LAPIN AMK
Lapland University of Applied Sciences

CONSTRUCTING ALMOST ABELIAN SQUARE-FREE WORDS ON THREE LETTERS IN C LANGUAGE

Hailu Nahom

Thesis
Lapland University of Applied Sciences
Information Technology
Bachelor of Engineering

2016

Author	Nahom Hailu
Supervisor	Veikko Keränen
Title of Thesis	Constructing Almost Abelian Square-Free Words on Three Letters in C Language
Number of pages	32 + 3

In this thesis, the process for extending almost abelian square-free words using the C program and the method used to implement parallel computing for faster execution time was reported. Moreover the method in checking for almost abelian square-freeness of a word was discussed as well. However main concern for this thesis was the programming task and optimizing the program finally obtained. The research final program does not make use of parallel computing. However the method to implement parallel computing for this program was discussed in hope to be used for future researches and improvements.

This research was supervised by principal lecturer of mathematics, Veikko Keränen. Material for this research were based on his previous research papers, programming books and websites such as tutorialpoint.com that facilitated the implementation of the program.

Through this research a program for extending almost abelian square-free words using suffix extension, i.e., extending a word by adding a letter at the suffix, were achieved. Additionally the implementation for parallel computing was discussed. Almost abelian square-free words were found for words of length over one thousand and three hundred. This result can be improved by making use of parallel computing, with the implementation of threads.

Key words: abelian square-free, C program, parallel computing,
threads.

CONTENT

1. INTRODUCTION 5
2. EXTENDING A WORD 7
2.1 Abelian Square Test 7
2.2 Mathematical Strategies Used 9
2.3 Avoiding a Set of Vectors 9
3. ALOGORITHM IN USE 11
4. COMPUTING ANALYSIS OF THE PROGRAM 14
5. PARALLEL COMPUTING 16
6. THREADS OF PTHREAD LIBRARY 19
6.1 Design for Threaded Programs 19
6.2 Thread Routines 20
6.3 Thread Management 21
7. THREADS SYNCHRONIZATION AND LIMITS OF THREADS 23
7.1 Thread Synchronization 23
7.2 Condition Variables 28
7.3 Thread Limits 28
8. CONCLUSION 29
BIBLIOGRAPHY 30
APPENDICES 31

LIST OF FIGURES

Figure 1. Example of the Extension Process for a Word 9
Figure 2. Main Component of the Program 11
Figure 3. A Representation of a Double Linked List 12
Figure 4. The abelianSquareAvoidance header file...13
Figure 5. Graph of Memory Allocation in Function of Time 14
Figure 6. A Representation of Parallel Computing 16
Figure 7. A Representation of Serial Computing 16
Figure 8. Example Code of a Threaded Program..21
Figure 9. Output of the Program in Figure 7 21
Figure 10. Example for Joining Threads...23
Figure 11. Program without the Use of Synchronization 25
Figure 12. Program Using Mutex...26

SYMBOLS AND ABBREVIATIONS

Alphabet	a set of symbols called letters (Keränen 2009, 3894)
Word	a sequence of letters belonging to the alphabet (Keränen 2009, 3894)
Parikh vector	a vector indicating the number occurrences of a letter in a word (Keränen 2009, 3894)
Data structure	a method of organizing data to use it efficiently for programming (Tutorials Point 2016)
Memory leak	a class of bug where a program does not release memory slots that are no longer needed (Microsoft 2016)
Flow of control	the order how instructions, function calls, and statement are evaluated or executed when the program is running (Oxford University Press 2016)
Overhead	resources that are not used directly to obtain the end result but are required by the technology or method used (TechTarget 2016)
API	Application program interface, is a set of routines protocols and tools used for building a software program (QuinStreet Inc 2016)

1. INTRODUCTION

In 1961 Paul Erdös raised the question whether it is possible to avoid abelian squares for arbitrarily long words. This question in the case of four letter alphabets was open until 1992 when Veikko Keränen presented an abelian square-free endomorphism. An abelian square word is by definition a non-empty word uv, where u and v are permutations (anagrams) of each other. For instance, word = *abcbac* is an abelian square word, since *abc* and *bac* are anagrams of each other. The word *abbbcbabca* is also another example of an abelian square word, here the end of the word, *cbabca* is composed of *cba* and *bca* that are anagrams of each other. Anyhow in this research *aa, bb, cc, aaa, bbb, ccc* have to be allowed when constructing almost abelian square-free words. Hence the use of the term almost abelian square-free instead of abelian square-free. (Keränen 2009, 3893.)

The goal of this research is to construct long almost abelian square-free words over three letter alphabet. To find these long words a program must be developed. This thesis will discuss and explain the method used for developing this program. It will primarily focus on the applied and development aspects of the program instead of the theoretical concept of abelian square avoidance.

There are three main parts in this thesis. At first it will introduce the almost abelian square-freeness test and the method used for extending words, then it will discuss and explain concepts of parallel computing and finally it will show ways of implementing parallel computing in C language with the use of Portable Operating System Interface (POSIX).

2. EXTENDING A WORD

2.1 Abelian Square-Freeness Test

This research aimed to find arbitrarily long almost abelian square-free words over three letter alphabet. In order to achieve the above goal a c program was developed. This program is supposed print on a file each time it finds a longer almost abelian square-free word. Then the programme is shared over github.com and Veikko Keränen computes it with a quite powerful computer available to him. A powerful computer is required when running the program, since it requires a larger memory beyond the capacity of a normal personal computer. The run time is mostly spent in checking for abelian square-freeness in the word. The test method used to check for abelian square freeness is elaborated next.

The method used, to check for abelian square freeness is a suffix test. A suffix test is to check for an abelian square freeness by going backward, from the last letter of the word. The testing is continued till the first letter of the word, however the test is stopped if an abelian square has been detected before reaching the first letter.
To perform a suffix test one must create two Parikh vectors. And then it should be checked whether the difference of the two vectors is a zero vector. However in this research abelian square words *aa, bb, cc, aaa, bbb, ccc* are allowed The example below illustrates this test. (Keränen 2016.)

Let u and v be Parikh vectors for the word: *abababb*
Phase 1: $\vec{u} = (1,1,0)$, $\vec{v} = (0,2,0)$
Phase 2: $\vec{u}$ = (1,2,0), $\vec{v}$ = (1,2,0)
Phase 3: $\vec{u}$ = (1,2,0), $\vec{v}$ = (1,3,0)

To check for abelian square-freeness of a word one needs to test if vector $(\vec{v} - \vec{u})$ or $(\vec{u} - \vec{v})$ are zero vectors. If true then it has an abelian square. This means at each phase $(\vec{v} - \vec{u})$ or $(\vec{u} - \vec{v})$ should be calculated. Note that it does not matter if one check only for $(\vec{u} - \vec{v})$ in this case since the target result is zero vector.

phase1: $(\vec{u} - \vec{v})$ = (1, -1, 0)

phase2: $(\vec{u} - \vec{v})$ = (0, 0, 0)

At phase 2 an abelian square is detected hence the test is terminated.

On the other hand a prefix test, does not start from the last letter as a suffix test does but instead it starts from the first letter. However beside the starting point and the direction to which Parikh vectors assign their value, both test use the same method to check for abelian square-freeness. (Keränen 2016.)

Once the word is tested and accepted, next task is to extend the word. This task may seem simple however a strategy is required in order to obtain a longer almost abelian square-free word, as fast as possible. But also in terms of memory use the program must be efficient since it requires high performance from the computer.

2.2 Mathematical Strategies Used

The method used in this research is to extend a word, by adding a suffix to it. That is, the extension take place only at the end of the word, the beginning of the word stays the same. Suffix test is faster than a prefix test for this case, since the extension take place at the end of the word, a suffix test identifies if the added new letter has created an abelian square word before the prefix test because it can find an abelian square without testing the whole word.

According to Veikko, the chosen method to extend a word defined over the alphabet, *abc*, is to insert the letter *a* first. Then if the above create a word that is not almost abelian square-free word, *a* is deleted and replaced by *b* if *b* also result a non-almost abelian square-free word, it is replaced by *c*. But if *c* causes a non-almost abelian square-free word, then the last two letters of the word are deleted. In case where the last block letters are all of letter *c* then they will be deleted until the next non *c* letter.

Figure 1. Example of the Extension Process for a Word

```
abcaaabaaa        [10 letters][AA2free]
abcaaabaaaa        [11 letters] [In backtrack]
abcaaabaaab        [11 letters] [In backtrack]
abcaaabaaac        [11 letters] [In backtrack]
abcaaabaab        [10 letters] [In backtrack]
```

As illustrated in Figure 1, the words with an invalid suffix are put to backtrack to then obtain an almost abelian square-free word.

2.3 Avoiding a Set of Vectors

After the user have given the base word that he would like to extend, the program will ask if he would like to avoid some set of vectors. This is an optional feature of the program, executed only if the user gives the vectors to be avoided. Once an almost abelian square-free word is obtained, the program will check whether the vector difference $\vec{u} - \vec{v}$ and $\vec{v} - \vec{u}$ are not found inside the given set.

Where vector $\vec{u}$ and $\vec{v}$ are all possible combination of two Parikh vectors found in the concern word. If the program indeed finds a match inside the set, the word is rejected hence the program will return to backtracking. However if the user does not want to give any vectors then the program will proceed without executing the function vectors_to_avoid() . For instance, if one has a word = *abcb*, and a set of vectors given by the user is, Set = { (0,2,1), (0,-1,1) }. The vectors $\vec{u}$ and $\vec{v}$ changes their value as described in the table below. Hence the program starts to check if $\vec{u} - \vec{v}$ and $\vec{v} - \vec{u}$ are not in the set as the following, at phase1 $\vec{u} - \vec{v}$ = (0, 1, 0) and $\vec{v} - \vec{u}$ = (0,-1, 0). Both vectors $\vec{u} - \vec{v}$ and $\vec{v} - \vec{u}$ do not belong to the set hence it continues to the next phase, phase 2. This process is repeated untill all possible combinations are checked or until a match is found in the set for $\vec{u} - \vec{v}$ or $\vec{v} - \vec{u}$.

Table 1. Values of vector $\vec{u}$ and $\vec{v}$ at the beginning phases.

Phase	$\vec{u}$	$\vec{v}$

1	(0,1,0)	(0,0,0)
2	(0,1,1)	(0,0,0)
3	(0,2,1)	(0,0,0)
4	(1,2,1)	(0,0,0)
5	(0,1,0)	(0,1,0)
6	(0,1,1)	(0,1,0)

3. ALOGORITHM IN USE

An algorithm in this research is implemented in a software program, to extend and print long almost abelian square-free words. The software program developed is in C programming language. C has been selected since this language allows faster computing compare to high level languages such as Java or C#. C programs runs as a native code without using a virtual machine. Additionally, C gives more memory access and control over the programmer compared to other languages. This means, dynamically programming for instance could be done more efficiently since the programmer can free and allocate memory space at any time. Secondly the thesis candidate is more comfortable in programming in C. Moreover, the thesis advisor suggested to develop a program written in C.

The program is composed of a main function and a component abelianSquareAvoidance. The main function is simply few lines of codes abstracting the process of the program. The figure 2 shows the implementation of this main component. The program begins by asking the user to enter the number of letters in the word that he plans to extend. Then after saving the words length, the program asks the user to enter the word to be extended. Afterwards, the program checks almost abelian square-freeness in the input word, if it the word is almost abelian square-free the extension begins, if the input word however is not an almost abelian square-free word, the program will inform the user immediately and refrains from starting the extension process.

Figure 2. Main Component of the Program

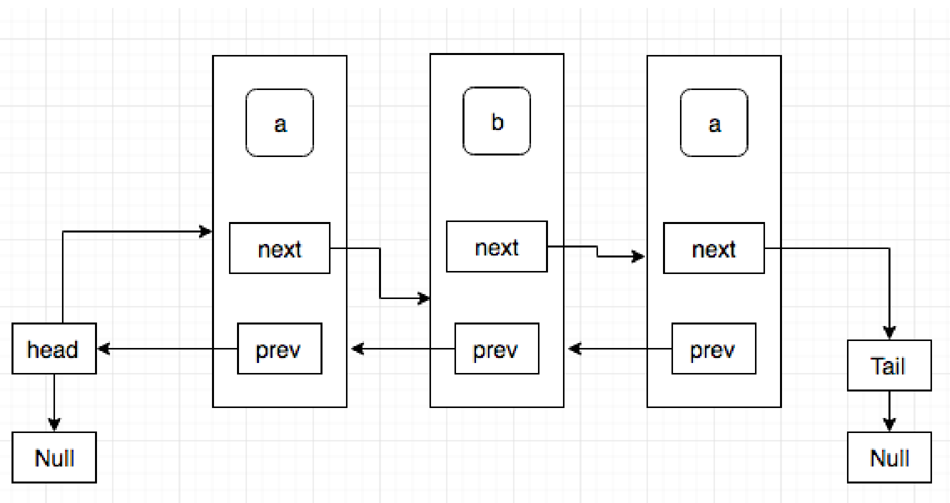
```
#include "abelianSquareAvoidance.h"
#include <stdio.h>

int main(){

    setWordSize("Enter the number of letters of your base word");
    read_baseWord("Enter the base word");

    if(is_abelian_square())
        printf("the word in the list has an abelian square it is not accepted for extending\n");
    else{
        printf("No abelian square detected this word can be extended\n");
        extendWord();
    }
    return 0;
}
```

As illustrated in Figure 2 user is required to give base word length. This function is necessary, although it is possible to use the built in function 'scanf' to read the input word without requiring the user to give the word's length. However, getting the number of letters from the user is preferable than scanning an input word without a length limit. Not fixing a length will cause the program to fix the length automatically. When the length exceeds the fixed amount, scanf will store the remaining characters on another memory slot which may already be in use. This leads to an unexpected behaviour of the program or memory leeks. (Washington State University.)

To save or store the words the program uses a double linked list data structure. A linked list is a sequence of data structures connected by links. These data structures are often referred as nodes. Nodes carry a data but also a link to the next node. A linked list is called a double linked list, when nodes of the linked list are connected to the previous and next node. This allows to navigate backward and forward through the linked list. (Tutorials Point 2016.)

A linked list is the best method for storing the word since the size of the word is changing dynamically. The word gets longer as we extend it and also shrink back, since it deletes the last letter when the word is not almost abelian square-free. Each node of the linked list contains one letter and one link to the next node and another one to the previous node. Figure 3 illustrates this concept.

Figure 3. A Representation of a Double Linked List



As seen in Figure 1 the word lengths changes during the computing of the program. And a double linked list allows to go through backward and forward in the list. During the extending process it is necessary to go forward from previous node to the next node. And when checking for almost abelian square-freeness of a word, it is necessary to go backwards from a node to the previous node. Each node contains one letter of the word and a key that allows it to be identified or distinguished from the other nodes. Identifying the nodes help to track the node and see the letter saved in it. This is useful when checking if the program is repeating the same path of extension, resulting to a dead end.

After setting the length for the input word, programs reads and save each of the word letters in a node of the double linked list. Figure 4 shows the tasks realised in abelianSquareAvoidance component.

Figure 4. The abelianSquareAvoidance header file

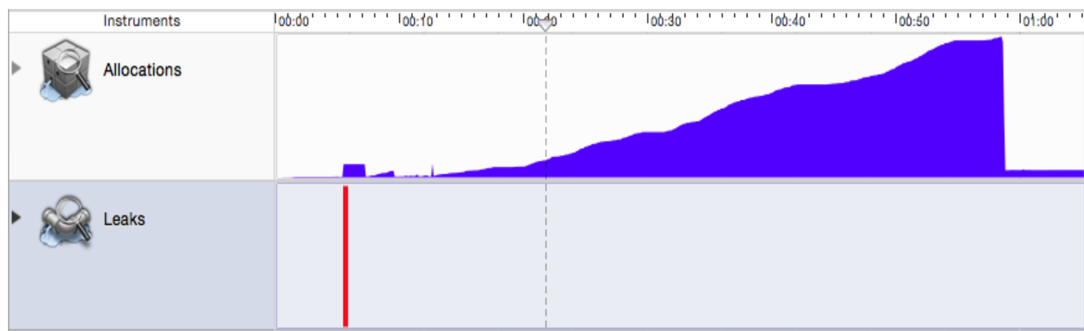
```
#ifndef __Thesis_project__abelianSquareAvoidance__
#define __Thesis_project__abelianSquareAvoidance__

#include <stdio.h>
#include <stdbool.h>

void setWordSize(char *prompt);
void read_baseWord(char *prompt);
char * extend_base_word(char *baseWord, int size);
int convert_letterToParikValue(char letter);
bool is_abelian_square();
void extendWord();

#endif /* defined(__Thesis_project__abelianSquareAvoidance__) */
```

Once the program executes the task at hand it is best to think of optimizing it. In this case the main concerns are the speed and the memory use. It is important to have a faster run time in order to obtain very long words in a shorter time interval. The memory use is also a concern as it may limit the machine from executing the program when the process demands memory space beyond the machine capacity. Before leaping to the optimization of the program it is good practice to analyze it first.

4. COMPUTING ANALYSIS OF THE PROGRAM

The Program profiling allows the developer to understand the program behaviour at run time. Profiling is used in this research to measure the space and time complexity. That is the memory space and the time used by the program are monitored. Profiling helps to optimize the program since the developer will discover the most important areas that needs to be optimized. For instance if a program calls a function only once during the run time and ends the function with a relatively small amount of time then, it is not necessary to focus on optimizing that function. Program profiling guides the developer where to optimize in order to produce a faster version of the program. (The MathWorks Inc 2016.)

Figure 5. Graph of Memory Allocation in Function of Time



The Figure 5 shows the change in memory allocation by the program and memory leaks in function of time. The research program starts running twenty second after the profiling began.

This graph was obtained by the Mac profiler called Instruments. The profiler used as seen in this graph, is a memory leak template profiler. This profiler is focussed at monitoring the memory allocation and leaks. This clearly shows that, there are no memory leaks occurring when the program is running.
As illustrated in this graph the memory allocation size grows when the program is running. The memory allocation draws an increasing straight line. Therefore it can be defined as a linear function, $f(t) = a * S$ where t is the time, S is the allocated memory size and a is a coefficient. Hence it can be deduced that allocated memory is directly proportional to time in this program.

The program may be optimised by extending with parallel computing method, this method will be elaborated later in this thesis. In order to implement parallel computing in C, one needs to distribute the programming tasks into threads. Threads allows the program to execute multiple tasks with greater speed. As a result the extension of the words would take less time. Additionally some of the program loops may be merged or replaced with a recursive functions but one must remember to keep the source code readable since program features may be added or modified by other research participants as long as this research topic is open.

5. PARALLEL COMPUTING

6.1 Parallel Computing and Serial Computing

Parallel computing is the alternative of serial computing. In serial computing a problem is solved by executing a series of instructions in a sequential manner. On the other hand, parallel computing is a simultaneous use of multiple compute resources to solve a computational problem. (Barney 2016a.)

Figure 6. A Representation of Parallel Computing

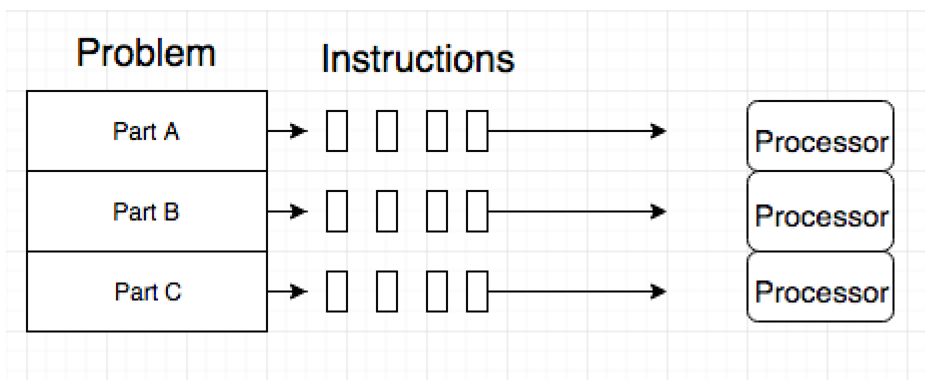
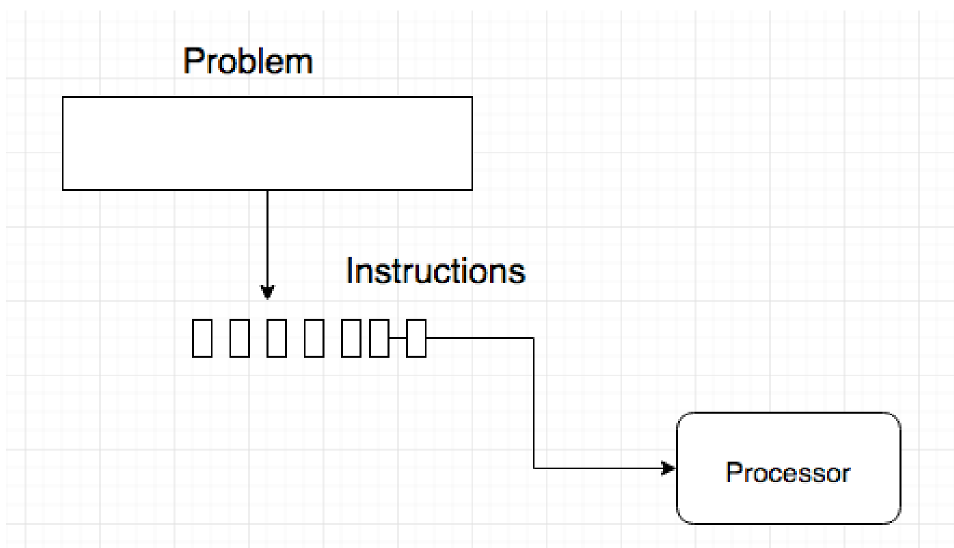


Figure 7. A Representation of Serial Computing



In order to use parallel computing method efficiently the application or program must be designed well. The first task is to define the problem to be solved by the program then, one must divide the problem into parts that can be solved simultaneously as illustrated in Figure 6. Only then the instructions should be written for the specific problem part. Once implemented in the program, these series of instruction will be functions that will be executed on a different processor. (Barney 2016a.)

6.2 Implementation of Parallel Computing In A C Program

After the design is done the following step is the implementation. One of the tools used to implement parallel computing in C and C++, is the threads model. The IEEE POSIX 1003.1c standard specified a programming interface allowing to use threads in C. In this application interface, a program can have multiple concurrent threads, which are simply running tasks of the program. Each thread run independently to one another however, they all share memory space, hence it is possible for threads to communicate and share information. A thread has a lifetime, it can be created and killed in the main program. Threads can be synchronized as well. (Barney 2016b.)

Understanding UNIX process will leads one to better understand how threads function. The operating system creates a process, when the creation of a process occurs the following requires already some amount of overhead. And the process itself contain information about the program execution state and the program resource. Such information includes, Process ID, Environment, working directory, registers and File description. Threads exist inside a process, however they are able to run independently and can also be scheduled by the operating system. This is due to the fact that threads keep an independent flow of control and duplicate only the most important resources allowing them to continue existing as an executable code. To have an independent flow of control threads maintain their own stack pointer, registers, scheduling properties such as policy or priorities, a set of pending or blocked signals and their own specific data given by

the developer. As discussed earlier threads are created inside the process when most of the overhead is already accomplished, hence they are lightweight that is they take relatively small amount of space and operate faster when executed. (Barney 2016b.)

In order to start using POSIX threads or Pthreads in C one must include the header file Pthread.h. POSIX threads is the standard interface for the c programming language although other library may exist according to the hardware vendor POSIX library allows to use threads for all IEEE standard hardware. Sometimes some hardware may offer their own thread library but they would also offer Pthreads. (Barney 2016b.)

Threaded application have some noticeable advantages over non threaded applications. For instance, in the case where a program has a section performing a long Input or output operation. The other task are not required to wait until the above operation ends. Instead a threaded application allows an overlap of Input or output process with a central processing unit work (CPU work) other thread may be used to perform the central processing unit work. Another advantage of threaded applications is the priority or real time scheduling feature, with it important tasks can interrupt other task that are of lower importance. This allow the developer a chance to prioritize running tasks which benefits him when the program is exhausted of resource such as time or memory space. Additionally threaded applications allow an asynchronous event handling meaning some tasks can be executed concurrently when the program is running. One most common application of this feature is used in the process of a web server. It can transfer previously received data while receiving and managing new data arriving. (Barney 2016b.)

6. THREADS OF PTHREAD LIBRARY

6.1 Design for Threaded Programs

The design of threaded programs consider similar principles as of general parallel computing. For a program to use threads efficiently, it must be organized into separated and independent task, able to run simultaneously. To identify a task that is a good candidate to be executed with a thread. One needs to check if the task uses separate resource from others, if its execution is dependant of other tasks or if others task are dependant of the task's result. At the designing phase one needs to limit the need for synchronization between threads and maximize concurrent running. (Barney, 2016b.)

Threaded programs have some common design models. One of these models is called the manager/worker model also known as the boss/worker model. In this model the manager thread creates all worker threads and assign a task to each of them. The manager thread handles inputs and according to the input it passes tasks to worker threads. This models comes commonly in two forms static work pool and dynamic work pool. The use of work pools is favourable when the program has a large number of short tasks to be done. Thread work pools are group of threads pre-instantiated and ready to be given work. This method prevents overhead caused by creating a large amount of new thread. Instead it assign task to existing threads on idle state without a need of creating a new one. Static and Dynamic work pool main difference is that static work pool size does not change, the number of threads stays the same however in dynamic work pool it is possible to destroy and create threads.(Buttlar, Farrell & Nichols 1996, 31-37.)

Pipeline is another design model common for threaded programs. This model is used when there is a long stream of input, many series of sub operations or stages in which each input is processed and when these sub operations are able to process different type of input unit. The best analogy that describes this model

is the car assembly line. Each car goes through different series of stages before it is ready. Even though these stages are in series but it is able to work on many cars at different stages concurrently. The assembly line does need to complete assembling the first car it took, to start assembling another car. Instead it takes another car while the previous one is in some stage of completion. Similarly in the Pipeline model a unit of input pass through different stages, at each stage work is performed on the input unit by a thread. Each thread process and prepare the input unit for the next stage. Multiple inputs at different stages can be processed simultaneously. (Buttlar, Nichols & Farrell 1996, 31-37.)

Peer design model, initially has a thread creating all other worker threads, however this thread does not delegate task, instead it participate in the work. All threads may process request from one input stream shared among them or each of them may have their own input stream. Nevertheless they are all equally responsible of the process.

(Buttlar, Farrell & Nichols 1996, 31-37.)

6.2 Thread Routines

The Ptherad API present different functionalities, all of them can be put into a group of four subroutines. At first there is the thread management routine this routine includes functions such as creating, detaching and joining threads. Then we have the Mutexes routine, mutexe is the short form of mutual exclusion. This routine has functionality such as creating, destroying, locking and unlocking mutexes. Mutex are used for synchronizing threads. Shared resource among threads will not be consistent since one thread might modify before the second one get to access it, mutex solves this issues by limiting access of a data at one thread at a time. Thirdly there is the subroutine of condition and variables this subroutine deals with communication between threads sharing the same mutex. Condition variables are mainly a signalling mechanism associated with mutexes. Finally there is the subroutine of synchronization this encompass routine manage

the reading or writing of mutexes and barriers. Where a barrier is a point in the program that threads stop and wait for every other thread to reach that point. Once all threads reach the barrier they can then continue executing the rest of the code. (Barney 2016b.)

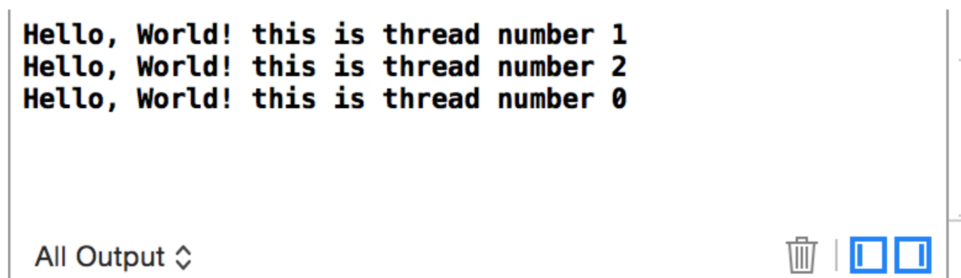
6.3 Thread Management

The program has already a single thread which is the main thread. To create additional thread and execute it, one must use the syntax pthread_create. pthread_create function has four arguments. The first argument is threads, it is an opaque identifier of the newly created thread. Then there is attribute, which is an opaque attribute object used for setting attributes for the new thread. Thirdly there is start_routine, which is the routine the thread will executed once created. Finally there is arg, which is an argument object that will be passed to start_routine. All attributes of the pthread_create function must be passed by reference. (Barney 2016b.)

Figure 8. Example Code of a Threaded Program

```
#include <stdio.h>
#include <pthread.h>


void *printHelloWithThreads(void *threadId){
    long idForThread = (long)threadId;
    printf("Hello, World! this is thread number %ld\n",idForThread);
    pthread_exit(NULL);
}
int main() {
    // insert code here...
    pthread_t tid[3];
    for (long i = 0; i < 3; i++) {
        pthread_create(&tid[i], NULL, printHelloWithThreads,(void *) i );

    }
    pthread_exit(NULL);
    return 0;
}
```

Figure 9. Output of the Program in Figure 7



To create a simple threaded program as illustrated in Figure 8, one must import the pthread.h library, then one must declare the threads using pthread_t data type syntax. One have the option of declaring threads in form of arrays if needed by using pthread_t followed by the name of threads and brackets. For example pthread_t tid[3], declare an array of threads called tid of size 3. After declaring the threads to be used, one must have a function ready to be passed as a start_routine for the corresponding thread. The function must be of a reference type. For instance as shown in Figure 8 the routine printHelloWithThreads is of type void * , since pthread_create attributes must all be of reference types or value types passed by reference. As seen Figure 8 tid[i] is a value type but it is passed by reference in this function. Once created, threads will have default attributes but this attributes may be initialized or deleted using the functions pthread_init and pthread_destroy. (Barney 2016b.)

Threads may be terminated in different ways and scenarios. The most natural way is when a thread has finished its work it will terminate and return back to the function that called it. Secondly, calling pthread_exit() inside a thread will terminate it even if the thread's work is not done. However this call will not make the thread release the process shared resources such as mutexes, condition variables and file descriptors. Another scenario which terminates a thread, is calling pthread_cancel() from another thread and specifying corresponding threads id as an argument. This function will return zero if it were successful, but if it fails it would return the error number.

A thread is also terminated if the process or the program is terminated since thread is part of the process. Finally if main thread finishes before the thread the concerned thread will be terminated. (Barney 2016b.

7. THREADS SYNCHRONIZATION AND LIMITS OF THREADS

7.1 Thread Synchronization

Joining threads is one way of synchronizing threads, to join threads one must use the subroutine pthread_join. When this subroutine is called, it blocks the execution of the calling thread and wait until the thread called completes its operation. One example in the use of this subroutine is to join threads inside the main thread, this ensures that all threads will complete their operations before the main threads terminates. For instance, as illustrated in Figure 10 the main thread will wait thread tid[0], tid[1], tid[2] to finish their operation before terminating the program. However threads are not joinable if they are created detached. To create a detached thread, one needs to set the pthread_attr_setdetachstate() to PTHREAD_CREATE_DETACHED after creating the thread and initializing its attributes with pthread_create() and pthread_init(). One can also detach a joinable thread by using the function pthread_detach(). (Barney 2016b; The Open Group 1997a)

Figure 10. Example for Joining Threads

```
#include <stdio.h>
#include <pthread.h>


void *printHelloWithThreads(void *threadId){
    long idForThread = (long)threadId;
    printf("Hello, World! this is thread number %ld\n",idForThread);
    pthread_exit(NULL);
}
int main() {
    // insert code here...
    pthread_t tid[3];
    for (long i = 0; i < 3; i++) {
        pthread_create(&tid[i], NULL, printHelloWithThreads,(void *) i );
    }

    for (int j = 0; j < 3; j++)
        pthread_join(tid[j], NULL);

    return 0;
}
```

Mutex, or mutual exclusion are the main methods used for synchronizing threads and protecting shared data as mentioned earlier. Mutex ensures that shared data are accessed by only one thread at a time, this means that when multiple threads update the same shared variable, mutex will ensure that the final result will be

the same as if a single thread updated the variable. A mutex variable is locked or owned by only one thread. This means a mutex variable could lock with a new thread if and only if the mutex has been unlocked by the previous thread, until then all attempt of locking this mutex is denied. The process of use of mutex goes as follows, a mutex variable is created and initialized, then several threads try to lock with this mutex and all but one fails, the successful thread locks the mutex and perform its operation on the shared resource, when done it unlocks the mutex, then the mutex is free and again the other threads attempt to access it. This process repeat itself until all threads sharing the same mutex are able to lock it or own it turn by turn, one at time. Finally the mutex is destroyed. (Butenhof 1997,47&48.)

A mutex variable is declared using the syntax pthread_mutex_t. Once declared a mutex variable needs to be initialized before its use. There are two way of initializing a mutex, statically and dynamically. To statically initialize a mutex one calls the routine PTHREAD_MUTEX_INITIALIZER. For example:
pthread_mutex_t newMutex = PTHREAD_MUTEX_INITIALIZER.
In the example above a mutex called newMutex is created and initalized statically. On the other hand dynamically initializing a mutex is done with the use of pthread_mutex_init(). This method allows also to set the mutex attribute objects. When the mutex is no longer needed in the program, it is freed with pthread_mutex_destroy(). (Butenhof 1997,49-51.)

In relation to threads, for a thread to lock a mutex it uses the syntax pthread_mutex_lock() and specifiy the mutex variable inside the brackets. It is also possible to make use of pthread_mutex_trylock() function which does the same task but also returns an error code when the mutex is locked. To unlock a mutex one must use pthread_mutex_unlock() funtion.

Figure bellow show the contrast of using synchronization with mutex and without any synchronization with two examples. The output on Figure 11 is not as expected since main threads terminates. In Figure 12 however a mutex is used to lock thread_deposit and thread_withdraw. Since both thread modify the value of the variable balance which is a shared variable. By locking the threads it is

ensured that the threads won't alter the value in unexpected way but access this shared variable turn by turn. Hence program produces a final result that would have been the same if only a single thread were to be used. (Barney 2016b.)

Figure 11. Program without the Use of Synchronization

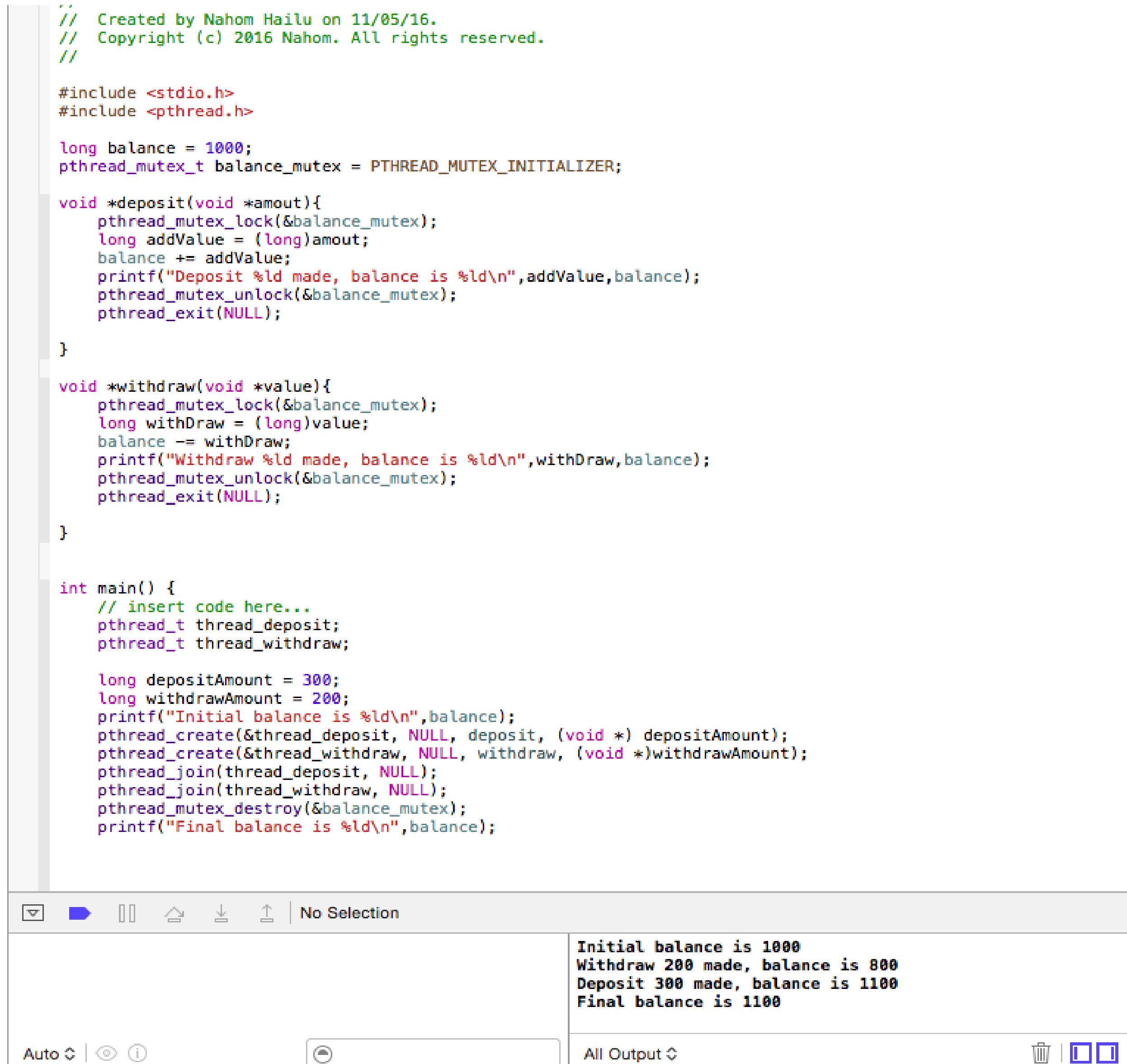
```
#include <stdio.h>
#include <pthread.h>

long balance = 1000;
void *deposit(void *amout){
    long addValue = (long)amout;
    balance += addValue;
    printf("Deposit %ld made, balance is %ld\n",addValue,balance);
    pthread_exit(NULL);

}

void *withdraw(void *value){
    long withDraw = (long)value;
    balance -= withDraw;
    printf("Withdraw %ld made, balance is %ld\n",withDraw,balance);
    pthread_exit(NULL);
}


int main() {
    // insert code here...
    pthread_t thread_deposit;
    pthread_t thread_withdraw;
    long depositAmount = 300;
    long withdrawAmount = 200;
    printf("Initial balance is %ld\n",balance);
    pthread_create(&thread_deposit, NULL, deposit, (void *) depositAmount);
    pthread_create(&thread_withdraw, NULL, withdraw, (void *)withdrawAmount);
    printf("Final balance is %ld",balance);
    pthread_exit(NULL);


    return 0;
}
```

No Selection

```
Initial balance is 1000
Final balance is 1000Deposit 300 made, balance is 1300
Withdraw 200 made, balance is 1100
```

Auto | All Output

Figure 12. Program Using Mutex



7.2 Condition Variables

Condition variables are useful when communicating the state of the shared data. They allow a signaling mechanism for threads. This variables are needed for instance in situation where queue is no longer empty or that it's empty. These variables allow to associate a condition to a thread. The routine pthread_cond_wait, makes the calling thread wait until the specified condition is satisfied.

The routine pthread_cond_signal() is a routine used to signal or wake up other threads when the condition is satisfied, this routine should only be called if the thread is blocked by pthread_cond_wait(), and after this routine is called the mutex should be unlocked. Mutex should be locked before calling pthread_cond_wait(). If one fails to lock the concern mutex before calling these routines, it may not be able to block the thread as pthread_cond_wait() supposed to do. And if one fails to unlock the mutex after pthread_cond_signal() call, the following may not work, the threads may still be blocked.

The routine pthread_cond_broadcasting should be used when there is more than one waiting thread and if the developer wants to signal all of them.
(Barney 2016b; The Open Group 1997b.)

7.3 Thread Limits

Using multiple threads present challenges for the programmer. Even though Pthread Api is ANSI/IEEE standard, the implementation could be different according to the hardware. For instance one need to consider the number of threads allowed or the default stack size for the specific hardware when designing a multiple threaded program. Additionally there are two issues that the developer needs to consider when working with threads. The first issue is collision. As seen earlier all threads share memory space, hence if the developer is not careful, multiple threads may write to the same memory location. This will result a corrupt data. Another issue is the race condition. Race condition occurs when multiple threads can access a shared data, and they try to change or access the data at the same time. Threads will race to access the data because the scheduler of threads may prioritize different threads at different times. This leads into unexpected modification of the shared data. (Barney 2016b.)

8. CONCLUSION

Almost abelian square-free words were found for words of length over one thousand and three hundred, if the program were computed on a powerful computer it would be possible to get longer words. Since the limit for such result was the memory space when run on a normal personal computer. This result can also be improved by making use of parallel computing, with the implementation of threads. Threads will improve the run time, if one does prefix extension and suffix extensions concurrently. And run other functions concurrently. The POSIX API is advised to use since it allows multiple operations on threads, facilitating threads implementation in C. The program of this research is written to be easily read and understood if ever a future research candidate is interested in improving it, or to use it to further the theoretical research of avoidance of Abelian Square. This paper will expedite his research.

BIBLIOGRAPHY

Barney,B. 2016a. Introduction to Parallel Computing. Accessed 20 April 2016 https://computing.llnl.gov/tutorials/parallel_comp.

Barney,B. 2016b. POSIX Threads Programming. Accessed 4 May 2016 https://computing.llnl.gov/tutorials/pthreads.

Butenhof, D. R. 1997. Programming with POSIX Threads. Boston: Addison-Wesley Professional.

Buttlar. D., Farrell. J,P& Nichols. B. 1996. Pthreads Programming. Sebastopol,CA. O'Reilly Media Inc.

Keränen, V. 2009. A Powerful AbelianSquare-Free Substitution over 4 Letters. Theoretical Computer Science. 410/2009, 3893-3900.

Keränen, V. 2016. Lapland University of Applied Sciences. Principal Lecturer of Mathematics's discussion 22 March 2016.

Microsoft 2016. Preventing Memory Leak in Windows Application. Accessed 6 May 2016
https://msdn.microsoft.com/en-us/library/windows/desktop/dd744766(v=vs.85).aspx.

Oxford University Press 2016. Control Flow. Accessed 16 May 2016 http://www.oxforddictionaries.com/definition/english/control-flow?q=control+flow.

QuinStreet Inc 2016. API – application program interface. Accessed 12 May 2016
http://www.webopedia.com/TERM/A/API.html.

TechTarget 2016. Overhead. Accessed 17 May 2016 http://whatis.techtarget.com/definition/overhead.

The MathWorks Inc 2016. Profile to Improve Performance. Accessed 18 May 2016 http://se.mathworks.com/help/matlab/matlab_prog/profiling-for-improving-performance.html?requestedDomain=www.mathworks.com.

The Open Group 1997a. Accessed 7 May 2016 http://pubs.opengroup.org/onlinepubs/007908799/xsh/pthread_cond_wait.html.

The Open Group 1997b. Accessed 5 May 2016 http://pubs.opengroup.org/onlinepubs/007908799/xsh/pthread_detach.html

Tutorials Point 2016. Data Structures - Basic Concepts. Accessed 16 April 2016 http://www.tutorialspoint.com/data_structures_algorithms/data_structures_basics.htm.

Washington State University. Input and the scanf Function. Accessed 18 April 2016 http://www.eecs.wsu.edu/~cs150/reading/scanf.htm.

APPENDICES

Appendix 1

AbelianSquareAvoidance.C

```
9   #include "abelianSquareAvoidance.h"
10  #include <stdio.h>
11  #include <string.h>
12  #include <stdlib.h>
13  #include <stdbool.h>
14  //nodes
15  typedef struct node{
16      char letter;
17      int key;
18      struct node *nextptr;
19      struct node *prev;
20  }word_node;
21  typedef word_node *word_nodePtr;
22
23  //array
24  typedef struct dynamicArray {
25      char* word;
26      int wordSize;
27  }String;
28  word_nodePtr head = NULL;
29  word_nodePtr last = NULL;
30  int the_word_size = 0;
31  bool isopeningFileFirstTime = true;
32
33  //prototypes
34  void printWord();
35  void printTextToFile(char* text);
36  void copyWordFromList(word_nodePtr *headptr, String *stringPtr);
37  void freeString(String *myString);
38  int isEmpty();
39  void insertLastNode(char letter);
40  void deleteLastNode();
41  int deltaVector_isNull(int *vectorU, int *vectorV);
42  void backtrack();
43
44  void read_baseWord(char *prompt){
45
46      enum boo{ yes, no};
47      enum boo allowlooping = yes;
48      char base_word[the_word_size + 1];
```

```
89
90      }
91      the_word_size = wordSize;
92  }
93
94  void extendWord(){
95
96
97      int counter = 0;
98      word_nodePtr tempPtr = last;
99      int prevKey = 0;
100     char prevLetter = ' ';
101     while (tempPtr->prev->prev != NULL ) {
102         prevLetter = last->letter;
103         prevKey = last->key;
104         insertLastNode('a');
105
106         //it should delete the last letter if it creates an abelian square in the word but if the last letter
107         // i need to know when
108
109         if (is_abelian_square()) {
110             backtrack();
111         }
112         if (prevLetter == last->letter && prevKey == last->key && last->key >the_word_size){
113             printf("If the word is the same ");
114             printWord();
115             printTextToFile(" [Repeated word]");
116             while (last->letter == 'c') {
117                 deleteLastNode();
118             }
119             switch (last->letter) {
120                 case 'a':
121                     deleteLastNode();
122                     insertLastNode('b');
123                     break;
124
125                 case 'b':
126                     deleteLastNode();
127                     insertLastNode('c');
128                     break;
129
130             }
131
132             //printf("changing word ");
133             printWord();
134             printTextToFile(" [Changing word] ");
135
```

```
182            }
183
184          //ready for compare
185          if(deltaVector_isNull(parik_vectorU, parik_vectorV))
186              return true; // abelian square detected
187          }
188
189      //Go to next node
190     // testing the values of the parik vectors
191
192    /*  printf("Vector U     \t  Vector V\n");
193      for (int i = 0; i < 3; i++){
194          printf("%d",parik_vectorU[i]);
195          printf("                \t %d",parik_vectorV[i]);
196          printf("\n");
197      }*/
198      tempPtrV = tempPtrV->prev;
199    }
200
201    return false;
202 }
203
204 int deltaVector_isNull(int *vectorU, int *vectorV){
205
206     int delta_vector[3];
207
208     for (int i = 0; i < 3; i ++) {
209          delta_vector[i]= vectorU[i] - vectorV[i];
210         if (delta_vector[i]!= 0) {
211             return 0; //vector is not null
212         }
213     }
214
215     return 1;
216 }
217
218 void printWord(){
219     FILE *fptr;
220     word_nodePtr temPtr = head;
221     int counter = 0;
```