Multi Threaded Pattern Searching of Large Files Using Limited Memory

Peter John Morley
University of Rhode Island, octopusprimetime@gmail.com

MULTI THREADED PATTERN SEARCHING OF LARGE FILES USING LIMITED MEMORY

By

PETER JOHN MORLEY

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF MASTER OF SCIENCE IN

ELECTRICAL ENGINEERING

UNIVERSITY OF RHODE ISLAND

2017

MASTER OF SCIENCE THESIS

OF

PETER JOHN MORLEY

APPROVED:

Thesis Committee:

Major Professor Resit Sendag

Jien-Chung Lo

Jean-Yves Hervé

Nasser H. Zawia

DEAN OF THE GRADUATE SCHOOL

UNIVERSITY OF RHODE ISLAND

2017

ABSTRACT

Pattern searching and discovery for large files is prohibitively slow and requires large amounts of memory for processing. As the number of patterns to process increase, the amount of memory needed increases exponentially exceeding the resources in a traditional computer system. The solution to this problem involves utilizing the hard drive to save pattern information. A program was created called Pattern Finder which saves patterns, keeps track of how much memory it uses and when that threshold is reached, it dumps the information to the hard drive. The other problem inherent with pattern searching besides limited resources is the amount of processing time it takes to complete. To speed up processing, we implement a multithreaded suffix tree pattern finding algorithm that utilizes multiple processing cores. The goal is to mimic Amdahl's law by adding more cores and therefore increasing throughput.

ACKNOWLEDGMENTS

I would like to thank my major professor Resit Sendag who has generously mentored me and helped me become a successful computer engineer. Resit cared about my well-being while I was working 40 hours a week in addition to being a part time student. I also would like to thank all my professors who pushed me harder when my work was not satisfactory. They inspired me and challenged me.

Most of all I would like to thank my parents for their help and support. My mom first introduced me into the realm of engineering and I would not be here without her guidance. My dad taught me that if I work hard and enjoy what I do, the sky is the limit.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

Introduction

1.1 Background

The main objective is finding microprocessor trace patterns to better understand program flow. The goal of this project is to develop a framework for quantitatively analyzing a program's behavior and thereby provide insights into the design of next-generation hardware prediction mechanisms. Our focus is to discover patterns, according to predetermined pattern scoring rules, that occur frequently in input sequences or are characteristic for certain subsets of the input data. Frequently recurring patterns are often indicative of the underlying structure and function. Once all possible patterns are found, one can analyze the pattern results to uncover program behavior.

A pattern at its most granular level is a byte which can be represented by an unsigned char using the C++ language. A byte can represent 256 different combinations and we number levels based on the number of unsigned chars are in the pattern. Level one has one unsigned char as the pattern which will potentially give us 256 different patterns. The number of patterns that can exist for each level is 256 to the Nth power where N is the current level being processed.

The Pattern Finder flow is quite involved but the high-level design focuses on whether each level can be processed using the hardware resources efficiently. The obvious goal is to use only memory (specifically, the Dynamic Random Access Memory (DRAM)) because it is considerably faster than the Hard Drive (HD). The Pattern Finder first begins by predicting how much memory will be used for the next level and how much memory is currently being used and decides to use only DRAM or use both DRAM and HD. The user can either input the DRAM memory limit or the computer will decide for it if not specified. If the prediction involves just DRAM, it will operate much faster but if we must access the drive there will be a timing penalty.

1.2 Objective and Scope

The pattern searching algorithm relies on a tree search methodology. If the tree is completely populated, then each parent node will have 256 child nodes. The relationship between the parent and child node is that the child will always have a level size of the parent's size plus one. The tree methodology allows complete parallelism for processing because each grouping of patterns does not need to know about the other patterns found in other threads. Each grouping of pattern nodes can't have similar patterns with the neighboring nodes because they live on different leaves of the tree. Another advantage of tree searching is that Pattern Finder never has to keep track of the pattern. If the size of the pattern and the location of the last unsigned char in the pattern is known, the program can always reference it later in the searched file.

Figure 1 below shows the pseudo code of how the tree suffix pattern works. The tree will grow in levels until there are no more patterns to be found or the pattern length has reached its maximum size. Each thread will grow its individual tree in parallel without the need for any shared memory. The tree algorithm is used because of its parallel friendly nature which allows the utilization of multi core technology.

```
1.  $Root \leftarrow$ **new node**
2.  $Root.plist \leftarrow (1, 2, \dots, |S|)$
3.  $enqueue(Q, Root)$
4.  **while** $N \leftarrow dequeue(Q)$
5.      $Output\ N.pattern\ and\ its\ occurences$
6.      **foreach** $c \in N.plist$
7.          $Set(c) \leftarrow \boldsymbol{\phi}$
8.      **foreach** $p \in N.plist$
9.          **add** $p + 1$ **to** $Set(S[p])$ **unless** $p = |S|$
10.     **foreach** $c$ **where** $|Set(c)| \geq k$
11.         $P \leftarrow$ **new node**
12.         $P.plist \leftarrow Set(c)$
13.         $N.child(c) \leftarrow P$
14.         $enqueue(Q, P)$
15.     **delete** $N.plist$
```

Figure 3: Frequent subsequences of an input sequence S. Subsequence patterns that occur at least k times in a sequence S.

Figure 2 visualizes how the suffix tree grows and displays how level N will keep track of patterns that are of length N. Another advantage of the suffix tree algorithm is that the previous level data only needs to be kept in memory. The previous level node data contains all of the information starting from the first level.

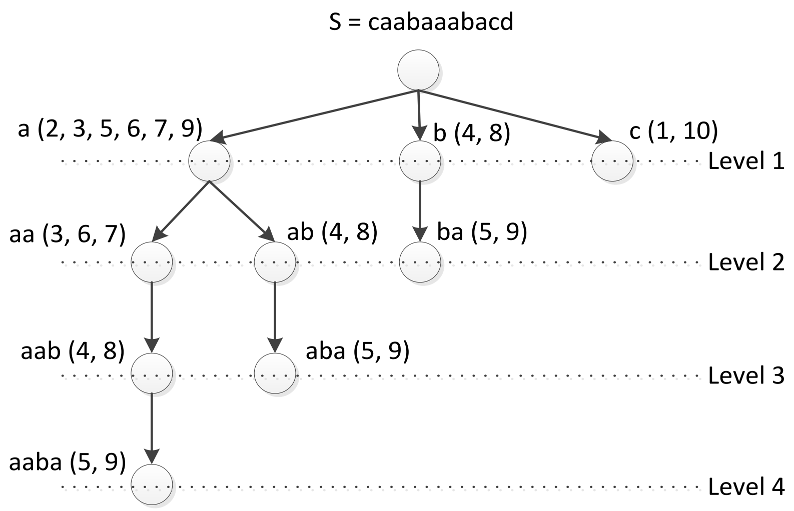


Figure 4: Discovering the subsequences of sequence S→caabaaabacd having at least 2 occurrences in S.

CHAPTER 2

Literature Review and Analysis

2.1 Literature Review

The Kretz paper[1] in section 4.2.1 discusses how random access into a vector can cause havoc on cache spatial locality and therefore increase memory latency problems. The solution discussed includes partitioning the vector data to keep certain memory regions in the cache and then moving on to the next partition bucket when that memory region is exhausted. One thing to point out with the 30 MB L3 shared cache architecture is that if the vector data is less than 30 MB, the algorithm will not suffer from cache misses. The reason this phenomenon occurs is that the data will live in the L3 shared cache because there will never be cache evictions. A 700 MB vector set of data that is randomly accessed will constantly need to be flushed and inserted into the L3 shared cache because the 30 MB capacity can't keep up with random access of a large file. As discussed in the architecture section, there is a 100-cycle penalty for a cache line fetch in L3 shared memory but it is much better than accessing main memory. From reading the Kretz paper[1], the obvious solution is to manage the vector set in partitions or grids of 30 MB sets. One could also infer that using 4 Xeon chips each containing 30 MB of shared cache should allow the Pattern Finder to use 120 MB partitions. That is not accurate because there is no direct way to make sure a certain core is being used by a certain software thread. Two separate software threads could be accessing the same 30 MB vector partition but the cores propelling those threads could be on two different chips, thus they will be using different L3 cache stores and not sharing cache data. The "Cache Performance Analysis of Traversals and Random Accesses" Ladner, Fix and LaMarca paper[2] gives statistics on memory latency penalties and how to overcome them by using certain data structures and access patterns.

CHAPTER 3

Materials and Methods

3.1 DRAM Processing Bottlenecks

Many bottlenecks were overcome while programming the parallel tree pattern processor. The first and most important bottleneck was memory management and access latency. To keep the processing memory size small, there must be many considerations into how each pattern is stored and if need be, offload the data onto the hard drive. Memory access latency is very important because the problem at hand deals with intensive dynamic memory allocation and access. To properly and efficiently use memory there is a need to understand the CPU (Central Processing Unit) architecture limitations, and more specifically how the cache is structured. Another key ingredient in battling bottlenecks is how the operating system manages thread allocation and how threads manage memory. To tackle these issues there will need to be a proper understanding of how the computer hardware is laid out.

3.1.1 Computer Architecture

The ADANA1 server is the main computer that will process large data files in parallel. ADANA1 has four CPU sockets each containing a 12 core Xeon E7-4860 @ 2.6 GHz CPU chip. Each core can utilize hyper-threading which doubles each chip's core count from 12 to 24. The computer gives the Pattern Finder access to running 96 threads in parallel without waiting for the CPU. Reference: https://en.wikipedia.org/wiki/Westmere_(microarchitecture)

The speed and core count of the CPU is a huge processing force but the major focus should be on the cache layout. All the processing power will be waiting for memory access latency if the program does not adhere to proper cache management. Each chip contains three levels of cache and each cache has a different memory access latency speed and size. Level 1 cache is the fastest at a 4-cycle memory latency and contains 64 KB per core. Level 2 cache is the mid-range with a 10-cycle memory latency and contains 256 KB per core. Level 3 cache is the slowest at a 100-cycle memory latency but is different in that all 30 MB of cache is shared with all the cores on the chip. The figure below shows how the cache is structured in relation to each core. To summarize the L3 cache there are 12 cores on each chip that share 30 MB of cache line data unlike the L1 and L2 cache that keep their own caches separated from other cores. The key will be keeping the file data in the 30 MB cache to prevent cache misses and main memory access which is even slower access than the L3 cache. One other thing to keep in mind is the 64-byte cache lines that get loaded from memory to the cache. Creating data objects that align with this cache line size is also another important aspect to maintaining memory in the cache and retrieving the bytes. Accessing data that crosses cache line boundaries can cause serious cache algorithm confusion and force unnecessary cache evictions.

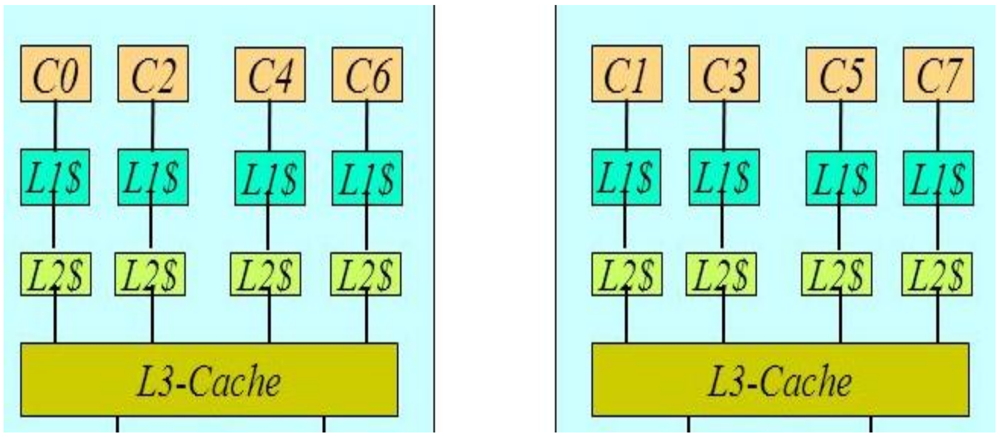


Figure 3: Cache hierarchies

3.1.2 Operating System Memory Management

Another key discussion point is how memory is managed per the OS. Some operating systems like Linux adhere more to the notion of parallel processing memory management while Windows tends to not do as well as some of the Linux Distros. Our research involves using tools from Windows for understanding memory profiling bottleneck issues while running speed tests on the Ubuntu Linux ADANA 1 machine. Most operating systems do not allocate memory in parallel which causes memory access bottlenecks in programs that dynamically allocate memory in a threaded environment. Using fine-tuned memory allocators with threading in mind increases the speed of dynamic allocation and will be explored more in the paper.

3.1.3 STL Vector Memory Approach

Understanding how STL vectors manage memory is essential in making the Pattern Finder memory access fast. Vectors store memory contiguously which utilizes the CPU's ability to keep blocks of memory in the cache while accessing it. Accessing memory linearly must be done to take advantage of memory already loaded into the cache. Later there will be a discussion about the spatial locality caching problem of randomly accessing a large vector of pattern data and how to overcome it. Matthias Kretz, in his "Efficient Use of Multi- and Many-Core Systems with Vectorization and Multithreading" paper[1] suggests partitioning data to minimize random access cache miss penalties when accessing a large vector set.

3.1.4 Memory Allocation Bottleneck

The memory allocation bottleneck is associated with how the Pattern Finder requests for memory from the operating system. The various flavors of memory allocators include Glibc Malloc, Hoard Malloc [13], JE Malloc [12], MT Malloc [11] and TC Malloc. The major issue stems from the necessity to constantly allocate memory when growing patterns. When multi-threading is introduced, the memory allocators need to request data from the memory manager in parallel. Operating system requests for memory must be processed serially which adds a blocking wait time when other threads request memory at the same time. One way to circumvent this bottleneck is to override the new and delete implementations which govern operating system memory allocation. The previously mentioned memory allocators override the new and delete operators and change the implementation of how memory allocation is done. The memory allocator Thread-Cached Malloc known as TC Malloc has been the most successful. In the TC Malloc world there are thread separated miniature heaps that are lockless. Memory is requested in very large blocks and handed out by TC Malloc in small chunks. The memory overhead associated with TC Malloc is much larger but the synchronization of threaded memory allocation is greatly reduced. The larger memory overhead can lead to over allocation because there is a minimum size that each thread will have access to and when the memory allocated per thread gets past a certain memory threshold, the throughput drops significantly. Careful memory management must therefore be a priority when implementing a program that is constantly straining the system for memory resources [6].

3.1.4.1 TC Malloc vs Glibc Malloc Allocation

The memory allocation bottleneck as previously described can be bypassed partially by using other heap memory manager implementations. Glibc (C++ Language) Malloc is the tried and true standard memory allocator. TC Malloc otherwise known as Thread Cached Malloc implements a different style of memory allocator [4]. The Thread Cached implementation utilizes mini heaps to decouple coherent memory allocation in which the Glibc version adheres to. Each spawned thread has its own agnostic miniature heap that will attempt to allocate memory on that heap. If the memory size of the mini heap exceeds a certain threshold, the mini heap must reallocate which takes more time and overhead. To efficiently use and harness the power of TC Malloc, the programmer must be aware that the thread must be relatively lightweight in memory allocation. This opens the idea of thread pooling which spawns many light weight pattern finding tasks. The first figure below uses Glibc Malloc and shows the pitfalls of freeing small chunks of memory very frequently. The reason why Glibc Malloc is so slow when using threads is that Pattern Finder is trying to free memory constantly which creates memory lock contention. At about 80 seconds of processing Pattern Finder slows down to a halt from this problem. The second figure below uses TC Malloc which handles freeing memory much better because it does not have lock contention when freeing small memory chunks. Small chunks of memory never go to the main heap free area in TC Malloc. These small chunks of memory remain in the Pattern Finder program and are cached to optimize memory allocation. Lazy deallocation/memory caching leads to larger memory consumption while improving the throughput of memory requests.

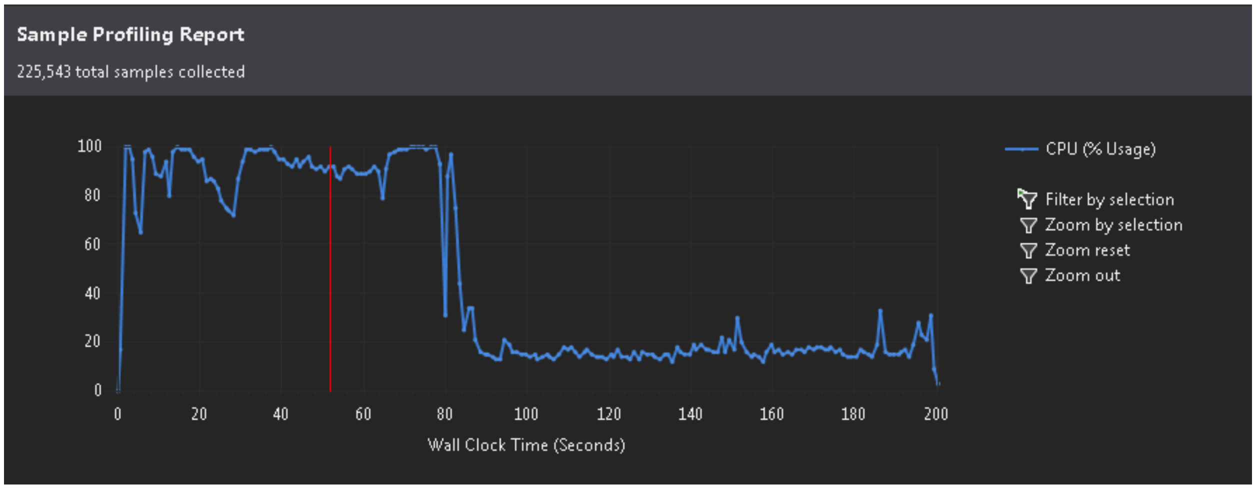


Figure 4: Pattern Finder implementing Glibc's memory allocator

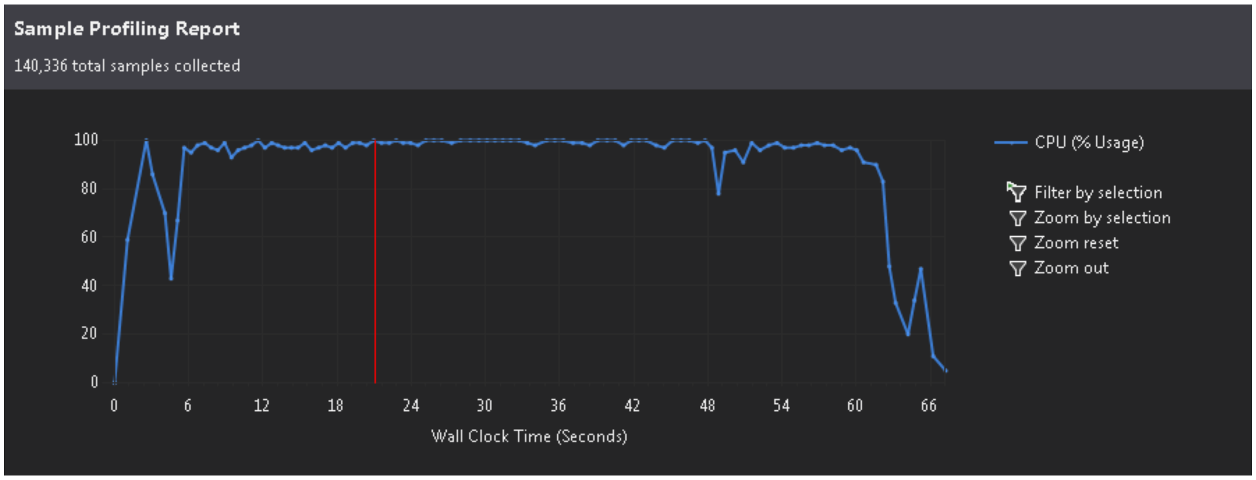


Figure 5: Pattern Finder implementing TC Malloc's memory allocator

TC Malloc has performed significantly better in initial tests than the basic Glibc Malloc heap implementation. The TC Malloc non-thread pooled version of Pattern Finder finished processing a 350 MB file in 5 minutes while the Glibc version lagged its predecessor finishing at the 6-minute mark. The speed improvement from TC Malloc is 6.5x without having to change Pattern Finder's implementation except linking against the TC Malloc library. If there is a way to efficiently create a light weight thread pooling scheme the processing throughput could be even better. This theory must be investigated further and more implementations of Malloc like Hoard and Je Malloc need to be tested.

Unfortunately, initial tests of the previous two allocators have not produced as good of result as TC Malloc thus far.

Figure 2 shows curve trends of thread count versus time taken to allocate a block of memory [1]. Between each thread marker the line trends show memory allocation sizes in each thread against allocation speed. The allocation sizes are between 2^6 and 2^{20} bytes. The trend lines show that between 2^{16} to 64 MB allocations slow down significantly. It is evident that there is a sweet spot in terms of memory allocation size that can be achieved. Current simulations are pre-allocating STL vectors with 2^{10} (1024) to 2^{14} (16384) byte allocations. The issue with this pre-allocation scheme is that memory overhead can start to dramatically increase and must be factored into memory predictions for hard drive or DRAM decision processing.

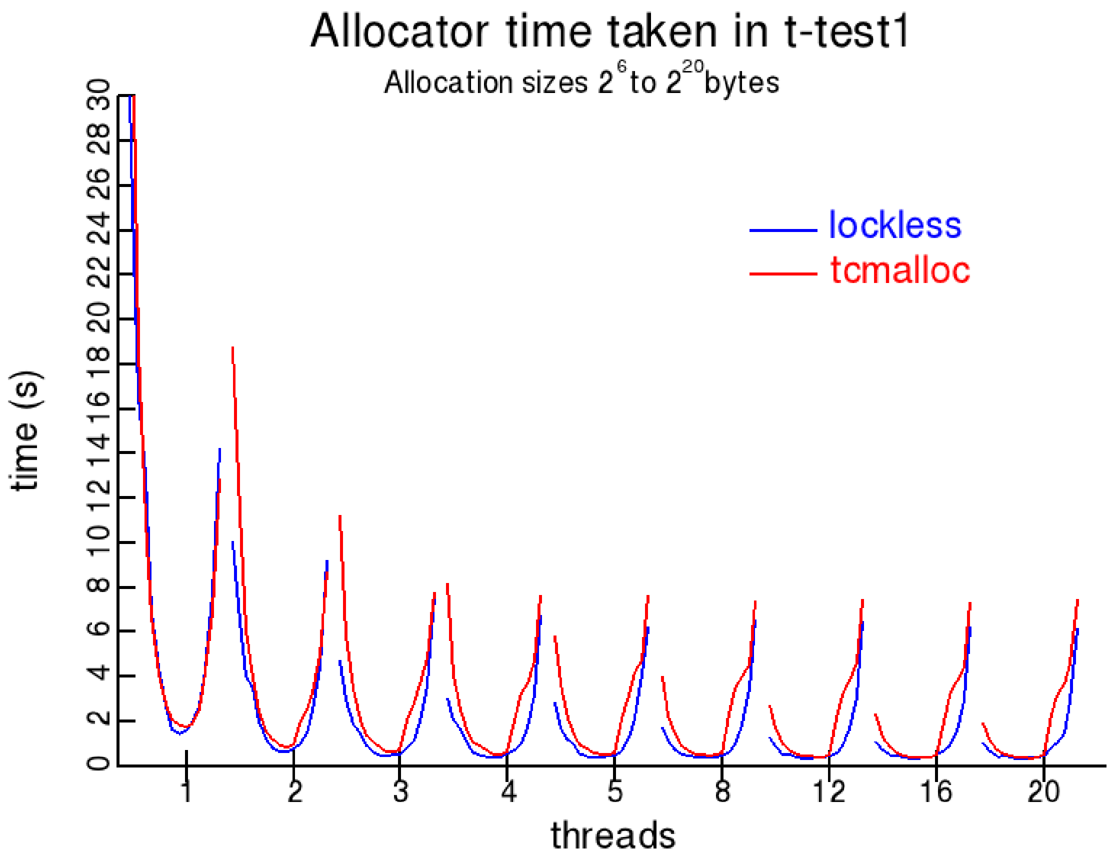


Figure 6: Benchmark tests using lockless and TC Malloc memory allocators with relation the number of threads run

The current implementation using TC Malloc might be inefficient when the thread total is greater than 10 because of how STL (Standard Template Library) vectors are handled. Vectors can grow infinitely and the problem with that is that they are contiguous and therefore constantly reallocating when growing. The thread cached version only works well with small allocation sizes and thus using linked lists to store PLISTS (pattern index lists) would be an implementation to try. The implementation of a linked list of small vectors that never need to be reallocated could show great improvement. This technique would reduce spatial locality caching of memory but would improve the throughput speed if the allocation sizes were somewhere between one kilobyte and 8 kilobytes per vector. Using 8-16 cache blocks per vector allocation would be preferable but still require testing.

3.1.4.2 Thread Building Block Allocation

Threading Building Blocks known as TBB is an Intel API that gives scalable and cache coherent memory allocation [3]. The two main custom vector allocators to take advantage of are called scalable allocator and cache coherent allocator. The main advantage of the scalable allocator is that it was created to be used specifically for threading allocation. Testing this new implementation that overrides the Glibc Malloc allocator received mixed results. The CPU usage went from 60% to 80% but slowed down the processing. Overall the TBB implementation is faster than Glibc but is slower than TC Malloc.

3.1.5 Amdahl's Law Bottleneck

Amdahl's law states that a program which is 95% parallel should get a 6x speedup while using 8 threads [2]. Pattern Finder achieves a 5.2x speedup per previous estimates which is close to the target but not quite there. The problem is that when scaling the thread count to 18, the thread increase only gives us a 6.8 speedup increase. Amdahl's law states that a program should get 9.5x using 18 threads for a 95% parallel program. As one can see in Figure 3, Amdahl's law states there are multiple trend lines for processing in a program that is run in parallel threads. The results follow a program that would run at a 90% parallel portion. At the 8-core mark there should be a 5x speedup and at the 18-core mark there should be a 6.5x speedup. This would mean that Pattern Finder is between 95 and 90 percent parallel. If Pattern Finder can become 95% parallel, there would be better throughput potentially.

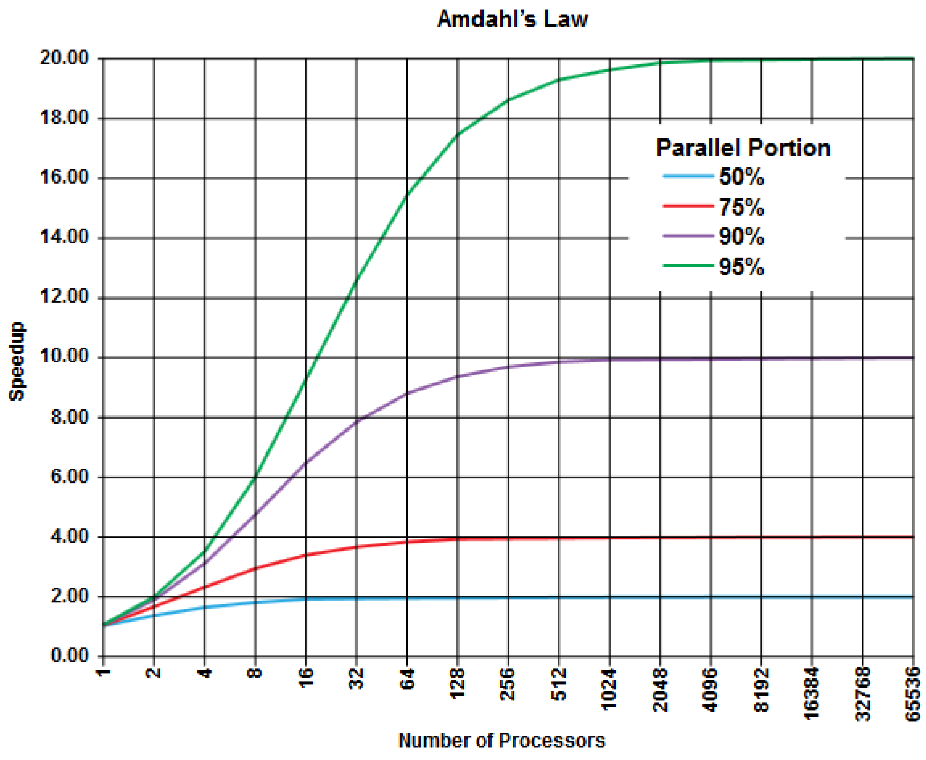


Figure 7: Amdahl’s law shows that a program that is 90% and 95% parallel have very different throughput speedups

3.1.5.1 92, 95 and 97 Percent Parallel Amdahl Design

The next phase in figuring out how to make Pattern Finder more efficient was to key in on Amdahl’s law. More specifically to understand the trend lines for programs that were 95 to 90 percent parallel. The difference in throughput between a 95 and 90 percent program is quite significant and is a 12 to 8 throughput ratio respectively. Previous trend line test results of Amdahl’s law confirmed Pattern Finder was at about a 90 to 92 percent parallel efficiency. To achieve 95 percent parallel portion throughput there was a need to redesign Pattern Finder. The new design dispatched one set of threads and processed all the patterns until completion. The previous design dispatched a set of threads per level which made Pattern Finder much more serial than it needed to be. Using this new design, Pattern Finder finally achieved the expected throughput of a 95 percent parallel program. Two tests were run; one with 16 threads and the other with 32 threads. The increase in throughput was 8.6x and 12.2x respectively. These results followed the trend line of a 95% parallel program to the T. Understanding Amdahl’s law was the key to realizing a more parallel friendly Pattern Finder. Figure 4 on the next page shows the speed versus CPU utilization of 8 cores using the 92 percent parallel Pattern Finder (Top Graph) and the 95 percent parallel Pattern Finder (Center Graph). Clearly the center graph is running at top gear the entire time and there exists shorter periods in which Pattern Finder runs serially. Both versions of Pattern Finder suffer from running serially at the initial level which can be seen in both graphs between the 7 and 14 seconds’ markers. The third and final Pattern Finder version (Bottom Graph) shows a program that is 97 percent parallel which isn’t covered in Amdahl’s trend lines from the previous figure. Implementing try locks on shared memory in the first level closed in on 100 percent parallel programming but still suffered from locking. There is a dip in performance for a short period in the bottom graph versus the previous graphs which show a significant serial approach to processing the data which slows processing down. The 97 percent parallel Pattern Finder version jumped from a speedup of 12.2x from the 95 percent parallel program to 15.2x by just adding that 2 percent of parallelism. At this point in the game, Pattern Finder is running 32 threads and produces a throughput enhancement of 15.2x faster than the serial process using 1 thread.

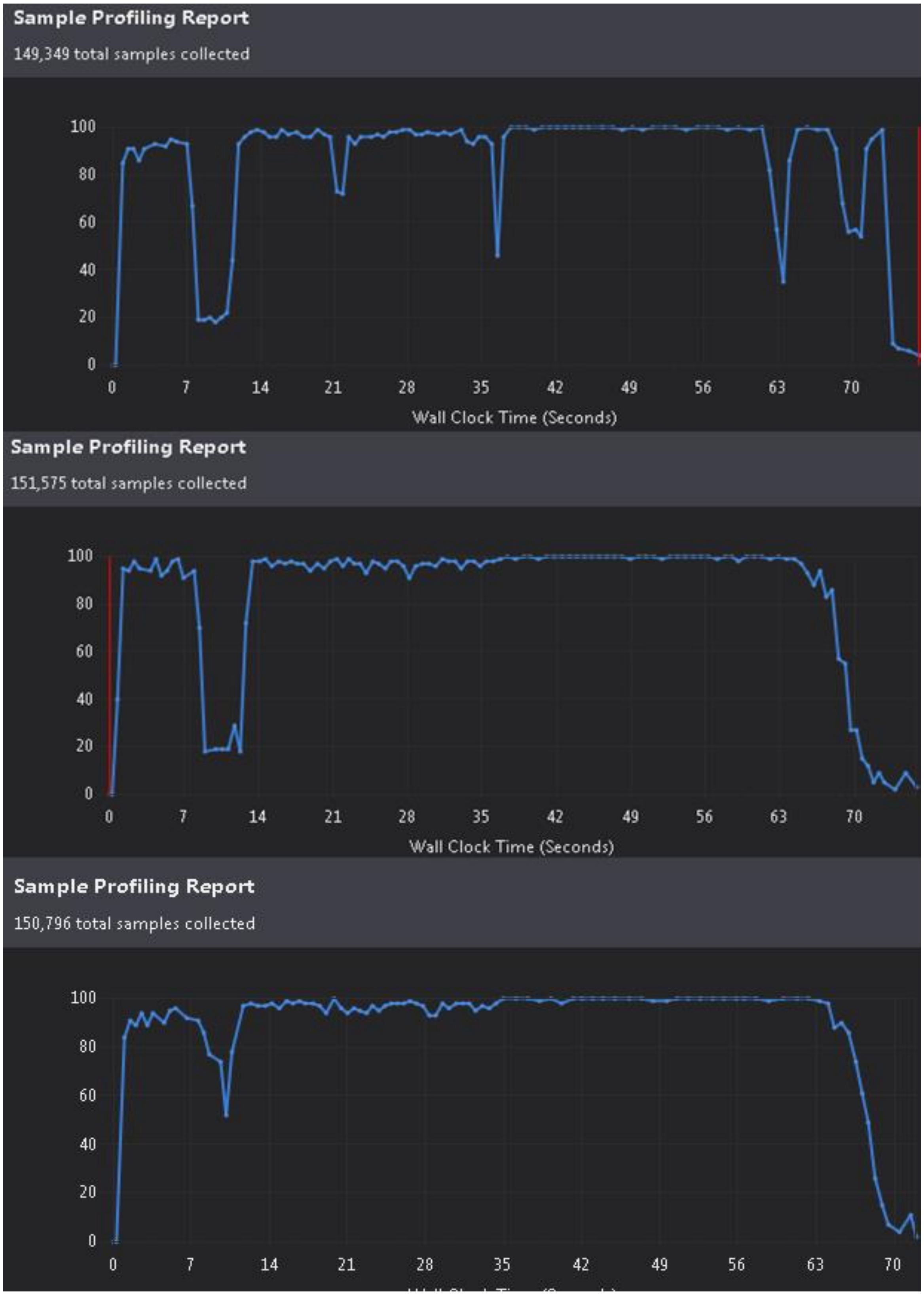


Figure 8: From top to bottom shows a 92, 95 and 97 percent parallel program’s CPU utilization

3.1.6 First Level Thread Improvements

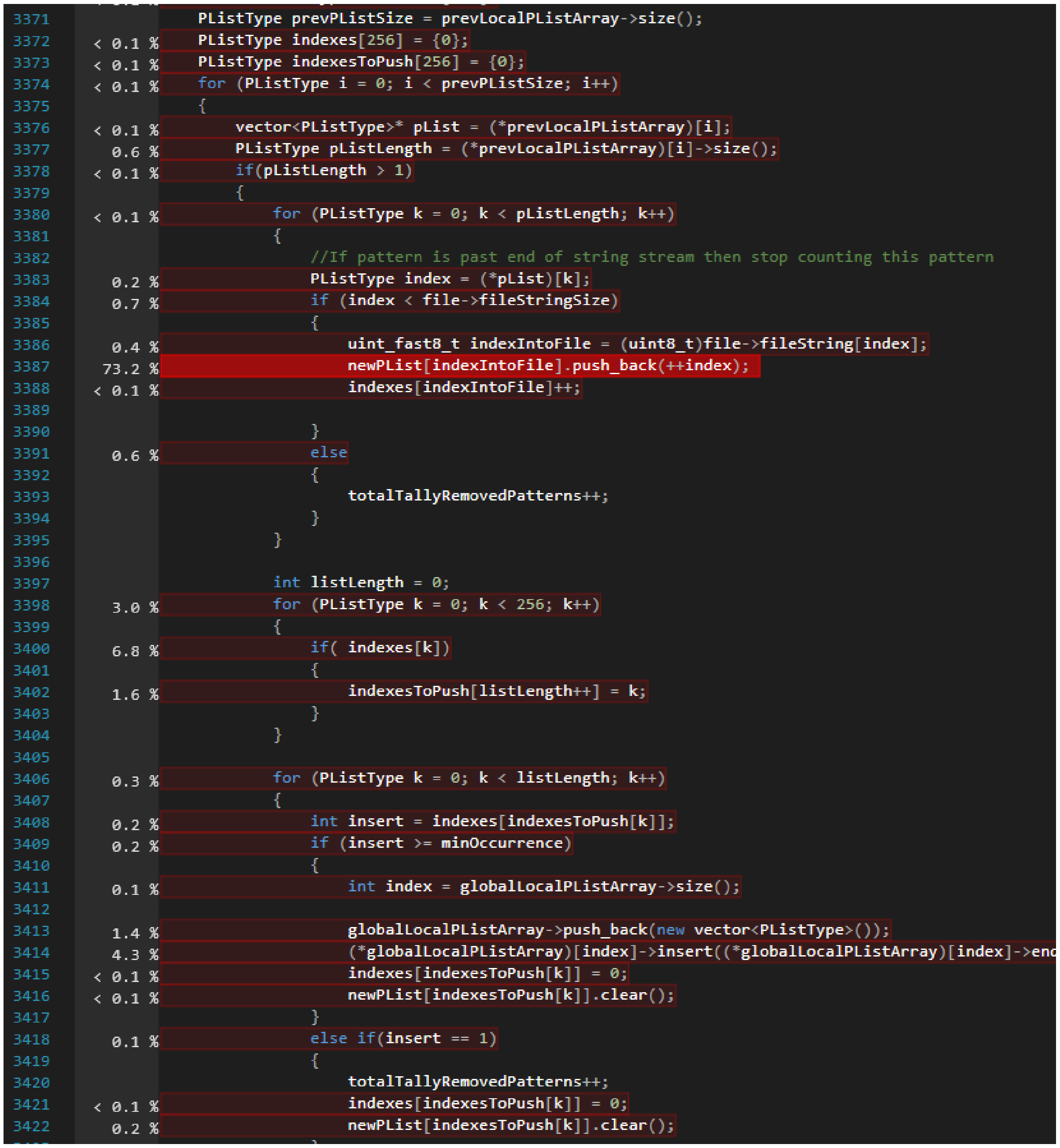
One difficult hurdle to overcome is implementing the first level to be processed in a completely parallel fashion. The issue is that the first level's job is to process the entire file and subsequently partition patterns into pattern buckets. Partitioning the patterns into a main bucket store involves bringing each thread's data together and copying over into one unit. To prevent this problem from occurring there is a need to create duplicate buckets that are labeled as having the same pattern. They are not contained in common vectors but are tagged as the same. Giving tags to the first level indicating their pattern prevents sequential amalgamations of data. Implementing this solution gives us about a 99% parallel Pattern Finder as discussed in the previous section.

3.1.7 Random File Access

The memory bottleneck of randomly accessing file information from a large file is displayed in this code sample. The Visual Studio profiler shows that line 3387 is taking the longest time in the application. The previous line is our show stopper because Pattern Finder is randomly accessing a very large file that the L3 Cache can't hold on to. This causes cache misses and memory access latency penalties of 100 CPU cycles. One improvement upon this design is to preload all the string information needed for the entire level processing. Preloading takes out all the other variables that will be stored in cache during the main processing loop and frees up more cache for our random access. Preloading in no way solves our problem but still manages to give Pattern Finder a nice boost in speed.

The preloading algorithm utilizes Memory Level Parallelism (MLP) which is the capability for a program to have multiple pending memory accesses in parallel. MLP in a 5-stage pipeline gives Pattern Finder the capability for one core to process 4 memory accesses in parallel because there are no dependencies between the memory accesses. Pattern Finder essentially gained a 4x throughput for the single threaded case from MLP due to the 5-stage pipeline.

The random string preloading operation creates a major pitfall for the threading aspect of Pattern Finder. What happens is that the sequential version of the program eliminates random string access latency. The result is a very fast sequential Pattern Finder while the multithreaded version still struggles to keep memory in the cache and therefore the throughput is not as scalable when increasing thread count. The problem is each thread accesses the string randomly at the beginning of each level and therefore introduces cache misses all over the place. Say for example 8 threads are accessing the same file but the memory location spans between the threads are very large. The constant memory jumping creates a situation where the cache is continually pulling and evicting memory from its stores and this phenomenon is called thrashing the cache.



```
            PListType prevPListSize = prevLocalPListArray->size();
< 0.1 %     PListType indexes[256] = {0};
< 0.1 %     PListType indexesToPush[256] = {0};
< 0.1 %     for (PListType i = 0; i < prevPListSize; i++)
            {
< 0.1 %         vector<PListType>* pList = (*prevLocalPListArray)[i];
  0.6 %         PListType pListLength = (*prevLocalPListArray)[i]->size();
< 0.1 %         if(pListLength > 1)
                {
< 0.1 %             for (PListType k = 0; k < pListLength; k++)
                    {
                        //If pattern is past end of string stream then stop counting this pattern
  0.2 %                 PListType index = (*pList)[k];
  0.7 %                 if (index < file->fileStringSize)
                        {
  0.4 %                     uint_fast8_t indexIntoFile = (uint8_t)file->fileString[index];
 73.2 %                     newPList[indexIntoFile].push_back(++index);
< 0.1 %                     indexes[indexIntoFile]++;

                        }
  0.6 %                 else
                        {
                            totalTallyRemovedPatterns++;
                        }
                    }

                    int listLength = 0;
  3.0 %             for (PListType k = 0; k < 256; k++)
                    {
  6.8 %                 if( indexes[k])
                        {
  1.6 %                     indexesToPush[listLength++] = k;
                        }
                    }

  0.3 %             for (PListType k = 0; k < listLength; k++)
                    {
  0.2 %                 int insert = indexes[indexesToPush[k]];
  0.2 %                 if (insert >= minOccurrence)
                        {
  0.1 %                     int index = globalLocalPListArray->size();

  1.4 %                     globalLocalPListArray->push_back(new vector<PListType>());
  4.3 %                     (*globalLocalPListArray)[index]->insert((*globalLocalPListArray)[index]->en
< 0.1 %                     indexes[indexesToPush[k]] = 0;
< 0.1 %                     newPList[indexesToPush[k]].clear();
                        }
  0.1 %                 else if(insert == 1)
                        {
                            totalTallyRemovedPatterns++;
< 0.1 %                     indexes[indexesToPush[k]] = 0;
  0.2 %                     newPList[indexesToPush[k]].clear();
```

Figure 9: Code Snippet highlighting the area where the cache is continually being missed after a subsequent fetch

3.1.8 Thread Pooling Management

The threading scheme used for this project utilized STL Threads which are an improvement over other threading libraries like pthreads (POSIX Threads). STL Threads have added support for polling if a thread has finished its execution. Polling a thread's status is vital in implementing a thread pooling hierarchy. Once a thread is finished with its job it can then be dispatched right away to do more work. The thread pool ensures that all threads will be put to work and thus increasing the throughput of Pattern Finder.

3.1.8.1 Thread Recursion

One issue with the design improvement of dispatching pattern searching threads is balancing the work load. After the initial first level processing, threads are dispatched with certain workloads that will most likely never be equal. Sometimes the workloads can be quite lopsided because some pattern nodes can be negligible. In the previous implementation threads were dispatched with balanced nodes and would terminate processing when all the patters were obtained. To achieve the same throughput as designed there would need to be a way to put threads back to work when they completed their task. The solution to the problem is to have each thread poll to see if any cores are available to help with a job. If a thread has finished working, then any thread can grab it and utilize its processing power. The smarter implementation would be to use a thread priority queue in which the thread who has completed the least amount of work should be given more thread resources to finish. Larger files usually fall victim to this issue where workloads are not equal. The recursive thread pool approach improved throughput from 13x to 14.8x using 32 threads for a 2.8 GB media file.

3.1.8.2 Balanced Workload Approach

One major bottleneck of Pattern Finder as previously described is managing each threads workload evenly. The best way to approach this problem is to do a simple count of each pattern's size and distribute them to each thread. If Pattern Finder can keep each thread's workload spread as evenly as possible then Pattern Finder will not have to spawn threads as frequently. There will always be the need to dispatch threads within threads because of the inherent nature of finding patterns. Pattern searching is random and thus there is no way to know how many patterns will be found on a certain tree node. The balanced workload approach for larger files tends to have inconsistent workloads and they yield a much better throughput. The previous unbalanced Pattern Finder processed a 2.8 GB file in 110 milliseconds while the latest and greatest balanced approach produced a processing speed of 89 milliseconds. The balanced load approach yielded a 1.24x speedup.

3.2 Limited Memory Hard Drive Processing

A computer will always have a limited amount of DRAM and therefore any program needs to know it's limitations while processing pattern data. At startup, Pattern Finder queries the system's information and assesses if a file can be processed completely in DRAM or if it needs help from the Hard Drive. The user can explicitly input a DRAM memory limit but most users should let the program decide if the system can handle the file. Every time Pattern Finder finishes processing a pattern level it must subsequently make a conservative judgment call on whether the memory resources will satisfy processing on DRAM alone. Finding the best way to process patterns on the Hard Drive will be introduced in the following topics. Techniques borrowed from the "Better External

Memory Suffix Array Construction" [15] will prove to be very beneficial in understanding and deconstructing the problem.

3.2.1 Hard Drive Processing Design

In the case of a file being larger than a computer's DRAM capacity, the computer will need to come up with a strategy for processing. Utilizing the Hard Drive to offload memory gives us the ability to process data without resorting to use Hard Drive as virtual memory. The three main memory components in processing each pattern level is composed of the previous level's nodes, the input file being processed and the new level's nodes being generated on the fly. The pattern data will be held in a map which keys off pattern data strings and points to the pattern's indexes in the corresponding vector [7]. The first step in dividing memory resources is taking a thread's memory limit and dividing that value by three for our three main processing components. The first processing memory component is populated with a portion of the file to be processed, the second allocation reads the previous level's pattern data from Hard Drive and the last allocation is left available for the processing of new pattern data. To not over step the memory bandwidth, Pattern Finder constantly monitors its memory usage and when it has reached the memory threshold limit, the new nodes get dumped to the Hard Drive utilizing a fast writing technique called Memory Mapping[5]. The strategy of loading in quadrants of the file, extracting previous pattern nodes from Hard Drive and creating new blocks of pattern nodes allows the processing of massive files. After all the new nodes have been processed and written to the Hard Drive, Pattern Finder takes the stored files and merges them into a coherent pattern tree. One major caveat in processing these partial pattern files is the existence of patterns occurring across multiple files.

3.2.1.1 Memory Mapping

Memory mapping is used to increase the throughput of reading and writing to Hard Drive. Instead of the standard way of accessing a file using OS system read/write calls, memory mapping creates a file handle that a program accesses as if it were a pointer to heap memory. Specifically, the memory being mapped is the OS kernel's page cache meaning that no middle man copies of data must be created in user space.

3.2.1.1.1 Writing to Hard Drive with Memory Mapping

Writing pattern data or any type of data to the Hard Drive is extremely slow. One way to make writing faster is to treat the data as a contiguous block of memory. Memory mapping essentially cordons off a region of memory on the disk and returns a pointer of the data type to be written. The memory size requested can be dynamic but the most efficient way to write to the disk is to grab 2 MB blocks for Windows and 4KB blocks for Linux. The reason behind the "arbitrary" 2 MB/ 4 KB blocks is in part due to the page size of the Hard Drive. The paging size of a disk can be different and is dependent upon the OS. The OS manages read/writes to the hard drive by loading in pages of the drive to DRAM and to make the process as efficient as possible there is a need to load in memory at the page boundary. Memory Mapping avoids normal OS memory management and creates a virtual memory space for blocks that are loaded in. Memory mapping gives us an abstraction layer to the virtual page memory by allocating a page size block of memory in DRAM that will eventually get pushed to the virtual page memory on disk when a flush command is issued.

Pattern data must be stored using two different types of files. The first file contains the pattern position data which contains several positions per pattern followed by the actual position data. The second file contains the pattern information so it does not have to be extracted again from file. The actual pattern must be stored in file because of the nature of hard drive pattern searching. First off, if DRAM is at capacity then the patterns must be offloaded to disk. Those patterns must be recovered later because there are other potential patterns that could exist in other areas of the file. After all parts of the file have been processed, then the pattern dump files can be folded together to compose a complete picture of the pattern data.

3.2.1.1.2 Reading from Hard Drive Example

Reading from the complete pattern files involves extracting segments of the data based on the memory constraints given. In one scenario Pattern Finder is run using 8 threads with a memory constraint of 8 GB. Each thread will then be given a memory cap of 1 GB each. The processing sequence utilizes three storage containers for memory and therefore must be then subdivided again leaving 333 MB per container. At the very epoch of processing a level, 333 MB of pattern data is read from the Hard Drive as mentioned previously. Subsequently 333 MB of the processed file is loaded into DRAM leaving a 333 MB DRAM block. The final DRAM block is used to create new patterns which get offloaded onto the Hard Drive when that 333 MB threshold has been exceeded. This process gets repeated until the entire file has been completely assessed. If the file size is

1 GB, then this process will be repeated 3 times and will yield 3 new pattern files stored on the Hard Drive. Post processing must be triggered which then takes the 3 pattern files and folds all the collected patterns into one coherent pattern file.

3.2.1.1.3 Reading from Hard Drive Bottleneck

One major bottleneck associated with Hard Drive processing is the time it takes for the OS to create a handle to a file when trying to acquire read/write access. What has been found is Pattern Finder spends a lot of time waiting for files to be opened. The files that are being opened have already been generated so there should be no overhead in terms of file creation. One way to circumvent this issue is to create a vector containing file handles on a per thread basis. If a file has been created, then the thread will keep that handle open until it has determined that the file is no longer necessary for processing.

3.2.1.1.4 Threading Memory Map Writes

The main reason why Memory Mapping is so powerful involves how it writes to the Hard Drive. When a pattern of sequence data is written using Memory Mapping, a buffer in DRAM is created that points to the data. The OS is responsible for deciding when to push the buffer to the Hard Drive and release the buffer from DRAM. A problem arises in which memory that should be offloaded to the Hard Drive can temporarily float around in a DRAM buffer until the OS makes the decision it is time to push the buffer to the Hard Drive. If the system is being stressed for resources and Pattern Finder intends to offload pattern data, there needs to be a guarantee that DRAM will be freed up. One way to circumvent this issue is to literally flush this buffer to Hard Drive by commanding the OS to push the buffer straight to the Hard Drive. A flush to the Hard Drive command is time consuming and so the decision was made to dispatch these force commands on threads. These threads alleviated the write processing from halting while waiting for a memory flush to be fully committed.

3.2.2 Dynamic Level Processing

If the user doesn't request that the pattern processing be done solely using DRAM or HD, then Pattern Finder is forced to dynamically manage the memory of the system. At the beginning of every level, Pattern Finder will decide if the system has enough memory to process the level without using the HD. Certain levels of processing take very large amounts of memory while other levels might only take a very small amount. The dynamic memory decision making exploits this concept by preventing potential HD processing slowdowns when the memory needed for a certain level is negligible compared to the size of available DRAM.

3.2.2.1 HD or DRAM Decision Algorithm

The algorithm to determine how a level is processed involves factoring in the previous level's pattern information. There is a potential that each pattern from the previous level can produce 256 new patterns because the next level adds on a new 8-bit value that map to 256 new unique values. This means Pattern Finder must always incorporate the worst-case scenario when doing memory prediction so the predicted pattern count for the next level must be the previous level count multiplied by 256. The other key factor is coming up with an actual memory count for the next level of processing. Each pattern is stored as a vector of 64 bit unsigned integers containing the locations of each individual pattern. The algorithm cannot predict how each of the patterns will be stored or if there will even by an instance of one but it can figure out the remaining potential patterns. Each pattern has a certain shelf life but the maximum size a pattern can be is the size of file minus 1. The algorithm takes advantage of this fact by keeping tally of eliminated patterns and using that value to predict the potential new patterns that can be generated. For example, if the file size is 20 MB and the number of eliminated patterns is 15 MB then Pattern Finder can deduce that the potential number of next level patterns can at a maximum be 5 million. It does not matter how many patterns are accounted for in the previous level, the potential count will always be truncated by 5 million if the previous level count multiplied by 256 exceeds that 5 million.

3.2.2.2 Reducing File I/O Slowdowns

File handling takes a very long time because Pattern Finder waits for the OS to properly create a handle for offloading pattern data. To avoid slowdowns associated with creating file handles, Pattern Finder must keep file handles open to prevent as much IO calls to the OS as possible. At first Pattern Finder created a file to write pattern data and then subsequently closed that file. Later that file would be opened again for processing or collating patterns together. Instead of having two open calls to the OS, Pattern Finder keeps the file handle alive until the file is no longer needed. Reducing the number of open calls significantly improves processing speed and the average throughput increase was a 2x speedup.

Another important topic to be aware of is that each OS has a maximum number of file handles that can be open at once. The default maximum number of file handles on Windows is 512 while Linux is 1024 which can cause processing failures if there are too many file handles open. One way to prevent this event from ever occurring is to increase the limit of file handles that can be opened at once. In Windows the limit of file descriptors can easily be changed in a program using a system call but Linux requires changes to system files which makes it harder to manage this problem. To make sure Pattern Finder works regardless of file handle limitations it must understand the limit and adjust appropriately by closing the file and then opening it later.

Offloading the deletion of files on another thread is also beneficial in the speedup of processing. An OS delete file call takes a lot of time which blocks the processing of further data until it is completely removed. Instead, the processing thread notifies a file removal thread that a file needs to be deleted allowing processing to be done more quickly by offloading the file delete call elsewhere.

3.2.2.3 Memory Watchdog

Even though Pattern Finder’s dynamic memory predictor should take care of most memory predictions, there remains the unreliable nature of how the OS (Operating System) allocates and deallocates memory. In C++, there is the notion that memory is handled by the programmer which is why they call C++ an unmanaged language. The programmer does not have as much control over the memory as one would think. When memory is allocated in C++, the OS will typically bring in more memory than is necessary. Typically, the reason behind this idea is that memory resides at certain boundaries and the OS is designed to realize that memory will most likely be requested soon after to be used again. Another unpredictable aspect of OS memory management is how memory gets returned to the OS. A C++ delete call notifies Pattern Finder that a certain block of memory no longer needs to be managed and can be released to the OS. The problem with this concept is that the OS can leave released memory cached for the process and will not return it back to the OS. This tradeoff is quite efficient because program caching allows memory to be reused quickly without requesting for more memory from the OS. Even though this will improve speed it will not always return memory back to the OS as expected. The main driver making the decisions on how memory is managed between the OS and Pattern Finder is called the Virtual Memory Manager. The VMM determines when memory gets returned and how memory gets fetched for Pattern Finder. Based on how the VMM was implemented, there can be many discrepancies between various OS memory controllers. The memory watchdog determines if the memory being used by Pattern Finder is nearing capacity and will make an executive decision to force a flush of all memory resources back to the OS. The flush resets all memory resources and begins processing with a blank memory slate. This process must be done to prevent virtual memory page thrashing which occurs when pages of Pattern Finder memory must be constantly read from disk to accommodate the entire program’s memory needs. Paging prevents an OS from crashing when memory is exceeded because virtual page memory is usually on the order of two times the size of DRAM.

3.2.2.3.1 Code portability

Pattern Finder is cross platform for Linux and Windows giving the user the capability of running the Pattern Finder on a personal windows box or scale up to a Linux super computer cluster.

CHAPTER 4

Conclusion

4.1 Results and Discussion

4.1.1 Multiprocessor Scalability

Overall performance improved significantly utilizing the knowledge of how the cache performs and understanding the CPU architecture. The figure below shows the throughput versus the thread count from a selection of files illustrating the potential variance in throughput. The throughput achieved models a program that is 95% parallel according to Amdahl's Law.

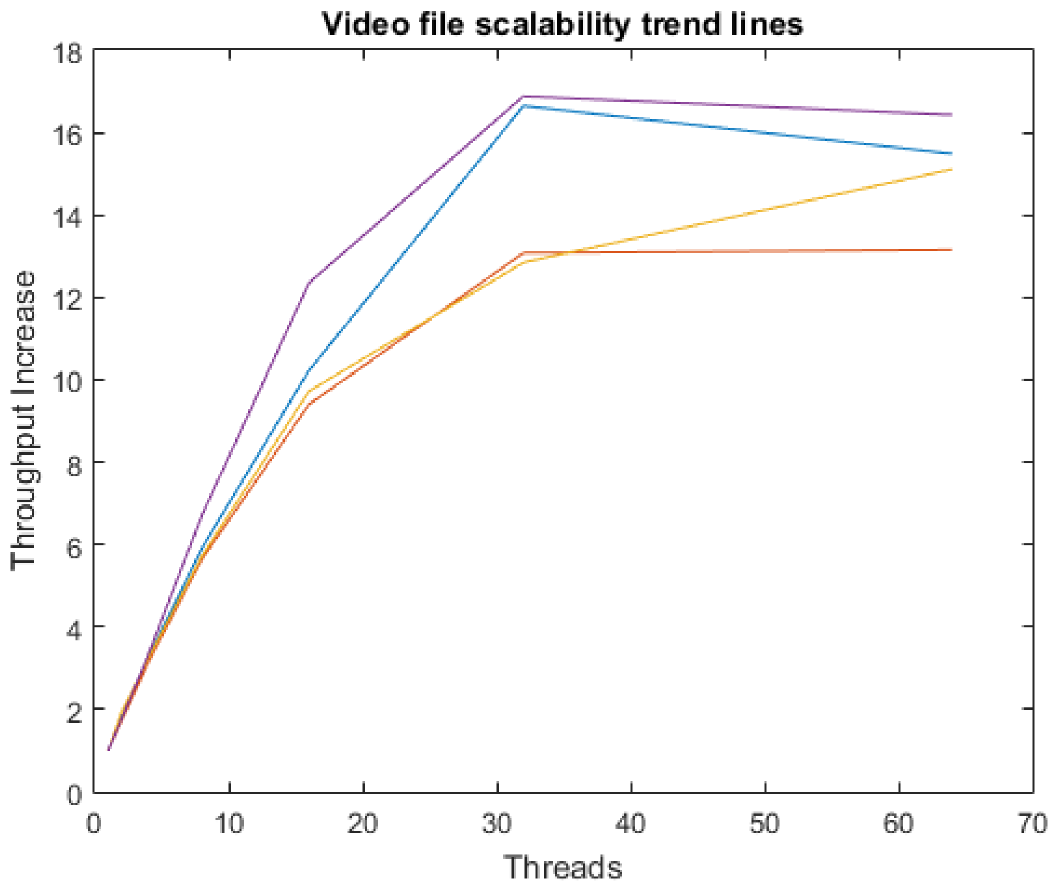
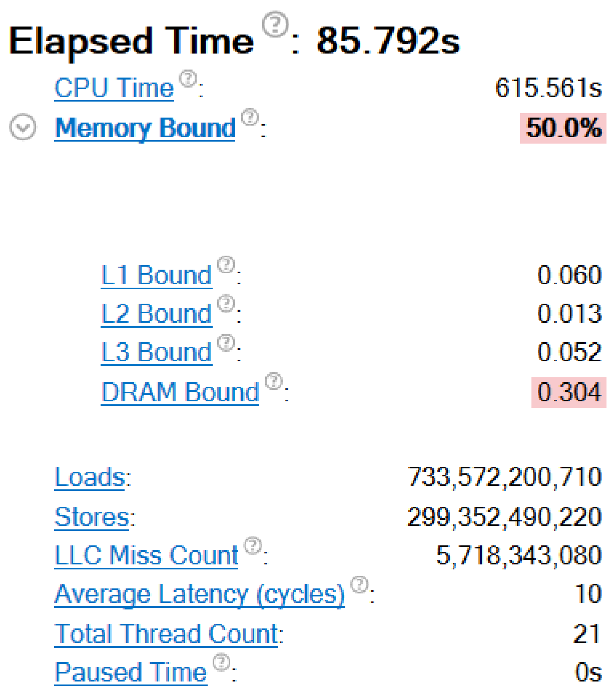


Figure 10: Multithreading throughput trend lines for 4 different video files

4.1.2 Cache Latency

Pattern searching is inherently a memory bound application that relies heavily upon accessing memory. To properly understand how Pattern Finder operates in relation to the hardware one can use profiling tools such as the Intel Profiling Suite. The major optimization strategy when dealing with a memory heavy application is to access data in a fashion that takes advantage of the memory latency hierarchy. Prefetching cache lines from DRAM will significantly improve the speed of memory access if the memory is sequentially and spatially accessed. If Pattern Finder accesses memory in long memory strides, the caching system will never yield L1 cache-speed memory access. Pattern Finder will continuously interrogate DRAM for new memory locations outside of the previous cache line scope. Profiling can show the L1, L2, L3 cache and DRAM statistics including how many cache miss events occurred. The profiling tools uncovered a scenario that if the file being processed was the size of the CPU's L3 cache then eventually the entire file's memory would reside in the L3 cache for quicker memory access. Unfortunately, an L3 sized file will not always be the case and therefore the user cannot rely on a program that can only process small files extremely efficiently. The more likely processing scenario would be a file that is larger than the CPU cache and so the program must therefore avoid random i.e. non-sequential memory access. The pattern searching algorithm inherently involves non-sequential access because each pattern node only processes locations where similar patterns were found. For example, pattern nodes are processed one at a time so there is a potential that one instance of a pattern could be at the beginning of a file and the next instance could be 100 MB down the line in memory. Caching memory will be of no use in this example because every memory access will require a new cache line fetch from main memory (DRAM). The intel profiling utility gives us access to individual cache hit rates which can give us insightful information into which type of cache (L1, L2, L3) most of the memory is being fetched from. Below is a benchmark of a file being processed that is small enough to fit into the L3 cache memory which has the largest memory latency of all the caches but is still 2-4 times faster than DRAM access. The tables show that Pattern Finder is 50% memory bound which is to be expected but the interesting metric is the average memory latency. An average wait period of 10 CPU cycles per memory fetch indicates that the memory access pattern facilitates L2 efficient caching. L1, L2 and L3 caching latencies are 4, 10 and 40-75 CPU cycles respectively. The L3 cache hit latency is not as defined because of the shared nature of the L3 memory between CPU cores. There is extra overhead for an L3 cache because it must know which core made the change to some piece of data. As mentioned before, the L3 cache is unique compared to the L1 and L2 cache because multiple cores share the L3 address space while the L1 and L2 are dedicated to a single core. This capability of the L3 cache allows sharing and synchronization of multiple cores accessing the same cache memory.



Elapsed Time:	85.792s
CPU Time:	615.561s
Memory Bound:	50.0%
L1 Bound:	0.060
L2 Bound:	0.013
L3 Bound:	0.052
DRAM Bound:	0.304
Loads:	733,572,200,710
Stores:	299,352,490,220
LLC Miss Count:	5,718,343,080
Average Latency (cycles):	10
Total Thread Count:	21
Paused Time:	0s

Figure 11: Memory cycle load latency for a video file

4.1.3 Large Pattern File Isolation

Another good utility of Pattern Finder is the ability to process a large collection of data files in a directory recursively. Pattern information about each file is collected and then compared against the neighboring files. An important part of Pattern Finder is to isolate files that contain large patterns so the user can identify what file to focus on in their reseearch. The matlab scatter plot displays processing time versus the size of the file. Each bubble represents one file in a large collection of files. There is a definitive linear correlation for this file set in terms of processing versus file size. The linear correlation gives the program a bias trend line which can be compared to individual files to find the larger pattern files. The bias acts like a machine learning mechanism in which Pattern Finder samples a large data set and comes up with a processing model to compare subsequent files with. The circled files on the scatter plot do not follow the processing trend and therefore must contain large patterns within.

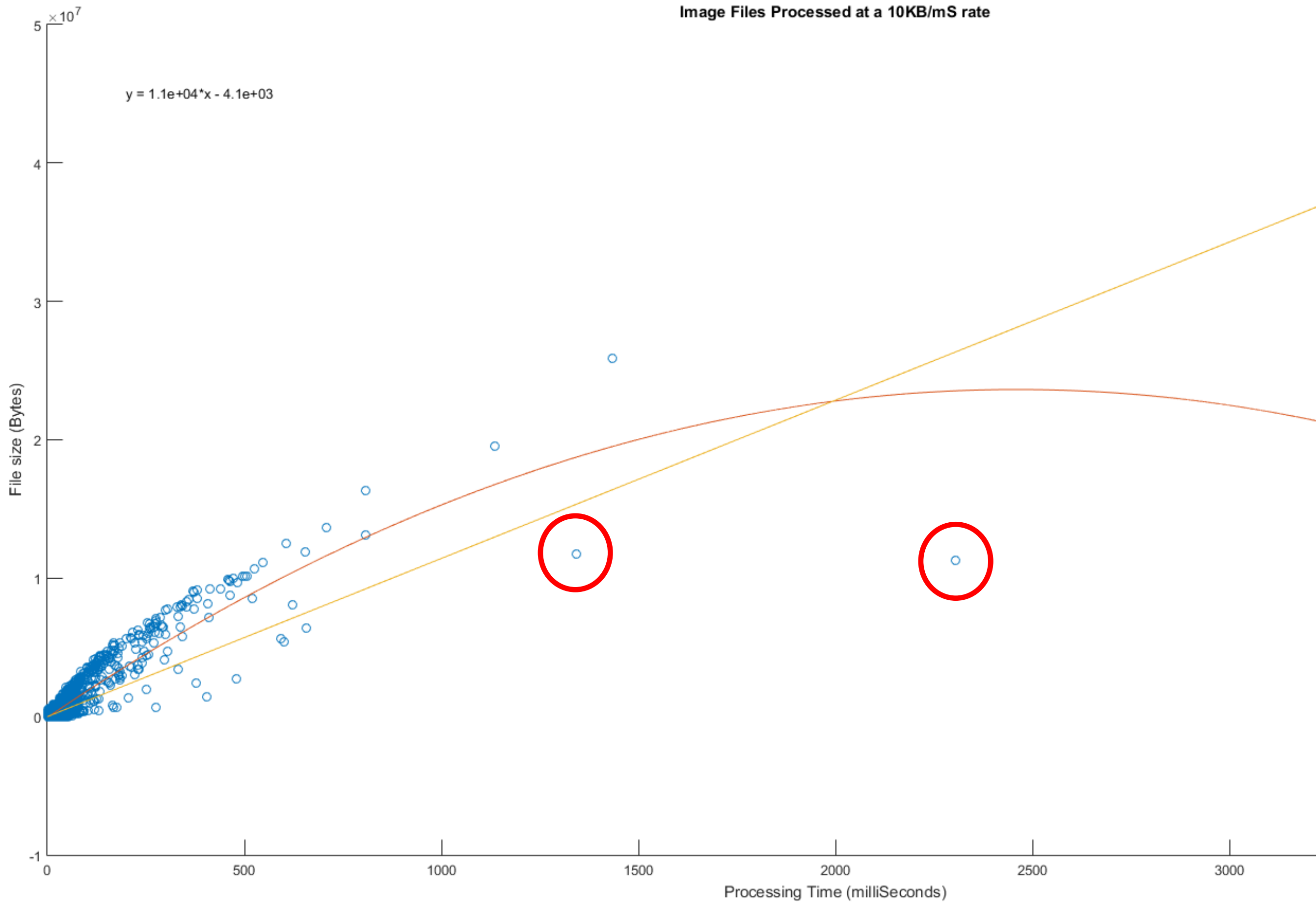


Figure 12: A large data set is processed and large pattern files can be singled out by looking at the outliers

4.1.4 Memory Limited Processing

The matlab bar graph below illustrates the slow down when processing with a memory limitation. The bar graph goes from left to right with the memory going from unlimited to 100 MB for a file that is 250 MB. Pattern Finder can simulate a DRAM size of 100 MB and can process a 250 MB file successfully without bringing down the system from paging. The slowdown from an unlimited DRAM supply to 100 MB is on the factor of 5500 times slower.

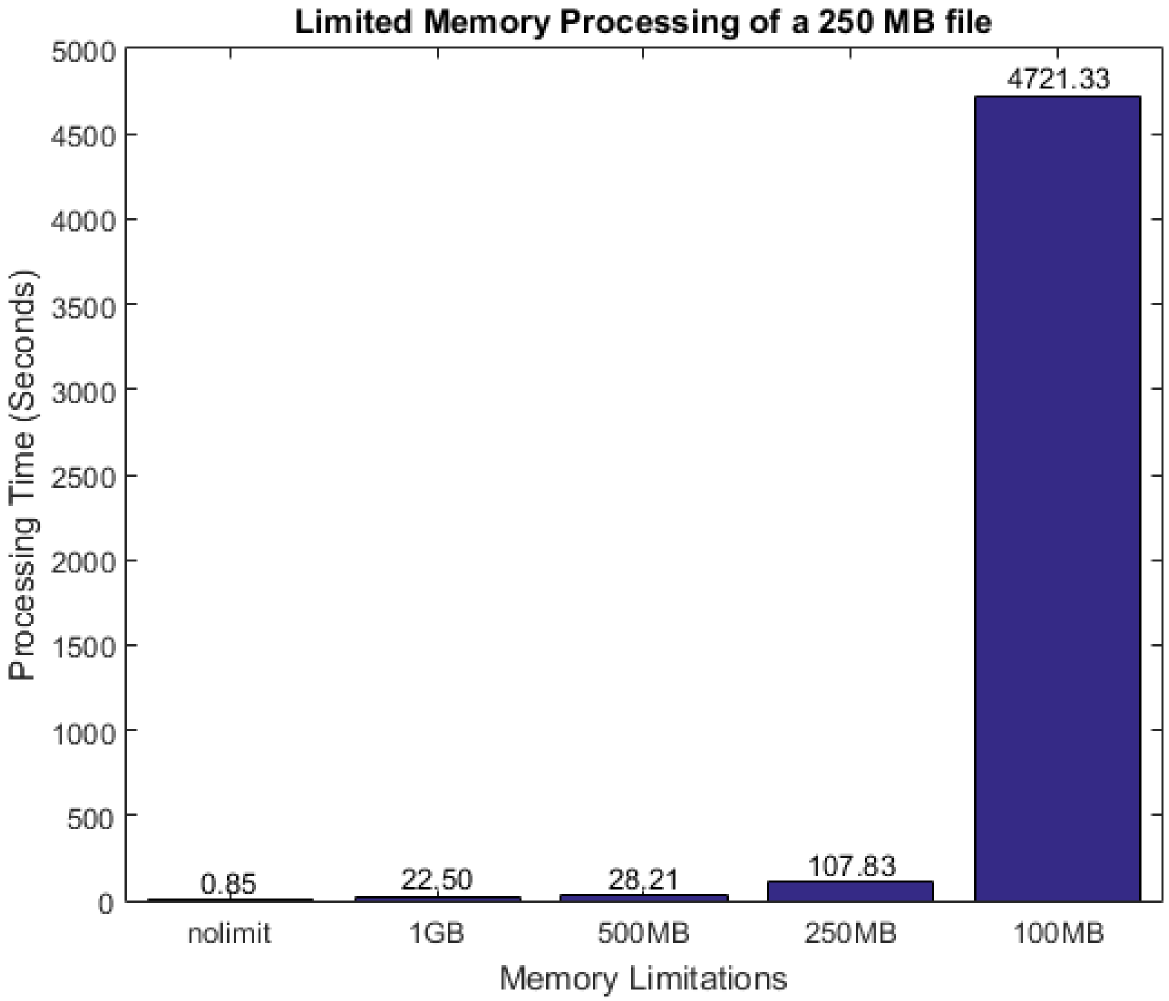


Figure 13: Memory limitation thresholds and resulting processing speed

4.1.5 Hard Disk versus DRAM Processing

The graph below shows that Hard Disk processing is 65 times slower than DRAM processing when utilizing the non-threaded version. As multiprocessing is introduced and threads are scaled up, the Hard Disk processor fails to scale the same way the DRAM processor does. Every time a thread is added, the Hard Disk processor loses 3x in throughput compared to the DRAM processor. This issue with Hard Disk scaling has to do with threads opening/closing and reading/writing to a file system that is not meant for multithreading.

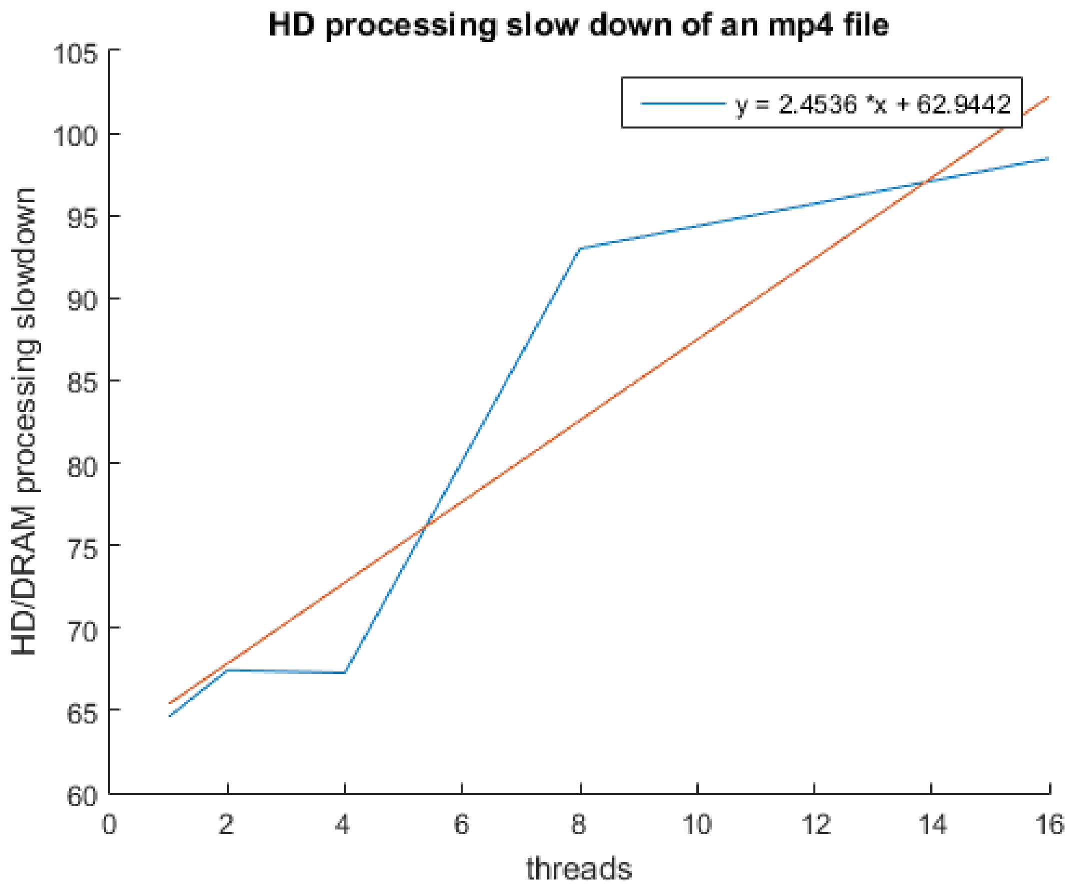


Figure 14: HD processing over DRAM processing throughput

4.1.5 Overlapping vs Non-Overlapping Patterns

Files containing large patterns benefit greatly in speed when utilizing the nonoverlapping search. An mp3 music file containing silence in a segment of the file was processed 180 times faster using the non-overlapping search. A blank png image file was processed 772 times faster. The processing time is greatly reduced while maintaining nearly 100% pattern accuracy. The graph below shows processing time of the two previously mentioned files.

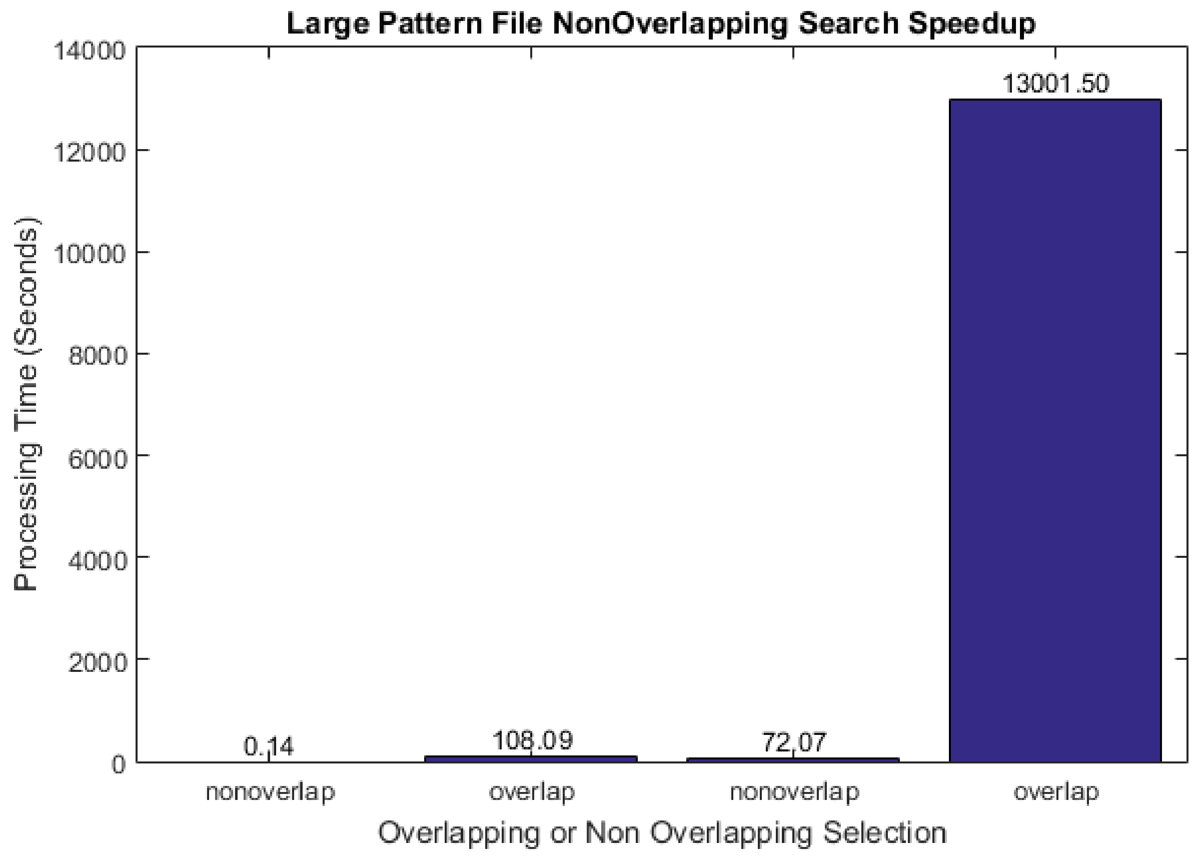
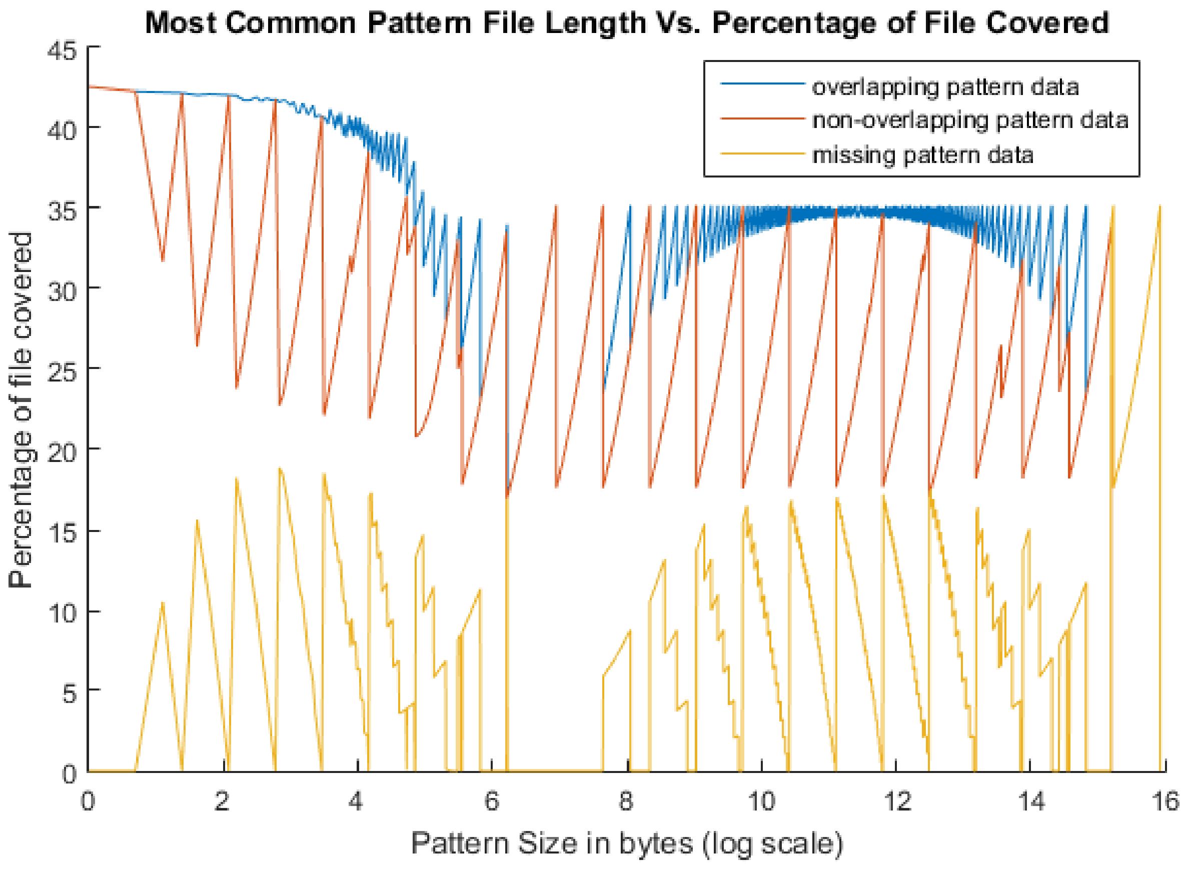


Figure 15: 2 files searched using overlapping and nonoverlapping techniques

4.1.6 Non-Overlapping Coverage

The matlab graph below displays the most common pattern for a certain pattern length vs the percentage that pattern covers the file. The blue graph shows the coverage utilizing the overlapping search method while the orange line displays the non-overlapping pattern coverage. The yellow graph shows the gap in coverage for the non-overlapping search. The next figure below shows the saw tooth drop in coverage when a large pattern has reached its overlapping maxim.



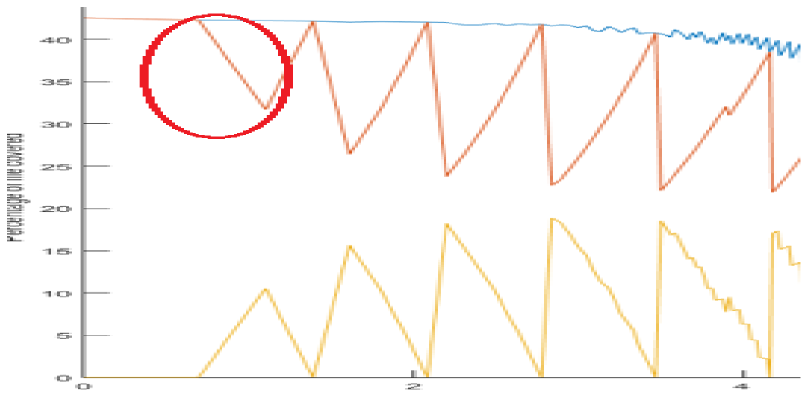


Figure 16: Pattern coverage lines of a overlapped and a nonoverlapped search

This sawtooth gap in coverage can be explained by this diagram below which illustrates that at processing patterns of level three, overlapping patterns get discarded and expose gaps in coverage but the pattern integrity is still intact and observed in the reflected data. The transition from patterns of length 2 to 3 show the coverage dip.

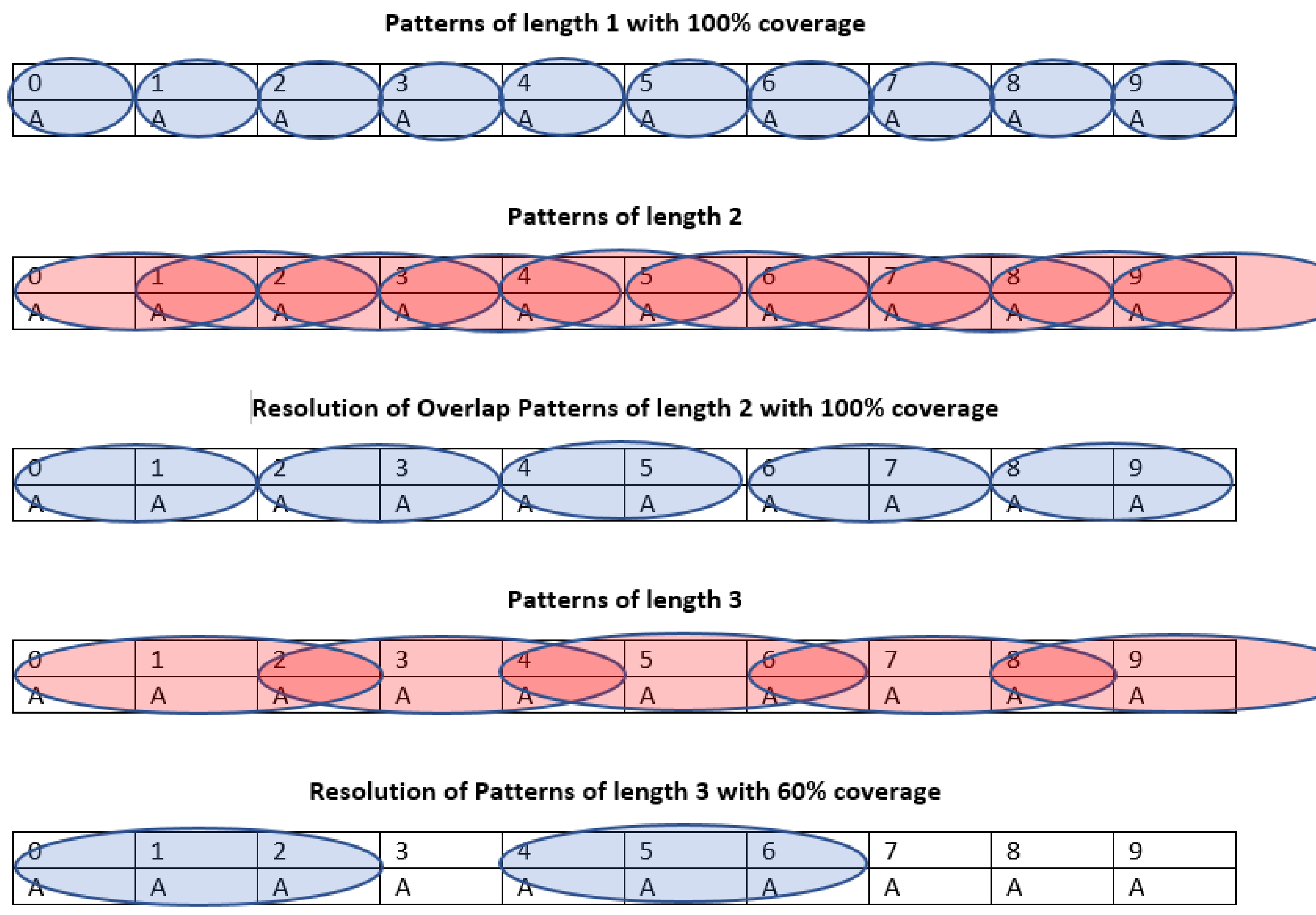

Patterns of length 1 with 100% coverage
Patterns of length 2
Resolution of Overlap Patterns of length 2 with 100% coverage
Patterns of length 3
Resolution of Patterns of length 3 with 60% coverage
0 1 2 3 4 5 6 7 8 9
A A A A A A A A A A

Figure 17: Pattern overlapping example

APPENDIX A

Source code for pattern finding loop

A.1 Pattern Finding Algorithm

```
vector<PListType> newPList[256];
PListType prevPListSize = prevLocalPListArray->size();
PListType indexes[256] = {0};
PListType indexesToPush[256] = {0};
//Banking off very random patterns
PListType firstPatternIndex[256] = {0};
for (PListType i = 0; i < prevPListSize; i++)
{
        vector<PListType>* pList = (*prevLocalPListArray)[i];
        PListType pListLength = (*prevLocalPListArray)[i]->size();
        if(pListLength > 1)
        {
                int listLength = 0;

                for (PListType k = 0; k < pListLength; k++)
                {
                        PListType index = (*pList)[k];

                        if (index < fileSize)
                        {
                                uint_fast8_t indexIntoFile =
(uint8_t)globalStringConstruct[stringIndexer++];

                                if(firstPatternIndex[indexIntoFile])
                                {
                                        if(newPList[indexIntoFile].empty())
                                        {

newPList[indexIntoFile].push_back(firstPatternIndex[indexIntoFile]);
                                        }
                                        newPList[indexIntoFile].push_back(index + 1);
                                        indexes[indexIntoFile]++;
                                }
                                else
                                {
                                        firstPatternIndex[indexIntoFile] = index + 1;
                                        indexes[indexIntoFile]++;
                                        indexesToPush[listLength++] = indexIntoFile;
                                }
                        }
                        else
                        {
                                totalTallyRemovedPatterns++;
                        }
                }
                for (PListType k = 0; k < listLength; k++)
                {
                        int insert = indexes[indexesToPush[k]];
                        if (insert >= minOccurrence)
                        {
                                int index = globalLocalPListArray->size();

                                globalLocalPListArray->push_back(new
vector<PListType>(newPList[indexesToPush[k]]));

                                indexes[indexesToPush[k]] = 0;
                                firstPatternIndex[indexesToPush[k]] = 0;
                                newPList[indexesToPush[k]].clear();
                        }
                        else if(insert == 1)
                        {
                                totalTallyRemovedPatterns++;
                                indexes[indexesToPush[k]] = 0;
                                firstPatternIndex[indexesToPush[k]] = 0;
                                newPList[indexesToPush[k]].clear();
                        }

                }
        }
        else
        {
                totalTallyRemovedPatterns++;
        }
        delete (*prevLocalPListArray)[i];
}
```

APPENDIX B

Pattern Finder Documentation and Usage

PatternFinder is a tool that finds non-overlapping or overlapping patterns in any input sequence.

Pattern Finder Input Parameters:

USAGE:

```
        PatternDetective.exe [
                -help
                /?
                -f       [filename]
                -min     [minimum pattern length]
                -max     [maximum pattern length]
                -c
                -threads [number of threads]
                -mem     [memory limit in MB]
                -ram
                -hd
                -i       [minimum times a pattern has to occur in order to keep track of it]
                -v       [verbosity level]
                -n
                -o
                -his     [hd files]
                -lr      [low range pattern search]
                -hr      [high range pattern search]
                -pnoname
                -plevel  [level to show detailed output]
                -ptop    [n most common patterns indicated by -plevel]
```

Options:

```
        -help                   Displays this help page
        /?                      Displays this help page
        -f [string]             Sets file name to be processed
        -min [unsigned long]    Sets the minimum pattern length to be searched
        -max [unsigned long]    Sets the maximum pattern length to be searched
        -c                      Finds the best threading scheme for computer
        -threads [unsigned int] Sets thread count to be used
        -mem [unsigned long]    Sets the maximum RAM memory that can be used in MB
        -ram                    Forces program to use only RAM
        -hd                     Forces program to use Hard Disk based on -mem
        -i [unsigned long]      Minimum occurrences to consider a pattern (Default occurences will be 2)
        -v [unsigned long]      Verbosity level, turn logging and pattern generation on or off with 1 or 0
        -n                      Only look for patterns that do not overlap each other
        -o                      Only look for patterns that overlap each other
        -his                    HD processing file history keeps or removes files level by level with a 1 or a 0
        -lr     Search for patterns that begin with the value lr to 255 if hr isn't set, otherwise lr to hr range
        -hr     Search for patterns that begin with the value hr to 0 if lr isn't set, otherwise lr to hr range
        -pnoname                Do not print pattern string data
        -plevel                 Sets the level the user wants to see detailed output for
        -ptop                   Display the top N most common patterns in detail for the level indicated by -plevel
```

How to build PatternFinder:
PREREQS:

cmake version 2.5 or higher
c++11 compatible compiler
python 2.7 to run parallel serial jobs
Visual Studio 2012 or 2015 for building with Windows
download repo using https address https://github.com/octopusprime314/PatternDetective.git
or use git and ssh using address git@github.com:octopusprime314/PatternDetective.git

BUILD INSTRUCTIONS:

!!!!!!!!!!!ALWAYS BUILD IN RELEASE UNLESS DEBUGGING CODE!!!!!!!!!!

Linux:

```
    create a build folder at root directory
    cd into build
    cmake -D CMAKE_BUILD_TYPE=Release -G "Unix Makefiles" ..
    cmake --build .
```

Windows:

```
    create a build folder at root directory
    cd into build
    cmake -G "Visual Studio 11 2012 Win64" ..   OR    cmake -G "visual Studio 14 2015 Win64" ..
    cmake --build . --config Release
```

How to run PatternFinder as a standalone executable:

LOCATION OF FILES TO BE PROCESSED:
Place your file to be processed in the Database/Data folder

EXAMPLE USES OF PATTERNFINDER:
1) ./PatternFinder -f Database -v 1 -threads 4 -ram
Pattern searches all files recursively in directory using DRAM with 4 threads

2) ./PatternFinder -f TaleOfTwoCities.txt -v 1 -c -ram
Finds the most optimal thread usage for processing a file

3) ./PatternFinder -f TaleOfTwoCities.txt -v 1
Processes file using memory prediction per level for HD or DRAM processing

4) ./PatternFinder -f TaleOfTwoCities.txt -v 1 -mem 1000
Processes file using memory prediction per level for HD or DRAM processing with a memory constraint of 1 GB

5) ./PatternFinder -f TaleOfTwoCities.txt -min 5 -max 100
Finds patterns of length 5 to 100 and then terminates processing

6) ./PatternFinder -f TaleOfTwoCities.txt -n
Processes file using non overlapping processing

7) ./PatternFinder -f TaleOfTwoCities.txt -v 1 -hd
Processes file using the hard disk only.

8) ./PatternFinder -f TaleOfTwoCities.txt -i 10
Processes patterns that occur at least 10 times or more. Default is 2.

How to run PatternFinder Python Scripts:

PYTHON RUN EXAMPLES:

1) python splitFileForProcessing.py [file path] [number of chunks]
Use splitFileForProcessing.py Python script to split files into chunks and run multiple instances of PatternFinder on those chunks

Ex. python splitFileForProcessing.py ~/Github/PatternDetective/Database/Data/TaleOfTwoCities.txt 4

equally splits up TaleOfTwoCities.txt into 4 files and 4 instances of PatternFinder get dispatched each processing one of the split up files.

2) python segmentRootProcessing.py [file path] [number of jobs] [threads per job]
Use segmentRootProcessing.py Python script splits up PatternFinder jobs to search for patterns starting with a certain value

Ex. python segmentedRootProcessing.py ../Database/Data/Boosh.avi 4 4

Dispatches 4 processes equipped with 4 threads each. Each PatternFinder will only look for patterns starting with the byte
representation of 0-63, 64-127, 128-191, 192-255.

PatternFinder Input Files:
PatternFinder accepts any type of input file because it processes at the byte level.
PatternFinder Output Files:
Nine outputs are available. One is a general logger using ascii text format, another is the Output file which generates patterns based on -pnoname, -plevel, ptop and the remaining seven are Comma Separated Variable files used for post processing in MATLAB.

1) Logger file: records general information including the most common patterns, number of time a pattern occurs and the pattern's coverage at every level until the last pattern is found. Simple text file.
2) Output file: generates patterns based on -pnoname, -plevel and -ptop
3) Collective Pattern Data file: records each level's most common pattern and number of times the pattern occurs in CSV format.
4) File Processing Time: records each file's processing time in CSV format. Used for processing large data sets with many files.
5) File Coverage: records the most common pattern's coverage of the file in CSV format.
6) File Size Processing Time file: records each file's processing time and corresponding size in CSV format. Used primarily to isolate files in a large dataset that contain large patterns.
7) Thread Throughput: records the processing throughput improvement while incrementing the number of processing threads in CSV format. Typically used with -c option which tests threads in multiples of 2 starting at 1 until the number of cores on the machine has been met.
8) Thread Speed: records the processing time taken while incrementing the number of processing threads in CSV format. Typically used with -c option which tests threads in multiples of 2 starting at 1 until the number of cores on the machine has been met.

Output file contents is pattern string, number of instances, occurrence, average distance and location:

```
./PatternFinder.exe -f TaleOfTwoCities.txt -plevel 2 -ptop 4 -threads 1 -ram
```

```
Level 1
unique patterns = 83, average occurrence frequency = 9490.39, frequency of top pattern: 129157
Level 2
unique patterns = 1401, average occurrence frequency = 562.024, frequency of top pattern: 21032
1. pattern = e , instances = 21032, coverage = 5.34006%, average pattern distance = 37.4536, first occurrence index = 4
2. pattern =  t, instances = 18017, coverage = 4.57454%, average pattern distance = 43.7134, first occurrence index = 75
3. pattern = he, instances = 16814, coverage = 4.2691%, average pattern distance = 46.8478, first occurrence index = 3
4. pattern = th, instances = 16713, coverage = 4.24346%, average pattern distance = 47.1234, first occurrence index = 91
Level 3
unique patterns = 7934, average occurrence frequency = 98.9629, frequency of top pattern: 12204
Level 4
unique patterns = 26273, average occurrence frequency = 29.5211, frequency of top pattern: 8907
Level 5
unique patterns = 57380, average occurrence frequency = 13.0067, frequency of top pattern: 6427
Level 6
unique patterns = 86748, average occurrence frequency = 7.87542, frequency of top pattern: 2153
Level 7
unique patterns = 103739, average occurrence frequency = 5.72812, frequency of top pattern: 974
Level 8
unique patterns = 108726, average occurrence frequency = 4.53085, frequency of top pattern: 780
Level 9
unique patterns = 102229, average occurrence frequency = 3.80874, frequency of top pattern: 339
Level 10
unique patterns = 88663, average occurrence frequency = 3.34268, frequency of top pattern: 276
```

PatternFinder post processing scripts using the seven available CSV outputs with MATLAB:

1) DRAM versus HD Processing Speeds->DRAMtoHDProcessingLiminationSpeeds.m
2) DRAM versus HD Performance->DRAMVsHardDiskPerformance.m
3) Most Common Pattern versus Coverage->MostCommonPatternLengthVsCoveragePercentage.m
4) Overlapping versus Non Overlapping Comparison->Overlapping_NonOverlappingComparison.m
5) Overlapping versus Non Overlapping File Speeds->OverlappingVsNonOverlappingFileSpeeds.m
6) Process Time versus File Size->ProcessTimeVsFileSize.m

Bibliography

[1] Matthis Kretz, " Efficient Use of Multi- and Many-Core Systems with Vectorization and Multithreading ", Diploma Thesis in Physics, University of Heidelberg.

[2] " Ladner, Fix and LaMarca, "Cache Performance Analysis of Traversals and Random Accesses", Tenth Annual ACM-SIAM.

[3] http://locklessinc.com/benchmarks_allocator.shtml - used for benchmarks with memory allocators versus threads (accessed April 4th 2015).

[4] https://en.wikipedia.org/wiki/Amdahl%27s_law – used to understand Amdahl's law and how it applies different to program that have certain levels of parallelism (accessed March 3rd 2016).

[5] https://www.threadingbuildingblocks.org/ - Intel's version of memory allocation (accessed April 3rd 2015).

[6] http://goog-perftools.sourceforge.net/doc/tcmalloc.html - Google's version of memory allocation called TcMalloc (accessed April 4th 2015).

[7] https://www.safaribooksonline.com/library/view/linux-system-programming/0596009585/ch04s03.html - Memory mapping files used to quickly read and write to the hard drive (accessed January 3rd 2016).

[8] https://jpassing.com/2008/09/01/effective-leak-detection-with-the-debug-crt-and-application-verifier/ - Advanced memory leak detection method which is critical for memory intensive programs (accessed November 10th 2015).

[9] http://info.prelert.com/blog/stl-container-memory-usage - Memory approximations of C++ STL containers (accessed August 8th 2016).

[10] https://msdn.microsoft.com/en-us/library/ms182372.aspx - Visual Studio performance analyzer which graphically displays program bottlenecks, etc (accessed July 24th 2016).

[11] http://www.phoronix.com/scan.php?page=article&item=gcc_49_optimizations&num=1 – Linux binary optimization for faster running Linux programs (accessed May 26th 2016).

[12] https://www3.ntu.edu.sg/home/ehchua/programming/cpp/gcc_make.html - Linux compilation using code built for Windows (accessed February 12th 2016).

[13] http://technopark02.blogspot.com/2004/09/solaris-malloc-vs-mtmalloc.html - MtMalloc memory allocator information (accessed April 5th 2015).

[14] http://www.canonware.com/jemalloc/ - JeMalloc memory allocator information (accessed April 5th 2015).

[15] http://www.hoard.org/ - Hoard memory allocation information (accessed April 6th 2015).

[16] Celal Ozturk, Ibrahim Burak Karsli, Resit Sendag, "An analysis of address and branch patterns with PatternFinder", 2014 IEEE International Symposium on Workload Characterization (IISWC), pp. 232-242

[17] Roman Dementiev, Juha Karkkainen, Jens Mehnert, Peter Sanders, "Better External Memory Suffix Array Construction", Journal of the ACM, Vol. V, pp. 1-24