PARALLEL IMPLEMENTATIONS FOR STRING MATCHING PROBLEM ON A CLUSTER OF DISTRIBUTED WORKSTATIONS

Panagiotis D. Michailidis\(^1\) and Konstantinos G. Margaritis\(^1\)

\(^1\)Parallel and Distributed Processing Laboratory
Department of Applied Informatics
University of Macedonia
156 Egnatia str., P.O. Box 1591
GR-54006 Thessaloniki, Greece

Abstract
We present three parallel implementations for solving the string matching problem. The advantages of these three parallel implementations are combined and incorporated into our new hybrid parallel implementation to produce optimal performance than either of the original three. We compare the performance of the four implementations on a cluster of workstations, interconnected by a high-performance Fast Ethernet network. Moreover, we have checked that by using the theoretical performance model we can predict the behaviour of the experimental results of the parallel string matching implementations.

Keywords - string matching, cluster of workstations, Message Passing Interface (MPI), performance evaluation, performance prediction model

1. INTRODUCTION

String matching has been one of the most extensively studied problems in computer science since it performs important tasks in many applications including information retrieval, Web search engines, library systems, DNA sequencing, signal processing, error correction, speech and pattern recognition and several other fields. Especially with the introduction of search engines dealing with tremendous amount of textual information presented on the World Wide Web (WWW) as well as the research on DNA sequencing, this problem deserves special attention and any improvements to speed up the process will benefit these important applications.

The basic string matching problem can be defined as follows. Let a given alphabet (a finite sequence characters) \(\Sigma\), a short pattern string \(P = P_1P_2...P_m\) of length \(m\) and a large text string \(T = T_1T_2...T_n\) of length \(n\), where both the pattern and the text are sequences of characters from \(\Sigma\), with \(m \leq n\). The string matching problem consists of finding one or more generally all the exact occurrences of a pattern \(P\) in a text \(T\). Survey and experimental results of well known sequential algorithms for this
string matching problem can be found in (Crochemore & Rytter, 1994; Michailidis & Margaritis, 2001; Michailidis & Margaritis, 1999; Stephen, 1994).

As current free textual databases are growing almost exponentially with the time, the string matching problem becomes impractical to use the fastest sequential algorithms on a conventional sequential computer system. To improve the performance of string matching on large text databases, we believe that recent advances in distributed processing techniques is currently mature enough and can provide powerful computing means convenient for overcoming this string matching problem. There are two general parallelization methods that can be used to search large text databases: one is based on the fine-grain parallelization of the string matching algorithm (Foster & Kung, 1980; Carroll et al., 1988; Ranganathan & Sastry, 1994; Margaritis & Evans, 1997) and the other is based on the distribution of the computation of character comparisons on supercomputers or network of workstations. In (Cringean et al., 1988; Cringean et al., 1991) an exact string matching implementation have been proposed and results are reported on a transputer based architecture. Parallel implementations of string matching algorithm under cluster environment are not available in the literature.

In this paper we present three parallel search methods using static and dynamic allocation of subtexts which solve the string matching problem. Further, we propose a new hybrid parallel search method that combines the advantages of static and dynamic methods in order to reduce the load imbalance and communication overhead. The four methods are implemented in C in conjunction with the Message Passing Interface (MPI) library (Gropp et al., 1994; Pacheco, 1997; Snir et al., 1996; Wilkinson & Allen, 1999) which follows the SPMD (Single Program Multiple Data) model and run on a high performance network of distributed workstations. Moreover, we present an analytical performance prediction model that can be used to predict the execution time, speedup and similar performance metrics of the parallel string matching implementations running on a cluster of workstations.

The remainder of this paper is organized as follows: In section 2, an outline of the text partitioning strategy and the parallel implementations are presented. Experimental results of the parallel implementations in the cluster environment used are presented in Section 3. In Section 4, the performance modeling of the parallel implementations is presented. Then, we compare these results with the experimental results. Finally, Section 5 contains our conclusions and future research issues.

2. PARALLEL STRING MATCHING IMPLEMENTATIONS

In this section involves the implementations on a cluster of workstations of string
matching algorithm.

2.1 Text partitioning

The exact string matching problem can achieve data parallelism with the following simple data partitioning technique: we decompose the text into \( r \) subtexts, where each subtext contains 

\[
 k = \left\lceil \frac{n - m + 1}{r} \right\rceil + m - 1
\]

successive characters of the complete text. There is an overlap of \( m - 1 \) pattern characters between successive subtexts, i.e. a redundancy of \( r(m - 1) \) characters. Alternatively it could be assumed that the database of an information retrieval system contains \( r \) independent documents. Therefore, in both cases the above partitioning yields a number of independent tasks each comprising some data (i.e. a pattern and a large subtext) and a sequential string matching procedure that operates on that data. Further, each task completes its string matching operation on its local data and returns the number of occurrences. Finally, we can observe that there are no communication requirements among the tasks but only global (or collective) communication is required.

The main issue that we have to address is how the several tasks (or \( r \) subtexts) are mapped or distributed to multiple processors for concurrent execution. In (Cringean et al., 1988) discuss several different ways of distributing the database across a multiprocessor network and conclude that the master-worker (or processor farm or processor pool) approach to distributed computation is the most appropriate for the string matching application. In the master-worker approach, the master process initiates the searching and then creates a series of separate, but identical worker processes, which perform any sequential string matching on local data concurrently. One of the most interesting and general aspects of this approach is that the same program can operate with any number of workers, i.e. it follows the SPMD model. We generally assign each worker to a specific processor, which limits the total number of workers to the number of processors available on our parallel machine (currently 9 on our network of workstations).

Let \( p \) be the number of processors in the network and \( r \) the number of subtexts of the whole text database. According to the above text partitioning we define as follows: if \( r = p \) then each subtext contains 

\[
 k = \left\lceil \frac{n - m + 1}{r} \right\rceil + m - 1
\]

characters. This case is called static allocation of subtexts. Further, if \( r \gg p \) then each subtext contains 

\[
 k = b + m - 1
\]

characters, where \( b \) is the optimal block size. This case is called dynamic allocation of subtexts. In next subsections we present a parallel implementation that is based on static allocation of subtexts and two implementations that are based on
dynamic allocation of subtexts using MPI library.

2.2 Static allocation of subtexts

In order to present the implementation that is called P1, we make the following assumptions. First, the processors have an identifier myid and are numbered from 0 to \( p - 1 \), second the each subtext is stored in local disk (or file) of a processor in name file format nametext.#myid in order to be separate and finally, the pattern is stored in main memory to all processors. Figure 1 demonstrates a very simplified version of the static parallel implementation in C-like pseudo-code using MPI library. From the master and the worker sub-procedures it is clear that the application ends only when all the local string matching operations have been completed and their results have been collected by the master processor. Further, it must be noted that the line 3 of the worker sub-procedure calls any sequential exact string matching algorithm. This entire program is constructed so that alternative sequential exact string matching algorithms can be substituted quite easily (Crochemore & Rytter, 1994; Michailidis & Margaritis, 2001; Michailidis & Margaritis, 1999; Stephen, 1994). In this paper we use the Brute Force (in short, BF) and Shift-Or (in short, SO) string matching procedures (Michailidis & Margaritis, 1999; Baeza-Yates & Gonnet, 1992).

The advantage of this simple implementation is low communication overhead. This advantage was achieved, a priori, by the search computation, assigning each worker to search its own subtext independently without have to communicate with the other workers or the master. However, the main disadvantage is the possible load imbalance because of the poor partitioning technique.

2.3 Dynamic allocation of subtexts

Before, we present the dynamic implementation that is called P2, we make the following assumptions: First, the entire text is stored on the local disk of the master processor. Second, the master has a text pointer \( offset \) that shows the current position in the text file. The master reads chunks of \( k \) characters from the text file with begging the \( offset \) and distributes these chunks to workers when they become idle. Finally, the text pointer is updated by the master for the next position of next chunk of text, i.e. \( offset + k - m + 1 \) after the distribution of the previous chunk. Figure 2 demonstrates the implementation of the dynamic load balancing algorithm with allocation of subtexts in C-like pseudo-code using MPI library. This form of the dynamic load balancing terminates when all the chunks of the text have been searched.

The advantage of this dynamic implementation is low load imbalance while the
Main procedure
main()
{
    1. Initialize message passing routines;
    2. If (process==master) then call master(); else call worker();
    3. Exit message passing operations;
}

Master sub-procedure
master()
{
    1. Broadcast the pattern (P) to workers; (MPI_Bcast)
    2. Broadcast the name of text to workers; (MPI_Bcast)
    3. Receive the results (i.e. matches) from all workers and returns
       the total matches; (MPI_Reduce)
    4. Print the total number of occurrences of P in distributed T;
       (i.e. total matches)
}

Worker sub-procedure
worker()
{
    1. Receive the pattern (P) and the name of text from master;
       (MPI_Bcast)
    2. Open the file of the local subtext (T) and store the local
       subtext (T) in memory;
    3. Call matches=search(P,m,T,k); //using any sequential string
       matching approach
    4. Send the results (i.e. matches) to the master; (MPI_Reduce)
}

Figure 1: Static master-worker parallel implementation
disadvantage is higher inter-processor communication overhead.

2.4 Dynamic allocation of text pointers

In this implementation that is called P3, we make the following assumptions: First, the complete text is stored on the local disks of all processors. Second, the master has a text pointer offset and distributes this pointer to workers so that each worker will read \( k \) characters from the file starting from the pointer that receives. Finally, the text pointer is updated by the master for the next position of next chunk of text, i.e. \( offset + k - m + 1 \). Figure 3 demonstrates the implementation of the dynamic load balancing algorithm with pointers in C-like pseudo-code using MPI library.

The advantage of this simple implementation is that reduces the inter-processor communication overhead since each processor in this scheme has an entire copy of the text on the local disk. However, this scheme requires more local space (or disk) requirements, but the size of the local disk in parallel and distributed architectures is large enough.

2.5 Hybrid allocation of subtexts

We know that the static allocation has disadvantage the high load imbalance by uneven distribution of database and the dynamic allocation has disadvantage the high communication overhead. These problems can be eliminated by a preprocessing allocation method, where the subtexts in the database are placed into one of the \( p \) processors so that the difference between the sum of the subtext lengths in the smallest and largest processors is minimized. The procedure for placing the subtexts in the processors is as follows: First, the entire text database is partitioned in many subtexts as the dynamic master-worker model (i.e. \( r \gg p \)). Second, the subtexts are sorted in decreasing length order. Then, starting from the longest one, each subtext is placed in the processor that has the current smallest sum of subtext lengths. In the case of a tie, the smallest numbered processor is selected. After this preprocessing phase, we can apply the parallel implementation of the Figure 1. Therefore, we have a hybrid parallel implementation that ensures the low communication cost and load imbalance. This implementation is called P4.

3. EXPERIMENTAL RESULTS

In this section we present the experimental results for the performance of the four parallel string matching implementations, which are based on static and dynamic master-worker model. These algorithms are implemented in ANSI C programming language (Kernighan & Ritchie, 1988) using the MPI library (Gropp et al., 1994;
Master sub-procedure
master()
{
  1. Broadcast the pattern (P) to workers; (MPI_Bcast)
  2. Let offset=active=0;
  3. Open the file of the entire text;
  4. for(i=1;i<=workers;i++) {
      4.1 Read from the file k characters with beginning the offset
           and store this local subtext (T) in memory;
      4.2 Send the local subtext (T) to i worker; (MPI_Send)
      4.3 active++;
      4.4 offset+=k-m+1;
  }
  5. do {
      5.1 Receive the result (i.e. matches) from s worker;
          (MPI_Recv)
      5.2 active--;
      5.3 total_matches+=matches;
      5.4 if (offset<sizefile) {
          5.4.1 Read from the file k characters with beginning the offset
                   and store this local subtext (T) in memory;
          5.4.2 Send the local subtext (T) to s worker; (MPI_Send)
          5.4.3 active++;
          5.4.4 offset+=k-m+1;
      } 
      5.5 else
          5.5.1 Send to s worker to exit; (MPI_Send)
  } while (active>0);
  6. Close the file of the text;
  7. Print the total number of occurrences of P in distributed T;
     (i.e. total_matches)
}

Worker sub-procedure
worker()
{
  1. Receive the pattern (P) from master; (MPI_Bcast)
  2. while (1) {
      2.1 Receive the local subtext (T) from master; (MPI_Recv)
      2.2 If (tag==die) break;
      2.3 Call matches=search(P,m,T,k); // using any sequential
           string matching approach
      2.4 Send the results (i.e. matches) to the master; (MPI_Send)
  }
}

Figure 2: Dynamic master-worker parallel implementation
Master sub-procedure
master()
{
  1. Broadcast the pattern (P) to workers; (MPI_Bcast)
  2. Let offset=active=0;
  3. for(i=1;i<=workers;i++) {
      3.1 Send the offset to i worker; (MPI_Send)
      3.2 active++;
      3.3 offset+=k-m+1;
    }
  4. do {
      4.1 Receive the result (i.e. matches) from s worker;
      (MPI_Recv)
      4.2 active--;
      4.3 total_matches+=matches;
      4.4 if (offset<sizefile) {
          4.4.1 Send the offset to s worker; (MPI_Send)
          4.4.2 active++;
          4.4.3 offset+=k-m+1;
        }
    4.5 else
        4.5.1 Send to s worker to exit; (MPI_Send)
    } while (active>0);
  5. Print the total number of occurrences of P in distributed T;
      (i.e. total_matches)
}

Worker sub-procedure
worker()
{
  1. Receive the pattern (P) from master; (MPI_Bcast)
  2. while (1) {
      2.1 Receive the offset from master; (MPI_Recv)
      2.2 If (tag==die) break;
      2.3 Open the file of the text (T), reads from this file k
          characters with beginning the offset and store this local
          subext in memory;
      2.4 Call matches=search(P,m,T,k); // using any sequential
          string matching approach
      2.5 Send the results (i.e. matches) to the master; (MPI_Send)
  }
}

Figure 3: Dynamic master-worker parallel implementation with pointers
Pacheco, 1997; Snir et al., 1996; Wilkinson & Allen, 1999) for the point-to-point and collective communication operations.

### 3.1 Experimental environment

The target platform for our experimental study is a personal computer cluster connected with 100 Mb/s Fast Ethernet network. More specifically speaking, the cluster consists of 9 PCs, based on 100 MHz Intel Pentium processors, with 64 MB RAM. The MPI implementation used on the network is MPICH version 1.2. During all experiments, the cluster of computers was dedicated. Finally, to get reliable performance results 10 executions occurred for each experiment and the reported values are the average ones. The text database we used were portion of the various web pages.

The number of processors, the pattern lengths and the several text sizes, can influence the performance of the parallel string matching significantly and thus these parameters are varied in our experimental study.

### 3.2 Comparing the four types of string matching implementations

Before, we present the performance results for the four implementations, we ran an experiment for the effect of block size for the two dynamic master-worker methods. We selected several block sizes for the two dynamic implementations P2 and P3 and determined that the block size nearly \( b=100,000 \) characters produces optimal performance, later experiments are all performed using this optimal value for the P2 and P3. Further, we observed that the worst performance is obtained for very small and large values of block size. This is because small values of block size increase the inter-processor communication, while large values of block size produces poorly balanced load.

Tables 1 and 2 show the execution times for four pattern lengths, for different number of processors and for BF and SO string matching algorithms respectively. Further, Figure 4 presents the speedup factors with respect to the number of processors. We define the speedup \( S_p \) in the usual form

\[
S_p = \frac{T_1}{T_p}
\]

where \( T_1 \) and \( T_p \) are execution times of the same algorithm (implemented for sequential and parallel execution) on 1 and \( p \) processors, respectively. It is important to note that the speedups, which are plotted in Figure 4 are result of average for four pattern lengths.

As we have expected, performance results show that the P2 implementation using dynamic allocation of subtexts is less effective than the other three implementations.
Table 1: Experimental execution times (in seconds) for text size of 3MB using BF string matching algorithm and several pattern lengths

<table>
<thead>
<tr>
<th>m/p</th>
<th>Par.</th>
<th>Implem.</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>P1</td>
<td>1.155</td>
<td>0.707</td>
<td>0.487</td>
<td>0.356</td>
<td>0.276</td>
<td>0.298</td>
<td>0.264</td>
<td>0.230</td>
<td></td>
</tr>
<tr>
<td></td>
<td>P2</td>
<td>1.155</td>
<td>1.355</td>
<td>1.207</td>
<td>1.142</td>
<td>1.090</td>
<td>1.055</td>
<td>1.035</td>
<td>1.020</td>
<td></td>
</tr>
<tr>
<td></td>
<td>P3</td>
<td>1.155</td>
<td>0.623</td>
<td>0.426</td>
<td>0.326</td>
<td>0.266</td>
<td>0.218</td>
<td>0.196</td>
<td>0.175</td>
<td></td>
</tr>
<tr>
<td></td>
<td>P4</td>
<td>1.155</td>
<td>0.622</td>
<td>0.419</td>
<td>0.312</td>
<td>0.252</td>
<td>0.203</td>
<td>0.181</td>
<td>0.160</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>P1</td>
<td>1.111</td>
<td>0.659</td>
<td>0.445</td>
<td>0.326</td>
<td>0.274</td>
<td>0.271</td>
<td>0.246</td>
<td>0.202</td>
<td></td>
</tr>
<tr>
<td></td>
<td>P2</td>
<td>1.112</td>
<td>1.318</td>
<td>1.173</td>
<td>1.108</td>
<td>1.057</td>
<td>1.023</td>
<td>1.003</td>
<td>0.989</td>
<td></td>
</tr>
<tr>
<td></td>
<td>P3</td>
<td>1.112</td>
<td>0.602</td>
<td>0.411</td>
<td>0.319</td>
<td>0.253</td>
<td>0.216</td>
<td>0.191</td>
<td>0.169</td>
<td></td>
</tr>
<tr>
<td></td>
<td>P4</td>
<td>1.111</td>
<td>0.596</td>
<td>0.405</td>
<td>0.301</td>
<td>0.247</td>
<td>0.205</td>
<td>0.183</td>
<td>0.160</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>P1</td>
<td>1.087</td>
<td>0.640</td>
<td>0.434</td>
<td>0.319</td>
<td>0.269</td>
<td>0.266</td>
<td>0.226</td>
<td>0.203</td>
<td></td>
</tr>
<tr>
<td></td>
<td>P2</td>
<td>1.087</td>
<td>1.307</td>
<td>1.165</td>
<td>1.102</td>
<td>1.053</td>
<td>1.019</td>
<td>1.001</td>
<td>0.986</td>
<td></td>
</tr>
<tr>
<td></td>
<td>P3</td>
<td>1.087</td>
<td>0.590</td>
<td>0.403</td>
<td>0.309</td>
<td>0.257</td>
<td>0.217</td>
<td>0.186</td>
<td>0.167</td>
<td></td>
</tr>
<tr>
<td></td>
<td>P4</td>
<td>1.087</td>
<td>0.584</td>
<td>0.396</td>
<td>0.296</td>
<td>0.242</td>
<td>0.197</td>
<td>0.173</td>
<td>0.155</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>P1</td>
<td>1.182</td>
<td>0.698</td>
<td>0.473</td>
<td>0.349</td>
<td>0.290</td>
<td>0.287</td>
<td>0.292</td>
<td>0.222</td>
<td></td>
</tr>
<tr>
<td></td>
<td>P2</td>
<td>1.183</td>
<td>1.353</td>
<td>1.195</td>
<td>1.125</td>
<td>1.070</td>
<td>1.033</td>
<td>1.013</td>
<td>0.996</td>
<td></td>
</tr>
<tr>
<td></td>
<td>P3</td>
<td>1.183</td>
<td>0.637</td>
<td>0.435</td>
<td>0.328</td>
<td>0.270</td>
<td>0.223</td>
<td>0.193</td>
<td>0.171</td>
<td></td>
</tr>
<tr>
<td></td>
<td>P4</td>
<td>1.182</td>
<td>0.631</td>
<td>0.428</td>
<td>0.318</td>
<td>0.266</td>
<td>0.212</td>
<td>0.179</td>
<td>0.161</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4: Speedup of parallel string matching with respect to the number of processors for text size of 3MB using BF and SO algorithms
<table>
<thead>
<tr>
<th>$m/p$</th>
<th>Par. Implem.</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>P1</td>
<td>1.024</td>
<td>0.626</td>
<td>0.434</td>
<td>0.331</td>
<td>0.285</td>
<td>0.263</td>
<td>0.251</td>
<td>0.232</td>
</tr>
<tr>
<td></td>
<td>P2</td>
<td>1.024</td>
<td>1.269</td>
<td>1.136</td>
<td>1.078</td>
<td>1.031</td>
<td>0.999</td>
<td>0.982</td>
<td>0.968</td>
</tr>
<tr>
<td></td>
<td>P3</td>
<td>1.024</td>
<td>0.558</td>
<td>0.382</td>
<td>0.293</td>
<td>0.239</td>
<td>0.196</td>
<td>0.177</td>
<td>0.159</td>
</tr>
<tr>
<td></td>
<td>P4</td>
<td>1.024</td>
<td>0.533</td>
<td>0.366</td>
<td>0.281</td>
<td>0.223</td>
<td>0.192</td>
<td>0.163</td>
<td>0.141</td>
</tr>
<tr>
<td>10</td>
<td>P1</td>
<td>1.024</td>
<td>0.626</td>
<td>0.444</td>
<td>0.329</td>
<td>0.288</td>
<td>0.266</td>
<td>0.251</td>
<td>0.233</td>
</tr>
<tr>
<td></td>
<td>P2</td>
<td>1.024</td>
<td>1.269</td>
<td>1.137</td>
<td>1.079</td>
<td>1.032</td>
<td>1.001</td>
<td>0.982</td>
<td>0.970</td>
</tr>
<tr>
<td></td>
<td>P3</td>
<td>1.024</td>
<td>0.557</td>
<td>0.382</td>
<td>0.296</td>
<td>0.235</td>
<td>0.201</td>
<td>0.178</td>
<td>0.158</td>
</tr>
<tr>
<td></td>
<td>P4</td>
<td>1.024</td>
<td>0.534</td>
<td>0.366</td>
<td>0.282</td>
<td>0.224</td>
<td>0.192</td>
<td>0.163</td>
<td>0.141</td>
</tr>
<tr>
<td>30</td>
<td>P1</td>
<td>1.023</td>
<td>0.624</td>
<td>0.436</td>
<td>0.328</td>
<td>0.287</td>
<td>0.265</td>
<td>0.252</td>
<td>0.233</td>
</tr>
<tr>
<td></td>
<td>P2</td>
<td>1.023</td>
<td>1.268</td>
<td>1.136</td>
<td>1.077</td>
<td>1.031</td>
<td>0.999</td>
<td>0.982</td>
<td>0.968</td>
</tr>
<tr>
<td></td>
<td>P3</td>
<td>1.023</td>
<td>0.557</td>
<td>0.381</td>
<td>0.302</td>
<td>0.244</td>
<td>0.206</td>
<td>0.177</td>
<td>0.159</td>
</tr>
<tr>
<td></td>
<td>P4</td>
<td>1.023</td>
<td>0.533</td>
<td>0.367</td>
<td>0.281</td>
<td>0.224</td>
<td>0.189</td>
<td>0.164</td>
<td>0.142</td>
</tr>
<tr>
<td>60</td>
<td>P1</td>
<td>1.024</td>
<td>0.625</td>
<td>0.437</td>
<td>0.328</td>
<td>0.286</td>
<td>0.261</td>
<td>0.254</td>
<td>0.233</td>
</tr>
<tr>
<td></td>
<td>P2</td>
<td>1.024</td>
<td>1.269</td>
<td>1.135</td>
<td>1.078</td>
<td>1.030</td>
<td>0.998</td>
<td>0.982</td>
<td>0.968</td>
</tr>
<tr>
<td></td>
<td>P3</td>
<td>1.024</td>
<td>0.557</td>
<td>0.382</td>
<td>0.288</td>
<td>0.238</td>
<td>0.196</td>
<td>0.170</td>
<td>0.151</td>
</tr>
<tr>
<td></td>
<td>P4</td>
<td>1.024</td>
<td>0.536</td>
<td>0.365</td>
<td>0.282</td>
<td>0.225</td>
<td>0.193</td>
<td>0.163</td>
<td>0.139</td>
</tr>
</tbody>
</table>
The P2 implementation gives longer execution time and lower speedup. When the number of processors increase from one to two, the execution time for \( m=10 \) using SO algorithm increases from 1.024 to 1.269 seconds and the speedup factor decreases almost 80%-90%. We think that the main reason of this performance degradation comes from the fact that during string matching each processor spends almost of its time to wait for the next chunks of text from master via network, though that network has been routed via an 100 Mbps Fast Ethernet switch.

Further, the P1 implementation using static allocation of subtexts produces better results than the P2 one. However, when the number of processors increase from one to two, the execution time slowly decreases. But speedup factor increases when the number of processors increase till 5 processors and slightly increases beyond that point. This fact due to the load imbalance because of the poor text partitioning strategy. Some processors that have less subtexts to perform string matching computation will finish their computing faster than the others and then stay idle.

Finally, the experimental results show that the P3 and P4 implementations seems to have the best performance compared with the others. More specifically, the P4 approach produces nearly optimal performance. From the results, we can see a clear reduction in the computation time of the algorithm when we use the P3 and P4 parallel implementations. For example, with text size of 3MB, \( m=10 \) and SO algorithm, we reduce the computation time from 1.024 seconds in the sequential version to 0.158 and 0.141 seconds in the parallel versions P3 and P4 respectively using 8 processors. In other words, we observe that for constant total text size there is an expected inverse relation between the parallel execution time and the number of processors. Further, the P3 and P4 methods achieve nearly perfect speedup curves for all pattern lengths, processors and the BF and SO string matching algorithms. Therefore, more time is spent in the string searching operation than communicating with the master processor.

### 3.3 Scalability issue

To study the scalability of the two proposed parallel implementations P3 and P4, we setup the experiments in the following way. We simple multiple by eight times the old text size. This new text collection is arround 24MB. Results from these experiments have been depicted in Tables 3 and 4 and in Figure 5.

The results show that the two parallel implementations still scales well though the text size has been increased eight times. The execution time for \( m=10 \) and SO algorithm similarly decrease to 1.184 and 1.063 seconds for the P3 and P4 implementations respectively when the number of processors have been added to 8. Moreover, speedup factors of the two methods also linearly increases when the processors are
Table 3: Experimental execution times (in seconds) for text size of 24MB using BF string matching algorithm and several pattern lengths

<table>
<thead>
<tr>
<th>m/p</th>
<th>Par. Implem.</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>P3</td>
<td>9.237</td>
<td>4.785</td>
<td>3.253</td>
<td>2.480</td>
<td>2.017</td>
<td>1.701</td>
<td>1.486</td>
<td>1.321</td>
</tr>
<tr>
<td></td>
<td>P4</td>
<td>9.237</td>
<td>4.642</td>
<td>3.095</td>
<td>2.334</td>
<td>1.875</td>
<td>1.558</td>
<td>1.331</td>
<td>1.165</td>
</tr>
<tr>
<td></td>
<td>P4</td>
<td>8.724</td>
<td>4.460</td>
<td>2.969</td>
<td>2.260</td>
<td>1.801</td>
<td>1.492</td>
<td>1.281</td>
<td>1.130</td>
</tr>
<tr>
<td>30</td>
<td>P3</td>
<td>8.513</td>
<td>4.516</td>
<td>3.073</td>
<td>2.356</td>
<td>1.914</td>
<td>1.621</td>
<td>1.409</td>
<td>1.254</td>
</tr>
<tr>
<td></td>
<td>P4</td>
<td>8.513</td>
<td>4.375</td>
<td>2.926</td>
<td>2.225</td>
<td>1.785</td>
<td>1.472</td>
<td>1.263</td>
<td>1.095</td>
</tr>
<tr>
<td></td>
<td>P4</td>
<td>9.284</td>
<td>4.769</td>
<td>3.201</td>
<td>2.421</td>
<td>1.962</td>
<td>1.631</td>
<td>1.370</td>
<td>1.198</td>
</tr>
</tbody>
</table>

Table 4: Experimental execution times (in seconds) for text size of 24MB using SO string matching algorithm and several pattern lengths

<table>
<thead>
<tr>
<th>m/p</th>
<th>Par. Implem.</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>P3</td>
<td>8.196</td>
<td>4.262</td>
<td>2.905</td>
<td>2.218</td>
<td>1.807</td>
<td>1.531</td>
<td>1.331</td>
<td>1.185</td>
</tr>
<tr>
<td></td>
<td>P4</td>
<td>8.196</td>
<td>4.140</td>
<td>2.779</td>
<td>2.098</td>
<td>1.684</td>
<td>1.408</td>
<td>1.211</td>
<td>1.064</td>
</tr>
<tr>
<td></td>
<td>P4</td>
<td>8.191</td>
<td>4.138</td>
<td>2.777</td>
<td>2.097</td>
<td>1.683</td>
<td>1.407</td>
<td>1.211</td>
<td>1.063</td>
</tr>
<tr>
<td>30</td>
<td>P3</td>
<td>8.194</td>
<td>4.257</td>
<td>2.901</td>
<td>2.215</td>
<td>1.806</td>
<td>1.529</td>
<td>1.330</td>
<td>1.183</td>
</tr>
<tr>
<td>60</td>
<td>P3</td>
<td>8.203</td>
<td>4.260</td>
<td>2.903</td>
<td>2.215</td>
<td>1.805</td>
<td>1.530</td>
<td>1.331</td>
<td>1.186</td>
</tr>
<tr>
<td></td>
<td>P4</td>
<td>8.203</td>
<td>4.138</td>
<td>2.777</td>
<td>2.097</td>
<td>1.683</td>
<td>1.407</td>
<td>1.211</td>
<td>1.063</td>
</tr>
</tbody>
</table>

increased.

4. PERFORMANCE PREDICTION MODEL

After presenting the experimental results, it is desirable and advantageous to develop an analytical model to describe the behaviour and the performance of the parallel implementations presented previously. The analytical model is not only used to verify the experimental results but also to predict the speedup tendency. Using, a large number of processors can be sometimes counter productive, that is, the speedup that they achieve is lower than that of a smaller number of processors. The main reason for that is the computation to communication ratio as applied to the specific cluster. Hence, this analytical model can save us a lot of time running different experiments to find the highest (or best) performance for a given string searching
Figure 5: Speedup of parallel string matching with respect to the number of processors for text size of 24MB using BF and SO algorithms

application. In next subsections we develop a performance model for the P2, P3 and P4 parallel implementations. Note that we do not model the performance of the P1 implementation since this approach is same with P4 one using the improved preprocessing allocation method.

4.1 I/O, matching and communication times

In order to formulate the equations that give us the expected speedup curves of the parallel implementations, we have to formulate the equations for the I/O time, matching time and communication time, which make up the execution time of the whole implementation. The I/O time is proportional to the size of the text. So, reading of the text requires \( n \) accesses to the disk. Let \( \gamma \) be the average time to perform one I/O step. Then, the I/O time, \( T_{i/o} \), is given by:

\[
T_{i/o} = n\gamma
\]  

(2)

If the workload is divided evenly over \( p \) processors, then each node’s processing time is given by:

\[
T_{i/o} = \left(\left\lfloor \frac{n - m + 1}{p} \right\rfloor + m - 1\right)\gamma
\]  

(3)

Further, the string matching of an \( m \) pattern string in a \( n \) text database requires \( n \) computation steps in practice for the BF and SO algorithms (Michailidis & Margaritis, 1999; Baeza-Yates & Gonnet, 1992). Let \( \delta \) be the average time to perform one computation step. Then, the searching time, \( T_{search} \), is given by:

\[
T_{search} = n\delta
\]  

(4)
In addition, each node’s processing time is given by:

$$T_{search} = \left( \left\lfloor \frac{n - m + 1}{p} \right\rfloor + m - 1 \right) \delta$$  \hspace{1cm} (5)

The total communication time of the parallel implementation is the summation of two components: latency time and transmission time. The latency time, $\alpha$, is a fixed startup overhead time needed to prepare sending a message from one processor to the other. The transmission time is proportional to the size of pattern string. Let $\beta$ be the incremental transmission time per byte. Note that it is usually the case that $\alpha \gg \beta$. Then the communication time, $T_{comm}$, to send P bytes of data (messages) is defined as:

$$T_{comm} = \alpha + P\beta$$  \hspace{1cm} (6)

### 4.2 Parallel performance for dynamic allocation of subtext

If the values of $\alpha$, $\beta$, $\gamma$ and $\delta$ are known, then the execution time of the dynamic string matching implementation can be easily estimated as will be shown below. To determine the parallel string matching time of the dynamic implementation for an $m$ pattern string, a $n$ text database and $p$ processors, we can broken up into five terms:

- $T_a$: It includes the communication time to broadcast the pattern string to all processors involved in the processing of the string matching. We may consider that the function MPI_Bcast is completed in $\log_2 p$ steps. In each step, one or two parallel send operations per processor are performed. The size of an $m$ pattern string is $m$ bytes. Therefore, the broadcast transfers $m$ bytes to the other $p - 1$ processors. The expression for this amount of the time is given by:

$$\log_2 p (\alpha + m\beta)$$  \hspace{1cm} (7)

- $T_b$: The total I/O time to read the entire text database from the local disk of the master processor. We know that the master processor reads the text from the local disk in main memory into several chunks of size $b + m - 1$ characters. Therefore, the master reads $n$ characters totally of text and the I/O time on master processor is given by:

$$(n - m + 1)\gamma$$  \hspace{1cm} (8)

- $T_c$: The communication time to send all chunks of the text to all worker processors. The master processor sends a chunk with size $b + m - 1$ characters to a worker processor. Therefore, the communication time to send all chunks to $p$ processors is given by:

$$\frac{n - m + 1}{b + m - 1}(\alpha + (b + m - 1)\beta)$$  \hspace{1cm} (9)
• \( T_d \): The average string matching time on a single processor. Since each processor has to search the pattern string in \( \left\lfloor \frac{n-m+1}{p(b+m-1)} \right\rfloor \) chunks of text where each chunk has size \( b + m - 1 \) characters. The string matching time on a single processor is given by:

\[
\left( \frac{n-m+1}{p} \right) + m - 1)\delta
\]

(10)

• \( T_e \): The communication time to receive the results of the string matching from one processor. We know that the master had to receive \( \frac{n-m+1}{b+m-1} \) results. Each processor sends back one value (i.e. the number of occurrences). Therefore, the communication time to receive these results from one processor is given by:

\[
\frac{n-m+1}{b+m-1} (\alpha + \beta)
\]

(11)

The total execution time of the dynamic string matching implementation, \( T_p \), using \( p \) processors is the summation of all the terms calculated above and is given by:

\[
T_p = T_a + T_b + T_c + T_d + T_e
\]

(12)

4.3 Parallel performance for dynamic allocation of text pointers

The total execution time of the dynamic implementation with pointers can be broken up into five terms:

• \( T_a \): The communication time for broadcasting of the pattern string to all processors involved. The amount of this time is same with the \( T_a \) of the previous implementation.

• \( T_b \): The communication time to send the text pointers instead of chunks of the text to all worker processors. Therefore, the communication time to send all text pointers to \( p \) processors is given by:

\[
\frac{n-m+1}{b+m-1} (\alpha + \beta)
\]

(13)

• \( T_c \): The average I/O time to read all text chunks from the local disk of a single processor. We know that each processor reads \( \left\lfloor \frac{n-m+1}{p(b+m-1)} \right\rfloor \) text chunks from the local disk in main memory where each chunk has size \( b + m - 1 \) characters, the I/O time on single processor is given by:

\[
\left( \frac{n-m+1}{p} \right) + m - 1)\gamma
\]

(14)

• \( T_d \): The average string matching time on a single processor. The amount of this time is same with the \( T_d \) of the previous implementation.
\[ T_e: \text{The communication time to receive the results of the string matching from one processor. The amount of this time is same with the } T_e \text{ of the previous implementation.} \]

The execution time of the dynamic string matching implementation, \( T_p \), using \( p \) processors is the summation of all the terms calculated above and is given by:

\[ T_p = T_a + T_b + T_c + T_d + T_e \]  
(15)

4.4 Parallel performance for hybrid allocation of subtexts

To formulate the parallel string matching execution time for an \( m \) pattern string, a \( n \) text database, and \( p \) workstations, we need to determine the value of each of the following terms:

- \( T_a \): It is mainly the time spent in text partitioning. Compared with the communication time and the string matching time, it is negligible.

- \( T_b \): The communication time for broadcasting of the pattern string to all processors involved. The amount of this time is same with the \( T_a \) of the dynamic implementation with allocation of subtexts.

- \( T_c \): The average I/O time for reading the subtext from the local disk of a single processor. We know that each processor has to reads the subtext from the local disk into a buffer in main memory with size \( \left\lceil \frac{n-m+1}{p} \right\rceil + m - 1 \) characters, the I/O time on single processor is given by:

\[ (\left\lceil \frac{n-m+1}{p} \right\rceil + m - 1) \gamma \]  
(16)

- \( T_d \): The average string matching time on a single processor. Since each processor has to search the pattern string in its subtext with size \( \left\lceil \frac{n-m+1}{p} \right\rceil + m - 1 \) characters, the searching time on a single processor is given by:

\[ (\left\lceil \frac{n-m+1}{p} \right\rceil + m - 1) \delta \]  
(17)

- \( T_e \): The master has to gather \( p \) results resulting from the string matching carried on the subtexts by \( p \) processors concurrently. Each processor sends back one value (in our case, the number of the occurrences). The function MPIReduce is completed in \( \log_2 p \) steps. Therefore, the communication time to gather these results from one processor is given by:

\[ \log_2 p (\alpha + \beta) \]  
(18)
Table 5: Values of $\gamma$ and $\delta$ for BF algorithm (left) and SO algorithm (right)

<table>
<thead>
<tr>
<th>$m$</th>
<th>$\gamma$</th>
<th>$\delta$</th>
<th>$m$</th>
<th>$\gamma$</th>
<th>$\delta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>8.87E-08</td>
<td>2.77E-07</td>
<td>5</td>
<td>7.59E-08</td>
<td>2.48E-07</td>
</tr>
<tr>
<td>10</td>
<td>7.91E-08</td>
<td>2.73E-07</td>
<td>10</td>
<td>7.62E-08</td>
<td>2.47E-07</td>
</tr>
<tr>
<td>30</td>
<td>7.95E-08</td>
<td>2.65E-07</td>
<td>30</td>
<td>7.61E-08</td>
<td>2.47E-07</td>
</tr>
<tr>
<td>60</td>
<td>7.90E-08</td>
<td>2.95E-07</td>
<td>60</td>
<td>7.60E-08</td>
<td>2.48E-07</td>
</tr>
</tbody>
</table>

The total execution time of the static string matching implementation, $T_p$, using $p$ processors is the summation of all the terms calculated above and is given by:

$$T_p = T_b + T_c + T_d + T_e$$  \quad (19)

4.5 Determination of $\alpha$, $\beta$, $\gamma$ and $\delta$

We show how the values of $\alpha$, $\beta$, $\gamma$ and $\delta$, which are used in our performance prediction model, are found. The I/O (or searching) factor, $\gamma$ (or $\delta$), is found by measuring the time taken which performs $n$ (or $n$) steps. Since the I/O (or searching) time, $T_{i/o}$ (or $T_{search}$) is equal to $n$ (or $n$ for BF and SO algorithms), $\gamma$ (or $\delta$) can be obtained easily in the following way:

$$\gamma = \frac{T_{i/o}}{n} \text{ or } \delta = \frac{T_{search}}{n}$$

In Table 5 we list the values of $\gamma$ and $\delta$ (in seconds) for different string matching algorithms and pattern lengths.

In order to find the values of $\alpha$ and $\beta$, we run some simple ping-pong tests which send/receive an number of messages between two processors. Figure 6 shows the corresponding communication times of sending/receiving messages using the ping-pong test which consists of two processes where each process resides on a single processor. Both processes do nothing but simply send and receive messages. All timings are average times over 100 separate rounds. Using the linear regression method to fit a straight line to the curve of the communication, we find the values of $\alpha$ and $\beta$ which are the message latency time and the incremental transmission time per byte. They are found to be 0.00062371 secs and 0.000000194885 secs respectively in our computing environment.

4.6 Expected results

Tables 6 and 7 show for some values of $m$, $n$ and $p$ using BF algorithm the execution times obtained using the equations 12, 15 and 19 for the P2, P3 and P4 implementations respectively. Similarly, Tables 8 and 9 show the execution times for the P2,
Figure 6: The communication time of the ping-pong test

Figure 7: The analytical and experimental speedup as a function of the number of processors for the P2 implementation using BF and SO algorithms

P3 and P4 implementations using SO algorithm. Figures 7, 8 and 9 present for some values n and p using BF and SO algorithms the speedups obtained in the experiments and those using the equations 12, 15, 19 and 1 for the P2, P3 and P4 implementations respectively. It is important to note that the speedups, which are plotted in Figures, is result of the average for four pattern lengths. Note that while there are small differences between the experimental and expected values, the overall actual behaviour is similar. An interesting point to note from the expected results for the P2 implementation is that theoretical times actually exceed experimental values for large number of workstations. This can be explained, since the theoretical prediction model assumes that \( T_p = T_{comp} + T_{comm} \), but in actual practice, computation and communication will be overlapped by an amount directly proportional to the number of processors.
Table 6: Analytical execution times (in seconds) for text size of 3MB using BF string matching algorithm and several pattern lengths

<table>
<thead>
<tr>
<th>m/p</th>
<th>Par. Implem.</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>P2</td>
<td>1.156</td>
<td>1.375</td>
<td>1.229</td>
<td>1.157</td>
<td>1.113</td>
<td>1.084</td>
<td>1.063</td>
<td>1.048</td>
</tr>
<tr>
<td></td>
<td>P3</td>
<td>1.156</td>
<td>0.598</td>
<td>0.406</td>
<td>0.310</td>
<td>0.252</td>
<td>0.213</td>
<td>0.186</td>
<td>0.165</td>
</tr>
<tr>
<td></td>
<td>P4</td>
<td>1.156</td>
<td>0.580</td>
<td>0.389</td>
<td>0.294</td>
<td>0.237</td>
<td>0.199</td>
<td>0.170</td>
<td>0.151</td>
</tr>
<tr>
<td>10</td>
<td>P2</td>
<td>1.113</td>
<td>1.338</td>
<td>1.195</td>
<td>1.123</td>
<td>1.080</td>
<td>1.051</td>
<td>1.031</td>
<td>1.016</td>
</tr>
<tr>
<td></td>
<td>P3</td>
<td>1.113</td>
<td>0.577</td>
<td>0.391</td>
<td>0.299</td>
<td>0.243</td>
<td>0.206</td>
<td>0.180</td>
<td>0.160</td>
</tr>
<tr>
<td></td>
<td>P4</td>
<td>1.113</td>
<td>0.559</td>
<td>0.375</td>
<td>0.283</td>
<td>0.228</td>
<td>0.192</td>
<td>0.172</td>
<td>0.152</td>
</tr>
<tr>
<td>30</td>
<td>P2</td>
<td>1.089</td>
<td>1.327</td>
<td>1.187</td>
<td>1.118</td>
<td>1.076</td>
<td>1.048</td>
<td>1.029</td>
<td>1.014</td>
</tr>
<tr>
<td></td>
<td>P3</td>
<td>1.089</td>
<td>0.565</td>
<td>0.383</td>
<td>0.293</td>
<td>0.238</td>
<td>0.202</td>
<td>0.176</td>
<td>0.157</td>
</tr>
<tr>
<td></td>
<td>P4</td>
<td>1.088</td>
<td>0.547</td>
<td>0.367</td>
<td>0.277</td>
<td>0.223</td>
<td>0.187</td>
<td>0.162</td>
<td>0.145</td>
</tr>
<tr>
<td>60</td>
<td>P2</td>
<td>1.183</td>
<td>1.373</td>
<td>1.218</td>
<td>1.140</td>
<td>1.094</td>
<td>1.063</td>
<td>1.041</td>
<td>1.024</td>
</tr>
<tr>
<td></td>
<td>P3</td>
<td>1.183</td>
<td>0.612</td>
<td>0.415</td>
<td>0.316</td>
<td>0.257</td>
<td>0.218</td>
<td>0.190</td>
<td>0.169</td>
</tr>
<tr>
<td></td>
<td>P4</td>
<td>1.183</td>
<td>0.594</td>
<td>0.398</td>
<td>0.300</td>
<td>0.242</td>
<td>0.203</td>
<td>0.175</td>
<td>0.155</td>
</tr>
</tbody>
</table>

Table 7: Analytical execution times (in seconds) for text size of 24MB using BF string matching algorithm and several pattern lengths

<table>
<thead>
<tr>
<th>m/p</th>
<th>Par. Implem.</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P4</td>
<td>9.225</td>
<td>4.615</td>
<td>3.079</td>
<td>2.311</td>
<td>1.850</td>
<td>1.544</td>
<td>1.322</td>
<td>1.157</td>
</tr>
<tr>
<td>10</td>
<td>P3</td>
<td>8.881</td>
<td>4.598</td>
<td>3.118</td>
<td>2.378</td>
<td>1.934</td>
<td>1.638</td>
<td>1.427</td>
<td>1.268</td>
</tr>
<tr>
<td></td>
<td>P4</td>
<td>8.881</td>
<td>4.443</td>
<td>2.964</td>
<td>2.225</td>
<td>1.782</td>
<td>1.486</td>
<td>1.275</td>
<td>1.115</td>
</tr>
<tr>
<td></td>
<td>P4</td>
<td>8.686</td>
<td>4.345</td>
<td>2.899</td>
<td>2.176</td>
<td>1.743</td>
<td>1.454</td>
<td>1.249</td>
<td>1.089</td>
</tr>
<tr>
<td>60</td>
<td>P3</td>
<td>9.439</td>
<td>4.877</td>
<td>3.304</td>
<td>2.518</td>
<td>2.046</td>
<td>1.731</td>
<td>1.507</td>
<td>1.338</td>
</tr>
</tbody>
</table>
Table 8: Analytical execution times (in seconds) for text size of 3MB using SO string matching algorithm and several pattern lengths

<table>
<thead>
<tr>
<th>m/p</th>
<th>Par. Implem.</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>P2</td>
<td>1.025</td>
<td>1.289</td>
<td>1.158</td>
<td>1.093</td>
<td>1.054</td>
<td>1.028</td>
<td>1.010</td>
<td>0.996</td>
</tr>
<tr>
<td></td>
<td>P3</td>
<td>1.025</td>
<td>0.533</td>
<td>0.362</td>
<td>0.277</td>
<td>0.225</td>
<td>0.191</td>
<td>0.167</td>
<td>0.149</td>
</tr>
<tr>
<td></td>
<td>P4</td>
<td>1.025</td>
<td>0.514</td>
<td>0.344</td>
<td>0.259</td>
<td>0.208</td>
<td>0.174</td>
<td>0.150</td>
<td>0.132</td>
</tr>
<tr>
<td>10</td>
<td>P2</td>
<td>1.024</td>
<td>1.289</td>
<td>1.159</td>
<td>1.094</td>
<td>1.055</td>
<td>1.029</td>
<td>1.010</td>
<td>0.997</td>
</tr>
<tr>
<td></td>
<td>P3</td>
<td>1.024</td>
<td>0.532</td>
<td>0.362</td>
<td>0.276</td>
<td>0.225</td>
<td>0.191</td>
<td>0.167</td>
<td>0.149</td>
</tr>
<tr>
<td></td>
<td>P4</td>
<td>1.024</td>
<td>0.513</td>
<td>0.343</td>
<td>0.258</td>
<td>0.208</td>
<td>0.174</td>
<td>0.150</td>
<td>0.132</td>
</tr>
<tr>
<td>30</td>
<td>P2</td>
<td>1.024</td>
<td>1.288</td>
<td>1.158</td>
<td>1.093</td>
<td>1.054</td>
<td>1.028</td>
<td>1.010</td>
<td>0.996</td>
</tr>
<tr>
<td></td>
<td>P3</td>
<td>1.024</td>
<td>0.532</td>
<td>0.361</td>
<td>0.276</td>
<td>0.225</td>
<td>0.191</td>
<td>0.167</td>
<td>0.149</td>
</tr>
<tr>
<td></td>
<td>P4</td>
<td>1.024</td>
<td>0.513</td>
<td>0.343</td>
<td>0.258</td>
<td>0.208</td>
<td>0.174</td>
<td>0.150</td>
<td>0.132</td>
</tr>
<tr>
<td>60</td>
<td>P2</td>
<td>1.024</td>
<td>1.289</td>
<td>1.158</td>
<td>1.093</td>
<td>1.054</td>
<td>1.028</td>
<td>1.010</td>
<td>0.996</td>
</tr>
<tr>
<td></td>
<td>P3</td>
<td>1.024</td>
<td>0.532</td>
<td>0.362</td>
<td>0.276</td>
<td>0.225</td>
<td>0.191</td>
<td>0.167</td>
<td>0.149</td>
</tr>
<tr>
<td></td>
<td>P4</td>
<td>1.024</td>
<td>0.514</td>
<td>0.344</td>
<td>0.259</td>
<td>0.208</td>
<td>0.174</td>
<td>0.150</td>
<td>0.132</td>
</tr>
</tbody>
</table>

Table 9: Analytical execution times (in seconds) for text size of 24MB using SO string matching algorithm and several pattern lengths

<table>
<thead>
<tr>
<th>m/p</th>
<th>Par. Implem.</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>P3</td>
<td>8.174</td>
<td>4.245</td>
<td>2.883</td>
<td>2.202</td>
<td>1.793</td>
<td>1.521</td>
<td>1.326</td>
<td>1.180</td>
</tr>
<tr>
<td></td>
<td>P4</td>
<td>8.174</td>
<td>4.088</td>
<td>2.727</td>
<td>2.046</td>
<td>1.638</td>
<td>1.366</td>
<td>1.171</td>
<td>1.026</td>
</tr>
<tr>
<td>10</td>
<td>P3</td>
<td>8.170</td>
<td>4.243</td>
<td>2.881</td>
<td>2.200</td>
<td>1.792</td>
<td>1.520</td>
<td>1.325</td>
<td>1.180</td>
</tr>
<tr>
<td></td>
<td>P4</td>
<td>8.170</td>
<td>4.086</td>
<td>2.725</td>
<td>2.045</td>
<td>1.637</td>
<td>1.365</td>
<td>1.171</td>
<td>1.025</td>
</tr>
<tr>
<td>30</td>
<td>P3</td>
<td>8.165</td>
<td>4.240</td>
<td>2.879</td>
<td>2.199</td>
<td>1.791</td>
<td>1.519</td>
<td>1.325</td>
<td>1.179</td>
</tr>
<tr>
<td></td>
<td>P4</td>
<td>8.165</td>
<td>4.084</td>
<td>2.724</td>
<td>2.044</td>
<td>1.636</td>
<td>1.364</td>
<td>1.170</td>
<td>1.024</td>
</tr>
<tr>
<td>60</td>
<td>P3</td>
<td>8.170</td>
<td>4.243</td>
<td>2.881</td>
<td>2.201</td>
<td>1.792</td>
<td>1.520</td>
<td>1.325</td>
<td>1.180</td>
</tr>
<tr>
<td></td>
<td>P4</td>
<td>8.170</td>
<td>4.086</td>
<td>2.725</td>
<td>2.045</td>
<td>1.637</td>
<td>1.365</td>
<td>1.171</td>
<td>1.025</td>
</tr>
</tbody>
</table>
Figure 8: The analytical and experimental speedup as a function of the number of processors for the P3 implementation using BF and SO algorithms
Figure 9: The analytical and experimental speedup as a function of the number of processors for the P4 implementation using BF and SO algorithms.
5. CONCLUSIONS

In this paper we have presented four parallel string matching implementations using static and dynamic text allocation and presented our experimental results on a low cost cluster of workstations. We have learned from this study that applying the hybrid parallel implementation P4 is an efficient way to solve the string matching problem. The good performance of this parallel implementation is due to two main reasons. First, it has low communication cost. Second, the load imbalance is low using the preprocessing allocation method. Further, we have observed that the static allocation of subtexts P1 and dynamic allocation of subtexts P2 produce worse experimental results than the P3 and P4 implementations.

We also introduced a performance prediction model, which can help in the systematic design, evaluation and performance tuning of parallel string matching implementations presented previously. The model requires only a small set of input parameters ($\alpha$, $\beta$, $\gamma$ and $\delta$) that can be obtained from cluster specifications or from trial runs on a mininal system. The execution of parallel string matching implementations were studied on a 9-cluster of workstations and were modelled using the performance model. It has been shown that the performance model approximates the actual behaviour of the experimental measurements. The maximal difference between theoretical and measurement values are less than 6-7%. Therefore, the model for the three parallel implementations is accuracy since the predictions need not be quantitatively exact, prediction errors of 10-20% are acceptable.

Finally, based on the findings of (Michailidis & Margaritis, 2001) it can be argued that the performance of any other sequential string matching algorithm incorporated in the proposed parallel schemes can be easily predicted to a reasonable accuracy.

Future research plans include experiments for static and dynamic load balancing algorithms on a heterogeneous network of workstations.

REFERENCES


design for the implementation of text searching using a multicomputer. Information
Processing and Management, v. 27, pp. 265-283.

Press.


Programming with the message passing interface. Massachusetts: The MIT
Press.


rithms: Survey and experimental results. International Journal of Computer

Technical Report, Dept. of Applied Informatics, University of Macedonia, (in
Greek).

Morgan Kaufmann.

International Journal of Pattern Recognition and Artificial Intelligence, v. 8,
pp. 815-843.


applications using networked workstations and parallel computers. Prentice Hall.