Parallellism in grep: A Case of Parallel Research to Common Computing

Lisa Lau
Computer Science and Engineering
The Ohio State University

General Terms

Pattern matching, Aho-Corasick, parallel matching. parallel computing, graphic processing units, grep, bitwise operations, SIMD

1. INTRODUCTION

The command line utility grep is a simple, but extremely useful tool that has had a tremendous amount of value to the users of Unix and Unix-like systems. The grep command, also known as "global regular expression print", does precisely what it's formal name implies. It has the capability of searching for regular expressions across a global domain, in this case, a global file system, and printing all found matching lines. The internal process of a grep command has not varied much over the course of its history. Consisting of three general steps, the first step takes the first line of a designated input file and copies the line into a buffer. Then a comparison between each string is made with the regular expression. If the string matches the regular expression, it gets printed to the screen. This process is then repeated until there are no more valid lines in the input file. This entire process consists of only reading strings and does not modify or store any string values.

Grep could be considered one of the most widely used instances of a regular expression pattern matching algorithm. But while many implementations of grep have been made such as NR-grep [5], and GPP-Grep research [15], the "regular expression pattern matching" or "regular expression search" problem, where matching regular expressions over large streams of input text data has continued to be a progressing area for improving algorithmic performance to many researchers. Much progress has been made in creating a group of more efficient string pattern matching algorithms. Some of these algorithms that have shown promise over the last couple decades in increasing the performance of string pattern matching have been the Aho-Corasick Algorithm [1], Parallel-Failureless Aho-Corasick [3] and most recently, a Bitwise Parallelism approach to the problem [12].

2. REVIEW OF AHO-CORASICK

The Aho-Corasick algorithm is a foundation in the finite set of patterns classification of string pattern matching algorithms and was also used in the original implementation of the fgrep command in Unix systems. This classification of algorithms allows for the search of multiple regular expressions.

The AC algorithm consists of building a non-deterministic finite state machine from a set of regular expressions and then loading input into the finite state machine to be processed for regular expression matches. The state machine uses several types of functions to control the flow of state transitions. The Goto function defines the mapping of state transitions. The Output function defines an output of a valid regular expression. The Fail function defines the state transition that would be received by the Goto function when a mismatched character is encountered. Failure transitions backtrack the state machine in order to recognize patterns at other possible start locations.

The time required to preprocess and construct the state machine is proportional to the sum of string lengths of all the regular expressions in a given set. Memory lookup has an $O(n)$ runtime where n is the total length of all regular expression patterns. Additionally, the number of state transitions used by input strings is independent of the number of keywords in the set, therefore the AC algorithm's suggested use criteria is in domains that call for searching large keywords sets with small string lengths. [1] also discusses the idea of translating the non-deterministic finite automaton in AC into a deterministic model could potentially reduce the number of state transitions by half, where a majority of the algorithm's time would be spent in the first state and, therefore, was not recommended for practice. But this observation is extremely important to the algorithm approach summarized in the next section.

3. REVIEW OF PARALLEL-FAILURELESS AHO-CORASICK

The AC algorithm was an early milestone in the research of a better regular expression pattern matching algorithm, and it was only natural that due to such a prolific reputation, many variations were implemented such as algorithms produced by Chen and Wang [2], Ando, Kinoshita, Shishibori and Aoe [7] and Do, Kang and Kim [9]. In the algorithm approach discussed in this section, the use of parallel computing is of particular interest.

As discussed in [3], one of the pitfalls of the Data Parallel Aho-Corasick approach is the boundary detection problem. That is, when input streams are divided into segments and then each segment is ran through the state machine in its own thread, there is the case where a regular expression will not get matched if portions of that regular expression become split amongst multiple thread segments. Typically, this issue is fixed by extending segmented threads to search beyond their assigned segment, but this causes large overhead as it is unknown how to efficiently distinguish by how much a segment would need to over search into another segment of an interleaved regular expression pattern. In the worst case each segment would need to search in addition to the length of the longest regular expression pattern causing a runtime of $O(n/s+m)$ where m is the longest pattern length, n is the total length and s is the number of segments. Memory lookup requires $O(n+ms)$ complexity, a negative byproduct of DPAC compared to the original AC algorithm with a memory lookup time of $O(n)$.

[3] presents a new approach to the DPAC algorithm. The algorithm, appropriately named as Parallel-Failureless Aho-Corasick or PFAC, completely removes the use of failure transitions in the original Aho-Corasick algorithm. Other than this, the finite state machine is built and used similarly as the traditional AC. PFAC works by assigning each byte to a thread. The assigned byte is used as a marker of the start location in the input stream in each thread. Each thread then proceeds to run the

modified AC finite state machine, moving in a linear fashion from their start location in the input stream to the next location. As a result of removing failure transitions from traditional the traditional AC algorithm, when an invalid state occurs, the thread immediately terminates. Most of the threads end up terminating early due to the low probability of starting at a valid pattern state in the state machine. In addition to this high chance of thread termination, the number of valid state transitions will be minimal since each regular expression pattern in the state machine is viewed as unique. PFAC is also more favorable in terms of efficiency in memory accesses compared to DPAC. Because each thread is assigned to each byte in the input text stream and all threads are running in parallel, and by the algorithm's nature, each byte data will be read many times by neighboring threads. Therefore preloading the input text into shared memory greatly decreases the need to fetch data from global memory.

A multitude of optimizations to the PFAC algorithm were recommended, many of such revolved around memory efficient features offered in specifically Nvidia Fermi architecture CUDA-enabled GPUs. To reduce memory transfers involving global memory, memory coalescing reads are exploited to more efficiently load input data from global to shared memory. Texture memory is also used to reduce the latency associated with memory lookups of the state transition table as it is optimized for 2D spatial locality. The state transition table is first bound to texture memory and the first row gets allocated into shared memory during the preprocessing stage of PFAC.

4. REVIEW OF BITWISE DATA PARALLELISM

Data level parallelism is a form of parallelization used in many multi-core CPUs, and in almost all modern GPUs. Many of these architectures feature "Single Instruction Multiple Data" operations or SIMD. Because of the presence of vector processing in these architectures, using SIMD operations, the compiler and programmer have the ability to apply a single set of instructions to multiple data sets at the same time using parallel threads. In [12], the Bitwise Data Parallelism approach, or referred to as BDP in this paper, presented utilizes these parallel techniques in addition to the easily parallelizable nature of bitwise functions to implement a completely new approach to regular expression matching.

The BDP algorithm acknowledges the relevant work of the bit-parallel XML parser using 128-bit SSE2 SIMD technology with a parallel scanning primitive based on addition [11] and similar regular expression matching in each parallel stream as the classical algorithms of Navarro and Raffinot [6] , Baez-Yates and Gonnet [10] and Wu and Manber [13] as strong contributions to its creation. This approach views input text streams as very large integers that first get partitioned into blocks. This process is dependent on the number of parallel resources available. Bytes of data within blocks get processed by an instruction set that involves bitwise logic and long stream addition that must be scaled to each block's size. Each byte stream is substituted with eight 8-bit parallel streams with a one-to-one direct mapping from each stream i to each i^{th} bit of each byte. Due to this direct mapping, each bitstream is identifiable by other parallel bit streams, such as the character class bit stream. In the case of making necessary calculations such as whether a character in the input data stream is in a class or not, this property becomes very useful. In addition to both character class and input bitstreams, marker bitstreams are also applied to specify positions of ongoing matches during the entire process. The heart of BDP is the MatchStar operation. MatchStar returns all reachable positions by advancing the marker bitstream zero or more times via the character class bit stream. This implementation of MatchStar is similar to that the ScanThru operation found in the Parabix tool chain [4], but it differs such that it finds all valid matches, not only the longest [12].

One of the distinguishing properties of BDP is the ability to process more than one byte at a time. Blocks are only limited by the number of parallel resources at hand and, as a result, the number of bytes getting processed at the same time is also only limited by this factor. In revelation to the recent introduction of 256-bit SIMD instructions and the equivalent AVX2 instructions offered in Intel's Haswell chips, experiments were conducted comparing the two implementations of each architecture to test the scalability of the BDP algorithm with results showing no discernible reduction in instruction count and demonstrated great scalability. Similar procedures were conducted on a GPU to further assess the bounds of scalability of BDP and results showed improvements up to 60% compared to the non-GPU implementations using AVX2 and SSE. Differences in improvements compare to either AVX2 and SSE depended on factors such as hardware limitations such as register usage and the fact that long stream addition is a more expensive operation on GPUs than on SSE or AVX2 implementations.

RELATED-WORK COMPARISON

In this section, a comparison of all three mentioned algorithms will be made to further offer any additional insightful analysis.

4.1 Similarities

Because PFAC is an algorithm based off of the traditional AC algorithm they share many similarities between implementation. The use of a state machine is the foundation of both of these algorithms. They both build their state machines in memory and then use those state machines to process an input text stream one byte at a time. Since memory-bound applications require the reading and writing of data to and from global memory, latency/throughput of memory accesses is an area of concern for performance. PFAC discusses the use of GPU features such as memory coalescing and texture memory to dismiss these concerns, as well as the use of shared memory used naturally in the algorithm. Many of PFAC's memory improvements were possible as a result of the parallel approach the algorithm takes as opposed to traditional AC.

The increased use of parallel computing to in recent years to improve the general efficiencies of traditional algorithms is relevant as shown in algorithms PFAC and BDP. Both use parallel techniques to create a faster algorithm. PFAC uses a thread from each byte in the input text stream while BDP partitions bytes into blocks based on the number of parallel resources available. Both approaches aim to increase the number bytes processed at once using threads. Memory efficiency is a common area of focus in terms of experimentation with different implementations for both PFAC and BDP as well. The threads in PFAC are able to advantageously use shared memory often due to the close spatial locality of neighboring threads reading the same data repeatedly. Whereas in BDP, the algorithm is able to store all intermediate bitstreams in it's loop body in registers while outside bitstreams that do need to be stored in memory have buffers allocated to decrease the latency of fetching successive memory locations [12].

4.2 Dissimilarities

As opposed to the traditional AC and PFAC implementations, BDP is of a completely new design and not based on traditional

sequential algorithms that were later adapted to fit parallel architectures. BDP was created with the use of a parallel architecture first in mind. As a result, the algorithm does not rely on finite automata to search for regular expression patterns and is not restricted to processing input stream data at a rate of one byte at a time. By partitioning the input stream into blocks that contain more than one byte, only limited by the number of available parallel resources, the matching process instruction set is applied to each byte in the block at once, resulting in a much greater amount of throughput compared to traditional AC and PFAC.

While both PFAC and BDP use parallel computing, BDP does fall victim to performance loss due to load imbalance, unlike PFAC. In order to gain significant optimizations in PFAC, a Nvidia Fermi architecture GPU is used in their implementation for the advantages memory coalescing and texture memory bring to memory efficiency. As a result, warps are used to implement the parallel threading that occurs in PFAC. Because the overall duration of a warp is determined by the longest duration thread, PFAC is vulnerable to cases of great load imbalance amongst each thread in a warp. The BDP algorithm is not a victim of this defect as individual thread durations do not rely on one another due to the nature of SIMD/SIMT operations.

5. ARCHITECTURE COMPARISON

In this section, we will compare the architectures of three different processors: the 2.66 GHz Intel Xeon X3330, 2.2 GHz Xeon E5-2660, and the 2.3 GHz AMD Opteron™ 6276. All of these architectures operate on a 64-bit data width, are SSE, SSE2, SSE3, SSSE3 and SSE4 core instruction compliant, and have multiple cores. It is also worth noting that all compared microarchitectures also are oriented towards the server market.

5.1 Microarchitecture

All three architectures have fundamentally different microarchitectures. The Intel Xeon X3330 has a Yorkfield processor core which implements Intel's Penryn microarchitecture. This particular chip uses a 45nm transistor stepping process to produce a cooler, more power efficient environment than it's predecessor, the Merom. The Xeon X3330 features two duo-core processors for a total of four cores capable of running a total of four threads, a base frequency of 2.66 GHz, a 1333 MHz bus speed and does not have multiprocessor capabilities.

The AMD Opteron 6276 has Interlagos processor cores which operate on AMD's Bulldozer microarchitecture. Bulldozer's architecture is very much focused on improving throughput as it leverages aspects of concurrent multi-threading and has an increased CPU pipeline length of 20 stages in order to reach higher clock frequencies with scalable IPU. This architecture supports AMD's Module system which is equivalent to that of Intel's Hyper-Threading technology such that both systems implement a second thread in a single core. Specifically the AMD Opteron 6276 has sixteen "cores" capable of running sixteen total threads (AMD's definition of what is a "core" will be further explained in section 6.3), a base frequency of 2.3 GHz, a bus speed of 3200 MHz, and has a multiprocessor limit of four.

The Intel Xeon E5-2660 has a Sandy Bridge-EP processor that implements Intel's Sandy Bridge microarchitecture. Sandy Bridge focuses on per-core performance by integrating a dedicated section on chip for graphics processing (the first of it's kind), a micro-op cache to cache instructions as they are decoded and a new branch prediction unit following the standard 2-bit predictor with modifications to use 1-bit for multiple branches, leading to more accurate predictions. The Intel Xeon E5-2660 has eight cores capable of running sixteen total threads, a base frequency of 2.2 GHz, a bus speed of 4000 MHz and has multiprocessor limit of two. This architecture also has support for Intel's Hyper-Threading technology.

In general, overall the microarchitecture design of the Intel Xeon E5-2660 and AMD Opteron 6276 have more focused motivations to increase performance through instruction and data level methods rather than relying on shrinking transistor technology like the Intel Xeon X3330. This is relevant to the ending era of Moore's law as the Intel Xeon X3330 is an older architecture, being introduced in 2008, while the other two architectures are more recent (AMD Opteron 6276 was introduced in 2011 and Intel Xeon E5-2660 was introduced in 2012).

5.2 Branch Prediction

When approaching performance improvements, creating a deeper pipeline can be a lucrative area to work in. But a deeper pipeline has a lot of negative setbacks such as branch mispredictions.

A deep pipeline is utilized in the Bulldozer architecture. With it's high branch misinterpretation penalty of 20 cycles, the performance of its branch predictor is vastly an area of performance concern. Intel's Sandy Bridge pipeline is not much shorter than the Bulldozer's, so accurate branch predictions are of also a concern. But the difference between the two is that Sandy Bridge introduced the use of a 6 KB micro-op cache that caches instructions as they are decoded. This cache has the ability to reduce its branch misinterpretation penalty from 17 cycles to 14 cycles depending on whether or not an instruction can be found in the cache. Overall Sandy Bridge is about the same efficiency in terms of branch prediction compared to Bulldozer, but will suffer less from branch misinterpretations with its micro-op cache. It should be noted that, the Intel Xeon X3330 does not have an as comparable pipeline depth (14 stages) compared to the other two architectures (around 20 stages) therefore was excluded in this particular comparison.

5.3 Cache Layout

The layout difference of caches on each processor is greatly influenced by different performance motivations AMD and Intel have in their architecture designs. The Intel Xeon X3330, being the older Intel processor in this section, has each dual core sharing a 3 MB L2 Cache for a total L2 cache size of 6 MB. The L1 cache has 128 KB data cache and a 128 KB instruction cache and does not implement an L3 cache. Compared the other two architectures, the older Intel Xeon X3330 is outdated.

The AMD Opteron 6276 has a unique "dual-core" module system that shares resources such as a 2 MB 16-way associative L2 cache and a 8 MB 64-way set associative L3 cache. With eight "dual-cores", this processor has a total of 16MB of L2 and L3 cache. It should be noted that these "dual-core" modules are not really a dual-core product like that of Intel because it does not physically have two CPUs inside the module, AMD claims that each module should behave on par as if they did have two separate CPUs though. A 64KB 2-way set associative L1 instruction cache is also a shared resource amongst "dual-core" modules, but each "core" has its own 16KB 4-way set associative L1 data cache. The very low associativity of the instruction caches is an area of performance concern as in the usage of two threads on a single instruction cache can contribute to higher cache miss rates.

In the Intel Xeon E5-2660 architecture, each individual core has its own 32KB 8-way set-associative instruction and data cache and 256 KB 8-way set associative L2 cache, but a 20MB L3 cache is shared by all cores. Any data residing in an L2 cache also resides in the L3 cache as this cache also acts to speed up inter-core memory operations. The L3 cache in this processor fundamentally acts as a staging area and switchboard for all cores, storing valuable information such as whether data has already been cached in another core's L2 cache or data processed on one core must be handed off to another.

6. CONCLUSIONS

Research in the regular expression matching domain is important as the applications of a more efficient algorithm has a vast reach amongst fields such as Image Retrieval in Computer Vision [8], bioinformatics [16] and file compression techniques [14]. And, in general, advancements made in all realms of computer science and technology alike have the ability to greatly influence one another as demonstrated by the work written about in earlier sections.

REFERENCE

[1] Alfred V. Aho , Margaret J. Corasick, Efficient string matching: an aid to bibliographic search, Communications of the ACM, v.18 n.6, p.333-340, June 1975 DOI= http://dl.acm.org/citation.cfm?doid=360825.360855

[2] C. Chen and S. Wang, An efficient multicharacter transition string-matching engine based on the aho-corasick algorithm. ;In Proceedings of TACO. 2013, 25-25.

[3] C. Lin, C. Liu, L. Chien, and S. Chang, "Accelerating Pattern Matching Using a Novel Parallel Algorithm on GPUs," IEEE Transactions on Computers, vol.62, no.10, pp.1906,1916, Oct. 2013. DOI= http://dl.acm.org/citation.cfm?doid=2628071.2628079

[4] D. Lin, N. Medforth, K. S. Herdy, A. Shriraman, and R. Cameron. Parabix: Boosting the efficiency of text processing on commodity processors. In 18th International Symposium on High Performance Computer Architecture (HPCA), pages 1{-12. IEEE, 2012.

[5] Gonzalo Navarro, NR-grep: a fast and flexible pattern-matching tool, Software—Practice & Experience, v.31 n.13, p.1265-1312, November 10, 2001 [doi>10.1002/spe.411]

[6] G. Navarro and M. Raffnot. A bit-parallel approach to suffix automata: Fast extended string matching. In *Combinatorial Pattern Matching*, pages 14-33.Springer, 1998.

[7] K. Ando, T. Kinoshita, M. Shishibori and J-I.Aoe. "An Improvement of the Aho-Corasick Machine", Information Sciences, Vol. 111, 1998, 139 - 151.

[8] Mei-Chen Yeh , Kwang-Ting Cheng, A string matching approach for visual retrieval and classification, Proceedings of the 1st ACM international conference on Multimedia information retrieval, October 30-31, 2008, Vancouver, British Columbia, Canada [doi>10.1145/1460096.1460107]

[9] P. Do, H. Kang, and S. Kim, Improving a hierarchical pattern matching algorithm using cache-aware Aho-Corasick automata. ;In Proceedings of RACS. 2012, 26-30.

[10] R. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Communications of the ACM*, 35(10):74-82, 1992

[11] R. D. Cameron, E. Amiri, K. S. Herdy, D. Lin, T. C. Shermer, and F. P. Popowich. Parallel scanning with bitstream addition: An XML case study. In *Euro-Par 2011 Parallel Processing*, pages 2-13. Springer, 2011.

[12] Robert D. Cameron, Thomas C. Shermer, Arrvindh Shriraman, Kenneth S. Herdy, Dan Lin , Benjamin R. Hull and Meng Lin (2014) ' Bitwise data parallelism in regular expression matching', *PACT '14 Proceedings of the 23rd international conference on Parallel architectures and compilation*, (), pp. 139-150 [Online]. DOI= http://dl.acm.org/citation.cfm?doid=2628071.2628079

[13] S. Wu and U. Manber. Agrep - a fast approximate pattern-matching tool. *Usenix Winter* 1992, pages 153-162, 1992.

[14] Tao Tao , Amar Mukherjee, Pattern Matching in LZW Compressed Files, IEEE Transactions on Computers, v.54 n.8, p.929-938, August 2005 [doi>10.1109/TC.2005.133]

[15] V. C. Valgenti, J. Chhugani, Y. Sun, N. Satish, M. S. Kim, C. Kim, and P. Dubey. GPP-Grep: High-Speed Regular Expression Processing Engine on General Purpose Processors. In Research in Attacks, Intrusions, and Defenses, pages 334-353. Springer Berlin Heidelberg, 2012.

[16] Vidya Saikrishna, Akhtar Rasool and Nilay Khare, "String Matching and its Applications in Diversified Fields",IJCSI International Journal of Computer Science Issues, Vol. 9, Issue 1, No 1, January 2012 Page-219-226. ISSN (Online): 1694-0814.

[17] N. Deng, C. Stewart, D. Gmach and M. Arlitt, “Policy and mechanism for carbon-aware cloud applications,” Network Operations and Management Symposium (NOMS), 2012

[18] N. Deng, C. Stewart, D. Gmach, M. Arlitt and J. Kelley, “Adaptive Green Hosting,” International Conference on Autonomic Computing (ICAC), 2012