Parallel Implementation of Exact Two Dimensional Pattern Matching Algorithms using MPI and OpenMP

Charalampos S. Kouzinopoulos and Konstantinos G. Margaritis
Parallel and Distributed Processing Laboratory
Department of Applied Informatics, University of Macedonia
156 Egnatia str., P.O. Box 1591, 54006 Thessaloniki, Greece
E-mail: {ckouz,kmarg}@uom.gr

Abstract—The need for processing power is constantly increasing as more processing-demanding and time-critical applications appear. Parallel processing has emerged as an efficient and cost-effective solution, since low-end workstations and multiprocessors are cheaply available in the commodity market. In this paper, experimental results are presented on the parallel processing of the Naive, Karp and Rabin, Zhu and Takaoka, Baeza-Yates and Regnier and the Baker and Bird exact two dimensional on-line pattern matching algorithms. The algorithms are implemented using MPI and OpenMP, the two most widely used APIs for distributed and shared memory parallelization respectively. The performance of the parallel implementations is evaluated when used on a homogeneous cluster of workstations and a multicore processor.

Index Terms—pattern matching, algorithms, parallelization, shared memory, distributed memory, OpenMP, MPI

I. INTRODUCTION

Parallel computing can be classified into two basic architectures depending on the way the communication between the processing elements occurs: the distributed memory architecture and the shared memory architecture. Distributed memory parallel computers exchange information by sending and receiving messages via a communications network while shared memory parallel computers (most commonly multiprocessor systems and/or multicore processors) have a shared access to a common memory area.

The most widely used API for shared memory parallel processing is OpenMP, a set of directives, runtime library routines and environmental variables that is supported on a wide range of multicore systems, shared memory processors, clusters and compilers [20]. OpenMP consists of a set of specifications for parallelizing programs on shared memory parallel computer systems without the explicit need for threads management. The first API specification was published in 1997 while updated specifications were released in 2000, 2002 and 2005. The current revision is version 3.0, released in 2008 [26]. To parallelize an existing algorithm, the parallel parts it contains must be identified; the speed-up to the execution will then be proportional to the level of the algorithm’s parallelism. The identification of parallelism includes finding instructions, sequences of instructions, or even large regions of code that may be executed concurrently by different processors [10].

According to Amdahl’s Law [2], the maximum speed-up that can be achieved will then be equal to

\[ \frac{1}{1 - P + \frac{P}{N}} \]  

(1)

where N is the number of processing elements and P the proportion of the algorithm that can be parallelized. After the parallel parts have been identified, the OpenMP implementation is responsible to actually create, synchronize and destroy the threads needed to perform the processing.

The de facto standard for distributed memory processing is MPI, an API specification that describes the way the workstations of a cluster can communicate. As opposed to the shared memory architecture, each workstation has its own private memory space that is not accessible directly by other nodes. To achieve communication, MPI addresses the message-passing parallel programming model, according to which data is moved from the address space of one process to that of another process through cooperative operations on each process [28]. The two key attributes of message passing as mentioned in [13] are the partitioned address space that is distributed between each node enforcing the communication and synchronization between them as well as, similar to the OpenMP paradigm, the support for only explicit parallelization.

Experiments on distributed and shared memory parallelization have been reported across many areas of computing including string and pattern matching [16][22], bioinformatics [12], computer forensics [6], data mining [14], biomedical image processing [24], image classification [23] and linear algebra [19] but to the best of our knowledge no prior work involving the performance evaluation of two dimensional pattern matching algorithms using both OpenMP and MPI exist. The contribution of this paper is the implementation of some well known exact two dimensional pattern matching algorithms in parallel using the distributed memory and shared address paradigms and a performance evaluation of the parallel implementation.

II. PATTERN MATCHING

Pattern matching is an important problem in text and image processing and is used to locate the appearances of an array
(the so called “pattern”) in an another array (the so called “text”) of equal or greater size. It is useful for content based information retrieval from image databases, image analysis and medical diagnostics. It is also used by some methods of detecting edges, where a set of edge detectors is matched against a picture and by some OCR systems [21] and is important in areas like motion analysis, cartography, aerial photography, remote sensing, document analysis and object tracking [27].

The exact two dimensional pattern matching problem is defined in [3] as: let \( \Sigma \) be an alphabet, given a text array \( T[n \times n] \) and a pattern array \( P[m \times m] \), report all locations \([i, j]\) in \( T \) where there is an occurrence of \( P \), i.e. \( T[i + k, j + l] = P[k, l] \) for \( 0 \leq k, l \leq n \) and \( m \leq l \leq n \). For the experiments of this paper, the Naive, Karp and Rabin [18], Zhu and Takaoka [25], Baeza-Yates and Regnier [9] and the Baker and Bird [5] algorithms were used, as they are some of the most widely used algorithms in the area of two dimensional pattern matching.

Naive is the most straightforward algorithm for pattern matching. It simply attempts to match the pattern in the target at successive positions from left to right and top to bottom by using a window of size \( m \times m \). In case of success in matching an element of the pattern, the next element is tested against the text until a mismatch or a complete match occurs. After each unsuccessful attempt, the window is shifted by exactly one position to the right, and the same procedure is repeated until the end of the target is reached.

Karp and Rabin [18] created an algorithm that uses a hashing technique to search for occurrences of a pattern in a text. This technique involves the computation of a hash value for each possible \( m \times m \) character substring in the text and the check of its equality to the hash value of the pattern. To perform the calculation of the hashes efficiently for the hashing function \( h(k) = k(\text{mod } q) \), where \( q \) is a large prime, a method called rolling hash is used. This method is based on computing the hash function of a substring starting at position \( i \) given the value of the substring starting at position \( i - 1 \). To keep the hash values small, some strings have to be assigned the same hash number and consequently strings could exist that although have the same hash value as the pattern, do not match. To be sure that there is a match, a direct comparison of that string with the pattern should be done.

Zhu and Takaoka [25] presented two algorithms to reduce the two dimensional matching problem to a string matching problem so that efficient string matching algorithms can be applied to the text. The first algorithm is a combination of the Karp and Rabin and Knuth-Morris-Pratt [17] algorithms while the second is based on the Karp and Rabin and Boyer-Moore [8] algorithms. The idea behind both algorithms is to use the hash function method proposed in the Karp and Rabin algorithm vertically. The two dimensional arrays of characters of text and pattern are translated to one dimensional arrays of numbers respectively and the text is then examined for the occurrences of the pattern using either the Knuth-Morris-Pratt or the Boyer-Moore algorithm.

Baker [5] and independently Bird [7] introduced the first worst-case linear time algorithm for two dimensional pattern matching. The idea behind the algorithm is to run a finite automaton in order to perform a linear scan on the text. At each location of the text, the examination consists of two distinct steps: row-matching and column-matching. The row-matching step is used to locate rows of the pattern that match a sub-string of the text. If this step determines that a row \( P_i \) of the pattern occurs at a particular location, at the column-matching step must also be determined if the rows \( P_0, P_1, \ldots, P_i \) occur immediately above \( P_i \) in order to find out if the complete pattern appears as a sub-array of the text. Both of these steps are performed using the Aho-Corasick algorithm [1]. Aho-Corasick treats each row of the pattern as a keyword and constructs a finite state pattern matching machine that can be used to process each row of the text in a single pass.

The Baeza-Yates and Regnier algorithm [9] uses multiple string searching to locate a two dimensional pattern in a two dimensional text. It considers each pattern as \( m \) strings, each one of them with a length of \( m \) and the search is being decomposed into a one-dimensional multi-string search. Essentially, the searching is performed horizontally on only \( n/m \) rows of the text, since these rows cover all possible positions where the pattern may occur. If a match is found starting on the \( k^{th} \) character of the \( i^{th} \) row of the text, then the \( m - i \) above and \( i - 1 \) rows below the current on the text are searched, starting from the same character position, to determine if a complete match occurs. The search is performed by using the Aho-Corasick algorithm as in the Baker and Bird algorithm.

The reader is referred to [15] for a detailed description of some of the algorithms.

III. TEXT PARTITIONING AND DATA DISTRIBUTION

The process of dividing a computation into smaller parts and assigning them to different processes for parallel execution are the two key steps in the design of parallel algorithms [13]. A major source of overhead in parallel systems is the time the processes stay idle due to uneven distribution of load. To decrease the execution time, the available data set must be decomposed and mapped to the available processes in such a way that this overhead is minimized.

There are two available mapping techniques, the static mapping technique and the dynamic mapping technique. Static mapping is commonly used in a homogeneous environment where all processes have the same characteristics while the dynamic mapping is best suited in heterogeneous set-ups. A simple and efficient way to divide the data set into smaller parts, especially when no interaction occurs between neighbour array elements, is by using a line partitioning where the data is divided on a per line basis. That way, a two dimensional array can be divided into chunks of lines in order to minimize the number of overlapping characters needed.

Let \( p \) be the number of available processes and \( r \) the number of parts that a text is decomposed to. In the case of static mapping, each process receives a part consisting of \( \lceil n^2/p \rceil + n(m-1) \) text characters prior to the execution of the algorithms. When a dynamic mapping is used instead, the data set is decomposed into more parts than the available processes
and each process receives $n(sb + m - 1)$ characters where $sb$ is the chosen chunk size during the execution of the algorithms. There is an overlap of $n(m-1)$ characters on each part to ensure that each process has all the data needed, resulting in $rn(m - 1)$ additional characters to be processed for the dynamic mapping and $pn(m - 1)$ for the static.

A number of models exist that define the way of distributing data including the Data-Parallel model, the Task Graph model, the Work Pool model and the Pipeline model. Master-Worker is a communication model that corresponds to the typical master-worker scheme: a node (the master) distributes data pieces to several nodes (the workers). Each worker processes the corresponding data pieces and when finished the results are gathered by the master. For the experiments of this paper was used the Master-Worker model, as it was concluded in [11] that is the most appropriate model for pattern matching on either message passing or shared address space systems.

To reduce the communication overhead during the distribution of data between the master and the worker nodes, a dynamic allocation of text pointers was used as detailed in [16]. Both text and patterns resided locally on each node. The master node then distributed a pointer offset to each worker to indicate the area of the text that was assigned to each processing element.

**IV. IMPLEMENTATION AND EXPERIMENTAL RESULTS**

In this section are presented implementation details and an evaluation of the way the performance of the algorithms is affected by a number of parameters such as the chunk size, the distribution method, the number of processing elements and the size of the text, when using both shared memory and distributed memory parallelization.

The experiments were performed locally on two systems: an Intel Core 2 Quad CPU with four processor cores, a 2.40GHz clock speed and 8 Gb of memory for the OpenMP experiments and a homogeneous computer cluster consisting of 15 Intel Xeon CPU workstations with 3.00GHz clock speed, 1 Gb of memory and an HP NC7771 1000T Gigabit Server Adapter for the MPI experiments. The Ubuntu Linux operating system was installed on both systems and during the experiments only the typical background processes ran.

Two sets of data were used to evaluate the performance of the algorithms when executed in parallel, a set of randomly generated binary texts with a dimension of 3000x3000 and 10000x10000 pixels and a collection of uncompressed satellite images from the NASA earth observatory service [29] with a 24-bit colour depth and a dimension of 3000x3000 pixels. The query patterns used were constructed of randomly chosen subsequences from each data set with a pattern size of $m = 25$.

To compare the pattern matching algorithms, the practical running time was used as a measure. Practical running time is the total time in seconds an algorithm needs to find all occurrences of a pattern in a text including any preprocessing time and was measured using the omp_get_wtime() and MPI_Wtime() functions of OpenMP and MPI respectively. To decrease random variation, the running time was measured as an average of 100 runs.

**A. Distributed memory parallelization**

![Graph A](image1.png)

**Fig. 1.** Average running time of all algorithms for different chunk sizes, for random text of size $n=3000$ (A) and $n=10000$ (B) and satellite images (C)

An important parameter that affects both the communication and the processing phase of the parallel implementation of the algorithms is the size of each chunk that is distributed from the master to the worker nodes. Depending on the type of data and the distribution method used, an optimal performance can be achieved when the appropriate chunk size is determined. Selecting a small size resulted in an increased communication cost between the master and the worker nodes and an additional overhead in the form of additional overlapping characters while a bigger chunk size had the benefit of a reduced communication cost but at the same time introduced
uneven distribution of data, even on homogeneous networks. Figure 1 presents the average running time of the Naive, Karp and Rabin, Zhu and Takaoka, Baeza-Yates and Regnier and the Baker and Bird exact two dimensional on-line pattern matching algorithms when using MPI for a static and dynamic distribution and a variable chunk size. From the Figure can be observed that when a dynamic distribution and a chunk size of 100n text characters was used, the algorithms had the best performance for either type of text.

Figure 2 depicts the way the running time of the algorithms was affected when executed in parallel on 1 to 10 workstations, for random texts of size n=3000 and n=10000 and satellite images, a dynamic distribution and a chunk size of 100 lines of text. It is clear that parallel processing of an algorithm on two processors resulted in the approximate doubling of its performance. On each subsequent workstation introduced, the performance of the algorithms increased but with a decreasing rate since the communication cost between the master and the worker nodes also increased. As mentioned in [16], on a distributed memory system there is an inverse relation between the parallel execution time and the number of workstations, since the total communication time is much lower than the processing time on each node.

From the same Figure can be also seen that the size and type of text also affected the performance of the algorithms. The running time of the algorithms increased by an average of $2n^2$ instead of the expected $n^2$ when a text of larger size was used. This indicates that the size of the text is an important performance factor that affects not only the computational time of the algorithms but the communication time as well. It can be also concluded that on average, the performance of the algorithms was slightly better on real data comparing to the case were random data were used. The satellite images used a 24-bit alphabet that caused more mismatches between pattern and text than the randomly generated texts.

B. Shared memory parallelization

As opposed to distributed memory parallelization, shared memory parallelization does not actually involve a distribution of data since all threads have access to a common memory area. OpenMP provides the programmer with a set of scheduling clauses to control the way the iterations of a parallel loop are assigned to threads, the static, dynamic and guided clauses. The scheduling clauses when used with the chunk size parameter, can greatly affect the performance of the algorithms. With the static schedule clause, the assignment of iterations is defined before the computation while with both the dynamic and guided clause, the assignment is performed dynamically at computation time [4]. The three scheduling clauses can take as an argument a chunk size or a default size is selected for the chunk if no such argument is provided.

When n is not specified, OpenMP divides the data set into p chunks of equal size for the static clause, where p is the number of processes, while for the dynamic and guided clause the default chunk size is 1 iteration per thread, which provides the best level of workload distribution but at the same the biggest overhead due to synchronization when scheduling work [30]. Figure 3 presents the average running time of all algorithms when executed using OpenMP for the static, dynamic and guided scheduling clauses and for a variable chunk size for types of text. It is clear that the static scheduling clause with the default chunk size was faster for either type of text.

Figure 4 presents the way the performance of all algorithms was affected when parallel processed using OpenMP for 1 to 4 threads, a static distribution and the default chunk size. As can be seen, the performance of the algorithms improved with each additional thread. Parallel processing of an algorithm on two processors resulted in almost doubling its performance.
and as with MPI, for each subsequent thread, the performance of the algorithms increased with a decreasing rate.

It can be also seen that similar to the MPI experiments, the size of the text also affected the performance of the algorithms. When a text of size $n=10000$ was used, the running time increased by an average of $n^2$ as opposed to $2n^2$ of MPI, since no communication cost involved. Additionally, when satellite images were used, the performance of the algorithms improved up to 1.2 times on average.

When comparing the parallelization achieved by OpenMP and MPI on the same algorithms and data sets it can be seen that OpenMP is more efficient, since the communication cost that is involved in a cluster of workstations is not existent between multiple processing elements in the same node. The drawback of the shared memory parallelization is the amount of parallelization that can be achieved since only limited cores are available up to now on a single processor.

V. CONCLUSIONS

In this paper, a parallel implementation of the Naive, Karp and Rabin, Zhu and Takaoka, Baeza-Yates and Regnier and the Baker and Bird exact two dimensional on-line pattern matching algorithms was presented using MPI and OpenMP and the performance of the algorithms was measured when executed on a homogeneous computer cluster and a multicore system.

It was determined that for the chosen partition method and types of data, the selected algorithms had the best performance when a dynamic distribution and a chunk size of 100 lines of...
text was used for the distributed memory parallelization and the static scheduling clause with the default chunk size for the shared memory parallelization.

Additionally it was shown that the parallel processing of the algorithms on either architecture was very effective when two processing elements were used but on each additional processing element, the performance improved with a decreasing rate. Finally it was concluded that using data sets with big alphabet sizes actually improved the performance of the algorithms due to the increase in the number of mismatches in the processing phase of the algorithms.

Future research in the area of two dimensional pattern matching could focus on further speeding up the existing algorithms. It would be interesting to examine the way the performance of the algorithms is affected when other types of parallelization are applied like processing on graphic processing units or on hybrid OpenMP/MPI systems.

REFERENCES