A Scalable, Memory Efficient Multicore TEIRESIAS Implementation

A thesis presented to
the faculty of
the Russ College of Engineering and Technology of Ohio University

In partial fulfillment
of the requirements for the degree
Master of Science

Lee J. Nau
June 2011
© 2011 Lee J. Nau. All Rights Reserved.
This thesis titled
A Scalable, Memory Efficient Multicore TEIRESIAS Implementation

by
LEE J. NAU

has been approved for
the School of Electrical Engineering and Computer Science
and the Russ College of Engineering and Technology by

________________________________________
Frank Drews
Assistant Professor of Electrical Engineering and Computer Science

________________________________________
Dennis Irwin
Dean, Russ College of Engineering and Technology
Abstract

NAU, LEE J., M.S., June 2011, Computer Science

A Scalable, Memory Efficient Multicore TEIRESIAS Implementation (62 pp.)

Director of Thesis: Frank Drews

The importance of biological pattern discovery has grown greatly due to the enormous amount of biological sequence data that exists and that is being generated on a daily basis. Many techniques exist to perform this pattern or motif discovery, each having its own strengths and weaknesses. One such algorithm, TEIRESIAS, has been used in a variety of computational biology applications. It possesses attributes which make it in some ways superior to other approaches, by reducing the computational complexity required to exhaustively find patterns which are maximal in form.

While this tool continues to be cited by various research papers, a robust, open-source version does not exist. Users must rely on an aging Web-based system to use the tool, which is no longer updated. The major contribution of this thesis is the implementation of this algorithm, as well as providing its availability for any interested users. It has been verified against the existing Web-based version, in order to demonstrate its correctness.

Modern processor architectures have steadily moved toward more processing cores and parallelism, with no signs of slowing down. Thus, this implementation utilizes these architectures effectively, reducing the time-to-completion for results. Therefore, presented here is a robust and efficient implementation.

The layout of this document guides the reader through the various steps necessary to understand and appreciate how this was achieved. First, biological background and motivation is given. The TEIRESIAS algorithm is then presented in detail, thoroughly familiarizing the reader with its operation. The fundamental data structures required to realize this implementation are then explained, and how each might be appropriately applied. The strategies used to parallelize the algorithm are then presented. Results of this
parallelization are also detailed, with numerous experiments showing its efficiency. A case study is performed, demonstrating the use of the algorithm for real biological analysis. Finally, the value of the work is concluded, as well as future directions that it could take.

Approved: 

Frank Drews
Assistant Professor of Electrical Engineering and Computer Science
ACKNOWLEDGEMENTS

I would like to acknowledge several people for helping me to complete this thesis. First off all, I would like to thank my parents and family in general for pushing me to finish. From the Ohio University Bioinformatics Group, I would like to thank Jens Lichtenberg, Rami Al-Ouran, and Xiaoyu Liang for working with me over the past couple years and providing invaluable biological and practical insight.

I would like to thank especially Frank Drews for being my advisor and supporting me through the process. Many of the ideas and original inspiration for this work came from his thoughts and suggestions. I would also like to thank Lonnie Welch for introducing me to the field of Bioinformatics and providing the opportunity to work in this area, as well as providing numerous personal connections and conference opportunities.

The Ohio Supercomputer Center provided the computational resources necessary to perform the experiments presented herein, so I would like to thank them for providing the state of Ohio with such a valuable resource.

To my friends both in Athens and elsewhere, thank you for your friendship throughout this process. It was invaluable to gain perspective and provide a break from school.
# Table of Contents

Abstract ................................................................. 3  
Acknowledgements .................................................. 5  
List of Tables .......................................................... 8  
List of Figures .......................................................... 9  
1 Introduction ......................................................... 10  
   1.1 Genomic Information ........................................... 10  
   1.2 Motif Discovery ................................................ 11  
      1.2.1 Enumerative Techniques .................................. 12  
      1.2.2 Alignment-Based Techniques .............................. 12  
      1.2.3 TEIRESIAS ................................................ 13  
2 Motivation .......................................................... 15  
   2.1 Introduction .................................................. 15  
   2.2 Biological Application ....................................... 15  
   2.3 Non-Biological Application ................................... 16  
   2.4 Extensions of TEIRESIAS ..................................... 16  
   2.5 Conclusion ................................................... 17  
3 Teiresias Algorithm ................................................. 19  
   3.1 Introduction .................................................. 19  
   3.2 Preliminaries ................................................ 19  
      3.2.1 Elementary Patterns ..................................... 20  
      3.2.2 Maximal Patterns ........................................ 20  
   3.3 Algorithm ..................................................... 21  
      3.3.1 Scanning ................................................ 21  
      3.3.2 Convolution ............................................. 22  
   3.4 Time Complexity ............................................... 24  
      3.4.1 Scan .................................................... 24  
      3.4.2 Convolve ............................................... 24  
   3.5 Example ........................................................ 25  
   3.6 IBM Implementation .......................................... 26
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2</td>
<td>Example partial orderings for several patterns.</td>
<td>23</td>
</tr>
<tr>
<td>3.3</td>
<td>The maximal pattern set found as a result of running TEIRESIAS.</td>
<td>26</td>
</tr>
<tr>
<td>5.1</td>
<td>Sequentially numbered equivalence classes with associated pattern sets</td>
<td>38</td>
</tr>
<tr>
<td>7.1</td>
<td>Top twenty words, with associated patterns and scores.</td>
<td>51</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 GenBank growth, 1982-2011</td>
<td>11</td>
</tr>
<tr>
<td>4.1 Radix trie encoding DNA string: \textit{ACCAGGTA}</td>
<td>29</td>
</tr>
<tr>
<td>4.2 Radix trie space usage at various depths for \textit{C. elegans}</td>
<td>31</td>
</tr>
<tr>
<td>4.3 Radix trie construction time at various depths for \textit{C. elegans}</td>
<td>31</td>
</tr>
<tr>
<td>4.4 Suffix array for DNA string: \textit{ACCAGGTA}</td>
<td>32</td>
</tr>
<tr>
<td>6.1 Relative sizes in bp of various genomes</td>
<td>42</td>
</tr>
<tr>
<td>6.2 Number of patterns and run times for ( L = 3, W = 10 )</td>
<td>43</td>
</tr>
<tr>
<td>6.3 Number of patterns and run times for ( L = 3, W = 15 )</td>
<td>43</td>
</tr>
<tr>
<td>6.4 \textit{E. coli} speedup for ( L=3, W=10, K=50000 )</td>
<td>44</td>
</tr>
<tr>
<td>6.5 \textit{E. coli} speedup for ( L=3, W=15, K=50000 )</td>
<td>45</td>
</tr>
<tr>
<td>6.6 \textit{S. cerevisiae} speedup for ( L=3, W=10, K=100000 )</td>
<td>45</td>
</tr>
<tr>
<td>6.7 \textit{C. elegans} speedup for ( L=3, W=10, K=1000000 )</td>
<td>46</td>
</tr>
<tr>
<td>7.1 An image of the \textit{Pseudomonas fluorescens} bacterium from <a href="http://genome.jgi-psf.org/psefl/psefl.home.html">http://genome.jgi-psf.org/psefl/psefl.home.html</a></td>
<td>49</td>
</tr>
</tbody>
</table>


1 INTRODUCTION

1.1 Genomic Information

Ever since the publication of the structure of DNA [1], molecular biology has become a crucial area of biological study. The ability to study the building blocks of life at such a level was unprecedented and has led to a revolution in understanding and interpretation of life processes.

Since the mid 1970’s, the ability to sequence such DNA information, or extract the ordering of individual nucleotides within a DNA strand, has further revolutionized the field of molecular biology. Since this time, it has been possible to interpret at a chemical level the makeup of those DNA strands. At first this was a slow and non widespread process. Only specific labs had the knowledge and equipment necessary to conduct such sequencing activities.

So-called high-throughput or next generation sequencing has emerged as a less expensive and much faster process of extracting DNA information. Several such techniques exist and are becoming more widespread. It is now possible to obtain data for organisms at a relatively low cost to individual researchers or small labs. On the horizon is third generation sequencing which promises to further reduce the costs and complexities associated with obtaining DNA sequences.

The prolific use of DNA sequencing technology has led to an explosion in the amount of such data which is publicly available. The most widely used and popular repository for such information is GenBank [2]. It is sponsored and housed by the NIH. This database has grown nearly exponentially in size since its inception in 1982. Figure 1.1 depicts this growth until the most current release as of this writing (February 2011) ¹.

As more such data is generated, it becomes increasingly important to be able to effectively interpret its function and meaning. In fact, the amount of data already outstrips the rate at which it is able to be processed and annotated.

1.2 Motif Discovery

One such area of genomic interpretation is motif discovery [3]. This technique aims to automatically identify important short words or DNA string patterns which might have some regulatory or functional role. That is, these strings could have control or influence over some important biological process in an organism.

A variety of these techniques exist, some of the more popular ones including TEIRESIAS [4], Weeder [5], MEME [6], WordSpy [7], and Speller [8]. Each has its own strengths and weaknesses and may be appropriate to apply in different situations depending on computational resources available, target organisms, and analysis goals. Generally, they fall into two broad categories: enumerative and alignment-based. It is important to understand these general techniques.
1.2.1 Enumerative Techniques

Enumerative techniques completely elucidate the available genomic repeats of a given length. That is, the entire word space is defined, usually in the form of a list of \( k \)-mers, where \( k \) is a specific word length. This exhaustive approach is essentially one which generates all possible short candidate patterns. Statistical information is usually also stored along with this \( k \)-mers, such as overall frequency of occurrence or number of unique sequences with occurrences. Those words which might be over- or under-represented are further considered and post-processed, as they are the most likely to have some biological function or role.

However, a great deal of space and time is taken in order to fully enumerate this word-space. Especially as desired word length increases, the time taken to enumerate, and moreover the space required becomes prohibitive for practical computing systems (see Figure 4.2) for the space use of a traditional trie data structure used for enumeration.

An example of such a enumeration-based approach is the YMF [9] algorithm. This approach fully enumerates all words from a given set of sequences by generating a hash table which stores the counts of occurrences of all possible words. That is, not only is space used for words which occur, but also for words which may not appear. This results in \( O(|\Sigma|^l) \) space usage, where \( |\Sigma| \) is the alphabet size, and \( l \) is the length of the words to be analyzed. For instance, if \( \Sigma \) is the DNA alphabet, and \( l \) is 10, this would be on the order of 1048576 hash table entries. Once this table is generated, the words are ranked based on assigned \( z \)-scores.

1.2.2 Alignment-Based Techniques

Alignment-based techniques rely on finding similarity between one or more sequences. It is an opposite approach to enumeration, wherein the fundamental atomic unit of operation is a sequence rather than a word. Sequence alignment is attempted
between input sets using some algorithm, such as the widely-used BLAST [10]. Matching regions between sequences are sought in order to align them as best as possible.

However, these techniques are mostly useful for finding large regions of similarity between sequences or organisms. In this way, very long patterns or similarities may be found easily. Fine-grained similarities are not as easily detected. This is in contrast to enumerative approaches which excel at identifying short, repeated patterns. Additionally, the inclusion of multiple sequences in this technique becomes problematic, since they must be aligned against a "master" sequence or all-against-all which becomes extremely computationally expensive.

An example of an alignment-based approach is the AlignACE [11] algorithm. This approach uses Gibbs sampling to align sequences and motif multiple motifs that meet some alignment score, and have some minimum number of occurrences. Those that meet this tests are then tested against expected values in order to determine their level of uniqueness.

1.2.3 TEIRESIAS

This thesis focuses on an efficient and scalable implementation of the TEIRESIAS algorithm. It was invented by the IBM Computational Biology Group at IBM TJ Watson Research Center and first published in 1998. It has seen a diverse set of uses and remains available as a Web-based service: http://cbcsrv.watson.ibm.com/Tspd.html (as do many tools of this type). Also available for download are the binaries and scripts which power this Web server. However, these are somewhat out of date and do not seem to have been updated since 2004.

The benefits of this algorithm over others are several. First, it does not generate simple strings as do some other tools. That is, the generated patterns can have variable
width, as well as may contain ambiguous characters, evaluating to *motifs* and not just simple *words*.

Second, it inherently generates *maximal* patterns. These patterns are defined in Section 3.2.2. Essentially, these patterns are as specific as possible without losing any information. Any simpler patterns that they encompass are automatically discarded. This prunes the output space for the end user, who can become overwhelmed with the amount of data generated by many motif discovery tools.

Third, it avoids both *enumeration* and *alignment*. Enumerative and alignment-based techniques are the two most common classes of motif discovery techniques, as detailed above. Each has inherent benefits and limitations. TEIRESIAS manages to avoid performing either, which is a novel approach. It does not discard any information through heuristics or statistical pruning, while still exploring the complete pattern-space. In other words, it avoids full enumeration, while still considering all possible candidate patterns. However, once these candidate or *elementary* patterns (see Section 3.2.1), are found, they may be combined together to form much longer patterns, effectively performing local alignment. In this way, the benefits of alignment techniques are realized, since patterns may combine and recombine to form increasingly complex, and finally maximal patterns.
2 Motivation

2.1 Introduction

Although published roughly a decade ago, TEIRESIAS continues to be applicable in a variety of areas, both biological, and non-biological. This chapter seeks to explore these applications and their use of the algorithm. By presenting these uses, it should become apparent that the algorithm continues to be useful and novel in its approach to pattern discovery.

2.2 Biological Application

The TEIRESIAS algorithm has been used in a variety of applications in both biology and other fields. Probably the most significant discovery made using this algorithm was that of pyknons [12]. Pyknons are variable length sequences that exist in both intergenic and intronic regions also with copies in untranslated regions or coding regions [13]. In other words, they are nonrandom repeats which exist in both coding and non-coding genomic regions. This disputes the conventional wisdom that “junk” DNA has no biological purpose.

The original study of pyknons was applied to the human genome, but they are not necessarily unique to that organism. It is postulated that they might very well occur in a variety of organisms. Already a slightly different technique has been applied to A. thaliana [14] to find pyknons, suggesting that these in fact do play a biological role in perhaps many organisms.

Another application of the algorithm is given in an overview from the IBM Bioinformatics and Pattern Discovery Group. It applies TEIRESIAS to a variety of pattern discovery applications [15]. These include tandem repeat discovery, multiple sequence alignment, homology searching, association discovery, and gene expression analysis.
The WOOF [16] function is a technique to verify multiple sequence alignment. It is used to weight extracted protein family patterns. TEIRESIAS acts as a basis for this method, generating sets of maximal amino acid patterns in order to score alignments. Patterns are assigned log-likelihood scores and positional weight scores and ranked accordingly. A subset of these top-ranked patterns are then used as input to the WOOF function. In this way, an unmodified TEIRESIAS is used as a fundamental step in this analysis.

TEIRESIAS is used as the basis of a phylogenetic tree reconstruction technique as well [17]. It is used to generate patterns according to the standard algorithm. These patterns are then used to calculate a distance estimation which is used for phylogenetic tree reconstruction.

2.3 Non-Biological Application

TEIRESIAS has also been used in areas outside of computational biology. In fact, it was used as the basis of an intrusion-detection system used for network security [18]. Another work, published in 2002, is a thesis describing the use of TEIRESIAS for intrusion detection [19]. A more recent publication again used TEIRESIAS. Specifically, it used the algorithm to initialize a gene library for computer viruses in order to improve the intrusion detection system [20]. These works demonstrate the broad application of such a generalized pattern discovery system as well as its continued relevance.

2.4 Extensions of TEIRESIAS

While the "generic" TEIRESIAS continues to be used in a variety of ways, there are also extensions and modifications of the algorithm which apply in some situations. They might be more applicable for some problem types or otherwise improve for some cases the default algorithm.
The Gemoda [21] algorithm is designed for motif discovery that extends some of the ideas originally posited by TEIRESIAS. It allows motif representations using position-weight matrices and relaxes the constraints on similarity. However, it still exhaustively discovers all maximal motifs.

Another algorithm that finds patterns in a similar manner to TEIRESIAS is given in [22]. It is claimed that it is capable of finding more complex pattern types. However, for the simpler pattern types, it performs no better than the original TEIRESIAS algorithm.

The SPACE [23] algorithm is a method for finding spaced motifs. It is a more modern approach, yet shares great similarity with TEIRESIAS. It too builds progressively more complex patterns from simpler ones. A claimed improvement is that it provides greater flexibility in mismatches, and has a more efficient convolution step.

2.5 Conclusion

A comprehensive review of the uses, applications, and extensions of TEIRESIAS has been given in this chapter. Some of the applications are slightly older and originate at the time the algorithm was first published. Others emerged several years after the original publication. Still others continue to be published in the past several years.

The main application area is computational biology, specifically motif/pattern discovery in sequences. These discovered patterns may be used in a variety of ways, both for direct analysis and as input for further pruning or processing steps. However, it has been demonstrated that the flexibility of the algorithm has leant it to use in other fields, such as computer security.

Some approaches borrow from TEIRESIAS and while they might differ from its techniques and processes, they build upon the capability provided. That is, they cite TEIRESIAS as a fundamental algorithm and seek to improve upon it in simple or more complex ways.
This time span of work based upon the algorithm, as well as uses and extensions of the work indicate its continuing relevance and usefulness. However, the available implementation is aging, is closed-source, and has limited support for even modest input data sets. Therefore, the implementation of a fast, efficient, parallel version of the algorithm is a useful contribution which can be used in the future for direct analysis, or extension using new techniques.
3 TEIRESIAS ALGORITHM

3.1 Introduction

This chapter gives an in-depth description and explanation of the inner workings of the TEIRESIAS algorithm. The concepts necessary to full comprehend the algorithm’s operation are presented, in order to prepare the reader to understand the parallelization techniques presented in Chapter 5.

3.2 Preliminaries

The TEIRESIAS [4] [24] algorithm is a combinatorial pattern discovery algorithm which can be used to search for variable length patterns. Such patterns may take them form of a real character or residue, followed by some combination of dots (any character) and real characters, followed by a final residue. That is, such a pattern $P$ may take the form: $P \in \Sigma(\Sigma \cup \cdot)^*\Sigma$. Examples of this include:

- $ACGT$
- $A..T$
- $A.T.A$

The above are all examples with $\Sigma = \{A, C, G, T\}$ (the DNA alphabet). The $\cdot$ character represents the ”any” or ”don’t care” character. That is, it may represent any character from $\Sigma$. A pattern $P$ which takes the form described above is essential a regular expression of limited form. It must contain a real character at its first and last positions, and ambiguous positions may not take the from $[A|C]$ as with standard regular expressions. In other words, each position must be exactly defined as being a precise element of $\Sigma$, or any element of $\Sigma$. 

A further restriction on a pattern $P$ is defined by the use of parameters $L$ and $W$. These two parameters form the pair $< L, W >$, where $L \leq W$. A pattern $P$ is an $< L, W >$ pattern if each 1 of $P$ with length of at least $W$ contains $L$ real characters.

The parameter $K$ is a third parameter to the TEIRESIAS algorithm. It specifies the support or quorum required for a pattern $P$ to emerge as a final output of the program. That is, $K$ may take one of two forms; either it refers to the minimum number of unique sequences in an input data set a pattern must occur in to pass, or the minimum number of overall occurrences a pattern must have. Often it is useful to use the latter version for single-sequence runs or situations where unique sequence occurrences are irrelevant.

Together these three parameters to TEIRESIAS, $L, W, K$, form the required options which specify the nature of a particular instance of the algorithm. Configuring even one of these to be slightly different might very well result in a different set of output patterns.

### 3.2.1 Elementary Patterns

An Elementary Pattern is a pattern which results from the initial scanning phase of TEIRESIAS, which is detailed in the next section. It must have support of at least $K$. Additionally, it must contain exactly $L$ real characters. It may have length at most $W$, and (obviously) at least $L$. Such patterns form the basis of the TEIRESIAS algorithm, and are further combined to form longer patterns.

### 3.2.2 Maximal Patterns

A Maximal Pattern is a pattern which is formed as a result of repeated convolution steps, detailed in the next section. Essentially, it is the recursive combination of elementary patterns in order to form longer, more interesting patterns. The maximality of such patterns is a crucial concept. This refers to the fact that changing any don’t care character to a real character in a maximal pattern will result in a pattern which occurs less frequently than the described maximal pattern. Also, appending any real character to
either the beginning or end of maximal pattern will result in a less frequently occurring pattern, thereby rendering it non-maximal.

3.3 Algorithm

The TEIRESIAS algorithm contains two primary steps: scanning and convolution. Together these form the steps required to locate all maximal patterns according to the parameters $L, W, K$ given. Depending on how these parameters are selected, the burden of the execution may fall evenly or mostly on one step or another. Generally, the larger $< L, W >$, the more complex and time consuming the scan phase is. The smaller $K$, the more complex and time consuming the convolution phase is.

The time complexity and some implementation suggestions can be found in the technical report from IBM Research [25]. This provides some practical insights into appropriate data structures and gives in-depth analysis of the time complexity of the various algorithm stages. Also included are some proofs of these claims.

The two algorithm steps are detailed in the next sections, with more detailed description and explanation of them. For pseudocode adapted from [24] please see Appendix A.

3.3.1 Scanning

The scanning phase is the first step in TEIRESIAS. Essentially, the role of this stage is to scan the input sequence(s) for candidate elementary patterns. That is, patterns satisfying the $< L, W >$ and $K$ parameters described earlier are extracted from the input data set in order to perform further processing on them.

It should be noted that this scan phase elements the necessity of word-space enumeration, a time-consuming step that is required for enumerative methods.
3.3.2 Convolution

The convolution phase is the second step in TEIRESIAS. It takes as input the elementary patterns found in the scan phase. These elementary patterns are combined or convolved to form longer, more complex patterns. This step is conducted with certain rules applied. This ensures minimal algorithmic complexity and removes unnecessary processing. An example of such a convolution on elementary patterns $A.CT$ and $CT.G$ is $A.CT.G$. Notice that the last two real characters of the first pattern, and the first two real characters of the second pattern overlap. This overlap is detected and used to form the new, combined pattern. In general, the prefix and suffix of a pattern must be found in order to test for convolvability. The operation $\text{prefix}(P)$ returns exactly the FIRST $(L - 1)$ real characters (not '.' characters) of $P$. Likewise, the $\text{suffix}(P)$ operation returns exactly the LAST $(L - 1)$ real characters of $P$. The convolution operation can then be formally defined as

$$R = P \oplus Q = \begin{cases} PQ' & \text{if } \text{suffix}(P) == \text{prefix}(Q) \\ \emptyset & \text{otherwise} \end{cases}$$

where $Q'$ is the remaining string after $\text{prefix}(Q)$ has been applied. The $\emptyset$ signifies a failed convolution, i.e. two strings that do not overlap sufficiently.

As previously mentioned, the ordering of these convolutions is extremely important, both from an efficiency and correctness standpoint. It is not a lexicographical ordering, but rather a partial ordering based on a set of rules. These rules are defined in Table 3.1. An example of this ordering is given in Table 3.2.
Table 3.1: Partial Ordering Rules for Prefix- and Suffix-Wise Less Than. Found in [24]

<table>
<thead>
<tr>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \sigma_1, \sigma_2 \in \Sigma ) and ( x, y \in (\Sigma \cup ').^* )</td>
</tr>
<tr>
<td>( \sigma_1 &lt;<em>{pf} \emptyset ) and ( \sigma_1 &lt;</em>{pf} . ) and ( \emptyset &lt;_{pf} . )</td>
</tr>
<tr>
<td>( \sigma_1 x &lt;_{pf} . y )</td>
</tr>
<tr>
<td>( \sigma_1 x &lt;<em>{pf} \sigma_2 y ) if ( x &lt;</em>{pf} y )</td>
</tr>
<tr>
<td>( x &lt;<em>{pf} . y ) if ( x &lt;</em>{pf} y )</td>
</tr>
<tr>
<td>and</td>
</tr>
<tr>
<td>( \sigma_1 &lt;<em>{sf} \emptyset ) and ( \sigma_1 &lt;</em>{sf} . ) and ( \emptyset &lt;_{sf} . )</td>
</tr>
<tr>
<td>( x\sigma_1 &lt;_{sf} y ).</td>
</tr>
<tr>
<td>( x\sigma_1 &lt;<em>{sf} y\sigma_2 ) if ( x &lt;</em>{sf} y )</td>
</tr>
<tr>
<td>( x. &lt;<em>{sf} y ) if ( x &lt;</em>{sf} y )</td>
</tr>
</tbody>
</table>

Table 3.2: Example partial orderings for several patterns.

<table>
<thead>
<tr>
<th>Pattern 1</th>
<th>Pattern 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAA</td>
<td>AAA</td>
</tr>
<tr>
<td>AAA &lt;( _{pf} ) A.A</td>
<td>AAA &lt;( _{pf} ) A.A</td>
</tr>
<tr>
<td>AAA</td>
<td>AAA</td>
</tr>
<tr>
<td>AAA &lt;( _{sf} ) AAA</td>
<td>AAA &lt;( _{sf} ) A.A</td>
</tr>
<tr>
<td>AAA &lt;( _{sf} ) AAA</td>
<td>AAA &lt;( _{sf} ) A.A</td>
</tr>
<tr>
<td>A.A &lt;( _{pf} ) A..</td>
<td>..A &lt;( _{sf} ) A..</td>
</tr>
</tbody>
</table>
The basic notion of these orderings is to begin comparing two strings at either the beginning (prefix-wise) or end (suffix-wise) and compare characters at corresponding positions. The first string to have a real character where the other has a dot is lesser. If this does not occur, the shorter string is lesser.

3.4 Time Complexity

It is important to analyze the time complexity of TEIRESIAS formally, in order to understand how the various parameters bound the run times, both for the scan phase and the convolution phase. The time complexities are derived mostly from the technical report from IBM Research [25]. This document provides in-depth analysis and some proofs of the derivation of the time complexities (these are not mentioned in the other publications).

3.4.1 Scan

The first phase, scanning, has a time complexity of $O(m^{w^L})$, where $m$ is the length of the input set, $W$ is the parameter specified at run time, and $L$ is another parameter specified at run time. The derivation of this is as follows,

$$m\sum A(iW - L + i) \leq mL(AW) = mL\left(\binom{W-1}{L-1}\right) \leq mLW^{L-1} \leq mW^L$$

The practical implications of this are that the length of the input sequence(s) has an effect on the scan time, but that the parameters $W, L$ have a much more profound influence on the run time bound. Thus, especially as $W$ and $L$ grow, the time complexity greatly increases for the scanning stage. Therefore, care should be taken to properly select $W$ and $L$.

3.4.2 Convolve

The second phase, convolution, has a time complexity of

$$O(W^L m \log m + WCm \sum_{P_{\text{maximal}}} rc(P))$$

where $W, L$ and $m$ are specified as above, $P_{\text{maximal}}$ is the maximal pattern set, and $rc(P)$ denotes the number of real characters in
pattern $P$. There are never more than $W m \sum_{P \text{maximal}} rc(P)$ patterns on the stack, and therefore this is the number of maximal patterns formed by performing convolution. The parameter $C$ refers to the average number of patterns in the maximal set which must be compared against for each candidate maximal pattern.

This means that again the value for $L$ has a large impact on the overall runtime of the algorithm, for convolution also. The parameter $W$ also greatly affects run time since it figures in two both terms in the convolve time complexity. Additionally, the larger the maximal pattern set, the longer the run time. This is due to each existing maximal pattern needing to be examined for each candidate pattern $R$. Relaxing parameter constraints too far can have drastic effects on time complexity, since the maximal pattern set is consulted for each candidate.

3.5 Example

Here is an example of what a run of the TEIRESIAS algorithm produces. It is illustrative for the purposes of understanding for a simple test case, what is produced as output. The input string in this case is $ACGTCCACTGCATGGACGATGAT$. Running the algorithm with the parameters $L = 3, W = 5, K = 2$ produces the patterns listed in Table 3.3.

Taking the pattern $T.CA..G$ as an example from the above table, it is informative to break up the scanning and convolution steps and describe them in detail. First, the scanning phase produces the elementary patterns: $T.CA$ and $CA..G$. Note that this is due to $L = 3$ and $W = 5$, which means there must be exactly three literal characters in each elementary pattern, and that the maximum width is five. Many other elementary patterns are generated, but these are of particular note since they overlap. Notice that the underlined regions in the two patterns, $CA..G$ and $T.CA$, match exactly. By referring to the fact that this overlap must be $L - 1$ characters, or two in this case, it can be seen that
Table 3.3: The maximal pattern set found as a result of running TEIRESIAS.

<table>
<thead>
<tr>
<th>Occr Count</th>
<th>Seq Count</th>
<th>Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>ACG</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>ATG</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>GA.GAT</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>TG.A.G</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>T.CA..G</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>AC..C</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>G.C.A</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>C..TG</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>T..AC</td>
</tr>
</tbody>
</table>

This subpatterns are of appropriate length. It is then a straightforward step to see that these subpatterns match, and using the convolution (⊕) operation, the final maximal pattern, T.CA..G is generated, since any patterns convolved with it fail to meet the support requirements.

3.6 IBM Implementation

The TEIRESIAS algorithm, first published by IBM TJ Watson Research Center, is still available as a Web-based system [26]. The site at which it may be found is http://cbsrv.watson.ibm.com/Tspd.html. Through this, a user may either submit jobs directly, or obtain the binaries which enable the user to set up his/her own Web server.

However, there are severe drawbacks to this version. First, only binaries and/or Web submissions are available. No source code is available and the published algorithm details can be somewhat vague. Second, this system is frequently down, as found throughout testing software used for this thesis. This can be an annoyance at best or a 1 at worst.
Third, the Web server available for download is somewhat out of date and not trivially configurable. Fourth, the binaries available seem to be 32-bit compatible only, severely limiting the input data sets and parameters which may be used for successfully completed runs.

Despite these drawbacks, the IBM version is considered the gold standard version of the software/algorithm, and was used for comparison purposes with the software written for this thesis. Whenever possible, the final output produced by the IBM binaries was used as a reference as to what the ”proper” output format and results should be. Thus, the output format reflected in this thesis software copies that of the IBM implementation.

By having compatible output formats, it was possible to rigorously verify runs of this software against IBM's. A variety of runs and test sequences were used, and in a vast array of tests, the output verified as correct. Therefore, it is believed that this implementation can be used to obtain maximal pattern set outputs which exactly match the release available from the IBM group.
4 DATA STRUCTURES

4.1 Introduction

A fundamental portion of this algorithm, and in fact across most motif discovery algorithms, is the ability to quickly determine statistics for a pattern. That is, it is necessary to be able to find, for this particular algorithm, the overall number of occurrences and unique sequence occurrences. By gathering this information, a pattern might be further considered for maximal pattern candidacy, or discarded due to insufficient number of appearances within the input data set. How quickly such information can be gathered has a profound effect on the performance of the implementation, acting as a bottleneck for the entire pipeline.

In particular, the lookup of both elementary, and candidate maximal patterns is a crucial and possibly time-consuming step in the TEIRESIAS algorithm. Pattern overall frequencies and unique sequence occurrences are both values which determine whether or not a pattern is further considered. A pattern must meet the minimum $K$ value in this regard. These occurrence lookups are used for both algorithm stages: scanning and convolution. Especially for convolution-heavy configurations, the majority of processing time is spent executing these data structure queries for pattern frequencies. Therefore, it is important to understand the underlying use of these data structures, and how and when they should be appropriately applied.

Several data structures are supported, each possessing its own strengths and weaknesses. Space efficiency, lookup efficiency, construction time, and access patterns are all important facets and might affect a user’s decision about which to use depending on the circumstances. The various structures are detailed in the following sections.
4.2 Radix Trie

The radix trie or PATRICIA [27] (Practical Algorithm To Retrieve Information Coded in Alphanumeric) is a data structure which encodes strings based on edges emerging from trie nodes. It is a versatile method for indexing large text files, as well as retrieval of information from them. It provides a reasonable tradeoff between algorithmic complexity and space efficiency. Radix tries differ from radix trees in that the former encodes multiple characters per edge, while the latter only encodes a single character per edge. Figure 4.1 presents a graphical representation of a trie for string: ACCAGGTA.

![Radix Trie Diagram](image)

Figure 4.1: Radix trie encoding DNA string: ACCAGGTA

The radix trie has been shown to be an effective data structure for word searching [28]. That is, patterns which contain no ambiguous characters may be very quickly retrieved using this data structure. This search time complexity is $O(m)$, where $m$ is the length of the search target. It is necessary for the TEIRESIAS algorithm to be able
to not only find these types of patterns, but also those represented by a regular expression. Therefore, it is necessary to have a search algorithm which supports these. The one implemented here was able to achieve this by implementing a recursive lookup. That is, whenever a ‘.’ character is encountered, all branches of that node are searched as well, returning an aggregate of all occurrences found. This introduces some overhead, but allows the traditional trie construction to be performed without any modification.

Space usage for a radix trie is a crucial concern, and is often the cause of out of memory problems. Formally, the space complexity is $O(n^2)$ in worse-case scenarios, where $n$ is the depth of the trie. In general, the space complexity is $O(c \times n \times m)$, where $c$ is some constant factor which indicates node size, $n$ is the trie depth, and $m$ is the average length of the words stored in the trie.

The radix trie is reasonably space efficient at small depths, but it can greatly grow in size as the length of words it encodes grows. Practically speaking, it was usually constructed to depth 20 for this implementation. This allowed memory usage to remain reasonable, while still supporting medium-length patterns. Figure 4.2 shows the space usage for a radix trie with the *C. elegans* organism. Figure 4.3 shows the construction time for the same trie.

As can be seen, the radix trie grows to an enormous size, even for a 100 Mb input set. The trie size for depth 27 reached approximately 59 GB, and attempts to build to depth 28 resulted in the crashing of the node being used. The machines used to build to such depths had 64 GB of main memory: the largest available at the Ohio Supercomputer Center. As such, many more "practical" machines could not even begin to construct to this depth without thrashing terribly. In fact, these effects can already be seen in Figure 4.3. The construction times consistently increase until main memory is almost exhausted, at which point they increase dramatically.
4.3 Suffix Array

The suffix array [29] is a space-efficient data structure inspired by the suffix tree [30] that is stored and processed much differently. Its chief merit is low space complexity, occupying only $O(n)$ space, where $n$ is the length of the input string [29]. Practically
speaking, there is usually a constant multiplier for $n$ that takes into account the size of an integer on a particular machine. This is frequently 4 bytes, making the practical space complexity $O(4n)$. For instance, a 4 megabyte input string would occupy approximately 16 megabytes of memory, (not including that input string which must persist in memory also). The suffix array is a sorted list of all prefixes for some input text. An example of this for the string $ACCAGGTA$ is given in Figure 4.4.

<table>
<thead>
<tr>
<th>SA Index ($i$)</th>
<th>Text Index ($t$)</th>
<th>Suffix ($S_t$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7</td>
<td>$A$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>$ACCAGGTA$</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>$AGGTA$</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>$CAGGTA$</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>$CCAGGTA$</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>$GGTA$</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>$GTA$</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>$TA$</td>
</tr>
</tbody>
</table>

Figure 4.4: Suffix array for DNA string: $ACCAGGTA$

A suffix array may be searched for a target string using a simple binary search, until the sought after string either matches an entry or no such entry is found. A target may match multiple entries, in which case the target is a prefix or complete match for those entries. For instance if the target was $A$ in Figure 4.4, array indices $\{0, 1, 2\}$ would match. This also shows that the target appears a total of three times, which amounts to an occurrence count.
Obviously this search technique is slower than that of a radix trie, with the binary search having a time complexity of $O(n \log n)$. Especially for shorter strings, the suffix array’s performance suffers. Like the radix trie, special measures must be taken to implement a regular expression search on a suffix array. The technique used for this implementation was to first binary search the first contiguous string of non-ambiguous characters, then use the boundary indices found to limit a linear search through those entries, where matches for real characters are sought.

### 4.4 Hybrid Approach

The above two data structures each have their benefits and drawbacks, and neither is suitable for all parameter and input space configurations. An approach taken in this implementation was a hybrid one in which both types of structures are constructed; the radix trie is built to a specific depth and the complete suffix array is created. All searches up to the depth of the radix trie are conducted on that trie, and if any target exceeds this length, the suffix array is consulted for pattern statistics. In this way, the speed of the radix trie is maintained for most searches, while keeping space usage in check. For lengthy patterns, the suffix array is searched, which maintains a low memory footprint.

### 4.5 Linear Search

The most naive search approach is a linear scan of the input sequence(s), which simply runs a sliding window over those sequences and maintains a count of occurrences. This technique can be useful for very small input sets where simplicity is desired or there is very little available memory. It can also be used as a verification step to ensure that more complex data structure routines are behaving properly. While not encouraged, this approach has been implemented for completeness.
4.6 Conclusion

The appropriate data structure for a set of input parameters and data set varies and depends on the available environment. If an extremely fast search is sought and memory is plentiful, the radix trie is by far the best approach taken here. If available memory is minimal, and only fast searches are less important, the suffix array is likely the most desirable. In many cases, there is a tradeoff between time and space complexity, and some hybrid approach is most appropriate, with the radix trie being built to the maximum depth the system is capable of acceptably supporting. As a practical rule of thumb, the radix trie was typically constructed to depth 20 in the implementation for this thesis.
5 Parallelization of the Teiresias Algorithm

5.1 Introduction

In order to achieve a performance improvement in the algorithm, parallel computing was used in order to reduce the time-to-completion of the algorithm. Chapter 4 focused on efficient sequential computation, using the most time-efficient data structures possible. Space use was also kept at a reasonable level as detailed in that chapter. However, this does not make full use of the computational resources available in modern processors.

All modern mainstream processors have moved from a single thread of execution, or sequential execution model to one of several parallel threads. These are not just realized through time-sharing of processing elements, but rather through the existence of multiple processing cores on a single chip. That is, fully capable processors are replicated multiple times across a single CPU chip, and may operate completely independently. The number of these replications varies, and increases each year. At first it was only two per chip, but for the experiments run here, four per chip was the standard. Eight per chip are soon to arrive, and such a doubling is not expected to halt anytime soon.

Thus, in order to make full use of the computational power of modern multicore chips, multiple threads of execution must operate concurrently. That is, multiple processing steps must be in operation simultaneously. The strategy taken for such parallelization of tasks depends on the nature of those tasks and the available inherent concurrency. The specific methods used are described in the following sections.

5.2 Scan

The scanning stage is what is commonly referred to as an embarrassingly parallel computing task. That is, it is nearly trivial to parallelize such tasks, since concurrency is readily available. For this stage, each alphabet element is considered in succession, and extended if appropriate, independent of the other alphabet elements. For instance, if
\[ \Sigma = \{A, C, G, T\} \], then the characters \( A \) and \( C \) can be extended into elementary patterns separately. If an elementary pattern stems from \( A \) this has no effect on those stemming from \( C \).

Thus, a separate thread was spawn to handle each alphabet character simultaneously. For the DNA alphabet, this lends itself extremely well to a quad-core processor. Each core handles an independent alphabet character and simultaneously reports elementary patterns.

The domain decomposition for this stage is \textit{task} decomposition, in that each alphabet character is given its own task and set to execute on a separate processor core. Each task independently generates elementary patterns, with no dependency on other tasks. Data decomposition also occurs in this phase, in a straightforward manner. The alphabet characters are placed into an array, thus the initial data for each task lies in a separate array cell. The data structures to be queried themselves are shared, since no active updates are performed during this stage.

\subsection*{5.3 Convolve}

The convolution stage is not as easily parallelized as scanning. As observed in Appendix A, the convolution stage is a recursive algorithm which defines a strict data processing order. Maximal patterns must be found in an order that guarantees any sub-maximal patterns subsumed by them are found \textit{after} those maximal patterns. This guarantees that no non-maximal pattern will accidentally be reported out of order. This ensures correctness and also efficiency. Maximal patterns could be reported out of order, but this would require an all-against-all check at the end of the convolution stage in order to weed out non-maximal patterns, which would be a very expensive step.

\subsection*{5.3.1 Equivalence Sets}

The prefix- and suffix-wise ordering as defined in Section 3.3.2 is a \textit{partial} ordering, ensuring that certain patterns be less than or greater than others. However, if two patterns
fail to be differentiated by these rules, then they are essentially equivalent from the standpoint of the order of convolution. In other words, some patterns are distinctly separated by these rules, and are therefore placed into what are essentially separate equivalence sets. These sets therefore have a relative ordering, from smallest to largest, either prefix- or suffix-wise.

While these require that sets be processed sequentially in prefix-wise less ordering, it is unspecified how elements within these sets must be processed. Since elements are order-wise equivalent, they belong to the same equivalence class, and may be processed simultaneously. This is because they do will not conflict for the generation of maximal patterns, and any such patterns generated can be produced concurrently.

Such concurrency is exploitable for a parallel implementation, but there must be a sufficient number of equivalent elements to process simultaneously, otherwise processing cores will sit idle. Analysis was done on the sizes of these equivalence classes, and it was determined, for moderate values of $K$ that these sets were suitably large to enable such $0$. Specifically, if $L = 3$ and $W = 10$, assume that all elementary patterns will be enumerated for each equivalence class. This means that $|\Sigma|^L$ candidate elementary patterns will exist in each set. For the case of the DNA alphabet, with $L = 3$, this means that there will be $4^3 = 64$ elements in each set. As the value of $K$ grows larger, more patterns will be pruned from the elementary pattern sets, reducing the available amount of concurrency. In practice, this did not present much of a problem, as only a few were removed from the total enumeration space of the elementary patterns. It is also important to note that this expresses the amount of concurrency for the smallest value of $L$ that was experimentally considered. As $L$ grows larger, there will be many more potential elementary patterns in each set.

In order to demonstrate patterns which belong to various equivalence classes, an example is given here which makes the concept more clear. These prefix-wise less
equivalence classes are ordered as such, with each class being numbered sequentially. The example patterns in each class is not an exhaustive list, but should be sufficiently illustrative. Notice that the ‘.’ character shifts to the right with each subsequent class. In this example, those patterns which belong to the same class may be processed simultaneously. These example classes are given in Table 5.1.

Table 5.1: Sequentially numbered equivalence classes with associated pattern sets

<table>
<thead>
<tr>
<th>Equivalence Class</th>
<th>Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AAA, ACA, GTA, GCC, TTT</td>
</tr>
<tr>
<td>2</td>
<td>AA.A, ACA, C.C.T, GA.G, T.T.T</td>
</tr>
<tr>
<td>4</td>
<td>A.AC, A.CG, C.C.T, T.AA, T.TC</td>
</tr>
<tr>
<td>5</td>
<td>A.AA, A.A.T, C.G.T, T.A.C, T.A.G</td>
</tr>
</tbody>
</table>

5.3.2 Decomposition

While the scanning stage consisted of a fairly easy parallel decomposition, this stage is more challenging. As noted above, dependencies exist in the order in which elementary patterns are their convolved combinations are processed. Improper processing order results in invalid maximal patterns which should not be reported. This particular factor makes it impossible to simply task decompose the elementary patterns and process them simultaneously.

As noted above, however, equivalence classes do exist for these elementary patterns. That is, since no ordering is defined between elements in the same class, they by definition may be processed simultaneously, since they may be processed in any order without
generating incorrect maximal patterns. This fact is exploited in order to perform a limited

type of task decomposition. Candidate elementary patterns which exist in the same

equivalence class are mapped to tasks and therefore processor cores. Since they represent
different seed patterns, any maximal patterns generated from each of them is independent
and need not be checked against the others for maximality. The data decomposition in this
case is the use of separate elementary patterns which are stored in an array. Each cell
represents one piece of data which is used by each corresponding task.

For all values of $L$, $W$ used, there existed significantly sufficient parallel tasks to fully
utilize the processor cores available, for the largest machines used. As was mentioned in
the previous section, the size of each equivalence class was usually 64, representing four
times the number of available cores.

### 5.4 OpenMP

The OpenMP\(^2\) parallel programming API is widely used in multicore processing, due
to its ease of use and ability to rapidly parallelize potentially concurrent code blocks. It
was selected for use in this implementation in order to minimally obscure the existing
sequential code base, and to enable a scalable number of processors/processing cores to
operate simultaneously.

This model was used for both the scanning and convolution phases, enabling as much
parallel processing as was made available by the inherent concurrency and total number of
processing cores. This approach allows the code base to support as many cores as are
available, and was tested for up to 16 experimentally.

As was described, the parallel model used was primarily task-based parallel
decomposition. This maps nicely to the OpenMP programming model, and allows a clean
way to specify the inherent concurrency available in the algorithm.

\(^2\) http://openmp.org/wp/
5.5 Conclusion

The methods described in this section form the basis of the parallelization techniques used to accelerate the completion times of the algorithm. The task decomposition and concepts used to generate independent parallel tasks have been thoroughly described, and indicate the best possible methods for concurrent processing discovered.

In the next chapter, results are given which indicate the feasibility of this approach. For a variety of datasets, this parallel model was used and successfully improved the run times of the implementation without prohibitive space consumption. The details are delayed until that chapter, where in-depth analysis of the run times is conducted.
6 Results

6.1 Introduction

This chapter presents the results obtained from experiments conducted to determine the performance speedup obtained from parallelization. A variety of organisms were used with varying $L$, $W$, $K$ parameters.

6.2 Computing Environment

All experiments were run at the Ohio Supercomputer Center on the Glenn Cluster. The particular nodes used were those available with the most processing cores and main memory. Specifically, the compute nodes were IBM System x3755, each with quad-socket, quad-core processors. The processors themselves were AMD Opteron(tm) 8378, operating at 2.4 GHz and having 512 KB cache. The GNU C Compiler (gcc) version 4.1.2 was used to compile in all cases, with the following flags: -O3 -funroll-loops -fopenmp. The operating system was Red Hat Enterprise Linux 4.1.2. Each node had 64 GB of RAM, amounting to 4 GB per core.

The use of this system allowed the building of deeper radix tries than would have been possible on less memory-rich nodes. Also, the use of 16 processing cores total resulted in significant speedup, and represents the best resources available at the time of this writing.

6.3 Organisms

Several model organisms were used, including *E. coli*, *S. cerevisiae*, and *C. elegans*. Each represents an increasingly larger set which demands higher computational resources for successful completion. Their sizes in number of base pairs are given in Table 6.3. A graphical representation is given in Figure 6.1.

---

3 http://www.osc.edu/
### Table 6.1: Organism Sizes

<table>
<thead>
<tr>
<th>Organism</th>
<th>Size (bp)</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>E. coli</em></td>
<td>4,639,221</td>
</tr>
<tr>
<td><em>S. cerevisiae</em></td>
<td>12,495,682</td>
</tr>
<tr>
<td><em>C. elegans</em></td>
<td>100,258,171</td>
</tr>
</tbody>
</table>

Figure 6.1: Relative sizes in bp of various genomes

#### 6.4 Effect of Parameter Space

A vast parameter configuration space is possible with the three variables $L$, $W$, $K$. Their combination can profoundly effect overall runtime, memory usage, number of elementary patterns, and number of maximal patterns. In order to study the effects of this space, several experiments were run with different values for $L$, $W$, $K$. The organism selected for these experiments was *E. coli*, due to it being large enough to demonstrate many of the effects of parameter configuration, while still being capable of finishing in a reasonable amount of time.
The first and perhaps most useful effect of this configuration space regulates the number of overall maximal patterns reported upon algorithm completion. As \( K \) is lowered, more patterns pass through the filters and are reported, since they are required to appear fewer times in the input sequence(s). This effect is depicted in Figure 6.2 for \( W = 10 \), which shows the number of overall patterns, as well as time taken to find them, for various values of \( K \). It is also depicted in Figure 6.3, which increases \( W \) to 15.

![Figure 6.2: Number of patterns and run times for \( L = 3, W = 10 \)](image1)

![Figure 6.3: Number of patterns and run times for \( L = 3, W = 15 \)](image2)
6.5 Speedup

The below figures give a spectrum of input sets and the speedups attained using various $L, W, K$ parameters. That is, speedup is a multiplier indicating how much less time a run took overall. The number of threads used for these experiments was \{1, 4, 8, 12, 16\}. This provides an increasing number which allows insight into how performance changes as more are added. The number of threads was controlled through changing the OpenMP environment variable `OMP_NUM_THREADS`. In general, these figures demonstrate the effectiveness of the parallelization by showing decreased run time with increased number of threads.

![Figure 6.4: E. coli speedup for L=3, W=10, K=50000](image-url)
### Table 1: Performance Comparison

<table>
<thead>
<tr>
<th>Threads</th>
<th>Speedup (convolution)</th>
<th>Speedup (total)</th>
<th>Speedup (linear)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>12</td>
<td>6</td>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>16</td>
<td>8</td>
<td>12</td>
<td>8</td>
</tr>
</tbody>
</table>

**Figure 6.5:** *E. coli* speedup for \( L=3, W=15, K=50000 \)

**Figure 6.6:** *S. cerevisiae* speedup for \( L=3, W=10, K=100000 \)
Figure 6.7: *C. elegans* speedup for \( L = 3, W = 10, K = 1000000 \)

### 6.6 Discussion

The parameters chosen for these runs were generally

\[ L = 3, W = 10 - 15, K = 50000 - 100000. \]

Only *E. coli* was able to extend \( W = 15 \), since the others took prohibitively long to complete. These might seem arbitrary but they correspond to real usage scenarios. For instance, in [4] (one of the original TEIRESIAS publications), several run-time analysis experiments are run. For these, \( L \) was set to 3 in all cases, \( W \) was set to either 10 or 15, and \( K \) was set to varying values in order to increase/decrease run time. The explanation given as to why \( L \) was chosen to be 3 in all cases was that it was the smallest number for which convolution was able to operate effectively. Larger values of \( L \) shift burden of computational complexity from the convolution phase to the scan phase.

In [17], the parameters were \( L = 4, W = 16, K = 2 \). Again, a small value for \( L \) is seen, as well as a moderate value for \( W \). The value of \( K \) was chosen to be 2 because pairs of sequences were being considered, and therefore the desired patterns had to exist in both.
In [16], the parameters were $L = 3, W = 15$, with $K$ unspecified. This indicated that at least three characters be conserved in a contiguous set of fifteen. Again, these settings are similar to the above two examples.

As can be seen in the above three examples, the values $L, W, K$ for these experiments were chosen to be realistically useful and mimic previously published configurations. Specifically, a small $L$ allows the maximum flexibility in positioning of real characters. This, coupled with moderate values of $W = 10 – 15$, results in convolution-heavy discovery of maximal patterns. The values for $K$ were somewhat organism-specific, and generally were selected to reduce the number of output patterns to an amount under 100,000. That is, more conserved patterns were sought.

The figures depicting speedup show generally sublinear but still moderate to high reduction in run times. The run times were split into two values: overall run time and also convolution time. Due to the selection of $< L, W >$, it was expected that convolution be the most intensive and computationally dominant stage in the algorithm. Thus, it is useful to see overall speedup as compared to convolution speedup. The other stages in the algorithm tend to be insignificant compared to the amount of time taken for convolution. It is a testament to the convolution parallelization strategy taken that such a speedup is still achieved in this stage.
7 Case Study

7.1 Introduction

In order to use the software developed here for a real-world experiment, a case study was conducted to prove the usefulness of the algorithm. This is essentially the biological application which validates the use of this bioinformatics algorithm for discovery. The algorithm is performed on an organism described in the next section, and the resultant maximal patterns are presented in the section thereafter.

7.2 Organism

The organism in question is a strain of the Pseudomonas fluorescens bacterium. It is similar to the Pf-5 strain, and is known as Wayn1R. The bacterium is known for secreting a fluorescent pigment, from which its name is derived. It inhabits aquatic, land, and plant environments. They can be used for bioremediation, since they are able to degrade pollutants. Of particular interest here is that they play a role in suppressing fungal infections in plants [31] and fighting parasitic nematodes [32]. Generally, the fighting of plant disease and general biocontrol of plant pathogens is one of the important properties of Pseudomonas fluorescens [33] [34] [35]. Due to its ability to fight fungal infections of plant roots, it is of agricultural interest and could be used to fight disease in corn and soybeans [36]. The organism is pictured in Figure 7.1.

7.3 Method

The contigs of the Wayn1R strain were obtained from a collaborator, and consisted of approximately 6.7 million base pairs. The Rapid Annotation using Subsystem Technology or RAST [37] service in order to annotate the genes in the contigs. This resulted in an annotation file which specified gene locations, lengths, strand types, and other useful

---

4 http://cmr.jcvi.org/tigr-scripts/CMR/GenomePage.cgi?org=gpf
information. For those genes which were protein encoding genes, the upstream promoter sequences were sought. For operons, which were defined to have fifty base pairs or less between their genes, only a single promoter was extracted.

A python script was written using the BioPython [38] library in order to extract the upstream promoter sequences. These sequences corresponded to promoter sequences that were at least 51 bp in length, and no more than 1000 bp in length. Also, sequences containing strings of $N$ characters were pruned, as these contained ambiguous data and were therefore not useful. A total of 3957 promoter sequences were obtained, and this final data set was used for further analysis.

7.4 Results

In order to conduct this case study, TEIRESIAS was run with $L = 3, W = 10, K = 100000$. The $L, W$ parameters were selected as default values due to reasons discussed previously. The value of $K$ was selected to limit the number of maximal patterns output to a reasonable amount, using only those which occurred many times. The
below Table 7.1 contains twenty such patterns extracted, their associated words, and scores for both. The scores for words were found using a second-order homogeneous Markov chain [39]. Only those words which actually occurred in the input sequences were considered (it is possible patterns could expand to non-existing words). The patterns themselves then received an aggregate score from the words they represented. This score was an average of those obtained for the individual words generated by that pattern. The table below is sorted by overall word scores, so some words appear as duplicates in the table. This lends credibility to the patterns they are expanded from, since these patterns might be associated with multiple words.

7.5 Discussion

The generated patterns/motifs in the results table identify possible functional elements within the promoters themselves. The fact that they must occur so many times ensures that they meet a minimum quorum. Additionally, since they are scored, the results indicate a ranking that is not possible to achieve with the standalone TEIRESIAS tool or as described in the literature. This extension combines the power of TEIRESIAS to find interesting maximal patterns with the information obtained by applying a scoring model to the words generated by those patterns.

The ability to sort the output data by either overall word score or overall pattern score is a powerful mechanism to identify either the top words or top patterns obtained by running the algorithm and applying the scoring technique. Thus, as shown in Table 7.1, duplicates of a word or pattern might appear, and are in fact even likely to do so. This indicates that a high-scoring word might be generated by several patterns. Also, since those patterns which are most highly scored are the aggregate of several words they generate, it is likely for those patterns to appear multiple times in the top scored ones.
Table 7.1: Top twenty words, with associated patterns and scores.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Word</th>
<th>Word Score</th>
<th>Pattern Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.....C.C</td>
<td>CGGCTCCTAC</td>
<td>38.2244</td>
<td>1.601</td>
</tr>
<tr>
<td>C..C.....C</td>
<td>CGGCTCCTAC</td>
<td>38.2244</td>
<td>1.55813</td>
</tr>
<tr>
<td>C.G......C</td>
<td>CGGCTCCTAC</td>
<td>38.2244</td>
<td>1.15861</td>
</tr>
<tr>
<td>C.....C.C</td>
<td>CTACTACGAC</td>
<td>37.0357</td>
<td>1.601</td>
</tr>
<tr>
<td>C..C.....C</td>
<td>CTACTACGAC</td>
<td>37.0357</td>
<td>1.55813</td>
</tr>
<tr>
<td>C.G......C</td>
<td>CTACTACGAC</td>
<td>37.0357</td>
<td>1.30259</td>
</tr>
<tr>
<td>G..G.....G</td>
<td>GTAGGAGCCG</td>
<td>36.0448</td>
<td>1.58048</td>
</tr>
<tr>
<td>G.....G..G</td>
<td>GTAGGAGCCG</td>
<td>36.0448</td>
<td>1.53167</td>
</tr>
<tr>
<td>G......G.C</td>
<td>GTAGGAGCCG</td>
<td>36.0448</td>
<td>1.15556</td>
</tr>
<tr>
<td>C.G......C</td>
<td>CTGTAGGAGC</td>
<td>34.2382</td>
<td>1.15861</td>
</tr>
<tr>
<td>C......GC</td>
<td>CTGTAGGAGC</td>
<td>34.2382</td>
<td>1.14814</td>
</tr>
<tr>
<td>G..G.....G</td>
<td>GTAGTAATAG</td>
<td>33.5829</td>
<td>1.58048</td>
</tr>
<tr>
<td>G..G.....G</td>
<td>GTAGGAGCTG</td>
<td>33.2156</td>
<td>1.53167</td>
</tr>
<tr>
<td>G..G.....G</td>
<td>GTAGGAGCTG</td>
<td>33.2156</td>
<td>1.53167</td>
</tr>
<tr>
<td>G......C.G</td>
<td>GTAGGAGCTG</td>
<td>33.2156</td>
<td>1.15556</td>
</tr>
<tr>
<td>C.....C.C</td>
<td>CAGCTCCTAC</td>
<td>31.5819</td>
<td>1.601</td>
</tr>
<tr>
<td>C..C.....C</td>
<td>CAGCTCCTAC</td>
<td>31.5819</td>
<td>1.55813</td>
</tr>
<tr>
<td>C.G......C</td>
<td>CAGCTCCTAC</td>
<td>31.5819</td>
<td>1.15861</td>
</tr>
</tbody>
</table>

These patterns and/or words can be further analyzed to detect their biological significance.
8 Conclusion

This thesis has presented a thorough explanation of the operation of the TEIRESIAS algorithm, describing each step in detail. Additionally, the considerations needed for an actual implementation were discussed. The fundamental data structures necessary to power the algorithm were explained, as well as their implementation uses and pitfalls. Parallelization strategies were discussed and showed the methods used to go about significantly improving the performance of the algorithm for real-world case studies. Even for a seemingly unparallelizable convolution stage, which often is the most computationally demanding step, strategies were found to exploit as much concurrency as possible.

Not only were the methods and details of a practical implementation given, but also the motivation to even begin such an endeavor. Recent and widely cited biological case studies were given, demonstrating the continuing relevance and importance of the TEIRESIAS algorithm. While other motif discovery tools and algorithms continue to emerge, this one continues to be used and modified in a variety of ways.

The lack of a robust, practical, and efficient implementation prompted this work, in order to provide a useful tool to the bioinformatics community at large, as well as the local group at Ohio University. This implementation was thoroughly tested and analyzed, and produced correct results across a collection of real biological data sets.

Not only is the implementation efficient sequentially, it also makes use of modern processor architecture by using multicore programming strategies in order to reduce the amount of time it takes to produce an output set. Although only tested with up to 16 processing cores, theoretically as many as 64 or beyond could be used to accelerate the performance of the algorithm.

Finally, a case study was analyzed in order to demonstrate the biological merit of this approach on real data. The implementation was applied to a real bacterium’s promoter
sequences, and used to generate motifs which can be further analyzed for biological significance.

8.1 Future Work

This work has presented a significant contribution to the available tools for maximal pattern discovery. However, additional improvements can always be made. Some of these are discussed here, lending ideas to future users about how this work could be extended.

First, the notion of the use of a limited regular expression for a pattern could be extended. Currently, by definition, a pattern can only consist of real characters or "don’t care" characters. If a true regular expression could be used, such as one with an OR rule instead of just a wild-card matching rule, the precision with which patterns could be described would be greater and possibly allow greater biological insight. For instance, the regular expression $A[A|C]T$ would match either $AAT$ or $ACT$, whereas currently the pattern $A.T$ would match $AAT$, $ACT$, $AGT$, or $ATT$.

Second, the flexibility of gaps could be used to refine which patterns are considered maximal and which are not. For instance, if flexible gaps were allowed, the patterns $A...C$ and $A.....C$ might be considered equivalent, thus removing redundant information from the output set, or be processed simultaneously due to their equivalence.

Third, the use of more processing cores could be attempted, in order to determine how much of a speedup could be achieved and if there might be some point of diminishing returns. As stated, the best available machines at the time of this writing were 16-core machines. It is not infeasible that in the near future this software or a modified version could be used on machines with double or more this number of cores.

Fourth, the use of a scoring model applied mid-algorithm could be explored in order to see if there is a way to reduce the output data space. While scoring is strictly left out of the traditional TEIRESIAS algorithm, limited use during the scanning phase for
elementary patterns and the convolution phase for maximal patterns could possibly produce more meaningful output for an end user. In general, more advanced scoring models could also be applied. These might further prune the output patterns/words to some more meaningful subset by taking into account other factors.

Finally, the application of this algorithm to additional datasets would be a useful measure of its usefulness across organism types and domains. While one case study has been analyzed here, future users could apply the same techniques in new and different ways to achieve their own results.
REFERENCES


[12] Isidore Rigoutsos, Tien Huynh, Kevin Miranda, Aristotelis Tsirigos, Alice Mchardy, and Daniel Platt. Short blocks from the noncoding parts of the human genome have instances within nearly all known genes and relate to biological processes.


A.1 Scan Phase

Require: set $E_P$ is empty
Require: input sequence set $S$ not empty

procedure Scan(sequence set $S$)

for all character $\sigma$ in alphabet $\Sigma$ do
    form pattern $P$ from character $\sigma$
    if $\text{Counts}(\text{pattern } P) \geq K$ then
        extend(pattern $P$)
    end if
end for

end procedure

Ensure: counts for pattern $P$ defined

procedure Counts(pattern $P$)

determine how many occurrences/sequences pattern $P$ has

return pattern $P$

end procedure

Require: input pattern $P$

procedure Extend(pattern $P$)

$A =$ number of real characters in $P$

if $A == L$ then
    Add $P$ to $E_P$ set return
end if

for $i = 0$ to $W - |P| - L + A$ do
    for all character $\sigma$ in alphabet $\Sigma$ do

end for
counts[σ] = 0

end for

$P' = P$ cat $i$ dots

for all character $σ$ in alphabet $Σ$ do

$P' = P'$ cat $σ$

counts[$P'$] = Counts(pattern $P'$)

if counts[$P'$] ≥ $K$ then

extend(pattern $P'$)

end if

end for

end for

end procedure

A.2 Convolution Phase

Require: set $EP$ ordered prefix-wise less

Require: set $MP$ is empty

Require: stack $S$ is empty

Ensure: all maximal patterns are added to set $MP$

procedure CONVOLVE(set $EP$, set $MP$, stack $S$)

while set $EP$ not empty do

$P$ is prefix-wise smallest element in $EP$

push $P$ onto stack $S$

while stack $S$ not empty do

start:

$T = $ top of stack $S$

▷ Left-convolve pattern $T$
$U = \text{set of patterns which are suffix-wise convolvable with } T$

for all pattern $Q$ in set $U$ do

$R = Q \oplus T$

if counts[$R$] == counts[$T$] then

pop stack $S$

end if

if $\text{Support}(\text{pattern } R) \geq K \text{ AND } \text{isMaximal}(\text{pattern } R)$ then

push $R$ onto stack $S$

goto start

end if

if counts[$R$] == counts[$T$] then

goto start

end if

end for

$\triangleright$ Right-convolve pattern $T$

$U = \text{set of patterns which are prefix-wise convolvable with } T$

for all pattern $Q$ in set $U$ do

$R = Q \oplus T$

if counts[$R$] == counts[$T$] then

pop stack $S$

end if

if $\text{Support}(\text{pattern } R) \geq K \text{ AND } \text{isMaximal}(\text{pattern } R)$ then

push $R$ onto stack $S$

goto start

end if

if counts[$R$] == counts[$T$] then

end if

end for
goto start

end if

end for

pop stack $S$

add $T$ to set $MP$

delete element $P$ from set $EP$

end while

end while

end procedure

procedure SUPPORT(pattern $R$)

if Occurrence Version Defined then return \texttt{counts}[$R$]

end if

if Sequence Version Defined then return \texttt{seqs}[$R$]

end if

end procedure

Ensure: maximality of pattern $R$ is determined

procedure ISMAXIMAL(candidate pattern $R$, set $MP$)

for all pattern $P$ in set $MP$ do

if pattern $R$ is subsumed by pattern $P$ then return $R$ not maximal

end if

end for

return $R$ is maximal

end procedure