Aldwairi *et al. EURASIP Journal on Information Security* (2017) 2017:9
DOI 10.1186/s13635-017-0062-7

EURASIP Journal on Information Security

RESEARCH Open Access

Pattern matching of signature-based IDS using Myers algorithm under MapReduce framework

Monther Aldwairi[1,2*], Ansam M. Abu-Dalo[1] and Moath Jarrah[1]

Abstract
The rapid increase in wired Internet speed and the constant growth in the number of attacks make network protection a challenge. Intrusion detection systems (IDSs) play a crucial role in discovering suspicious activities and also in preventing their harmful impact. Existing signature-based IDSs have significant overheads in terms of execution time and memory usage mainly due to the pattern matching operation. Therefore, there is a need to design an efficient system to reduce overhead. This research intends to accelerate the pattern matching operation through parallelizing a matching algorithm on a multi-core CPU. In this paper, we parallelize a bit-vector algorithm, Myers algorithm, on a multi-core CPU under the MapReduce framework. On average, we achieve four times speedup using our multi-core implementations when compared to the serial version. Additionally, we use two implementations of MapReduce to parallelize the Myers algorithm using Phoenix++ and MAPCG. Our MapReduce parallel implementations of the Myers algorithm are compared with an earlier message passing interface (MPI)-based parallel implementation of the algorithm. The results show 1.3 and 1.7 times improvement for Phoenix++ and MAPCG MapReduce implementations over MPI respectively.

Keywords: Information security, Intrusion detection systems, Signature-based, Pattern matching, MapReduce

1 Introduction

Digital devices and services are essential to ease our personal and business lives. However, as connectivity between different services increases, the number of possible threats and their suspicious behaviors grow accordingly. Therefore, an efficient detection process of attacks is needed to ensure the availability of the different services [1].

The most common technique used to inspect packets' payloads for malicious threats is intrusion detection. Performing packet inspection requires matching attack signatures with each captured packet. The sheer volume of new attack signatures coupled with the rapid increase in links' speeds poses a challenge in terms of processing time, area, and power. Aldwairi et al. stated that the signature matching operation is the most CPU-intensive task [2]. For example, Snort, the most commonly used open-source IDS, includes a large rule set containing more than 10,000 signatures [3]. Consequently, providing an efficient matching algorithm to improve the IDS operations at low cost remains an open research problem.

Several researchers proposed enhanced IDS implementations either using software or hardware [4–11]. Software-based IDS applies a pattern matching algorithm to scan the system files for attack and malware signatures. This operation results in decreasing the system performance by adding significant overheads on the CPU, memory, and power. Hence, software solution's efficiency still needs to be improved. Specialized hardware devices can use parallelism and provide large enough memory to store the growing data. They can satisfy the high performance requirements of the computationally intensive pattern matching task. However, they are difficult to program, provide poor flexibility, and have a high cost.

MapReduce framework provides parallelism and data distribution among multiple processors [12]. It was

*Correspondence: munzer@just.edu.jo; monther.aldwairi@zu.ac.ae
[1]Computer and Information Technology Faculty, Jordan University of Science and Technology, Irbid 22110, Jordan
[2]College of Technological Innovation, Zayed University, P.O. Box 144534 Abu Dhabi, United Arab Emirates

designed to hide the complexity of the programming and to facilitate the development process of data-driven applications. As a result, MapReduce can be used to process the large number of signatures in an efficient way. In this paper, we propose a MapReduce implementation of the Myers algorithm. Myers algorithm is the fastest approximate string matching algorithm available [13]. We use two different implementations of MapReduce to parallelize the algorithm: Phoenix++ and MAPCG. We compare our implementations with a popular message passing interface (MPI) parallel implementation. Our MapReduce implementation is faster than the MPI implementation by an order of magnitude.

The rest of this paper is organized as follows. Section 2 provides the necessary background. Section 3 provides a brief review of the related work. Section 4 describes our two implementations in details. The evaluation metrics and experimental results are presented and discussed in Section 5. Section 6 concludes the paper and presents suggestions for future work.

2 Background

This section provides the necessary background to understand the problem in hand. Sections 2.1 and 2.2 shed the light on intrusion detection systems, Snort rules, and pattern matching. Section 2.3 introduces Myers algorithm and dynamic programming. Section 2.4 briefly reviews the related parallel programming techniques, and Section 2.5 presents the MapReduce framework and the two different implementations.

2.1 Intrusion detection systems

The noticeable growth in the number of diverse attacks on today's network and computer systems brings a significant attention to security. IDSs play a major role to protect the networks against those threats. These systems use a deep packet inspection process to monitor network traffic against intrusions and abnormal activities. Based on the detection methodology, intrusion detection systems can be classified into two groups: signature-based IDS and anomaly-based IDS. A signature-based IDS inspects network traffic using the signatures of predefined attacks that are stored in a database. The signature matching process is considered the most intensive task in terms of CPU time and power. Anomaly-based IDS on the other hand finds malicious activities by measuring the traffic deviation from the expected baseline. It relies on machine learning and artificial intelligence (AI) to classify the network traffic into normal or abnormal [2].

Snort is a popular open-source signature-based IDS. It is widely used by researchers and production environments [3]. Snort has a large database of rules, which covers known attacks. As new attacks are discovered, Snort's database increases in size. Unfortunately, experts must write the rules manually. Figure 1 shows the basic elements of a Snort's rule. The rule consists of two parts: the header and the options. The rule's header consists of the action to be taken, the protocol, and the packet's flow information. The header takes the form [Action] [Source IP Address] [Source Port number] → [Destination IP Address] [Destination Port number]. The options part contains more than 24 different keywords such as alert messages to be logged if a match occurs. In addition, the options part contains the content which is the most important keyword of a rule. Content holds the string or the attack's signature to be matched with the packet's payload. For example, the Snort rule in Fig. 1 can be read as follows: Alert the system administrator by firing the message "WEB-MISC phf attempt," if any incoming TCP packet flows to any address with port number 80 and contains the string: "/cgi-bin/phf". Additionally, the rule defines the risk level of the attack which is 10 as shown by Fig. 1.

2.2 Pattern matching

In order to find the suspicious packets, most of IDSs employ a pattern matching algorithm. The algorithm checks the presence of a signature in the incoming packet sequence and outputs the location of the string within the packet. The algorithm must be fast enough to detect the malicious behavior, and it must be scalable in order to meet the increase in both the number of signatures and the link speed.

String matching algorithms can be categorized into single and multiple pattern matching algorithms. In the single pattern matching, one pattern is matched against the entire text at a time [14]. In contrast, the multiple pattern matching approach compares the text sequence against all signatures all at once [15]. Obviously, the multiple matching approach is a better choice for intrusion detection to avoid sweeping the packet many times. However, it consumes more memory and requires a pre-processing phase to program the patterns before matching can commence.

2.3 Myers algorithm

The Myers algorithm is an approximate string matching algorithm. An approximate matching algorithm matches

```
alert tcp any any → any 80 (msg:"WEB-MISC phf attempt"; \content:"/cgi-bin/phf"; priority:10;)
```

Fig. 1 Snort rule sample

a large text of length n with a short pattern p of length m allowing up to k differences, where k is a chosen threshold error [16]. The Myers algorithm relies on a simple dynamic programming (DP) concept [17]. It uses recursive formulas and simple bit operations to compute the edit distance between the text and patterns to find the equalities or differences [18]. The edit distance between two strings is expressed as the minimum edit operations required to transform a text *t1* to another text *t2* or vice versa. Commonly, there are three typical variations of edit distance. The first form is called the Hamming distance [19]. It computes the number of positions in the text that has different characters, i.e., how many characters are needed to convert a text *t1* to another text *t2*. The compared texts or strings must be of the same length. The second form is called the Levenshtein distance, which does not have any restriction over the text size [20]. The edit distance is the minimum number of edit operations: insertion, deletion, and substitution, which are needed to convert two strings into each other. The third one is the Damerau edit distance. It allows the transposition of two adjacent characters to complete the conversion between the two strings [21]. Figure 2 shows examples of the edit operations.

The Myers algorithm uses the Levenshtein distance to compute the matches. It considers two strings similar if the edit distance (*ed*) between the two strings (A, B) is less than or equal to a predefined threshold (k) ($ed\ (A, B) <= k$).

The formal approach to solve the problem of approximate string matching and to find the minimum edit distance is to use dynamic programming. Dynamic programming is an approach which uses a recursive formula to compute new values based on a prior knowledge of previous values. For two strings of lengths m and n, a matrix of size $m \times n$ is filled column-wise or row-wise, respectively, with distances between the strings. The patterns are arranged vertically and the packet is arranged horizontally. Initially, the matrix cells are initialized as follows. Cells $C[0, j] = 0$ and cells $C[i, 0] = i$, and next, the matrix is expanded according to the recursive formulas in Eq. (1) [22].

$$C[i, j] = min \begin{cases} C[i-1, j] + 1 \\ C[i, j-1] + 1 \\ C[i-1, j-1] + \delta_{i,j} \end{cases} \quad (1)$$

Where

$$\delta_{i,j} = \begin{cases} 0, if p_i = t_i \\ 1, otherwise \end{cases}$$

Each cell of the matrix, $C[i, j]$, represents the edit distance between the two strings *P[1→i], T[1→j]* ending at positions i, j. The addition of one in Eq. 1 is the penalty if a match does not exist. This case represents a substitution.

As an example, we explain how to find an occurrence of the pattern “execut” in the text “feexecutingprogram” with difference k equals to 0. This case represents an exact match. The dynamic programming (DP) matrix is filled as shown in Table 1.

The last row in Table 1 indicates the number of differences between the two strings in that position. The zero in the last row (underlined and bold) is a solution to the problem $C[m, j] <= k$ (occurrences with k or less distance value). A value of zero in this row indicates an exact match; a value of one indicates that there is at least one transformation required to change the strings into each other and so on. The Myers algorithm encodes the DP matrix and uses a bit-vector to process each column of the matrix [23]. Bit parallelism is a mechanism that takes advantage of bit-level parallelism in a computer word at the hardware level. This is efficient because the processor can process the entire computer word in one memory cycle. Each cell in the DP matrix differs only by −1, 0, or +1 from its neighbor cells (upper, left, and upper left). Based on this observation, Myers algorithm re-encodes the DP matrix and finds the differences between the adjacent cells in each successive row or column using the formulas in Eqs. (2) and (3) [13].

$$\Delta h_{i,j} = C_{i,j} - C_{i,j-1} \quad (2)$$

$$\Delta v_{i,j} = C_{i,j} - C_{i-1,j} \quad (3)$$

In addition, each cell in the matrix relies on three possible states for the horizontal and vertical deltas {−1, 0, and +1} and on two states for the diagonal deltas {0 and 1}. Since the main goal is to represent the entire matrix in a bit-vector format, clearly the diagonal deltas are in the required format {0, 1}. However, the vertical and horizontal deltas are not in the appropriate format because they have a −1 value. In order to solve this problem, Myers algorithm expresses each value with a vector. That is, one vector (HP) represents the positive values and

a) ABC	b) ABC	d) ABC	c) ABC
ABCD	AC	ADC	ACB

Fig. 2 Edit operations. **a** Insertion. **b** Deletion. **c** Substitution. **d** Transposition

Table 1 DP matrix

		F	E	E	X	E	C	U	T	I	N	G	P	R	O	G	R	A	M
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
E	1	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1
X	2	2	1	1	0	1	1	2	2	2	2	2	2	2	2	2	2	2	2
E	3	3	2	1	1	0	1	2	3	3	3	3	3	3	3	3	3	3	3
C	4	4	3	2	2	1	0	1	2	3	4	4	4	4	4	4	4	4	4
U	5	5	4	3	3	2	1	0	1	2	3	4	5	5	5	5	5	5	5
T	6	6	5	4	4	3	2	1	**0**	1	2	3	4	5	6	6	6	6	6

The zero in the last row (underlined and bold) is a solution to the problem $C[m,j] <= k$

the second vector (HN) represents the negative values. HP represents the vector when $\Delta h = +1$, and HN represents the vector when $\Delta h = -1$. Finally, Myers produces five vectors that are denoted by VP, VN, HP, HN, and D0 to compute the distance of each cell. These vectors have the same length of 32 or 64 depending on the machine word size.

Algorithm 1 shows the Myers algorithm. HP and HN represent the horizontal deltas, VP and VN represent the vertical deltas, and D0 represents the diagonal delta. In addition, X represents the current horizontal and vertical state of a column. The search phase in this code computes the vectors, scoring, and the matching results of the approximate matching problem. For example, if $m = 6$ and the horizontal positive vector value equals to 100,000 in line 9 of the algorithm, the if statement in line 9 evaluates to true and the score is increased by one.

Each cell in the matrix can be represented in one bit (either 0 or 1). Therefore, 32 or 64 cells are concatenated in one-bit vector. Additionally, the Myers formulates the required operations to compute the next complete column from its previous one using the minimum number of bit operations performed on the bit-vectors. If the vertical text (the pattern in this case) exceeds the machine word length, several bit-vectors are concatenated to represent the matrix column (called a block) [13]. Furthermore, the patterns are encoded using bit-vectors, and each of length equals to the pattern length. A bit, in this vector, is set to one if the character of the pattern matches one of the alphabet characters. For example, consider the alphabets $\sum$ composed of {A, B, C, D}, and the pattern P = ABDCCABD, the bit mask for the pattern will be:

- B [A] = 00100001
- B [B] = 01000010
- B [C] = 00011000
- B [D] = 10000100

2.4 Parallel programming

Parallel programming is a technique in which many computations are performed concurrently. Parallel computation divides a big task into smaller sub-tasks to be executed simultaneously. Parallelism can utilize multi-core processors in a single machine or multi-processors in a cluster of machines.

The parallel execution on a multi-core or a cluster can take many forms. It can be categorized into bit-parallelism, data parallelism, or task/function parallelism. The focus of bit-parallelism is to minimize the count of instructions to execute an operation. This can be done by increasing the processor word size.

Algorithm 1: Myers Algorithm

```
   begin
       foreach $C \in \Sigma$   /* Preprocessing phase */
       do
           $B[C] = 0^m$
       for $j \in [1, m]$
       do
           $B[P_j] = B[P_j] \vee 0^{m-j}10^{j-1}$
       $VP = 1^m$; $VN = 0^m$   /* Vectors of size m initialized to ones or zeros */
       $Score = m$
1      foreach $pos \in [1, n]$   /* Searching, Scoring, and output phases */
       do
2          $X = B[t_{pos}] \vee VN$
3          $D0 = ((VP + (X \wedge VP)) \vee VP) \vee X$
4          $HN = VP \wedge D0$
5          $HP = VN \vee \neg(VP \vee D0)$
6          $X = HP << 1$
7          $VN = X \wedge D0$
8          $VP = (HN << 1) \vee \neg(X \vee D0)$
9          if $(HP \wedge 10^{m-1}) \neq 0^m$ then
10             Increment Score
11         if $(HN \wedge 10^{m-1}) \neq 0^m$ then
12             Decrement Score
13         if $(Score <= k)$ then
14             Report occurrence at pos
```

In data parallelism, the data is split into many pieces and is distributed to multiple cores or processes. All processors run the same code simultaneously but on different data piece. This is also known as single instruction multiple data (SIMD) approach. In contrast to data parallelism, task parallelism is a form of parallelism where multiprocessors run different codes or tasks on the same piece of data simultaneously.

2.4.1 MPI

MPI is a standard interface that contains a set of libraries and routines to write a parallel program and distributes it over a cluster of machines or a multi-core processor. There are two basic components which are implemented in each MPI library. The first one deals with the compilation such as *mpicc* [24]. It is a wrapper that links the MPI library and provides an easy operation to set the appropriate paths of both the library and the included files. *mpicc* also passes its argument to the C compiler which is required to run the program. The second tool is an agent which is responsible for executing the code in a distributed environment such as the *mpirun* or *mpiexe*.

2.4.2 Open multi-processing (OpenMP)

MPI is just a standard. It has several implementations such as MVAPICH, Intel MPI, and OpenMP. OpenMP [25] is a high-quality open-source implementation of the latest MPI standard with a superior performance compared to other implementations [26]. It provides a set of application programming interfaces (APIs) that is easy to use. OpenMP supports shared memory in a multi-processor environment, which gives more flexibility to programmers to develop their distributed applications. It is comprised of a set of library routines and compiler directives that allow the master processor to distribute the data and tasks among the processing units.

2.5 MapReduce

MapReduce is a programming model released by Google to handle the processing of large dataset in parallel [27]. The idea behind this framework is to hide the complexity of parallelism from the programmer. Moreover, the framework provides the programmer with a simple API to present the logical perspective of an application.

Recently, MapReduce has been widely used in both academia and industry. MapReduce has become a standard computing platform used by large companies such as Google, Yahoo!, Facebook, and Amazon. Statistics show that Google uses MapReduce framework to process more than 20 petabytes of data per day [27].

2.5.1 MapReduce basic programming model

MapReduce framework consists of two primitive functions defined by the user: Map and Reduce. Additionally, it has a runtime library to automatically manage the parallel computations without the need for user interventions. The library handles data parallelization, fault tolerance, and load balancing.

In MapReduce model, the input and output data take the form of key/value pairs $< key, value >$. Map and Reduce are the main two operations that are applied to the key/value pairs. A large input data is split into chunks of a specified size. For example, Google's implementation partition the data into M pieces each of size 16–64 MB. The Map function takes the input as a series of key/value pairs $< k1, v1 >$ and performs the task assigned by the programmer. The output of the Map function is a series of intermediate key/value pairs $< k2, v2 >$.

The framework performs a shuffle phase in order to group the values of the same key. Afterwards, the intermediate data pairs are sent to the appropriate Reduce function. The Reduce phase takes the combined intermediate values as a key and a list of values and then executes the user-defined Reduce function to produce the final result.

2.5.2 Phoenix MapReduce

Phoenix is a MapReduce implementation introduced by Stanford University [28]. It targets multi-core and multiprocessor systems. Phoenix relies on the same principles of the original MapReduce implementation. It provides a runtime system library and a set of APIs that handle the underlying parallelism issues automatically. Unlike Google's MapReduce, Phoenix uses threads instead of clusters to perform the Map and Reduce tasks. Additionally, it uses shared memory for the purposes of communication.

Phoenix provides two distinct types of APIs: the first set is defined by the user such as Map, Reduce, and Partition. The second set includes the runtime APIs that deal with system initialization and emitting of the intermediate and final output as key/value pairs. Pthreads library is the basic development package of Phoenix runtime. The execution flow of the runtime passes through four basic stages: Split, Map, Reduce, and Merge. At the beginning, the user program initiates a scheduler that controls the creation of Map and Reduce threads. A buffer is used to provide the communication between workers. Data and function arguments, such as the number of workers, the input size, and the function pointers, are passed to the scheduler.

The Map phase splits the input pairs into equal chunks and provides a pointer for each data chunk to be processed by the mappers. The intermediate data that is generated by the mapper is partitioned into units as well and ordered by the partition function to be ready for the reduce phase. The reduce phase does not start until the entire Map task is finished.

The scheduler assigns the tasks to the reducer dynamically. Each unique key with its associated list of values is processed at a reducer node or thread. At the end, the final outputs of the tasks are merged into a buffer and sent back to user program.

2.5.3 MAPCG MapReduce

Nowadays, hardware accelerators are being widely used to perform general-purpose computations. Some researchers have implemented MapReduce on hardware accelerators such as FPGAs and GPUs [29]. These implementations try to exploit the parallelism capabilities of the accelerators to improve the execution time of the MapReduce framework. GPUs have received a great attention due to their high performance capabilities.

MAPCG framework relies on the accelerators context that says "Write once, run anywhere". It was designed to provide a portable source code between CPU and GPU using MapReduce framework [30]. MAPCG's runtime library allows the user to focus on the logical perspective of the algorithm implementation rather than dealing with the side-burdens of parallelism such as load balancing and communication issues. In short, MAPCG was designed to parallelize data-intensive applications on multi-core CPUs and GPUs. It automatically generates a portable code that can schedule the MapReduce tasks on both CPU and GPU. In addition, it implements a dynamic memory allocator for the CPU and GPU that behaves efficiently even when using a massive number of threads. Moreover, a hash table was designed to group the intermediate data on GPUs to enhance the sorting phase of MapReduce keys.

Basically, MAPCG has two main parts, a high-level programming language and the runtime library. The high-level language facilitates and unifies the programming task, while the runtime is responsible of executing the code on different processing units such as the CPU or GPU. As other MapReduce frameworks, the execution begins by splitting the data inputs into chunks and sending them to the Map phase. The function Map is applied to each data chunk and outputs the intermediate data. The Reduce function performs the appropriate computations on the intermediate data to produce the final outputs.

3 Related work

An intrusion detection system that meets the growth of Internet speeds and the increase in attack numbers have become a necessity. Many software and hardware solutions are proposed in the literature to provide the required high performance and throughput rates. Software solutions generally lack the scalability and do not achieve the desired performance. On the other hand, hardware solutions address these issues and additionally provide parallelism capabilities with a high performance. Unfortunately they suffer from low configurability and high cost.

IDS parallelism enables the manipulation of traffic packets simultaneously to reduce the overall processing time. It also may divide the signatures among threads in data parallelism multi-threading techniques. Additionally, it may benefit from task parallelism and distribute the matching tasks among many processing units. There is plenty of work in literature to parallelize IDSs using either data parallelism or function parallelism. Aldwairi and Ekailan proposed a hybrid partitioning multi-threaded algorithm [4]. Aho-Corasick (AC) and Wu-Manber (WM) algorithms [15] were used to implement the system. The authors divided the matching operation over multiple threads. The pattern signatures were partitioned into classes based on their length. The classes had almost equal share of patterns according to the length percentages in the signature database. For the shorter strings, the authors used AC algorithm to perform the matching, while the WM algorithm was used for longer patterns. The authors achieved 33% reduction time over the serial implementation by using four threads.

Kharbutli et al. provided a task and data parallelism implementations of Wu-Manber algorithm [5]. The authors proposed three different approaches to parallelize the algorithm. The first one is the shared position algorithm where several scanning windows were executed at the same time on different processing units. A position variable is shared between the units. The second implementation used the trace distribution algorithm to divide the traffic traces equally between the threads. The final one is a combination of the previous two implementations. These implementations provided excellent performance results. The evaluation results for these implementations achieved two times speedup over the serial one [5].

Su et al. proposed a parallel implementation of the AC algorithm on a multi-core processor [6]. They used a data parallelism technique to improve the performance. The parallel implementation divides the text into n parts, where n equals to the number of forked threads. The overlap concept was used to guarantee the accuracy of the search algorithm. The results of their parallel AC showed at least two times speedup over the serial AC.

Holtz, David, and Timoteo proposed an architecture that collected the Internet traffic from different distributed IDSs residing in different locations of the network. The analysis of the collected data, in terms of matching, correlation, and others, was carried out on a cloud environment using MapReduce framework. The authors conducted several experiments to verify their method. They aimed to distribute and decentralize both the data collection and storage. Their implementation used Snort IDS [7].

Kouzinopoulos and Margaritis implemented Naïve, Kunth-Morris-Pratt, Boyer-Moore-Horspool, and Quick-Search pattern matching algorithms on NVIDIA GTX 280 GPU card [31, 32]. The GPU card contains 30 multi-processors and 240 cores. They were the first to use CUDA to program the GPU to parallelize the algorithms. A speedup of 24 times was achieved when compared with the serial version on a CPU.

Hu et al. tried to optimize the lookup table of Aho-Corasik algorithm in Gnort [8, 33]. The goal was to solve two issues: the increase in the number of signatures and the decrease in performance. Gnort used a modified algorithm, called CSAC, on a GPU card. The authors conducted their experiments using different dataset sizes. In addition, they tested their GPU implementation on different GPU cards: GTS 8800 and Tesla C2050. The authors concluded that as the number of pattern increases, CSAC can still achieve a good performance improvement and reduce the memory usage using compression techniques.

Xu, Zhang, and Fan improved the execution time of the traditional WM algorithm by implementing a parallel version (G-WM) on GPU [9]. They achieved a remarkable speedup of 12 times over the serial implementation. They used a hash table to reduce the number of patterns that may match the text. Additionally, they replaced the original linked list in the original WM with a two-dimensional array to reduce the computational time. The authors divided the large data into pages. The pages can be processed by different threads concurrently, and they are stored as follows. First, the pages are stored in the global memory so threads can access them easily. Second, the pages are stored in a shared memory that is 20 times faster than the global one. Theoretically, the approach of using the shared memory should be faster. However, the paper's results showed the opposite. This is because the data block to be copied to the shared memory is larger in size than the shared memory. In addition, there is a high overhead of copying the data from the global memory to the shared memory. These two factors decreased the performance.

Xu et al. proposed a new bit parallel algorithm impMASM for approximate string matching, called impMASM [10]. The original MASM algorithm required the entire patterns to be of the same length [10]. In impMASM, the researchers addressed this issue and made their new implementation of the algorithm to have no restriction on the pattern length. They concatenated the entire patterns into one single long pattern and stored it in a table. impMASM splits the text into pages of the same size. This pre-processing stage is placed on the CPU side while the matching process is placed on the GPU side. The matching results are sent back to the CPU at the end of execution. impMASM was tested using GeForce 310M card and achieved a speedup of 28 times over the CPU one.

Hung et al. proposed an intrusion detection system that targets several GPU memory architectures [11]. Their approach tried to balance the load among several threads based on a hierarchical hash table. The results showed that the proposed method achieved a throughput of 2.4 Gbit/s while processing the traffic.

4 Parallel implementations of Myers algorithm

This section explains our parallel implementations of the Myers IDS algorithm. Section 4.1 explains previous attempts to parallelize Myers algorithm using MPI. Section 4.2 details our multi-core implementation of Myers algorithm using Phoenix++. Section 4.3 presents our parallel implementation using MAPCG framework.

4.1 MPI implementation

Chibli parallelized Myers algorithm using a multi-processor technique in [34]. The author divided the data between clusters of processors communicating through MPI. To assure correctness, the author used overlapping to make a redundant copy of the data where the cut occurs. Chibli parallelized the DP matrix over the processors based on the fact that the current column computations depend only on its previous one.

Chibli's implementation was tailored to the bioinformatics search problem. He assumed that the data and the signatures are stored locally on each processor's disk and the matrix computation is the part that needs to be parallelized. To evaluate the implementation, a workstation of 13 processors was used to run the experiments. We re-implement Chibli's approach in this paper for comparison purposes. We managed to achieve a significant speedup over Chibli's work using Phoenix and MAPCG.

4.2 Phoenix++ implementation

Phoenix originally implemented a simple string matching application. This program took a decrypted word from a file, encrypted it, and applied a comparison to a list of encrypted words from another file. The comparison process aims to find words that were used to generate the words in the encrypted file. It simply uses *strcmp()* C function to compare the two words. Additionally, only four words are used for evaluation. These four words are hand-coded directly into the code. We modify Phoenix++ (C++ implementation of Phoenix) and replace the string matching algorithm with Myers algorithm [35]. In addition, all the patterns are read offline and stored in an array. Then, the arrays are sent to the *encodePattern()* function to generate a bit mask for each pattern as described in Section 2.

The Map function implements the search algorithm. The main challenge in writing the Map function is to make the search algorithm deal with a distributed data. Original implementation of Myers algorithm reads the data from a text file and compares it against a single keyword. That is, it reads the data and the search commences in a sequential manner. However, in our implementation of the Map function, the data is distributed among many workers and it needs to be searched for each encountered pattern.

Data reading was modified in the original algorithm to meet the requirements of the Map function. Finally, a loop is executed over the entire stored patterns and three functions are applied at each round to find the matches. The three functions are *encodePattern()*, *setupSearch()*, and *search()*.

Our implementation follows the same steps as in the original Phoenix flow. First, the input traffic is read and split into chunks, and then, the Map function is applied to output the total number of matches. In our implementation, we did not need to use the Reduce function since we tailored the Map function to spit out desired output. The Map function prints and counts the matches outputted from the search algorithm (function *search()*).

4.3 MAPCG implementation

String matching is one of the benchmarks used to evaluate MAPCG framework. It simply compares a single pattern that is passed as an argument by the user with a large text file searching for a possible match. Because we plan to use Myers algorithm to match the Snort signatures against a packet traces, we re-implement the string match to fit our needs.

A function is implemented to read the patterns and store them in a vector. Additionally, we use the modified Myers algorithm, the same that was used in Phoenix++, to perform the matching. The Myers algorithm is implemented in the Map function as well.

5 Evaluation and analysis

This section summarizes the experiments that were used to evaluate our implementations. We present the experimental results of both memory usage and execution time. In addition, we compare between the different implementations.

5.1 Experiments' environment

The experiments of the aforementioned implementations are conducted on Intel core i5 2410M machine with 4 GB of RAM. The packets and signatures are read offline from text files. The experiments of performing the matching are repeated 10 times and then averaged to compute the execution time and memory usage.

5.2 Traffic and signature extraction

Snort 2.9.0.4 rule set released on March 2011 is used to extract the signatures from the content keyword [3]. Six traces are chosen from the work by Aldwairi and Alansari to run our experiments [36]. The traces are read using Wireshark analyzer [37]. They differ from each other in the number of intrusions, and they cover the best and worst performance test cases. Both bad and ugly traces represent the worst cases, and the good traces represent the best case. The ugly: traces 51 and 58 are the most malicious and are composed on average of 43.3 and 44.6% of malicious packets, respectively. The bad: traces 1 and 22 are less malicious and on average contain 15.9 and 20.4% of malicious packets, respectively. The good traces represent daily network traffic such as good download (gd) and audio and video traces (av). Those traces contain the least number of intrusions and are used to represent the best or everyday normal traffic. Good download trace contains 2% malicious packets, and audio and video trace contains 1% malicious packets.

5.3 Execution time

The execution time is the time from reading the pattern until the end of the matching process. The matching process compares patterns with packet traces and writes out the results. This time includes the execution of pattern reading, packet reading, pattern encoding, MapReduce scheduler initialization, and the searching of the Map function. Moreover, it includes the output of the intermediate data and the matching count.

We compare four CPU implementations: original serial Myers algorithm, MPI parallelized version, Phoenix++, and MAPCG, in terms of the execution time. To do so, we use the six traces described in Section 5.2. Figure 3 shows the execution time for all Snort signatures. The result shows that our MapReduce implementations perform better than the serial and the MPI versions from literature. In addition, it shows that our MAPCG implementation of the algorithm is the fastest compared with the rest of implementations for all kinds of traces.

5.4 Speedup

The speedup is computed using the serial code as the baseline. Figure 4 shows the speedup for the three parallel implementations using the six traces over serial

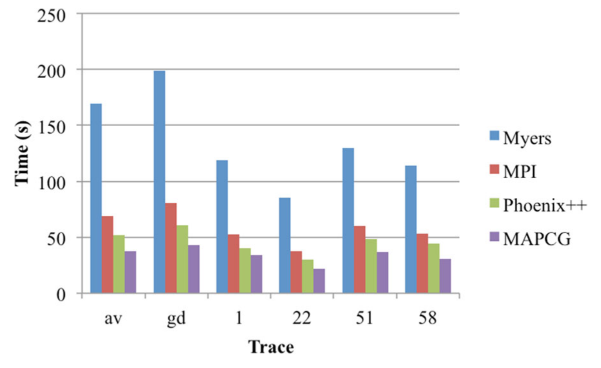

Fig. 3 Execution time of Myers algorithm using all signatures

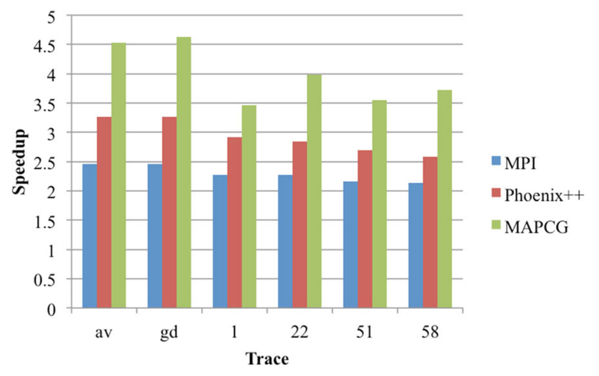

Fig. 4 Speedup over the serial implementation

Myers algorithm. MAPCG achieves the best speedup of all implementations. The speedup was more than four times for the normal (gd) network traffic trace. The (gd) trace is the largest dataset that was used in the evaluation. This is because a large data size allows the MapReduce runtime to better utilize its underlying design and remove the overheads of buffer allocation and task management. MapReduce is actually a framework designed for use in big data analytics.

Additionally, we cannot help but notice the significant increase in speedup of MAPCG over Phoenix++. This is due to the different strategies that are used in memory allocation and key/value management. Phoenix uses sorting for intermediate keys; however, MAPCG does not, therefore eliminating the sorting overhead. Overall, both MapReduce frameworks (Phoenix++ and MAPCG) provide excellent solutions for data-intensive applications.

Both implementations considerably over-perform traditional MPI parallelism solution. Table 2 summarizes the speedup values for both parallel Myers implementations, Phoenix++ and MAPCG, over the MPI parallel implementation. It is clear that MapReduce performs better than MPI even on CPU. This is because MPI spends more time in process communication to send the data between nodes. If the data is large, the high overhead degrades the performance. In addition, in MPI framework, if one of the processes fails, the whole task is terminated. However, MapReduce implements fault tolerance mechanisms that overcome this shortcoming. Therefore, MapReduce is a favorable choice. It is clear from Table 2 that MAPCG parallel implementation outperforms the Phoenix++ implementation for all traces.

Table 2 Speedup of MapReduce over MPI

Trace	Phoenix++	MAPCG
av	1.328	1.847
gd	1.326	1.882
1	1.289	1.527
22	1.25	1.753
51	1.244	1.64
58	1.207	1.742

5.5 Memory usage

Nowadays, large memories are more affordable and even small personal computers are shipped with large memories of 4 GB or more. Memory usage measures the maximum memory consumption of an application during its runtime. In our computation of memory consumptions, we include the memory space needed for the pattern vector, the trace file, and the intermediate data generated by the MapReduce paradigm. Figure 5 shows the memory consumption for all implementations, all traces, and all of Snort signatures. It shows that the MapReduce implementations (MAPCG, Phoenix++) consume more memory compared to the serial Myers and the MPI version. This is due to the intermediate data produced by the Map phase. The serial implementation processes the data items one by one, and there is no intermediate data to store and emit. Given the fact that Snort has almost 10,000 signatures and each of the traces has over 700,000 packets, we believe memory usage of under 2 GB is acceptable. With the current memory sizes of commercial machines, it is reasonable to trade-off this overhead in memory usage in order to achieve IDS speeds fast enough to prevent network attacks in real-time.

5.6 Complexity analysis

The complexity of Myers algorithm is easily computed as $O\left(m \times \text{SizeOf}\left(\sum\right) + n\right)$, where $\text{SizeOf}\left(\sum\right)$ is the size of the alphabet $\sum$, m is the length of the pattern or attack signature p, and n is the length of the text or packet [13]. This is assuming an exact matching of edit distance $k = 0$. Because of the different operations that might take place within Map and Reduce functions as well as combining the results of all Mappers and Reducers, computing

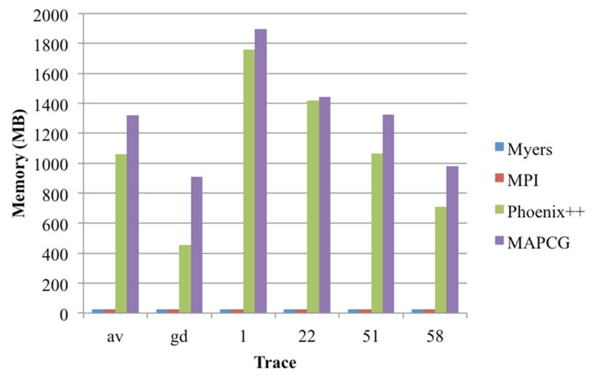

Fig. 5 Memory usage of the parallel implementations

an exact complexity model does not lead to easy algorithmic analysis. Even if we consider Mappers and Reducers separately, the different nature of each network traffic trace and attack signatures dataset makes the task of coming up with formal model more difficult. Finally, different implementations of MapReduce such as Phoenix++ and MAPCG do not make the complexity analysis any easier. This is because Phoenix++ adds Split and Merge, and MAPCG adds a runtime library functions. However, the main tasks that affect our algorithm complexity are the sum of maximum size, the running time, and the memory of all $< key, value >$ pairs input to or output from all Mappers/Reducers.

6 Conclusions and future work

A network intrusion detection system is a powerful system to inspect traffic and discover malicious activities. Pattern matching is the most intensive task in the detection process in terms of CPU time and memory. In this paper, we introduce different parallel implementations of the Myers algorithm to accelerate the matching process. Moreover, we evaluate the suitability of MapReduce framework as a new programming model to parallelize the algorithm. We parallelize Myers algorithm using different MapReduce implementations on a multi-core CPU. Myers algorithm with its various implementations are thoroughly evaluated under different workloads. Experiments show that Myers using MapReduce on multi-core CPU achieves an average speedup of more than four times compared to the serial implementation. In addition, MapReduce implementations achieved better performance than the traditional MPI parallelism techniques. Therefore, our performance improvement decreases the detection time and can lend a helping hand to meet the changes in the wire speed and the increasing number of attack signatures.

Several frameworks for parallel and distributed processing of big data have been proposed after the conclusion of this work. Apache introduced Spark in 2014 as an open-source framework for big data processing. Spark batch processing is tens of times faster than MapReduce because of its in-memory sharing. Later, Apache introduced stream processing Flink. It is faster than Spark's batch method. Both of these systems are being investigated to extend this work in the near future.

Acknowledgements
We would like to acknowledge the efforts of Yaser Jararweh and the valuable feedback of the reviewers.

Funding
This work was supported by the Deanship of Research at Jordan University of Science and Technology, grant 2011/287, and in part by Zayed University Research Office, Research Incentive Grant R15121.

Availability of data and material
Not applicable.

Authors' contributions
MA suggested the problem, solution, methodology, evaluation procedure, metrics, and experiments and supplied the datasets. MA prepared, cleansed, and parsed the traffic traces and Snort attack signatures. MA is the main author, he modified the write-up, proofread and submitted the final version. AA carried out the actual research, simulation implementation, coding, testing, and measurements. She prepared the raw initial draft of the paper. MJ was instrumental in the parallel multi-core and GPU implementations, and he was the major paper author. In response to the reviews, MA and MJ wrote the algorithm, performed the complexity analysis, and took care of the rest of the comments. All authors read and approved the final manuscript.

Competing interests
The authors declare that they have no competing interests.

Consent for publication
Not applicable.

Ethics approval and consent to participate
Not applicable.

Publisher's Note

Received: 22 July 2016 Accepted: 17 May 2017
Published online: 02 June 2017

References

1. Computer Insecurity (2017). https://www.sciencedaily.com/terms/computer_insecurity.htm. Accessed 19 Apr 2017
2. M Aldwairi, Y Khamayseh, M Al-Masri, Application of artificial bee colony for intrusion detection systems. Secur. Commun. Netw. **8**(16), 2730–2740 (2015)
3. Snort: the open source network intrusion detection system. (2017). http://www.snort.org. Accessed 19 Apr 2017
4. M Aldwairi, N Ekailan, Hybrid multithreaded pattern matching algorithm for intrusion detections systems. J. Inform. Assur. Secur. **6**(6), 512–521 (2011)
5. M Kharbutli, A Mughrabi, M Aldwairi, Function and data parallelization of Wu-Manber pattern matching for intrusion detection systems. Netw. Protoc. Algorithms J. **4**(3), 46–61 (2012)
6. X Su, Z Ji, A Lian, in *Proceedings of the 2nd International Conference on Computer Science and Electronics Engineering*. A parallel AC algorithm based on SPMD for intrusion detection system, (2013). https://www.researchgate.net/publication/266646801_A_Parallel_AC_Algorithm_Based_on_SPMD_for_Intrusion_Detection_System
7. M Holtz, B David, R Junior, Building scalable distributed intrusion detection systems based on the MapReduce framework. Revista Telecomunicacoes J. **13**(2) (2011)
8. L Hu, Z Wei, F Wang, X Zhang, K Zhao, An efficient AC algorithm with GPU. Procedia Engineering. **29**, 4249–4253 (2012). ISSN 1877-7058, doi:10.1016/j.proeng.2012.01.652. http://www.sciencedirect.com/science/article/pii/S1877705812006625
9. D Xu, H Zhang, Y Fan, The GPU-based high-performance pattern-matching algorithm for intrusion detection. J. Comput. Inform. Syst. **9**(10) (2013)
10. K Xu, W Cui, Y Hu, L Guo, Bit-parallel multiple approximate string matching based on GPU. Procedia Computer Science. **17**, 523–529 (2013). ISSN 1877-0509, doi:10.1016/j.procs.2013.05.067. http://www.sciencedirect.com/science/article/pii/S1877050913002007
11. CL Hung, CY Lin, Hh Wang, CY Chang, in *2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems.* Efficient packet pattern matching for gigabit network intrusion detection using GPUs, (Liverpool, 1612). doi:10.1109/HPCC.2012.235. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6332370&isnumber=6331993
12. J Dean, S Ghemawat, in *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*. MapReduce: Simplied data processing on large clusters (Berkeley, CA, USA, p. 2004)

13. G Myers, A fast bit-vector algorithm for approximate string matching based on dynamic programming. J ACM. **46**(3), 395–415 (1999)
14. R Boyer, J Moore, A fast string searching algorithm. Commun. ACM. **20**(10), 762–772 (1977)
15. S Wu, U Manber, *A fast algorithm for multi-pattern searching, Technical Report TR-94–17*. Department of Computer Science, University of Arizona, 1994)
16. L Langner, *Parallelization!of!Myers Fast!BitDVector! Algorithm!using GPGPU*. (Diploma/Thesis, Freie Universität, Berlin, 2011). http://www.mi.fu-berlin.de/en/inf/groups/abi/theses/master_dipl/langner_bitvector/dipl_thesis_langner.pdf
17. G Navarro, A guided tour to approximate string matching. ACM Comput. Surv. **33**(1), 31–88 (2001)
18. Edit_distance (2017). http://en.wikipedia.org/wiki/Edit_distance. Accessed 6 Mar 2017
19. RW Hamming, Error detecting and error correcting codes. Bell Syst. Tech. J. **29** (147). http://garfield.library.upenn.edu/classics1982/A1982NY35700001.pdf
20. V Levenshtein, Binary codes capable of correcting deletions. Insertions Reversals. Sov. Phys. Dokl. **10**(8), 707 (1966)
21. F Damerau, A technique for computer detection and correction of spelling errors. Commun. ACM. **7**(3), 171–176 (1964)
22. S Wandelt, D Deng, S Gerdjikov, S Mishra, P Mitankin, M Patil, E Siragusa, A Tiskin, W Wang, J Wang, U Leser, *State-of-the-art in string similarity search and join*, vol. 43, (2014), pp. 64–76. doi:10.1145/2627692.2627706. http://dl.acm.org/citation.cfm?id=2627706
23. Bit array (2017). https://pypi.python.org/pypi/bitarray/0.8.1. Accessed 22 July 2016
24. mpicccomplier (2016). http://www.mcs.anl.gov/research/projects/mpi/www/www1/mpicc.html. Accessed 19 Apr 2017
25. OpenMP (2017). http://www.openmp.org/. Accessed 19 Apr 2017
26. OpenMP (2017). https://computing.llnl.gov/tutorials/openMP/. Accessed 19 Apr 2017
27. J Dean, S Ghemawat, MapReduce: simplified data processing on large clusters. Commun. ACM. **51**(1), 107–113 (2008). doi:10.1145/1327452.1327492. http://dl.acm.org/citation.cfm?id=1327492
28. C Ranger, R Raghuraman, A Penmetsa, G Bradski, C Kozyrakis, in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. Evaluating MapReduce for multi-core and multiprocessor systems, (Scottsdale, 2007), pp. 13–24. doi:10.1109/HPCA.2007.346181. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4147644&isnumber=4147636
29. Y Shan, B Wang, J Yan, Y Wang, N Xu, H Yang, in *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays (FPGA '10)*. FPMR: MapReduce framework on FPGA (ACM, New York, 2010), pp. 93–102. doi:10.1145/1723112.1723129. http://dl.acm.org/citation.cfm?id=1723129
30. C Hong, D Chen, W Chen, W Zheng, H Lin, in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques (PACT '10)*. MapCG: writing parallel program portable between CPU and GPU (ACM, New York, 2010), pp. 217–226. doi:10.1145/1854273.1854303. http://dl.acm.org/citation.cfm?id=1854303
31. CS Kouzinopoulos, KG Margaritis, in *2009 13th Panhellenic Conference on Informatics*. String matching on a multicore GPU using CUDA, (Corfu, 2009), pp. 14–18. doi:10.1109/PCI.2009.47. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5298766&isnumber=5298190
32. RN Horspool, Practical fast searching in strings. Software Practice and Experience. **10**(6), 501–506 (1980). doi:10.1002/spe.4380100608. http://onlinelibrary.wiley.com/doi/10.1002/spe.4380100608/full
33. G Vasiliadis, S Antonatos, M Polychronakis, EP Markatos, S Ioannidis, in *Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection (RAID '08)*, ed. by R Lippmann, E Kirda, and A Trachtenberg. Gnort: High Performance Network Intrusion Detection Using Graphics Processors (Springer-Verlag, Berlin, 2008), pp. 116–134. doi:10.1007/978-3-540-87403-4_7. http://dl.acm.org/citation.cfm?id=1433016
34. E Chibli, *A multiprocessor parallel approach to bit-parallel approximate string matching, Master thesis*. (Faculty of California State University, San Bernardino, 2008)
35. J Talbot, RM Yoo, C Kozyrakis, in *Proceedings of the second international workshop on MapReduce and its applications (MapReduce '11)*. Phoenix++: modular MapReduce for shared-memory systems (ACM, New York, 2011), pp. 9–16. doi:10.1145/1996092.1996095. http://dl.acm.org/citation.cfm?doid=1996092.1996095
36. M Aldwairi, D Alansari, in *2011 7th International Conference on Next Generation Web Services Practices*. Exscind: Fast pattern matching for intrusion detection using exclusion and inclusion filters (Salamanca, Spain, 2011), pp. 24–30. doi:10.1109/NWeSP.2011.6088148. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6088148&isnumber=6088142
37. A Orebaugh, G Ramirez, J Beale, J Wright. 1st, in *Syngress Media Inc*. Wireshark & Ethereal Network Protocol Analyzer Toolkit, (2007). https://www.elsevier.com/books/wireshark-andampamp-ethereal-network-protocol-analyzer-toolkit/orebaugh/978-1-59749-073-3