Performance Evaluation of Multiple Approximate String Matching Algorithms Implemented with MPI Paradigm in an Experimental Cluster Environment

Panagiotis D. Michailidis and Konstantinos G. Margaritis
Department of Applied Informatics, University of Macedonia
156 Egnatia str, 54006 Thessaloniki, Greece
Email: {panosm, kmarg}@uom.gr

Abstract

This paper describes and analyzes two high performance multiple string searching algorithms that we have optimized for cluster of heterogeneous workstations. The proposed parallel algorithms are based on dynamic master - worker programming paradigm. Experimental results show that these parallel algorithms are efficient: their search time is cut down significantly, whereas their communication cost stay at a low level.

1. Introduction

Multiple string searching is the computationally intensive kernel of many security and network applications like search engines, intrusion detection systems, web filtering, virus scanners and spam filters. The multiple string matching problem can be defined as follows: Given a text $T = t_1 t_2 ... t_n$ of length $n$ and a set of $r$ patterns (a dictionary) $P = \{p^1, p^2, ..., p^r\}$, where each $p^i$ is a string $p^i = p^i_1 p^i_2 ... p^i_m$ of length $m$ over a finite character set $\Sigma$ and task is to find all occurrences of any of the patterns in the text. Moreover, some single-pattern search algorithms resort to multipattern searching of pattern pieces. Depending on the application, $r$ may vary from a few to thousands of patterns. The naive approach to this problem is to perform $r$ separate searches with one of the sequential algorithms [5]. This leads to a total worst-case complexity of $O(|P|)$ for the preprocessing, where $|P|$ is the sum of the lengths of the strings in $P$ and $O(r \times n)$ for the search. The multipattern problem has received much less attention, not because of lack of interest but because of its difficulty. There exist sequential algorithms that search too few patterns [1, 2, 5]. However, no efficient algorithm exists to search for many patterns in a reasonable amount of time.

High performance string searching is traditionally dominated by hardware-based techniques. In most cases, solutions employ Field Programmable Gate Arrays (FPGAs) or Application Specific Instruction Processors (ASIPs) [3, 8, 4] and adopt algorithms which try to exploit the large amount of parallelism available on these devices. Most FPGA solutions provide relatively high performance with small dictionaries. Traditional software solutions being capable of representing large dictionaries but offering limited performance.

In this paper we propose a class of high performance multiple approximate string matching solutions on a heterogeneous cluster architecture. Our goal is to study data allocation and workload balancing issues in order to improve the parallel execution of multiple approximate string matching algorithms on a cluster of heterogeneous workstations. It is not our intention to prove that the use of a cluster is better than the use of a specific purpose machine nor to make any change to the multiple approximate string matching algorithms. We must note that the target heterogeneous cluster platform is composed of computational resources or workers with different computing powers and workers are connected to the master by links of same capacities.

2. Parallel Algorithms

In this section we present two parallel multipattern approximate string matching algorithms based on the master - worker programming paradigm or more precisely for a computational scheme where the master assigns data and computations to other resources, the workers. The proposed algorithms take into account two criteria: the dynamic load balancing strategy and the data allocation strategy. In dynamic load balancing strategy, the text is partitioned into small subtexts and these subtexts are assigned dynamically to idle workstations in order to keep all the workstations busy. The size of each subtext is $sb + m - 1$ successive characters of the complete text where $sb$ is the optimal block size. There is an overlap of $m - 1$ pattern characters between successive subtexts. The block size is an important
parameter which can affect the overall performance. More specifically, this parameter is directly related to the I/O and communication factors. For this reason, we use a block-oriented approach to partition the text into blocks of size \( s_b \).

On the other hand, the data allocation strategy also is divided into two categories: allocation of subtexts and allocation of text pointers. In allocation of subtexts, the subtexts that are obtained by dynamic load balancing strategy are distributed to corresponding workstations. In allocation of text pointer, some master workstation of the cluster has a text pointer (i.e. \( \text{offset} \)) that shows the current position in the text and the master distributes the text pointers instead of the subtexts to corresponding workers in order to reduce the communication overhead. Dynamic allocation of subtexts and dynamic allocation of text pointers are presented in next subsections.

2.1. Dynamic Allocation of Subtexts

Our dynamic allocation of subtexts is based on the following assumptions: First, the number of workstations in the cluster is denoted by \( p \) and we assume that \( p \) is power of 2. Further, the workstations have an identifier \( \text{myid} \) and are numbered from 1 to \( p \). Second, we load into the local disk of the master workstation all blocks of the text collection and the multiple patterns. The algorithm of the dynamic allocation of subtexts that is called A1 is made of two parts: Algorithm 1 outlines the program of the master, while Algorithm 2 is the program of each worker.

The master program is responsible for broadcasting all patterns to workers and distributing chunks of the text collection to corresponding workers during the first pattern only. The master terminates when there are not any patterns. Note that in order to terminate, the number of tasks outstanding in the workers is counted (\( \text{active} \)). It is also possible simply to count the number of results returned. The worker program receives all patterns and corresponding subtexts. It also uses the procedure \( \text{dp} \) for performing a sequential dynamic programming algorithm [7] between the pattern and the corresponding chunk of the text that are received in order to generate the number of occurrences (\( \text{count} \)). Note that each worker stores the subtexts that are received in the local disk so that they are used for searching next patterns. This operation is executed during the first pattern only.

2.2. Dynamic Allocation of Text Pointers

Our dynamic allocation of text pointers is based on the following assumptions: First, we load into the local disk of the master workstation all chunks of the text collection and the multiple patterns and second, the master workstation has a pointer that shows the current position in the text collection. The algorithm of the dynamic allocation of text pointers that is called A2 is made of two parts: Algorithm 3

---

**Algorithm 1: Master program**

```plaintext
while not at end of the pattern file do
    read current pattern from the local disk;
    broadcast current pattern to all workers;
    if current pattern is first then
        \( \text{offset} \leftarrow 0; \text{active} \leftarrow 0; \)
        for \( i = 0 \) to \( p \) do
            read current text chunk of size \( s_b + m - 1 \) chars starting from the position \( \text{offset} \) of file;
            send current text chunk to worker \( P_i \);
            \( \text{active} \leftarrow \text{active} + 1; \)
            \( \text{offset} \leftarrow \text{offset} + (s_b + m - 1); \)
        end
        repeat
            receive \( \text{count} \) from any worker;
            \( \text{active} \leftarrow \text{active} - 1; \)
            sender \( \leftarrow P_{\text{anyworker}}; \)
            if \( \text{offset} \) is not at end of the text file then
                read current text chunk of size \( s_b + m - 1 \) chars starting from the position \( \text{offset} \) of file;
                send current text chunk to worker \( P_{\text{sender}}; \)
                \( \text{active} \leftarrow \text{active} + 1; \)
                \( \text{offset} \leftarrow \text{offset} + (s_b + m - 1); \)
            else
                send a terminator message to worker \( P_{\text{sender}}; \)
            end
            until \( \text{active} > 0; \)
        until \( \text{active} > 0; \)
    else
        \( \text{active} \leftarrow p - 1; \)
        repeat
            receive \( \text{count} \) from any worker;
            \( \text{active} \leftarrow \text{active} - 1; \)
            until \( \text{active} > 0; \)
        end
    end
    for \( i = 0 \) to \( p \) do
        send a terminator message to worker \( P_i; \)
    end
end
```

---
Algorithm 2: Worker program

while true do
    receive pattern from master;
    if there is terminator message then break;
    if pattern is first then
        pieces ← 0;
        while true do
            receive current text chunk from master;
            if there is terminator message then break;
            pieces ← pieces + 1;
            count ← dp(pattern, textchunk);
            store current text chunk in local disk;
            send count to master;
        end
    else
        for i = 0 to pieces do
            count ← dp(pattern, textchunk);
            send count to master;
        end
    end
end

Algorithm 3: Master program

while not at end of the pattern file do
    read current pattern from the local disk;
    broadcast current pattern to all workers;
    if current pattern is first then
        offset ← 0; active ← 0;
        for i = 0 to p do
            send offset to worker \( P_i \);
            active ← active + 1;
            offset ← offset + (sb + m - 1);
        end
        repeat
            receive count from any worker;
            active ← active - 1;
            sender ← \( P_{\text{anyworker}} \);
            if offset is not at end of the text file then
                send offset to worker \( P_{\text{sender}} \);
                active ← active + 1;
                offset ← offset + (sb + m - 1);
            else
                send a terminator message to worker \( P_{\text{sender}} \);
            end
        until active > 0;
    else
        active ← p - 1;
        repeat
            receive count from any worker;
            active ← active - 1;
        until active > 0;
    end
end

for i = 0 to p do
    send a terminator message to worker \( P_i \);
end

outlines the program of the master, while Algorithm 4 is the program of each worker.

3. Experimental Results

In our experiments we have been implemented these algorithms A1 and A2 in C programming language using the MPI communication library [6] and tested them on a cluster of 16 heterogeneous workstations running Linux operating system (RedHat 7.1). The communication between workstations is performed through a 100 Mbps Fast Ethernet network. The MPI implementation used on the network is MPICH version 1.2.

During all experiments, the cluster of workstations was dedicated. Moreover, to get reliable performance results 5 executions occurred for each experiment and the reported values are the average ones. Finally, the text collection we used was composed of documents, which were portion of the various web pages. We also selected simple patterns randomly from the same text collection.

Figure 1 shows the execution times for MPI-based A1 and A2 algorithms on 2, 4, 8 and 16 workstations for text size of 40MB and block size of 100,000 characters. Similarly, Figure 2 shows the execution times for MPI-based A1 and A2 algorithms on 2, 4, 8 and 16 workstations for text size of 40MB and block size of 500,000 characters. As can be seen from Figures that the total execution time is decreased as the number of workstations is increased. In other words, for constant set of patterns there is an expected inverse relation between parallel execution time and the number of workstations. We must note that the obtained time of communication is increased whereas the string matching time is decreased as the number of workstations is increased. As a general conclusion we say that the total communication and I/O times are much more lower than the total searching times. This means that there is more important to improve local computations than communication rounds.

Further, we observe that the parallel execution time is increased as the set of patterns is increased. This result means that when set of patterns is large enough, more time is spent on string searching than communicating with the master workstation. Further, we observe that the performance of a parallel single pattern matching algorithm can be affected by many unexpected affects, such as system load, cache/memory access, query property, etc. Evaluating par-
Algorithm 4: Worker program

while true do
    receive pattern from master;
    if there is terminator message then break;
    if pattern is first then
        pieces ← 0;
        while true do
            receive offset from master;
            if there is terminator message then break;
            read current text chunk from the local disk of the master (through NFS) $sb + m − 1$ chars of the file starting from position of $fset$;
            pieces ← pieces + 1;
            count ← dp(pattern, textchunk);
            store current text chunk in local disk;
            send count to master;
        end
    else
        for $i = 0$ to pieces do
            count ← dp(pattern, textchunk$_i$);
            send count to master;
        end
    end
end

Parallel multiple patterns can significantly decrease these unexpected affects. Moreover, we must note that both these parallel algorithms run faster on a larger number of workstations and the difference in execution time between 8 and 16 workstations is greater than the difference between 2 and 4 workstations. This is due to the fact that the communication requirements of two algorithms are low, i.e. two algorithms distribute subtexts or pointers to all workers during the first iteration (pattern) only.

From these experimental Figures we also observe that block size is an important parameter which can affect the total performance of two parallel algorithms. More specifically, the block size affects the communication and I/O phases of the algorithms. In other words, high communication requirements and high cost of reading of data from the local disk of the master workstation are obtained for small values of block size such as in Figure 1. These high communication requirements mean that more time is spent on distributing of subtexts or text pointers or collecting of results from all workers for small values of block size. However, the execution time of the two parallel algorithms A1 and A2 is improved slightly as the block size is increased because low communication requirements and low I/O cost are obtained as in Figure 2. Therefore, we conclude that there is an optimal value of block size nearly to 500,000 characters for the parallel algorithms that this value produces the better performance.

When we compare the performance of two parallel algorithms, we conclude that the algorithm A1 gives better results than the algorithm A2. Although the reduced communication cost of the algorithm A2 in relation to the algorithm A1 (i.e. the algorithm A2 distributes text pointers instead of distributing subtexts) there is performance degradation. The additional overhead of the algorithm A2 is due to the fact that the text collection is stored in the local disk of the master instead of the local disks of workers and therefore there is delay in reading of text collection through the network or NFS.

Figure 3 shows the execution times for the two parallel algorithms A1 and A2 against the number of workstations for text size of 80MB and block size of 500,000 characters. The 80MB is obtained by doubling the old text size. We observe that as the text size is increased, the parallel execution time of the two proposed algorithms is increased for both sets of 100 and 500 patterns. We must note that as the text size is increased, the time of communication, I/O and string searching phases is increased whereas the time of the other phases remain constant. However, more time is spent on string searching phase than communicating with the master workstation and reading of the text collection. In this case the communication cost between the master workstation and worker workstations is increased slightly as the text size is increased but this communication overhead is not seen to affect the overall performance of the two algo-
Figure 2. The execution times for MPI based A1 and A2 algorithms for a set of 100 and 150 patterns and block size of 500,000 characters

Figure 3. The execution times for MPI based A1 and A2 algorithms for a set of 100 and 150 patterns and text size of 80MB

4. Conclusions

In this paper we proposed and evaluated two parallel algorithms for multiple approximate string matching using the master - worker programming paradigm and data allocation schemes. These algorithms have been implemented and tested on a cluster of 16 heterogeneous workstations connected together using 100 Mbps Fast Ethernet network. As a general conclusion we can say that testing the proposed parallel algorithms on a cluster environment indicates that varying parameters such as the number of workstations, block size, number of patterns and text size can produce different performances.

The specific performance conclusions are: The obtained experimental results show that the algorithms A1 and A2 are efficient because their search time is cut down significantly and their communication cost stay at low level. The experimental results also show the algorithm A1 outperforms algorithm A2. We believe that the algorithm A2 could be improved and it is outperformed algorithm A1 whether the chunks of the text collection are stored in the local disks of the workers and therefore we alleviate the delay in reading of the text collection through the NFS.

We plan to develop a theoretical performance model in order to confirm the experimental behavior of the proposed parallel algorithms on a heterogeneous cluster. Further, this model can be used to predict the execution times for the two parallel algorithms on larger clusters and problem sizes.

References