A Framework for Distributed Pattern Matching Based on Multithreading

Najib Kofahi and Ahmed Abusalama
Department of Computer Science, Yarmouk University, Jordan

Abstract: *Despite of the dramatic evolution in high performance computing we still need to devise new efficient algorithms to speed up the search process. In this paper, we present a framework for a data-distributed and multithreaded string matching approach in a homogeneous distributed environment. The main idea of this approach is to have multiple agents that concurrently search the text, each one from different position. By searching the text from different positions the required pattern can be found more quickly than by searching the text from one position). Concurrent search can be achieved by two techniques; the first is by using multithreading on a single processor, in this technique each thread is responsible for searching one part of the text. The concurrency of the multithreading technique is based on the time sharing principle, so it provides us of an illusion of concurrency not pure concurrency. The second technique is by having multiprocessor machine or distributed processors to search the text; in this technique all of the processors search the text in a pure concurrent way. Our approach combines the two concurrent search techniques to form a hybrid one that takes advantage from the two techniques. The proposed approach manipulates both exact string matching and approximate string matching with k-mismatches. Experimental results demonstrate that this approach is an efficient solution to the problem in a homogeneous clustered system.*

Keywords: *Pattern matching, online search algorithms, multithreading, concurrency, java space technology, distributed processing.*

Received April 7, 2009; accepted November 5, 2009

1. Introduction

The problem of finding exact or non-exact occurrences of a pattern P in a text T over some alphabet is a central problem of combinatorial pattern matching and has a variety of applications in many areas of computer science [19]. String searching algorithms can be accomplished in two ways:

1. Exact match, meaning that the passages returned will contain an exact match of the key input.
2. Approximate match, meaning that the passage will contain some part of the key word input [17].

Although the dramatic evolution of processor technology and other advances have reduced search response to negligible times, pattern matching problem still remains a useful area of research and development for a number of reasons. Firstly, as the size of data continues to grow, sequence searches will become increasingly taxing on search engines. Secondly, the pattern matching still remains an integral part of faster matching algorithms, typically comprising the final part of a search. Lastly, researchers have to understand the classical methods of pattern matching to develop new efficient algorithms [12].

With the developments of new pattern matching techniques, efficiency and speed are the main factors in deciding among different options available for each application area. Each application area has certain special features that can be used by pattern matching technique best suited for that area [24].

This study presents a new approach to solve the problem of pattern matching depending on the idea of search distribution over multiple connected nodes. At each node we adopt the multithreading paradigm to speedup the searching process. By using multithreading and distributed search over connected nodes the text can be searched concurrently from different positions. This will decrease the time needed to find the required pattern.

For our implementation purposes, Java threads - a built-in parallelism support- is used to implement the multithreaded approach. To implement the distributed processing, we use the Java space technology, which is easy to implement and satisfy our problem requirements. At each node in the distribution, the multithreaded approach works in a timesharing manner.

The rest of the paper is organized as follows. Section 2 gives a brief introduction to string searching algorithms. In section 3, the subject of threads and multithreading is introduced, and distributed computing and distributed algorithms are discussed in section 4. The main contribution of this research is given in section 5: The multithreaded distributed pattern matcher. Implementation and experimental results are explained in section 6. The conclusions and future work is given in section 7.

2. String Searching Algorithms

In general, there are two ways to search a text T to find a pattern P depending on weather the algorithm performs some preprocessing on the text T or not. Depending on the problem domain, some of the matching algorithms have to preprocess the text and build a data structure to aid in the search process. Such algorithms are called indexers. (i.e., suffix arrays. suffix trees, and inverted files). Other algorithms directly perform the search on the text without any preprocessing. Such algorithms are called sequential or online search algorithms (i.e., Knuth-Morris, Boyer Moore, and Brute-Force) [28]. In this paper, we are interested in enhancing the online search algorithms.

3. Threads and Multithreading Motivation

A thread is simply a path of execution within a process or it's a low weight process [13]. In single threaded applications, all operations, regardless of type, duration or priority, execute on a single thread. Such applications are simple to design and build and all operations are serialized. That means there is one thread running at a time. However, there are many situations where it's useful to have multiple threads of execution that run simultaneously based on the principle of timesharing [26].

Concurrency is very important in many computer applications, but most of the programming languages do not enable programmers to specify concurrent activities. Rather, programming languages generally provide only a simple set of control structures that enable programmers to perform one action at a time then precede to the next action after the previous one is finished. The kind of concurrency that computers perform today normally is implemented as operating system "primitives" available only to high experienced system programmers [7].

Java is unique among popular general-purpose programming languages in that it makes concurrency primitives available to the applications programmer. The programmer specifies that applications contain threads of execution, each thread designating a portion of a program that may execute concurrently with other threads. Multithreading gives the Java programmer powerful capabilities that are not available in C and C++, the languages on which Java is based [7].

CPU can process only one instruction at time (regardless to the pipelining technology). When a multithreaded application runs on a single processor it's impossible to have a complete or pure parallelism, but it gives us an illusion of parallelism depending on the timesharing principle. The idea is behind the very fast context switching between threads that can be performed by the CPU. Because of that fast context switching the different running threads seems to be running at the same time.

This study implements a multithreading text searching approach to improve text searching performance at a single CPU machine. The idea is to have more than one searcher thread that search the text from different positions. Since the required pattern may occur at any position, having multiple searchers is better than searching the text sequentially from the first character to the last one.

4. Distributed Computing and Distributed Algorithms

The new technologies of networking and the dramatic evolution of the internet and intranet impacts the way that we use computers and changes the way we create applications for them. Distributed applications are becoming the natural way to build software. "Distributed computing" is all about designing and building applications as a set of processes that are distributed across a network of machines and work together as an ensemble to solve a common problem [8].

Distributed algorithms are algorithms designed to run on a distributed system; where many processes cooperate by solving parts of a given problem in parallel. For this purpose, the processes have to exchange data and synchronize their actions. In contrast to so called parallel algorithms, communication and synchronization is solely done by message passing -there are no shared variables- and usually the processes do not even have access to a common clock. Since message transmission time cannot be ignored, no process has immediate access to the global state. Hence, control decisions must be made on a partial and often outdated view of the global state which is assembled from information gathered gradually from other processes [16].

Distributing computing requires a tool by which the distributed machines can communicate. Many tools are available such as Remote Method Invocation (RMI), CORBA and Java Space. Each tool has its own specifications; the application designer chooses the appropriate one for his application requirement.

5. The Multithreaded Distributed Pattern Matcher

5.1. Multithreading Approach

The main idea by using multithreading to solve the pattern matching problem (on a single CPU machine) is to have multi searching threads that search the target text simultaneously in a timesharing manner. Each thread starts searching the target text from different position. By searching the text from different positions (instead of one position) the speed of finding the required pattern will increase. The speed up obtained by using this approach comes from the nature of the

threads work (time sharing) and the nature of the target text that can be accessed from any position. Logically, having more than one person searching for something is better than having one searcher. To illustrate the idea let's see the following example:

Suppose that we have an 80 character text (80 characters length) and we search for a pattern occurs at position 61 using brute force algorithm, as shown in Figure 1.

1	2	3	4	5	6	7	8	9	10
p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}
p_{11}	p_{12}	p_{13}	p_{14}	p_{15}	p_{16}	p_{17}	p_{18}	p_{19}	p_{20}
p_{21}	p_{22}	p_{23}	p_{24}	p_{25}	p_{26}	p_{27}	p_{28}	p_{29}	p_{30}
p_{31}	p_{32}	p_{33}	p_{34}	p_{35}	p_{36}	p_{37}	p_{38}	p_{39}	p_{40}
p_{41}	p_{42}	p_{43}	p_{44}	p_{45}	p_{46}	p_{47}	p_{48}	p_{49}	p_{50}
p_{51}	p_{52}	p_{53}	p_{54}	p_{55}	p_{56}	p_{57}	p_{58}	p_{59}	p_{60}
p_{61}	p_{62}	p_{63}	p_{64}	p_{65}	p_{66}	p_{67}	p_{68}	p_{69}	p_{70}
p_{71}	p_{72}	p_{73}	p_{74}	p_{75}	p_{76}	p_{77}	p_{78}	p_{79}	p_{80}

Figure 1. Searching an 80 character text.

In sequential search (having one working process to search the text from the first character to the last one) the CPU examines 60 characters to reach the required pattern. Now let's move to the multi threading effect on this text.

To force each thread to search the text from different position, the text is divided into equal parts and each thread is responsible for searching a particular part. If the text length is not divisible by the number of threads then the last thread will search the division remainder.

Threads work in a time sharing manner, that means (in the simple form) the first thread examines the first character of its assigned text part, then the CPU makes a context switch to the second thread to examine the first character of its part, and then the CPU makes a context switching to the third thread and so on until the CPU makes context switching to all of the threads. Then CPU switches back to the first thread to examine the second character of its part, and so on. This process is repeated until the whole text is examined, as shown Figure 2.

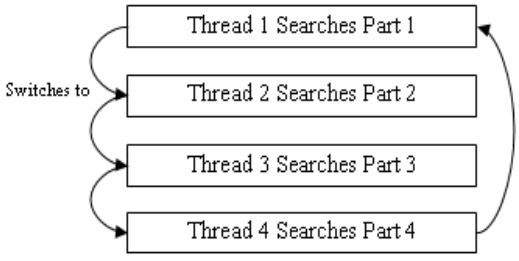

Figure 2. Searcher threads context switching.

- *Using two threads to search the text*: We have a text with 80 characters (1-80), so we divide it into two parts. The first part is from 1-40 and the second part is from 41-80, as shown in Figure 3.

	1	2	3	4	5	6	7	8	9	10
Thread 1	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}
	p_{11}	p_{12}	p_{13}	p_{14}	p_{15}	p_{16}	p_{17}	p_{18}	p_{19}	p_{20}
	p_{21}	p_{22}	p_{23}	p_{24}	p_{25}	p_{26}	p_{27}	p_{28}	p_{29}	p_{30}
	p_{31}	p_{32}	p_{33}	p_{34}	p_{35}	p_{36}	p_{37}	p_{38}	p_{39}	p_{40}
Thread 2	p_{41}	p_{42}	p_{43}	p_{44}	p_{45}	p_{46}	p_{47}	p_{48}	p_{49}	p_{50}
	p_{51}	p_{52}	p_{53}	p_{54}	p_{55}	p_{56}	p_{57}	p_{58}	p_{59}	p_{60}
	p_{61}	p_{62}	p_{63}	p_{64}	p_{65}	p_{66}	p_{67}	p_{68}	p_{69}	p_{70}
	p_{71}	p_{72}	p_{73}	p_{74}	p_{75}	p_{76}	p_{77}	p_{78}	p_{79}	p_{80}

Figure 3. Using two threads for text searching.

By considering the context switching illustrated in Figure 2, the CPU examines 40 characters to reach the required pattern. So in this case using two threads is better than using one thread. Note that this is not always the case; it depends on the position of the pattern in the text. This matter will be illustrated in the following.

- *Using three threads to search the text:* The text is divided into three parts; part one from 1-26, part two from 27-52 and part three from 53-80, as shown in Figure 4. In this case, the CPU examines 21 characters to reach the required pattern.

	1	2	3	4	5	6	7	8	9	10
T1	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}
	p_{11}	p_{12}	p_{13}	p_{14}	p_{15}	p_{16}	p_{17}	p_{18}	p_{19}	p_{20}
	p_{21}	p_{22}	p_{23}	p_{24}	p_{25}	p_{26}				
T2	p_{27}	p_{28}	p_{29}	p_{30}	p_{31}	p_{32}	p_{33}	p_{34}	p_{35}	p_{36}
	p_{37}	p_{38}	p_{39}	p_{40}	p_{41}	p_{42}	p_{43}	p_{44}	p_{45}	p_{46}
	p_{47}	p_{48}	p_{49}	p_{50}	p_{51}	p_{52}				
T3	p_{53}	p_{54}	p_{55}	p_{56}	p_{57}	p_{58}	p_{59}	p_{60}	p_{61}	p_{62}
	p_{63}	p_{64}	p_{65}	p_{66}	p_{67}	p_{68}	p_{69}	p_{70}	p_{71}	p_{72}
	p_{73}	p_{74}	p_{75}	p_{76}	p_{77}	p_{78}	p_{79}	p_{80}		

Figure 4. Using three threads for text searching.

- *Using four threads to search the text:* The text is divided into four parts; part one from 1-20, part two from 21-40, part three from 41-60 and part four from 61-80, as shown in Figure 5. In this case, the CPU examines only 3 characters to reach the required pattern. So, it is an incredible improvement to search this text considering the using of one and two threads. Increasing the number of threads is not

the idea to speed up this approach; the idea is to have an appropriate number of threads (or text parts) by which the pattern occurs at a near position to the beginning of any text part. When a pattern occurs at a near position to the beginning of a text part it can be found quickly.

	1	2	3	4	5	6	7	8	9	10
T1	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}
	p_{11}	p_{12}	p_{13}	p_{14}	p_{15}	p_{16}	p_{17}	p_{18}	p_{19}	p_{20}
T2	p_{21}	p_{22}	p_{23}	p_{24}	p_{25}	p_{26}	p_{27}	p_{28}	p_{29}	p_{30}
	p_{31}	p_{32}	p_{33}	p_{34}	p_{35}	p_{36}	p_{37}	p_{38}	p_{39}	p_{40}
T3	p_{41}	p_{42}	p_{43}	p_{44}	p_{45}	p_{46}	p_{47}	p_{48}	p_{49}	p_{50}
	p_{51}	p_{52}	p_{53}	p_{54}	p_{55}	p_{56}	p_{57}	p_{58}	p_{59}	p_{60}
T4	p_{61}	p_{62}	p_{63}	p_{64}	p_{65}	p_{66}	p_{67}	p_{68}	p_{69}	p_{70}
	p_{71}	p_{72}	p_{73}	p_{74}	p_{75}	p_{76}	p_{77}	p_{78}	p_{79}	p_{80}

Figure 5. Using four threads for text searching.

- *Using five threads to search the text*: The text is divided into five parts; part one from 1-16, part two from 17-32, part three from 33-48, part four from 49-64 and part five from 65-80, as shown in Figure 6. In this case, the CPU examines 60 characters to reach the required pattern. Although the number of threads have increased, this case is worse than others. Each time we use different number of threads the pattern position become closer or distant from the beginning of the data parts.

	1	2	3	4	5	6	7	8	9	10
T1	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}
	p_{11}	p_{12}	p_{13}	p_{14}	p_{15}	p_{16}				
T2	p_{17}	p_{18}	p_{19}	p_{20}	p_{21}	p_{22}	p_{23}	p_{24}	p_{25}	p_{26}
	p_{27}	p_{28}	p_{29}	p_{30}	p_{31}	p_{32}				
T3	p_{33}	p_{34}	p_{35}	p_{36}	p_{37}	p_{38}	p_{39}	p_{40}	p_{41}	p_{42}
	p_{43}	p_{44}	p_{45}	p_{46}	p_{47}	p_{48}				
T4	p_{49}	p_{50}	p_{51}	p_{52}	p_{53}	p_{54}	p_{55}	p_{56}	p_{57}	p_{58}
	p_{59}	p_{60}	p_{61}	p_{62}	p_{63}	p_{64}				
T5	p_{65}	p_{66}	p_{67}	p_{68}	p_{69}	p_{70}	p_{71}	p_{72}	p_{73}	p_{74}
	p_{75}	p_{76}	p_{77}	p_{78}	p_{79}	p_{80}				

Figure 6. Using five threads for text searching.

5.2. The Problem of Text Partitioning Among the Threads

Since there are multiple threads that search the text form different positions, the text has to be divided into subtexts and each subtext is allocated to a particular thread. Texts or (strings) in Java are stored in a single array data structure and its characters can be accessed via the indexes of that array. To avoid the problem that occurs when a pattern is found at the boundaries of two subtexts the text is virtually divided into subtext. Instead of searching independent subtexts, all of the threads perform the search on the same text (array) but each one with different indexes. In this case if a pattern found at the last positions of a subtext *i* and the first positions of the next subtext, then the thread that searches the subtext *i* can find the pattern since it can access the characters of the next subtext. Text is divided among threads according to the following pseudo code:

```
For i=1 to numberOfThreads
search( ((i-1) * (( text.length() /numOfThreads ) –1))
 +i ,( i* ( ( text.length() /numOfThreads ) –1 ) ) +i )
i= i+ 1
End do
```

Where: *numberOfThreads*: The number of the working threads.
Search (x, y): Search the text from position x to position y.
text.length: The length of the target text.

Consider the case of using two threads discussed earlier to find a pattern P with a length of 6 characters and occurs at position 38 in the text. The first thread searches the text from position 1 to 40, and the second thread searches the text from position 41 to 80. When the first thread detects a partial match at positions 38-40 it can continue to test the positions 41-43.

5.3. Implementing and Testing the Multithreaded Approach

The multithreaded approach is implemented with Java threads on Intel P4 CPU with 2.53 GHz speed. Text size is 2.6 MB. Pattern size is 1 KB (to make more computations). Search algorithm used is the brute force algorithm. The pattern occurs -in the text- at the position number 1,639,400. The results obtained for different number of threads is shown in Table 1.

The best time gained by using five threads (1500 ms instead of 8125 ms by using one thread) as shown in Figure 7.

Table 1. Time obtained in milliseconds by using varying numbers of threads on a single CPU.

Number of Threads	Time in ms
1	8125
2	3350
3	11360
4	6453
5	1500
6	9750
7	5109
8	13312
9	8485
10	3578

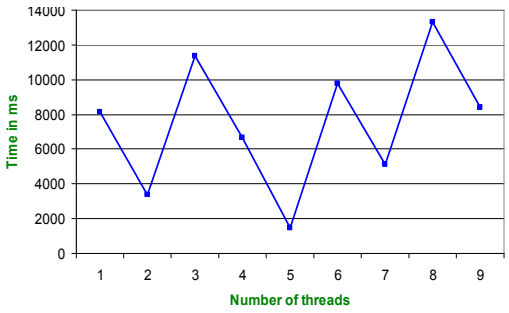

Figure 7. Relation between number of threads and time required to find the pattern.

5.4. Analyzing Multithreading Approach Behavior

From the experimental results it is shown that the behavior of the multithreaded approach is not predictable. As seen from the experimental results each number of threads gives a different result. To have good results to find the required pattern, the pattern should occur at the beginning of any subtext. The pattern position (according to the beginning of subtexts) depends on the number of threads (subtext= text length/ number of threads) so we need to know the number of threads by which the pattern occurs at a near position to the beginning of a subtext. The relation between pattern position and number of threads is described in the following formula:

$$((N/Tn) * Ti) + 1 \leq \text{pattern position} \leq ((N/Tn) * Ti) + N/2 * Tn$$

Where:
N: Target text length.
Tn: Number of threads.
Ti: Thread number i, i= 0, 1, 2 … n.

The shaded area in Figure 8 shows the pattern positions described by this formula. The problem in this formula is the two unknown variables (pattern position and Tn).

T0	1	2	3	4	5	6	7	8	9	10
T1	11	12	13	14	15	16	17	18	19	20
T2	21	22	23	24	25	26	27	28	29	30
…										
Tn										N

Figure 8. The best pattern positions in the multithreaded approach.

This unpredictable behavior depends on the text length. If we have a small text that will be divided into smaller subtexts and the pattern occurs outside the range specified by the above formula then the required time to find it will not be much large than the time required to find the pattern positions specified by the formula. The larger text length led to more unpredictable behavior. To obtain good results we have to work on small texts, but what if we have large texts?

5.5. The Distributed Matcher

The multithreaded approach illustrated in the previous section shows very good results; but there is a problem of the behavior of the threads. The problem is how we can determine the number of the threads that gives us the highest speed up. This problem because all of the threads running on a single CPU in a time sharing manner. As mentioned in the previous section the problem can be smoothed by working on small texts. From this point we move our thinking to another approach in which we have multiple computers that running the multithreaded approach at the same time; each computer with different subtext (that's smaller than the target text) to search. So now we move to the distributed computing to solve our new problem.

In the distributed matcher approach the text is divided into equal subtexts and each computer in the distributed system is running the multithreaded text searching approach on a different subtext. In this case the text is partitioned two times; one by the distributed matcher and the other by the multithreaded matcher. For example, if we have five computers in the distributed matcher and each of them running the multithreaded matcher with four threads, then we have twenty searcher threads that search the text from different positions at the same time. The more searchers (on different machines) the more speed to find the pattern.

Our distributed approach consists of one client and (n) servers as shown in Figure 9. The client is responsible for broadcasting the pattern to the servers and receiving the results from these servers. Each server has an independent copy of the text. The client is not responsible for distributing the text to avoid communication overhead. The first server that finds the pattern (or patterns) sends the result back to the client.

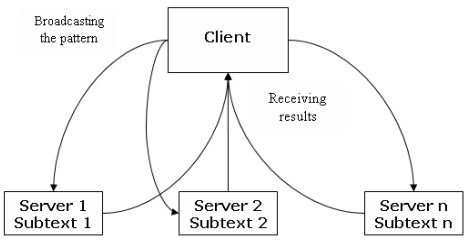

Figure 9. Distributed matcher.

5.6. The Problem of Text Partitioning Among Servers

As in the multithreaded approach the virtual portioning is used to partition the text across the multiple servers. Each sever machine has a copy of the

text and searches a part of it depending on the index partitioning that determined by the client machine. The virtual partitioning of the thread subtexts and servers subtexts is done according to the following pseudo code:

```
For each server
Do i=1 to numberOfThreads
bruteForce (((startSearch + (threadDataLen * i))) +
(sId -1) * partLen , ((startSearch + (threadDataLen*
(i+1)))+((sId-1 ) * partLen ))-1);
i=i+1
End do
```

Where:
startSearch: The position from which the server starts the search.
threadDataLen: Length of the subtext to be searched by each thread at the server.
Sid: Server number.

5.7. Analyzing Distributed Multithreading Approach Behavior

The main contribution of using distributed processing in this study is to distribute search load among multiple servers and to implement the multithreaded approach on small subtexts (to have a smoothed multithreading behavior). As mentioned above, the distributed multithreaded matcher partitions the target text in two stages. This two partition stages affect the pattern position according to the subtexts assigned to the searcher threads at each server. Consider the following example. Suppose that we have a text with 400 characters searched by the distributed multithreaded matcher with four servers and 4 threads at each server. The following partitioning scheme will be performed as shown in Table 2. In this case, there are 16 threads (4*4) that search the text from different positions at the same time.

6. Implementation and Experimental Results

For exploiting parallelism in each server, the multithreaded approach is used and implemented with Java threads. For distributed processing, the distributed approach is used and implemented with Java space technology. For experiments, we used four computers connected by a high performance local network. Computers in the network have a high speed network interface card (NIC with Gigabit speed) and connected via a high speed switch (Gigabit thernet switch). Each computer in the network has 2 GHz Intel Pentium 4 CPU and 1 GB of RAM. Experiments are done under windows XP professional edition environment. The graph in Figure 10 shows the comparison results of the sequential search and our distributed multithreaded approach on a variable text size and fixed pattern size. In this experiment we use four servers on which the multithreaded approach is running with five threads. The distributed multithreaded approach produces better performance than the sequential approach.

Table 2. The partition of text with 400 character length among four servers and four threads at each server.

	Thread ID	**Search Range**
Server 1 searches the text from position 1 to position 100	Thread 1	1 – 25
	Thread 2	26 – 50
	Thread 3	51 – 75
	Thread 4	76 – 100
Server 2 searches the text from position 101 to position 200	**Thread ID**	**Search Range**
	Thread 5	101 – 125
	Thread 6	126 – 150
	Thread 7	151 – 175
	Thread 8	176 – 200
Server 3 searches the text from position 201 to position 300	**Thread ID**	**Search Range**
	Thread 9	201 – 225
	Thread 10	226 – 250
	Thread 11	251 – 275
	Thread 12	276 – 300
Server 4 searches the text from position 301 to position 400	**Thread ID**	**Search Range**
	Thread 13	301 – 325
	Thread 14	326 – 350
	Thread 15	351 – 375
	Thread 16	376 – 400

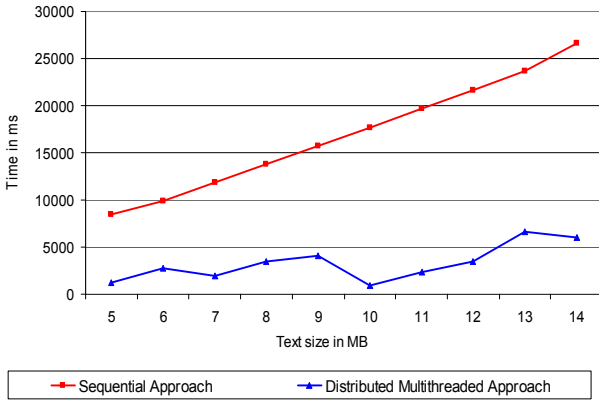

Figure 10. Sequential search vs distributed multithreaded search with four servers and five threads at each server. Text size is variable and pattern length is 500 characters.

As shown in the graph, while the text size is increasing, the time required to find the pattern by the sequential approach is increased. In the distributed multithreaded approach this is not always the case, i.e., at text sizes of (10, 11, 12 MB) the time required to find the pattern is less than the time required to find it with a text size of 9 MB. This improvement results from the multithreaded approach, where at the text sizes of (10, 11, 12 MB) the pattern occurs at a position near to the beginning of a subtext searched by a particular thread. The graph in Figure 11 shows the comparison results of the sequential search and our distributed multithreaded approach on a fixed text size

and variable pattern size. In this experiment we use four servers on which the multithreaded approach is running with five threads. The distributed multithreaded approach produces better performance than the sequential approach.

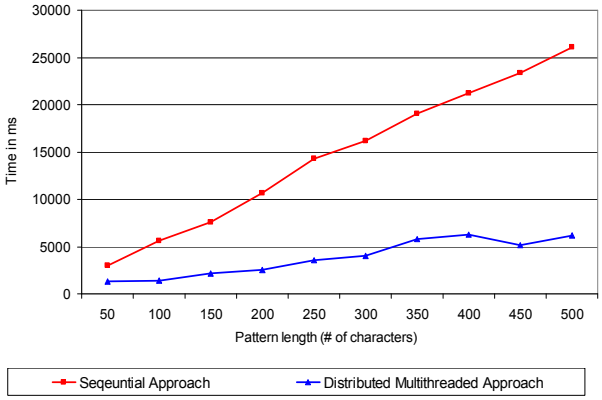

Figure 11. Sequential search vs distributed multithreaded search with four servers and five threads at each server. Text size is 14 MB and pattern length is variable.

The graph in Figure 12 shows the effect of the number of servers on the time required to find the pattern.

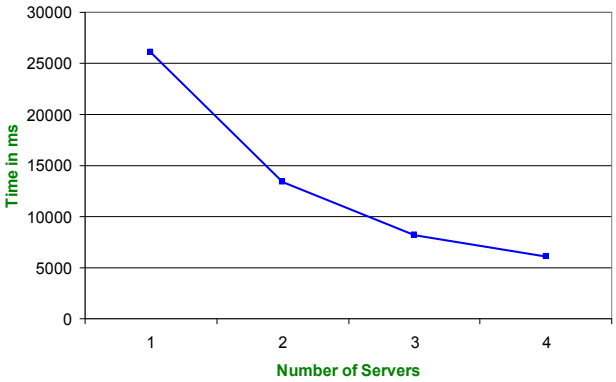

Figure 12. Relation between number of servers and time required to find the pattern in the distributed multithreaded approach. Text size is 14 MB and pattern length is 500 characters.

As seen from the graph in Figure 12, the more servers the less time to find the pattern. The improvement achieved by increasing number of servers comes from distributing the load of the search process and the affect of text partitioning on the pattern position according to the thread's subtexts. Note that the data is partitioned two times; the first one is for distribution search load among the servers and the other is for assigning subtexts to the searcher threads at each server. Both of the partitioning stages affect the position of the pattern according to the thread's subtexts.

As mentioned in the previous sections, we can not determine the number of threads that will produce the best results. For some pattern positions in a very large texts having one thread is better than having multiple threads. By the using of distributed processing the multithreaded approach is implemented on smaller texts (since the target text is partitioned into subtexts). The distributed approach smoothes the multithreaded approach behavior. Table 3 shows the smoothed behavior of the multithreaded approach on large text which is divided into smaller subtexts among four servers. The time required to find the pattern using 2-9 threads is less than or near the time required to find it using one thread.

Table 3. The effect of number of threads on the time required to find the pattern using 4 servers. Text size is 14 MB and pattern length is 500 characters.

Number of Threads	Time Using 4 Servers
1	6109
2	6109
3	3102
4	7124
5	5640
6	4640
7	1500
8	6842
9	5520

Table 4 shows the rough behavior of the multithreaded approach on large text using single computer.

Table 4. The effect of number of threads on the time required to find the pattern using single computer. Text size is 14 MB and pattern length is 500 characters.

Number of Threads	Time Using 4 Servers
1	28609
2	26344
3	22391
4	21047
5	18564
6	16672
7	14524
8	12754
9	10828

7. Conclusions and Future Work

Several researches and techniques were developed to solve the pattern matching problem which considered as a hot area for research. In this report we have presented a new efficient technique to solve this problem using multithreading and distributed processing. Multithreaded search on a single CPU have a nondeterministic behavior, since there is a particular number of threads that will produce the highest performance. This number of threads is data dependent number so it cannot be predetermined before performing search process. To minimize the effect of this problem the distributed processing is merged with the multithreaded search. By having

multiple machines that running the multithreaded approach the behavior of the multithreaded search becomes more reliable.

The experiments on the distributed multithreaded approach produce better results than the sequential search. For experiments we implement the brute force algorithm (its complexity is O (mn)) to ease the time tracking. Any sequential search algorithm can be applied to the distributed multithreaded approach.

In this paper, we introduce a distributed multithreaded string matcher that runs on a homogeneous environment which all of its machines have the same capabilities. But what if we have a heterogeneous environment that consists of workstations (with multiple CPUs) and PCs with different specifications? The problem of implementing our approach in a heterogeneous environment results from the need for a dynamic load balancing among the different servers, i.e. workstations must have work load more than PCs. The problem of dynamic load balancing of the distributed pattern matching is not a simple problem. Many techniques were developed to solve this problem and new techniques are waiting to be discovered.

References

[1] Ababneh M., Oqeili S., and Abdeen R., "Occurrences Algorithm for String Searching Based on Brute-Force Algorithm," *Computer Journal of Science Publication*, vol. 2, no. 1, pp. 82-85, 2006.

[2] Amihood A., Moshe L., and Ely P., "Approximate Subset Matching with Don't Cares," *in Proceedings of 12th Annual ACM-SIAM Symposium on Discrete Algorithms*, USA, pp. 305-306, 2001.

[3] Andersson A. and Thorup M., "Dynamic String Searching," *in Proceedings of 12th Annual ACM-SIAM Symposium on Discrete Algorithms*, USA, pp. 307-308, 2001.

[4] Bao R., "Distributed Computing via RMI and CORBA," *in Proceedings of Department Computer Since*, USA, pp. 24-33, 2001.

[5] Bishop P. and Warren N., *JavaSpaces in Practice*, Addison Wesley, 2002.

[6] Boyer R. and Moore J., "A Fast String Searching Algorithm," *Computer Journal of Communications of the ACM*, vol. 20, no. 10, pp. 762-772, 1977.

[7] Deitel P. and Deitel H., *Java How to Program*, Prentice Hall, 2003.

[8] Freeman E., Hupfer S., and Arnold K., *JavaSpaces Principles Patterns and Practice*, Addison Wesley, 2004.

[9] Gonzalo N., "A Guided Tour to Approximate String Matching," *Association for Computing Machinery Computing Surveys,* vol. 33, no. 1, pp. 31-88, 2001.

[10] Huston1 L., "Dynamic Load Balancing for Distributed Search, High Performance Distributed Computing," *in Proceedings of 14th IEEE International Symposium*, USA, pp. 157-166, 2005.

[11] Jin H. and Bernard A., "High Performance Pattern Matching with Dynamic Load Balancing on Heterogeneous Systems," *in Proceedings of 14th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, USA, pp. 285-290, 2006.

[12] Jonathan L., "Analysis of Fundamental Exact and Inexact Pattern Matching Algorithms," *Technical Document*, Stanford University, 2004.

[13] Juval L., *Programming NET Components*, O'Reilly and Associates, 2005.

[14] Knute D., Morris J., and Pratt V., "Fast Pattern Matching in Strings," *SIAM Journal of Computing*, vol. 6, no. 2, pp. 323-350, 1977.

[15] Lecroq T. and Charras C., *Handbook of Exact String Matching Algorithms*, King's College London Publications, 2004.

[16] Mattern F., "Tuning Distributed Control Algorithms for Optimal Functioning," *Computer Journal of Global Optimization*, vol. 2, no. 2, 1992.

[17] Mhashi M., Rawashdeh A., and Hammouri A., "A Fast Approximate String Searching Algorithm," *Computer Journal of Science Publication*, vol. 1, no. 3, pp. 405-412, 2005.

[18] Michailidis P. and Margaritis K., "On-Line Approximate String Searching Algorithms: Survey and Experimental Results," *International Journal of Computer Mathematics,* vol. 79, no. 8, pp. 867-888, 2002.

[19] Badoiu M. and Indyk P., "Fast Approximate Pattern Matching with Few Indels via Embeddings," *in Proceedings of 15th Annual ACM-SIAM Symposium on Discrete Algorithms*, Louisiana, pp. 651-652, 2004.

[20] Newmarch J., *A Programmer's Guide to Jini™ Technology*, Springer-Verlag New York, 2000.

[21] Panagiotis D. and Konstantinos G., "Performance Analysis of Approximate String Searching Implementations for Heterogeneous Computing Platform," *in Proceedings of IEEE International Conference on Parallel Processing Workshops*, Berlin, pp. 173-180, 2003.

[22] Sedgewick R., *Algorithms*, Addison Wesley, 1983.

[23] Galvin S. and Gagne P., *Operating Systems Concepts*, John Wiley, 2004.

[24] Simon Y. and Inayatullah M., "Improving Approximate Matching Capabilities for Meta Map Transfer Applications," *in Proceedings of 3rd International Symposium on Principles and*

Practice of Programming in Java, Dublin, pp. 143-147, 2004.
[25] Steven S., *The Algorithm Design Manual*, Springer-Verlag, New York, 1997.
[26] Sun Microsystems Incorporation, "Multithreading in the Solaris Operating Environment," *Technical Paper*, 2002.
[27] Thierry L. and Christian C., *Handbook of Exact String Matching Algorithms*, ACM Medium: Paperback, 2004.
[28] Yates R. and Neto B., *Modern Information Retrieval*, First Edition, Addison Wesley, 1999.

Najib Kofahi is a professor of computer science at Yarmouk University (YU). He received his PhD in computer science from the University of Missouri-Rolla, USA in 1987. Currently, he is the Dean of the Faculty of Information Technology and Computer Sciences at YU, Jordan. He has several journal and conference research publications in a number of research areas including e-learning, operating systems, distributed systems, and performance evaluation. His teaching interests focus on operating systems, algorithms and data structures, computer organization and assembly language programming, and computer architecture.

Ahmed Abusalama is currently working as a lecturer at Al Qassim University, The Kingdom of Saudi Arabia (KSA). He is teaching various compute modules at the preparatory year college. He received his Master in computer science from Yarmouk University, Jordan in 2006. He received his Bachelor degree in computer science from Al Albayt University, Jordan in 2004. He worked as a lecturer at Yarmouk University from 2006 to 2007 where he taught advanced object oriented analysis and design and basic computer skills courses (windows, MS office and internet).