sCharsetName, nPlainLenMin,
nPlainLenMax))
		return false;
	if (!SetRainbowTableIndex(nRainbowTableIndex))
	{
		printf("invalid rainbow table index %d\n",
nRainbowTableIndex);
		return false;
	}

	return true;
}

string CChainWalkContext::GetHashRoutineName()
{
	return m_sHashRoutineName;
}

int CChainWalkContext::GetHashLen()
{
```

```
	return m_nHashLen;
}

string CChainWalkContext::GetPlainCharsetName()
{
	return m_sPlainCharsetName;
}

string CChainWalkContext::GetPlainCharsetContent()
{
	return m_sPlainCharsetContent;
}

int CChainWalkContext::GetPlainLenMin()
{
	return m_nPlainLenMin;
}

int CChainWalkContext::GetPlainLenMax()
{
	return m_nPlainLenMax;
}

uint64 CChainWalkContext::GetPlainSpaceTotal()
{
	return m_nPlainSpaceTotal;
}

int CChainWalkContext::GetRainbowTableIndex()
{
	return m_nRainbowTableIndex;
}

void CChainWalkContext::Dump()
{
	printf("hash routine: %s\n", m_sHashRoutineName.c_str());
	printf("hash length: %d\n", m_nHashLen);

	printf("plain charset: ");
	int i;
	for (i = 0; i < m_nPlainCharsetLen; i++)
	{
		if (isprint(m_PlainCharset[i]))
			printf("%c", m_PlainCharset[i]);
		else
			printf("?");
	}
	printf("\n");

	printf("plain charset in hex: ");
	for (i = 0; i < m_nPlainCharsetLen; i++)
		printf("%02x ", m_PlainCharset[i]);
	printf("\n");

	printf("plain length range: %d - %d\n", m_nPlainLenMin,
m_nPlainLenMax);
	printf("plain charset name: %s\n", m_sPlainCharsetName.c_str());
```

```
	//printf("plain charset content: %s\n",
m_sPlainCharsetContent.c_str());
	//for (i = 0; i <= m_nPlainLenMax; i++)
	//    printf("plain space up to %d: %s\n", i,
uint64tostr(m_nPlainSpaceUpToX[i]).c_str());
	printf("plain space total: %s\n",
uint64tostr(m_nPlainSpaceTotal).c_str());

	printf("rainbow table index: %d\n", m_nRainbowTableIndex);
	printf("reduce offset: %s\n",
uint64tostr(m_nReduceOffset).c_str());
	printf("\n");
}

void CChainWalkContext::GenerateRandomIndex()
{
	// create 8 random bytes, which is then used as a 64 bit random
number
	RAND_bytes((unsigned char*)&m_nIndex, 8);
	// use this random number modulo the size of the total key space
to get an index that is guaranteed
	// to be less than the size of the m_nPlainSpaceTotal.
	// NOTE: On the x86 architecture, this is stored on disk in
little-endian format (little end first), so the byte
	// streams will look something like this: BF9A09BC 00000000,
should the m_nPlainSpaceTotal be 0x00000000FFFFFFFF
	// and the integer would be: 3154746047. Windows calculator would
display this as: BC09 9ABF
	m_nIndex = m_nIndex % m_nPlainSpaceTotal;
}

void CChainWalkContext::SetIndex(uint64 nIndex)
{
	m_nIndex = nIndex;
}

void CChainWalkContext::SetHash(unsigned char* pHash)
{
	memcpy(m_Hash, pHash, m_nHashLen);
}

// convert the index into a character string. Lots of math is done here
to map the numeric index into a simple array of characters.
// this is essentially using a different base counting system
void CChainWalkContext::IndexToPlain()
{
	int i;
	// Find the length of the plaintext that this index points to and
store it in m_nPlainLen.
	// It is entirely possible that it could be anything between the
min/max lengths specified as the plain length range
	for (i = m_nPlainLenMax - 1; i >= m_nPlainLenMin - 1; i--)
	{
		if (m_nIndex >= m_nPlainSpaceUpToX[i])
		{
			m_nPlainLen = i + 1;
			break;
```

```
            }
      }


      uint64 nIndexOfX = m_nIndex - m_nPlainSpaceUpToX[m_nPlainLen -
1];

      /*
      // Slow version
      for (i = m_nPlainLen - 1; i >= 0; i--)
      {
            m_Plain[i] = m_PlainCharset[nIndexOfX %
m_nPlainCharsetLen];
            nIndexOfX /= m_nPlainCharsetLen;
      }
      */

      // Fast version
      for (i = m_nPlainLen - 1; i >= 0; i--)
      {
#ifdef _WIN32
            // break to the 32 bit version of this code if nIndexOfX is
a 32 bit unsigned integer
            if (nIndexOfX < 0x100000000I64)
                  break;
#else
            if (nIndexOfX < 0x100000000llu)
                  break;
#endif
            // create a plaintext character string one character at a
time, based on the index and starting with the
            // least significant character.
            m_Plain[i] = m_PlainCharset[nIndexOfX %
m_nPlainCharsetLen];
            nIndexOfX /= m_nPlainCharsetLen;
      }

      // this is the 32 bit version of the above code. It uses assembly
to be much faster
      unsigned int nIndexOfX32 = (unsigned int)nIndexOfX;
      for (; i >= 0; i--)
      {
            //m_Plain[i] = m_PlainCharset[nIndexOfX32 %
m_nPlainCharsetLen];
            //nIndexOfX32 /= m_nPlainCharsetLen;

            unsigned int nPlainCharsetLen = m_nPlainCharsetLen;
            unsigned int nTemp;
#ifdef _WIN32
            __asm
            {
                  mov eax, nIndexOfX32
                  xor edx, edx
                  div nPlainCharsetLen
                  mov nIndexOfX32, eax
                  mov nTemp, edx
            }
```

```
#else
            __asm__ __volatile__ (  "mov %2, %%eax;"
                                                "xor %%edx, %%edx;"
                                                "divl %3;"
                                                "mov %%eax, %0;"
                                                "mov %%edx, %1;"
                                                : "=m"(nIndexOfX32),
"=m"(nTemp)
                                                : "m"(nIndexOfX32),
"m"(nPlainCharsetLen)
                                                : "%eax", "%edx"
                                            );
#endif
            m_Plain[i] = m_PlainCharset[nTemp];
      }
}

void CChainWalkContext::PlainToHash()
{
      m_pHashRoutine(m_Plain, m_nPlainLen, m_Hash);
}

// I think this is basically a reduction function. The m_nReduceOffset
will be unique for each
// numbered rainbow table. Even within the same rainbow table, the nPos
that is used will assist
// in ensuring that the index values are unique from hashes that
eventually map to the same value
// somewhere in the chain
void CChainWalkContext::HashToIndex(int nPos)
{
      // treat the hash as a pointer to a uint64. Take the value and
add m_nReduceOffset + nPos
      // then mod this by m_nPlainSpaceTotal to get the index
      // NOTE: This essentially just uses the first 64 bits of the hash
and introduces a reduction function
      // to help make sure that it is unique on this iteration.
      // NOTE: It appears that the use of the m_nReduceOffset is to
keep this reduction function
      // unique even between rainbow table files
      m_nIndex = (*(uint64*)m_Hash + m_nReduceOffset + nPos) %
m_nPlainSpaceTotal;
}

uint64 CChainWalkContext::GetIndex()
{
      return m_nIndex;
}

string CChainWalkContext::GetPlain()
{
      string sRet;
      int i;
      for (i = 0; i < m_nPlainLen; i++)
      {
            char c = m_Plain[i];
            if (c >= 32 && c <= 126)
```

```
                sRet += c;
            else
                sRet += '?';
        }

        return sRet;
}

string CChainWalkContext::GetBinary()
{
        return HexToStr(m_Plain, m_nPlainLen);
}

string CChainWalkContext::GetPlainBinary()
{
        string sRet;
        sRet += GetPlain();
        int i;
        for (i = 0; i < m_nPlainLenMax - m_nPlainLen; i++)
            sRet += ' ';

        sRet += "|";

        sRet += GetBinary();
        for (i = 0; i < m_nPlainLenMax - m_nPlainLen; i++)
            sRet += "  ";

        return sRet;
}

string CChainWalkContext::GetHash()
{
        return HexToStr(m_Hash, m_nHashLen);
}

bool CChainWalkContext::CheckHash(unsigned char* pHash)
{
        if (memcmp(m_Hash, pHash, m_nHashLen) == 0)
            return true;

        return false;
}
```

HashAlgorithm.h file contents

```
/*
   PRTGen - A Parallel implementation of RainbowCrack using MPI.
   Copyright (C) 2008 Mike Taber <mstaber@gmail.com>

   This source code is based on:
   RainbowCrack - a general propose implementation of Philippe
Oechslin's faster time-memory trade-off technique.
   Copyright (C) Zhu Shuanglei <shuanglei@hotmail.com>
*/

#ifndef _HASHALGORITHM_H
#define _HASHALGORITHM_H

void HashLM(unsigned char* pPlain, int nPlainLen, unsigned char*
pHash);
void HashMD5(unsigned char* pPlain, int nPlainLen, unsigned char*
pHash);
void HashSHA1(unsigned char* pPlain, int nPlainLen, unsigned char*
pHash);

#endif
```

HashAlgorithm.cpp file contents

```
/*
   PRTGen - A Parallel implementation of RainbowCrack using MPI.
   Copyright (C) 2008 Mike Taber <mstaber@gmail.com>

   This source code is based on:
   RainbowCrack - a general propose implementation of Philippe
Oechslin's faster time-memory trade-off technique.
   Copyright (C) Zhu Shuanglei <shuanglei@hotmail.com>
*/

#include "HashAlgorithm.h"

#include <openssl/des.h>
#include <openssl/md5.h>
#include <openssl/sha.h>
#ifdef _WIN32
      #pragma comment(lib, "libeay32.lib")
#endif

void setup_des_key(unsigned char key_56[], des_key_schedule &ks)
{
      des_cblock key;

      key[0] = key_56[0];
      key[1] = (key_56[0] << 7) | (key_56[1] >> 1);
      key[2] = (key_56[1] << 6) | (key_56[2] >> 2);
      key[3] = (key_56[2] << 5) | (key_56[3] >> 3);
      key[4] = (key_56[3] << 4) | (key_56[4] >> 4);
      key[5] = (key_56[4] << 3) | (key_56[5] >> 5);
      key[6] = (key_56[5] << 2) | (key_56[6] >> 6);
      key[7] = (key_56[6] << 1);

      //des_set_odd_parity(&key);
      des_set_key(&key, ks);
}

void HashLM(unsigned char* pPlain, int nPlainLen, unsigned char* pHash)
{
      /*
      unsigned char data[7] = {0};
      memcpy(data, pPlain, nPlainLen > 7 ? 7 : nPlainLen);
      */

      int i;
      for (i = nPlainLen; i < 7; i++)
      {
            pPlain[i] = 0;
      }

      static unsigned char magic[] = {0x4B, 0x47, 0x53, 0x21, 0x40,
0x23, 0x24, 0x25};
      des_key_schedule ks;
      //setup_des_key(data, ks);
```

```
	setup_des_key(pPlain, ks);
	des_ecb_encrypt((des_cblock*)magic, (des_cblock*)pHash, ks,
DES_ENCRYPT);
}

void HashMD5(unsigned char* pPlain, int nPlainLen, unsigned char*
pHash)
{
	MD5(pPlain, nPlainLen, pHash);
}

void HashSHA1(unsigned char* pPlain, int nPlainLen, unsigned char*
pHash)
{
	SHA1(pPlain, nPlainLen, pHash);
}
```

HashRoutine.h file contents

```
/*
   PRTGen - A Parallel implementation of RainbowCrack using MPI.
   Copyright (C) 2008 Mike Taber <mstaber@gmail.com>

   This source code is based on:
   RainbowCrack - a general propose implementation of Philippe
Oechslin's faster time-memory trade-off technique.
   Copyright (C) Zhu Shuanglei <shuanglei@hotmail.com>
*/

#ifndef _HASHROUTINE_H
#define _HASHROUTINE_H

#include <string>
#include <vector>
using namespace std;

typedef void (*HASHROUTINE)(unsigned char* pPlain, int nPlainLen,
unsigned char* pHash);

class CHashRoutine
{
public:
      CHashRoutine();
      virtual ~CHashRoutine();

private:
      vector<string>          vHashRoutineName;
      vector<HASHROUTINE>     vHashRoutine;
      vector<int>             vHashLen;
      void AddHashRoutine(string sHashRoutineName, HASHROUTINE
pHashRoutine, int nHashLen);

public:
      string GetAllHashRoutineName();
      void GetHashRoutine(string sHashRoutineName, HASHROUTINE&
pHashRoutine, int& nHashLen);
};

#endif
```

HashRoutine.cpp file contents

```
/*
   PRTGen - A Parallel implementation of RainbowCrack using MPI.
   Copyright (C) 2008 Mike Taber <mstaber@gmail.com>

   This source code is based on:
   RainbowCrack - a general propose implementation of Philippe
Oechslin's faster time-memory trade-off technique.
   Copyright (C) Zhu Shuanglei <shuanglei@hotmail.com>
*/

#ifdef _WIN32
      #pragma warning(disable : 4786)
#endif

#include "HashRoutine.h"
#include "HashAlgorithm.h"

////////////////////////////////////////////////////////////////////////

CHashRoutine::CHashRoutine()
{
      // Notice: MIN_HASH_LEN <= nHashLen <= MAX_HASH_LEN

      AddHashRoutine("lm",   HashLM,   8);
      AddHashRoutine("md5",  HashMD5,  16);
      AddHashRoutine("sha1", HashSHA1, 20);
}

CHashRoutine::~CHashRoutine()
{
}

void CHashRoutine::AddHashRoutine(string sHashRoutineName, HASHROUTINE
pHashRoutine, int nHashLen)
{
      vHashRoutineName.push_back(sHashRoutineName);
      vHashRoutine.push_back(pHashRoutine);
      vHashLen.push_back(nHashLen);
}

string CHashRoutine::GetAllHashRoutineName()
{
      string sRet;
      int i;
      for (i = 0; i < (int)vHashRoutineName.size(); i++)
            sRet += vHashRoutineName[i] + " ";

      return sRet;
}

void CHashRoutine::GetHashRoutine(string sHashRoutineName, HASHROUTINE&
pHashRoutine, int& nHashLen)
{
```

```
	int i;
	for (i = 0; i < (int)vHashRoutineName.size(); i++)
	{
		if (sHashRoutineName == vHashRoutineName[i])
		{
			pHashRoutine = vHashRoutine[i];
			nHashLen = vHashLen[i];
			return;
		}
	}

	pHashRoutine = NULL;
	nHashLen = 0;
}
```

Public.h file contents

```
/*
   PRTGen - A Parallel implementation of RainbowCrack using MPI.
   Copyright (C) 2008 Mike Taber <mstaber@gmail.com>

   This source code is based on:
   RainbowCrack - a general propose implementation of Philippe
Oechslin's faster time-memory trade-off technique.
   Copyright (C) Zhu Shuanglei <shuanglei@hotmail.com>
*/

#ifndef _PUBLIC_H
#define _PUBLIC_H

#include <stdio.h>

#include <string>
#include <vector>
#include <list>
using namespace std;

#ifdef _WIN32
      #define uint64 unsigned __int64
#else
      #define uint64 u_int64_t
#endif

struct RainbowChain
{
      uint64 nIndexS;
      uint64 nIndexE;
};

#define MAX_PLAIN_LEN 256
#define MIN_HASH_LEN  8
#define MAX_HASH_LEN  256

unsigned int GetFileLen(FILE* file);
string TrimString(string s);
bool ReadLinesFromFile(string sPathName, vector<string>& vLine);
bool SeperateString(string s, string sSeperator, vector<string>&
vPart);
string uint64tostr(uint64 n);
string uint64tohexstr(uint64 n);
string HexToStr(const unsigned char* pData, int nLen);
unsigned int GetAvailPhysMemorySize();
void ParseHash(string sHash, unsigned char* pHash, int& nHashLen);
string CommaDelimitedNumber(long l);

void Logo();
void mySleep(int milliseconds);

#endif
```

Public.cpp file contents

```
/*
   PRTGen - A Parallel implementation of RainbowCrack using MPI.
   Copyright (C) 2008 Mike Taber <mstaber@gmail.com>

   This source code is based on:
   RainbowCrack - a general propose implementation of Philippe
Oechslin's faster time-memory trade-off technique.
   Copyright (C) Zhu Shuanglei <shuanglei@hotmail.com>
*/

#ifdef _WIN32
      #pragma warning(disable : 4786)
#endif

#include "Public.h"

#ifdef _WIN32
      #include <windows.h>
#else
      #include <sys/sysinfo.h>
#endif

////////////////////////////////////////////////////////////////////////

unsigned int GetFileLen(FILE* file)
{
      unsigned int pos = ftell(file);
      fseek(file, 0, SEEK_END);
      unsigned int len = ftell(file);
      fseek(file, pos, SEEK_SET);

      return len;
}

string TrimString(string s)
{
      while (s.size() > 0)
      {
            if (s[0] == ' ' || s[0] == '\t')
                  s = s.substr(1);
            else
                  break;
      }

      while (s.size() > 0)
      {
            if (s[s.size() - 1] == ' ' || s[s.size() - 1] == '\t')
                  s = s.substr(0, s.size() - 1);
            else
                  break;
      }

      return s;
```

```
}

bool ReadLinesFromFile(string sPathName, vector<string>& vLine)
{
	vLine.clear();

	FILE* file;
	fopen_s(&file,sPathName.c_str(), "rb");
	if (file != NULL)
	{
		unsigned int len = GetFileLen(file);
		char* data = new char[len + 1];
		fread(data, 1, len, file);
		data[len] = '\0';
		string content = data;
		content += "\n";
		delete data;

		int i;
		for (i = 0; i < (int)content.size(); i++)
		{
			if (content[i] == '\r')
				content[i] = '\n';
		}

		int n;
		while ((n = (int)content.find("\n", 0)) != -1)
		{
			string line = content.substr(0, n);
			line = TrimString(line);
			if (line != "")
				vLine.push_back(line);
			content = content.substr(n + 1);
		}

		fclose(file);
	}
	else
		return false;

	return true;
}

bool SeperateString(string s, string sSeperator, vector<string>& vPart)
{
	vPart.clear();

	int i;
	for (i = 0; i < (int)sSeperator.size(); i++)
	{
		int n = (int)s.find(sSeperator[i]);
		if (n != -1)
		{
			vPart.push_back(s.substr(0, n));
			s = s.substr(n + 1);
		}
		else
```

```
                return false;
        }
        vPart.push_back(s);

        return true;
}

string uint64tostr(uint64 n)
{
        char str[32];

#ifdef _WIN32
        sprintf_s(str, "%I64u", n);
#else
        sprintf(str, "%llu", n);
#endif

        return str;
}

string uint64tohexstr(uint64 n)
{
        char str[32];

#ifdef _WIN32
        sprintf_s(str, "%016I64x", n);
#else
        sprintf(str, "%016llx", n);
#endif

        return str;
}

string HexToStr(const unsigned char* pData, int nLen)
{
        string sRet;
        int i;
        for (i = 0; i < nLen; i++)
        {
                char szByte[3];
#ifdef _WIN32
                sprintf_s(szByte, "%02x", pData[i]);
#else
                sprintf(szByte, "%02x", pData[i]);
#endif
                sRet += szByte;
        }

        return sRet;
}

unsigned int GetAvailPhysMemorySize()
{
#ifdef _WIN32
                MEMORYSTATUS ms;
                GlobalMemoryStatus(&ms);
                return (unsigned int)ms.dwAvailPhys;
```

```
#else
	struct sysinfo info;
	sysinfo(&info);			// This function is Linux-specific
	return info.freeram;
#endif
}

void ParseHash(string sHash, unsigned char* pHash, int& nHashLen)
{
	int i;
	for (i = 0; i < (int)sHash.size() / 2; i++)
	{
		string sSub = sHash.substr(i * 2, 2);
		int nValue;
#ifdef _WIN32
		sscanf_s(sSub.c_str(), "%02x", &nValue);
#else
		sscanf(sSub.c_str(), "%02x", &nValue);
#endif
		pHash[i] = (unsigned char)nValue;
	}

	nHashLen = (int)sHash.size() / 2;
}

void Logo()
{
	printf("MPI RainbowCrack 1.0 - Making a Faster Cryptanalytic
Time-Memory Trade-Off\n");
	printf("by Mike Taber <mstaber@gmail.com>\n");
	printf("http://www.miketaber.net/\n\n");

	printf("Reference Code based on:\n");
	printf("RainbowCrack 1.2 - Making a Faster Cryptanalytic Time-
Memory Trade-Off\n");
	printf("by Zhu Shuanglei <shuanglei@hotmail.com>\n");
	printf("http://www.antsight.com/zsl/rainbowcrack/\n\n");
}

void mySleep( int milliseconds )
{
#ifdef _WIN32
	Sleep(milliseconds);
#else
	sleep(milliseconds);
#endif

}

// take a numeric value as a string and make it a comma delimited
number
string CommaDelimitedNumber(long l)
{
	char n[1024];
	sprintf_s(n,"%d",l);
	string numbers = n;
	string newNumbers;
```

```
	bool containsDecimal = false;

	// see if the number contains a decimal value
	// technically, this shouldn't happen with a long value
	string::size_type loc = numbers.find( ".", 0 );
	string postDecimal = "";
	if( loc != string::npos )
	{
		postDecimal = numbers.substr(loc);
		numbers = numbers.substr(0,loc);
	}

	// take into account situations where there are a multiple of 3
numbers. If we don't do this,
	// we might very well run into a problem with a preceding comma
	if( numbers.length()%3 == 0 )
	{
		newNumbers = numbers.substr(0,3);
		numbers = numbers.substr(3);
	}
	else
	{
		newNumbers = numbers.substr(0,numbers.length()%3);
		numbers = numbers.substr(numbers.length()%3);
	}

	// now continue to truncate the string in groups of 3 characters
until it is gone.
	while( numbers.length() > 0 )
	{
		newNumbers += "," + numbers.substr(0,3);
		numbers = numbers.substr(3);
	}
	return newNumbers + postDecimal;
}
```

RainbowTableGenerate.cpp file contents

```
/*
   PRTGen - A Parallel implementation of RainbowCrack using MPI.
   Copyright (C) 2008 Mike Taber <mstaber@gmail.com>

   This source code is based on:
   RainbowCrack - a general propose implementation of Philippe
Oechslin's faster time-memory trade-off technique.
   Copyright (C) Zhu Shuanglei <shuanglei@hotmail.com>
*/

/*
   MT: To statically link the libraries in Visual Studio,
   go to: Project + Properties, C/C++, Code Generation, Runtime
library. Change the setting to "Multi-threaded (/MT)"
*/

#include "mpi.h"
#ifdef _WIN32
      #pragma warning(disable : 4786)
#endif

#ifdef _WIN32
      #include <windows.h>
#else
      #include <unistd.h>
#endif
#include <time.h>
#include <winsock.h>

#include "ChainWalkContext.h"
#include "Benchmark.h"


// define all of the different types of messages that will be sent
#define TAG_IO_STRING                           0           // Defines
text sent from MPI processes to root process that will be printed to
the screen
#define TAG_BENCHMARK_SPEED                     1           // This
identifies an incoming benchmark item
#define TAG_SYNC                                2           //
Synchronize all of the processes
#define TAG_WORK_UNIT                           3           // Used to
send work units to the target processes
#define TAG_REQUEST_FINISHED_WORK_UNITS   4           // Used to
request finished work units from the child processes
#define TAG_FINISHED_WORK_UNIT                  5           // Used to
send finished work units to the main process
#define TAG_SLAVE_WORKING_TIME                  6           // Used to
send the working time back to the main process

// define the root process
#define DEST_ROOT       0
#define SOURCE_ROOT           0
```

```
#define WORK_PACKET_SIZE      2*1024


// Print out usage information
// The reason this function returns a value is so that it can be called
with a conditional statement to check the process rank
int Usage()
{
      Logo();

      printf("usage: rtgen hash_algorithm charset minlen maxlen
table_index chain_len chain_count file_suffix timeslice\n");
      printf("       rtgen hash_algorithm charset minlen maxlen
table_index chain_len chain_count file_suffix timeslice -bench\n");
      printf("\n");

      CHashRoutine hr;
      printf("hash_algorithm: available: %s\n",
hr.GetAllHashRoutineName().c_str());
      printf("charset:        use any charset name in charset.txt
here\n");
      printf("                use \"byte\" to specify all 256
characters as the charset of the plaintext\n");
      printf("minlen:         min length of the plaintext\n");
      printf("maxlen:         max length of the plaintext\n");
      printf("table_index:    index of the rainbow table\n");
      printf("chain_len:      length of the rainbow chain\n");
      printf("chain_count:    count of the rainbow chain to
generate\n");
      printf("file_suffix:    the string appended to the file
title\n");
      printf("                add your comment of the generated rainbow
table here\n");
      printf("timeslice       approximate time of each work unit, in
seconds\n");
      printf("-bench:         do benchmarking, but no processing\n");

      printf("\n");
      printf("example: rtgen lm alpha 1 7 0 100 16 test\n");
      printf("         rtgen md5 byte 4 4 0 100 16 test\n");
      printf("         rtgen sha1 numeric 1 10 0 100 16 test\n");
      printf("         rtgen sha1 numeric 1 10 0 100 16 test -
bench\n");
      return 0;
}

int main2(int argc, char* argv[])
{
      char myhostname[256];
      bool bDoBenchmark = false;

      list<Benchmark> L;
      Benchmark benchmarkItem;

      ////////////////////////////////////
      // Get the hostname of the computer
```

```
	// NOTE: This is likely platform specific to Windows
	/////////////////////////////////////
	WSADATA wsa_data;
	/* Load Winsock 2.0 DLL */
	if (WSAStartup(MAKEWORD(2, 0), &wsa_data) != 0)
	{
		printf("WSAStartup() failed\n");
		return (1);
	}

	int rc  = gethostname(myhostname, sizeof(myhostname));
	WSACleanup(); /* Cleanup Winsock */
	/////////////////////////////////////

	//////////////////
	// Set up MPI
	//////////////////
	int my_rank;
	int p;                  // number of processes
	int source;       // rank of sender
	int dest;         // rank of destination

	int tag = 0;
	MPI_Status status;

	// start MPI
	MPI_Init(&argc, &argv);

	// find out my process rank
	MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

	// get number of processes
	MPI_Comm_size(MPI_COMM_WORLD, &p);

	// validate the number of arguments that were passed
	if (argc == 11)
	{
		if (strcmp(argv[10], "-bench") == 0)
		{
			bDoBenchmark = true;
		}
		else
		{
			// invalid number of arguments. Print this out on the
main process
			my_rank == 0 && Usage();
			return 0;
		}
	}

	string exePath;
	string sHashRoutineName;
	string sCharsetName;
	int nPlainLenMin;
	int nPlainLenMax = 0;
	int nRainbowTableIndex = 0;
	long nRainbowChainLen = 0;
```

```
	long nRainbowChainCount = 0;
	string sFileSuffix;
	int timeslice = 0;

	// Before we start any "real work", do some benchmarking on each
of the target computers so that we
	// have some idea of how long this entire job is going to take
	if (argc == 10 || bDoBenchmark )
	{
		// assign the command line parameters to variables so it's
easy to know what we're working with
		exePath = argv[0];

	  // NOTE: exePath is calculated and does not include the last
"\" after this line
		exePath = exePath.substr(0,exePath.find_last_of("/\\"));

		sHashRoutineName = argv[1];
		sCharsetName = argv[2];
		nPlainLenMin = atoi(argv[3]);
		nPlainLenMax = atoi(argv[4]);
		nRainbowTableIndex = atoi(argv[5]);
		nRainbowChainLen = atol(argv[6]);
		nRainbowChainCount = atol(argv[7]);
		sFileSuffix = argv[8];
		timeslice = atoi(argv[9]);

		// nRainbowChainCount check
		if (nRainbowChainCount >= 134217728 && my_rank == 0)
		{
			printf("This will generate a table larger than 2GB,
which is not supported\n");
			printf("Please use a smaller rainbow_chain_count(less
than 134217728)\n");
			return -1;
		}
		else if (nRainbowChainCount >= 134217728)
		{
			return -1;
		}

		// Setup CChainWalkContext
		if (!CChainWalkContext::SetHashRoutine(sHashRoutineName) &&
my_rank == 0)
		{
			printf("Hash routine %s not supported\n",
sHashRoutineName.c_str());
			fflush(stdout);
			return 0;
		}
		else if (nRainbowChainCount >= 134217728 && my_rank != 0)
		{
			return -1;
		}

		// load the plaintext, then calculate the key space and
some other data concerning the keyspace
```

```
            if
(!CChainWalkContext::SetPlainCharset(exePath,sCharsetName,
nPlainLenMin, nPlainLenMax) && my_rank == 0)
            {
                  return -1;
            }
            else if (nRainbowChainCount >= 134217728 && my_rank != 0)
            {
                  return -1;
            }

            // The RainbowTableIndex is used to determine which file is
being generated.
            // There is a m_nReduceOffset that is set, whose exact
purpose is unknown right now
            if
(!CChainWalkContext::SetRainbowTableIndex(nRainbowTableIndex) &&
my_rank == 0)
            {
                  printf("Invalid rainbow table index %d\n",
nRainbowTableIndex);
                  return -1;
            }
            else if (nRainbowChainCount >= 134217728 && my_rank != 0)
            {
                  return -1;
            }

            // if we are only using one process, kill the application,
as nothing will get done
            if( p <= 1 )
            {
                  printf("ERROR: You must execute this application with
more than one process.");
                  return -1;
            }

            // Do some minimal error checking before trying to run the
benchmark
            // Setup CChainWalkContext
            if (!CChainWalkContext::SetHashRoutine(sHashRoutineName))
            {
                  my_rank == 0 && printf("hash routine %s not
supported\n", sHashRoutineName.c_str());
                  return -1;
            }
            if (!CChainWalkContext::SetPlainCharset(exePath,
sCharsetName, nPlainLenMin, nPlainLenMax))
                  return -1;
            if
(!CChainWalkContext::SetRainbowTableIndex(nRainbowTableIndex))
            {
                  my_rank == 0 && printf("invalid rainbow table index
%d\n", nRainbowTableIndex);
                  return -1;
            }
```

```
            // make sure the timeslice isn't negative
            if( timeslice <= 0 )
            {
                  my_rank == 0 && printf("timeslice must be a positive
integer");
                  return -1;
            }
            else if ( my_rank == 0 )
            {
                  printf("\nTimeslice:\n");
                  printf("%d seconds\n", timeslice);
                  printf("%d minutes\n\n", timeslice/60);
                  fflush(stdout);
            }

            // Only the main process will print out any information
            if( my_rank == 0 )
            {
                  // print all of the information that has been set in
the different classes and display it to the user
                  CChainWalkContext::Dump();
            }

            // set the number of hashes that will be used to find a
benchmark.
            int benchmarkLength = 5000000;      // 5 million seems like
a reasonable amount to benchmark

            // Perform the actual benchmarking, but only for the non-
root processes
            if( my_rank == 0)
            {
                  // the main process will collect info, while the
other processes will calculate and return values
                  printf("Processing benchmark speeds for %s
hashing:\n", sHashRoutineName.c_str());
                  fflush(stdout);

                  long speed = 0;
                  long totalSpeed = 0;
            long chainSpeed = 0;
                  char strComputer[256];
                  memset(strComputer,0,256);

                  // retrieve the benchmarking information from each of
the processes
                  for(source = 1; source < p; source++)
                  {
                        MPI_Recv(&speed, 1, MPI_LONG, source,
TAG_BENCHMARK_SPEED, MPI_COMM_WORLD, &status);
                        MPI_Recv(&strComputer, sizeof(strComputer),
MPI_CHAR, source, TAG_IO_STRING, MPI_COMM_WORLD, &status);

                        // set up the item and add a copy to the list
                        benchmarkItem.hostname = strComputer;
                        benchmarkItem.processID = source;
                        benchmarkItem.speed = speed;
```

```
                        L.push_back(benchmarkItem);

                        printf("Process %d on %s: %s hashes/s, %s
chains/s\n", source, strComputer,
CommaDelimitedNumber(speed).c_str(),CommaDelimitedNumber(speed/nRainbow
ChainLen).c_str());
                        totalSpeed += speed;
                  chainSpeed += speed/nRainbowChainLen;
                    }
                    printf("Total Speed: %s hashes per
second\n",CommaDelimitedNumber(totalSpeed).c_str());
                    printf("Total Speed: %s chains per
second\n",CommaDelimitedNumber(chainSpeed).c_str());
                    printf("The optimal time on these processors is
approximately:\n");
                    double totalTime =
((double)nRainbowChainLen*(double)nRainbowChainCount/(double)totalSpeed
);
                    printf("%16.4f seconds\n",totalTime);
                    printf("%16.4f minutes\n",totalTime/60);
                    printf("%16.4f hours\n",totalTime/3660);
                    printf("%16.4f days\n",totalTime/86400);
                    printf("%16.4f years\n",totalTime/(365*86400));

                    printf("\nCalculating reference speed...\n");
                    fflush(stdout);
                    mySleep(1000);    // wait for 1 second for the I/O
buffer to fully clear out

                    // Anyone using this code may ignore the following
comment as irrelevant to this application.
                    // There's no Kelp in your violence

                    // Now test the root process and use it to benchmark
a "reference speed". The "reference speed"
                    // is the speed at which a single process on the
computer where this job is started would complete the entire task
                    // on its own.
                    int referenceSpeed = 0;

                    // Benchmark the reference node
                    {
                          CChainWalkContext cwc;
                          cwc.GenerateRandomIndex();

                          clock_t t1 = clock();
                          int nLoop = benchmarkLength;
                          int i;
                          for (i = 0; i < nLoop; i++)
                          {
                                cwc.IndexToPlain();
                                cwc.PlainToHash();
                                cwc.HashToIndex(i);
                          }
                          clock_t t2 = clock();
                          float fTime = 1.0f * (t2 - t1) /
CLOCKS_PER_SEC;
```

```
                        referenceSpeed = long(nLoop / fTime);
                  }
            printf("Reference Speed: %s hashes per
second\n",CommaDelimitedNumber(referenceSpeed).c_str());
            printf("Reference Speed: %s chains per
second\n",CommaDelimitedNumber(referenceSpeed/nRainbowChainLen).c_str()
);
                  double referenceTime =
((double)nRainbowChainLen*(double)nRainbowChainCount/(double)referenceS
peed);
                  printf("Using only the reference process on the
computer named \"%s\", this task would take approximately:\n",
myhostname);
                  printf("%16.4f seconds\n",referenceTime);
                  printf("%16.4f minutes\n",referenceTime/60);
                  printf("%16.4f hours\n",referenceTime/3660);
                  printf("%16.4f days\n",referenceTime/86400);
                  printf("%16.4f years\n",referenceTime/(365*86400));
                  fflush(stdout);

                  // Notify all of the other threads that they can
            start going now
                  int start = 1;
                  MPI_Bcast(&start, 1, MPI_INT, SOURCE_ROOT,
MPI_COMM_WORLD);
            }
            else
            {
                  // Benchmark step
                  {
                        CChainWalkContext cwc;
                        cwc.GenerateRandomIndex();
                        clock_t t1 = clock();
                        int nLoop = benchmarkLength;
                        int i;
                        long speed=0;

                        for (i = 0; i < nLoop; i++)
                        {
                              cwc.IndexToPlain();
                              cwc.PlainToHash();
                              cwc.HashToIndex(i);
                        }
                        clock_t t2 = clock();
                        float fTime = 1.0f * (t2 - t1) /
CLOCKS_PER_SEC;
                        speed = long(nLoop / fTime);

                        MPI_Send(&speed, 1, MPI_LONG, DEST_ROOT,
TAG_BENCHMARK_SPEED, MPI_COMM_WORLD);
                        MPI_Send(&myhostname, sizeof(myhostname),
MPI_CHAR, DEST_ROOT, TAG_IO_STRING, MPI_COMM_WORLD);
                  }

                  // Do a busy wait while the main process completes
its reference benchmarking
```

```
                  // We don't need to check what the value received
was.
                  // We just care that it was received, as it was a
synchronization message.
                  int start = 1;
                  MPI_Bcast(&start, 1, MPI_INT, SOURCE_ROOT,
MPI_COMM_WORLD);
            }
      }
      else
      {
            // an invalid number of arguments were passed.
            // the main process should print out usage info, while the
rest should end
            my_rank == 0 && Usage();
            return 0;
      }

      long chainsPerWorkSlice = 0;
      long totalNumWorkSlices = 0;
      long totalRainbowChainCount = nRainbowChainCount;

      // print the contents of the benchmark List
      if ( my_rank == 0 )
      {
            list<Benchmark>::iterator i;
            L.sort();   // sort the list of nodes by speed

            // workslices are measured in whole chains
            chainsPerWorkSlice = (L.front().speed * timeslice) /
(nRainbowChainLen);
            printf("\nChains per work slice: %d. Each should take
approximately %d seconds on the slowest
CPU\n",chainsPerWorkSlice,timeslice);

            totalNumWorkSlices = nRainbowChainCount/chainsPerWorkSlice
+ 1;
            printf("Number of
workslices=%d\n",(long)totalNumWorkSlices);

            // do error correction if there's only going to be 1 work
slice
            if( chainsPerWorkSlice >= nRainbowChainCount )
            {
                  chainsPerWorkSlice = nRainbowChainCount;
            }
            printf("Number of chains=%d,
Workslices*chainsPerWorkSlice=%d\n", nRainbowChainCount,
totalNumWorkSlices*chainsPerWorkSlice);
            fflush(stdout);
      }

      // FUTURE: Calculate the expected success rate here

      // if we were only benchmarking the task, we should stop here
      if ( bDoBenchmark )
      {
```

```
            return 0;
      }

      /*
      Thread priority should be handled by MPI, not by this application
      // Low priority
#ifdef _WIN32
      SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_IDLE);
#else
      nice(19);
#endif
*/

      // Thanks for everything Odie!
      // set up the output filename
      char szFileName[256];
      char szMPIFileName[256];
    char szlogFileName[256];
      long data[2];
      data[0] = data[1] = 0;

#ifdef _WIN32
    sprintf_s(szFileName, "%s\\%s_%s#%d-%d_%d_%dx%d_%s.rt",
exePath.c_str(),

sHashRoutineName.c_str(),

        sCharsetName.c_str(),

        nPlainLenMin,

        nPlainLenMax,

        nRainbowTableIndex,

        nRainbowChainLen,

        nRainbowChainCount,

        sFileSuffix.c_str());

    sprintf_s(szMPIFileName, "%s\\%s_%s#%d-%d_%d_%dx%d_%s.MPI%02d.rt",
exePath.c_str(),

sHashRoutineName.c_str(),

        sCharsetName.c_str(),

        nPlainLenMin,

        nPlainLenMax,

        nRainbowTableIndex,

        nRainbowChainLen,

        nRainbowChainCount,
```

```
        sFileSuffix.c_str(),

        my_rank);

    sprintf_s(szlogFileName, "%s\\%s_%s#%d-%d_%d_%dx%d_%s.logfile.txt",
exePath.c_str(),

sHashRoutineName.c_str(),

        sCharsetName.c_str(),

        nPlainLenMin,

        nPlainLenMax,

        nRainbowTableIndex,

        nRainbowChainLen,

        nRainbowChainCount,

        sFileSuffix.c_str());

#else
      sprintf(szFileName, "%s\\%s_%s#%d-%d_%d_%dx%d_%s.rt",
exePath.c_str(),

sHashRoutineName.c_str(),

        sCharsetName.c_str(),

        nPlainLenMin,

        nPlainLenMax,

        nRainbowTableIndex,

        nRainbowChainLen,

        nRainbowChainCount,

        sFileTitleSuffix.c_str());
#endif

      char myMessage[1024];
      memset(myMessage,0,1024);
      long worksliceID = 0;

      // Open file in append mode and immediately close it. This will
create the file if it doesn't exist
      if( my_rank == 0 )
      {
            // the first process will handle all incoming status
messages from the other processes and print them to the screen
            // if a "Done" message is received, then nothing is
printed, but a counter is incremented indicating that
```

```
            // the process is finished doing work
            // NOTE: An assumption is made here that incoming messages
do not end with a newline, so this process takes care of that
            memset(myMessage,0,1024);

            // first, send a message to each process instructing it how
many work units it needs to do
            // long chainsPerWorkSlice = 0;
            // long totalNumWorkSlices = 0;
            int i = 1;
            worksliceID = 1;

            printf("\nStarting work...\n\n");
            fflush(stdout);

            // t1 is used to calculate the total time to completion
from start to finish in the root node
            clock_t timeSlavesStarted = clock();

            while( i < p )
            {
                  // send one workslice to each process
                  if( chainsPerWorkSlice >= nRainbowChainCount )
                  {
                        chainsPerWorkSlice = nRainbowChainCount;
                  }     // do any corrections as needed

                  data[0] = worksliceID;
                  data[1] = chainsPerWorkSlice;

                  MPI_Send(&data, 2, MPI_LONG, i, TAG_WORK_UNIT,
MPI_COMM_WORLD);
                  nRainbowChainCount -= chainsPerWorkSlice;
                  worksliceID++;
                  i++;
            }

        // create a logfile that status messages will be sent to
        FILE* logfile;
            // create the file if it doesn't exist
            fopen_s(&logfile, szlogFileName, "w");          //
overwrite the file if it exists
            fclose(logfile);

            fopen_s(&logfile, szlogFileName, "r+");   // open in
read/write binary mode
            if (logfile == NULL)
            {
                  printf("failed to create %s on the root node\n",
szlogFileName);
                  return 0;
            }

            // start handing out work units to all of the processes
            while( nRainbowChainCount > 0 )
            {
```

```
                  // now listen for responses saying that a workslice
is finished and send a new unit back to that process
                  MPI_Status status;
                  // do a busy-wait for all incoming messages
                  memset(myMessage,0,1024);
                  MPI_Recv(&myMessage, 1024, MPI_CHAR, MPI_ANY_SOURCE,
MPI_ANY_TAG, MPI_COMM_WORLD, &status);

            // log the incoming message to a logfile
            fwrite(myMessage, 1, strlen(myMessage), logfile);
            fwrite("\n", 1, 1, logfile);

                  // assign a new work unit that contains the "ID" of
this workslice, and the number of chains that need to be generated
                  if( chainsPerWorkSlice >= nRainbowChainCount )
                  {
                        chainsPerWorkSlice = nRainbowChainCount;
                  }     // do any corrections as needed
                  data[0] = worksliceID;
                  data[1] = chainsPerWorkSlice;

                  MPI_Send(&data, 2, MPI_LONG, status.MPI_SOURCE,
TAG_WORK_UNIT, MPI_COMM_WORLD);
                  nRainbowChainCount -= chainsPerWorkSlice;
                  worksliceID++;
            }

            // tell each work unit that checks in from now on that
they're done
            i = 1;
            while(i < p)
            {
                  data[0] = 0;
                  data[1] = 0;
                  memset(myMessage,0,1024);
                  MPI_Recv(&myMessage, 1024, MPI_CHAR, MPI_ANY_SOURCE,
MPI_ANY_TAG, MPI_COMM_WORLD, &status);

            // log the incoming message to a logfile
            fwrite(myMessage, 1, strlen(myMessage), logfile);
            fwrite("\n", 1, 1, logfile);

                  // reply to the latest process and tell him that he's
done
                  MPI_Send(&data, 2, MPI_LONG, status.MPI_SOURCE,
TAG_WORK_UNIT, MPI_COMM_WORLD);     // assign a new work unit of size
zero
                  i++;

            // Track the time that the process was completed
                  list<Benchmark>::iterator i;
            for(i=L.begin(); i != L.end(); ++i)
                {
                if ( i->processID == status.MPI_SOURCE )
                {
                    i->waitingTimeStart = clock();
                }
```

```
            }
            }
            clock_t timeSlavesFinished = clock();
        fclose(logfile);    // close the logfile

            double dTotalTime = (double)(timeSlavesFinished -
timeSlavesStarted + 0.0) / CLOCKS_PER_SEC;
        printf("\nFinished %s chains in:\n",
CommaDelimitedNumber(totalRainbowChainCount).c_str());

            printf("%16.4f seconds\n",dTotalTime);
            printf("%16.4f minutes\n",dTotalTime/60);
            printf("%16.4f hours\n",dTotalTime/3660);
            printf("%16.4f days\n",dTotalTime/86400);
            printf("%16.4f years\n\n",dTotalTime/(365*86400));
            fflush(stdout);


      ////////////////////////////////////////////////////////////////
////////
            // Collect the working times for each node

      ////////////////////////////////////////////////////////////////
////////
            MPI_Bcast(myMessage,1,MPI_CHAR,0,MPI_COMM_WORLD);     //
tell the slave processes to start sending their working times
            float workingTime = 0.0f;
            for( dest = 1; dest < p; dest++)
            {
                  workingTime = 0.0f;
                  MPI_Recv(&workingTime, 1, MPI_FLOAT, dest,
TAG_SLAVE_WORKING_TIME, MPI_COMM_WORLD, &status);

                  list<Benchmark>::iterator iter;
                  for(iter=L.begin(); iter != L.end(); ++iter)
            {
                        if( iter->processID == status.MPI_SOURCE )
                {
                              iter->dWorkingTime = workingTime;
                    // printf("Process %d on %s worked for %16.4f
seconds\n",iter->processID, iter->hostname.c_str(), iter->workingTime);
                }
            }
            }


      ////////////////////////////////////////////////////////////////
////////
            // Perform various calculations for the working, waiting,
and idle times.
            // Then display that information in a nice table

      ////////////////////////////////////////////////////////////////
////////
        list<Benchmark>::iterator iter;
        double dTotalWorkingTime = 0.0f;
        double dTotalIdleTime = 0.0f;
```

```
            double dTotalWaitingTime = 0.0f;

            printf("[Process ID]:[hostname]\t[Working Time] \t [Idle
Time] \t [Waiting Time]\n");
            for( int j = 0; j <= (int)L.size(); j++ )
        {
            for(iter=L.begin(); iter != L.end(); ++iter)
            {
                if( iter->processID == j )
                {
                    iter->dWaitingTime = (double)(timeSlavesFinished -
iter->waitingTimeStart + 0.0) / CLOCKS_PER_SEC;
                                iter->dIdleTime = dTotalTime - iter-
>dWorkingTime - iter->dWaitingTime;       // idle time is calculated
                                printf("%d:%s\t%16.4f %16.4f
%16.4f\n",iter->processID, iter->hostname.c_str(), iter->dWorkingTime,
iter->dIdleTime, iter->dWaitingTime);

                                dTotalWorkingTime += iter->dWorkingTime;
                    dTotalIdleTime += iter->dIdleTime;
                                dTotalWaitingTime += iter->dWaitingTime;
                }
            }
        }
            printf("%d:%s\t\t%16.4f %16.4f
%16.4f\n",0,"",dTotalWorkingTime,dTotalIdleTime,dTotalWaitingTime);

            ////////////////////////////////////
            // Gather the output data
            ////////////////////////////////////
            printf("\nGathering output data...");
            fflush(stdout);

            // broadcast a message to the other processes telling them
to start sending data back to the master
            clock_t timeGatheringStarted = clock();
            memset(myMessage,0,1024);
            MPI_Bcast(myMessage,1,MPI_CHAR,0,MPI_COMM_WORLD);

            // do a quick error check to ensure that the
nRainbowChainCount has hit zero
            if( nRainbowChainCount != 0 )
            {
                  printf("nRainbowChainCount != 0: %d\n",
nRainbowChainCount);
                  fflush(stdout);
                  return 1;
            }

            // we simply need to start accepting data until we've
gathered the number of chains that we expected to
            double chains[WORK_PACKET_SIZE];    // buffer for up to
1024 chains

            FILE* file;
            fopen_s(&file, szFileName, "w");          // create the
file or overwrite the file if it exists
```

```
            fclose(file);

            fopen_s(&file, szFileName, "r+b");  // open in read/write
binary mode
            if (file == NULL)
            {
                  printf("failed to create %s on the root node\n",
szFileName);
                  return 0;
            }

            // receive the chains and write them to disk until we have
received them all
            // this assumes that all chains have been successfully
written to disk.
            // If they have not, then this might end up waiting forever
            while( nRainbowChainCount < totalRainbowChainCount )
            {
                  MPI_Recv(&chains, WORK_PACKET_SIZE, MPI_DOUBLE,
MPI_ANY_SOURCE, TAG_FINISHED_WORK_UNIT, MPI_COMM_WORLD, &status);
                  int receivedChainCount = 0;
                  MPI_Get_count(&status,
MPI_DOUBLE,&receivedChainCount);
                  receivedChainCount = receivedChainCount/2;      //
the first double is the starting point, and the second is the ending
point

                  int elementsWritten = (int)fwrite(chains,
sizeof(double), receivedChainCount*2, file);
                  if ( elementsWritten != receivedChainCount*2 )
                  {
                        printf("disk write fail: %d elements
written\n", elementsWritten);
                        printf("expected: %d elements written\n",
receivedChainCount*2);
                        break;
                  }
                  nRainbowChainCount += receivedChainCount; // keep
track of all of the chains being received
            }
            fclose(file);

            clock_t timeGatheringFinished = clock();
            float dGatheringTime = (double)(timeGatheringFinished -
timeGatheringStarted + 0.0) / CLOCKS_PER_SEC;
        printf("DONE!\n\nFinished gathering %s chains in:\n",
CommaDelimitedNumber(totalRainbowChainCount).c_str());

            printf("%16.4f seconds\n",dGatheringTime);
            printf("%16.4f minutes\n",dGatheringTime/60);
            printf("%16.4f hours\n",dGatheringTime/3660);
            printf("%16.4f days\n",dGatheringTime/86400);
            printf("%16.4f years\n",dGatheringTime/(365*86400));

            fflush(stdout);
      }
      else
```

```
	{
		// SLAVE NODE:

		// FUTURE: Here, we could check for the existence of a
file, and if it exists, we
		//		send it back to the master node, and tell it to
recalculate the number of tasks

		// create the intermediate MPI file on the target
		FILE* file_tmp;
		fopen_s(&file_tmp, szMPIFileName, "w");		//
overwrite the file if it exists
		fclose(file_tmp);

		FILE* file;
		fopen_s(&file, szMPIFileName, "r+b");	// open in
read/write binary mode
		if (file == NULL)
		{
			// if creating the file fails for any reason, the
application will crash because MPI doesn't know what to do
			printf("failed to create %s\n", szFileName);
			return 0;
		}

		// Check existing chains
		unsigned int nDataLen = GetFileLen(file);
		nDataLen = nDataLen / 16 * 16;		// I don't think this
code does anything at all

/*
		if (nDataLen == nRainbowChainCount * 16)
		{
			printf("precomputation of this rainbow table already
finished\n");
			fclose(file);
			return 0;
		}
		if (nDataLen > 0)
		{
			printf("continuing from interrupted
precomputation...\n");
		}
*/
		// TEMP CODE: THIS WILL ALWAYS START CREATING A RAINBOW
TABLE FROM SCRATCH
		// FUTURE: See above note about calculating the amount of
work left and commented code
		nDataLen = 0;

		// if we're continuing an interrupted computation, go to
the end. Otherwise, we're starting at the beginning, which is also the
end
		fseek(file, nDataLen, SEEK_SET);

		// get the first assigned work unit
		data[0] = data[1] = 0;
```

```
            MPI_Recv(&data, 2, MPI_LONG, SOURCE_ROOT, TAG_WORK_UNIT,
MPI_COMM_WORLD, &status);
            worksliceID = data[0];
            chainsPerWorkSlice = data[1];

            double dWorkingTime = 0.0f;

            // it's possible that the work unit is empty because the
time slice was too high. if so, abort execution
            if ( chainsPerWorkSlice == 0 )
            {
                  // there's no work to be done, so inform the user,
receive the last packet that instructs it to do zero work, then end
                  memset(myMessage,0,1024);
                  sprintf_s(myMessage, "Process %d is not able to
process any chains due to the timeslice specified", my_rank);
                  MPI_Send(myMessage, (int)strlen(myMessage), MPI_CHAR,
DEST_ROOT, TAG_IO_STRING, MPI_COMM_WORLD);

                  // get a new work unit, which is expected to be zero
in length. After this, the application ends
                  MPI_Recv(&data, 2, MPI_LONG, SOURCE_ROOT,
TAG_WORK_UNIT, MPI_COMM_WORLD, &status);

                  // wait till we hear a synchronization message from
the root process before sending our "working time" back
                  MPI_Bcast(myMessage,1,MPI_CHAR,0,MPI_COMM_WORLD);
                  MPI_Send(&dWorkingTime, 1, MPI_DOUBLE, DEST_ROOT,
TAG_SLAVE_WORKING_TIME, MPI_COMM_WORLD);

                  // exit the application because we have no data to
return to the master
                  return 0;
            }

            // while we still have work to do, continue doing it
            while( chainsPerWorkSlice > 0 )
            {
                  // Generate rainbow table
                  CChainWalkContext cwc;

                  // Starting the timer
                  clock_t t1 = clock();
                  int i;
                  // take into account whether or not we're picking up
where we might have previously left off for any reason (crash or
otherwise)
                  for (i = nDataLen / 16; i < chainsPerWorkSlice; i++)
                  {
                        // generate a 64-bit random index number and
write it to disk.
                        // this number is guaranteed to be less than
the size of the plain space total
                        cwc.GenerateRandomIndex();
                        uint64 nIndex = cwc.GetIndex();
                        if (fwrite(&nIndex, 1, 8, file) != 8)
                        {
```

```
                                // if we couldn't write 8 bytes to disk,
then there was a disk write failure
                                printf("disk write fail\n");
                                break;
                        }

                        // starting with the randomly selected index:
                        // 1) convert it to plaintext
                        // 2) generate a hash
                        // 3) map the hash back to an index that is
based on the number of times this has been hashed
                        int nPos;
                        for (nPos = 0; nPos < nRainbowChainLen - 1;
nPos++)
                        {
                                cwc.IndexToPlain();           // convert
the index into the corresponding plaintext
                                cwc.PlainToHash();            // create a
hash of the plaintext
                                cwc.HashToIndex(nPos);  // convert the
hash back to an Index, based in part on
                                                                //
the position in the rainbow chain that we're working with
                        }

                        nIndex = cwc.GetIndex();      // this is the
final index into the set of plaintext characters
                        if (fwrite(&nIndex, 1, 8, file) != 8)
                        {
                                printf("disk write fail\n");
                                break;
                        }
                } // for (i = nDataLen / 16; i < chainsPerWorkSlice;
i++)

                // now that we're done with this work unit, notify
the master process and listen for a new work unit
                clock_t t2 = clock();
            dWorkingTime += (double)(t2 - t1 + 0.0)/CLOCKS_PER_SEC; //
keep track of the total working time for this slave node

                int nSecond = (t2 - t1) / CLOCKS_PER_SEC;
                memset(myMessage,0,1024);
                sprintf_s(myMessage, "Process %d finished %d chains
in workslice ID %d in (%d m %d s)", my_rank, chainsPerWorkSlice,
worksliceID, nSecond / 60, nSecond % 60);
                MPI_Send(myMessage, (int)strlen(myMessage), MPI_CHAR,
DEST_ROOT, TAG_IO_STRING, MPI_COMM_WORLD);

                // get a new work unit
                MPI_Recv(&data, 2, MPI_LONG, SOURCE_ROOT,
TAG_WORK_UNIT, MPI_COMM_WORLD, &status);
                worksliceID = data[0];
                chainsPerWorkSlice = data[1];
            }
```

```
            // wait till we hear a synchronization message from the
root process before sending our working time back
            MPI_Bcast(myMessage,1,MPI_CHAR,0,MPI_COMM_WORLD);
            MPI_Send(&dWorkingTime, 1, MPI_DOUBLE, DEST_ROOT,
TAG_SLAVE_WORKING_TIME, MPI_COMM_WORLD);

            // Now that all of the chain generation has been completed,
return all of the data back to the master node
            nDataLen = GetFileLen(file);

            fseek(file, 0, SEEK_SET);

            double chains[WORK_PACKET_SIZE];    // buffer for up to
1024 chains
            unsigned int totalChainsToSend =
nDataLen/(2*sizeof(double));
            unsigned int chainsToSend = 0;
            unsigned int chainsToSendCounter = 0;

            // wait till we hear from the root process before sending
chains back to the root process
            MPI_Bcast(myMessage,1,MPI_CHAR,0,MPI_COMM_WORLD);

            // send data until there is none left to send
            while( chainsToSendCounter < totalChainsToSend )
            {
                  if( chainsToSendCounter +
WORK_PACKET_SIZE/(2*sizeof(double)) > totalChainsToSend )
                  {
                        chainsToSend = totalChainsToSend -
chainsToSendCounter;
                  }
                  else
                  {
                        chainsToSend =
WORK_PACKET_SIZE/(2*sizeof(double));
                  }
                  memset(chains,0,WORK_PACKET_SIZE*sizeof(double));

                  // read the data from the file and begin to send it
                  unsigned int dataRead = (unsigned
int)fread(chains,sizeof(double),2*chainsToSend,file);
                  if ( dataRead != 2*chainsToSend )
                  {
                        printf("Unable to read %d chains from file on
Process %d\n",2*chainsToSend,my_rank);
                        fflush(stdout);
                        return -1;
                  }

                  MPI_Send(chains, chainsToSend*2, MPI_DOUBLE,
DEST_ROOT, TAG_FINISHED_WORK_UNIT, MPI_COMM_WORLD);
                  chainsToSendCounter += chainsToSend;
            }

            // Close the output file
            fclose(file);
```

```
	}

	return 0;
}

int main(int argc, char* argv[])
{
	// call the main part of the application
	main2(argc, argv);

	////////////////////////////////
	// Finalize the MPI Interface
	// This is done here to prevent issues with the "return 0" calls
that are frequently used in the main2 function
	// which are in place for error handling
	////////////////////////////////
	MPI_Finalize();
}
```