MULTI THREADED PATTERN SEARCHING OF LARGE FILES USING LIMITED MEMORY

Submission by: Peter Morley

Graduate Professor: Resit Sendag Ph.D.

Pattern Discovery Approach

- Discover patterns using the suffix tree algorithm
- A single threaded pattern finding program exists but pattern searching speeds could be greatly improved by implementing a multithreaded algorithm
- Files that are larger than a computer’s DRAM capacity must be partially processed using the Hard Disk otherwise processing can not be accomplished.

Previous work on Pattern Discovery

- The “An Analysis of Address and Branch Patterns with PatternFinder” paper implemented a single threaded, branch and address specific pattern discovery tool.
  - Specific to branch and address discovery, not generic
  - Slow because it is single threaded
- The “Better External Memory Suffix Array Construction” paper discussed an implementation of the Suffix Tree algorithm when DRAM alone is not capable of processing large files.
  - Many unique algorithms but no mention of HD processing speed improvement techniques
  - Pattern Finder’s memory mapping solution improves HD read/write time

Suffix Tree Algorithm

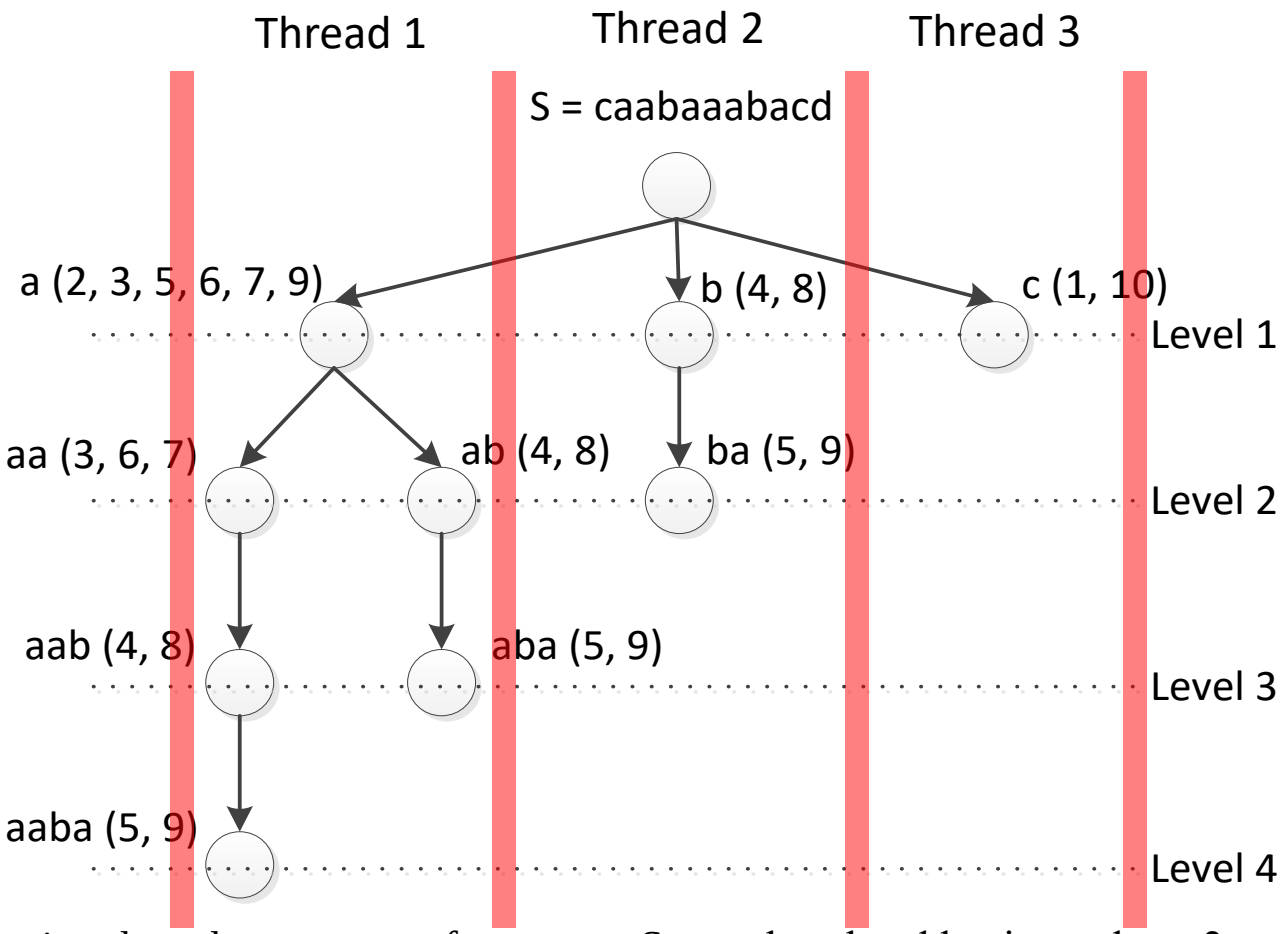
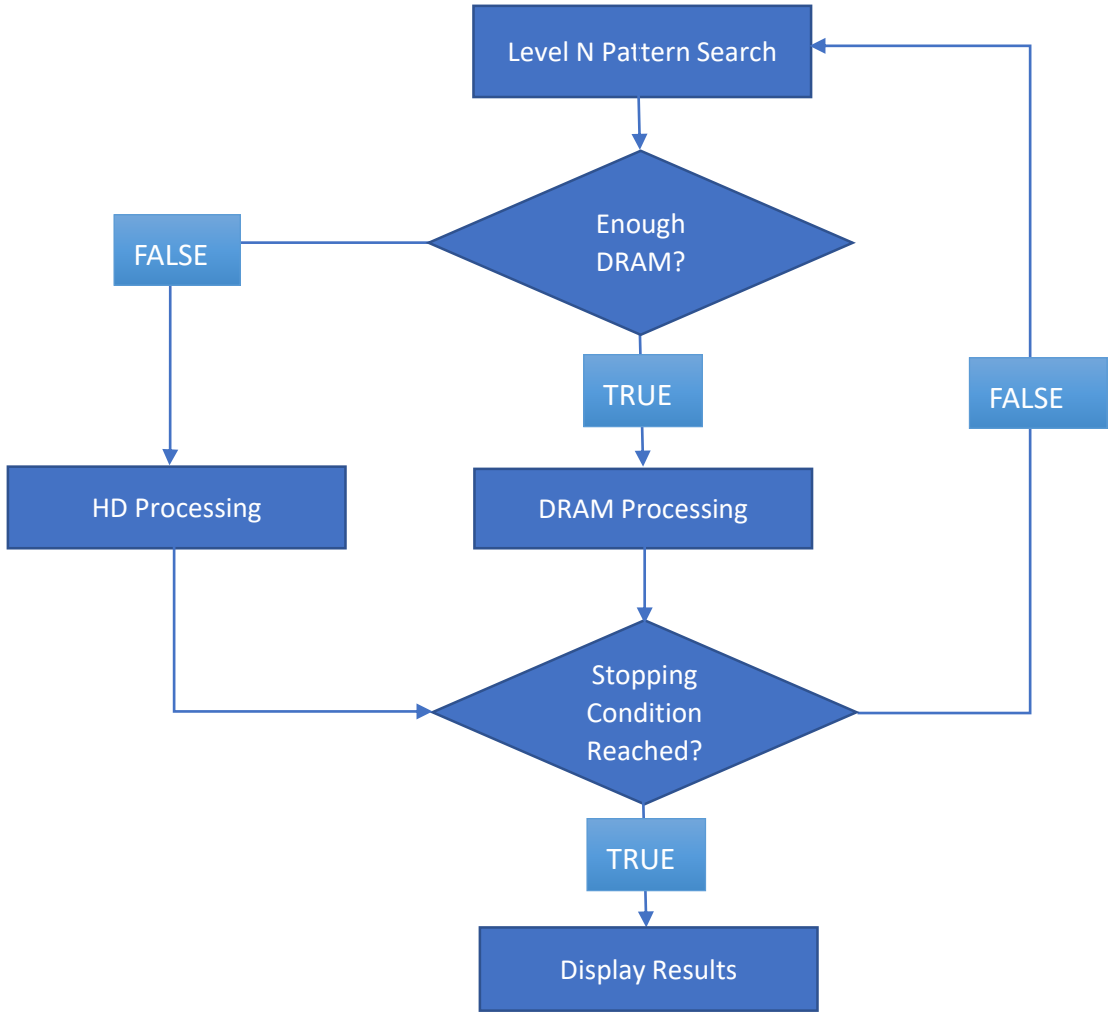


Figure 1: Discovering the subsequences of sequence S→caabaaabacd having at least 2 occurrences in S.

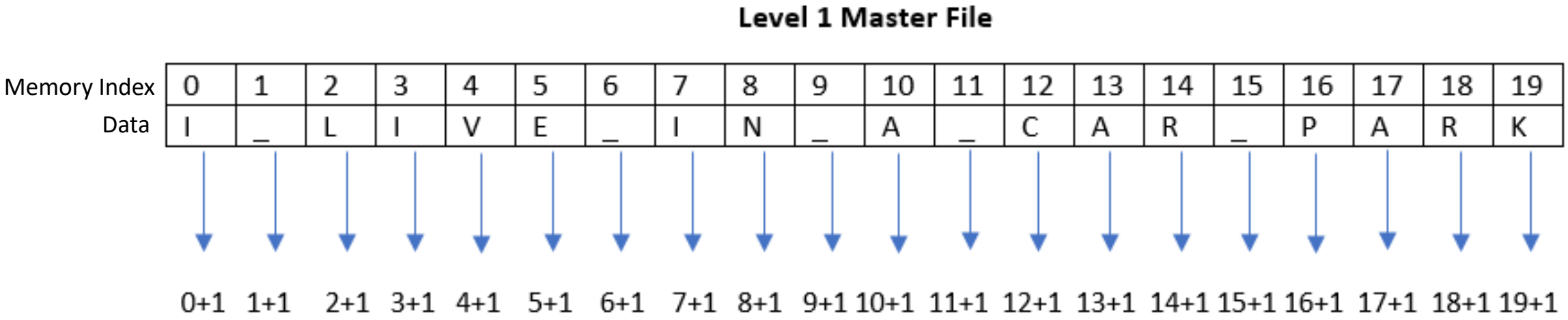
Pattern Finder Design and Methods

- Core Components of Pattern Finder
  - HD (Hard Disk) Processing
  - DRAM (Dynamic Random Access Memory) Processing
- Pattern Finding using Distributed Computing
- Performance Analysis of Pattern Finder
  - Overlapping vs Non-overlapping Search
  - Multiprocessor scalability
  - HD vs DRAM processing speed comparison

High Level Processing Flow

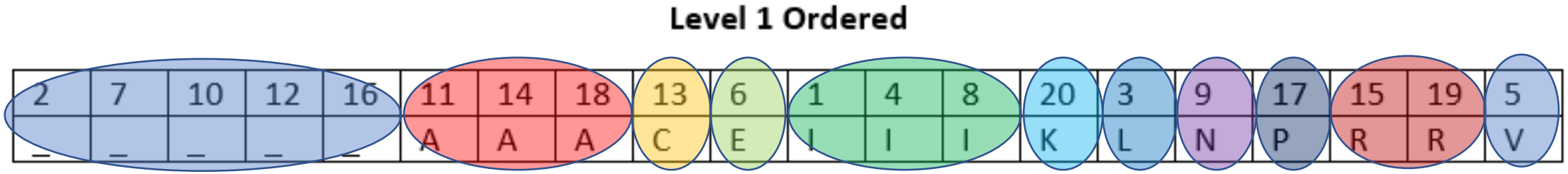


Level Node Processing



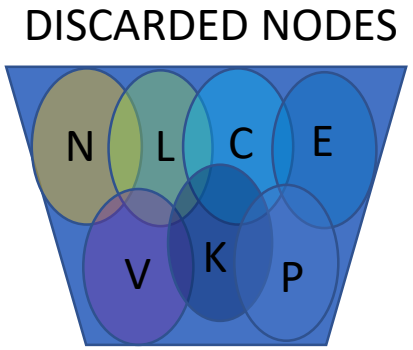
- A potential pattern candidate only requires the endpoint position of the pattern sequence
- A pattern can be recovered knowing the level (length) and the last index in the master file

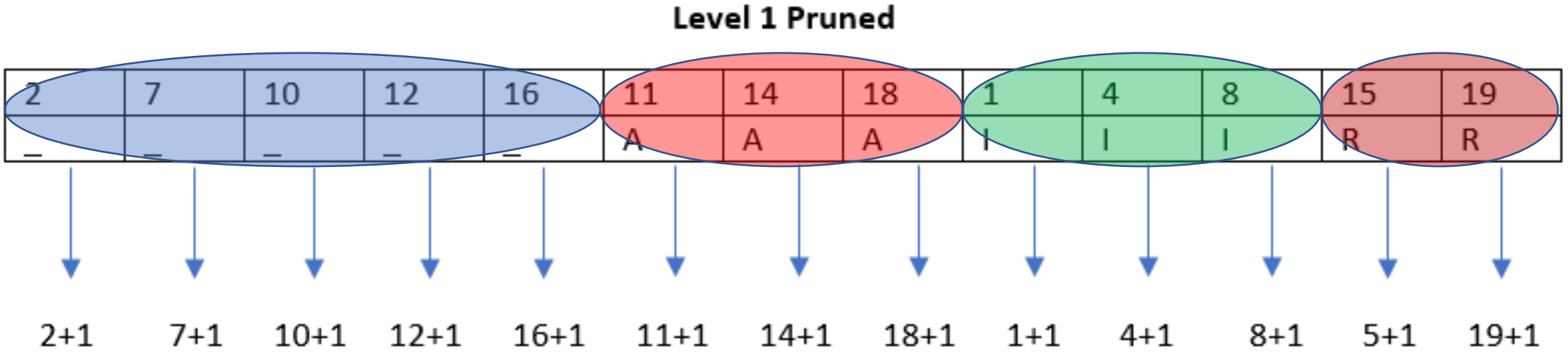
Level Node Ordering



- Each colored vector represents a node in the tree
- Pattern data is ordered based upon ascii mapping 0-255
  - For example A and C in ascii respectively have the values of 41 and 43 and are ordered as such in the array
- Colored circles represent each level node

Level Node Pruning

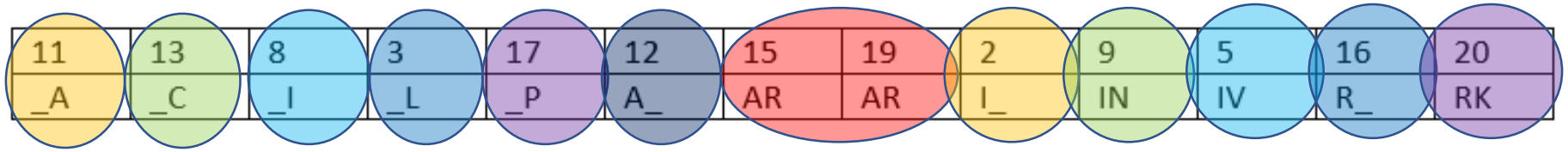




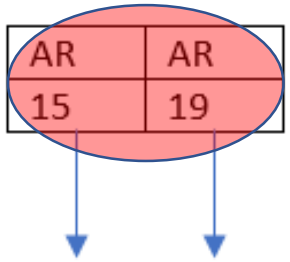
- If a pattern occurs only once, it is pruned
- The next level of pattern searching can be started
- The previous level pattern data can be discarded after the new level data is generated, but for the first level there is no previous data

Level Searching Resolution

Level 2 Ordered

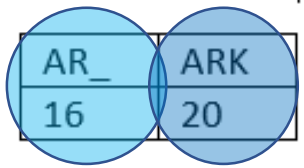


Level 2 Pruned



15+1 19+1

Level 3 Ordered



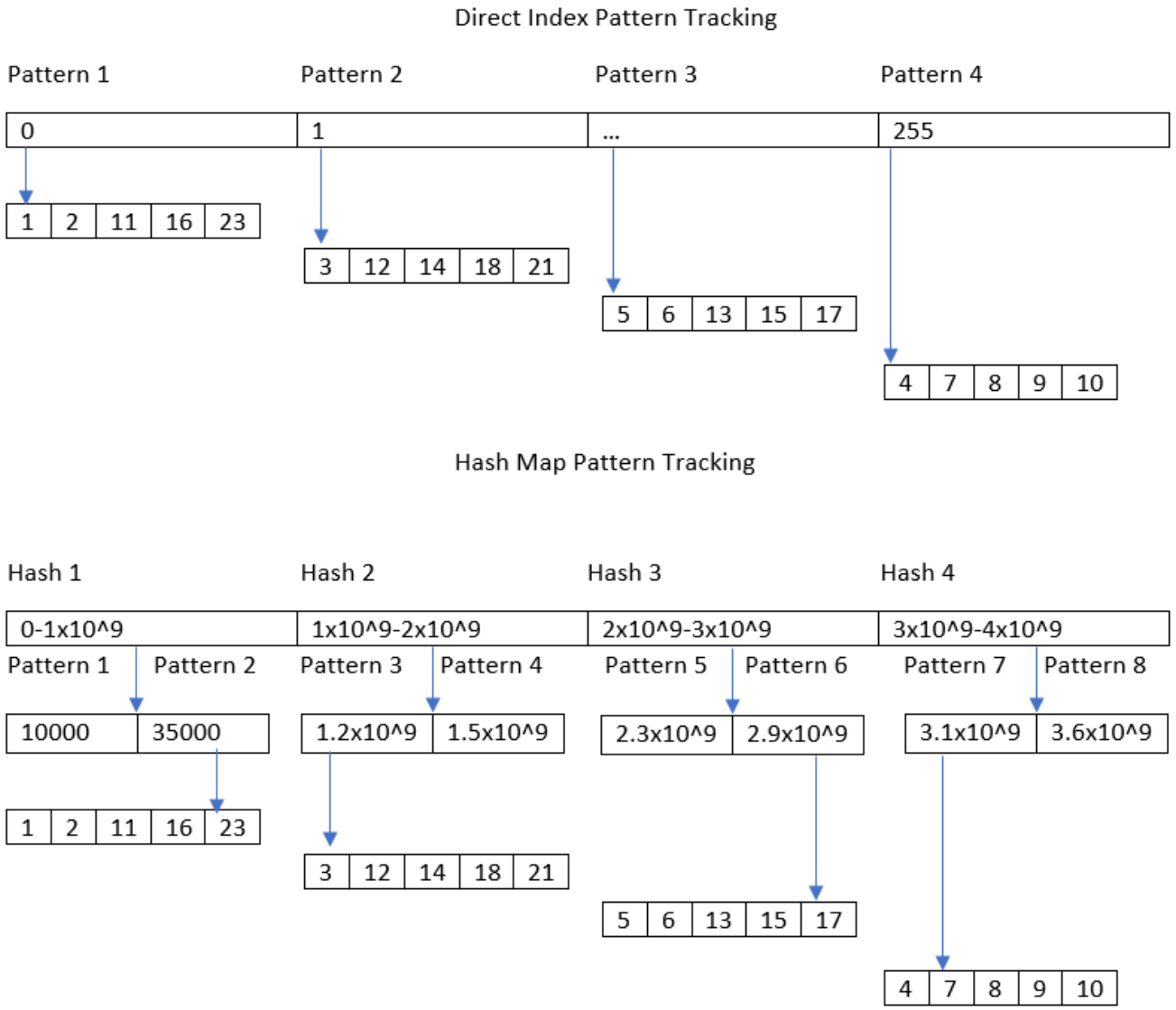
Level 3 Pruned

NO MORE PATTERNS!

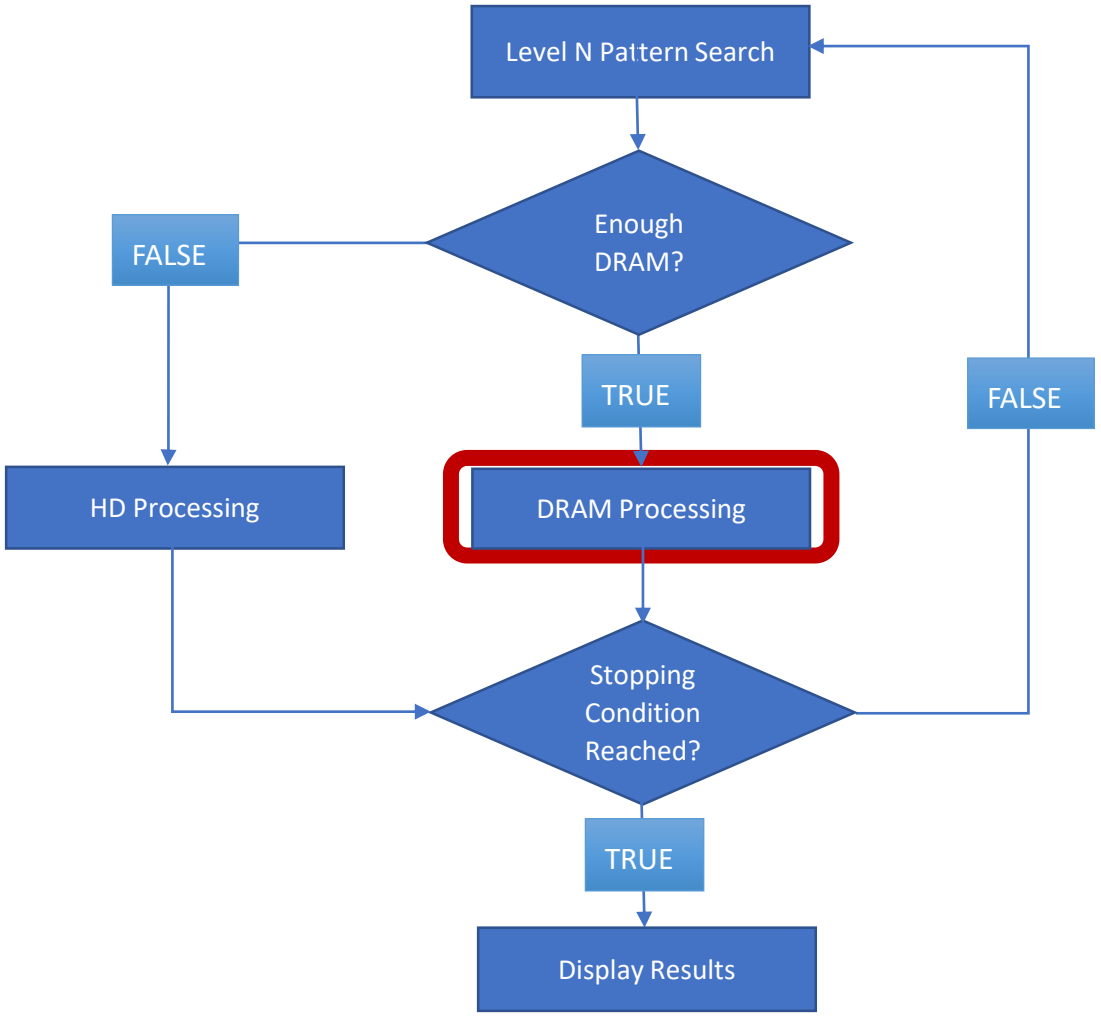
Implementation Improvement

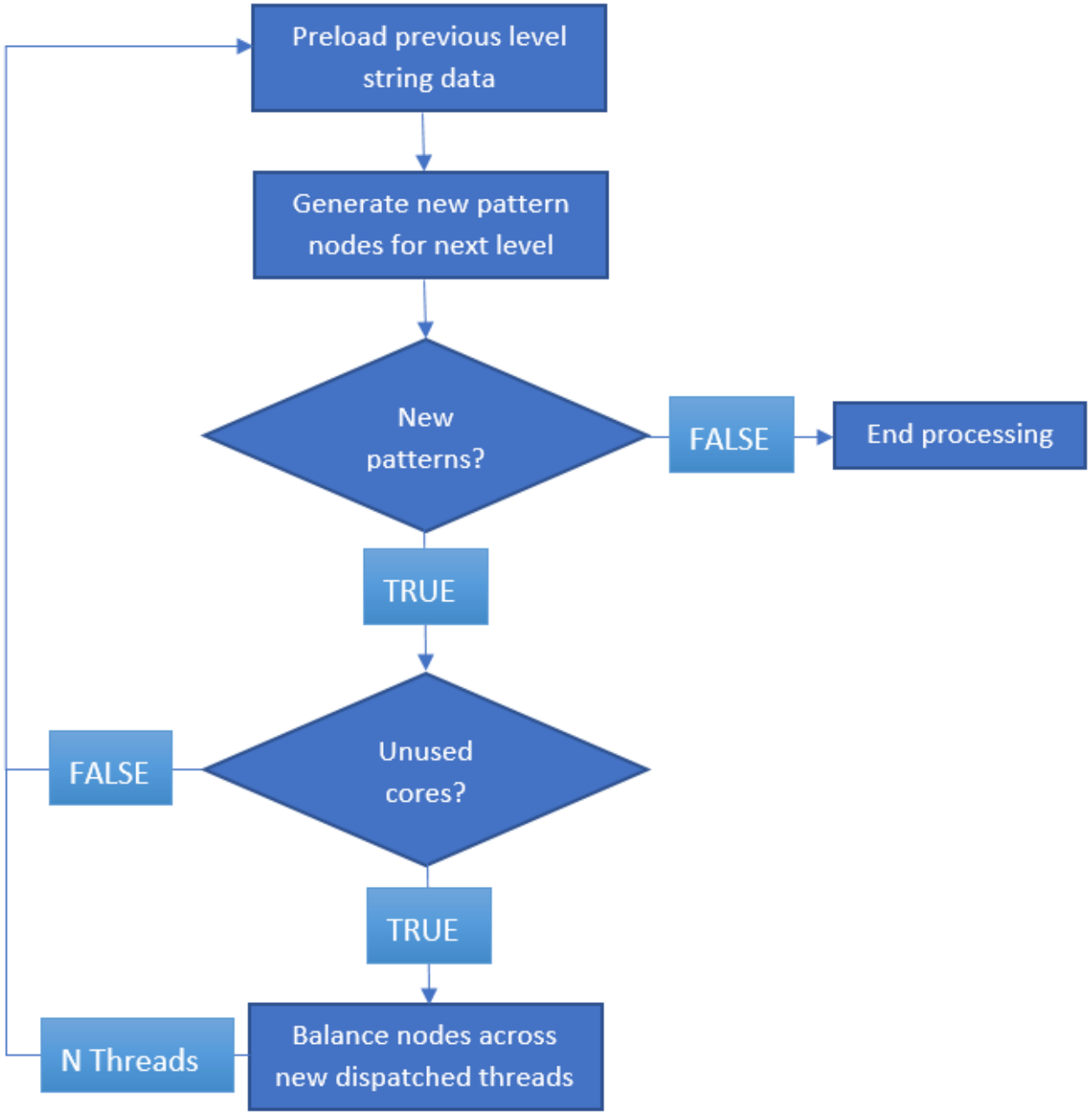
- Previous Pattern Finder algorithm organized next level patterns using a hash map because the integer patterns generated could range between values of 0 - 4294967296.
- New Pattern Finder searches at the byte granularity thus new patterns could be directly mapped to a vector of size 256 that would not require any hash map searches for pattern categorization.
- Searching through large hash maps is very time consuming and by eliminating the need to do map searches, the new implementation runs much faster.

Direct Indexing Versus Mapping Patterns



High Level Processing Flow

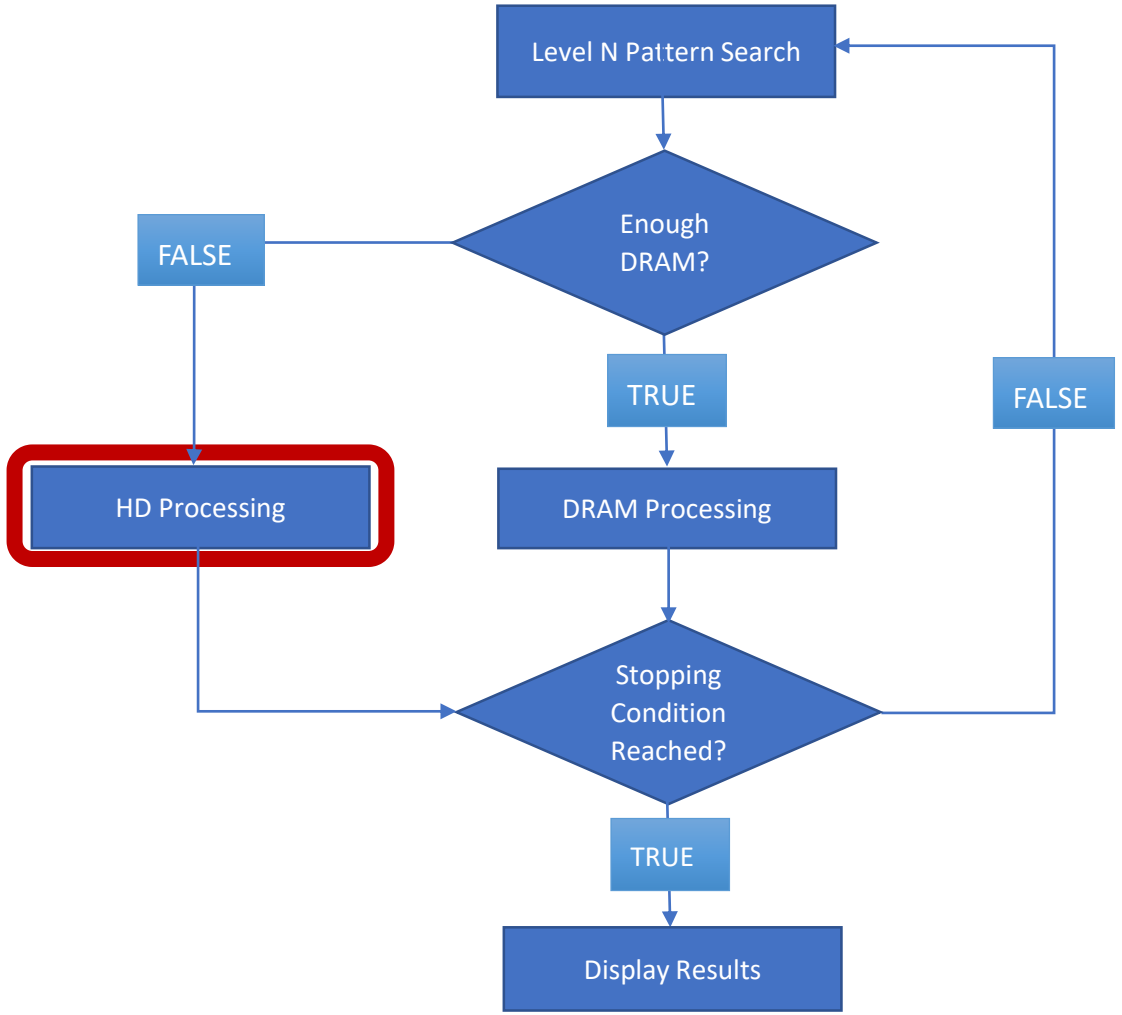


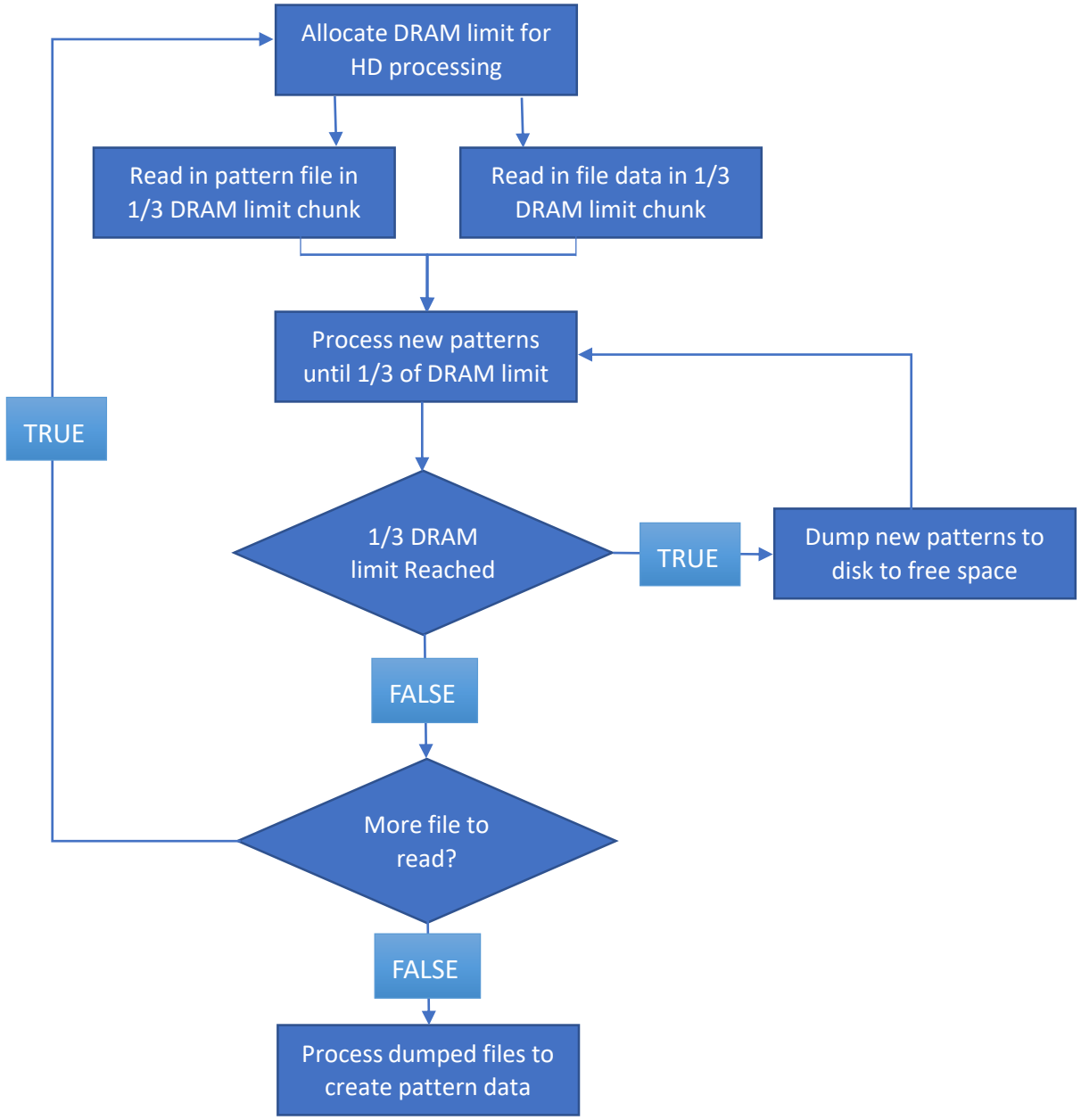


DRAM Processing

- Stream of reads improves cache locality
- Process preloaded string data and generate new pattern nodes
- If new patterns are generated continue processing
- If unused cores are available then dispatch processing threads recursively which implements thread pooling
- Nodes are distributed evenly across newly spawned worker threads

High Level Processing Flow



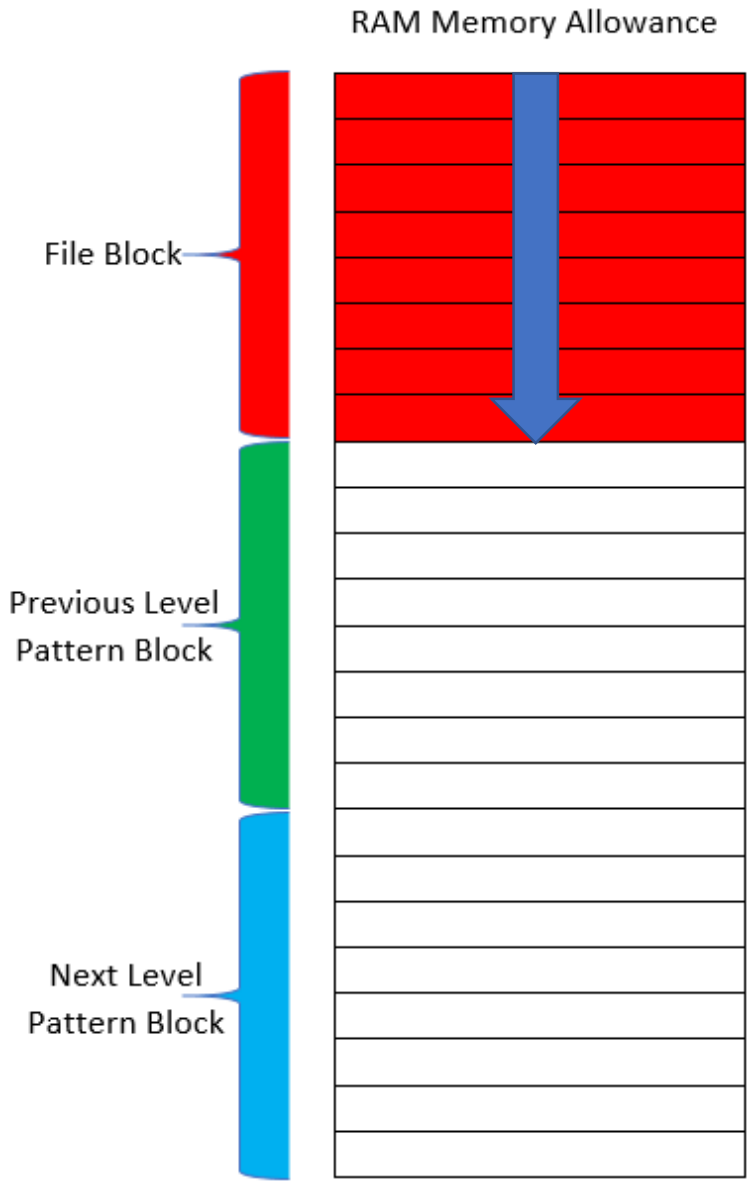


HD Processing

- Allocates three partitions of DRAM space
- The first partition is used to load in a slice of the previous pattern data
- The second partition is used to load in a chunk of the file being processed
- The third partition acts as available memory for new patterns to be created
- When the available memory is completely used it must offload the data to the HD

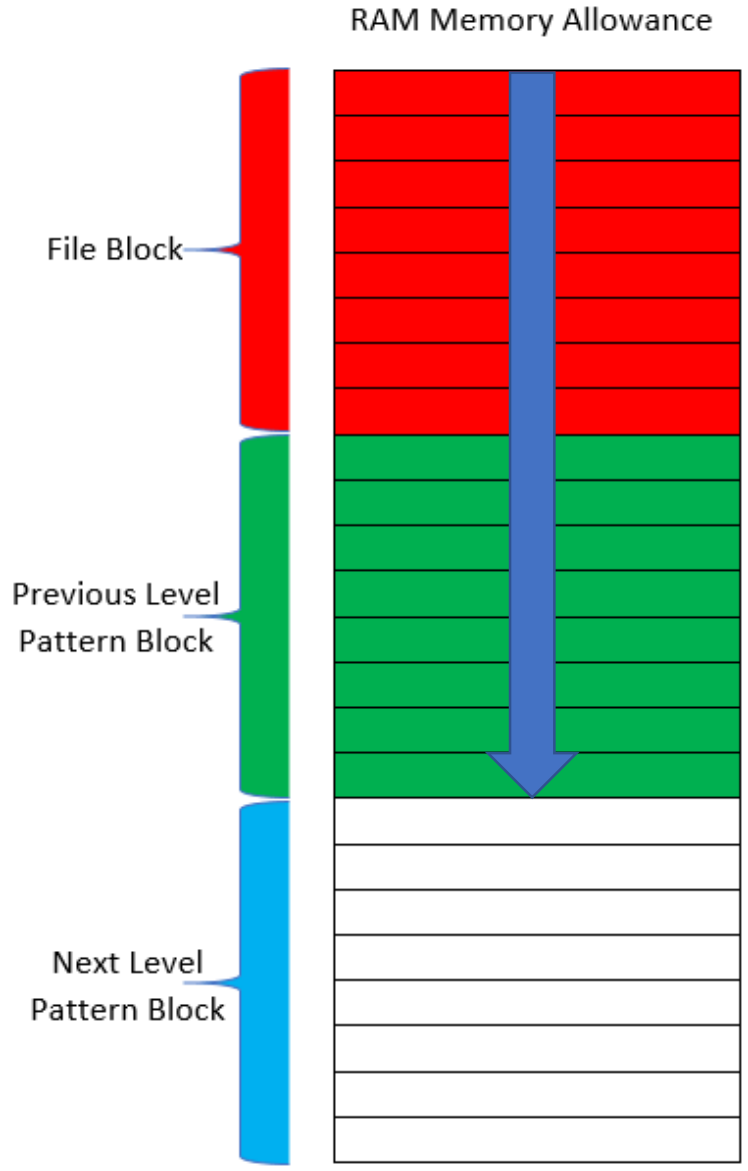
File Block Handling

- The red block of memory indicates the loading of file data into DRAM
- Once that allocation size has been reached, the program will not continue to pull the entire file up to DRAM



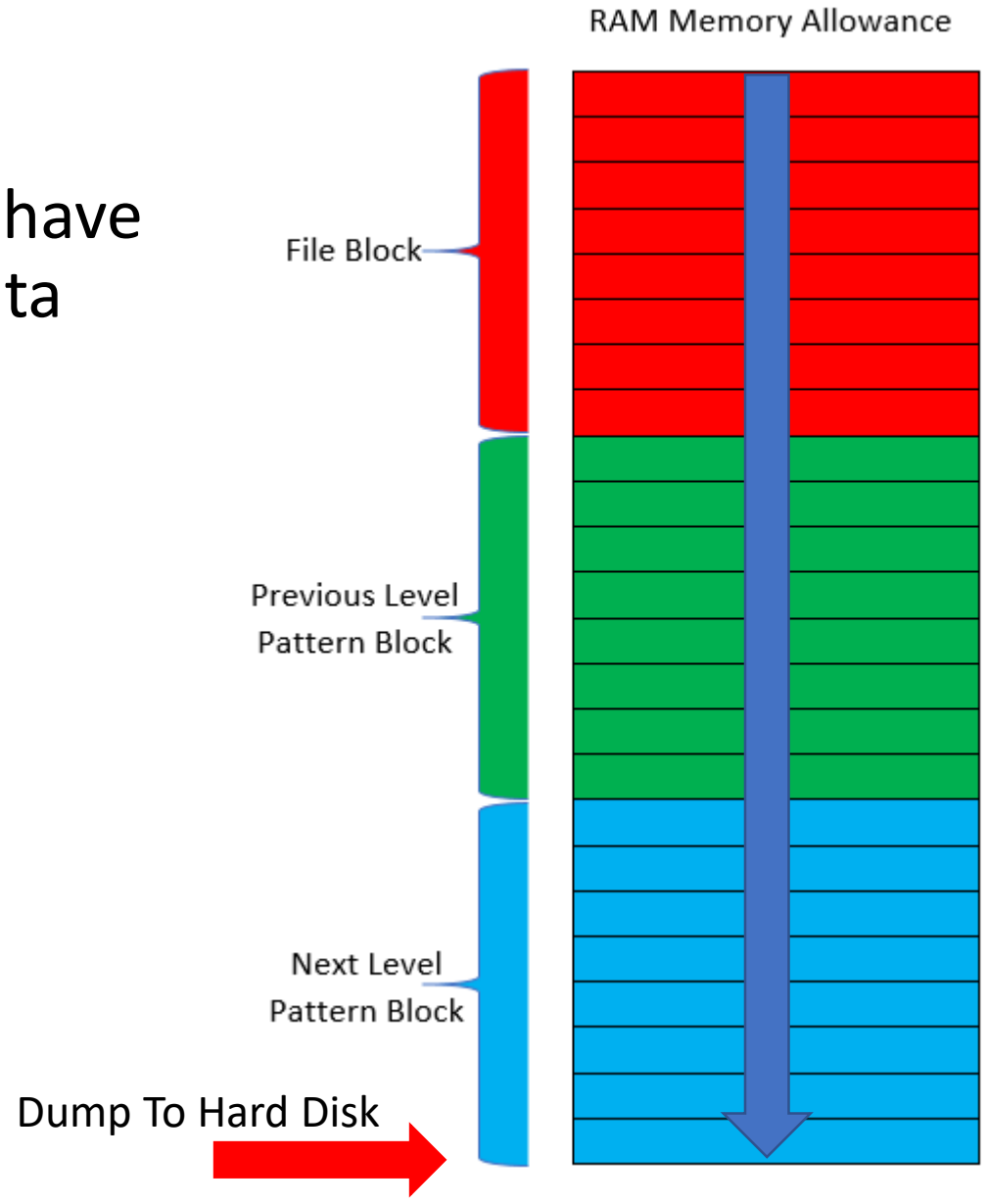
Previous Level Pattern Block Handling

- The green block indicates memory allocation of the previous node level data
- If a file containing previous node data is larger than the DRAM allotment then the program will discontinue loading data
- The last loaded node must have the entire pattern index sequence so there typically occurs a negligible overflow
- Not possible to load in segments of a pattern node's index sequence



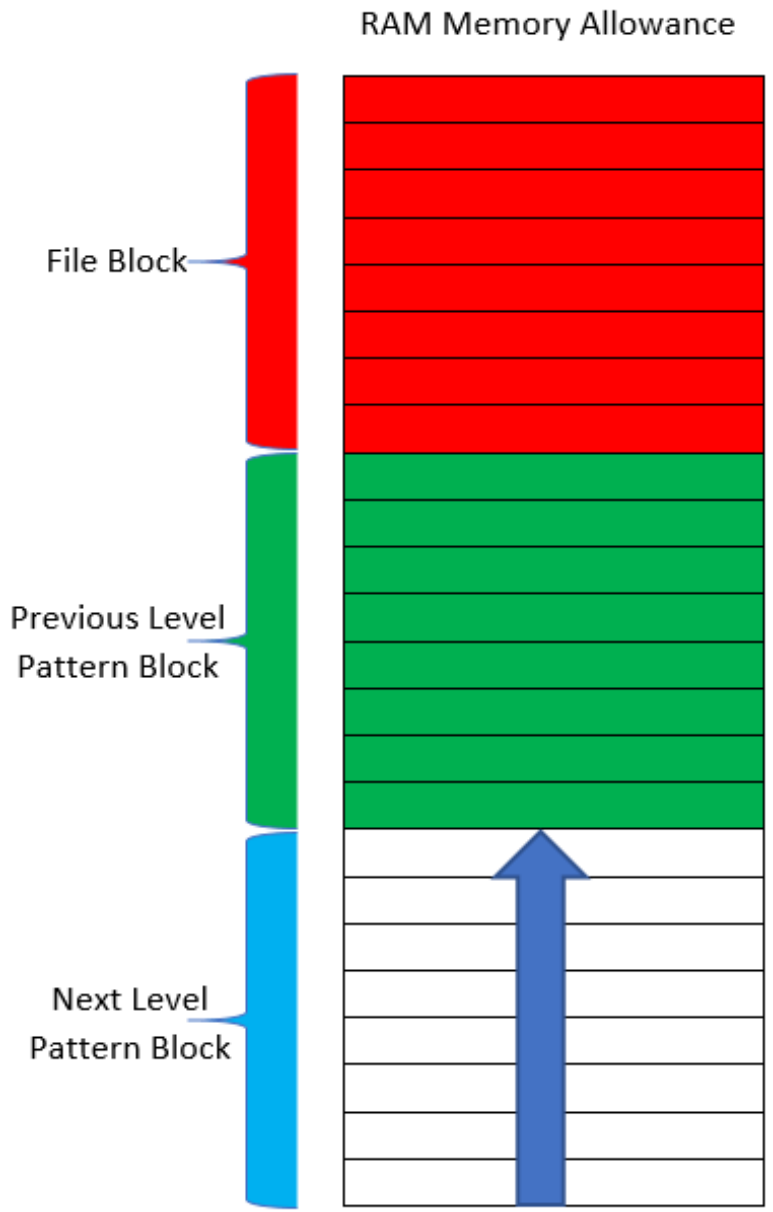
Next Level Pattern Block Handling

- When the next level pattern allocations have exceeded the memory allowance the data must be pushed to the hard drive
- Writing the data to the hard drive while deallocating DRAM alleviates space for continued processing

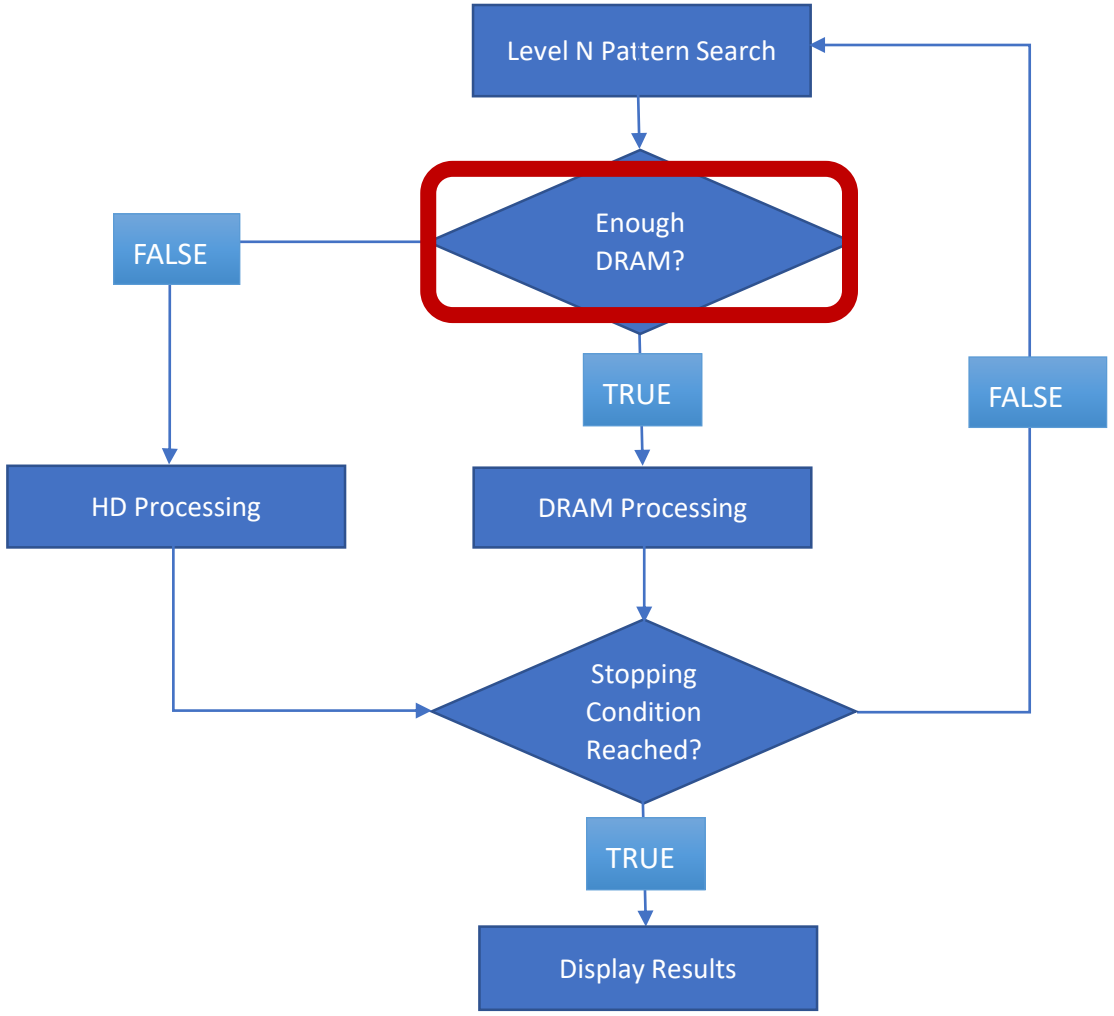


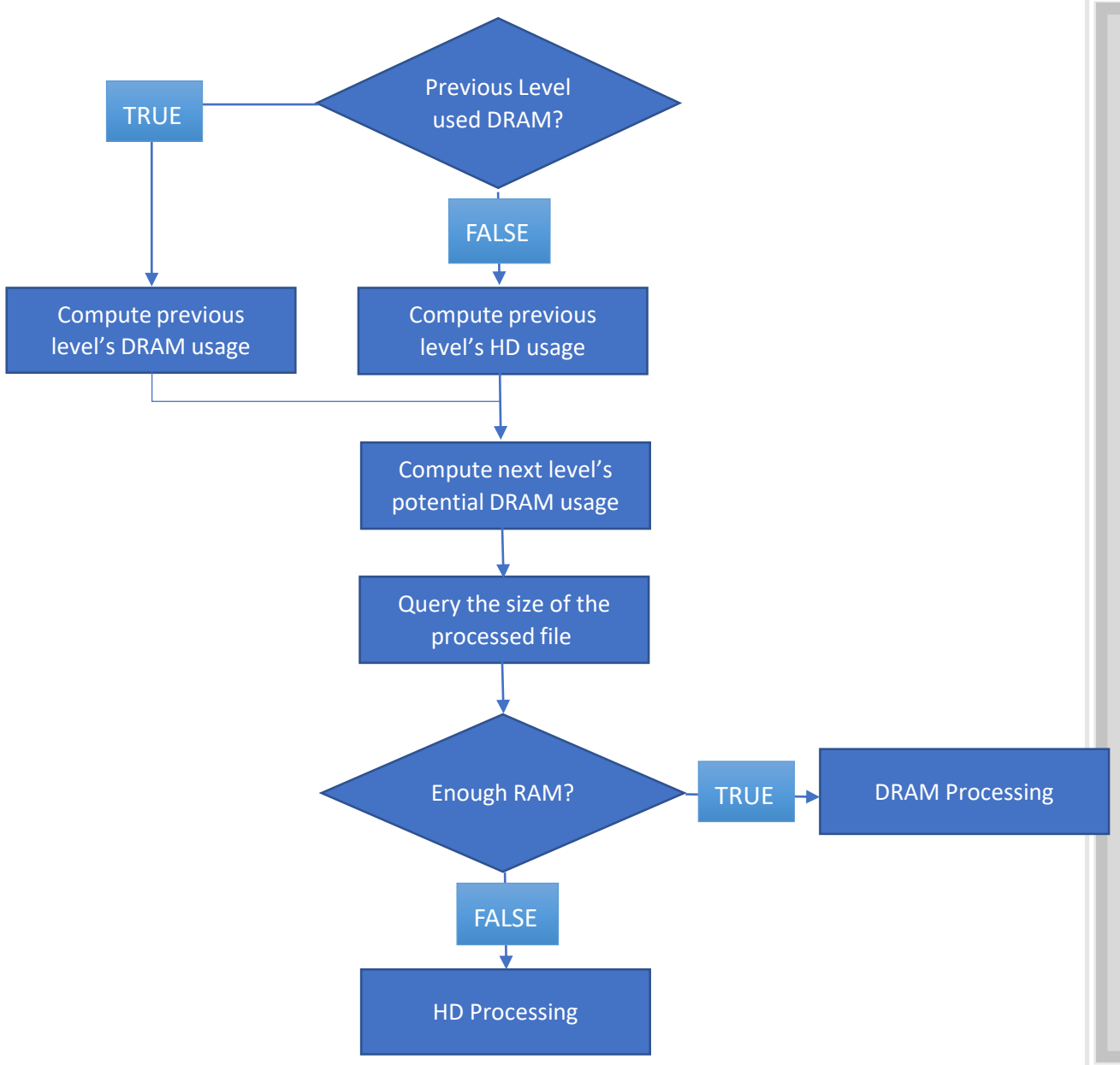
Offloading Node Data to Hard Disk

- Pattern data is offloaded to the hard disk for later retrieval
- DRAM space is again open for creating new pattern nodes



High Level Processing Flow





DRAM or HD Processing Decision

- Has the previous level used DRAM or HD? Transitioning between processing in DRAM to HD or vice versa is very time consuming.
- Compute previous level's memory consumption
- Figure out potential patterns generated based on previous pattern data
- Query the size of the file being processed
- After all memory allocations have been added then the decision is made to process with DRAM or DRAM/HD.

Distributed Computing using Python

- Segment Processing
  - 100% accurate, slower but takes N times more memory
  - Dispatches N Pattern Finder programs equipped with T threads each to detect patterns that start with a specific byte value. N <= 256
  - Ex. python segmentedRootProcessing.py ../Database/Data/Boosh.avi N T
    - Dispatches N processes equipped with T threads each

- Split Processing
  - Less than 100% accurate but much faster
  - Splits a file into portions and dispatches N single threaded Pattern Finder programs to process the portions individually
  - Ex. python splitFileForProcessing.py ../Database/Data/Boosh.avi N

Segment Processing Job Distribution

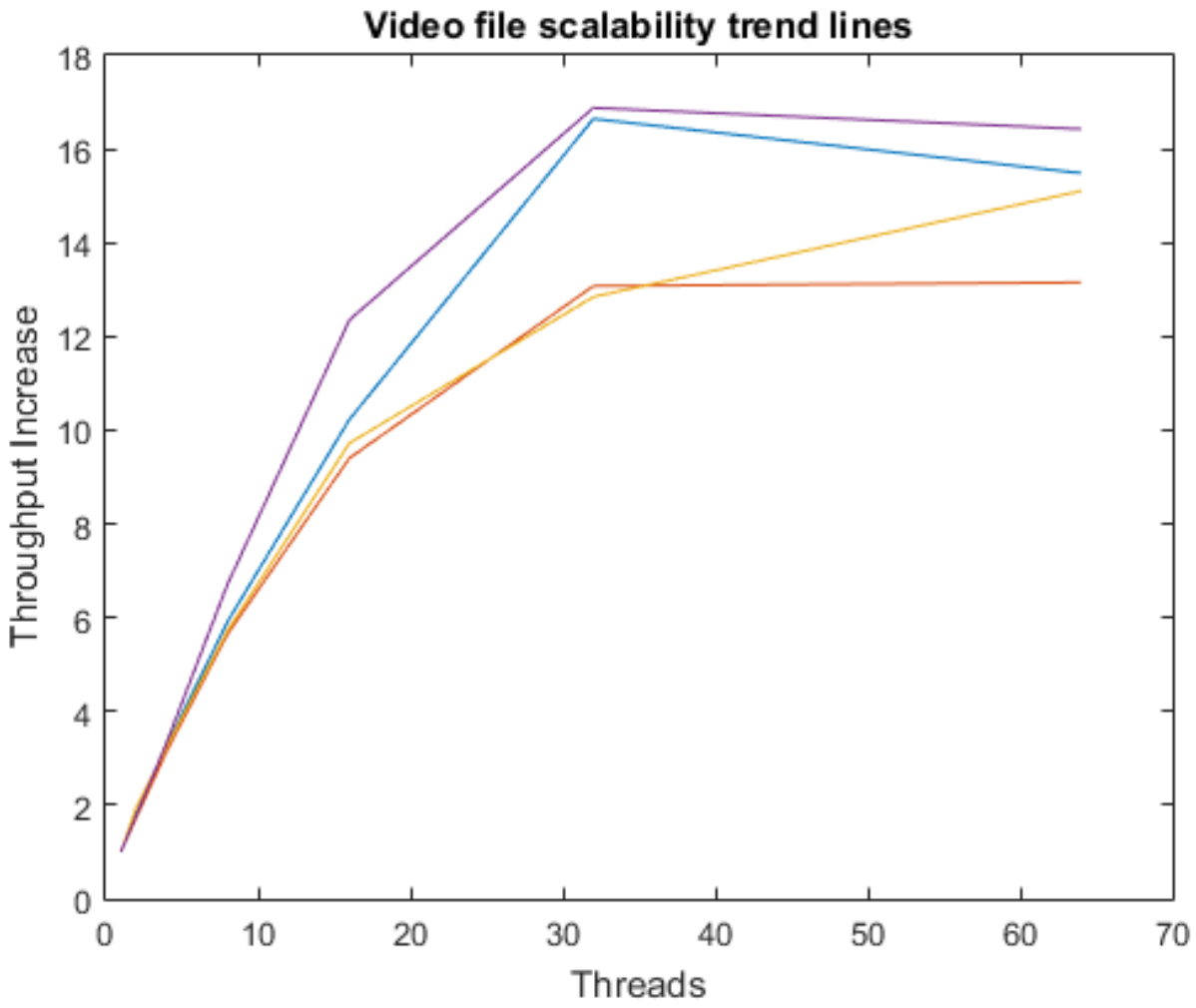
- Job distribution is key to getting better throughput
- For example a segment job dispatched 8 Pattern Finders each equipped with 4 processing threads which equates to 32 threads of total processing power
- Results
  - 1.5x speedup compared to single Pattern Finder process with 32 threads
  - 4 threads per job is the sweet spot for processor distribution

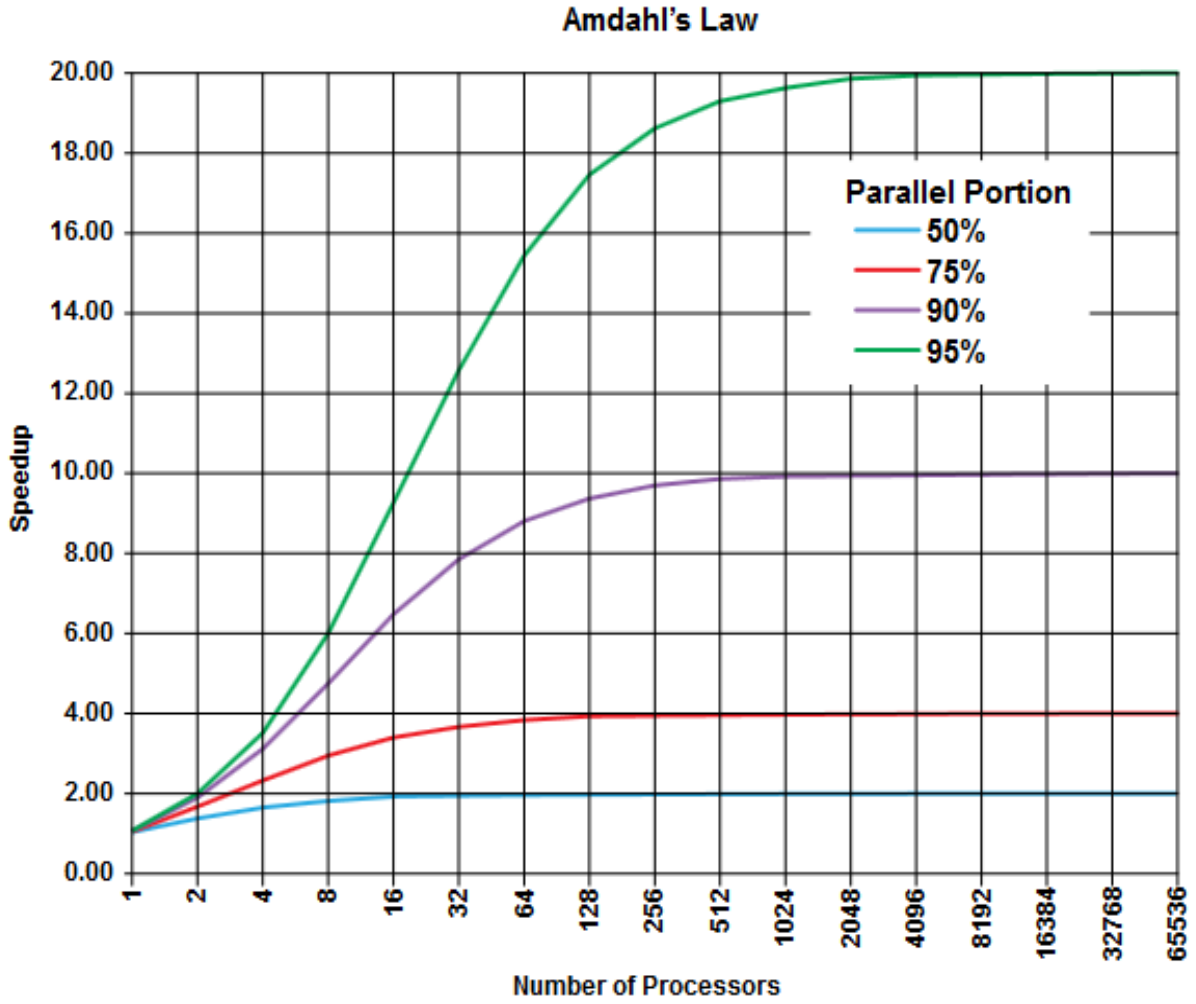
Pattern Finder Analysis

- Measure multithreading scalability
- Large data set characterizations
- Memory limited data processing speed comparisons
- Pattern coverage of overlapping and non-overlapping searches
  - Accuracy vs speed tradeoff
- Memory Access Latency statistics
- LLC (Last Level Cache) miss counts

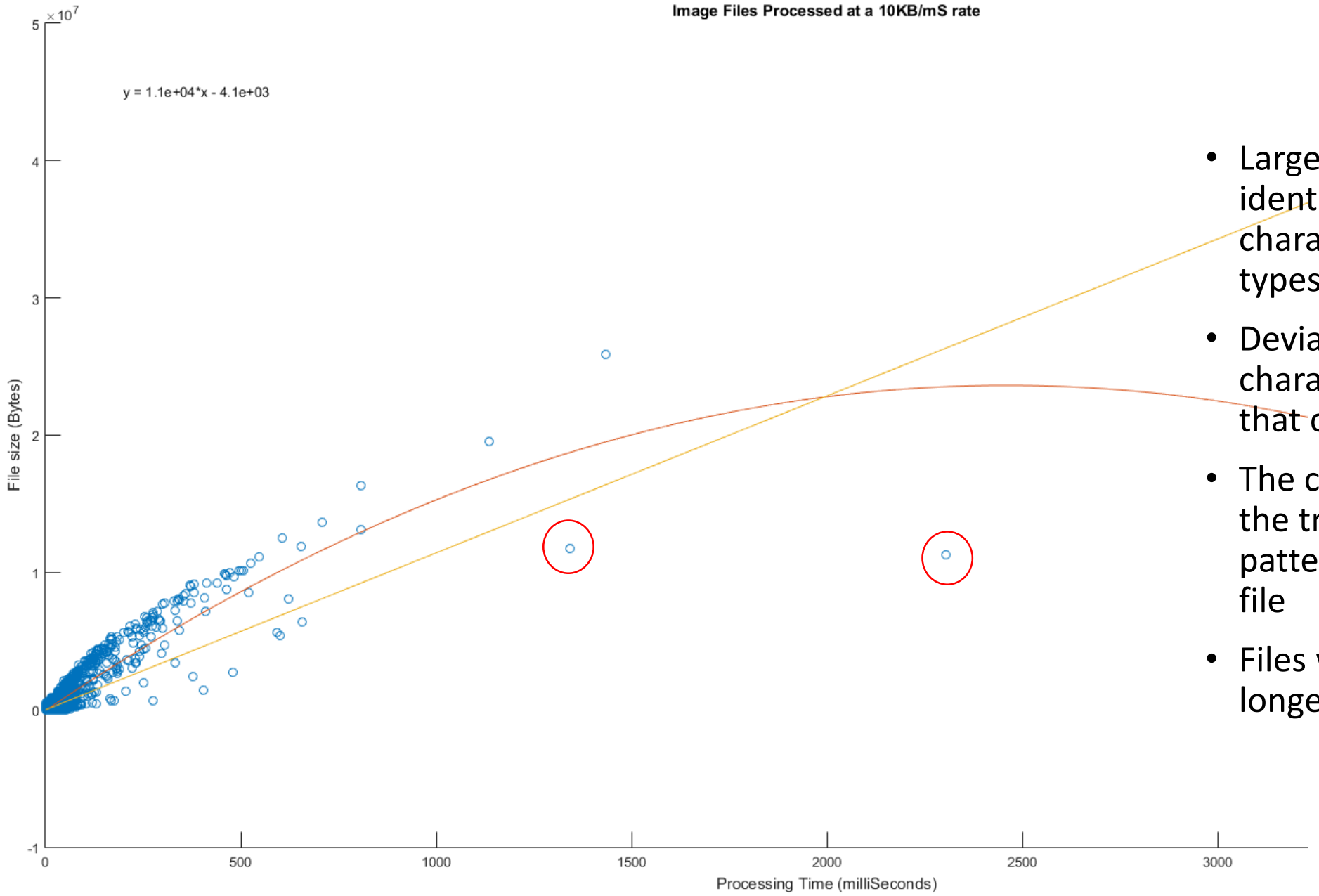
Multiprocessor Scalability

- The previous scalability results match the 98% parallel program trend line.
- Pattern Finder using 32 threads in parallel results in a 16x speed improvement





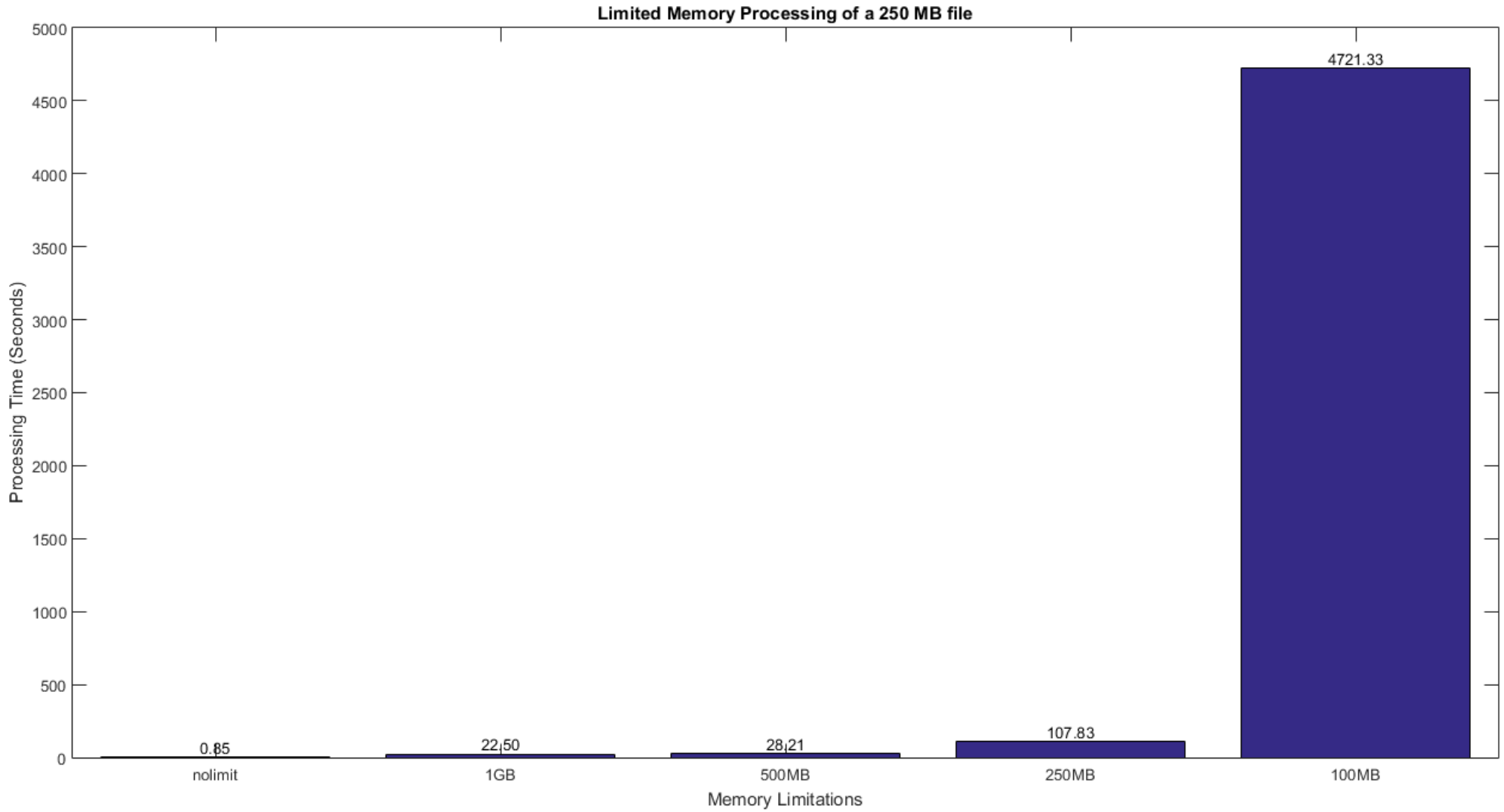
Dataset Characterizations to Isolate Large Pattern Files



- Large dataset processing identifies pattern characterizations for certain file types
- Deviations from this characterization are isolates files that contain large patterns
- The circled scatter points below the trend line indicate large pattern conditions within in a file
- Files with long patterns take longer to process

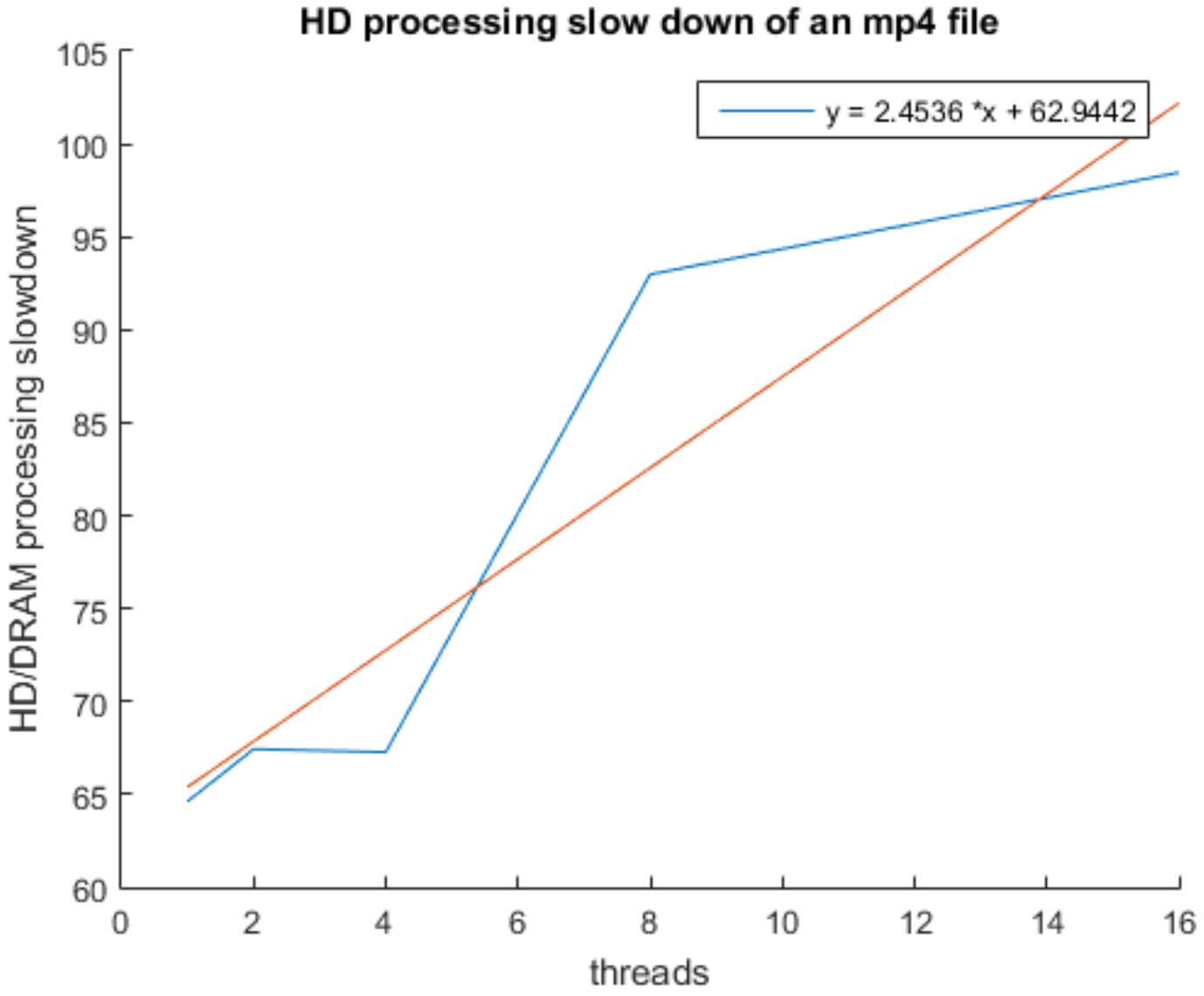
Limited Memory Processing

Memory Limit	Processing Slowdown
100 MB	5554
500 MB	33



HD vs DRAM Processing

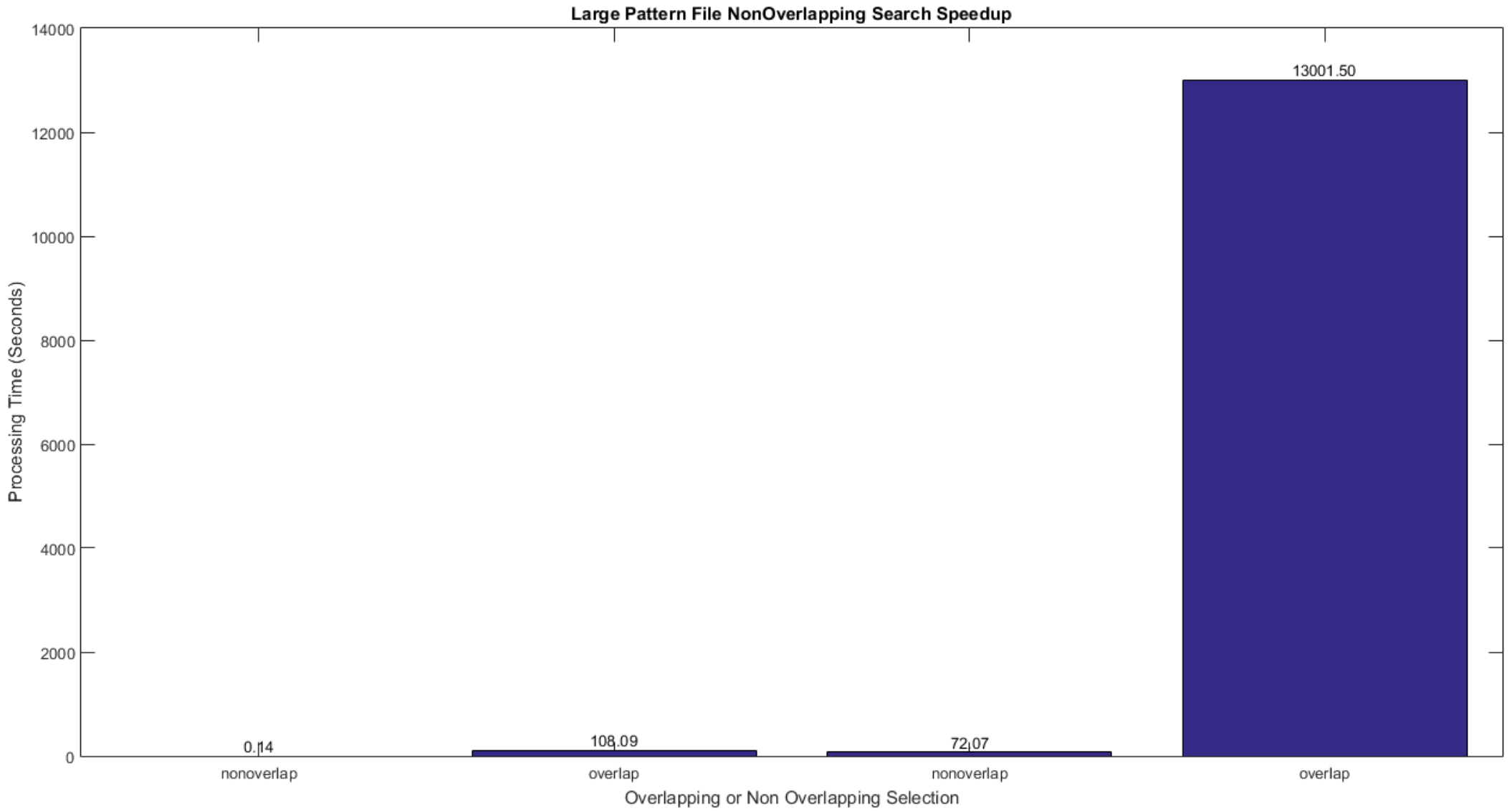
- The HD processing does not scale as well as the DRAM processing as the number of threads increase
- HD processing is about 63 times slower than serial DRAM
- As the threads increase, HD processing starts to linearly slow down at a 2.5x rate per thread compared to DRAM processing.
- HD processing does not scale well due to bad parallelization from reading and writing to the Hard Disk.



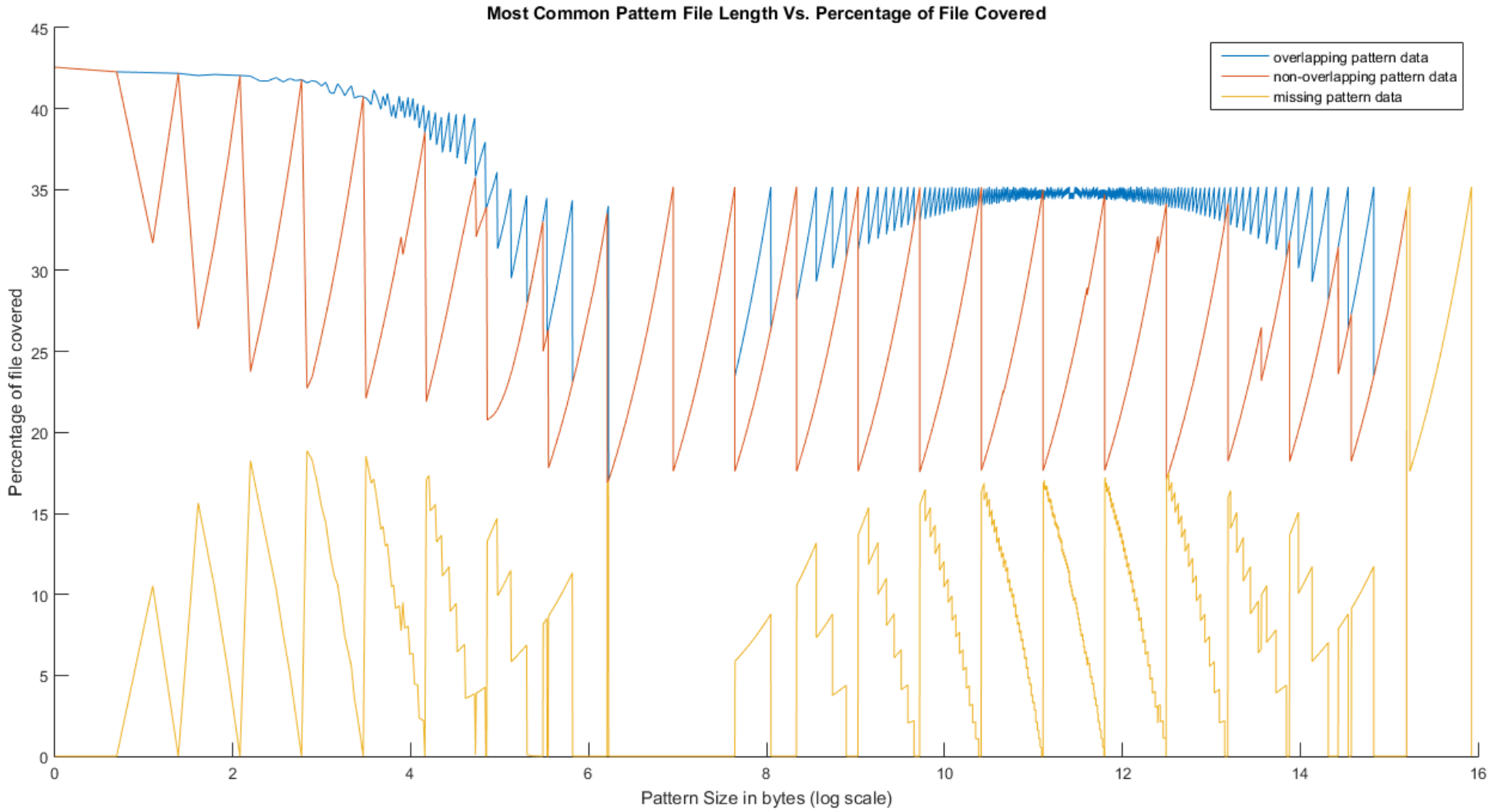
Overlapping vs Non-overlapping Search

- Files containing large patterns benefit greatly in speed when utilizing the non-overlapping search.
- The processing time is much faster while maintaining nearly 100% pattern accuracy
- Instances within a pattern set can be removed for overlapping while pattern instances that overlap other pattern instances do not get pruned.

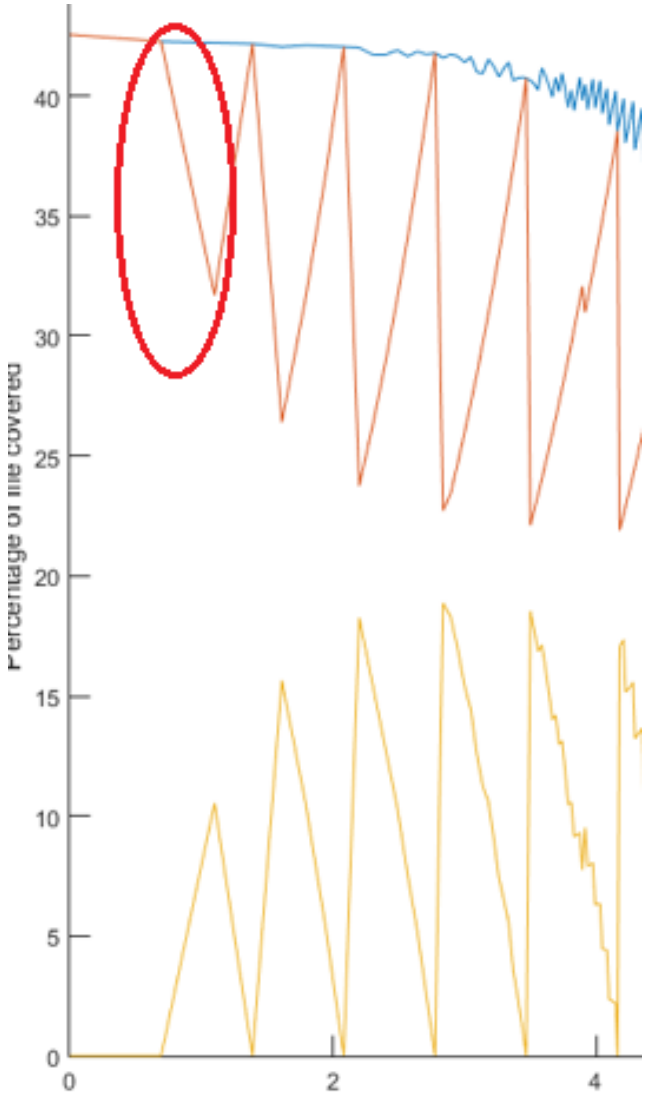
Files Containing Large Patterns Speedups



Coverage of Files Containing Large Patterns



Non-overlapping Pattern Resolution



Patterns of length 1 with 100% coverage

0	1	2	3	4	5	6	7	8	9
A	A	A	A	A	A	A	A	A	A

Patterns of length 2

0	1	2	3	4	5	6	7	8	9
A	A	A	A	A	A	A	A	A	A

Resolution of Overlap Patterns of length 2 with 100% coverage

0	1	2	3	4	5	6	7	8	9
A	A	A	A	A	A	A	A	A	A

Patterns of length 3

0	1	2	3	4	5	6	7	8	9
A	A	A	A	A	A	A	A	A	A

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

Profiling and Optimizing Pattern Finder

- Linux tools
  - Perf used for finding program bottlenecks
    - Performance test commands
      - Perf record –g ./PatternFinder –f Boosh.avi –v 1 –threads 64 -ram
      - Perf report –call-graph
  - Valgrind used for memory leak detection
- Windows tools
  - Intel Profiling Suite is a powerful memory leak and cache metric analyzer
  - Visual Studio profiler used for bottleneck detection
- Hardware Platform
  - 4 Xeon processors totaling 96 cores of processing power with 1 TB of RAM

Memory Bottleneck Results

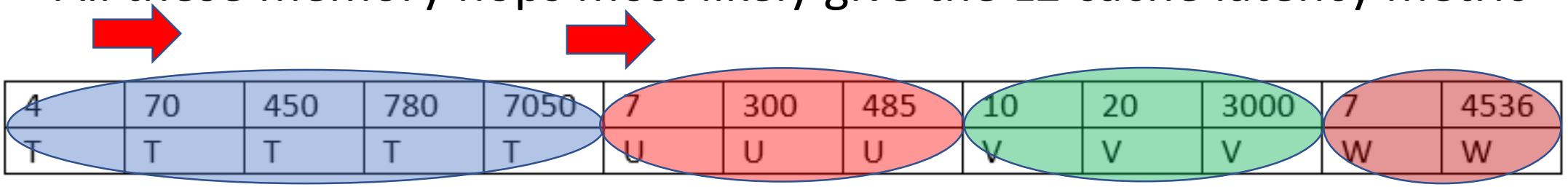
Elapsed Time: 85.792s

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

- 250 MB video file
- Cache management is essential to improving speed when scaling up with threads
- The average latency for data access is 10 cycles for this file which corresponds to an L2 cache hit performance metric
- Latency is dependent upon file composition so results may differ

Cache Spatial Locality in Large Files

- Most files processed will be very large and therefore create memory jumps in a node's address list when a node transition occurs
- Within the T node there are memory transitions from 4 to 70 which will not be covered in one 64 byte aligned cache block
- The memory transition from T to U is 7050 to 7 which is another cache miss
- All these memory hops most likely give the L2 cache latency metric



Pattern Finder as a Tool

- Generates most common pattern data per level
- Generates pattern data for overlapping and non-overlapping searches for post processing coverage comparisons
- Compounds pattern data for large file databases
- Multiple instances of Pattern Finder can be dispatched for distributed computing solutions.

Pattern Finder Options

- -f [filename | filedirectory] specific file or directory to pattern search
- -n indicates a non overlapping pattern search
- -i [patterncount] number of instances of a sequence that constitutes a pattern, default is 2
- -min [patternsize] indicates first pattern size to begin searching for
- -max [patternsize] indicates the largest pattern size that can be searched
- -threads [threadcount] indicates threads to use for processing
- -mem [MegaBytes] indicates memory limit for processing

Pattern Finder Examples

- ./PatternFinder –f Database –threads 64 –ram
  - Pattern searches all files recursively in directory using DRAM with 64 threads
- ./PatternFinder –f TaleOfTwoCities.txt –min 5 –max 100
  - Finds patterns of length 5 to 100 and then terminates processing
- ./PatternFinder –f Boosh.avi -n
  - Processes file using non overlapping processing
- ./PatternFinder –f StairwayToHeaven.mp3 –mem 1000
  - Processes file using memory prediction per level for HD or DRAM processing with a memory constraint of 1 GB

Summary and Future Work

- Thesis Accomplishments
  - A multi threaded Pattern Finder resulted in up to 23 times faster processing.
  - The best example of improvement stems from a trace file that took 12.5 days to process single threaded, 36 hours to process multithreaded and 3 hours to process using a non overlapping multithreaded search.
  - Processing large files using limited memory is possible but the processing times are still very slow if the memory limit is too small.
- Future Work
  - Find a better way to limit a program's DRAM usage
  - Implement a job distributed pattern finder script to process files on a super computer grid.

Acknowledgements

- Resit Sendag for mentoring me and guiding me through the implementation of a multi threaded Pattern Finder
- My parents for keeping me sane