EFFICIENT MULTITHREADED HASHING FOR MESSAGE AUTHENTICATION CODE

Arpit M Christi
(B.E., Dharmsinh Desai Institute of Technology, India, 2003)

PROJECT
Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE
In
COMPUTER SCIENCE
at
CALIFORNIA STATE UNIVERSITY, SACRAMENTO

FALL
2006
EFFICIENT MULTITHREADED HASHING FOR MESSAGE AUTHENTICATION

CODE

A Project

By

Arpit M Christi

Approved by:

__________________________, Committee Chair
Dr. Ted Krovetz
Date: ______________________

__________________________, Committee Member
Dr. Scott Gordon
Date: ______________________
Student: Arpit M Christi

I certify that these students have met the requirements for format contained in the University format Manual, and that this Project is suitable for shelving in the Library and credit is to be awarded for the project.

_______________________________
Dr. Cui Zhang, Graduate Coordinator

_______________________________
Department of Computer Science
Abstract

of

EFFICIENT MULTITHREADED HASHING FOR MESSAGE AUTHENTICATION

CODE

by

Arpit M Christi

The processor technology is moving from single core to dual core and quad core processor. Most of the MAC algorithms are designed and implemented to take advantage of processor architecture of single core processor, register length and instruction set. The major component of MAC algorithm is underlying HASH algorithm. The main aim of this project is to come up with a parallel solution of HASH algorithm such that the work is divided into two equal disjoint parts. The solution is then implemented using threads such that two equal parts are handled by different threads. Because load is balanced equally, when these threads will run simultaneously on different cores of dual core processor, it will give a theoretical twice as good a performance as single core processor.

A parallel solution and implementation is given for quad core processor also by dividing the algorithm into four equal parts.

___________________, Committee Chair       Date __________________
Dr. Ted Krovetz
ACKNOWLEDGEMENT

Many people have helped me directly or indirectly to achieve my goals. I would like to thank all of them for their support and advice. I would also like to thank my family for their moral support, blessings and for having confidence in my abilities.

I would like to express my sincere gratitude to Dr. Ted Krovetz, my project advisor, for her valuable expertise, guidance and encouragement throughout this project. I am also thankful to Dr. Scott Gordon for spending his valuable time on reviewing my work.

I am also thankful to Dr. Denise Sour for proof reading my work.
# Table of Content

Acknowledgment ........................................................................................................... v

List of Tables ................................................................................................................... viii

List of Figures .................................................................................................................. ix

CHAPTER 1 INTRODUCTION ......................................................................................... 1

CHAPTER 2 BACKGROUND OF THE PROJECT......................................................... 3
  2.1 Security in Data Transmission and Cryptography ............................................. 3
  2.2 Message Authentication and Message Authentication Codes (MAC) ........ 3
  2.3 Use of Wegman-Carter Types of MAC to Authenticate the Message .......... 4
  2.4 Universal Message Authentication Code (UMAC) and Its Properties ....... 8
  2.5 Phases of UMAC – NH, PolyR and InnerProduct Hash ......................... 8

CHAPTER 3 EVOLUTION OF THE PROJECT ............................................................ 12

CHAPTER 4 THREADS: POSIX THREAD AND WINTHREAD ............................. 14
  4.1 Thread Overview ............................................................................................... 14
  4.2 Linux POSIX Thread ....................................................................................... 15
  4.3 WinThread ......................................................................................................... 17

CHAPTER 5 PARALLEL IMPLEMENTATION RELATED ISSUES ................... 48
  5.1 NH Hash and Parallel Implementation ............................................................ 48
  5.2 Polynomial Hash and Parallel Implementation .............................................. 49
  5.3 Other Implementation Issues with Polynomial Hash .................................... 49

CHAPTER 6 RESULTS AND COMPARISIONS ....................................................... 48

vi
CHAPTER 7 CONCLUSION.................................................................48
APPENDIX A SOURCE CODE FOR SINGLE THREAD IMPLEMENTATION.......49
APPENDIX B SOURCE CODE FOR MULTI THREAD IMPLEMENTATION FOR DUAL CORE PROCESSOR.................................................................49
APPENDIX C SOURCE CODE FOR MULTI THREAD IMPLEMENTATION FOR QUAD CORE PROCESSOR.................................................................49
BIBLIOGRAPHY.......................................................................................97
List of Tables

Table 6.1 Performance Gain for Various Message Size Using Dual Core Processor......15

Table 6.2 Performance Gain for Various Message Size Using Quad Core Processor with
   Different method of Measuring gain.............................................................15
List of Figures

Figure 6.1 Performance Gain for Various Message Size Using Dual Core Processor…..10

Figure 6.2 Performance Gain for Various Message Size Using Dual Core Processor with Different Method of Measuring Gain……………………………………………..20
CHAPTER 1
INTRODUCTION

Over the years, as design and manufacturing techniques are improving, a restless change is observed in processor technology. The areas in which improvements are observed significantly in recent times are register length, increase in number of cores, cache technology etc. In recent years a shift is experienced from 32 bit to 64 bit processor, from single core to dual core or quad core processor. In future 64 bit architecture and dual core and quad core processors are going to dominate the market.

Many performance critical algorithms are designed to take advantage of particular processor architecture to gain high performance. As the architecture changes the design of algorithms also needs to be changed in order to get maximum out of the new features of architecture. This is particularly true for computationally intensive cryptographic MAC (Message Authentication Code) algorithms. The fastest known software optimized MAC for 32 bit architecture is UMAC (Universal Message Authentication Code) [2]. In his paper “Message Authentication on 64 bit architecture”, Dr. Krovetz discussed how to change the design of UMAC algorithm in order to gain high performance on 64 bit architecture [1]. The remaining major change in processor technology is dual core processor. This project is an attempt to explore the change in design of UMAC algorithm in order to optimize its performance on dual core processor. The major component of any MAC algorithm is its underlying hash algorithm. Changes are mostly made to design and implementation of hash algorithm in order to improve MAC. I designed and implemented
a parallel solution for hash algorithm. Preliminary analysis of the design and implementations are used to compare the performance enhancement while moving from single to dual core processor with a change in design of the algorithm. The project then moves to optimized implementation of algorithm for dual core processor. All the results are generated for Intel-32 bit dual core processor.
2.1 Security in Data Transmission and Cryptography

Security in Data transmission is becoming an important issue as intruders are becoming smarter and smarter. Cryptography deals with security problem. Rivest defines Cryptography as “Communication in the presence of adversaries”. The important assumption here is that the adversary is resourceful enough that she has uninterrupted access to communication channel so that she can gather enough information about the message being transmitted. Two well known cryptography problems are “encryption” and “message authentication”.

Let’s look at the first problem: Encryption. Alice (A) wants to transfer some message (M) to Bob (B) in the presence of an adversary Eva (E) which has complete access of communication channel so that she can observe all the data being transferred from A to B. We want to prevent E from gathering any useful information from the data being transferred though she can observe the data. This is typically achieved by encrypting the message on one side and then decrypting it on the other side using block ciphers.

2.2 Message Authentication and Message Authentication Code (MAC)

Encryption doesn’t solve the data transmission problem completely. “Message Authentication” is another issue. Alice (A) again wants to transfer Message (M) to Bob (B). This time Eva (E) is not passively eavesdropping. She is trying to impersonate B.
This means she is tampering with the message and changing the content of the message so that B will receive tampered message. To prevent this, A will apply Message Authentication Code (MAC) generation algorithm to the message and generate a Tag. A will send the Tag along with the encrypted message. Upon receiving the message, B will first decrypt the message and then apply the MAC once again on the message and generate the tag. Both the tags are then compared for equality. If they are equal then message is not tampered with along the way.

Message authentication scheme using MAC consists of MAC generation and MAC verification algorithm. Sender uses MAC generation algorithm that takes as input a message M, key K and generates a short tag. The tag is transmitted along with the message. The key K is random and shared between sender and receiver. Upon receiving the message, receiver uses MAC verification algorithm that takes as input message M, key K, tag and finds out if the message is tampered with along the way or not. MAC verification algorithm does so by regenerating tag again using M, K and MAC generation algorithm and comparing it to the transmitted tag.

2.3 Use of Wegman-Carter Types of MAC to Authenticate Message

In Wegman-Carter type of MAC scheme sender and receiver share two secret functions.

\[ f: Z \rightarrow Z_n \] , a randomly chosen random function

\[ h: \{0,1\}^* \rightarrow Z_n \] a randomly chosen hash function from an \(\varepsilon\)-\(A\Delta U\) hash family

(Defined later)

\[ Z = \{ \ldots, -3, -2, -1, 0, 1, 2, 3 \ldots \} \]
\( Z_n = \{ 0, 1, 2, 3, 4, \ldots n-1 \} \)

To send message \( m \), the \( n \)th in a sequence, sender computes
\[
\tau = f(n) + h(m)
\]
and sends \((m, n, \tau)\). The receiver gets \((m', n', \tau')\) and checks to see if
\[
\tau' = f(n') + h(m')
\]
If so then the receiver thinks that the message is authentic.

Model of security in case of Wegman-Carter MAC is defined as follows. The adversary has carried out enough experiments and found out a series of \((m, n, \tau)\) pairs. So, adversary knows
\[
(m_1, n_1, \tau_1) \text{ where } f(n_1) + h(m_1) = \tau_1
\]
\[
(m_2, n_2, \tau_2) \text{ where } f(n_2) + h(m_2) = \tau_2
\]
\[
(m_3, n_3, \tau_3) \text{ where } f(n_3) + h(m_3) = \tau_3
\]
\[\ldots\]
\[
(m_q, n_q, \tau_q) \text{ where } f(n_q) + h(m_q) = \tau_q
\]
Now, adversary guesses tag \( \tau \) for a new \((m,n)\) pair and if his guess is correct than forgery has occurred. To find the probability of forgery, we need to find probability of adversary guessing correct tag \( \tau \) for given messagenonce pair and that is:
\[
Pr[ h(m) + f(n) = \tau ].
\]
To calculate this probability we have two options.

1. The nonce is not previously generated one and in that case
   \( n \) doesn’t belong to \( \{n_1, n_2, n_3, \ldots, n_q\} \)

   Then \( f(n) \) is random and so \( h(m) + f(n) \) is also random.
So, \( Pr[ h(m) + f(n) = \tau ] = 1/n. \)

2. The nonce is previously generated.

\( n \in \{ n_1, n_2, n_3, \ldots n_q \} \)

if \( n = n_i \)

then \( f(n) = f(n_i) \) and \( h(m_i) + f(n_i) = \tau_i \), meaning

\( f(n) = f(n_i) = \tau_i - h(m_i) \)

So,

\[ Pr[ h(m) + f(n) = \tau ] \]

\[ = Pr[ h(m) + (\tau_i - h(m_i)) = \tau ] \]

\[ = Pr[ h(m) - h(m_i) = \tau - \tau_i ] \]

\[ = Pr[ h(m) - h(m_i) = d ] , \text{ where } \tau - \tau_i = d = \text{constant} \]

\( <= \epsilon \) (for \( \epsilon \)-AΔU hash function)

A hash-function family \( H \) is \( \epsilon \)-almost delta universal (\( \epsilon \)-AΔU) if the probability is no more than \( \epsilon \) that \( h(m) - h(m') = d \) for any two distinct inputs \( m, m' \) and any chosen constant \( d \) when hashed by a randomly selected member \( h \) of \( H \) [5]. Combining both the cases adversary’s probability of success is \( \max(1/n, \epsilon) \).

UMAC is Wegman-Carter type of MAC. The hash function (h) used in UMAC is uhash and it is \( \epsilon \)-AΔU hash function. First phase is NH hash which compresses the message apart from universally hashing the message. Next phase is PolyR which produces constant length tag which is independent of input message size.

### 2.4 Universal Message Authentication Code (UMAC) and Its Properties
Universal Message Authentication Code (UMAC) is the fastest known MAC and uses key length of around 1KB to generate the MAC. On a 350 MHz Pentium-2 machine UMAC gives $2^{-60}$ forgery probability with performance of .98 cycles/ byte [2]. UMAC is a Wegman-Carter type of MAC, so it also has a secure shared hash function and shared key. It first hashes the message using $\varepsilon$-AU universal hash function NH hash. The next hash function is polynomial hash function that is $\varepsilon$-AΔU hash function.

2.5 Phases of UMAC – NH, PolyR and InnerProduct Hash

Let’s look at all three phases of UMAC in detail. NH phase acts as an accelerator by compressing the message into smaller strings at a very high speed. NH phase divides the input into chunks of 1024 bytes and than hashes each into 8 bytes giving a compression ratio of 128:1. The hashed blocks are then concatenated and a single string is generated to supply to the next phase. The message should be in multiples of 2w bits where w is register size of the processor. If not then appropriate padding is applied. Message M is divided into chunk of 1024 bytes and Key K is also 1024 bytes. Each chunk is further divided into n chunks $m_1$, $m_2$ …. $m_n$ And key is divided into $k_1$, $k_2$……$k_n$ (each of length w bits) and hash is calculated as

$$\text{NH}_k(M) = \sum_{i=1}^{k/2} (m_{2i-1} \times m_{2i} + k_{2i-1} \times k_{2i}) \mod 2^{2w}$$

Each of such NH chunks generated for every 1024 bytes of data chunks are than
Concatenated and a hashed string is generated for the next phase.

NH is a universal hash family and choosing a random function from hash family is done by choosing a random key of K=nw bits. NH is extremely fast as it uses processor word length efficiently (w is 32 for 32 bit architecture and 64 for 64 bit architecture). NH is known to be $\varepsilon$-AU ($\varepsilon = 2^{-w}$) hash function and hence secure too.

The next phase is polynomial hash function which produces tag of constant length. NH phase output length is dependent on input length. Polynomial hash produces fixed output length hash. Polynomial hash uses a large prime number p and divides the input message into $M_1, M_2, M_3, \ldots, M_q$ such a way that when the blocks are interpreted as unsigned integers $m_1, m_2, m_3, \ldots, m_q$ each less than p. Key k is chosen in between 0 to p and hash output is defined as

$$H_k(M) = m_1k^1 + m_2k^2 + m_3k^3 + \ldots + m_qk^q \mod p$$

Polynomial hash defined above is $\varepsilon$-AΔU hash family for $\varepsilon = l/p$ [1], so it is secure. The only concern here is how to make polynomial hash efficient.

Some of the optimizations done by UMAC designer are discussed. I have used similar kind of optimizations in my design and implementations and those will be discussed in the future. Polynomial evaluation is calculated using Horner’s rule making polynomial hash as

$$H_k(m) = (\ldots ((m_1k + m_2)k) + m_3)k\ldots)k + m_q)k \mod p$$

Now k need not be raised by some power that is a time consuming operation. Time consuming mod operation can also be reduced further by using smart choice of k and p and making efficient use of underlying processor architecture. By choosing p of the form
of $2^a - b$ for some small $b$, mod operation can be reduced to a shift and addition operation. Consider 3 bit number system and $p = 2^3 - 1 = 7$ ($a = 3$ and $b = 1$). Consider two numbers in the range of 0 to 7 say 5 and 6 with their binary representation 101 and 110. The addition of these two numbers 11 has binary representation 1001 and 11 mod 7 is 4 binary 100. In 0 to 7 number system additional 4th bit in 11 is carry. If this carry is added to lower 3 bits it will give the number 4 in binary which is 11 mod 7. Adding carry to leftmost 3 bits means shifting carry bit to 3 bits and than adding the carry to leftmost 3 bits. Similarly all the bits generated after leftmost 3 bits can be left shifted 3 bits and then added to leftmost 3 bits to generate required result with proper handling of carry situation. So, the mod operation is reduced to addition and shift operation. The same concept can be expanded for any values of $a$ and $b$. Number $p$ is usually selected as $2^{61} - 1$ for UMAC and then similar kind of calculations are applied to replace mod operation by addition and shift operation. UMAC also multiplies two 64 bit numbers using 32 bit architecture. Consider two 64 bit numbers $w = ab$ and $x = cd$, $a$ and $c$ being upper 32 bits and $b$ and $d$ being lower 32 bits. $w = a2^{32} + b$ and $x = c2^{32} + d$ and $wx = ac2^{64} + (ad + bc)2^{32} + bd$. The lower 32 bits of $(ad + bc)$ are added to upper 32 bits of $bd$ and upper 32 bits are added to lower 32 bits of $ac$. Possibilities of carry is eliminated using turning of some bits of the message which in turn will increase the forgery probability but it is allowed for the sake of efficiency. The same multiplication principle is also applied for NH-hash multiplication but carry is not taken care in that case considering the fact that if the carry is not taken care of by both sender and receiver in the same manner it will produce the same result. One can look to “Fast Universal Hashing with small keys and no

The last phase is InnerProduct hash. Let M be the message and p be the large prime. M is broken into chunks M₁, M₂, ..., Mₚ such a way that M₁, M₂, M₃, ... are interpreted as unsigned integers m₁, m₂, m₃, ..., mₚ each less than p. Chose a key K = (k₁, k₂, k₃, ..., kₚ) then inner product hash is defined as

\[ H_K(M) = m₁k₁ + m₂k₂ + m₃k₃ + \ldots + mₚkₚ \mod p \]

Inner product hash is \( \epsilon\)-AΔU hash family with \( \epsilon = 1/p \) and hence secure. InnerProduct hash requires as much key as the length of message and hence not useful for long messages.
CHAPTER 3
EVOLUTION OF THE PROJECT

UMAC was designed keeping 32 bit architecture in mind. It fully takes advantage of 32 bit processor architecture and its instruction set. But processor technology observed changes recently with 64 bit processor and dual core processor. 32 bit UMAC design is not sufficient to take advantage of 64 bit architecture and its instruction set. One such 64 bit design VMAC was developed and presented by Dr. Krovetz in his paper “Message Authentication for 64 bit architecture” [2] and detail analysis is given about all the issues and gain obtained by the new design. On the athlon 64 processor, VMAC gives performance of 0.5 cycles/byte, twice faster than the UMAC implementation for 32 bit processor [2].

Is the same true about dual core processor? Is there a possible design for dual core processor which can give a performance enhancement of the sort of enhancement achieved by VMAC over UMAC? The quest to find answer to these questions is the main reason for the evolution of this project. The project started with study of existing UMAC and VMAC algorithm, their design and implementation. Then I tried to find a possible parallel design such that the algorithm is divided into two equal parts. Our assumption is that if parallel design is implemented carefully using threads and when it is supplied with large amount of data the operating system will schedule both the threads to run on different cores of dual core processor. Assuming synchronization overhead to be small I
am expecting almost twice better performance than the sequential design of UMAC. Project includes providing preliminary implementation of uhash design for both sequential and parallel design and carrying out timing analysis on them to find out gain obtained by parallel design. Also while we were going through our project quad core processor also came into market so the case of quad core processor was also included though it was not originally intended. Though the quality of my implementations was nowhere close to the quality of implementation provided by Dr. Krovetz for UMAC and VMAC, it serves as a starting point for the parallel implementation of MAC algorithm and hopefully in future other students will take my work and explore the possibilities in more depth.
Before I go into the detail of my design and implementation it would be a good idea to iterate through basics of threads as I have used threads to break up the algorithm into two equal chunks and made these threads run parallel. So, how to create thread, how to pass data to thread, how to synchronize threads, memory and processor overhead associated with thread creation and execution etc are some of the important issues that I would like to discuss.

4.1 Thread Overview

Thread is independent set of instructions of a program that can be scheduled to run independently from other parts of the program or even of the main program. Threads use and exist within process resources, yet are able to be scheduled by the operating system and run as independent entity largely because they duplicate only the bare essential resources that enable them to exist as executable code. Thread has independent flow of control than the main process or other processes as long as OS supports it or the parent process exists. Thread may share resources with other threads. Efficient sharing mechanism is required in that case to avoid deadlock and/or long waiting. One of the most important characteristic of thread is “Thread is lightweight”. Most of overhead associated with thread creation and existence is taken care by its parent processes.
4.2 Linux Posix Thread [6]

The following discussion is taken from “Posix Thread Programming” [6]. For more detail on Linux posix thread you can refer corresponding website.

Creating and terminating threads:

**Pthread_create(thread, attr, start_routine, arg)** – Thread creation and execution

**Pthread_exit(status)** – exiting thread

**Pthread_attr_init(attr)** – setting and initializing thread attribute

**Pthread_attr_destroy(attr)** – destroying attribute

**pthread_create arguments:**

- **thread**: An opaque, unique identifier for the new thread returned by the subroutine.
- **attr**: An opaque attribute object that may be used to set thread attributes. You can specify a thread attributes object, or NULL for the default values.
- **start_routine**: the C routine that the thread will execute once it is created.

Passing arguments to threads:

- The **pthread_create()** routine permits the programmer to pass one argument to the thread start routine. For cases where multiple arguments must be passed, this limitation is easily overcome by creating a structure which contains all of the arguments, and then passing a pointer to that structure in the **pthread_create()** routine.
• All arguments must be passed by reference and cast to (void *).

**Joining threads:** Making main thread to wait for the completion of child thread

**Pthread_join(threaded, status)**

**Pthread_attr_setdetachstate(attr, detachstate)**

**Pthread_attr_getdetachstate(attr, detachstate)**

• The pthread_join() subroutine blocks the calling thread until the specified thread terminates.

• The programmer is able to obtain the target thread's termination return status if it was specified in the target thread's call to pthread_exit().

• To explicitly create a thread as joinable or detached, the attr argument in the pthread_create() routine is used. The typical 4 step process is:
  o Declare a pthread attribute variable of the pthread_attr_t data type
  o Initialize the attribute variable with pthread_attr_init()
  o Set the attribute detached status with pthread_attr_setdetachstate()
  o When done, free library resources used by the attribute with pthread_attr_destroy()

4.3 WinThread [7]
There exists multiple ways to create and use windows thread. I am just discussing the model I use for my implementation. Most of the following discussion is taken from “Overview of Windows Thread Function” [7]. For more detail you can refer the website.

**Creating thread:**

```c
HANDLE CreateThread(LPSECURITY_ATTRIBUTES secattr, SIZE_T stacksize,
                     LPTHREAD_START_ROUTINE threadfunc, LPVOID param,
                     DWORD flags, LPWORD threadid)
```

- **threadfunc**: threadfunc defines the function that executes when thread is created. Thread execution begins with call to this thread function and ends when this function exits. Prototype of threadfunc is

  ```c
  DWORD WINAPI threadfunc(LPVOID param)
  ```

- **param**: It is the parameter passed to the thread function. It should be cast to void long pointer.

- **Threadid**: Unique threaded by which thread is identified

- **Flags**: Determines the status of execution of thread. If it’s 0 thread execution has begun.

- **Return value**: The function returns a handle to the thread. The handle is null if there is an error in thread creation. This thread can be used to control the thread in the future.

**Joining the threads**
DWORD WINAPI WaitformultipleObjects(DWORD count, const HANDLE* lphandle, Bool waitall, DWORD milliseconds)

- **count**: Number of objects or threads handled by lphandle parameter for joining.
- **lphandle**: An array of threads for which main thread or calling thread is going to wait.

As one can see there is large difference in how threads are used in Linux and Windows environment. One should be extremely careful if one wants to write a portable code which works on both windows and Linux machines. Also Linux Posix threads use pthread.h and windows thread use windows.h. I have written separate code for both these cases.
CHAPTER 5
PARALLEL IMPLEMENTATION RELATED ISSUES

This chapter discusses the parallel design that I came up with. It also discusses the implementation issues that I face and the solution to those issues. I divided all the phases of algorithm discussed before into two parts in such a way that work is divided equally into two parts.

The basic algorithm is very simple. The message is divided into chunks of 1 Kb and nh step is applied into each 1Kb chunk. Each of these 1Kb chunks is further divided into chunks of 64 bits to apply each of the NH step to it. So for each 1Kb of data we are getting 64 bits of hashed string. So apart from universally hashing the message, we are getting compression factor of 8. We should have collected all these strings, concatenated them, produce a complete hashed message and supplied it to Polynomial hash function. Instead of wasting our time into doing all these we directly supply each of 64 bits hashed string to the polynomial step to hash it further. The smart design and partition of the algorithm saves a lot of computational resources. The 32 bit output generated by the polynomial hash is the final output.

The normal simple algorithm implementation is single process without any threads. For parallel implementation two threads are created and alternative 1Kb chunks are handled by different threads. So, odd and even chunks of message are handled by different threads. Each thread follows the same step as the simple nonthreaded algorithm goes through. First comes the NH phase compressing and universally hashing the message.
Then goes the polynomial hash phase, hashing the message to a fixed length hash string. A lot of tricks and design related issues come into picture to design and implement polynomial hash.

5.1 NH Hash and Parallel Implementation

The parallel design of NH phase is very simple and requires no extra effort as it is naturally parallel algorithm. Its design and implementation stays exactly the same, just apply NH1 and NH2 (NH in two threads) on odd and even chunks of data. NH is defined as follows.

$$\text{NH} (K, M) = \sum_{i=1}^{\lfloor \frac{l}{2} \rfloor} ((M_{2i-1} + K_{2i-1} \mod 2^{32}) \times (M_{2i} + K_{2i} \mod 2^{32})) \mod 2^{64}$$

NH1 and NH2 do the same thing as NH just on odd and even chunks of data. Here $l$ is equal to number of bytes of message $M$ divided by length of each chunk. N is number of bytes of message and $w$ is processor word length. $K$ is the key and $M$ is the message. For each step when $M_i$ and $K_i$ are added result can be 33 bits considering the generation of carry. But just the 32 bit value that’s available in register is kept and carry is not taken care of considering the fact that if sender and receiver follow the same process, throwing carry won’t hurt us. Also 32 bit multiplication will produce 64 bit result which will be kept in 32 bit high register and 32 bit low register and can be utilized easily for further manipulation. So all the three design NH, NH1 and NH2 are tailored to optimize the use of 32 bit architecture. Considering the fact that NH1 and NH2 are equally load balanced,
when run simultaneously on different core of dual core processor they should give twice as good a performance as the NH.

5.2 Polynomial Hash and Parallel Implementation

There is a lot more going on in Polynomial hash than NH hash. Polynomial hash evaluates polynomial in large modulo domain. Evaluation of polynomial is done using Horner’s rule. Polynomial hash function and simple algorithm to evaluate it using Horner’s rule are as follows. Message $M$ is divided into $M_0$, $M_1$, $M_2$, $M_{n-1}$ such that each one is treated as unsigned integer $m_0$, $m_1$, $m_2$, $m_{n-1}$

$$H_k(M) = m_0k^{n-1} + m_1k^{n-2} + m_2k^{n-3} + \ldots + m_{n-2}k + m_{n-1} \mod p$$

The same equation using Horner’s rule can be defined as

$$H_k(M) = \left(\ldots((m_0k + m_1)k + m_2)k\ldots)k + m_{n-1}\right) \mod p$$

The simple algorithm for polynomial evaluation using Horner’s rule is as follows.

$$H_k(M) = 0$$

For $i = 0$ to $n-1$ do

$$H_k(M) = H_k(M) \times k + m_i \mod p$$

Here within the pseudo code of algorithm $H_k(M)$ is used as variable and not as function notation. As one can see this is a sequential algorithm as previous value of
$H_k(M)$ is used to calculate the new value of $H_k(M)$ and hence can not be made parallel easily. I divided the algorithm into two equal parts as follows.

$$H_{1k}(M) = \sum_{i=0}^{n/4-1} m_0 k^{n-1} + m_1 k^{n-2} + m_2 k^{n-5} + m_3 k^{n-6} \ldots + m_{4i+1} k^{n-(4i+1)-1} + \ldots + m_{4i} k^{n-(4i)-1} + \ldots \mod p$$

$$H_{1k}(M) = \sum_{i=0}^{n/4-1} m_{4i}+1 k^{n-(4i+1)-1} \ldots + m_{4i+1} k^{n-(4i)-1} \mod p$$

AND

$$H_{2k}(M) = \sum_{i=0}^{n/4} m_2 k^{n-3} + m_3 k^{n-4} + m_4 k^{n-7} + m_5 k^{n-6} \ldots + m_{4i+2} k^{n-(4i+2)-1} \ldots + m_{4i+3} k^{n-(4i+3)-1} + \ldots \mod p$$

$$H_{2k}(M) = \sum_{i=0}^{n/4} m_{4i+2}+1 k^{n-(4i+2)-1} \ldots + m_{4i+3} k^{n-(4i+3)-1} \mod p$$

Algorithm to compute $H_{1k}(M)$ and $H_{2k}(M)$ is as follows.

$H_{1k}(M) = 0$

For $i = 0$ to $n/4-1$

$H_{1k}(M) = H_{1k}(M) \times k^4 + m_{4i-3} \times k + m_{4i-4} \mod p$

Here within the pseudo code of algorithm $H_{1k}(M)$ is used as variable and not as function notation. Just like Horner’s rule is used for evaluating $H_{3k}(M)$, $H_{1k}(M)$ and $H_{2k}(M)$ are evaluated using Horner’s rule. It is not obvious from the equations that $H_{1k}(M)$ is out of synchronization by $k^2$ but if one expands the series one can see that. This is taken care by the algorithm as follows.
Final Hash output = $H_{1k}(M) \times k^2 + H_{2k}(M)$

The algorithm is now divided into exactly two equal parts. Both the parts are part of different threads. When heavily loaded with the data, both threads will be scheduled to run on different cores of dual core processor and finish their work in half the time as they are doing exactly half of the work. So, theoretically parallel polynomial hash should give twice as good a performance as normal polynomial hash.

### 5.3 Other Implementation Issues with Polynomial Hash

Selection of large modulo prime $p$: $p$ is selected as to be $2^{30} - 1$. This way $p$ is close to $2^{32} - 1$, the largest value allowed by 32 bit register. Keeping its value large will make $\epsilon = 1/p$ small and hence will make polynomial hash more secure. In parallel implementation of polynomial hash, two 30 bit numbers are multiplied and 60 bit result is generated. If $p=2^{30} - 1$ than upper 30 bits can be directly added to lower 30 bits in order to generate the modulus. If adding lower 30 bits generate a carry, then carry is again added to the 30 bit sum to get final value. This way the modulus operation is completely eliminated by addition and shift operations. This sounds a bit confusing at first, so let’s understand it with a simple example. Consider a 3 bit number system consisting of values 0, 1….., 7 and modulus domain 7.

**Multiplying two numbers 5 and 6 will yield 30 mod 7 = 2 mod 7 = 010**

**Multiplying 5(101) and 6(110) = 30 = 011110.**

```
  011
+ 110
1001
```
Still carry is generated so add carry to lower 3 bits.

\[
\begin{align*}
001 \\
+ & 1 \\
010 & = 2 \text{ mod } 7
\end{align*}
\]

Keeping \(m_1, m_2, m_3, \ldots\) Within limit \(2^{30} - 1\): When Message \(M\) is divided into chunks \(M_1, M_2, M_3, \ldots\) and treated as unsigned integer \(m_1, m_2, m_3, \ldots\) their values can go beyond the limit as each one is a 32 bit number. In order to keep them within the limit, the upper two bits of each of the chunks are turned off (set to zero). By doing this we are providing adversary certain information which in turn increases the chances of forgery, but to gain advantage in calculation we are allowing it. Also turning off upper two bits has other advantage. If one looks at polynomial hash step carefully it is

\[
H_{1k}(M) = H_{1k}(M) + m_{4i+1}k^{n-(4i+1) - 1} + \ldots + m_{4i}k^{n-(4i) - 1} \mod p
\]

It is implemented in algorithm as

\[
H_{1k}(M) = H_{1k}(M) \times k^4 + m_{4i-3} \times k + m_{4i-4} \mod p
\]

As one can see here \(H_{1k}(M)\) and \(k^4, m_{4i-3}\) and \(k\) are all 32 bit numbers. The multiplication of these two numbers will produce 64 bit result and when this 64 bit results are added it can throw carry. We want to prevent this carry generation. As upper two bits of \(m_1, m_2, m_3, \ldots\) are turned off it makes them 30 bit numbers. The multiplication now generates 62 bit number and addition is 63 bit number and hence carry is never generated.
CHAPTER 6
RESULTS AND COMPARISION

As the output of the project, performance of multithreaded implementation is compared with the sequential implementation. Output is provided for windows environments only. Windows results are generated using Intel 1.60 GHz machine using 1GB of RAM. While executing all the implementations on windows machine, I make sure that CPU and memory are least utilized by making sure that no unnecessary processes are running. So theoretically, CPU and memory are completely available to execute the code. The size of the data is varied from 1MB to 1GB. To have better average results, the UHASH code is run several times and the average performance is used for comparison. Table 6.1 and Figure 6.1 compare implementation for single thread and multi thread. Here performance gain is calculated as

Performance gain = Multi Core implementation ticks / Single Core implementation ticks

Both single core and dual core implementation are fairly optimized. The use of inline assembly also aids to the optimization. A parallel solution and implementation are also provided for quad core processor by dividing the algorithm into four equal parts and implementing it using four threads.
Table 6.1 Performance Gain for Various message Size Using Dual Core Processor

<table>
<thead>
<tr>
<th>Message Size</th>
<th>Performance Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;1MB</td>
<td>1.06</td>
</tr>
<tr>
<td>4MB</td>
<td>1.24</td>
</tr>
<tr>
<td>16MB</td>
<td>1.42</td>
</tr>
<tr>
<td>64MB</td>
<td>1.43</td>
</tr>
<tr>
<td>250MB</td>
<td>1.34</td>
</tr>
<tr>
<td>512MB</td>
<td>1.37</td>
</tr>
<tr>
<td>1GB</td>
<td>1.24</td>
</tr>
<tr>
<td>&gt;2GB</td>
<td>&lt;1.24</td>
</tr>
</tbody>
</table>

Figure 6.1 Performance Gain for Various Message Size Using Dual Core Processor
The Problem: As one can see the gain obtained is very low than the gain excepted at the beginning of the paper. I would like to discuss possible reasons for the low gain. The main reasons are Thread creation, wait and destruction time, scheduling and memory overhead. Both the threads are launched at almost the same time, but the time they spend in execution can vary depending upon amount of resources available to them. Also depending upon this time the main thread has to wait for the more time consuming thread to finish. This adds an overhead too. Also at the end both threads needs to be destroyed. When message size is small, amount of time spent in thread creation, wait and destruction is significant and hence the gain is low. As the message size increases we can see significant gain. When message size is greater than 1GB, the RAM is utilized completely and a lot of thrashing needs to be carried out in order to get the data to and from virtual memory and gain decreases. To overcome these problems I came up with a different method to measure the gain. Instead of creating thread for each hash, I am doing hash several times in a single thread and than measure the gain. Also I kept the buffer size small so that RAM is not fully utilized and memory thrashing is reduced. For example, instead of hashing 1GB of message, I divided the message into 10 100MB message and did hash 10 times in single thread. This way program is spending more time in actual work than in thread creation, wait and destruction. Table 6.2 and Figure 6.2 represent the gain for various message sizes in this case. Results obtained by this method is comparatively high than the previous case.
Table 6.2 Performance Gain for Various Message Size Using Dual Core Processor with Different Method of Measuring Gain

<table>
<thead>
<tr>
<th>Message Size</th>
<th>Performance Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>1MB</td>
<td>1.3</td>
</tr>
<tr>
<td>4MB</td>
<td>1.3</td>
</tr>
<tr>
<td>16MB</td>
<td>1.5</td>
</tr>
<tr>
<td>64MB</td>
<td>1.5</td>
</tr>
<tr>
<td>256MB</td>
<td>1.6</td>
</tr>
<tr>
<td>512MB</td>
<td>1.6</td>
</tr>
<tr>
<td>1GB</td>
<td>1.6</td>
</tr>
</tbody>
</table>

Figure 6.2 Performance Gain for Various Message Size Using Dual Core Processor with Different Method of Measuring Gain.
CHAPTER 7
CONCLUSION AND FUTURE WORK

Project derives a parallel solution for UHASH and implements it for dual core processor to prove the gain obtained by the parallel solution. The paper discusses issues that I encounter while working with parallelism of algorithms and parallel implementation. It also discusses how linux-posix threads and Winthreads can be used for parallel implementation. Theoretically gain should be doubled and in the beginning we expected a gain of the factor of 1.6 to 1.8. Gain that I got was quite less compared to what we expected. The possible reasons for low gain are also discussed. The implementation is still very primitive and it mostly gives a proof of design but still more optimized implementation is possible with certain change in design. All the results are generated for Intel-32 architecture with dual core processor assuming that the same gain will follow for other architectures too. A parallel solution and implementation is given for Quad core processor too. VMAC and VHASH are corresponding 64 bit algorithm for UMAC and UHASH designed and implemented for AMD-64 bit architecture. Parallel solution of VMAC and VHASH for AMD-64 dual core processor is one immediate future work. I used 32 bit multiplication using 32 bit architecture which is fairly simple and straightforward. UMAC uses 64 bit multiplication using 32 bit architecture that also needs to be accommodated. I would say my work is just a beginning in parallel implementation of MAC algorithm.
// Single Thread Implementation

// Written by Arpit Christi, contact me: arpit_christie@yahoo.com

#include <windows.h>    // for thread
#include <strsafe.h>
#include <stdio.h>
#include <process.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include <sys/types.h>

typedef unsigned __int32 u32;
typedef unsigned __int64 u64;

#define nhbytes 64     // 128 * u32 = 1K
#define MEMORY_ALLOCATION_ERROR 2

unsigned int UINT32_C(char* s)
{
    unsigned int n;
    int r;

    r = sscanf(s, "%x", &n);

    if (r != 1) {
        printf("error when change hex string to integer.");
        exit(0);
    }
    return n;
}

unsigned int large_prime = 1073741823;
u32 p30;
u64 p64 = UINT32_C("0xfffffffffffffff");
u32 mytest[8];

u32 mypolyans, polyans1, polyans2;
u32 testans = 0;

void PMUL(u32 a, u32 b, u64 result)


```c
{
    // multiplies 2 32 bit number and generates 64 bit result
    // can be used for future implementation
}

void ADD(u64 a, u64 b, u64 result)
{
    // add two 64 bit number and throws carry away
    // can be used for future implementation
}

u32 ModByLargePrime(u64 num, u32 lp)
{
    return (num % lp);
}

void poly32(u32 m, u32 k, u32 *ans)
{
    //u64 temp1 = (u64)m;
    u32 temp;
    u64 num = (u64)(*ans) * (u64)k + (u64)m;

    _asm
    {
        mov         esi, dword ptr [num]
        and         esi, dword ptr [p30]   // (u32)num & p30 seperating lower 30 bits
        mov         eax, dword ptr [num]
        shr         eax, 1Eh               // (u32)num >> 30 seperating higher 2 bits
        add         esi, eax
        //mov         eax, dword ptr [ebp-8] // here is where it brings upper 28 bits in eax register
        mov         cl, 20h                // i dont need lower 32 bits anymore. they are already taken care of
        mov         edx, 4h               // multiplying upper 28 bits by 4. better operation is shift left by 2
        mul edx     // edx * eax result stored in EDX:EAX so 30 bit multiplication result is stored in EAX
        add esi, eax
        mov ebx, esi
        shr ebx, 1Eh
        and esi, dword ptr [p30]
        add esi, ebx  // adding upper 2 bits of 32 bit register which represents query
    }
}
```
mov ebx, esi
and esi, dword ptr[p30]
shr ebx, 1Eh
add esi, ebx /* addition if adding upper 2 bits again
generate query, after this it is guaranteed that
mov dword ptr[temp], esi /* query is not going to generated

(*ans) = temp;

void nh(u32 *m, u32 *k, const int n, u64 *ans)
{
    // m is 1KB, key k is 1 KB same key is used everytime
    // nh is invoked
    // here n represents nhbytes
    int i;
    u32 t1,t2;
    static u64 answer = (u64)0;
    for(i=0;i<16; i+=2) /* value 16 is hard core need to be changes
    {
        //answer += ((m[i] + k[i]) * (m[i+1] + k[i+1]));
        (*ans) += ((m[i] + k[i]) * (m[i+1] + k[i+1]));
    }
}

int main(int argc, char **argv)
{
    unsigned int nbytes;
    unsigned char *data;
    unsigned char key[nhbytes];
    int i;
    int count1;
    int count2;
    u64 result,temp;
    u32 *mptr;
    u32 *kptr;
    clock_t ticks,ticks1,ticks2;
    double secs;
    double cpb;
    u32 resultl,resultl;
    u32 polyans;
    DWORD start, finish;
    int test;
u32 polykey = UINT32_C("0x3bcdef21");
polykey = polykey % large_prime;
p30 = UINT32_C("0x3fffffff");
printf("here ");
//nbytes = atoi(argv[1]);
nbytes = 1024 * 1024 * 8   ;
data = (unsigned char *)malloc(nbytes + 16);
//key = (unsigned char *)malloc(nhbytes);
printf(" before if");
if(data == NULL)
{
    printf("allocation error \\
    
");
    exit(MEMORY_ \\
    
);
}
memset(data+nbytes,0,16);   // padding 0s at the end
for(i=0;i<nbytes;i++)
    data[i] = (unsigned char) (i * (i+1));

for(i=0;i<nhbytes;i++)
    key[i] = (unsigned char) (i * (i+1));

i = nbytes / nhbytes;

mptr = (u32 *)data;
kptr = (u32 *)key;
printf("entering into nh\n");
result = (u64)0;
mpolyans = (u32)0;
ticks = clock();
start = GetTickCount();

mptr = (u32 *)data;
kptr = (u32 *)key;
result = (u64)0;
i = nbytes / nhbytes;
polyans = (u32)0;
polyans1 = polyans2 = polyans;
test = 1;
int k = 0;
while(i--) 
{
    result = (u64)0;
    nh(mptr,kptr,nhbytes, &result);
    result = result & p64;

    result1 = (u32)result;
    result = result >> 32;
    resulth = (u32)result;
    resultl = resultl & p30;
    resulth = resulth & p30;

    poly32(resulth,polykey,&polyans);
poly32(resultl,polykey,&polyans);

mptr += (nhbytes / sizeof(u32));
}
temp = result;

int l;
ticks = clock() - ticks;
finish = GetTickCount();
DWORD duration = finish - start;
cpb = (double)ticks / nbytes;
cpb = (double)duration / nbytes;
secs = (double)ticks / CLOCKS_PER_SEC;

printf("poly answer = %lx \n",polyans);
printf("poly answer = %lx \n",testans);

return 0;
APPENDIX B

SOURCE CODE FOR MULTI THREAD IMPLEMENTATION FOR DUAL CORE PROCESSOR

```c
#include <windows.h>
#include <strsafe.h>
#include <stdio.h>
#include <process.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include <sys/types.h>

#define nhbytes 64 // 128 * u32 = 1K
#define MAX_THREADS 2
#define BUF_SIZE 255
#define MEMORY_ALLOCATION_ERROR 2
#define CLOCKS_PER_SEC 1000

typedef unsigned __int32 u32;
typedef unsigned __int64 u64;

u64 ans1 = 0;
u64 ans2 = 0;

u32 polykey2,polykey4;
u32 polyans1,polyans2;
u32 x,x2,x3,x4;
u32 p30;

u32 mytest1[4],mytest2[4];
// 13806395525
int nbytes;
clock_t ticks,ticks1,ticks2,ticks3;
int noofiterations;

unsigned int large_prime = 1073741823;

struct nh_struct{
    u32 *mm;
    u32 *kk;
};
```
typedef struct _MyData {
    int val1;
    int val2;
} MYDATA, *PMYDATA;

unsigned int UINT32_C(char* s)
{
    unsigned int n;
    int r;

    r = sscanf(s, "%x", &n);

    if (r != 1) {
        printf("error when change hex string to integer.");
        exit(0);
    }
    return n;
}

u32 polykey  = UINT32_C("0x3bcdef21");
u64 p64 = UINT32_C("0xffffffffffffffff");

void intialize()
{
    // This function will set up all the keys using aes_encryption
    // algorithm.
    // I need to set up polykey, innerproduct key and nh key here
    // and than use them.
    //
    // polykey  = UINT32_C("0x3bcdef21");
    polykey = polykey % large_prime;
    x = polykey;
    x2 = polykey2 = (polykey * polykey) % large_prime;
    x4 = polykey4 = (polykey2 * polykey2) % large_prime;
    x3 = (polykey2 * polykey) % large_prime;
    p30 = UINT32_C("0x3fffffff");
    polyans1 = (u32)0;
    polyans2 = (u32)0;
}

u32 ModByLargePrime1(u64 num, u32 lp)
{
u32 num1 = (u32)num & p30;
u32 temp = (num1 + (num1 >> 30) + (num >> 32) * 4);
if(num1 < temp)
    temp++;
return temp;

void poly32_1(u32 m0, u32 m1)
{
    u64 num;
    num = (u64)polyans1 * (u64)x4 + (u64)m0 * (u64)x + (u64)m1;
    u32 temp;

    _asm{
        mov    esi, dword ptr [num]
        and    esi, dword ptr [p30]  // (u32)num & p30 seperating lower 30 bits
        mov    eax, dword ptr [num]
        shr    eax, 1Eh              // (u32)num >> 30 seperating higher 2 bits
        add    esi, eax
        //mov    eax, dword ptr [num] I don't need lower 32 bits anymore. they are already taken care of
        mov    eax, dword ptr [ebp-8] // here is where it brings upper 28 bits in eax register
        mov    cl, 20h               // i dont know why this instruction is there
        mov    edx, 4h               // multiplying upper 28 bits by 4. better operation is shift left by 2
        mul    edx                  // edx * eax result stored in EDX:EAX so 30 bit multiplication result is stored in EAX
        add    esi, eax
        mov    ebx, esi
        shr    ebx, 1Eh
        and    esi, dword ptr [p30]
        add    esi, ebx             // adding upper 2 bits of 32 bit register which represents query
        mov    ebx, esi
        and    esi, dword ptr [p30]
        shr    ebx, 1Eh
    }
}


add esi, ebx     // addition if adding upper 2 bits again
generate query, after this it is guaranteed that
mov dword ptr[temp], esi     // query is not going to generated
}
polyans1 = temp;
}

void poly32_2(u32 m2, u32 m3)
{
  u64 num;
  num = (u64)polyans2 * (u64)x4 + (u64)m2 * (u64)x + (u64)m3;
  u32 temp;

  _asm{
    mov esi,dword ptr [num]
    and esi,dword ptr [p30 ]
    mov eax,dword ptr [num]
    shr eax,1Eh
    add esi,eax
    //mov eax,dword ptr [num]  I don't need lower 32 bits anymore. they are already taken care of
    mov eax,dword ptr [ebp-8] // here is where it brings upper 28 bits in eax register
    mov cl,20h     // i dont know why this instruction is there
    mov edx, 4h
    mul edx     // edx * eax result stored in EDX:EAX so 30 bit multiplication result is stored in EAX
    add esi, eax
    mov ebx, esi
    shr ebx, 1Eh
    and esi, dword ptr [p30]
    add esi, ebx
    mov ebx, esi
    and esi, dword ptr[p30]
    shr ebx, 1Eh
    add esi, ebx
    mov dword ptr[temp], esi
  }
  polyans2 = temp;
}

void nhl(u32 *m, u32 *k, const int n, u64 *ans)
{
  // m is 1KB, key k is 1 KB same key is used everytime
// nh is invoked
// here n represents nhbytes

int i;
u32 t1,t2;
static u64 answer1 = (u64)0;

for(i=0;i< 16; i+=2) // value 16 is hard core need to be changes
{
    answer1 += ((m[i] + k[i]) * (m[i+1] + k[i+1]));
    (*ans) += ((m[i] + k[i]) * (m[i+1] + k[i+1]));
}

//printf(" answer1 = %lld  %llx \n", *ans,*ans);

}

void nh2(u32 *m, u32 *k, const int n, u64 *ans)
{
    // m is 1KB, key k is 1 KB same key is used everytime
    // nh is invoked
    // here n represents nhbytes

    int i;
u32 t1,t2;
static u64 answer2 = (u64)0;
u64 temp;

for(i=0;i< 16; i+=2) // value 16 is hard core need to be changes
{
    answer2 += ((m[i] + k[i]) * (m[i+1] + k[i+1]));
    temp = (m[i] + k[i]) * (m[i+1] + k[i+1]);
    (*ans) += ((m[i] + k[i]) * (m[i+1] + k[i+1]));
}

//printf(" answer2 = %lld  %llx \n", *ans,*ans);

}

DWORD WINAPI NhThread1( LPVOID lpParam )
{

    nh_struct* nhstruct1;
u32 *data;
u32 *key;

    struct nh_struct *temp;
    int i;
u32 resultl,resulth;
temp = (struct nh_struct *)lpParam;
data = temp->mm;
key = temp->kk;
i = nbytes/nhbytes;
u64 result=0;
int k=0;
ticks2 = clock();
for(int j=0; j<noofiterations; j++)
{
    while(i>0)
    {
        i = i-2;
        result = (u64)0;
        nh1(data, key, nbytes, &result);
        result = result & p64;

        result1 = (u32)result;
        result = result >> 32;
        resultl = (u32)result;
        result = result >> 32;
        resultl = resultl & p30;
        resultl = resultl & p30;
        poly32_1(resultl, result1);
        (data) += (2*(nhbytes / sizeof(u32)));
    }
}
ticks2 = clock() - ticks2;
ans1=result;
nhstruct1 = (nh_struct*)lpParam;
return 0;
}

DWORD WINAPI NhThread2( LPVOID lpParam )
{
    DWORD dwChars;

    u32 *data;
u32 *key;
struct nh_struct *temp;
int i;
u32 resultl, result1;
temp = (struct nh_struct *)lpParam;
data = temp->mm;
key = temp->kk;
i = nbytes/nhbytes - 1;
u64 result=0;
data = data + (nhbytes / sizeof(u32));
int k=0;

ticks3 = clock();
for(int j=0; j<noofiterations; j++)
{
    while(i>0)
    {
        i= i-2;
        result = (u64)0;
        nh2(data,key,nhbytes,&result);
        result = result & p64;

        resultl = (u32)result;
        result = result >> 32;
        resulth = (u32)result;
        resultl = resultl & p30;
        resulth = resulth & p30;

        poly32_2(resulth,resultl);

        (data) += (2*(nhbytes / sizeof(u32)));
    }
}
ticks3 = clock() - ticks3;
an2=result;

return 0;
}

u32 hash(nh_struct *nhstruct, u32 *mptr, u32 *kptr, int numbytes)
{
    // all the code of hashing will come here from main
    // return value will be 32 bit tag

    DWORD dwThreadId[MAX_THREADS];
    HANDLE hThread[MAX_THREADS];
    u32 finalans;
    int i;

    initialize();

    nh_struct *nhstruct1;
    nhstruct1 = (struct nh_struct *)malloc(sizeof(struct nh_struct));
    nhstruct1->mm = mptr;
    nhstruct1->kk = kptr; // this should come from encryption function
hThread[0] = CreateThread(NULL,0,NhThread1,nhstruct1,0,&dwThreadId[0]);
    if(hThread[0] == NULL)
        exit(2);

hThread[1] = CreateThread(NULL,0,NhThread2,nhstruct1,0,&dwThreadId[1]);
    if(hThread[1] == NULL)
        exit(2);

// Wait until all threads have terminated.
WaitForMultipleObjects(MAX_THREADS, hThread, TRUE, INFINITE);

u64 temp1 = (u64)polyans1 * (u64)x2;
polyans1 = ModByLargePrime1(temp1,large_prime);

//printf("here poly ans 1 = %lx \n",polyans1);
finalans = (polyans1 + polyans2) % large_prime;
printf("final poly answer = %lx \n",finalans);

// Close all thread handles upon completion.
for(i=0; i<MAX_THREADS; i++)
{
    CloseHandle(hThread[i]);
}

return finalans;

}

void mac()
{
    // how i am going to write mac function here.
}

void main(int argc, char *argv[])
{
    PMYDATA pData;
    //DWORD dwThreadId[MAX_THREADS];
    //HANDLE hThread[MAX_THREADS];
    int i;

    long int j,k;
    u64 ans;
    unsigned char *data;
    unsigned char key[nhbytes];
int count1;
int count2;

u64 result, temp;
u32 *mptr;
u32 *kptr;

u32 finalans;

noofiterations = atoi(argv[1]);

int rc1, rc2;

int status;
double cpb;

struct nh_struct *nhstruct;
double secs;

printf("here ");

nbytes = 1024 * 1024;
polykey = polykey % large_prime;
x = polykey;
x2 = polykey2 = (polykey * polykey) % large_prime;
x4 = polykey4 = (polykey2 * polykey2) % large_prime;
x3 = (polykey2 * polykey) % large_prime;
p30 = UINT32_C("0x3ffffffff");
polyans1 = (u32)0;
polyans2 = (u32)0;

//nbytes = 4096;
data = (unsigned char *)malloc(nbytes + 16);
//key = (unsigned char *)malloc(nhbytes);
printf(" before if");
if(data == NULL)
{
    printf("allocation error 
");
    exit(MEMORY_ALLOCATION_ERROR);
}

memset(data+nbytes,0,16); // padding 0s at the end

for(i=0;i<nbytes;i++)
    data[i] = (unsigned char) (i * (i+1));

for(i=0;i<nhbytes;i++)
    key[i] = (unsigned char) (i * (i+1));

mptr = (u32 *)data;
kptr = (u32 *)key;
printf("entering into nh
");
result = (u64)0;

/* filling the struct */
nhstruct = (struct nh_struct *)malloc(sizeof(struct nh_struct));
nhstruct->mm = mptr;
nhstruct->kk = kptr;

// Create MAX_THREADS worker threads.
ticks = clock();

finalans = hash(nhstruct, mptr, kptr, 1000);

ticks = clock() - ticks;

cpb = (double)ticks / nbytes;
secs = (double)ticks / CLOCKS_PER_SEC;

}
APPENDIX C

SOURCE CODE FOR MULTI THREAD IMPLEMENTATION FOR QUAD CORE PROCESSOR

```c
#include <windows.h>
#include <strsafe.h>
#include <stdio.h>
#include <process.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include <sys/types.h>

#define nhbytes 64 // 128 * u32 = 1K
#define MAX_THREADS 4
#define BUF_SIZE 255
#define MEMORY_ALLOCATION_ERROR 2
#define CLOCKS_PER_SEC 1000

typedef unsigned __int32 u32;
typedef unsigned __int64 u64;

u64 ans1 = 0;
u64 ans2 = 0;
u64 ans3 = 0;
u64 ans4 = 0;

u32 polykey2,polykey4,polykey6,polykey8;
u32 polyans1,polyans2,polyans3,polyans4;
u32 x,x2,x3,x4, x6;
u32 p30;

u32 mytest1[4],mytest2[4];
// 13806395525

int nbytes;
clock_t ticks,ticks1,ticks2;

unsigned int large_prime = 1073741823;

struct nh_struct{
    u32 *mm;
}
```
typedef struct _MyData {
    int val1;
    int val2;
} MYDATA, *PMYDATA;

unsigned int UINT32_C(char* s) {
    unsigned int n;
    int r;
    
    r = sscanf(s, "%x", &n);
    
    if (r != 1) {
        printf("error when change hex string to integer.");
        exit(0);
    }
    return n;
}

u32 polykey = UINT32_C("0x3bcdef21");
u64 p64 = UINT32_C("0xffffffffffffffff");

void intialize()
{
    
    // This function will set up all the keys using aes_encryption algorithm.
    // I need to set up polykey, innerproduct key and nh key here and than use them.
    //
    polykey = UINT32_C("0x3bcdef21");
    polykey = polykey % large_prime;
    x = polykey;
    x2 = polykey2 = (polykey * polykey) % large_prime;
    x4 = polykey4 = (polykey2 * polykey2) % large_prime;
    x3 = (polykey2 * polykey) % large_prime;
    x6 = x4 * x2 % large_prime;
    p30 = UINT32_C("0x3fffffff");
    polyans1 = (u32)0;
    polyans2 = (u32)0;
    polyans3 = (u32)0;
    polyans4 = (u32)0;
u32 ModByLargePrime1(u64 num, u32 lp) {
    u32 numl = (u32)num & p30;
    u32 temp = (numl + (numl >> 30) + (num >> 32) * 4);
    if(numl < temp)
        temp++;
    return temp;
}

u32 ModByLargePrime2(u64 num, u32 lp) {
    u32 numl = (u32)num & p30;
    u32 temp = (numl + (numl >> 30) + (num >> 32) * 4);
    if(numl < temp)
        temp++;
    return temp;
}

void poly32_1(u32 m0, u32 m1) {
    u64 num;
    num = (u64)polyans1 * (u64)x4 + (u64)m0 * (u64)x + (u64)m1;
    u32 temp;
    _asm{
        mov         esi,dword ptr [num]
        and         esi,dword ptr [p30 ]  // (u32)num & p30 seperating lower 30 bits
        mov         eax,dword ptr [num]
        shr         eax,1Eh              // (u32)num >> 30 seperating higher 2 bits
        add         esi,eax
        //mov         eax,dword ptr [ebp-8]  // here is where it brings upper 28 bits in eax register
        mov         ecx,20h              // i dont know why this instruction is there
        mov edx, 4h                     // multiplying upper 28 bits by 4. better operation is shift left by 2
        mul edx                         // edx * eax result stored in EDX:EAX so 30 bit multiplication result is stored in EAX
        add esi, eax
        mov ebx, esi
        shr ebx, 1Eh
        and esi, dword ptr [p30]
add esi, ebx // adding upper 2 bits of 32 bit register which represents query
mov ebx, esi
and esi, dword ptr[p30]
shr ebx, 1Eh
add esi, ebx // addition if adding upper 2 bits again
generate query, after this it is guaranteed that
mov dword ptr[temp], esi // query is not going to generated
}
polyans1 = temp;

void poly32_2(u32 m2, u32 m3)
{
    u64 num;
    num = (u64)polyans2 * (u64)x4 + (u64)m2 * (u64)x + (u64)m3;
    u32 temp;
    _asm{
        mov esi,dword ptr [num]
        and esi,dword ptr [p30]
        mov eax,dword ptr [num]
        shr eax,1Eh
        add esi,eax
        //mov eax,dword ptr [num] I don't need lower 32 bits anymore. they are already taken care of
        mov eax,dword ptr [ebp-8] // here is where it brings upper 28 bits in eax register
        mov cl,20h // i dont know why this instruction is there
        mov edx, 4h
        mul edx // edx * eax result stored in EDX:EAX so 30 bit multiplication result is stored in EAX
        add esi, eax
        mov ebx, esi
        shr ebx, 1Eh
        and esi, dword ptr [p30]
        add esi, ebx
        mov ebx, esi
        and esi, dword ptr[p30]
        shr ebx, 1Eh
        add esi, ebx
        mov dword ptr[temp], esi
    }
polyans2 = temp;
void poly32_3(u32 m2, u32 m3)
{
    u64 num;
    num = (u64)polyans2 * (u64)x4 + (u64)m2 * (u64)x + (u64)m3;
    u32 temp;
    _asm{
        mov esi, dword ptr [num]
        and esi, dword ptr [p30 ]
        mov eax, dword ptr [num]
        shr eax, 1Eh
        add esi, eax
        //mov eax,dword ptr [num] I don't need lower 32 bits
        //anymore. they are already taken care of
        mov eax, dword ptr [ebp-8] // here is where it brings
        //upper 28 bits in eax register
        mov cl, 20h // i dont know why this instruction is there
        mov edx, 4h
        mul edx // edx * eax result stored in EDX:EAX so 30 bit
        //multiplication result is stored in EAX
        add esi, eax
        mov ebx, esi
        shr ebx, 1Eh
        and esi, dword ptr [p30]
        add esi, ebx
        mov ebx, esi
        and esi, dword ptr[p30]
        shr ebx, 1Eh
        add esi, ebx
        mov dword ptr[temp], esi
    }
    polyans3 = temp;
}

void poly32_4(u32 m0, u32 m1)
{
    u64 num;
    num = (u64)polyans1 * (u64)x4 + (u64)m0 * (u64)x + (u64)m1;
    u32 temp;
    _asm{

mov     esi,dword ptr [num]
and     esi,dword ptr [p30]  // (u32)num & p30 seperating
//lower 30 bits
mov     eax,dword ptr [num]
shr     eax,1Eh             // (u32)num >> 30 seperating
//higher 2 bits
add     esi,eax
//mov     eax,dword ptr [num] I don't need lower 32 bits
//anymore. they are already taken care of
mov     eax,dword ptr [ebp-8] // here is where it brings
//upper 28 bits in eax register
mov     cl,20h               // i dont know why this instruction is there
mov     edx, 4h              // multiplying upper 28 bits by 4. better
mul     edx                 // edx * eax result stored in EDX:EAX so 30 bit
//multiplication result is stored in EAX
add     esi, eax
mov     ebx, esi
shr     ebx, 1Eh
and     esi, dword ptr [p30]
add     esi, ebx            // adding upper 2 bits of 32 bit register which
//represents query
mov     ebx, esi
and     esi, dword ptr[p30]
shr     ebx, 1Eh
add     esi, ebx           // addition if adding upper 2 bits again
//generate query, after this it is guaranteed that
mov     dword ptr[temp], esi // query is not going to generated
}

polyans4 = temp;
}
void nh1(u32 *m, u32 *k, const int n, u64 *ans)
{
  // m is 1KB, key k is 1 KB same key is used everytime
  // nh is invoked
  // here n represents nhbytes
  int i;
u32 t1,t2;
static u64 answer1 = (u64)0;
for(i=0;i< 16; i+=2)  // value 16 is hard core need to be changes
{
  (*ans) += ((m[i] + k[i]) * (m[i+1] + k[i+1]));
}
//printf(" answer1 = %lld  %llx \n", *ans,*ans);
```c
void nh2(u32 *m, u32 *k, const int n, u64 *ans)
{
    // m is 1KB, key k is 1 KB same key is used everytime
    // nh is invoked
    // here n represents nhbytes

    int i;
    u32 t1,t2;
    static u64 answer2 = (u64)0;
    u64 temp;

    for(i=0;i< 16; i+=2) // value 16 is hard core need to be changes
    {
        (*ans) += ((m[i] + k[i]) * (m[i+1] + k[i+1]));
    }
}
}

void nh3(u32 *m, u32 *k, const int n, u64 *ans)
{
    // m is 1KB, key k is 1 KB same key is used everytime
    // nh is invoked
    // here n represents nhbytes

    int i;
    u32 t1,t2;
    static u64 answer1 = (u64)0;

    for(i=0;i< 16; i+=2) // value 16 is hard core need to be changes
    {
        (*ans) += ((m[i] + k[i]) * (m[i+1] + k[i+1]));
    }
}
}

void nh4(u32 *m, u32 *k, const int n, u64 *ans)
{
    // m is 1KB, key k is 1 KB same key is used everytime
    // nh is invoked
    // here n represents nhbytes
```
int i;
uint32_t t1, t2;
static uint64_t answer1 = (uint64_t)0;

for (i = 0; i < 16; i += 2)  // value 16 is hard core need to be changes
{
    (*ans) += ((m[i] + k[i]) * (m[i+1] + k[i+1]));
}

DWORD WINAPI NhThread1(LPVOID lpParam)
{
    nh_struct* nhstruct1;
    uint32_t* data;
    uint32_t* key;
    struct nh_struct* temp;
    int i;
    uint32_t resulth, resultl;
    temp = (struct nh_struct*) lpParam;
    data = temp->mm;
    key = temp->kk;
    i = nbytes/nhbytes;
    uint64_t result = 0;
    int k = 0;
    while (i > 0)
    {
        i = i - 4;
        result = (uint64_t)0;
        nh1(data, key, nhbytes, &result);
        result = result & p64;

        resultl = (uint32_t)result;
        result = result >> 32;
        resulth = (uint32_t)result;
        resultl = resultl & p30;
        resulth = resulth & p30;

        poly32_1(resulth, resultl);
        (data) += (4*(nhbytes / sizeof(uint32_t)));
    }

    ans1 = result;
    nhstruct1 = (nh_struct*) lpParam;
    return 0;
}
DWORD WINAPI NhThread2( LPVOID lpParam )
{

    DWORD dwChars;

    u32 *data;
    u32 *key;
    struct nh_struct *temp;
    int i;
    u32 resulth,resultl;
    temp = (struct nh_struct *)lpParam;
    data = temp->mm;
    key = temp->kk;
    i = nbytes/nhbytes - 1;
    u64 result=0;
    data = data + (nhbytes / sizeof(u32));
    int k=0;
    while(i>0)
    {
        i = i-4;
        result = (u64)0;
        nh2(data,key,nhbytes,&result);
        result = result & p64;

        resultl = (u32)result;
        result = result >> 32;
        resulth = (u32)result;
        resultl = resultl & p30;
        resulth = resulth & p30;

        poly32_2(resulth,resultl);

        (data) += (4*(nhbytes / sizeof(u32)));
    }

    ans2=result;

    return 0;
}

DWORD WINAPI NhThread3( LPVOID lpParam )
{
    nh_struct* nhstruct1;
    u32 *data;
    u32 *key;
struct nh_struct *temp;
int i;
uint32 result, resultl;
temp = (struct nh_struct *)lpParam;
data = temp->mm;
key = temp->kk;
i = nbytes/nhbytes - 2;
data = data + (nhbytes / sizeof(uint32)) + (nhbytes / sizeof(uint32)) ;
u64 result=0;
int k=0;
while(i>0)
{
    i= i-4;
    result = (u64)0;
    nh3(data,key,nhbytes,&result);
    result = result & p64;

    resultl = (u32)result;
    result = result >> 32;
    resulth = (u32)result;
    resultl = resultl & p30;
    resulth = resulth & p30;

    poly32_1(resulth,resultl);
    (data) += (4*(nhbytes / sizeof(uint32)));
}

ans3=result;
nhstruct1 = (nh_struct*)lpParam;
return 0;

DWORD WINAPI NhThread4( LPVOID lpParam )
{

    nh_struct* nhstruct1;
    uint32 *data;
    uint32 *key;
    struct nh_struct *temp;
    int i;
    uint32 result, resultl;
temp = (struct nh_struct *)lpParam;
data = temp->mm;
key = temp->kk;
i = nbytes/nhbytes - 3;
data = data + (nhbytes / sizeof(uint32)) + (nhbytes / sizeof(uint32)) +
(nhbytes / sizeof(uint32)) ;
u64 result=0;
int k=0;
while(i>0)
{
    i= i-4;
}
result = (u64)0;
nh4(data,key,nhbytes,&result);
result = result & p64;

resultl = (u32)result;
result = result >> 32;
resulth = (u32)result;
resultl = resultl & p30;
resulth = resulth & p30;

poly32_1(resulth,resultl);
(data) += (4*(nhbytes / sizeof(u32)));

} ans4=result;
nhstruct1 = (nh_struct*)lpParam;
return 0;
}

u32 hash(nh_struct *nhstruct, u32 *mptr, u32 *kptr, int numbytes)
{
    // all the code of hashing will come here from main
    // return value will be 32 bit tag

    DWORD dwThreadId[MAX_THREADS];
    HANDLE hThread[MAX_THREADS];
    u32 finalans;
    int i;

    initialize();

    nh_struct *nhstruct1;
    nhstruct1 = (struct nh_struct *)malloc(sizeof(struct nh_struct));
    nhstruct1->mm = mptr;
    nhstruct1->kk = kptr; // this should come from encryption

    function
    hThread[0] =
    CreateThread(NULL,0,NhThread1,nhstruct1,0,&dwThreadId[0]);
    if(hThread[0] == NULL)
        exit(2);
    hThread[1] =
    CreateThread(NULL,0,NhThread2,nhstruct1,0,&dwThreadId[1]);
    if(hThread[1] == NULL)
        exit(2);
    hThread[2] =
    CreateThread(NULL,0,NhThread3,nhstruct1,0,&dwThreadId[2]);
    if(hThread[2] == NULL)
        exit(2);
hThread[3] = CreateThread(NULL, 0, NhThread4, nhstruct1, 0, &dwThreadId[3]);
    if (hThread[3] == NULL)
        exit(2);

    // Wait until all threads have terminated.
    WaitForMultipleObjects(MAX_THREADS, hThread, TRUE, INFINITE);

    u64 temp1 = (u64)polyans1 * (u64)x6;
    polyans1 = temp1 % large_prime;

    temp1 = (u64)polyans2 * (u64)x4;
    polyans2 = temp1 % large_prime;

    temp1 = (u64)polyans3 * (u64)x2;
    polyans3 = temp1 % large_prime;

    finalans = ((polyans1 + polyans2) % large_prime) + ((polyans3 +
    polyans4) % large_prime);

    printf("final poly answer = %lx \n", finalans);

    // Close all thread handles upon completion.
    for (i=0; i<MAX_THREADS; i++)
    {
        CloseHandle(hThread[i]);
    }

    return finalans;

}

void mac()
{
    // how i am going to write mac function here.
}

void main()
{
    PMYDATA pData;
    //DWORD dwThreadId[MAX_THREADS];
    //HANDLE hThread[MAX_THREADS];
int i;

long int j,k;
  u64 ans;
  unsigned char *data;
  unsigned char key[nhbytes];

int count1;
int count2;
  u64 result,temp;
  u32 *mptr;
  u32 *kptr;

u32 finalans;

int rcl,rc2;

int status;
  double cpb;
  struct nh_struct *nhstruct;
  double secs;
  printf("here ");

nbytes = 1024 * 1024 * 8 ;
  polykey = polykey % large_prime;
  x = polykey;
  x2 = polykey2 = (polykey * polykey) % large_prime;
  x4 = polykey4 = (polykey2 * polykey2) % large_prime;
  x3 = (polykey2 * polykey) % large_prime;
  p30 = UINT32_C("0x3fffffff");
  polyns1 =(u32)0;
  polyns2 = (u32)0;
  //nbytes = 4096;
  data = (unsigned char *)malloc(nbytes + 16);
  //key = (unsigned char *)malloc(nhbytes);
  printf(" before if");
  if(data == NULL)
    {
      printf("allocation error \
");
      exit(MEMORY_ALLOCATION_ERROR);
    }

memset(data+nbytes,0,16);  // padding 0s at the end
  for(i=0;i<nbytes;i++)
    data[i] = (unsigned char) (i * (i+1));

  for(i=0;i<nhbytes;i++)
    key[i] = (unsigned char) (i * (i+1));

  mptr = (u32 *)data;
kptr = (u32 *)key;
printf("entering into nh\n");
result = (u64)0;

/* filling the struct */
nhstruct = (struct nh_struct *)malloc(sizeof(struct nh_struct));
nhstruct->mm = mptr;
nhstruct->kk = kptr;

// Create MAX_THREADS worker threads.
ticks = clock();

finalans = hash(nhstruct, mptr, kptr, 1000);

ticks = clock() - ticks;

cpb = (double)ticks / nbytes;
secs = (double)ticks / CLOCKS_PER_SEC;

}
BIBLIOGRAPHY


