MPS: Improving Exact String Matching Through Pattern Character Frequency

Vijay Sahota[1], Maozhen Li[2], Richard Bayford[3]
[1]Canterbury Christ Church University
Kent, UK
[2]Brunel University
London, UK
[3]Middlesex University
London, UK

ABSTRACT: *One of the initial hurdles in taking advantage of big data is the ability to quickly analyze and establish its relevance. Intrinsic to this big data analysis problem is the need for tools to scale well in proportion to the growth of data and have the flexibility to operate across disparate datasets. Exact string matching is a fundamental tool used to search through data, though many existing algorithms do not scale well since their computational cost occurs and grows during reading of data. A proof of concept is presented in this paper which transfers all the required calculations to a pre-processing stage, removing all calculations during the reading stage; creating a search trie which incorporates the statistical distribution of characters in the search pattern to reduce overall calculation and lookups in a search. The resulting algorithm produces a total calculation costs which is independent of data (source) size. Preliminary results shows the new algorithm out performing existing general algorithms, as the search pattern becomes large for natural English text and when searching a small alphabet source (DNA).*

Keywords: String matching, Look up Table, Experimental Comparison, On-line Algorithms, Pattern Search

Received: 29 May 2013, Revised 3 July 2013, Accepted 9 July 2013

1. Introduction

The data deluge [7, 17] is very much a reality, with the recent explosion of very large scale experiments, ubiquitous information-sensing mobile devices and the general growth of human data production (i.e. social media). The world is now producing on average 2.5 Exabyte's of data each day [18], leaving existing methods of analysis defunct for this rate of big data production. Current popular solutions tend to pre-process large data, indexing its contents (classification) creating a map, as seen in the inner workings of map reduce [16]; which represents the data in a smaller, manageable form allowing for quicker searches (in trade for a one off costly pre-process). Given the size and growth of big data the idea of having to pre-process it is highly undesirable or often near impossible, furthermore the creation of maps require an agreement of classification metrics which goes against the core concept of big data analysis in that it inhibits true/open operation across multiple disparate datasets.

Online algorithms present a more flexible option which do not require any pre-processing as they search for patterns as the source is being read/ created; thus proving a the better solution for growing data. Another aspect to reinforce the need for better online algorithms is the ability for the extraction of knowledge from multiple disparate sources in its raw/ unprocessed form,

which will allow the formation of new original information relations that generally will not conform to the perceived parameters of any directed pre-processing (classification).

One of the main areas where big data will be produced and have a greater impact on humanity is the research and development which will harness genetic information on a personal level (a complete genome would be a text file containing 3 billion base pairs, per person). Searching for a specific DNA sequence in a genome is essentially searching for an exact pattern within a body of source text, which is a well-studied problem and is one subset of the many ways to analyses/mine DNA data. Given the nature of DNA (short alphabet) when searching for a specific sequence/ string large parts of the source data (DNA) seem very similar in nature to the specific search string triggering off many false positives leading to excessive calculations, hence excessive search times when using traditional algorithms. Specialized algorithms for DNA searches such as BLAST uses the concept of seeds for approximate matches and then further these to exact matches, however the main drawback with this is the large amounts of memory required to search a complete genome.

Another main area of big data production is in the production of data by the common user via social media and their usage of search engines; both resulting in tools such as Google trends that processes this big data to forecast online trends; which in turn has been used to assist in the prediction of stock market shares [19]. Often data is now stored/transported in the de facto XML format, hence for the near future; searching English text will still be a dominating factor for the internet. A similar case stands for searching though English text as does with DNA, despite a longer alphabet the English language has rule on spelling and sentence construct which too leads to a case of triggering off false positives (though considerably less than DNA) which increases if key phrases are being searched. Although most of the existing search algorithms for English have very little memory requirements, they still do suffer from '*on the fly*' calculations which grows in direct proportion to the growth of the source data.

The focus of this paper is to address the issues of falsely checking patterns within a source that resembles the search string (false positives) and to reduce the overall number of calculations needed. This paper presents a proof of concept online exact string matching algorithm – the maximal probabilistic shift method (MPS). Emphasis in the MPS is placed on constructing a look up table that will incorporate both maximal shifts/ sub match rules. These rules are based around reoccurring patterns within the search pattern and the statistical distribution of the alphabet within the search pattern, which further circumnavigates the need to analyze the whole source (data) text to gain a character distribution. The viewpoint is that despite a one off pre-processing cost associated with each pattern (keyword/s), most searches are often repeated on different sources; such as the searching for a genetic marker across many different genomes. Due to theses repeated searches the pre-processing cost will be offset by the returns in time saved in performing multiple searches. A brief background is presented next followed by a thorough description of the MPS algorithm through example in section III. Section IV outlines the experimental procedure adopted as well as presenting the results. Section V discusses and analyses the results of these experiments; with the final section presenting concluding remarks and future directions.

2. Background

Exact string searching algorithms such as the Boyer-Moore (BM) [1] and its derivatives such as Sunday's Quick Search algorithm (QS) [2] and Maximal Shift (MS) [9], have been the longest standing fastest practical algorithms, with most still being used in modern text editors to provide their '*search*' functionality. The upsurge in needing to process large amount of data, and the recent arrival of big data has resulted in a burst (50%) of research within the last ten years [13]; producing many new algorithms that have evolved and over took BM/QS/MS in performance, although still inheriting their fundamental traits. Key to the reign of the BM/QS/MS algorithms was to view the source text through a window and then evaluate if a possible match could be contained within this window. Using a sliding window allowed the algorithms to intrinsically break down large data into manageable chunks whilst at the same time give a natural skipping distance if it was evaluated not to contain the search pattern.

The QS/MS methods are an optimized form of the BM algorithm and so a closer look at BM follows. The use of windows gave rise to the concept of checking the last character in a pattern first; this allowed an algorithm to disregard an entire window size (bad suffix shift) if a complete mismatch (read value does not exist in the search pattern) was found, given a pattern length n the cursor is shifted by n relative to the last read character as depicted in Figure 1. In an ideal scenario where the search pattern consist of very rare (least frequent) characters then the BM algorithm would operate towards its optimal speed, since the majority of the shifts would be of size n.

In the case where the read character matches the last character of the pattern, the algorithm continues to compare the characters,

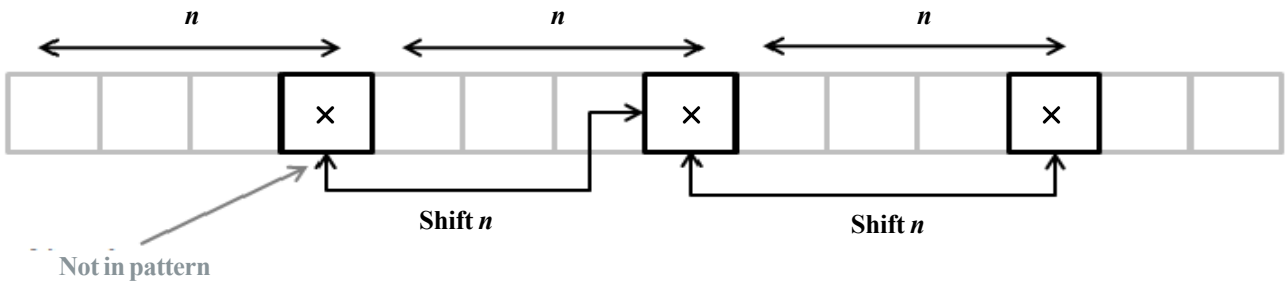

Figure 1. Bad Suffix Shift

confirming from right to left until all characters are matched and hence the pattern found. Alternatively if at any point the read character is a mismatch - where the read value is not what was expected, but still presented a suffix of the search pattern; the window would slides accordingly (good suffix shift) and restart the comparison from this new position as depicted in Figure 2. The evaluation of the read characters to be a suffix and the shift amount is calculated on the fly/ during the search and in every case. In a worst case scenario where the search pattern and the source text consists of many similar patterns BM algorithm would operate at a sub optimal speed as for the majority of the shifts would be of minimal sizes as well as re-reading many already read characters.

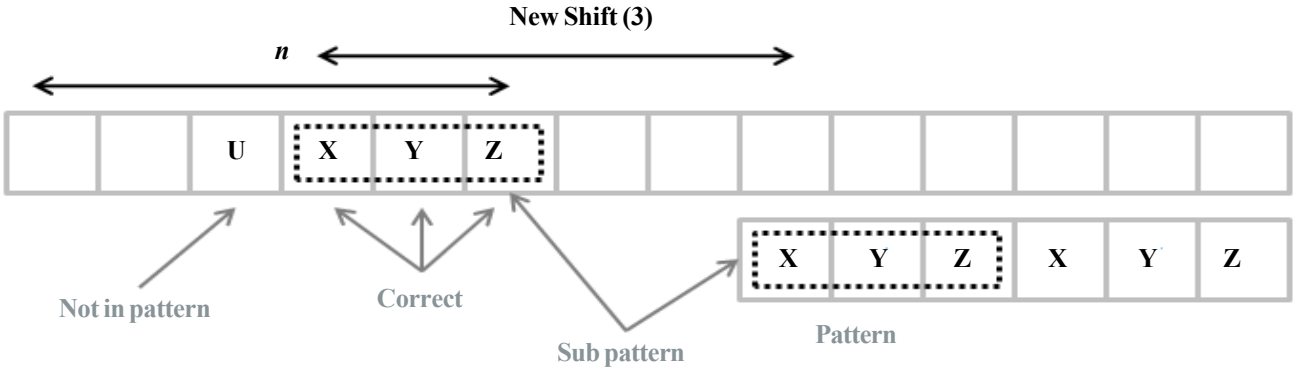

Figure 2. Good Suffix Shift

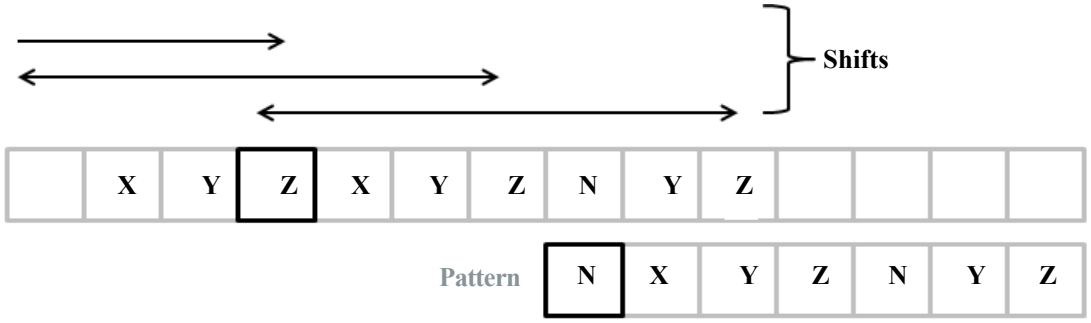

Figure 3. Repeat Reads, No Match

However in the pursuit of having a very simple and low memory cost algorithm, the natural tradeoff is often to do more calculations. The linear nature of the BM algorithm when checking characters from right to left, one by one could present a case where it could verify all but the first character to be correct resulting in an unassay delay. A more severe example is when a pattern appears to be very similar to the source text; this could lead to small shifts, the reading of character multiple times and still no match, where if there was a memory between shifts, multiple reads could be avoided. Figure 3 depicts the case where the character *Z* highlighted is read three times and is the last character to be verified before giving a negative outcome; had there been some form of memory between shifts the algorithm would see that the sequence *XYZXYZ* does not exist in the pattern.

Current works have consistently improved their performances through small increments, either by reducing the number of comparisons made, or by taking advantage of recent improvements in technology such as vector optimized instruction sets (CPU) or Graphical Processing Units [3,4,5,6]. Given past works to optimize the string searching problem, a remaining path for further improvement is to reduce the computational load when searching through a source text. The reasoning behind this approach is that as a source text becomes large, the computational load (thus search times) also increases in direct proportion. Computations are often repeated many times in a search and so the main challenge is to shift all of the computational costing into a pre-processing stage, eliminating wasteful in search calculations. Furthermore, given that current mismatch rules (BM/QS/MS) do not compensate for new or reoccurring false sub matches and often recheck already read characters; the aim of MPS is to incorporate such rules into the pre-processing stage to further reduce the reading and computations of already processed characters.

3. Algorithm

The key component for the MPS is the creation of the look up table, the majority of the algorithm exists here as it removes all complex instructions (character comparison/ shift calculations) from the search process. The pre-processing exploits the nature of character distribution within the pattern itself (avoiding pre-processing of data) such that it can produce a table that can:

- Give the optimal and safe right shift
- Give the optimal mismatch when shifting left
- Detect re-occurring patterns within the pattern
- Detect the start of a new pattern (during right shift)

The aim is to reduce the number of unnecessary reading of characters whilst maximizing the right shifting. As per the standard practice the MPS confirms its matches from right to left and adopts the sliding window convention. MPS also incorporates the frequencies of characters within the search pattern, so as to give priority to search the least frequent characters to optimize the probability of producing a mismatch earlier in the verification stages (reducing overall reads). MPS furthermore incorporates the concept of memory between shifts such that MPS will not re-read characters more than once and avoid reading non-existing patterns (Figure 3) via its lookup table.

Avoiding the trivial cases (last character match or nonexistent character in the pattern alphabet), Figure 4 depicts the case when an initial match ($S_{[0]}$) exists well within the pattern; presenting a section to the left of this match ($S_{[-]}$) and to the right ($S_{[+]}$). It is general convention to check $S_{[+]}$ before checking $S_{[-]}$ and so the MPS consist of two parts, right shifting and left shifting respectively.

3.1 Right Shifting

In checking $S_{[+]}$, the MPS employs a maximal shift mechanism that will discover the starting of a new match. Figure 5 depicts the case when the initial character read is '*C*', for the pattern '*GCAGAGAG*', with all the possible starting positions and characters shown after the *S*[0] position. The safest right shift amount is then selected represented by the furthest shift possible; where all possible characters in this new position are unique (default is + 1).

Figure 4. Pattern Segments

In this instance (Figure 5) the safest jump is +1, leading to two possibilities; a '*G*' resulting in a different (new starting) pattern match and the adjustment of the sliding window; or an '*A*' resulting in a continuation to check the remaining $S_{[+]}$. Once a match is found the process is repeated until the last character is matched. Keeping to the example in Figure 5, the total shifts would be +1, +2, +4, +5 representing the characters *A*, *A*, *A* and *G* respectively. Having reached the end the remaining characters are checked with the right most lest frequent checked first, increasing the ability of finding a mismatch quicker.

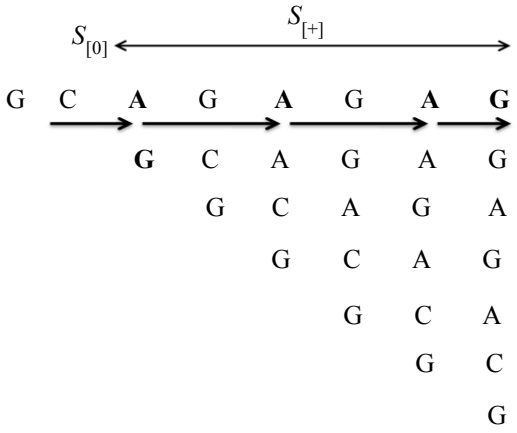

Figure 5. $S_{[+]}$ checking

If all of $S_{[+]}$ are matches then the remaining $S_{[-]}$ can be checked. In the case of any mismatch $S_{[0]}$ will be shifted to a new initial starting point shown below (1), where $S_{[n]}$ represents the last right most shift and Pn represents the pattern length. Due to this safe shifting we eliminate the case of verifying a pattern that does not exit given the characters that have already been read are considered/ known between shifts.

$$S_{[0]_{new}} = (S_{[n]} - S_{[0]}) + S_{[0]} + Pn \quad (1)$$

3.2 Left Shifting

When checking $S_{[-]}$, In order to check efficiently for repetitive patterns within the pattern, we limit our left shifts so that we discover any prior sub pattern matches as soon as possible. In Figure 6 we are presented with an already checked $S_{[+]}$ of '*AG*', however this pattern occurs 3 times in the search pattern. The MPS eliminates any wasteful reads by assuming that $S_{[+]}$ is actually the first occurrence in the pattern and calculates the maximal shift left of the least frequent character. In this instance a shift of −1 will reveal a '*C*' match; if true this verifies that the checked $S_{[+]}$ is in fact the first occurrence of this sub pattern and so an adjustment to the sliding window (new $S_{[0]}$, bigger $S_{[+]}$ which needs to be checked first). If a '*G*' match was made we are still left with the situation that is could be either remaining occurrences of '*AG*'. Another left shift is calculated, again for a '*C*' match position (−3), this leaves us with only 2 possibilities a '*C*' or a '*G*' differentiating which occurrence we are looking at.

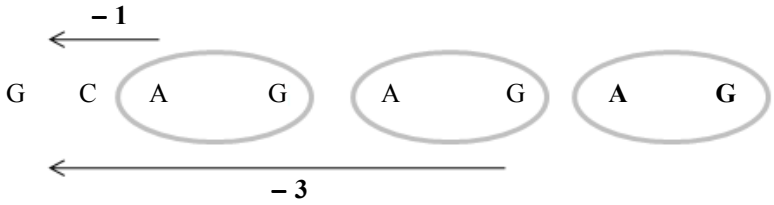

Figure 6. $S_{[-]}$ checking

Once we have eliminated any doubt of sub pattern occurrences the remaining characters will be checked in order of rightmost least frequent. In the case of a mismatch we will shift *S*[0] in the same way as a discovered mismatch when right shifting (equation (1)).

3.3 Table complexity

Once we have calculated our critical paths we are presented with a table that factors in maximal shifts and sub pattern reoccurrence. Keeping with the '*GCAGAGAG*' pattern Figure 7 shows the trie extracted from the table when S[0] is an '*A*' and depicts the shifts to be carried also with their expected characters. As can be seen there are three occurrences of '*A*' in our pattern, hence there are only three critical paths in our trie. Any deviation from this tree will result in new *S*[0] which dictates the next trie to operate on; similar tries exist for each character in the pattern alphabet and so the number of critical paths is equal to the length of the pattern. For example, if after on the second shift we do not get a '*G*' or '*C*' but in fact we actually read an '*A*', we still stay on this trie but go back to the starting '*A*'.

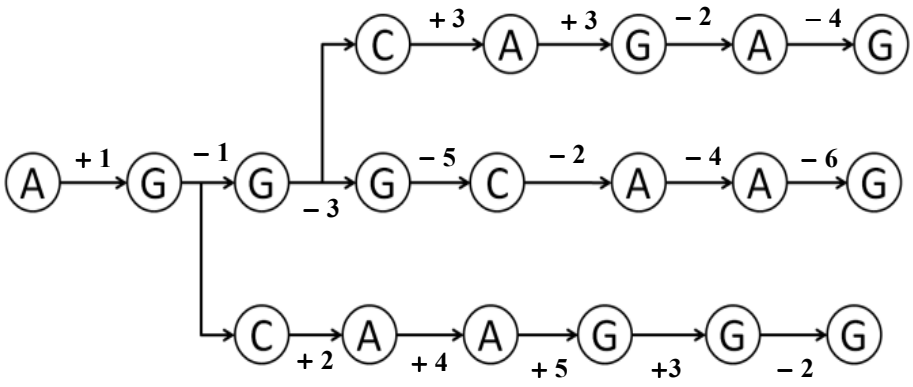

Figure 7. Pattern Search Trie

3.4 Data Structure

Due to the naive initial implementation to provide a proof of concept (hence the absence of an algorithm outline); our implemented data structure consisted of arrays of length 128, such that the literal read ASCII values of the characters are used as an index value. These arrays consisted of object which contained variables relating to shift values, pointer to the next location within the table and a flag representing a match (5 bytes in total). In order to accommodate all the desired features of our table, our data structure took its form as a 3D array, with dimensions $Pn \times Pn \times 128$ (Where Pn is the pattern length). So the memory requirement to implement this table can be calculated (2):

$$\text{Memory}_{(\text{bytes})} = (Pn)^2 \times 128 \times 5_{(\text{bytes})} \quad (1)$$

With this naive implementation there is a need to strike a balance between pattern length and memory available, however given that modern computers have a memory capacity of a few GB, even a pattern of 1000 character coupled with our naive implementation would require 640 MB of RAM. Even though data latencies from a systems ram is significantly slower that the CPU cache, our experiments show that the MPS still out performs the existing methods that have been optimized for CPU cache operation.

4. Experimental Results

As a proof of concept, a naive java implementation of the MPS is tested against reference implementations (java) of the Maximal Shift (MS) [9], Quick Search (QS) and the Reverse Colussi algorithms (RC) [8]. Both the MS and QS algorithms are generally seen as the accepted de facto standard bench mark of the classical algorithms. The RC [10] was chosen because it is shown to be the fastest algorithm that performs consistently across varying pattern lengths, alphabet size and the different data source types [13]. Further in support of RC, it also has an element of preprocessing and is of a similar type of algorithm (character based and not automata / bit parallel), presenting a better and more fair comparison with MPS. All experiments were carried out on a Linux machine with an Intel Core2Duo 1.66GHz (T5450) processor with 4GB of RAM running the Java 1.6 virtual machine.

4.1 The Experiment

Due to the statistical nature of generating the look up table, binary searches are omitted, as the probabilities would converge to equality. Two experiments were carried out; one to measure the performance on a small alphabet (DNA) and another on a larger alphabet (English Text). The results were produced from an average of one hundred runs. In both experiments the source text was kept the same and the pattern text was slowly incremented from 10-300 characters long. The pattern text was initially of length 300; with subset patterns created from this to produce the varying length search patterns. Once a pattern was truncated to the desired size the search would be repeated 100 times, producing an average that would minimize JIT compilation variance. Furthermore, rather than repeat the experiments with a variety of search patterns of the same length, which would help mask anomalies of an algorithm detecting false positives, the use of a single pattern should amplify any detection of false positives and its impact on search times.

The source text for the English plain text was the combined five texts of Adam Smith's '*Wealth of Nations*' [11] which provided a source consisting of 2.2 Million characters; while the source of the DNA text came from the Fruit Fly [12] which consisted of

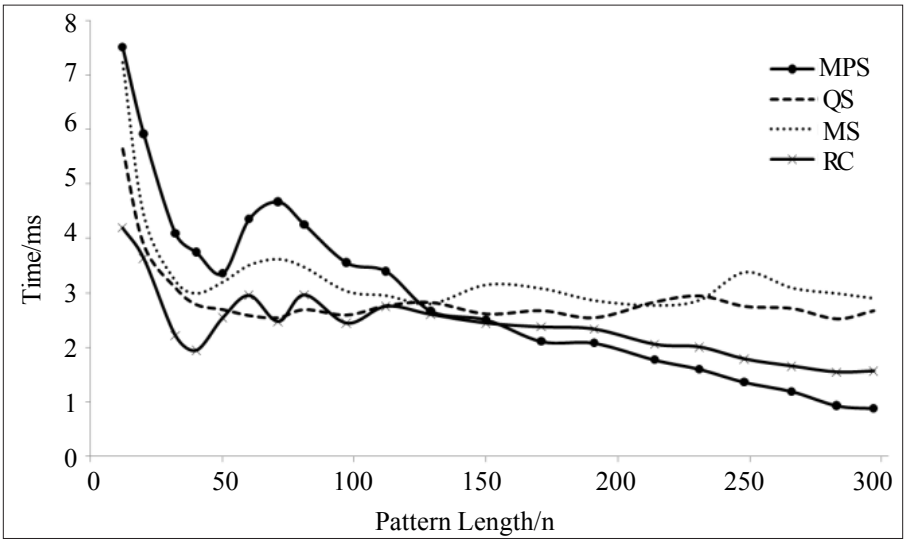

Figure 8. English Text Search Results

56.7 Million characters. Both examples were chosen to provide real life performance metrics, since when dealing with statistical variations, randomly generated data would not reflect real life performances. Since our viewpoint is that most searches are repeated and that this current implementation is a naive one; the results are presented in two stages, pattern (pre-) processing and the searching stage. This will provide the best platform for comparison in evaluating the concepts of the MPS algorithm. Furthermore as the exact string search problem is an embarrassingly parallel problem, we have extrapolated the obtained search results under the assumption of linear performance for growing sizes of source text.

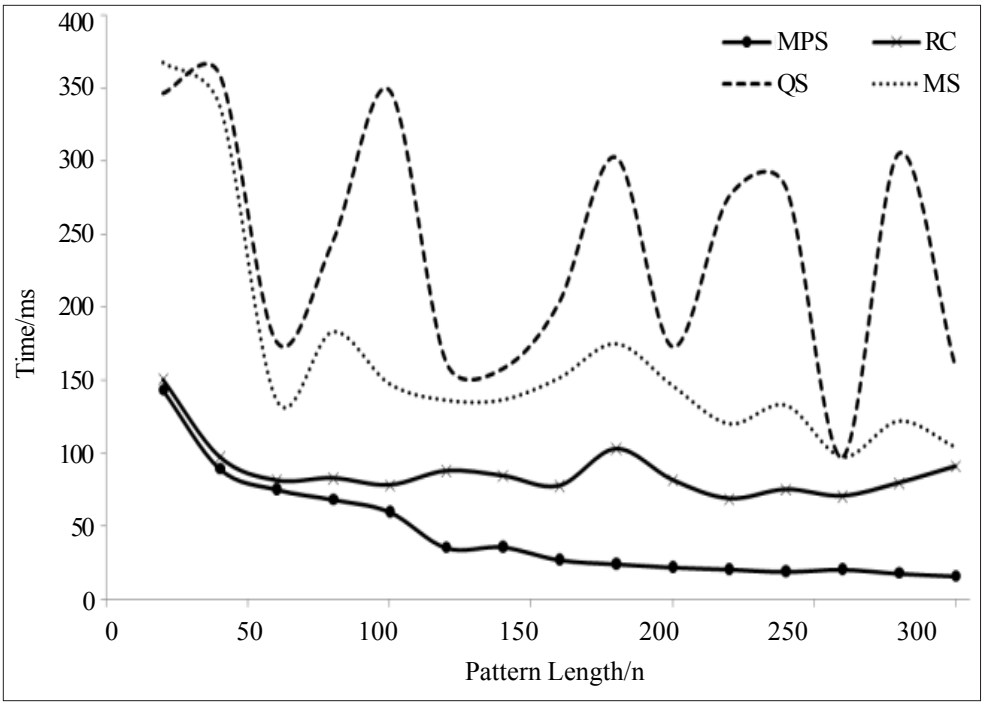

Figure 9. DNA Text Search Results

4.2 Results

The searching results from our first experiment (Figure 8) shows that the MPS performs slightly slower when searching though the English source text. However when the pattern length gets sufficiently large (above 150 characters) the MPS start to execute

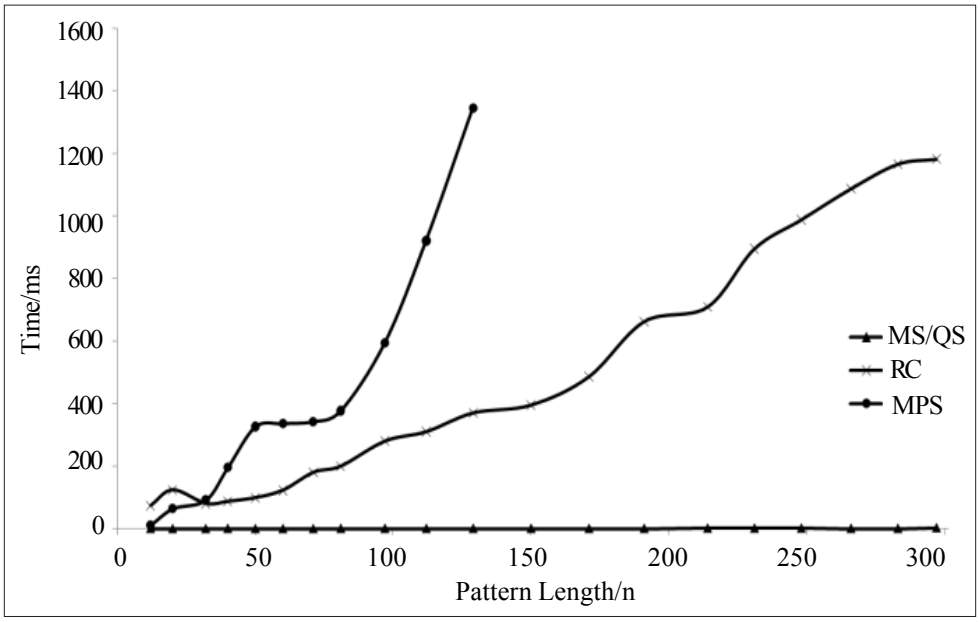

Figure 10. English Pattern Pre-processing Results

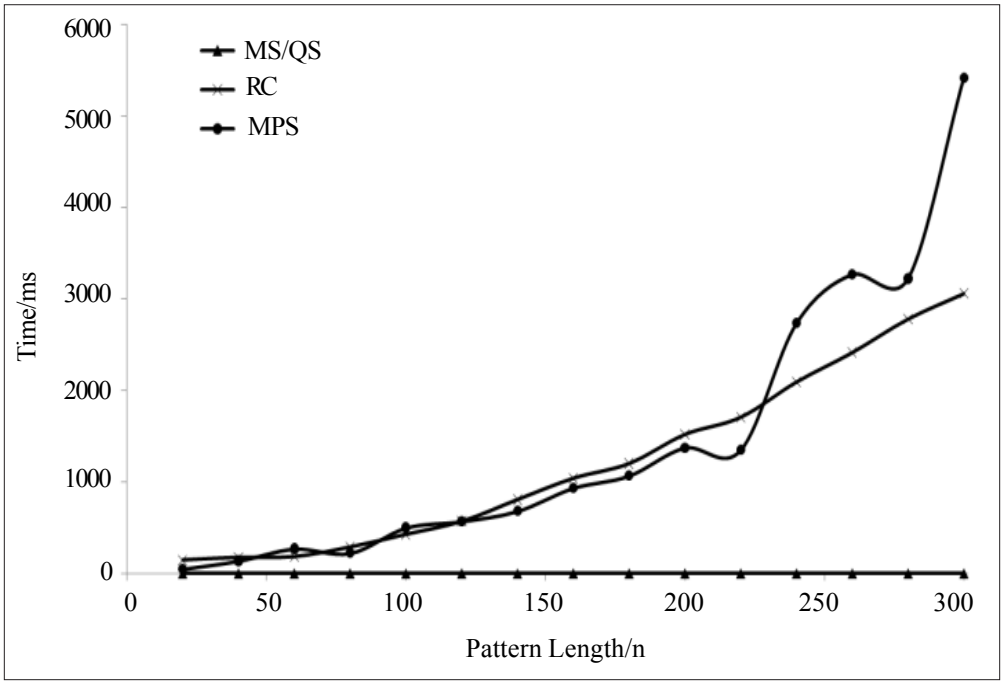

Figure 11. DNA Pattern Pre-processing Results

faster and in a linear fashion. Results show the MPS to be on average 2-3 times faster than the tested methods and in the case of a pattern 300 characters long it was 3.3, 3.1 and 1.7 times faster than the MS, QS and RC respectively. Similarly, Figure 9 shows the results from the second experiment using the DNA source text. The new method is consistently faster than all of the tested algorithms, irrespective of pattern length. On average the MPS is 2- 6 times faster than tested methods providing a linear decline in run times as the pattern grew. In the case for patterns of length 300 characters the MPS was 6.5, 9.9 and 5.7 times faster than the MS, QS and RC respectively.

As expected, the naive implementation caused the pre-processing times of the patterns to be excessive, as shown in Figures 10

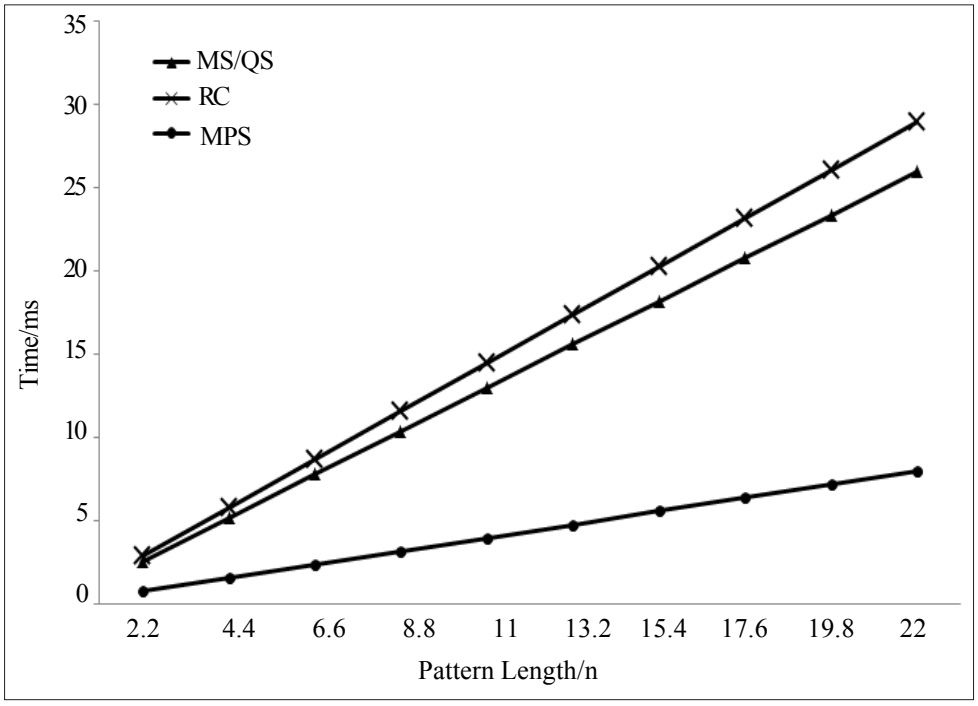

Figure 12. Extrapolated English Text Search Results

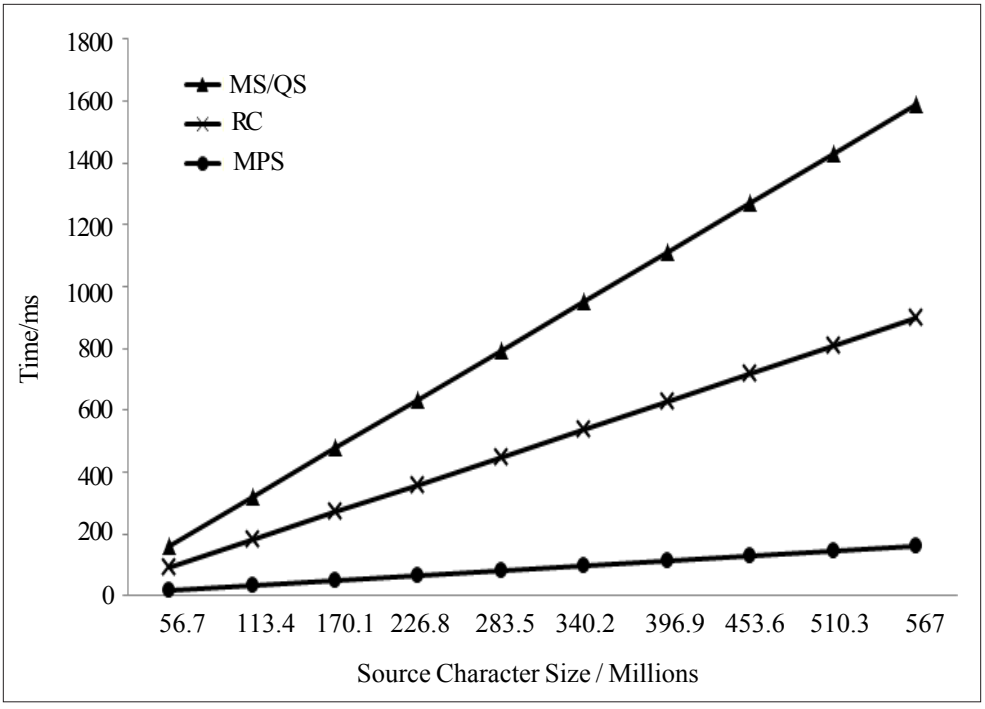

Figure 13. Extrapolated DNA Text Search Results

and 11 (English and DNA source respectively). These results were presented here for completion, due to the similarity of both the MS and QS algorithms pre-processing results, a single line is used to represent both. Figure 11 does however, highlight that the naive implementation, when dealing with the nature of DNA, (small alphabet with long very similar looking sub strings) still performs very closely to the RC algorithm. This suggests, given optimization pre-processing of this new method should easily fall between the RC and MS/QS algorithms.

The assumption of linearity in performance should hold true as exacting matching is an embarrassingly parallel problem, thus an accurate extrapolation. Searches for smaller pattern lengths produce similar linear results, in the case for 20 character lengths for English - all the algorithms perform in a similar manor, for DNA MPS/RC performed identically beating MS/QS by a factor of 2 (Figures are omitted for clarity).

Figures 12 and 13 depicted extrapolated results for a search pattern of length 297/ 300 respectively. From these extrapolated results it is quite clear that the MPS algorithm is a very good algorithm for long search patterns and is extremely suited for short alphabets such as DNA. If a complete human genome of 3 billion base pairs (6 billion characters) is take into consideration extrapolated search times would take 19080ms (MS/QS), 10800ms (RC), 1908ms (MPS), which will easily consolidate the expense of the naive preprocessing costs.

5. Discussion

In summary the results show that performance of all algorithms vary differently depending on the source being used and statistical distribution of characters within the search pattern; verified by the large variations in search times for the QS algorithm on DNA source, re-checking many false positives (seen as oscillation in Figure 9) and the shallow variations for an English source where false positive matches are less likely to occur (Figure 8). These results hint that the tested algorithms emit characteristics that show best performances when the alphabet is large and the patterns are short. A fundamental concept in English is that many repeated words/ characters are unlikely; hence these algorithms for the majority of the search phase are performing maximal shifts, as seen in Figure 8 by the leveling off in runtimes after the pattern exceeds 100 characters.

The search results generally show a leveling off in performance for all tested methods, with the MPS giving a linear decay in runtimes (since statistical distribution of characters become more significant). Extrapolating from these results one could assume that as the pattern length grows even further, speed ups observed will increase significantly as the costing of calculating/ actual shifts/ comparisons become the dominate factor (RC shows signs of this in Figure 8 after the 250 character length) and increase their search times. Given that bacterial generally as a sequence length of +1000 characters, it is apparent that the MPS would be an ideal candidate for this situation.

Another aspect of reducing the search phase to a simple read operation is that this method could in concept work on slower processors without significant impact to performance, allowing for patterns to be processed on a powerful server and out sourcing the search phase to energy efficient processors (at no loss in speed). Furthermore given the low demands of MPS, it could be possible to run n (n = number of cores in a CPU) concurrent searches on a CPU, producing near linear scalability – decreasing MPS search times by a factor of n.

As shown in equation (2), memory requirements were calculated independent of the actual alphabet size (fixed to 128); this was due to its naive implementation. If for example in the case of DNA and a modern processor with 6 Mb cache, with an optimized data structure could in theory fit a table for a pattern size of 500 characters long in this cache size and significantly reduce the execution time of the MPS even further, by removing table look up latencies caused by the system memory.

6. Conclusion and future directions

In this article a new method for exact string matching (MPS) is presented which places emphasis on the pre-processing of the search pattern to reduce the searching phase to a simple look up table problem. MPS provides the proof of concept that incorporating memory between shifts and the use of character distribution within the search pattern does produce a significant improvement.

Experimental results show that the new method is very fast for searches on a small alphabet size (DNA) when compared to the existing algorithms. The new method is also faster than existing works when searching English plain text as the pattern becomes greater than 150 characters. This method uses statistical distribution of characters in the search pattern to optimize its search look up table; a future aim is to optimize the pre-processing stage (runtime and memory requirements), implement a more efficient data structure and implement the code in the de facto standard C language to perform a much greater/ wider accurate comparison with many modern search algorithms, as well as specific ones such as BLAST.

As a consequence of shifting all the calculations to the execution of the search phase, the actual performance of MPS would be limited in the majority, to memory/ disk access times. Based on the previous assumption to be true, this would make MPS largely independent of CPU type allowing for the use of low power cost effective technologies such as the Intel MIC architecture [14] or mobile processors such as the ARM processor [15] paving the way for greener computing.

References

[1] Boyer, R. S., Moore, J. S. (1977). A fast string searching algorithm, *Comm. ACM*, 20 (10) 762–772, Oct.

[2] Hume, A., Sunday, D. M. (1991). Fast string searching, *Software Practice & Experience*, 21 (11) 1221–1248, Nov.

[3] Lecroq, T. (2007). Fast exact string matching algorithms, *Information processing letters*, 102 (6) 229–235, May.

[4] Durian, B., Holubb, J., Peltola, H., Tarhioc, J. (2010). Improving practical exact string matching, *Information Processing Letters*, 110 (4) 148–152, Jan.

[5] Kouzinopoulos, C., Margaritis, K. (2009). String matching on a multicore GPU using CUDA, IEEE panhellenic conference on informatics, Corfu, Greece, p.14-18.

[6] Navarro, G., Raffinot, M. (2000). Fast and flexible string matching by combining bit-parallelism and suffix automata, *ACM J. Exp. Algorithmics*, 5 (4), Dec.

[7] Berman, F., Fox, G., Hey, T., Trefethen, A. (2003). Grid Computing, Wiley & Sons Ltd.

[8] Kiourtzoglou, B. (2010, Sept). Java Best Practices – String performance and Exact String Matching Available: http:// www.javacodegeeks.com/2010/09/string-performance-exact-string.html.

[9] Sunday, D. M. (1990). A very fast substring search algorithm, *Comm. ACM*, 33 (8) 132-142, Aug.

[10] Colussi, L. Fastest pattern matching in strings, *Journal of Algorithms*, 16 (2) 163-189, Mar.

[11] Smith, a. An Inquiry into the Nature and Causes of the Wealth of Nations, Methuen and Co. Ltd , 1904.

[12] BDGP. (2012, Oct). Berkeley Drosophila Genome Project, Available: http://www.fruitfly.org/.

[13] Faro, S., Lecroq, T. (2010). The Exact String Matching Problem: a Comprehensive Experimental Evaluation, CoRR, abs/1012.2547, Oct .

[14] Intel. (2013, Jul). Intel MIC architecture, Available: http://www.intel.com/content/ www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html.

[15] ARM. (2013, Jul). ARM processor, Available: http:// www.arm.com/products /processors/ instruction-set-architectures/index.php.

[16] Dean, J., Ghemawat, S. (2008). Map Reduce: Simplified Data Processing on Large Clusters, *Comm. ACM*, 51 (1) 107-113, Jan.

[17] Faniel, I., Zimmerman, A. (2011). Beyond the Data Deluge: A Research Agenda for Large-Scale Data Sharing and Reuse, The *International Journal of Digital Curation*, 6 (1) 58-69.

[18] IBM. (2013, Aug). What is big data? Available: http://www.ibm.com/big-data/us/en/.

[19] Choi, H., Varian, H. (2012). Predicting the present with google trends, Economic Record, 88, p. 2-9.