New Pattern Matching Approaches Comparison

Nima Esmaili Mokaram
Ohio State University
78 W 10th Ave., Apt. A
Columbus, Ohio, 43201
(571)480-2053
esmailimokaram.1@osu.edu

ABSTRACT

In this paper, some of the newly researched and developed methods and applications related to regular expression matching will be described. In the past, there were a significant amount of effort on making the regular expression matching problem faster using different algorithms. There were a lot of success using different techniques such as implementing the regular expressions based on NFAs, DFAs, Backtracking, or Bitwise Representation. However that has lost its momentum due to lack of improvement simply by solving the problem using a different algorithm. An era has begun were computing problems have started to scale out meaning using multiple of many cores or changing the underlying architecture. This paper will specifically go over scaling, algorithms, performance measures, resources used, and the selection of architecture discussed in three different papers.

Categories and Subject Descriptors

3.74, 3.71, 5.22, 5.25

General Terms

Verification, Reliability, Performance, Measurements, Experimentation, Design, Algorithms.

Keywords

Comparison, grep, regular expression, architecture, comparisons, GPUs, Aho-Corasick, DPAC, PFAC

1. INTRODUCTION

The use of parallelism has been dominating the world of computer science because of many reasons that computer science has been facing for the last decade such as power wall. One solution and maybe the most feasible solution so far has been to "scale out" meaning to do the same thing using more resources rather than faster or better resources. As the result, computer architects changed direction and started working on designing CPUs with multiple cores and it has recently reached many-core designs.

This is great from an architecture point of view if you look at it from a theoretical point of view. This means that theoretically, we should get better performance using this new technology. However, one of the main issues is that this new door that has been opened to the world of computer science – multi-core/many-core era – is still not quite adapted with all the existing software. All the existing software to get the full benefit of this new technology must be changed in a way to really take advantage of the available resources. This will then introduce a whole wave of developers starting to work on all the existing programs and software to make them able to use all the capabilities of the new technology. This includes the classical grep (global regular expression print) program.

Grep is one of the most commonly used program for text searching and there has been a lot of work and research done to make it fast and efficient. Grep is no exception and has been the subject to multiple groups of researchers and developers have been working on parallelizing it and keeping its performance within the current level of demand. A significant proportion of this research will be spent on showing related works and trying to show the similarities and differences between the different approaches to this common problem from both an algorithmic and architectural stand point.

2. Related Work

2.1 Bitwise Data Parallelism

In this paper, there are two main approaches to this problem. Cameron et all [1] both approached it algorithmically and architecturally. First they came up with an algorithm that used bitwise operations that can outperform the existing grep implementations. It is also designed in such a way that can be scaled out using more resources. Some comparisons to other currently used greps are shown using this algorithm on Intel AVX2 and using GPUs.

This problem is usually approached with either representing the actually regular expression using NFAs, or DFAs. However, in this paper, they approach it using a completely different way. Bitwise approach, they represent the text as series of bits based on the character encoding and similarly with regular expressions. One main different that was noticed in their approached was that they store the class of regular expression and then just mark the input text with only that particular class of regular expression all throughout the input text and then move to the next class of regular expressions and advance the currently marked input text until they reach the end of classes of regular expression that was specified in the input.

Obviously, this means that they need to divide up the input text to many reasonably sized sections. In their algorithm, they use bitwise operations to locate the matches for a particular regular expression. This means that they will need to handle carry-outs and carry-ins of different sections of the input. They used Parabix Toolchain [2] to achieve this. Using Parabix was both beneficial and somewhat hindering. This is due to the fact that using Parabix allow them to make the parallelization of the problem a bit easier to approach, but on the other hand, it introduced a new set of difficulties. These difficulties include the limited field size of vectors available in SIMT and SIMD instructions. This value is 64 bits for most new Instruction Set Architecture.

There are comparisons of this algorithm being implemented for SSE2 (Streaming SIMD Extension 2), Intel's AVX2, and on GPU. All the comparisons are done with four different sets of regular expressions.

- Simplest regular expressions
- Commonly used regular expressions
- Repeated regular expressions
- Very odd regular expression

Each of these categories express a certain feature of a program that is written for such purpose. Main comparisons are done on Instructions per Cycle and Cycles per byte and comparing the bitstream grep with nrgrep and gre2p. These of two of the most popular greps that are being used today.

Doing 5X better on average is a really performance increase for a newly introduced algorithm. The complete details of the comparison could be found on the paper [1].

2.2 Efficient String Matching

In this paper, they introduce a new algorithm to find all the occurrences of keywords in a given text. There is no further analysis as to how this algorithm can be improved on different architectures or if they run in on multicore CPUs or GPUs. This is indicating that the main focus of the paper is on the algorithm itself and proving that the algorithm actually does find all the instances of the keywords in the text with a certain time and space complexity. There is a great deal of theorems, lemmas, and proofs on those time complexity as well.

The paper is using a variant of the Knuth-Morris-Pratt along with finite state machines to come up with a simple and efficient algorithm to locate all the occurrences of any of a finite number of keywords in a text. This implementation is used in a software for bibliographic search in libraries and has improved the search speed by a factor of 5 to 10 [3].

In the algorithm offered by Alfred Aho and Margaret Corasick the keywords are not particularly regular expressions. They are a restricted set of regular expressions which only consist of keywords with no symbols. In this algorithm, what they are trying to solve is if we are given a set of keywords $\{y_1, y_2, \dots, y_k\}$ namely K and x be an arbitrary string, find all the substrings of x that are keywords in K. In order to do this, they use three main functions:

1. Goto function: g
2. Failure function: f
3. Output function: $output$

2.2.1 Goto Function

This function is how the algorithm moves through the given text to find the keywords. It is very similar to a Finite State Machine. The goto function, g will contain all the keywords in it. The best way to visualize this function is a directed graph that the vertices are the different states and the edges are the letters of the keywords. You can see in figure 1, we have the goto function for the keywords $\{he, she, hers, his\}$.

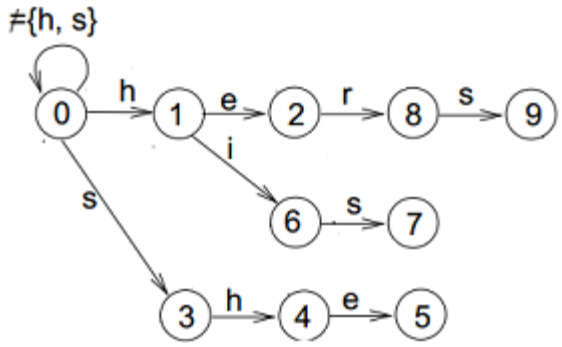

Figure 1. Goto Function

As you can see in the graph above, there is an edge connecting the starting vertex to itself. This edge will help the algorithm to go through the letters of the string that are no in the keywords without stopping. An example of using the goto function will look like this [5].

$$g(1, e) = 2$$

This structure can be stored in many different ways and each will have advantages and disadvantages. Some of the ways mentioned include using 2D Arrays, linked lists along with lookup tables for the more frequently used symbols, and binary tree.

2.2.2 Failure Function

The failure function without going into too much detail is a function that will return the state that the algorithm should move to when it fails to find its next state from the goto function. For example, in the above graph, the failure function will look be as follow [3].

i	1	2	3	4	5	6	7	8	9
f(i)	0	0	0	1	2	0	3	0	3

The above table shows that for example if the algorithm is currently at state 3 (the first letter read is an s) and the next letter is no h, the algorithm will start from the state 0. However, if the algorithm is currently at state 4 (the first two letters read are s and h) and the next input character is not an e, the algorithm does not have to start over. Instead it can start from state 1 which is equivalent to having read an h (which was the last character read).

2.2.3 Output function

The output function simply is a mapping of states and what keyword is found that that particular position. Going along with the our example, the output function will be

i	output(i)
2	{he}
5	{she, he}
7	{his}
9	{hers}

The paper goes into a great detail as to how these three functions are generated, are getting their values, and even their pseudo codes. It also talks about their time complexity and how the non-printing portion of the algorithm can be implemented to process a text of length n in cn steps, where c is independent of the number of keywords.

There is further explanation on how the failure states can be eliminated using the next move function of a deterministic finite automaton (DFA) instead of the goto function. Using DFA, it replaces both the goto function and the failure function as it contains both of those in it. Using DFA could potentially decrease the number of states transitions by 50% [3].

Finally at the end, show that they actually used this algorithm in a real world application for a bibliography search at a public library. The results from the comparisons of the old search and the newly implemented search is shown in the table below. It's worth noting that he numbers in the table are in hours.

	15 keywords	24 keywords
Old	0.79	1.27
New	0.18	0.21

Table 1. CPU Time

2.3 Accelerating Pattern Matching Using a Novel Parallel Programming Algorithm on GPUs

In this paper from Cheng-Hung Lin, Chen-Hsiung Liu, Lung-Sheng Chien, and Shih-Shieh Chang, they are focusing on the use of pattering matching in Network Intrusion Detection Systems (NIDS). NIDS have been using pattern matching algorithms for a long time as way to identify and protect computer systems from network attacks such at Denial-of-Service (DOS), port scans, and/or malwares [4]. One reason for them to work on this problem is the increasing speed of the networks. The higher the speed of the networks get, we need faster ways to process these information to find out if they are network attacks or not.

GPUs have been used for acceleration of the pattern matching problems before. However, this paper brings in an algorithm that use GPUs in their best. GPUs are for solving problems that are highly parallelizable such as matrix manipulation for a lot of graphics problems. The problem of pattern matching is not highly parallelizable by nature.

In this paper, they use the idea of Aho-Corasick algorithm [3] (section 2.2 of this paper) as the base of their algorithm. The closest that other researchers have come to making the pattern matching very parallel is to divide the text into multiple sections and giving that portion if the input text to a thread or processers depending on whether they are using OpenMP or GPUs. This approach will first introduce a new problem called the "boundary problem" which is explained in section 2.3.1 of this paper. The way that they are approaching the problem is to take full advantage of the GPU providing an enormous number of threads. By maximizing the parallelism, they ultimately will be increasing the throughput of the algorithm due to the fact that the GPU will not be doing no ops and will always be doing something which would result in 100% efficiency.

2.3.1 Boundary Problem

The boundary problem happens when the input string of the pattern matching problem is divided into many section and is given to different processes or threads.

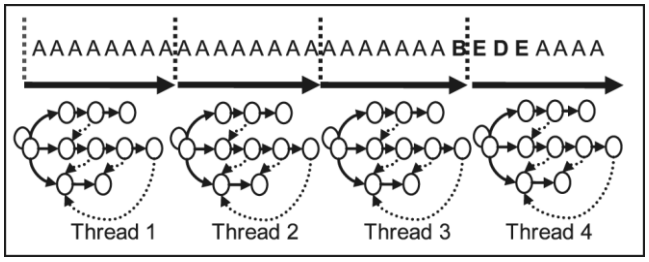

Figure 2. Boundary Problem

As shown in figure 2, if separate threads are doing the different sections of the input text, they won't be able to recognize the "BEDE" pattern (if that is one of the keywords).

Solution to this problem is not very hard, however it comes with a cost. The boundary problem can be resolved by simply letting each thread check some of the neighboring letters (length of the longest pattern). Since we are doing this for all the threads, this will add to the time complexity. The resulting time complexity will be

$$O\left(\left(\frac{n}{s}+m\right)*s\right)=O(n+ms)$$

It is worth mentioning here that the bottleneck of most GPU applications is the memory lookup time.

2.3.2 Parallel Failureless-AC Algorithm

Parallel failureless Aho-Corasick (PFAC) algorithm is what this paper comes up with and uses with GPUs to improve the performance of the pattern matching problem. The basic of the PFAC is that each byte of the input text is given to a different thread. This means that we start a new search at each letter of the input string. Due to the fact that there is a search starting at each letter, there is no need to the failure states in the state machine (the goto function of the AC). This property allows each thread to terminate when it cannot advance (where normally failure function comes to play) figure 3. This behavior will result in an efficiency of $O(1)$ in the best case – when the thread terminates at the very first state or finds the result in the very first state – or $O(m)$ where m is the length of the longest pattern.

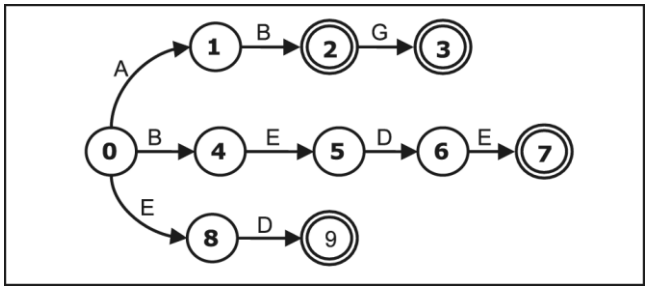

Figure 3. State Machine of PFAC

Similar the previous situation, the problem of boundary still exists where threads go to each other's search space. The problem now is that it does not result in wrong results. It is simply an efficiency issue. The total overhead could be calculated using the below formula.

$$(m-1)*s=(m-1)*\frac{n}{w}$$

Where n is the input length, m is the longest pattern length, s is the number of chunks (which in this problem is equal to n), and w is the chunk size (which is this problem is equal to 1).

As it was mentioned before, the dominating factor in the latency is the memory access time, especially in GPUs. There are many ways to decrease the memory latency when programming for GPUs. Those include:

1. Using shared memory
2. Coalescing memory accesses
3. Binding the state transitions table to texture memory

From the list above the easiests are shared memory and coalescing the memory accesses in GPUs. Since each threads is reading each letter starting at its own location, this will result in different threads loading the same memory location multiple times which will result in a longer latency due to global memory accesses. One way to improve this latency is to use the shared memory local to each thread block. Each thread would load the corresponding letter from the input text to the shared memory of the thread block. By loading the data into the shared memory, all the threads from that block can access that memory with a significantly lower latency compared to the global memory. Another advantage of this method is that when other threads are accessing the neighboring input characters it would be in the shared memory and would not add to the global memory access latency at all.

Another advantage of this method is that since the threads are accessing the data from the input text in order – thread 1 is access the first byte of the input data, thread 2 is accessing the second byte and so on – it will result in data coalescing which will cause a large chunk of the data to be brought to the shared memory instead of byte by byte (this is a GPU hardware feature).

2.3.3 Experiment Setup

The experiments environments were setup such that the host was running on an Intel Core i7 950 with 32KB of L1 cache per Streaming Multiprocessor (SM), 256KB of L2 cache per SM, and 8MB of L3 cache for all SMs. The kernel was running on a NVIDIA GTX580 with 16SMs and 512 cores and 1,536 threads per SM. 16KB of L1 cache, 48KB of shared memory (per block), and 768KB of L2 cache for all the SMs.

There were four program implemented for the experiments:

1. CPU implementation of AC algorithm compiler optimized. (AC_{CPU})
2. CPU implementation of AC using OpenMP with 8 threads. ($DPAC_{OMP}$)
3. CPU implementation of AC using OpenMP with n threads where n is the length of input string. ($PFAC_{OMP}$)
4. GPU implementation of AC algorithm with 256 threads per block (1D block of threads) – the grid size would be $\frac{n}{256}$ in the x direction. ($PFAC_{GPU}$)

The results show that the system throughput of the GPU implementation – PFAC – is 7.85, 1.55, 1.19 times higher than the AC_{CPU}, $DPAC_{OMP}$, and $PFAC_{OMP}$ respectively.

This is showing that the algorithm also can be used for CPU multithreaded programing and will show improvements over the AC algorithm.

It is worth noting that most of the throughput is taken up by the data transferring between the host and device (CPU and GPU). The actual computation has a throughput of 143.16 and is 74.95, 14.74, and 11.35 times higher than the AC_{CPU}, $DPAC_{OMP}$, and $PFAC_{OMP}$ respectively.

3. Similarities

In this section, this paper describes the similarities between all three papers described above (section 2). The most important similarity between all three papers was that they all were trying to solve the pattern matching problem. Each paper had a different approach which we will further explain in section 4.

Between the second and third paper (Accelerating Pattern Matching Using a Novel Parallel Algorithm on GPUs and the Efficient String Matching: An Aid to Bibliographic Search), it can be seen that both papers use the same technique. They are both based on the KMP algorithms. These category of pattern matching algorithms are the ones that take advantage finite state machines that allows them to find the matches simply by one pass over the input string. This is given that we have the finite state machine). Furthermore, the third paper is basing its initial algorithms on the AC algorithm which is what the second paper concluded (They have references to the second paper).

Between the first and second paper, it can be seen that both of them are approaching the problem of pattern matching using parallelism. Both papers first come up with a new algorithm and then scale out from there by applying that algorithm to multicore processors or GPUs. They both use also GPUs to implement their algorithm. This means that both algorithms can take advantage of high degrees of parallelism. Both techniques can be applied to GPU programming and multithreaded CPU programming even though the third paper is really only written for GPUs.

In both the first and last paper, the GPU implementation had a significant improvement over other approaches which shows that the nature of the problem may not seem so parallelizable, but in reality it may be (as you saw) a great candidates for high degrees of parallelism.

4. Dissimilarities

This section of the paper will describe the differences between the three different approaches to solving the same pattern matching problem.

The biggest thing that can be seen is that the first paper (Bitwise data Parallelism in Regular Expression Matching) is focusing on the fact that the keywords are regular expressions. This by nature means that the number of keywords increase due to the fact that you can express many keywords by only few characters in a regular expressions. This has caused the first paper to put extra emphasis on how to turn regular expressions into data structure that can be used into searching a text string. For that they have used different bitwise operations to do so.

Another main different between the first paper and the other two is that in the first paper, all the operations are bitwise which is first citizen to computers and in the two other papers, they are left at characters and are represented by bytes. This is causing the first approach to be slightly harder to grasp at first.

In the third approach (GPU with AC algorithm), there has been a significant more stress on memory accesses which is absolutely necessary due to the slow nature of DRAM compared to GPUs and CPUs. However, in the bitwise approach, there is very limited hints to the problem of slow memory access time and potentially solutions to it similar to the third paper.

Another big difference between the approaches is that in the second paper the author is only mentioning the algorithm, the proofs related to the algorithm, and some implementations of the algorithm. On the other hand, in the first and third paper, the issue of pattern matching is taken to a whole new level. Not only they come up with a new algorithm (or an alteration of an existing algorithm), but also they work on scaling out and making the algorithm work on many cores. This is due to the high demands of software, networks, and users. They both execute on the idea of parallelism very well unlike the second paper.

5. Architectural Comparison

From an architectural stand point, the three approaches are somewhat fundamentally different. The Bitwise data parallelism approach is mainly focusing on SSE2 and AVX2 systems. Even though they did implement their algorithm on GPUs (AMD), some features of the algorithm were too complex to implement and therefore were omitted. On the other hand, The Accelerating Pattern Matching approach is putting almost all of its focus on GPUs and is only really targeting GPUs. It turned out that their algorithm is also beneficial when used on many core CPUs.

Similarly, the bitwise approach is trying avoid the problem of long-time-taking global memory accesses by simply moving from NVIDIA to AMD and using the zero-copy memory region. This results in their analysis to not include the best throughput between the CPU and GPU and vice versa.

In the second paper by Aho and Corasick, they really are not involving themselves with architectures in the first place. They are simply implemented a new approach/algorithm for solving the pattern matching problem when the keywords are not regular expressions.

6. Conclusion

In conclusion, after reading these three paper, each paper is really targeting a specific area of the problem that if and is an answer to a specific questions. For example, the "Efficient String Matching: An Aid to Bibliography Search" is an approach that is an answer for bibliography search for papers. The "Accelerating Pattern matching Using a Novel Parallel Algorithm on GPUs" is the solution for the pattern matching problem for Network Intrusion Detections Systems due to its high throughput demanding. Finally the "Bitwise Data Parallelism in Regular Expression Matching" is a great solution for when the input can be regular expression for example grep application and for personal computers.

References

1. R. D. Cameron, T. C. Shermer, A. Shriraman, K. S. Herdy, D. Lin, B. R. Hull, M. Lin, "Bitwise Data Parallelism In Regular Expression Matching,"
2. 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
3. A. V. Aho and M. J. Corasick, "Efficient String Matching on Bibliographic Search," Comm. ACM, vol. 18, no. 6, pp. 333-340, 1975.
4. C. Lin, C. Liu, L. Chien, and S. Chang. "Accelerating Pattern Matching Using a Novel Parallel Program on GPUs," IEEE, *Transactions on Computers*, 62(10), 2013
5. Saima Hasib et al, / (IJCSIT) International Journal of Computer Science and Information Technologies, Vol. 4 (3) , 2013, 467-469 www.ijcsit.com 467