

## Parallel Overlap and Similarity Detection in Semi-Structured Document Collections

Krisztián Monostori, Arkady Zaslavsky, Heinz Schmidt

School of Computer Science and Software Engineering  
Monash University, Melbourne, Australia  
{krisztian.monostori, arkady.zaslavsky, heinz.schmidt}  
@infotech.monash.edu.au

**Abstract.** Proliferation of digital libraries plus high availability of electronic documents from the Internet have created new challenges for computer science researchers and professionals. This paper discusses the problems of using parallel and cluster computing systems for detecting plagiarism in large collections of semi-structured electronic texts, including software written in formal languages at one end of the spectrum and natural language texts at the other end. The main component of the system is using string matching algorithms and suffix trees. Implementation and performance issues are also discussed.

### 1. Introduction

Digital libraries provide vast amounts of digitised information on-line. Preventing these documents from unauthorised copying and redistribution is a hard and challenging task, which often results in not putting valuable documents on-line [8]. Copy-prevention mechanisms include distributing information on a separate disk, using special hardware or active documents [9]. One of the more recent areas of copy-detection applications is plagiarism detection. With the enormous growth of the information available on the Internet users have a handy tool for "creating" assignments. With the numerous search engines users can easily find relevant articles and papers for their research. However, the Internet is a two-edge sword. Documents are available in electronic form too easily for cut-and-paste or drag-and-drop operations. Subsequently, without tools, it may be hard to determine the amount of original work.

There are several systems built for plagiarism detection including SCAM [9], Glatt [12], plagiarism.org [17] etc. SCAM and plagiarism.org are similar in their approach. They build an index on a collection of registered documents by using hashing algorithms and compare these hashed values. Our approach uses computations similarly to [9] and Plagiarism.org [17], but considers exact string matching algorithms rather than hashing, which has a finite probability of failure and is reported in [10]. How to identify candidate documents is beyond the scope of this paper. This problem is also addressed in the dScam prototype developed at Stanford University and described in [11].

Another current issue is code plagiarism. Program code is also an intellectual property and it is much easier to change the code in such a way that it still produces the same result while looks different. For example, consistent replacement of variable or function names, modification in comments, or the reordering of program entities may not change the program execution and outcome. In the context of program code we have to identify similarity rather than perfect match.

In case of textual documents similarity detection can also be used. There are large collections of papers, especially from the 70's and 80's, which are not digitized. Scanning these documents and recognizing them with the OCR software is error-prone, so we need tools to identify those documents, which differ slightly, perhaps by several characters.

Powerful processors, high-speed networks and standard tools for distributed computing make clusters of PCs or workstations an appealing platform for parallel computing. Several systems have been built including the Beowulf project at NASA [3], the Network of Workstations (NOW) project [16] at University of California, Berkeley, and the Solaris-MC project at Sun Labs [18]. PC clusters can be built at a very low cost from off-the-shelf components and they can provide parallel processing power, network RAM, software RAID, and multipath communications [4]. The most common operating systems include Linux, Solaris, NT, and AIX. In this paper we will analyze how our algorithm utilizes this platform.

The paper is structured as follows. In Section 2 we give an overview of well-known efficient string matching algorithms and introduce suffix trees and their different applications including the problem of identifying the overlapping texts. We also discuss the different algorithms for building suffix trees in a time-efficient manner. Subsection 2.2 introduces a new approach for representing textual documents by suffix trees. Section 3 outlines the test document set for the algorithm and presents some performance analysis results in the sequential case. Section 4 discusses some aspects of existing cluster systems and describes how our algorithm can utilize the parallel processing power of PC clusters. In Section 5 we describe work in progress and future endeavours.

## 2. Fast and Efficient String Matching Algorithms

The basic problem of string matching algorithms is to find occurrences of  $P$  in a string  $T$  of length  $n$  given a pattern  $P$ , which is a string of length  $m$ . There are three well-known and widely published efficient algorithms for this problem: the Karp-Rabin algorithm, the Knuth-Morris-Pratt algorithm, and the Boyer-Moore algorithm [1]. Modifications of the basic problem include matching regular expressions and matching with "don't cares".

Below, we will describe more sophisticated string matching problems, which are closely related to the overlapping texts problem. This list is not comprehensive in any manner though.

*Longest common subsequence*: finding the longest sub-string contained in two given strings by successively deleting characters. This problem is strongly related to the *edit*

*distance* problem, that is, finding the minimum number of edit operations, which transforms the first string into the second. There are well-known dynamic programming solutions for this problem [14] and by using the Hirschberg's algorithm this space can be reduced. The Unix *diff* command produces the edit distance between two files.

*Squares in a string*: Let  $w$  be a substring of  $x$ .  $w$  is a square of  $x$  if  $x = uwwv$  where  $u$  is a prefix of  $x$  and  $v$  is a suffix of  $x$ . Related problems include *square free words*, *finding the shortest  $u$* , etc. [2,15].

*Suffix-prefix matching*: Given two strings  $S$  and  $T$  a suffix-prefix match is a suffix of  $S$  that matches a prefix of  $T$  [14].

The problems mentioned above as well as many other string matching problems are closely related to DNA matching and string matching algorithms have been extensively used in microbiology applications.

## 2.1 Suffix trees

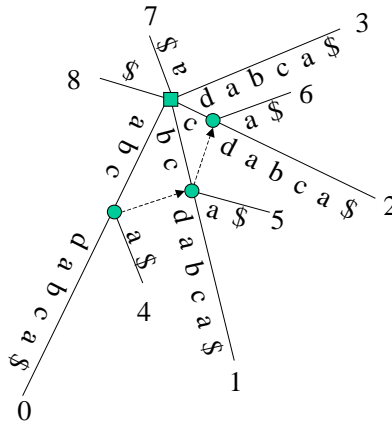
A *suffix tree* is a data structure, which represents all suffixes in a string. Suffix trees are also referred to as position trees, subword trees and complete inverted files [1,2]. [14] describes several applications of suffix trees including exact string matching, exact set matching, longest common substring, all-pairs suffix-prefix matching etc. [2] gives a brief overview of straightforward applications of suffix trees. According to [14] there is no other data structure, which can be used more efficiently than suffix trees.

We give the definition of a suffix tree as it is given in [14]: a suffix tree  $T$  for a string  $S$  length of  $m$  is a rooted directed tree with exactly  $m$  leaves numbered 0 to  $m-1$ . Each internal node, excluding the root, has at least two leaves labeled with substrings of  $S$  starting with different characters. Concatenation of edge-labels from the root to the leaf  $i$  identify the suffix of  $S$  starting at position  $i$ . This definition does not guarantee that such a tree exists for  $S$  but if we add a unique termination symbol (e.g.  $\$$ ) the tree exists and it is unique for a given  $S$ . *Figure 1* depicts the suffix tree of the string 'abcdabca\$'. The root node is depicted by a square.

There are two problems associated with the construction of a suffix tree: the space requirement of a suffix tree is  $O(m^2)$ , and the time required to build a suffix tree using a naive algorithm, that is inserting all suffixes of  $S$  starting from left to right, takes  $O(m^2)$  time. The first problem can easily be overcome by not labelling the edges with the exact substring but by only two pointers: one to the beginning and one to the end of the substring. It also requires storing  $S$  itself to resolve the pointers when needed. For example, in *figure 1* the leaf labelled 'a\$' can be represented as {7, 8}.

The first linear time algorithm for constructing the tree was given by Weiner [14]. A cleaner version of Weiner's algorithm is described in [7]. A few years later, McCreight [14] proposed the more space-efficient algorithm. The most elegant linear-time construction of a suffix tree was given by Ukkonen [19], which can be viewed as a variant of McCreight's algorithm [14]. Ukkonen's algorithm uses suffix links during the construction of the tree, which are represented by dashed arrows in *figure 1*. We keep the suffix links because they are needed by the matching statistics algorithm used

for detecting overlap. For a detailed description of Ukkonen's algorithm the reader is referred to [14] or [19].



**Fig. 1.** Example of a suffix tree with suffix links

[6] describes an algorithm for finding so called *matching statistics* -  $ms(i)$ , which is the longest substring of  $T$  starting at position  $i$  that matches a substring somewhere in  $P$ . Having built a suffix tree for  $P$ , matching statistics of  $T$  can be found in  $O(n)$  time where  $n$  is the length of  $T$ . The main idea of the algorithm is that starting from the first position in  $T$  we calculate the matching statistics  $ms(0)$  by traversing the tree of  $P$ . Then, in order to calculate  $ms(i)$  while having calculated  $ms(i-1)$  we have to back up to the node above our current position, follow the suffix link of that node and then traverse down from that node. This algorithm will also be tailored to the overlapping texts problem in the following section.

## 2.2 Suffix Tree for Textual Documents

With the algorithms described in the preceding sections we can now easily identify matching parts of documents. If we extend Ukkonen's algorithm by keeping an index value on each node, which corresponds to one of the indices under that node then overlapping can be calculated with the following algorithm [14].

Let  $T$  denote the document that we want to check if it contains parts of other documents. Let  $P$  denote one of the candidate-documents that we want to check. We build a suffix tree for  $P$  and calculate the matching statistics of  $T$ . In the end, we will know the largest part of text for every position of  $T$  starting at that position and also appearing in  $P$ . During the matching statistics algorithm run, when we finish a match starting from the  $i$ -th position, we can also store the index of the node at or right above the point where we finished. Let this node index be  $j$  and the size of the chunk be  $n$ .

Now we know that starting at position  $i$  in  $T$  there is a matching chunk of length  $n$ , which also appears in  $P$  starting at position  $j$ .

Now we will refine this algorithm. First we have to convert the files that we want to analyze into a unified format. We need this step because we consider two chunks overlap even if within the text different periods are used, or more new lines are inserted. So we leave all alphanumerical characters as they are, convert each alphabetical character to lowercase, and replace all other characters by another single character that is not alphanumerical. We convert any contiguous repetitions of these characters into a single character. We also shift numerical characters because we need a contiguous range of characters, so that the algorithm can work more efficiently. As described in Subsection 2.1 we also need a termination symbol, which can be chosen at either end of the range. Finally, we end up with a 38-character alphabet, which is important, because the size of the suffix tree and the time of Ukkonen's algorithm depend on the size of the alphabet.

We have not considered so far that textual documents are made up of words and we are not interested in overlaps starting in the middle of a word. As an example, consider two texts: 'His thesis made him popular among professors' and 'That car was made in Italy'. We would consider 'made' as overlap but not 's made'. This example reveals another problem, namely what size of overlap should be reported as a matching chunk in order not to report every single word match. We found that 60 characters is an acceptable threshold because institute and organization names could easily exceed lower values.

So how to utilize words? We may consider words as symbols then we could have much less symbols in the document but our alphabet, that is all English words, would be very large. In case of an alphabet of this size edges cannot be represented as arrays. A linked list would also be prohibitive because from the root there would be an edge for each different word in the document.

The main idea of our algorithm is to eliminate those parts of the tree that represent suffixes starting in the middle of a word. What we do is to follow Ukkonen's algorithm not character-by-character but rather word-by-word. So in the case of the above-mentioned example 'His thesis made him popular among professors' we would only insert suffixes starting from positions 0,4,11,16,20,28,34. For practical reasons we insert a leading blank to each document and we insert suffixes starting at blanks, which means one additional node under the root and we have a single edge running out of the root labelled by the blank character.

### 3. Performance Analysis of the Sequential Algorithm

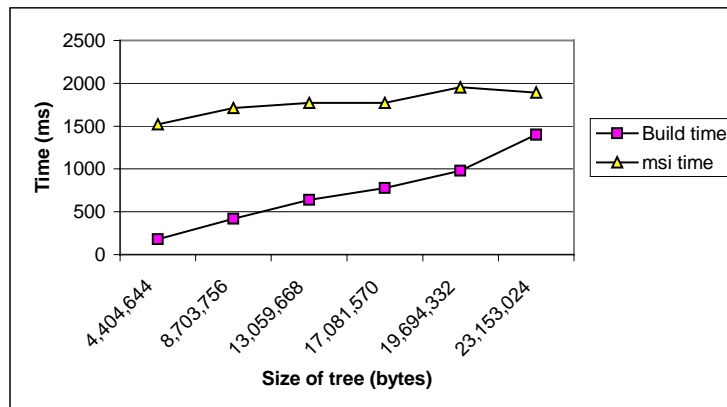
The modified Ukkonen algorithm and the modified suffix tree work well if we want to compare two documents. How can it be applied most efficiently if we want to compare many documents to a given document (e.g. detecting plagiarism)? In this section we will describe two closely related approaches for this problem and we will analyse how they perform in hypothetical scenarios.

The straightforward application of the algorithm given in Section 2.2 is to read the document to be analysed (a suspicious document) into memory and build a suffix tree for each document that we want to compare to this file (candidate-documents) and then calculate the matching statistics of the suspicious document for each candidate-document. From now on we will refer to this algorithm as the STCD algorithm (Suffix Tree for Candidate Documents).

We can save on building the suffix tree for each document by reversing the use of matching statistics algorithm and build the suffix tree only once for the suspicious document and analyze candidate documents using the tree (STOD algorithm - Suffix Tree for the Original Document).

Based on the overlap context we can set a threshold, which defines the overlap percentage considered as plagiarism. We also can define a chunk size (e.g. 1000 characters), which reports the document plagiarized if it has a chunk of overlap with another document above this threshold, though the overall overlap is under the given percentage.

Figure 2 depicts the relation between the size of the suffix tree and the time required to build the suffix tree (depicted by squares). Triangles depict the time required to calculate the matching statistics if we have candidate documents with total size of 9.84M. The size of the suffix tree is app. 23 times of the size of the original document. Measurements were taken on our test machine (Intel Pentium II 433MHz, 128M RAM, Windows NT Workstation).



**Fig. 2.** Running time for building suffix tree and calculating matching statistics.

The running time for building the suffix tree is proportional to the size of the document. Calculating matching statistics is theoretically independent from the size of the tree. In figure 2 there is a slight increase in time as the size of the tree is increasing. It contradicts with the linear time bound of the matching statistics algorithm, which states that the running time of the algorithm is independent from the size of the tree and only depends on the strings compared to the tree, which are the candidate-documents of 9.84M in our case. We found two issues, which influence the running time of the algorithm in practice. Firstly, in case of a small suffix tree the cache-hit ratio is higher. Secondly, it is true that the so called skip/count steps [14] are bounded

by 3m but the bigger the tree is the more the skip/count steps occur, which adds up to the running time. It is still true that the algorithm has linear time worst-case bound but in reality the case is worse if the tree is bigger.

#### 4. Parallel String Matching Algorithms

As already mentioned in Section 1 there are several cluster systems built for different applications of commodity workstations. The Berkeley NOW project [16] uses Solaris Workstations and applies an OS layer called GLUnix (Global Layer Unix) above Unix to provide a single system image. This layer provides transparent remote execution, support for interactive parallel and sequential jobs, load balancing, and a cluster-wide namespace. It also provides network RAM, which enables heavily loaded workstations to use the memory of idle workstations. The NOW project was also investigated in network interface hardware and fast communication protocols.

The Beowulf project [3] emphasizes the use of mass-market commodity components. They use PCs running a modified Linux, which is able to handle multiple parallel Ethernet networks. Research has shown that up to three networks can be bundled together to obtain significant throughput. The project includes several programming environments, i.e. PVM, MPI, BSP etc.

Solaris-MC [18] is a distributed operating system for a cluster of Solaris-based PCs and workstations. It is built on the top of the Solaris kernel to provide a global single system image. The most interesting feature of Solaris-MC is that it uses an object-oriented framework (CORBA) for interprocess communication. It also features a distributed file system called ProXy File System (PXFS).

In our project we use the Monash Parallel Parametric Modelling Engine (PPME) at the School of Computer Science and Software Engineering, Monash University. The PPME is a cluster of high-end PCs. The current system specifications include 22 x 330 MHz Pentium II processors; 8 x 500 MHz Pentium III processors; 2.4 Gbyte RAM; 61 Gbyte disk space.

All fifteen machines are dual-processor Pentiums. 10 machines can be booted either Linux or Windows NT while the remaining five machines are running Linux. One of the Linux-only machines serves as a cluster server. The second 32-processor cluster has been built and is connected to the first cluster by an ATM network.

These clusters are shared resources. For the first few benchmarks of our code we have only used a limited number of cluster nodes. Parametric experiments can be easily executed using the Active Tools Cluster program [5].

The capabilities of Cluster tool are briefly described below. Cluster easily enables to execute jobs with different parameters. If you have for example 5 parameters and you want to test your algorithm with 5 different values for each parameter it means that you have to execute  $5^5 = 3125$  jobs. These jobs are dispatched and scheduled by Cluster according to a plan file, which can be generated by a graphical tool. Basically Cluster executes an executable file 3125 times with different parameters using as many processors as available and balances the load among them.

Firstly, we will analyze how we can use Cluster for parallel execution of our algorithms, which requires no effort in implementing job distribution and load-balancing because all these tasks are performed by Cluster. We only need to provide a plan file, which describes the parameters we want to test. Parallelizing both sequential approaches (STCD, STOD) is straightforward. Different approaches discussed in the following subsections share the post-processing phase, in which the overall overlap is calculated.

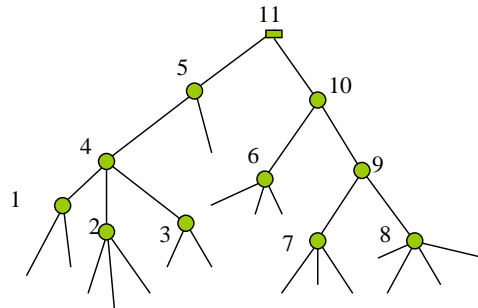
#### 4.1 Parallel Algorithm Using the STCD Approach with Cluster

In the case of the first approach when we have the suspicious document as a string and create a suffix tree for each candidate document, we can execute the same program as many times as the number of candidate documents by providing the name of the suspicious document and one candidate document at a time. This approach has one drawback: the suspicious document has to be read into memory on every execution because Cluster repeatedly executes the same executable and we cannot keep the file persistently in memory. However Windows NT provides file caching, which means that we have really good chances that the suspicious document will be read only once from disk and all successive executions will read from memory rather than from disk, which is still some time penalty but remember that we have a parallel execution with very little effort. We can reduce the time wasted on reading the suspicious file into memory by batching a certain number of files together. For example, each execution analyzes 10 documents against the suspicious one. In this case if we have only a few files it can easily lead to load-imbalance. This problem is less significant as the number of files increases but we still have to read the suspicious document once for every 10 documents.

#### 4.2 Parallel Algorithm Using the STOD Approach with Cluster

Using the second algorithm, which builds the suffix tree once and compares the candidate documents against the tree, will pay more for not having the suffix tree persistently in memory. Basically we have two choices: we either read the suspicious document into memory several times and build the suffix tree on every execution or we save the suffix tree on disk and read the suffix tree from disk on every execution. If we use this second algorithm we have to linearize the suffix tree in order to save it on disk. Since we also need to store the suffix links it is not a simple tree linearization. We will outline the algorithm for suffix tree linearization below.

In the first step we label the nodes with a number between 1 and the 'total number of nodes' by a depth-first search of the tree. An example of this numbering scheme is depicted in *figure 3*. Suffix links and edge labels are omitted from the figure for the sake of readability.



**Fig. 3.** Suffix Tree Labelling

In the second step we write all nodes to disk in the order of the numbers. First we write out the number of edges running out of the given node then we write out the edges one by one. Each edge excluding leaves includes the address of the node it points to. Instead of the address we write out the number of that node. Note that applying this numbering scheme all possible nodes that can be referred by an edge are written to disk prior to the edge itself is written. It means that on reading the tree into memory we can always replace the node number by the real address of a node because it is already in memory. Having written out all edges of a node we write out node-related information including the number of the node pointed by the suffix link, if any. Note that we do not need to write out the node number itself because on reading back nodes into memory we can number the nodes.

Now let us detail the reading algorithm. We need to maintain a look-up table for nodes, so that we can substitute node numbers with real memory locations. This table is built during the process of reading the nodes into memory. As described above all node numbers referred by edges can be resolved at the time of reading the edge. At the end of this step we have the tree in memory without the explicit suffix links. Also note that the root node is read in the end, which means that the root is assigned to the last node.

In the next step we can resolve the suffix link numbers by a depth-first traversal of the tree. After this step the look-up table can be deleted to free memory. In *figure 4* we give a pseudo code for the write and read algorithms respectively.

Unfortunately reading a suffix tree from disk takes more time than reading the file from disk and using Ukkonen's algorithm for creating the tree. We used a 1.67M file, which took 2153ms to build and 1191ms to read into memory when we read the file itself. Reading the tree from disk took 4236ms and an additional 1957ms to build. When the file is repeatedly read into memory we can again rely on file-caching of Windows NT, which makes the two approaches almost equal in speed but using the second approach we have to build and store the tree on disk first. As the first approach is less complex and has approximately same performance we analyzed this approach.

Returning to the problem of parallelizing the second algorithm we have to face the same problems as in the case of the first algorithm with the modification that the I/O or rebuilding the suffix tree can cost more.

**Write algorithm**

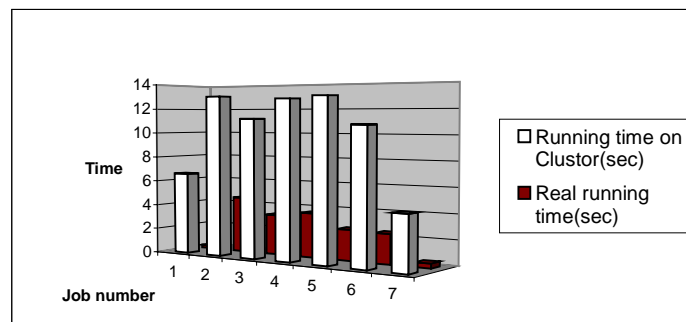
1. Number nodes by a depth-first traversal starting from 1
2. **for** i:=1 **to** 'number of nodes'
  - 2.1 Write number of edges running out of node
  - 2.2 Write edges one by one
  - 2.3 Write node-related information
3. **end for**

**Read algorithm**

1. **while not**(EOF)
  - 1.1 read *num\_of\_edges*
  - 1.2 **for** i:=1 **to** *num\_of\_edges*
    - 1.2.1 read edge
    - 1.2.2.replace node number by memory pointer
  - 1.3 **end for**
  - 1.4 read node-related information
  - 1.5 add node to look-up table
2. **end while**
3. root := 'last node'
4. Traverse down the tree and resolve suffix links
5. delete look-up table

**Fig. 4.** Suffix tree storing algorithms

Performance results of using the STOD approach with Cluster are depicted in *figure 5*. Seven different jobs were created with different total file sizes. This figure depicts the time needed to run jobs by Cluster, including copying files to nodes and running the algorithm, and the real running time, which is spent only on running the algorithm. As we can see depending on the file size the overhead of Cluster is between 4 and 10 seconds. If we compare this value to the real running time of the algorithm we can conclude that at least 4 nodes are needed to beat the running time of the sequential algorithm when using Cluster. Cluster is optimized for computation intensive jobs rather than data intensive jobs.

**Fig. 5.** Cluster running time vs. real running time

### 4.3 Parallel Algorithm Using the MPI Interface

If we want to bypass Cluster in order not to waste time on reading files and building trees repeatedly we can write our own parallelism management program using the MPI library. We read the suspicious document into memory (STCD) or build the suffix tree (STOD) on each processor and we start one more process, which will provide the files to the analyzing processes. In practice, we can use the shared disk space and the scheduling process only has to supply the name of the file to be analysed. This file can be retrieved from disk by the process. Alternatively, the scheduling process can send the whole file in a message to the process to analyze. In this case the node providing the files can become a bottleneck if many processors are available for processing.

If the files are distributed among the nodes we can apply a different load-balancing scheme. Every node primarily analyzes those files that are located on their local disks. Having been finished with all files the process can request another file from another process, which still has files left to process. This scheme is only useful if the time required to transfer a file from one node to another is less than the time required to analyze the document.

## 5. Conclusions and Future Work

In this paper we described the extended use of string matching algorithms for overlap in large collections of textual digital documents.

We presented applications of suffix trees and described an algorithm for building the more efficient representation of a suffix tree for relatively large documents. The matching statistics algorithm of [6] was used in two algorithms for defining overlaps between documents. With these algorithms we can efficiently find overlaps between documents and parameters can be used to fine-tune these algorithms.

We also introduced different approaches to parallelize our algorithms using Cluster and the MPI library. We have analysed efficient representation of a suffix tree on disk. Performance analysis of these parallel approaches is under way.

The discussion on how to find candidate-documents is outside the scope of this paper. This problem is a hot issue because potentially the whole World Wide Web might be analysed.

Section 2.2 described the space requirements of a suffix tree, which can be prohibitive in the case of very large documents. Efficient parallel algorithms must be analysed to build and use suffix trees for the matching statistics algorithm. This algorithm has to be able to distribute the suffix tree structure among different nodes of the cluster.

As today information is highly distributed we plan to analyse the mobile agent technologies to use our algorithm. One possible application is to send out an agent to different sites with the suspicious documents, build the suffix tree on that site and compare candidate-documents there. After having analysed that site, the agent can return the information on overlaps for that site and the overall statistics can be calculated at a central site.

## Acknowledgement

Support from Distributed Systems Technology Centre (DSTC Pty Ltd) for this project is thankfully acknowledged.

## References

1. Aho A.V. (1990). Algorithms for Finding Patterns in Strings. *Handbook of Theoretical Computer Science*. (Elsevier Science Publisher B.V.) Chapter 5 pp. 257-300.
2. Apostolico A. (1985). The Myriad Virtues of Subword Trees, in A. Apostolico and Z.Galli. *Combinatorial Algorithms on Words*. (Springer-Verlag Heidelberg) pp. 85-96.
3. Beowulf Project (1994). URL <http://beowulf.gsfc.nasa.gov/>
4. Baker M., Buyya R. (1999). Cluster Computing at a Glance in Buyya R. High Performance Cluster Computing. (Prentice Hall) pp. 3-47.
5. Clustor Manual (1999). URL <http://hathor.cs.monash.edu.au/clustor/>
6. Chang W.I., Lawler E.L. (1994). Sublinear Approximate String Matching and Biological Applications. *Algorithmica* 12. pp. 327-344.
7. Chen M.T., Seiferas J. (1985). Efficient and Elegant Subword Tree Construction, in A. Apostolico and Z.Galli. *Combinatorial Algorithms on Words*. (Springer-Verlag Heidelberg) pp. 97-107.
8. Garcia-Molina H., Shivakumar N. (1995a). The SCAM Approach To Copy Detection in Digital Libraries. *D-lib Magazine*, November.
9. Garcia-Molina H., Shivakumar N. (1995b). SCAM: A Copy Detection Mechanism for Digital Documents. *Proceedings of 2nd International Conference in Theory and Practice of Digital Libraries (DL'95), June 11 - 13, Austin, Texas*.
10. Garcia-Molina H., Shivakumar N. (1996a). Building a Scalable and Accurate Copy Detection Mechanism. *Proceedings of 1st ACM International Conference on Digital Libraries (DL'96) March, Bethesda Maryland*.
11. Garcia-Molina H., Gravano L., Shivakumar N. (1996b). dSCAM: Finding Document Copies Across Multiple Databases. *Proceedings of 4th International Conference on Parallel and Distributed Information Systems (PDIS'96), Miami Beach, Florida*.
12. Glatt Plagiarism Screening Program (1999). URL <http://www.plagiarism.com/screen.id.htm>.
13. Gropp W., Lusk E., Skjellum A. (1994). *Using MPI. Portable Parallel Programming with the Message-Passing Interface*. (The MIT Press)
14. Gusfield D. (1997). *Algorithms on Strings, Trees, and Sequences. Computer Science and Computational Biology*. (Cambridge University Press)
15. Main M.G., Lorentz R.J. (1985). Linear Time Recognition of Squarefree Strings, in A. Apostolico and Z.Galli. *Combinatorial Algorithms on Words*. (Springer-Verlag Heidelberg) pp. 271-278.
16. NOW Project (1997). URL <http://now.cs.berkeley.edu/>
17. Plagiarism.org, the Internet plagiarism detection service for authors & education (1999). URL <http://www.plagiarism.org>
18. Solaris-MC Project (1996). URL <http://www.sunlabs.com/research/solaris-mc/>
19. Ukkonen E. (1995). On-Line Construction of Suffix Trees. *Algorithmica* 14. pp 249-260.