Abstract— STRIKE was introduced and implemented to predict protein-protein interactions where proteins interact if they contain similar substrings of amino acids. On the yeast protein interaction literature, STRIKE was shown to improve upon the existing state-of-the-art methods for protein-protein interaction prediction. Herein, we describe the parallelization of STRIKE and its multithreaded implementation and performance enhancement on multicore systems. On large protein sequence sets, the execution time of a 16-thread implementation of this bioinformatics algorithm was reduced from about a week on a unithreaded implementation on a serial uniprocessor machine to 1.5 days on one quad core x86 machine, down to 4.5 hours on 8 such quad core machines. Key optimizations to the implementation are also discussed.

Keywords— protein-protein sequence matching; multithreaded computing; performance analysis; SIMD computing

I. INTRODUCTION

The prediction of protein-protein interaction (PPI) is one of the fundamental problems in computational biology as it can aid significantly in identifying the function of newly discovered proteins. Understanding protein-protein interactions is crucial for the investigation of intracellular signaling pathways, modeling of protein complex structures and for gaining insights into various biochemical processes. To solve this problem, many experimental techniques have been developed to predict the physical interactions which could lead to the identification of the functional relationships between proteins. These experimental techniques are however, very expensive, significantly time consuming and technically limited, resulting in a growing need for the development of computational tools that are capable of identifying PPIs. To this end, many impressive computational techniques have been developed. Each of these techniques has its own strengths and weaknesses, especially with regard to the sensitivity and specificity of the method. Some of the state-of-the-art techniques such as the Association Method (AM) [1], Maximum Likelihood Estimation (MLE) [2], Maximum Specificity Set Cover (MSSC) [3] and Domain-based Random Forest [4] have employed domain knowledge to predict PPI. The motivation behind this employment is that molecular interactions are typically mediated by a great variety of interacting domains. PIPE (Protein-Protein Interaction Prediction Engine) [5] was also developed and it is based on the assumption that some of the interactions between proteins are mediated by a finite number of short polypeptide sequences. These sequences are typically shorter than the classical domains, and are used repeatedly in different proteins and contexts within the cell. However, identifying domains or short polypeptide sequences is a long and computationally expensive process. These techniques are also not universal because the accuracy and reliability of these methods is dependent on the domain information of the protein partners.

In this paper, we introduce a novel algorithm termed STRIKE which employs String Kernel to predict PPI. The string kernels (SK) approach has been shown to achieve good performance on text categorization tasks [6] and protein sequence classification [7]. The basic idea of this approach is to compare two protein sequences by looking at common subsequences of fixed length. The string kernel is built on the kernel method introduced by [8] and [9]. The kernel computes similarity scores between protein sequences without ever explicitly extracting the features. A subsequence is any ordered sequence of amino acids occurring in the protein sequence, where the amino acids are not necessarily contiguous. The subsequences are weighted by an exponentially decaying factor of their full length in the sequence, hence emphasizing those occurrences that are more contiguous.

We understand that the subsequences’ similarity between two proteins may not necessarily indicate interaction, however, it is evidence that we can’t ignore. Subsequence similarity helps in inferring homology. Homologous sequences usually have the same or very similar structural relationships.

A drawback of this approach is observed when the level of similarity between the protein pairs is too low to pick up interaction. The reasonable explanation is that in the case of low sequence, there are always similar patterns of identical amino acid residues which could be seen in the two sequences. The pattern of sequence similarity reflects the similarity between experimentally determined structures of the respective proteins or at least corresponds to the known key elements of one such structure [10]. Structural evidence indicates that, interacting pairs of close homologs usually interact in the same way [11]. Other evidences are shown by utilizing a combination of some genomic features such as structurally known interacting Pfam domains and sequence homology as a means to assign reliability to the PPIs in Saccharomyces cerevisiae [12]. The Likelihood ratio in this study expresses the reliability of such genomic feature. In
our case, there is no doubt that the SK method is a good indicator of homology between protein pairs. The intensive comparison between subsequences exists in protein pair may capture structural domain knowledge or typically subsequences which are shorter than the classical domains and could appear repeatedly in the protein pairs of interest. We are also encouraged by the success of a recently published work employing pairwise alignment as a way to extract meaningful futures to predict PPI. The PPI based on Pairwise Similarity (PPI-PS) method consists of a representation of each protein sequence by a vector of pairwise similarities against large subsequences of amino acids created by a shifting window which passes over concatenated protein training sequences. Each coordinate of this vector is typically the E-value of the Smith-Waterman score [13]. One major drawback of the PPI-PS is that each protein is represented by computing the Smith-Waterman score against a large subsequence created by concatenating protein training sequences. However, comparing short sequences to very long ones will result in some potentially valuable alignments to be missed out. The SK however, tackles this weakness by capturing any match or mismatch which exists in the protein sequence of interest.

In Section 2, we explain the parallel protein sequence decomposition and matching algorithm on a multi-core machine. Section 3 formally presents the parallel sequence decomposition and matching algorithm on a multi-core machine. Section 4 reports algorithm, code, and compiler optimizations on x86 multi-core computers and performance results. The paper concludes in Section 5.

II. PARALLEL PROTEIN SEQUENCE MATCHING ALGORITHM

We start explaining how the algorithm works by a simple example which compares the two short protein sequences s1="lql" and s2="lqal", where there exists one string of characters in each sequence. For computational simplicity and to meet common memory capacities of modern computers, we set the length of substring (patterns to match) to 2. In other words, these sequences are implicitly transformed into feature vectors, where each feature vector is indexed by the substrings of length 2. Table 1 shows the decomposition of each of the two sequences into 2-character substrings. Each sequence is decomposed into all possible ordered (from left to right) combinations of characters included in the sequence such that the 2 characters need not be consecutive. The first three (from the left) 2-character substrings represent the decomposition of the s1 sequence, while all 6 2-character substrings represent the decomposition of the second sequence s2.

When a 2-character substring appears in a sequence such that these 2 characters are consecutive in the sequence, the substring’s dom—degree of matching—in that sequence is represented by $\lambda^2$, where $\lambda$ is a decay factor. For instance, the substring “lq” fits this case in the first sequence. When these 2 characters are separated by another character (gap of 1), the substring’s dom is $\lambda^2 \cdot \text{gap}$. The doms for the 3 substrings for the first sequence and all substrings for the second sequence are computed in that fashion as shown in Table I. When matching the 2 sequences, the 2-character substrings to impact and increase the degree of matching must exactly appear in both sequences.

To reflect the degree of matching between the s1 and s2 sequences, the un-normalized string kernel (SK) for the 2 sequences, $k(lql,lqal)$ can clearly be computed as the dot product of the 2 rows of Table 1 containing the doms, i.e. $\lambda^4 + \lambda^2 + \lambda^2$. Assuming that the decay factor $\lambda$ is equal to 0.5, $k(lql,lqal)=0.102$. The higher the un-normalized kernel the higher the indication of matching between the 2 sequences and the higher is the interaction.

To parallelize this algorithm, we describe a highly parallel algorithm consisting of the following 3 steps:

i. Decomposition
ii. Sorting
iii. Inner Product

In the decomposition step, the amino acid sequences are allocated to processing cores, one sequence per core. For instance let us assume that the SKs of the 4 amino acid sequences “lyq,” “qyla”, “yqla” and “qla” are computed on a parallel computer with at least 4 processing cores. The goal is to find mutual interaction between these 4 sequences. The processing core allocation of the 4 sequences proceeds as shown in Fig. 1.a.

In all processing cores, the decomposition of each protein sequence proceeds in parallel and their execution times overlap in time. Each sequence is decomposed into 2-amino acid substrings starting with adjacent amino acids as shown in Fig. 1.b. The “2” in “(ly 2)” refers to the power of the weighted decay factor ($\lambda$) (i.e. $\lambda^2$) indicating no gap (i.e. 3rd character) between the “l” and the “y.”

Since the amino acids in the resulting substring are not necessarily required to be contiguous, the decomposition into 2-amino acid substrings with a non-adjacent amino acid separated by another amino acid takes place as illustrated in Fig. 1.c.

Table: Mapping Two Strings “lql” and “lqal” to Six Dimensional Feature Spaces

<table>
<thead>
<tr>
<th></th>
<th>lq</th>
<th>ll</th>
<th>ql</th>
<th>la</th>
<th>qa</th>
<th>Al</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1=Φ(lql)</td>
<td>$\lambda^2$</td>
<td>$\lambda^3$</td>
<td>$\lambda^2$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>S2=Φ(lqal)</td>
<td>$\lambda^2$</td>
<td>$\lambda^4$</td>
<td>$\lambda^3$</td>
<td>$\lambda^3$</td>
<td>$\lambda^2$</td>
<td>$\lambda^2$</td>
</tr>
</tbody>
</table>
substrings composed of non-adjacent amino acid separated by 3 other amino acids takes place as shown in Fig. 1.d.

As processing cores 1 and 4 have shorter sequences to process in step 1, they will complete step 1 ahead of processing cores 2 and 3. Thus processing cores 1 and 4 can immediately proceed to step 2, while processing cores 2 and 3 will proceed to step 2 immediately after completing step 1.

In the second step of the parallel algorithm, the 2-amino acid substrings generated in the first step are sorted alphabetically based on their 2-letter string content. Again each core sorts its strings alphabetically in parallel with the other cores so the string sortings in all 4 processing cores overlap in time. After step two completes, the 4 processing cores will have for content the sorted strings shown in Fig. 1.e.

In the third step, the inner products are carried out on half (4/2=2) the cores with the largest substring set cardinality. This choice is made to minimize the total inter-core communication time. In our example, cores 2 and 3 have the highest number of generated substrings. Each of these cores maintains its 2-amino acid substrings and receives 3 amino acid strings generated by the core which is allocated the other sequence to match with its sequence. To simplify this example, let us say that our goal is to match the protein sequences “lyq” (processing core 1) and “qyla” (processing core 2) together, and the protein sequences “yqla” (processing core 3) and “qla” (processing core 4) together, in step 3, and not all the 4 sequences with each other. As a result, the following data communications will take place, as shown in Fig. 1.f.
In case of a message-passing system, the data communication takes place in the form of messages sent by the sender cores (1 and 4) to the destination cores (2 and 3). In case of a shared memory system, processing cores 2 and 3 read the 2-amino acid substring data generated by cores 1 and 4 from shared memory. After the data is received or read by the destination cores, the processing cores 2 and 3 will hold the substrings shown in Fig. 1.g. Processing cores 1 and 4 need not remain active.

In our example, processing cores 2 and 3 then start performing the inner products between their strings generated in step 2 and the received strings generated by the neighboring core are shown in Fig. 1.h. The inner product \( \alpha n \) succeeds when both strings match i.e. \( \alpha = \beta \), producing the number \( n^m \) (representing \( \lambda^{n^m} \)). Otherwise if \( \alpha \) is different from \( \beta \), then it’s a mismatch (resulting in 0). Thus cores 2 and 3 will simultaneously perform the following inner products. Note that core 2 will take the product of ly (followed by lq, and yq, respectively) and the substrings in the other set. The results are presented by each core involved in the inner product step as follows:

<table>
<thead>
<tr>
<th>Processing core</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0 (no match)</td>
</tr>
<tr>
<td>3</td>
<td>4, 6, 4: ( \alpha^2 + \alpha^n + \lambda^4 )</td>
</tr>
<tr>
<td></td>
<td>( = 2 \alpha^2 + \lambda^4 )</td>
</tr>
</tbody>
</table>

On a processing core, matching two substrings starting with the same amino acid will speed up the kernel computation step. After matching a string with all other substrings starting with the same amino acid, the remaining strings in the second sequence can be skipped as the strings have been sorted in alphabetical order in the second step. For instance, referring to above results, after matching \( \text{ly}(2) \) to \( \text{la}(2) \), processing of the string \( \text{ly}(2) \) stops as the remaining strings in the second set do not start with the amino acid l.

This could be implemented by a simple indexing mechanism based on the starting amino acid of the substrings. In the absence of such a mechanism, a string will have to be matched (i.e., its inner product taken) with all strings in the other set until a match is found or until all the strings in the other set have been exhausted.

Step 3 can be repeated as many times as needed to match other protein sequences allocated to other processing cores. For instance to match the lyq (allocated to processing core 1) and qla (allocated to processing core 4) sequences, processing core 4 sends its 2-amino acid substrings generated in step 2 to processing core 1 which carries out the inner product step. Thus the parallel algorithm is capable of matching as many sequences in parallel as desired based on the availability of processing cores. STRIKE is highly parallel and should achieve excellent performance scalability with increasing hardware resources.

III. MULTITHREADED IMPLEMENTATION ON MULTICORE COMPUTERS

In our multithreaded implementation, we make two changes to the previous algorithm. For efficiency and proper indexing, we skip the sorting step and perform matching between all the 2-character substrings of the two protein sequences to match. Second, to increase the matching accuracy, we modify the SK to be the weighted inner product of the doms \( \alpha n \) and \( \beta m \). Thus cores 2 and 3 will simultaneously perform the matching.

The parallel implementation consists of a main procedure which reads the input protein sequences, one from the training set and the other from the testing set, and amino acid matrix, launches parallel threads which are assigned an equal number of sequences to match and which generate the pairs of amino acids and their inter-distances and compute the portion of the score matrix corresponding to the sequences assigned to these threads. The matrix contains all amino acid weights corresponding to all characters in the protein sequence. Afterwards, control is passed back to the main procedure for printing the score matrix, and computing the execution time. The basic STRIKE procedure proceeds as follows.

Algorithm STRIKE

Data: - Files train.txt, test.txt containing the protein sequences
- File matrix.txt containing the weights of each amino acid in the training sequence
- value of lambda, \( \lambda \), 0.8 by default
- NumThreads, number of parallel threads to launch

Begin

1. Read all amino acid sequences from their data files. The training sequences were read from a train.txt file, while the testing sequences were read from a test.txt file, while the matrix values assigning each of the amino acid characters a weight were read from a matrix.txt file.
2. For each sequence in the training and testing sets, pair each amino acid with one subsequent amino acid, and store this pair of amino acids along with the distance between these 2 amino acids.
3. Launch NumThreads parallel threads with an equal load of sequences (except for last thread) each performing the following sequential steps:

   3.1 Set the matching score, \( s_{ij} \) corresponding to the 2 amino acid sequences \( i \) and \( j \) to 0.

   3.2 For any 2 amino acid sequences, match the pairs of amino acids of the first sequence \( i \) in the testing set with the pairs of amino acids of the second sequence \( j \) in the training set.
3.3 If both amino acid pairs exactly match, then
3.3.1 add their distances together,
\[ \text{dist}_{i,j} = \text{dist}_i + \text{dist}_j \]
3.3.2 update score_{i,j} as follows
\[ \text{score}_{i,j} = \text{score}_{i,j} + \lambda \text{dist}_{i,j} \times \]
\[ \text{matrix}(\text{amino acid}_i) \times \text{matrix}(\text{amino acid}_j), \]
where matrix(\text{a}) is the matrix value corresponding to amino acid letter "a."

4. When all threads are done all pairs of sequences assigned to them, print the sequence score_{i,j}'s as a matrix of floating point numbers with row index i and column index j. Also, calculate and print the execution time.

End

Complexity-wise, step 1 is \( O(l_1 \times n_1 + l_2 \times n_2) \) where \( n_1 \) and \( n_2 \) are the lengths of the protein sequences, and \( l_1 \) and \( l_2 \) are the numbers of protein sequences in both testing and training sets. Step 2 is \( O(l_1 \times n_1^2 + l_2 \times n_2^2) \). Step 3 is \( O(l_1 \times l_2 \times n_1 \times n_2) \). Step 4 is \( O(l_1 \times l_2) \). Therefore the entire algorithm is \( O(l_1 \times l_2 \times n_1 \times n_2) \).

IV. PERFORMANCE OPTIMIZATION AND RESULTS

The application was implemented on an x86 dual-core PC with Microsoft Visual Studio .NET 2003 with fast code optimization and the following command line options

```
/Ox /Ot /GL /D "WIN32" /D " _DEBUG" /D " _CONSOLE" /D " _VC80_UPGRADE=0x0710" /Gm /EHsc /MTd /arch:SSE2 /fp:fast /Fo"Debug\"/Fd"Debug\vc90.pdb" /FR"Debug\" /W4 /nologo /c /Wp64 /Zi /TP /errorReport:prompt
```

The linker’s Stack Reserve Size and Stack Commit Size were set to 20M to allow for the processing of large number of long sequences consuming large memory space.

The code was optimized as follows. Initially the multithreaded application took 4.5 minutes on a standard PC to process a test set of protein sequences. This was reduced to 50 seconds with compiler options related to fast code generation and single-instruction multiple-data vectorization (SSE2 SIMDization) and further down to 2 seconds by limiting the pairs of characters (proteins) in the 2-character substring to those with gaps not exceeding 8 protein characters in step 2, and only matching protein pairs in both sequences if their resulting distance dist_{i,j} does not exceed 8. As \( \lambda \) is raised to the power dist_{i,j} and 0<\( \lambda \)<1, when dist_{i,j} exceeds 8, \( \lambda \text{dist}_{i,j} \) becomes negligible thus ignoring (i.e. skipping) amino acid pairs at such large interdistances saved a lot of computation time while not compromising the accuracy of the protein sequence matching. Fig. 2 plots the execution times (in seconds) of the original application binary (exe), the application binary after compiler optimizations, and the application binary after code and compiler optimizations, all with the short protein sequence file.

![Figure 2. Application Performance on Short Sequence Set](image)

Testing the application on a quad core x86 system with input files with many long protein sequences revealed that the execution time drops to 1.5 days on the quad core system versus a whole week on the single threaded implementation. This was achieved by the 16 thread version. Other versions with 8, 4, 2 threads consumed longer execution times on quad core and dual-core systems in comparison with the 16 thread version.

With exceeding long sequence files, the sequences were partitioned into 8 (different files and 8 different) runs executing concurrently on 8 independent quad core x86 systems cutting the execution time to 4.5 hours (compared to 1.5 days on a single quad core x86 computer, and a week on the single threaded version). Launching more than 16 parallel threads further resulted in shrinking speedup gains.

Fig. 3 plots the execution times of the optimized application on a dual-core laptop (Intel Core 2 Duo dual-core 2GHz processor, 1GB DRAM, Windows XP SP3), a quad-core computer (Intel Core 2 quad-core processor, 2.66GHz, 4GB DRAM, Windows Server 2003), and 8 such quad-core computers.

![Figure 3. Application Performance on Long Sequence Set](image)

Fig. 4 plots the ratio of execution time with 2 threads over the execution time with 4, 8 and 16 threads measured on the Core 2 Dual core 2GHz machine reflecting the
performance with increasing number of threads. Higher relative performance with 8 and 16 threads is obtained on the quad-core machine with more hardware resources, and in general on machines with more cores and memory. To help in validating the results, we also developed another application which computed the dot products of all pairs of matrix rows and compared the match results with the previously verified and known results obtained on other serial machines.

![Relative Performance of i Threads wrt 2 Threads](image)

Figure 4. Relative Performance with 2, 8, and 16 Threads with respect to Performance with 2 Threads on Dual Core Laptop

V. CONCLUSION

We introduced a protein-protein sequence matching application which we refer to as STRIKE. STRIKE was shown to improve upon the existing state-of-the-art methods for Protein-protein interaction prediction. We described the parallelization of STRIKE and its multithreaded implementation and performance enhancement, specific algorithm enhancements and compiler flag enhancements, on dual and quad-core Wintel systems. On large protein sequence sets, the execution time of a 16-thread implementation of this bioinformatics algorithm was reduced from about a week on a unithreaded implementation on a serial uniprocessor machine to 1.5 days on one quad core x86 machine, down to 4.5 hours on 8 such quad core machines. 16 threads provided good performance (compared to a lower number of threads) and a higher number of threads did not improve performance much beyond 16 threads. Our implementation was shown to scale very well with increasing data size and number of cores.

REFERENCES