SCALABLE PARALLEL METHODS FOR ANALYZING METAGENOMICS DATA AT EXTREME SCALE

By

JEFFREY ALAN DAILY

A dissertation submitted in partial fulfillment of
the requirements for the degree of

DOCTOR OF PHILOSOPHY

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

MAY 2015

To the Faculty of Washington State University:

The members of the Committee appointed to examine the dissertation of JEFFREY ALAN DAILY find it satisfactory and recommend that it be accepted.

Ananth Kalyanaraman, Ph.D., Chair

John Miller, Ph.D.

Carl Hauser, Ph.D.

Sriram Krishnamoorthy, Ph.D.

ACKNOWLEDGEMENT

First, I would like to thank Dr. Ananth Kalyanaraman for his supervision and support throughout the work. I would like to thank Dr. Abhinav Vishnu who encouraged my pursuit of my Ph.D. before anyone else, suggested Dr. Kalyanaraman as my advisor, and has been a valuable mentor and friend. I would like to thank Dr. Sriram Krishnamoorthy for his role on my graduate committee, for his research support at our employer, Pacific Northwest Northwest Laboratory (PNNL), and for his mentoring role while performing my research. I would like to thank Dr. John Miller and Dr. Carl Hauser for being on my graduate committee. Last, but not least, I would like to thank my employer, PNNL, for the tuition reimbursement benefit that assisted my pursuit of this degree.

SCALABLE PARALLEL METHODS FOR ANALYZING METAGENOMICS DATA AT EXTREME SCALE

Abstract

by Jeffrey Alan Daily, Ph.D.
Washington State University
May 2015

Chair: Ananth Kalyanaraman, Ph.D.

The field of bioinformatics and computational biology is currently experiencing a data revolution. The exciting prospect of making fundamental biological discoveries is fueling the rapid development and deployment of numerous cost-effective, high-throughput next-generation sequencing technologies. The result is that the DNA and protein sequence repositories are being bombarded with new sequence information. Databases are continuing to report a Moores law-like growth trajectory in their database sizes, roughly doubling every 18 months. In what seems to be a paradigm-shift, individual projects are now capable of generating billions of raw sequence data that need to be analyzed in the presence of already annotated sequence information.

While it is clear that data-driven methods, such as sequencing homology detection, are becoming the mainstay in the field of computational life sciences, the algorithmic advancements essential for implementing complex data analytics at scale have mostly lagged behind. Sequence homology detection is central to a number of bioinformatics applications including genome sequencing and protein family characterization. Given millions of sequences, the goal is to identify all pairs of sequences that are highly similar (or “homologous”) on the basis of alignment criteria. While there are optimal alignment algorithms to compute pairwise homology, their deployment for large-scale is currently not feasible; instead, heuristic methods are used at the expense of quality.

In this dissertation, we present the design and evaluation of a parallel implementation for conducting optimal homology detection on distributed memory supercomputers. Our approach uses a combination of techniques from asynchronous load balancing (viz. work stealing, dynamic task counters), data replication, and exact-matching filters to achieve homology detection at scale. Results for a collection of 2.56M sequences show parallel efficiencies of ∼75-100% on up to 8K cores, representing a time-to-solution of 33 seconds. We extend this work with a detailed analysis of single-node sequence alignment performance using the latest CPU vector instruction set extensions. Preliminary results reveal that current sequence alignment algorithms are unable to fully utilize widening vector registers.

TABLE OF CONTENTS

Page

CHAPTER

LIST OF TABLES

LIST OF FIGURES

Page

Dedication

To my wife, Nicole. I’m afraid I have no way to repay you.

And to my kids, Abigail, Grace, and Emmett — may you never remember my absence.

CHAPTER ONE

INTRODUCTION

The field of bioinformatics and computational biology is currently experiencing a data revolution. The exciting prospect of making fundamental biological discoveries is fueling the rapid development and deployment of numerous cost-effective, high-throughput next-generation sequencing (NGS) technologies that have cropped up in a span of three to four years (AppliedBio; HelicosBio; Illumina; PACBIO; Roche454). Touted as *next-generation* sequencing, to now "3^{rd} generation" technologies, these instruments are being aggressively adopted by large sequencing centers and small academic units alike.

The result is that the DNA and protein sequence repositories are being bombarded with both raw sequence information (or "reads") and processed sequence information (e.g., sequenced genomes, genes, annotated proteins). Traditional databases such as the NCBI GenBank (NCBI) and UniProt (Consortium, 2015) are continuing to report a Moore's law-like growth trajectory in their database sizes, roughly doubling every 18 months. Other projects such as the microbiome initiative (e.g., human microbiome, ocean microbiome) are contributing a significant volume of their own into metagenomic repositories (Sun et al., 2011; Markowitz et al., 2008). In what seems to be a paradigm-shift, individual projects are now capable of generating billions of raw sequence data that need to be analyzed in the presence of already annotated sequence information. Path-breaking endeavors such as personalized genomics (PersonalGenomics), cancer genome atlas (CancerGenomeAtlas), and the Earth Microbiome Project (Gilbert et al., 2010) foretell the continued explosive growth in genomics data and discovery.

While it is clear that data-driven methods are becoming the mainstay in the field of computational life sciences, the algorithmic advancements essential for implementing complex data analytics at scale have lagged behind (DOEKB; National Research Council Committee on Metagenomics and Functional, 2007). With a few notable exceptions in sequence search routines (Rognes, 2011; Lin et al., 2008; Farrar, 2007; Oehmen and Nieplocha, 2006; Darling et al., 2003) and phylogenetic tree construction (Ott et al., 2007), bioinformatics continues to be largely dominated by low-throughput, serial tools originally designed for desktop computing.

The method addressed in this dissertation is that of *sequence homology detection*. That is, given a set of N sequences, where N is large, detect all pairs of sequences that share a high degree of sequence homology as defined by a set of alignment criteria. The sequences are themselves typically short, a few hundred to few thousand characters in length. The method arises in the context of genome sequencing projects, where the goal is to reconstruct an unknown (target) genome by aligning the short DNA sequences (aka. "reads") originally sequenced from the target genome (Emrich et al., 2005). The expectation is for reads sequenced from the same genomic location to exhibit significant end-to-end overlap, which can be detected using sequence alignment computation. A similar use-case also arises in the context of transcriptomics studies (Wang et al., 2009) where the goals are to identify genes, and measure their level of activity (aka. expression) under various experimental conditions. A third, emerging use-case arises in the context of functionally characterizing metagenomics communities (Handelsman, 2004). Here, the goal is to identify protein families (Bateman et al., 2004; Tatusov et al., 1997) that are represented in a newly sequenced environmental microbial community (e.g., human gut, soil, ocean). This is achieved by first performing sequence homology detection on the set of predicted protein sequences (aka. Open Reading Frames (ORFs)) obtained from the community, and subsequently identifying groups of ORFs that are highly similar to one another (Yooseph et al., 2007; Wu and Kalyanaraman, 2008).

At its core, the *sequence homology detection* problem involves the computation of a large number of pairwise sequence alignment operations. A brute force computation of all $\binom{N}{2}$ pairs is not only infeasible but also generally not needed as with the sequence diversity expected in most practical inputs only a small fraction of pairs tend to survive the alignment test with a high quality alignment. The key is in identifying such a subset of pairs for pairwise alignment computation, using computationally less expensive means, without filtering out valid pairs. To this end, there are several effective filtering techniques using exact matching data structures (Altschul et al., 1990; Kalyanaraman et al., 2003).

After employing some of the most effective pair filters, several billions of pairwise alignments remain to be computed even for modest input sizes of $N \approx 10^6$. The most rigorous way of computing a pairwise alignment is to use optimality-guaranteeing dynamic programming algorithms such as Smith-Waterman (Gotoh, 1982; Smith and Waterman, 1981; Needleman and Wunsch, 1970). However, guaranteeing optimality is also computationally expensive — the algorithm takes $\mathcal{O}(m \times n)$ time for aligning two sequences of lengths m and n respectively. In the interest of saving time, current methods resort to faster, albeit approximation heuristic techniques such as BLAST (Altschul et al., 1990), FASTA (Pearson and Lipman, 1988), or USEARCH (Edgar, 2010). This has been the approach in nearly all the large scale genome and metagenome projects conducted over the last 4-5 years, ever since the adoption of NGS platforms.

On the other hand, several studies have shown the importance of deploying optimality-guaranteeing methods for ensuring high sensitivity (e.g., (Pearson, 1991; Shpaer et al., 1996)). For example, a recent study of an arbitrary collection of 320K ocean metagenomics amino acid sequences shows that a Smith-Waterman-based optimal alignment computation could detect 36% more homologous pairs than was possible using a BLAST-based run under similar parameter settings (Wu et al., 2012). Improving sensitivity of homology detection becomes particularly important when dealing with such environmental microbial datasets (National Research Council Committee on Metagenomics and Functional, 2007) due to the sparse nature of sampling in the input. For large-scale metagenomics initiatives, it is important to use optimal alignments. Otherwise, a lot of information is lost in addition to the already highly fragmented, sparse data.

1.1 Quantifying Scaling Requirements

In this dissertation, we evaluate the key question of *feasibility* of conducting a massive number of PSAs through the more rigorous optimality-guaranteeing dynamic programming methods at scale. To define feasibility, we compare the time taken to generate the data to the time taken to detect homology from it. Consider the following calculation: The Illumina/Solexa HiSeq 2500[1], which is one of the more popular sequencers today, can sequence $\sim 10^9$ reads in $\sim$11 days (Illumina). A brute-force all-against-all comparison would imply $\sim 10^{18}$ PSAs. Whereas, using an effective exact matching filter such as the suffix tree could provide 99.9% savings (based on our experiences (Wu et al., 2012; Kalyanaraman et al., 2003, 2006)). This would still leave $\sim 10^{15}$ PSAs to perform. Assuming a millisecond for every PSA, this implies a total of 277M CPU hours. To complete this scale of work in time comparable to that of data generation (11 days), we need the software to be running on 10^6 cores with close to 100% efficiency. This calculation yields a target of 10^9 PSAPS to achieve, where *PSAPS* is defined as the number of <u>P</u>airwise <u>S</u>equence <u>A</u>lignments <u>P</u>er <u>S</u>econd.

In addition to achieving large PSAPS counts, achieving fast turn-around times (in minutes) for small- to mid-size problems also becomes important in practice. This is true for use-cases — in which a new batch of sequences needs to be aligned against an already annotated set of sequences, or in analysis involving already processed information (e.g., using open reading frames from genome assemblies to incrementally characterize protein families) — where the number of PSAs required to be performed could be small (when

[1] While there are other faster technologies, we use Illumina as a representative example.

compared to that generated in *de novo* assembly) but needs to be performed multiple times due to the online/incremental nature of the application.

Some key challenges exist in the design of a scalable parallel algorithm that can meet the scale of 10^9 PSAPS or more. Even though the computation of individual PSAs are mutually independent, the high variance in sequence lengths and the variable rate at which those PSA tasks are identified using an exact matching filter can result in load imbalance. In addition, the construction of the exact matching filter (such as the suffix tree) and the use of it to generate pairs for PSA computation on-the-fly need to be done in tandem with task processing (PSA computation), in order to reduce the memory footprint[2].

1.2 Contributions

In this dissertation, we present the design of a scalable parallel framework that can achieve orders of magnitude higher PSAPS performance than any contemporary software. Our approach uses a combination of techniques from asynchronous load balancing (viz. work stealing and dynamic task counters), remote memory access using PGAS, data replication, and exact matching filters using the suffix tree data structure (Weiner, 1973) in order to achieve homology detection at scale. Several factors distinguish our method from other work: i) We choose the all-against-all model as it finds a general applicability in most of the large-scale genome and metagenome sequencing initiatives, occupying an upstream phase in numerous sequence analysis workflows; ii) To ensure high quality of the output, each PSA is evaluated using the *optimality-guaranteeing* Smith-Waterman algorithm (Smith and Waterman, 1981) (as opposed to the traditional use of faster sub-optimal heuristics such as BLAST); iii) We use protein/putative open reading frame inputs from real world datasets to capture a more challenging use-case where a skewed distribution in sequence lengths can cause nonuniformity in PSA tasks; and iv) To the best of our knowledge, this effort

[2] Note that it is not reasonable to assume that all of the generated pairs from the filter can be computed and stored prior to PSA calculations.

represents the first use of work stealing with suffix tree filters.

The key contributions are as follows:

1. A comprehensive solution to scalable optimal homology detection at the largest reported scale of 8K cores.

2. A new, scalable implementation of constructing string indices at large scale.

3. To the best of our knowledge, this is the first work in this domain to use distributed memory work stealing for dynamic load balancing.

4. Analysis of the homology detection problem on emerging architectures.

This dissertation is organized as follows: Chapter 2 presents the sequence homology problem in more detail and addresses the current state of computational solutions for the problem of homology detection. Chapter 3 presents the overall system architecture of our solution. Chapter 4 describes and experimentally evaluates our parallel algorithm. Extensions to this work appear in Chapter 5. Key findings and future lines of research are outlined in Chapter 6.

CHAPTER TWO

BACKGROUND AND RELATED WORK

This chapter briefly introduces the reader to the notation, data structures, and algorithms used in homology detection. In addition, the foundational work in this area is highlighted to motivate the contributions in Chapter 3.

2.1 Algorithms and Data Structures for Homology Detection

The protein homology detection problem is concerned with finding pairs of similar sequences given a set N of sequences. Similar sequences can be identified by performing an alignment. An exhaustive, brute-force evaluation of all $\binom{N}{2}$ pair combinations of input sequences is not feasible given the large values of N expected in practice. As a result, filters need to be used to identify only a subset of pairs for which alignment computation is likely to produce satisfactory results. Filters often employ one of a few exact matching string indices. This section first introduces the reader to the notation used in sequence alignment and exact matching string indices.

2.1.1 Notation

Let Σ denote an alphabet, e.g., $\Sigma = \{a, c, g, t\}$ for DNA, implying $|\Sigma| = 4$ for DNA, whereas $|\Sigma| = 20$ for amino acids. An input string s of length $n + 1$ is a sequence $s = c_0c_1 \ldots c_{n-1}\$$, where $c_i \in \Sigma, 0 \leq i \leq n-1$ and $\$ \notin \Sigma$; \$ is the end-of-string terminal character. The i^{th} character of s is referred to as $s[i]$. A prefix of s is a sequence $prefix(s, i) = s[0..i] = c_0c_1 \ldots c_i; (0 \leq i \leq n, c_n = \$)$ which may include the terminal character. A suffix of s is a sequence $suffix(s, i) = s[i..n] = c_ic_{i+1} \ldots c_{n-1}\$; (0 \leq i \leq n-1)$ which always includes the terminal character. We also consider prefixes of suffixes of s, commonly called a substring of s, as $S - prefix(s, i, j) = s[i..i + j - 1] = c_ic_{i+1} \ldots c_{i+j-1}$, where j indicates the length of the S-prefix starting at index i. The unique terminal symbol

\$ ensures that no suffix is a proper S-prefix of any other suffix. As convenient, we will use the terms "strings" and "sequences" interchangeably.

2.1.2 Sequence Alignment

An *alignment* between two sequences is an order-preserving way to map characters in one sequence to characters or gaps in the other sequence. There are many models for computing alignments — the most common models are *global alignment* (Needleman and Wunsch, 1970) where all characters from both sequences need to be involved, *semi-global alignment* where the aligning portions may ignore characters at the beginning or end of a sequence, and *local alignment* (Smith and Waterman, 1981; Gotoh, 1982) where the aligning portions can be restricted to a pair of substrings from the two sequences. An alignment is scored based on the number of character substitutions (matches or mismatches) and the number of characters aligned with gaps (insertions or deletions). For DNA sequences, positive scores are given to matches and negative scores to penalize gaps and mismatches. For protein/amino acid sequences, scoring is typically based on a predefined table called a "substitution matrix" which scores each possible $|\Sigma| \times |\Sigma|$ combination (Dayhoff et al., 1978; Henikoff and Henikoff, 1992). An *optimal alignment* is one which maximizes the alignment score.

Sequence alignments are computed using dynamic programming because it is guaranteed to find an optimal alignment given a particular scoring function. Regardless of the class of alignment being computed, a dynamic programming recurrence of the following form is computed. Given two sequences $s_1[1 \ldots m]$ and $s_2[1 \ldots n]$, three recurrences are defined for aligning the prefixes $s_1[1 \ldots i]$ and $s_2[1 \ldots j]$ as follows (Gotoh, 1982): Let $S_{i,j}$ denote the optimal score for aligning the prefixes such that the alignment ends by substituting $s_1[i]$ with $s_2[j]$. $D_{i,j}$ denotes the optimal score for aligning the same two prefixes such that the alignment ends in a deletion, i.e., aligning $s_1[i]$ with a gap character. Similarly, $I_{i,j}$ denotes the optimal score for aligning the prefixes such that the alignment ends in an insertion, i.e., aligning $s_2[j]$ with a gap character. Given the above three ways to end an alignment, the optimal score for aligning the prefixes corresponding to the subproblem $\{i, j\}$ is given by:

$$T_{i,j} = max(S_{i,j}, D_{i,j}, I_{i,j}) \tag{2.1}$$

The dependencies for the individual dynamic programming recurrences are as follows: $S_{i,j}$ derives its value from the solution computed for the subproblem $\{i-1, j-1\}$, while $D_{i,j}$ and $I_{i,j}$ derive their values from the solutions computed for subproblems $\{i-1, j\}$ and $\{i, j-1\}$, respectively.

A typical implementation of this dynamic programming algorithm builds a table of size $\mathcal{O}(m \times n)$ with the characters of each sequence laid out along one of the two dimensions. According to common practice, we call the sequence with characters along the rows i of the table the "query" sequence and the sequence with characters along the columns j of the table the "database" sequence. Each cell (i, j) in the table stores three values $S_{i,j}$, $D_{i,j}$, and $I_{i,j}$, corresponding to the subproblem $\{i, j\}$. Given the dependencies of the entries at a cell, the dynamic programming algorithms for all three sequence alignment classes can be represented using the pseudocode outlined in Algorithm 1. The algorithm has a time complexity of $\mathcal{O}(mn)$.

The three classes of sequence alignment initialize the first row and column differently (lines 1 and 2 in Algorithm 1). SW and SG alignments initialize the first row and column of the table to zero, while NW alignments initialize the first row and column based on the gap function. The table values for SW alignments are not allowed to become negative, while NW and SG allow for negative scores. An optional post-processing step retraces an optimal

Algorithm 1 Dynamic Programming Algorithm

```
Align($s_1[1 \dots m], s_2[1 \dots n]$)
Initialize the first row of the dynamic programming table
Initialize the first column of the dynamic programming table
for $i$: 1 to $m$ do
    for $j$: 1 to $n$ do
        $S_{i,j} \leftarrow T_{i-1,j-1} + W(i,j)$.
        $D_{i,j} \leftarrow \max(D_{i-1,j}, T_{i-1,j} + G_{\text{open}}) + G_{\text{ext}}$.
        $I_{i,j} \leftarrow \max( I_{i,j-1}, T_{i,j-1} + G_{\text{open}}) + G_{\text{ext}}$.
        $T_{i,j} \leftarrow \max( S_{i,j}, D_{i,j}, I_{i,j})$.
```

alignment and can be completed in $\mathcal{O}(m+n)$ time assuming the entire table is stored. Details of that step are omitted.

Due to the computational complexity of the dynamic programming approaches, faster, heuristic methods such as BLAST (Altschul et al., 1990), FASTA (Pearson and Lipman, 1988), or USEARCH (Edgar, 2010) were developed as an alternative. By using heuristics, these tools run faster than the optimal methods, however they run the risk of producing sub-optimal alignments (Pearson, 1991; Shpaer et al., 1996).

2.1.3 Sequence Homology

The sequence homology detection problem is as follows: Given a sequence set $S = \{s_1, s_2, \dots s_n\}$, identify all pairs of sequences that are "homologous". There are several ways to define homology depending on the type of sequence data and the intended use-case. Since for this dissertation, we deal with protein/amino acid sequences, we use the following definition consistent with some of the previous work in the area (Yooseph et al., 2007; Wu and Kalyanaraman, 2008; Wu et al., 2012): Two sequences s_1 and s_2 of lengths n_1 and n_2, respectively, are *homologous* if they share a local alignment whose score is at least τ_1% of the ideal score (with n_1 matches), and the alignment covers at least τ_2% of n_2 characters. The above is assuming $n_1 \leq n_2$ w.l.o.g. The parameters τ_1 and τ_2 are user-specified, with defaults for protein sequences set as $\tau_1 = 40\%$ and $\tau_2 = 80\%$ (Wu et al., 2012). Note that for DNA sequences, these cutoffs typically tend to be higher as more similarity is expected at the nucleotide level. The lower cutoffs used in protein sequences make the homology detection process more time consuming because more pairs of sequences typically need to be evaluated. Many methods that happen to use even fast alignment heuristics such as USEARCH (Edgar, 2010) and CD-HIT (Li and Godzik, 2006) do *not* even allow specifying such lower settings due to computational constraints. If one were to deploy dynamic programming methods to evaluate alignments, an optimal alignment will be computed regardless of the specified cutoff thus making the solution more generic. The key lies in scaling the number of alignments computed to the extent that evaluation of the identified pairs becomes feasible. However, to the best of our knowledge, no such parallel implementations exist. Consequently, all the genome and metagenome scale projects so far have resorted to BLAST-like heuristics to compute homology.

2.1.4 String Indices for Exact Matching

An exhaustive, brute-force evaluation of all $\binom{n}{2}$ pair combinations is not feasible given the large values of n expected in practice (even if alignment heuristics are to be used). As a result, filters need to used to identify only a subset of pairs for which alignment computation is likely to produce satisfactory results (as per the pre-defined cutoffs). A popular filtering data structure is that of the look-up table (Aluru and Ko, 2005), which is also internally used in numerous programs that are variants of BLAST and FASTA (Pearson and Lipman, 1988; Altschul et al., 1997; Li and Godzik, 2006; Edgar, 2010). While it is easy to construct and process this data structure, its use is restricted to identifying short, fixed-length exact matches between pairs of sequences. This is owing to its space complexity, which is exponential in the length of the exact match sought after — more specifically, $\mathcal{O}(|\Sigma|^k)$ where k is the length of the exact match. Furthermore, a smaller value of k (typically, 3 or 4 used in practice) significantly increases the number of pairwise sequence alignments

(PSAs), as more pairs of sequences are likely to share a shorter exact match by random chance. As an alternative to the look-up table, the use of suffix trees[1](Weiner, 1973) overcomes these limitations as its space complexity is linear in the input size and it has the ability to allow detection of arbitrarily long exact matches in constant time per matching pair (Kalyanaraman et al., 2003).

A suffix tree Γ is a trie that indexes all suffixes of s. See Figure 2.1 for an example. For an input of length n, there are $n + 1$ leaves in the tree. For any leaf v_i, the concatenation of the edge labels on the path from the root to v_i spells out suffix(s, i). Each internal node other than the root has at least two children and each edge is labeled with an S-prefix of s. No two edges out of a node can have edge labels starting with the same symbol. Storing edge labels as strings requires $\mathcal{O}(n^2)$ space for the tree, so typically they are stored as two integers representing the starting and ending index of the substring in s which brings the space requirement down to $\mathcal{O}(n)$.

The suffix tree can be divided into a set of sub-trees; Γ_α denotes the sub-tree that indexes suffixes sharing a prefix α. In Figure 2.1 for example, the tree could be divided into $\Gamma_\$, \Gamma_i, \Gamma_p, \Gamma_s$. These example sub-trees correspond to the exact match cutoff $k = 1$ of a look-up table as mentioned above. Further, Γ_{ssi} and Γ_{issi} are a few examples of sub-trees rooted at depth/cutoff $k = 3$ and $k = 4$, respectively.[2] Recall that the suffix tree can be constructed in linear space compared to the exponential space of the look-up table even though any one bin of the look-up table corresponds to a sub-tree.

The edges emanating from each internal node in Figure 2.1 are sorted lexicographically.

[1] Since we have multiple sequences as input, the appropriate data structure here is the "generalized suffix tree", which is nothing but a unified suffix tree corresponding to all suffixes of all the input sequences; however, for convenience, we simply use the term suffix tree in this dissertation.

[2] The examples all choose subtrees rooted at an internal node. However, it is possible to have subtrees rooted at a location where there is no naturally occurring internal node, such as at Γ_m where suffix$(s, 0)$ would reside.

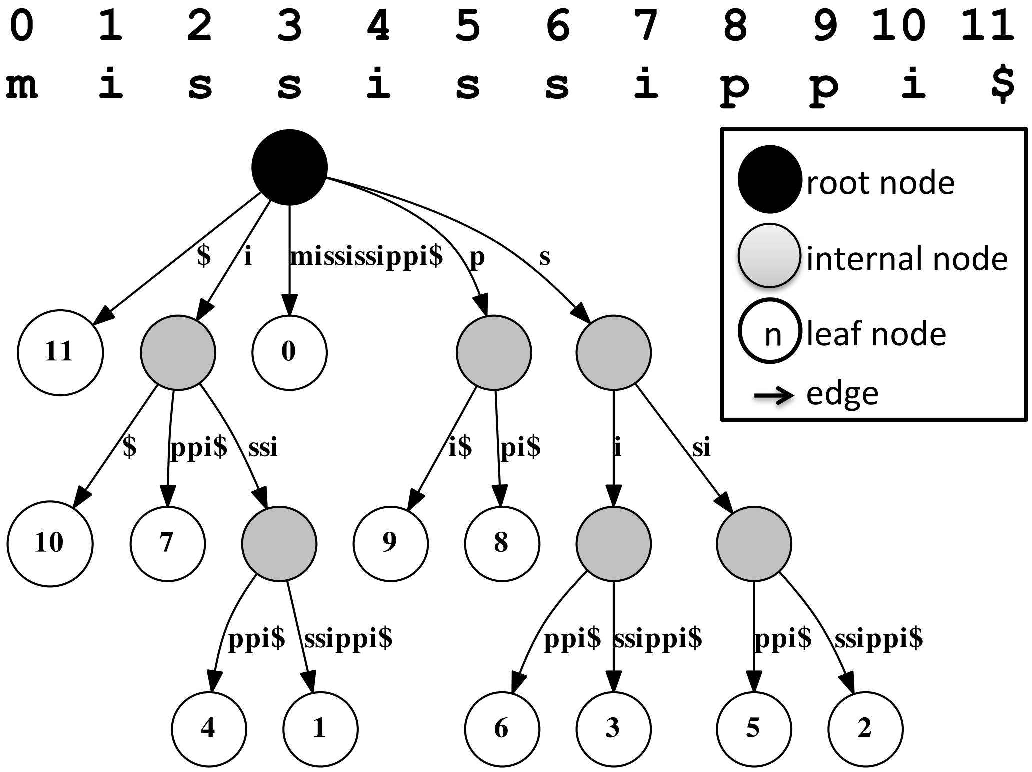


Figure 2.1: Input string $s = mississippi\$$ and corresponding suffix tree Γ.

A depth-first traversal of the sorted tree results in the related suffix array (SA) data structure with the suffixes listed in lexicographical order as shown in Table 2.1. Suffix arrays are not typically useful by themselves, instead they are often accompanied with a longest common prefix (LCP) array indicating the length of the common prefix of two adjacent suffixes. The LCP array is also indicated in Table 2.1. In addition to the LCP array, some algorithms using suffix arrays also require the Burrows-Wheeler Transform (BWT) array which indicates the character immediately preceding the suffix. All together, the SA, LCP, and BWT arrays are called the enhanced suffix array. All algorithms that can be performed on a suffix tree can be equivalently performed on an enhanced suffix array (Abouelhoda et al., 2004).

Table 2.1: Enhanced Suffix Array for Input String $s = mississippi\$$.

i	s[i]	SA[i]	LCP[i]	BWT[i]	suffix
0	m	11	0	i	$
1	i	10	0	p	i$
2	s	7	1	s	ippi$
3	s	4	1	s	issippi$
4	i	1	4	m	ississippi$
5	s	0	0	$	mississippi$
6	s	9	0	p	pi$
7	i	8	1	i	ppi$
8	p	6	0	s	sippi$
9	p	3	2	s	sissippi$
10	i	5	1	i	ssippi$
11	$	2	3	i	ssissippi$

Since we are concerned with the case of multiple sequences as input, the appropriate data structures here are actually the “generalized suffix tree” and the related “generalized suffix array”. A generalized suffix tree is simply the suffix tree of the n concatenated input sequences such that a unique terminal character $ separates each input sequence. If two identical suffixes exist, they share a common internal node with at least two leaf nodes, one for each uniquely terminated suffix. With respect to generalized suffix arrays, the two identical suffixes would be adjacent to each other (SA[i] and SA[i+1]) and the LCP value between them would be equivalent to their length.

2.2 Parallelization of Homology Detection

Parallelizing homology detection focuses on the principle operation, the pairwise sequence alignment. Pairwise alignments are naturally parallelizable; there are no data dependencies between any two pairwise alignments. As many alignments can be performed as there are number of processing elements, e.g., threads, to perform them. Alignments of long sequences can be accelerated by applying additional processing elements and breaking the problem into smaller pieces and managing the data dependencies between them. We focus here on the relatively short protein sequences that are prevalent in metagenomics studies. There are then two problems to address, accelerating the singular pairwise alignment and load balancing many alignment tasks across computational resources.

2.2.1 Vectorization Opportunities in Sequence Alignment

There have been numerous efforts to parallelize optimal sequence alignments using GPUs (Sarkar et al., 2010), Xeon Phi accelerators (Liu and Schmidt, 2014; Wang et al., 2014), or CPU vector instructions (Wozniak, 1997; Rognes and Seeberg, 2000; Farrar, 2007; Rognes, 2011). However, not all of these approaches necessarily address the same bioinformatics application. For example, database search may group database sequences to improve performance (Rognes, 2011), while protein homology graph applications may prohibit such optimizations (Daily et al., 2014). That said, pairwise sequence alignments generally fall into two categories: inter-task and intra-task. Aligning numerous independent pairs of sequences represents a case of inter-task parallelism, while intra-task parallelism describes the alignment of a single (query) sequence against a set of (database) sequences (Rognes, 2011). We focus here on the more generally applicable inter-task pairwise alignments.

Figure 2.2 enumerates the ways to vectorize sequence alignment. Each approach operates in a series of *vector epochs*, where each vector epoch signifies a timestep during execution when all processing elements (p) of the vector processor are concurrently working on different parts of computation, contributing to the calculation of different cells in the dynamic programming table.

In the *Blocked* approach (Figure 2.2 (Blocked)), proposed by (Rognes and Seeberg, 2000), a vector epoch spans a subset of p contiguous cells along the dimension of the query sequence (i.e., columns). Each vector initially ignores the contributions of the upward cells. After computing a block, the new cell values are checked for correctness and potentially recomputed. Once the values converge, the last value of the current vector is used by the

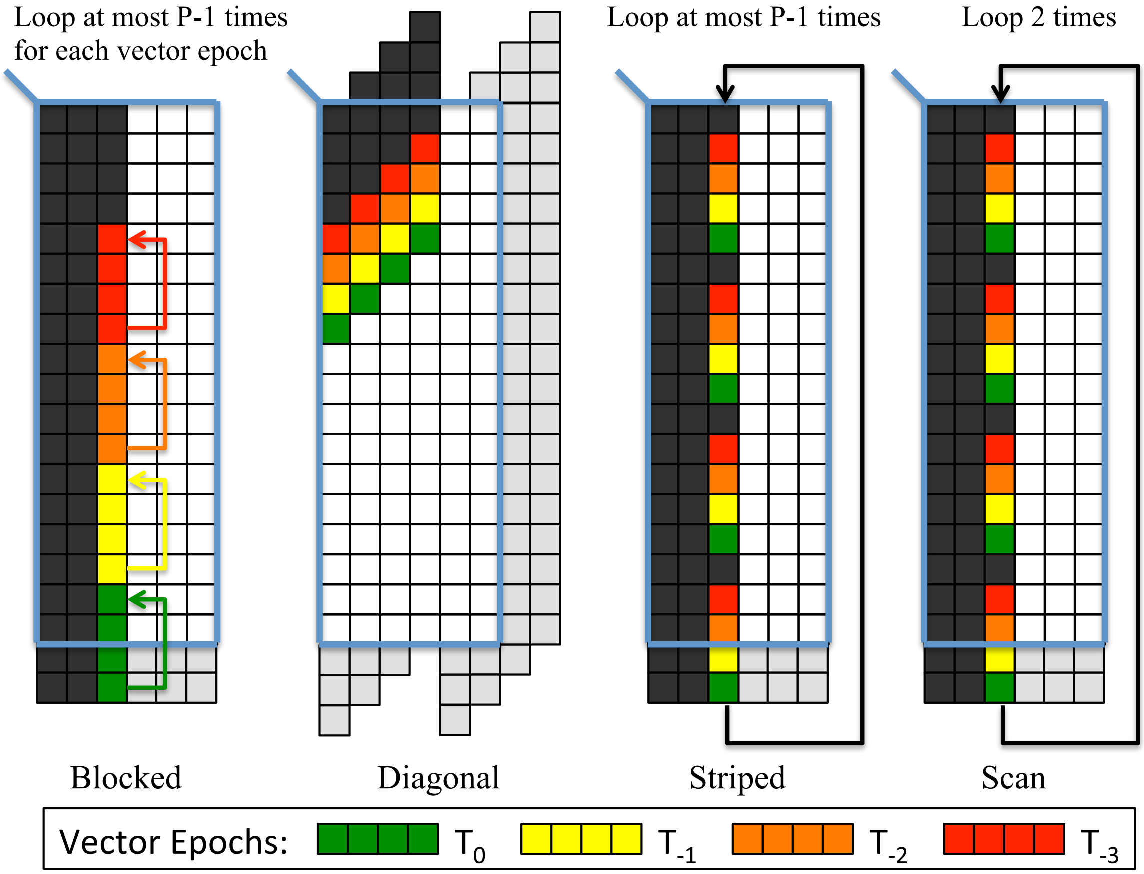


Figure 2.2: Known ways to vectorize Smith-Waterman alignments using vectors with four elements. The tables shown here represent a query sequence of length 18 against a database sequence of length 6. Alignment tables are shown with colored elements, indicating the most recently computed cells. In order of most recently computed to least recently, the order is green, yellow, orange, and red. Dark gray cells were computed more than four vector epochs ago. Light gray cells indicate padded cells, which are required to properly align the computation(s) but are otherwise ignored or discarded. The blue lines indicate the relevant portion of the tables with the table origin in the upper left corner. (Blocked) Vectors run parallel to the the query sequence. Each vector may need to recompute until values converge. First described by Rognes and Seeberg (Rognes and Seeberg, 2000). (Diagonal) Vectors run parallel to the anti-diagonal. Fist described by Wozniak (Wozniak, 1997). (Striped) Vectors run parallel to the query using a striped data layout. A column may need to be recomputed at most $P - 1$ times until the values converge. First described by Farrar (Farrar, 2007). (Scan) This is the approach taken in this dissertation. It is similar to (Striped) but requires exactly two iterations over a column.

next vector. The drawback of the Blocked approach is that the data dependencies both within and between vectors limit the overall performance.

In the *Diagonal* approach (Figure 2.2 (Diagonal)), an epoch spans a subset of p contiguous cells along a single diagonal of the table (Wozniak, 1997). Note that the cells along the same diagonal have no interdependencies as their dependent values come from the cells in the previous two diagonals. However, wasteful computation is caused in this approach by padding the table with cells to properly align the computation. Another disadvantage is the irregular memory access along the diagonal.

In the *Striped* approach (Figure 2.2 (Striped)), proposed by Farrar (Farrar, 2007), a vector epoch spans a subset of p evenly spaced cells along the dimension of the query sequence. This scheme eliminates the data dependencies both within and between vectors by striping the vector parallel to the query sequence. Similar to Blocked, this approach also initially ignores the contributions of the upward values and makes additional passes over each column until the values converge. Often, the values converge before having to compute the column entirely a second time. This significantly improves overall performance. That said, in the worst case, the column would be recomputed as many times as there are elements in the vectors.

Lastly, our solution leverages the striped layout, but it uses a prefix scan formulation of the dynamic programming recurrence (Khajeh-Saeed et al., 2010). The prefix scan recurrence is straightforward though it was initially designed for GPUs and requires a lengthy proof to confirm its equivalence to the original problem. Compared to Blocked and Striped, which initially ignore the upward cells, the prefix scan calculates a temporary value and later uses the temporary value to find the final cell value. As shown in Figure 2.2 (Scan), our solution requires exactly two iterations over each column.

All four vectorization schemes can be summarized using the generic pseudocodes in Algorithm 2 and Algorithm 3, with Blocked, Striped, and Scan mapping to Algorithm 2

Algorithm 2 A generic pseudocode for a column-wise vectorized sequence alignment

```
Align($s_1[1 \dots m], s_2[1 \dots n]$)
    for each character in database sequence do
        for each vector epoch in column do
            Load substitution scores from query profile.
            Load previous column's corresponding cell values.
            Compute next cell values.
```

Algorithm 3 A generic pseudocode for a diagonal vectorized sequence alignment

```
Align($s_1[1 \dots m], s_2[1 \dots n]$)
    for every $p$ characters in database sequence do
        for each vector epoch in diagonal do
            Gather substitution scores for each $s_1[i], s_2[j]$ pair.
            Use previous vector epoch directly.
            Compute next cell values.
```

and Diagonal to Algorithm 3.

2.2.2 Distributed Alignments

The first attempts at large-scale parallel homology detection can be attributed to mpiBLAST (Darling et al., 2003) and ScalaBLAST (Oehmen and Nieplocha, 2006). These initial approaches were extensions of the original BLAST (Altschul et al., 1990) heuristic approach, distributing and load balancing the database search across a cluster. Thereafter, updated versions of mpiBLAST appear for each new high performance computing system (Thorsen et al., 2007; Lin et al., 2008, 2011).

There are some design challenges presented by the use of suffix trees for homology detection. Firstly, constructing suffix trees on massively parallel distributed memory machines is nontrivial, owing to the inherent irregularity of the underlying data access patterns (Kalyanaraman et al., 2003; Ghoting and Makarychev, 2009; Mansour et al., 2011). Secondly, although the data structure has a linear space complexity, the constant of proportionality is high, typically around 40-50. Therefore, the data structure needs to be generated and stored in a distributed manner in order for scalability. Thirdly, despite the high selectivity of pairs, the number of pairs identified could still be in several billions or more for modest sized inputs containing millions of sequences, precluding the possibility of storing them before processing them for alignment.

The algorithm presented in this dissertation improves on previous efforts (Wu et al., 2012; Daily et al., 2012) and tackles the challenges outlined above through the use of work stealing and task counters. Wu et al. (Wu et al., 2012) use a hierarchy of master and worker processes on a compute cluster to balance the load of generating pairs from a precomputed, out-of-core sequence filter while concurrently aligning the generated pairs. They report scaling up to 2K processors. Daily et al. (Daily et al., 2012) were the first to apply a work stealing technique to scale sequence homology to over 100K processors but did so by simulating an arbitrary filter which did not introduce compute overhead or load imbalance, thus their work focused primarily on the work stealing of the brute force $\binom{N}{2}$ set of sequence pairs. This work represents the first comprehensive solution to scalable optimal homology detection given an input set of sequences; nothing is computed beforehand and no portions of the pipeline are simulated. The pipeline applies work stealing to the creation and processing of the suffix tree filter concurrently with the pair alignments. Lastly, while there are numerous solutions available for hardware acceleration of individual PSA computations on various specialized multicore platforms such as GPUs, FPGAs, etc. (reviewed in (Sarkar et al., 2010)), the implementation presented in this dissertation does not incorporate those (future work).

CHAPTER THREE

SCALABLE PARALLEL METHODS

In this chapter, we explore many ways of solving the problem of optimal homology detection. We first attempt to reduce the task space using known filtering techniques. Then we propose a solution to the load balancing issue caused by using the filters in addition to the load imbalance inherent to the problem.

3.1 Filtering Sequence Pairs

As noted in Chapter 2, exact matching filters need to be used in practice to reduce the task space from $\binom{n}{2}$ PSAs. One of the most effective filters designed to date is the suffix tree filter used by (Wu et al., 2012); however the search for better filters is an open area of research. We describe the suffix tree filter as well as an alternative length-based filter in the following sections. Key notation used in this dissertation is summarized in Table 3.1.

3.1.1 Suffix Tree Filter

Using suffix trees to identify "promising" sequence pairs for alignment computation is detailed in (Wu et al., 2012). We improve upon their work by not precomputing and storing the suffix trees to disk, and instead generate the suffix tree on-the-fly and use it to identify promising pairs when different subtrees of the suffix tree become available.

To build the suffix tree in parallel, we independently construct subtrees of the suffix tree. We first partition all suffixes of the input sequences into $|\Sigma|^k$ "buckets" based on their first k characters, where k is a short, fixed-length parameter e.g., 5 for amino acid sequences. We represent a suffix as a 3-tuple of the sequence index, the offset from the start of the sequence, and the bucket index. The reason for the sequence index and offset are clear, however our choice of associating the bucket index with each suffix was for memory

Notation	Description
Σ	Alphabet for sequences, e.g., $\|\Sigma\| = 4$ for DNA, $\|\Sigma\| = 20$ for amino acids (proteins).
s	A sequence of length $n + 1$, $s = c_0c_1 \dots c_{n-1}\$$, where $c_i \in \Sigma, 0 \leq i \leq n - 1$ and $\$ \notin \Sigma$.
n	Length of a sequence.
S	Set of sequences $S = s_1, s_2, \dots s_N$.
N	Number of sequences.
$s[i]$	i^{th} character of s.
$	End-of-string terminal character.
$prefix(s, i)$	Prefix of sequence s from 0 through character at position i.
$suffix(s, i)$	Suffix of sequence s from i through and including the terminal character.
τ_1, τ_2	Homology threshold heuristics. Alignments must be at least τ_1 of the ideal score and cover at leaset τ_2 of the characters.
p	Number of processing elements.
Γ	Suffix tree.
Γ_α	Suffix subtree where all suffixes share a prefix α.
k	Cut depth for forest of generalized suffix subtrees.
ψ	Exact-match length cutoff. All sequence pairs generated from the generalized suffix tree filter contain an exact match $\geq \psi$.

Table 3.1: Notation used in the dissertation.

considerations as well as for ease of implementation. With respect to memory, the number of suffixes (3-tuples) depends on the size of the input sequences, whereas the number of buckets depends on $|\Sigma|^k$ which grows quickly as either k or $|\Sigma|$ becomes large. Our implementation allows for much larger k than would normally be allowed given memory constraints. With respect to ease of implementation, we can store the suffixes as a contiguous array instead of using a sparse representation of the buckets. This contiguity enables easy sorting of the suffixes as well as the direct exchange of the suffixes subtrees when load balancing.

Once the buckets are constructed, by definition of the suffix tree, each such bucket contains suffixes that fall into a distinct subtree rooted at a depth of k of the tree. The idea is to subsequently process all buckets in parallel so that the individual subtrees corresponding to buckets can be constructed in an independent manner. A challenge here is that the size of each bucket is not necessarily uniform as it is input dependent, and the amount of work is proportional to the number of suffixes contained in the tree. Consequently, one option is to statically partition the buckets onto each process in an attempt to balance the total number of suffixes to be handled on each process. However, this would require global knowledge as to the size of each bucket, and if $|\Sigma|^k$ is large this approach is not feasible. As an alternative, we partition the buckets based on the bucket index modulo the number of processes, then we apply work stealing to further load balance this problem. The initial static distribution of the buckets is a simple calculation. In addition, since adjacent buckets, e.g., “AAB”, “AAC” where $k = 3$, tend to be similar in size when they share a common prefix (here “AA”), the initial distribution keeps adjacent buckets from being stored on the same process in case their shared prefix occurs frequently. Lastly, each subtree requires a variable amount of suffixes to be present in memory, along with their corresponding sequences, before processing begins. This may increase the amount of communication in our implementation, especially when sequences are not stored locally. Non-local sequences are always fetched as needed, which works well for aligning two sequences with at most two fetches, but in the case of suffix subtree processing which may require many fetches, we cache non-local sequences until the subtree processing is complete. Caches are not shared between processes and are discarded once the subtree is no longer being processed.

The suffix subtrees are themselves constructed in a depth-first manner by recursively bucketing the set of suffixes at increasing node depths. If r denotes the number of suffixes in a given subtree, and l_r denotes the mean length of those suffixes, then the time complexity to build the subtree using our recursive method is $\mathcal{O}(rl_r^2)$. We do not construct or use any auxiliary suffix tree data structures such as suffix links. A depth-first traversal of the constructed subtree generates the promising pairs as described in (Kalyanaraman et al., 2003), which detects and reports all pairs that share a maximal match of a minimum length. For our purpose, we generate the tree as a forest of disjoint subtrees emerging at a specified cut depth $k \leq \psi$. The pair generation algorithm detects each pair corresponding to a maximal match of length at least ψ in constant time (Kalyanaraman et al., 2003). However, if two sequences contain more than one such maximal match between them it is possible that the pair is generated multiple times from different parts of the generalized suffix tree (i.e., leading to duplicates). The individual subtrees can be independently traversed in parallel to generate pairs.

3.1.2 Length Based Cutoff Filter

The suffix tree filter, although generally effective in terms of reducing the number of alignments to perform, takes a non-negligible time to create and process the suffix subtrees. One way to achieve further savings in the number of PSAs performed, without impacting the final output, is as follows: we can rule out pairs based upon the length of the two sequences involved in the potential alignment. As a user-supplied heuristic, if the two sequences could not possibly produce a positive optimal score because the sequences differ too greatly in length, or if one of the sequence lengths is less than the minimal length cutoff, the pair is discarded. This length-based filter calculation is in fact used by the suffix tree filter as an additional filter after it has identified a promising pair using the tree alone. We explore the merit of using the length based filter on its own in Chapter 4.

3.1.3 Storing Large Numbers of Tasks

Using work stealing as in (Daily et al., 2012) required the tasks to be explicitly enumerated and stored for a total of $\binom{n}{2}$ tasks stored across P processes. The largest dataset we explored was 2.56M sequences which resulted in approximately 3.28 trillion tasks. The tasks were stored as two 8-byte integer sequence identifiers. This could be reduced to a single 8-byte integer using a combinatorial number system of degree 2, but even so this would require nearly 24TB of aggregate memory or at minimum nearly 800 compute nodes with 32GB of usable memory each. This solution of computing and storing the enumerated pairs does not scale with respect to memory constraints, even if we are able to filter out pairs – eventually larger datasets will produce enough pairs to invalidate this approach.

An alternative is to dynamically generate and process the pairs using a dynamic load balancing scheme. The strategy in (Wu et al., 2012) was to use a hierarchy of masters and workers in such a way to handle pairs being generated faster than they could be consumed. We use a similar strategy but apply it using work stealing, dynamically creating new work to be consumed as suffix trees are processed.

3.1.4 Storing Large Sequence Datasets

A significant challenge in the design of parallel homology detection is the management of the sequence data. The strategy in (Daily et al., 2012) was to store the sequence database once per compute node rather than once per worker process. In a hybrid MPI+*pthread* model this is accomplished by running one MPI process per compute node to hold the sequences and then using *pthreads* to access the read-only sequence database. In a standard

MPI model, the sequences can be stored in shared memory.

By storing the entire sequence database per compute node, the authors did not address memory constraints such that the sequences would not fit within a single compute node. This is a problem as the database sizes continue to grow faster than the amount of memory per node. Our solution to this problem relies on a PGAS model rather than a shared-nothing MPI model or a hybrid MPI+*pthread* model. The PGAS model provides a shared memory interface to the sequence database while transparently distributing the sequences across compute nodes.

Using the PGAS model, the aggregate memory of multiple compute nodes is available with the trade-off of having to communicate sequences that are no longer local. We reduce the chances of having non-local sequences by replicating the sequence database once per subset of nodes such that each subset of nodes has enough aggregate memory to store the complete sequence dataset. As an improvement over (Wu et al., 2012), non-local sequences are communicated using one-sided operations rather than periodic collective communications or the alternative of using non-blocking two-sided operations which would require explicit progress. We use an efficient one-sided communication library (Vishnu et al., 2012) which performs better than the one-sided primitives of the MPI-2 standard, making this a viable implementation strategy.

3.2 Load Imbalance

There is significant incidence of load imbalance throughout this problem. We look at the causes and solutions in detail next.

3.2.1 Load Imbalance Caused by Filters

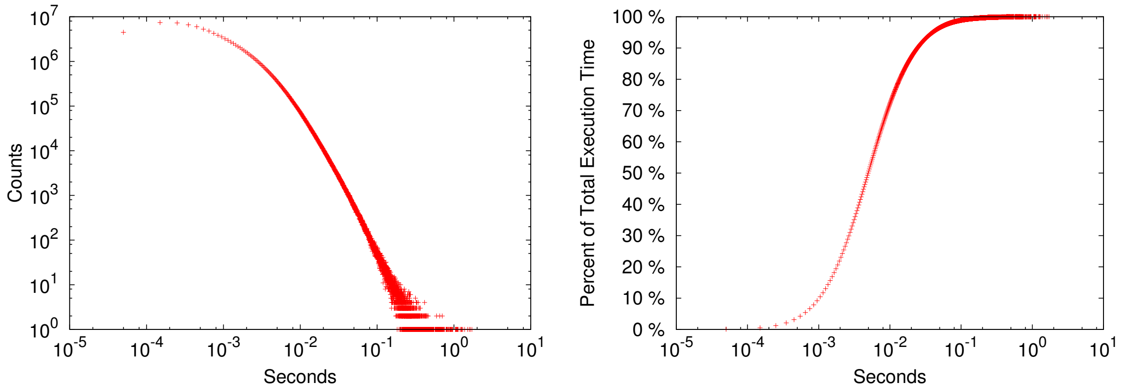
For the suffix tree filter, each suffix is placed in a bucket based on its first k characters resulting in at most $|\Sigma|^k$ buckets. Each bucket is processed to yield a distinct subtree of the suffix tree, which is subsequently processed to generate sequence promising pairs. k must be sufficiently large to create enough work to distribute. Subtree creation is linearly proportional to the sum of the length of all suffixes that constitute the subtree. Pair generation on the other hand takes time linearly proportional to the number of output pairs. Since the sizes of the buckets may not be uniform, load imbalance could occur. Further, the number of pairs generated by a tree is completely dependent on the content of the trees, which also varies (quadratic in the worst case).

The length-based filter does not directly cause load imbalance since it requires negligible computation time on its own. However, when used as part of the brute force strategy, it will reject pairs as they enqueue for computation and will ultimately alter the already imbalanced workload but in a similarly imbalanced way.

3.2.2 Load Imbalance in Homology Detection

Figure 3.1 shows the histogram and normalized cumulative distribution of alignment processing times for all-against-all alignment of 15,000 sequences obtained from a metagenomics sequence database (Sun et al., 2011). We observe from Figure 3.1a that a significant fraction of tasks are of the order of milliseconds or lower, with a non-negligible fraction consuming well above a millisecond. The large number of tasks together with the wide disparity in the task processing times exacerbates problems associated with static load balancers due to small errors in estimation of alignment times. The alignments include a few large tasks taking few tenths to over one second.

Figure 3.1b shows the cumulative distribution of time spent in processing all tasks that can be processed under a particular time. As we anticipated, despite their counts, the smallest alignment operations consume a negligible fraction of the total processing time. On the other hand, alignment operations that can be processed in 1ms to 100ms consume almost 90% of the total processing time. This shows that the alignment operations critical to load balanced execution vary by up to two orders of magnitude in their processing time.



(a) Distribution of the number of alignment operations for different binning times. x-axis — alignment time in log scale; y-axis — number of alignment operations.

(b) Normalized cumulative distribution time spent aligning tasks of different sizes. x-axis — alignment time in log scale; y-axis — normalized cumulative time to process all alignments in that time or lower.

Figure 3.1: Characterization of time spent in alignment operations for an all-against-all alignment of a 15K sequence dataset from the CAMERA database.

3.2.3 Solutions to Load Imbalance

The means to load balance computations falls into three broad categories, namely static partitioning, dynamic repartitioning, and asynchronous repartitioning. In static partitioning, the work is collectively distributed among available compute resources based on available load information. Dynamic repartitioning is similar to static repartitioning; however, rather than performing once at the beginning of the computation it is performed periodically and collectively. The last strategy is to asynchronously migrate work between compute resources without exchanging information collectively. In the case of homology detection, as shown by the characteristics in Figure 3.1 as well as due to the dynamic nature of suffix subtree processing and pair alignments, the best load balancing approach would also need to be dynamic and asynchronous. Examples of asynchronous load balancing include work stealing and distributed task counters.

Work Stealing: Scalable work stealing as a general approach to asynchronous load balancing is detailed by Dinan et al. (Dinan et al., 2009) and Lifflander et al. (Lifflander et al., 2012) while its application to sequence alignment is covered in (Daily et al., 2012). Briefly, work stealing models a shared task pool. The task can be represented by any fixed-size datatype including structures. The implementation of Dinan et al. places a portion of the task pool on each process in a double-ended queue (deque) which is split into shared and private portions. Tasks can be released from the private portion to the shared portion without locks; acquiring tasks from the shared portion to the private portion requires locking. Tasks may also create additional tasks as part of their execution; however dynamically adding tasks to the pool is done into the private portion and the process becomes lock free. When a worker runs out of tasks in both the private and shared portions of its deque, it becomes a thief. Thieves choose a random victim and attempt to steal half of their tasks, if available. A termination detection algorithm is used to end the task pool execution. The implementation of work stealing in Lifflander et al. (Lifflander et al., 2012) and Daily et al. (Daily et al., 2012) uses an MPI+*pthreads* execution model and an active message programming model instead of the PGAS model used by Dinan et al; however it follows the same model of a shared task pool. The implementation requires one core per compute node be reserved as a progress thread. Even so, it was shown to scale to over 100K cores with 75% efficiency (Daily et al., 2012).

Work Stealing with Iterators: A special form of work stealing can be utilized when the tasks are a finite countable set and can therefore be represented as a contiguous sequence of natural numbers. Instead of implementing the task pool with one deque per worker, each worker stores a range of numbers from the task set as a $[low..high]$ interval. Therefore, a steal operation splits the victims range in half and only transfers two integer values instead of half of a queue's tasks. This results in both memory and communication bandwidth savings. We use a combinatorial number system of degree 2 in order to translate a non-negative index to a lexicographically ordered 2-combination which represents the two sequences to align as described by (Daily et al., 2012; Knuth, 2005). We explore using work stealing iterators to improve the efficiency of work stealing for sequence alignments.

Distributed Task Counters: Work stealing iterators are a form of a distributed task counter. Many high-speed interconnects provide hardware-accelerated implementations of an atomic integer fetch-and-add instruction which can be used to implement a distributed task counter. A process requesting a new task increments the value of the counter while reading the old value. The atomicity of the instruction guarantees that each calling process reads a unique counter value. We translate the counter value into a pair of sequence IDs using the same combinatorial number system of degree 2 as with work stealing iterators. Using distributed task counters does not necessarily require one core per node to be reserved, especially on high speed interconnects. This can result in improved efficiency with respect to work stealing. Further, although less important, distributed task counters only allocate space for the counter on a single process which avoids the need to allocate portions of the task pool on each process. We explore using distributed task counters to improve the efficiency of work stealing for sequence alignments.

3.3 Implementation

Having evaluated many approaches (see Chapter 4), we arrived at the architecture detailed in Figure 3.2. The basis of our implementation relies on the work stealing model as described in Subsection 3.2.3. One thread per compute node is reserved to facilitate the transfer of tasks. Tasks are stolen only from the shared portion of a victim's task deque and are delivered to the thief's private portion. Computing (removing) or alternatively adding a task to the worker's deque causes the local work to rebalance between the shared and private portions. After an all-to-all exchange of suffixes, both to statically load balance subtree work as well as to place all suffixes needed for a given subtree on a single process, the task pool is initially seeded with only all of the subtree processing (pair generation) tasks. However, as subtree tasks are processed they add pair alignment tasks to the pool,

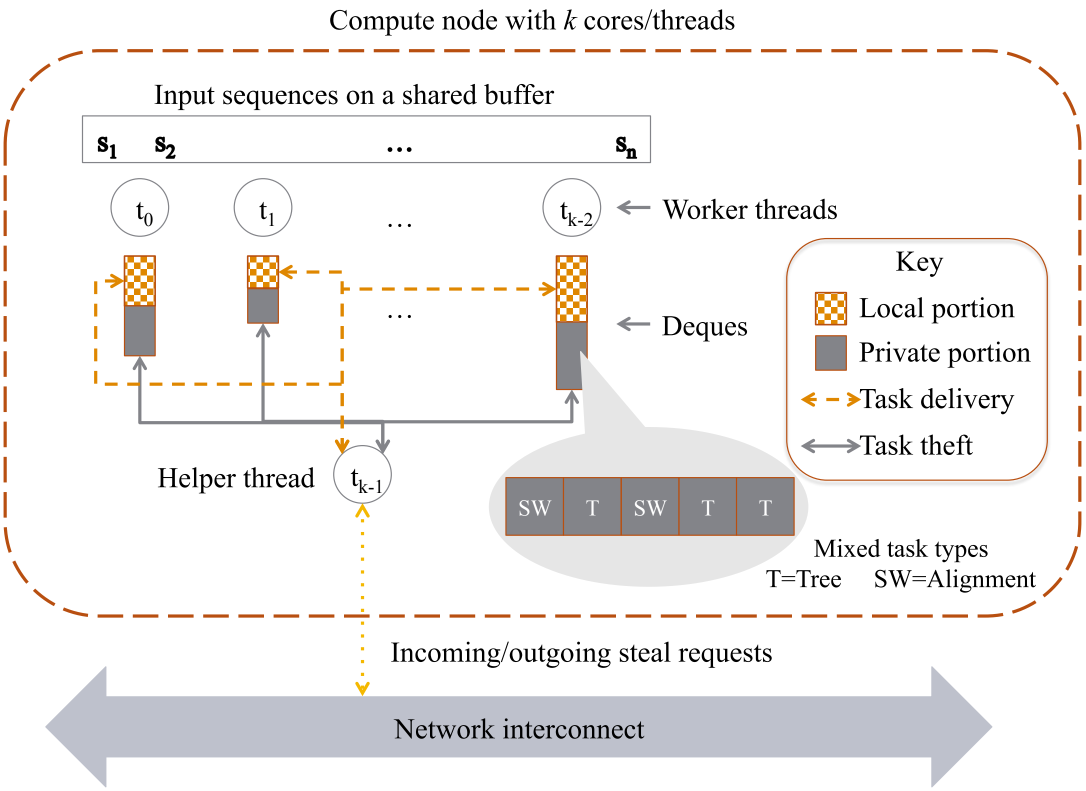


Figure 3.2: Schematic illustration of the execution on a compute node. One thread is reserved to facilitate the transfer of tasks. Tasks are stolen only from the shared portion of the deque and delivered only to the private portion. The set of input sequences typically fits entirely within a compute node and is shared by all worker threads. The task pool starts having only subtree processing (pair generation) tasks but as subtree tasks are processed they add pair alignment tasks to the pool, as well. The dynamic creation and stealing of tasks causes the tasks to become unordered.

as well. The dynamic creation and stealing of tasks causes the tasks to become unordered. The input sequence database is only distributed if there is insufficient memory on a compute node. Although the system we tested had ample resources, we evaluate both the limited and unlimited memory cases.

CHAPTER FOUR

RESULTS AND DISCUSSION

Here we present our performance analysis which covers our exploration of alternative load balancing strategies and sequence alignment pair filters for scalable homology detection.

4.1 Compute Resources

Experiments were performed on the *Hopper* supercomputer at the National Energy Research Scientific Computing Center (NERSC)[1]. It is a 1.28 petaflop/sec Cray XE6 consisting of 6,384 compute nodes made up of 2 twelve-core AMD 'MagnyCours' 2.1 GHz processors and 32GB RAM per node. Hopper's compute nodes are connected by the Cray Gemini Network which is a custom high-bandwidth (8.3GB/s), low-latency ($< 1\mu$s) network with a topology of a 3D torus. We compiled our application using the the Intel® C++ 64 Compiler XE, version 12.1.2.273 using the flags -O3 -pthread. The MPI library is a custom version of mpich2 for Cray XE systems, version 5.4.4.

4.2 Datasets

The following evaluations were performed using input datasets containing 80K, 1280K, 2560K, and 5120K amino acid sequences in FASTA format. The datasets were created by randomly sampling from the Sorcerer II Global Ocean Sampling dataset (Yooseph et al., 2007) made available by the CAMERA (Sun et al., 2011) data portal. The 80K, 1280K, 2560K, and 5120K datasets have total sequence character lengths of 43M, 221M, 390M, and 727M respectively and average sequence lengths of 541.7, 173.1, 152.5, and 142.2.

[1] https://www.nersc.gov/users/computational-systems/hopper/

4.3 Length-Based Filter

The benefit of the length-based filter is that it does not require global knowledge of all sequences while also taking negligible time to compute. On the other hand, being a local filter it requires examining all $\binom{n}{2}$ pairs. Because we must examine all pairs we can then enumerate all pairs which allows us to use the load balancing strategies of work stealing iterators and distributed task counters in addition to the original work stealing strategy (see Subsection 3.2.3). Figure 4.1 shows the strong scaling performance of work stealing all pairs, adding the length-based filter, using task counters, and adding the length-based filter (Brute, BruteLength, Counters, and CountersLength, respectively) for the 80K dataset. The work stealing iterators approach is not shown here because it performed similarly to work stealing with tasks.

What we see in Figure 4.1 is that the dynamic task counters performed better with and without the length-based filter. This is due to the work stealing approach reserving one core per compute node for communication progress. The dynamic task counters do not have that limitation. This amounts to a 6.7% increase in performance which is reasonable considering on the hopper system we are utilizing the otherwise reserved 24^{th} core (4.2% of the available cores). There are additional modest gains in performance due to the reduced communication requirements of the dynamic task counter approach compared to the frequent stealing attempts of work stealing.

When considering the length-based filter, the wall clock savings are 20%. When we look at the number of alignments performed, the efficiency of the length-based filter is 30%. For the four approaches considered here, the scalability is nearly perfect. However, when considering the 99% filter efficiency of the suffix tree approach, the perfect scalability of the length-based filter approach is overshadowed. The length-based filter simply leaves too much work to be performed by the remaining pairs to make this an effective filter on its

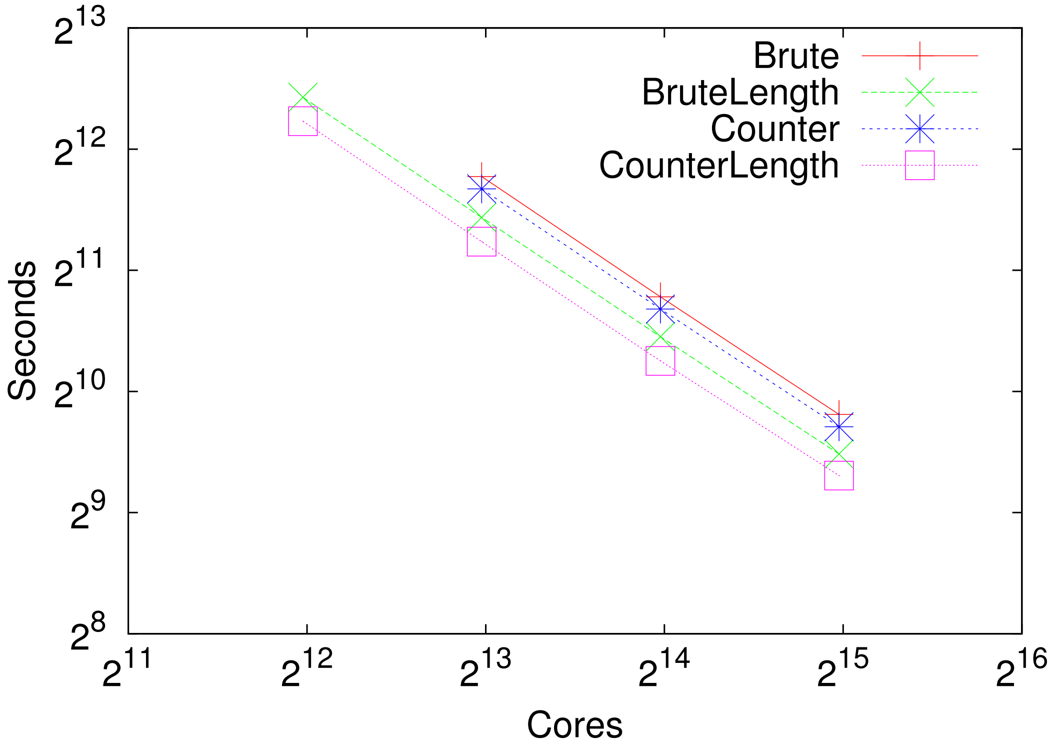


Figure 4.1: Execution times for the 80K dataset using the dynamic load balancing strategies of work stealing ('Brute') and distributed task counters ('Counter'). Additionally, a length-based filter is applied to each strategy. Work stealing iterators are not considered as they performed similarly to the original work stealing approach.

own.

4.4 Suffix Tree Filter

The suffix tree filter was already known to eliminate $\sim 99\%$ of the alignments (Wu et al., 2012). However, it may produce the same pair for alignment more than once. The theoretical maximum number of duplicates per pair (i, j) is bounded by the number of distinct maximal matches between those two strings (Kalyanaraman et al., 2003). To analyze the cost of duplicate pairs, we augmented the suffix tree filter with a C++ STL set and inserted the pairs as they were generated in order to discard duplicate pairs. We processed the suffix tree for the 80K dataset on a single compute node (without performing alignments) as well as in parallel on 4K cores. The suffix tree constructed entirely on a single compute node (therefore eliminating all duplicates) generated 6,401,316 pairs out of a possible 3,199,960,000 (eliminating 99.8% of pairs). The distributed suffix tree filter, while able to eliminate duplicates within each subtree, produced 15,136,463 pairs which is an increase of 136.5% over the perfect duplicate elimination (eliminating 99.5% of pairs).

In order to take advantage of perfect duplicate elimination for distributed suffix subtree processing, we implemented a simple distributed hash table. The entire time spent removing duplicates via the distributed hash table never amounted to more than one second of the total application runtime for all datasets and all core counts we tested. Globally removing duplicate pairs was thus a viable approach. We use this approach for the remainder of our evaluations.

Figure 4.2 shows how the length filter compares to the suffix tree filter using the 80K sequence dataset. For this input, the running time when using the suffix tree filter is already less than a minute at 1K cores so it's not surprising that scalability is limited to 4K cores. What should be noted is the drastic difference in the time to solution and resource needs; even at the smallest core count the suffix tree filter is over an order of magnitude faster than the best alternative filter strategy of the length-based filter with distributed task counters.

4.4.1 Suffix Tree Heuristic Parameters

The tree cut depth k and the minimum exact match length cutoff parameters for the suffix tree can have a direct impact on the number of promising pairwise sequence alignments suggested by the suffix tree filter. In order to measure this impact, we varied these two parameters independently for the 80K dataset while keeping the number of processors fixed at 240.

We found that changing the cut depth k only changed the number of subtrees to create and process which directly impacts the time to solution. Our fastest times had $k = 3$ which

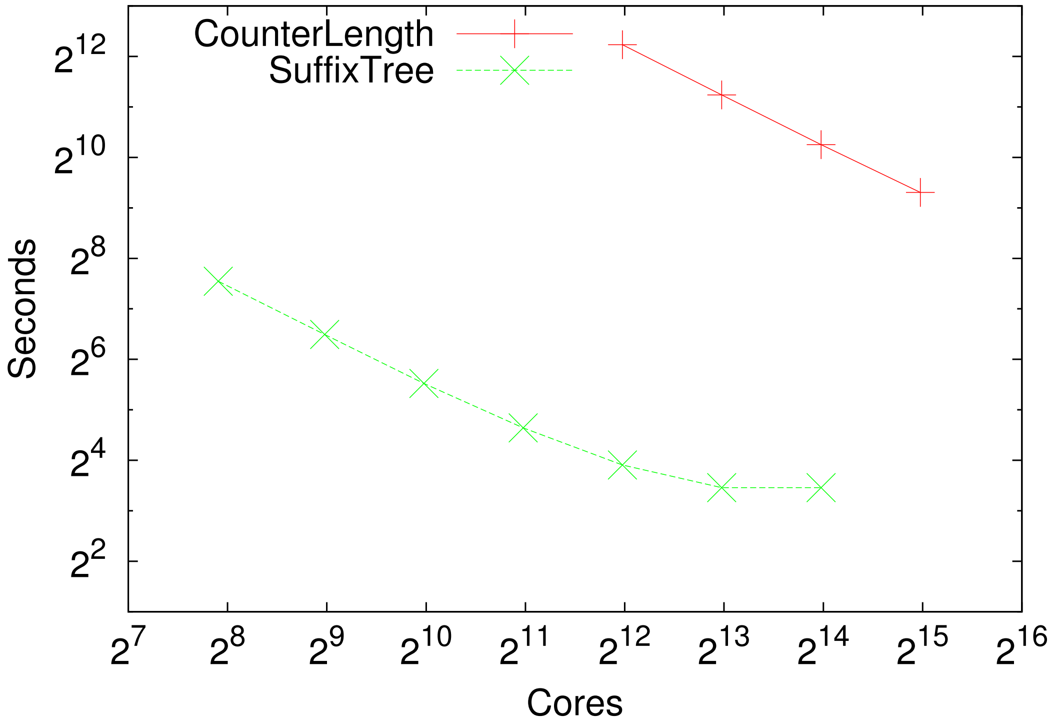


Figure 4.2: Execution times for suffix tree filter and best non-tree filter strategy for the 80K sequence dataset.

generated $8,353$ subtrees. Setting $k = 5$ generated more subtrees $(2,790,772)$ but each of the subtrees contained fewer suffixes and were processed more quickly. However, the additional subtrees eventually caused modest slowdown compared to $k = 3$ since after processing a subtree the duplicate pairs are eliminated which caused contention for the distributed hash table. Setting $k = 1$ caused significant slowdown since the number of buckets generated (21) was much smaller than the number of processes such that there wasn't enough work available to be performed in parallel. For all inputs considered in this evaluation, setting $k = 3$ was sufficient. Setting k did *not* have any impact on the number of alignments to perform because alignment decisions are based on the minimum exact match length cutoff parameter, which is always greater or equal to the cut depth chosen.

Changing the minimum exact match length cutoff dramatically changed the number of alignments to perform but had less impact on the number of homologous pairs identified. Our default of 7 produced $2,707,143$ pairs of which $435,152$ were homologous and ran for 179 seconds. Setting it higher to 9 produced $1,303,842$ pairs of which $404,735$ were homologous and ran for 101 seconds. Setting it lower to 6 produced $6,401,179$ pairs of which $442,828$ were homologous and ran for 499 seconds. Setting it to 5 or lower caused excessive running times. For all experiments hereafter we set this cutoff to 7.

4.4.2 Distributed Datasets

Figure 4.3 shows how the suffix tree filter performs when the 80K dataset has been distributed across multiple nodes. Although the hopper system has ample resources available, it is important to measure the effect of a distributed sequence dataset. We limited the resources available to each node's processes such that the 80K dataset was split across every two nodes in a round robin fashion. If a sequence was no longer local to a node, it would request the remote sequence from the nearest rank with that sequence. Distributing the dataset had no effect on the time taken to compute alignments since any alignment would require at most two sequence fetches. However, the number of sequence fetches needed for any particular suffix subtree could be large. The suffix subtree creation and processing is the primary reason for the decrease in performance when using a distributed sequence database. The results in Figure 4.3 are from our implementation which caches all needed sequences during tree construction and processing. In addition, fetching and caching remote sequences one at a time as they are needed by the tree construction algorithm performed better than a bulk request of all needed sequences at the start of tree creation due to communication contention.

4.4.3 Strong Scaling

Hereafter again considering replicated sequences, Figure 4.4 shows the strong scaling performance of the suffix tree filter running concurrently with sequence pair alignment using

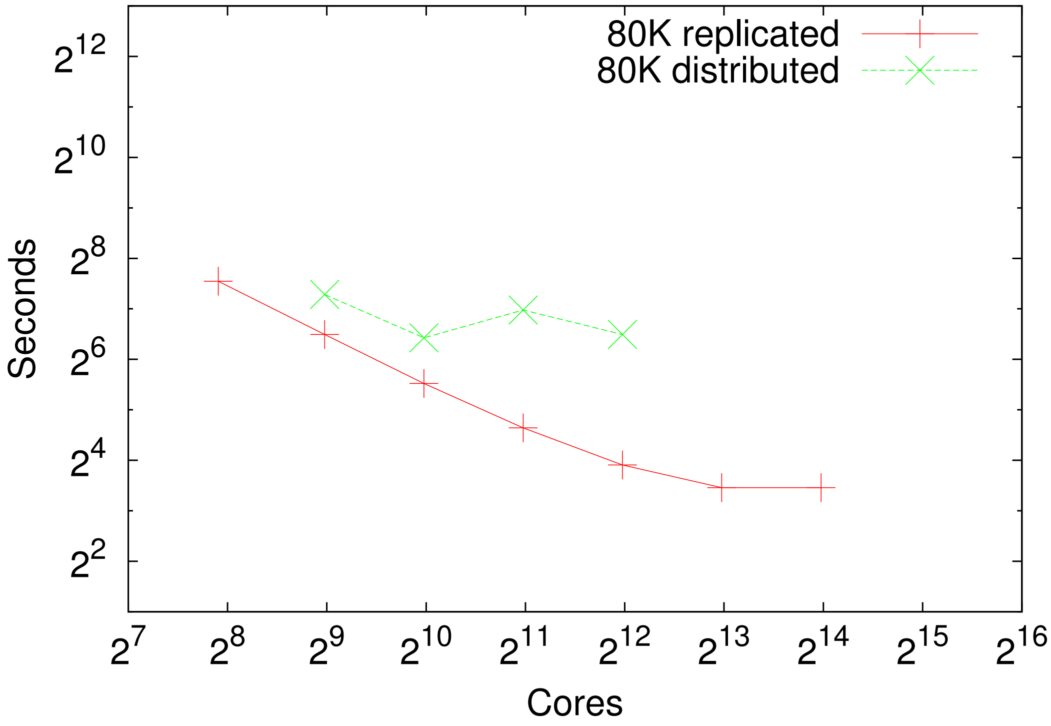


Figure 4.3: Execution times for suffix tree filter for the 80K sequence dataset when it is replicated on each node or distributed across nodes.

the 1280K, 2560K, and 5120K sequence datasets. Although using the suffix tree filter is far better than any other known filter strategy, it does not scale for larger inputs. This is because the real world datasets we tested have a few highly occurring substrings, thus resulting in some subtrees containing an inordinate number of suffixes – more than 10 standard deviations away from the average number of suffixes per subtree. The suffix tree filter is only as fast as its longest-processing subtree. In the case of the 2560K sequence dataset, this amounts to approximately 25 seconds for the largest single tree which is why we see the wall clock time never go far below approximately 32 seconds even as we increase the number of processors. In the case of the 5120K sequence dataset, the most time spent processing a single tree was 389 seconds, limiting the scalability to 1K cores. Figure 4.5 further illustrates the poor scaling due to the single long-running subtree task.

There are two options for mitigating the scalablity challenges imposed by large suffix subtrees. First, such large subtrees could simply be ignored since a highly repetitive substring will not produce meaningful homology results, but this comes with the trade-off of missing some valid homologous pairs. Second, such large subtrees could be subdivided into additional subtrees rooted at greater depths within the larger subtree. We attempted the latter approach by dynamically cutting large trees if they were two or more standard deviations away from the global average number of suffixes per subtree. We continued to recursively subdivide large trees until either the number of suffixes in the resulting trees were small enough or if the cut depth reached the minimum exact match length criteria. However, this did not significantly or consistently improve performance because the commonly occurring substrings were as long or longer than the minimum exact match length criteria – subtrees would be further divided without significantly reducing the number of suffixes in the problematic subtrees. This result highlights the worst case scenario where the commonly occurring substring might still be longer than the minimum exact match length criteria requested by the user (this value was 7 in our tests compared to the cut depth of 3 also used in our tests). Cutting the suffix tree any deeper than the minimum exact match length criteria would likely result in missed pairs. This is a possible indication that either the minimum exact match length cutoff is too short for this subtree or the prefix exact matching sequence corresponding to this subtree is a highly repetitive sequence in the input and hence the subtree can be discarded. Removing the bottleneck of large subtrees will be addressed in future work.

Compared to our preliminary work (Daily et al., 2012) as well as to our non-tree filters evaluated above, by using the suffix tree filter our time to solution was greatly improved while parallel efficiencies were reduced. The simulated filter in our prior work was computed in constant time and removed arbitrary pairs such that those prior performance results cannot be directly compared to the real suffix tree filter which accurately removes candidate

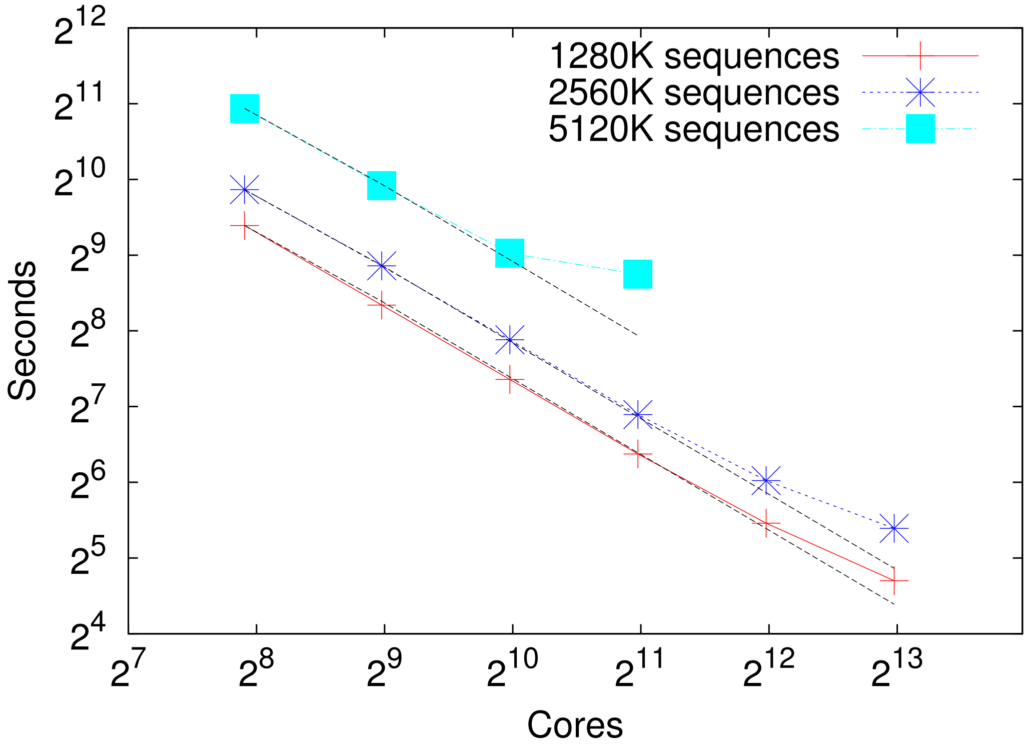


Figure 4.4: Execution times for suffix tree filter strategies for the 1280K, 2560K, and 5120K sequence datasets. The ideal execution time for each input dataset is shown as a dashed line.

pairs while introducing its own processing costs and load imbalance. If not using a suffix tree filter, we are then left with the choice of either using a less-effective but computationally insignificant filter, or not using a filter at all. Either choice would waste computation on poor alignments but would scale better. We believe that in light of trying to process ever-bigger datasets, a reduced time to solution is preferable over pure scalability. Future work will continue to address the scalability challenges.

Our approach here running on the same hardware and with the same datasets *outperformed our preliminary work even when including the suffix tree processing time* up to 8K cores. Compared to Wu et al. (Wu et al., 2012), our parallel efficiencies of over 99% on 2K cores were comparable to their 95% efficiencies on 2K cores. In addition, we were able to have good parallel efficiencies out to 8K cores. Further, our wall clock time (albeit on

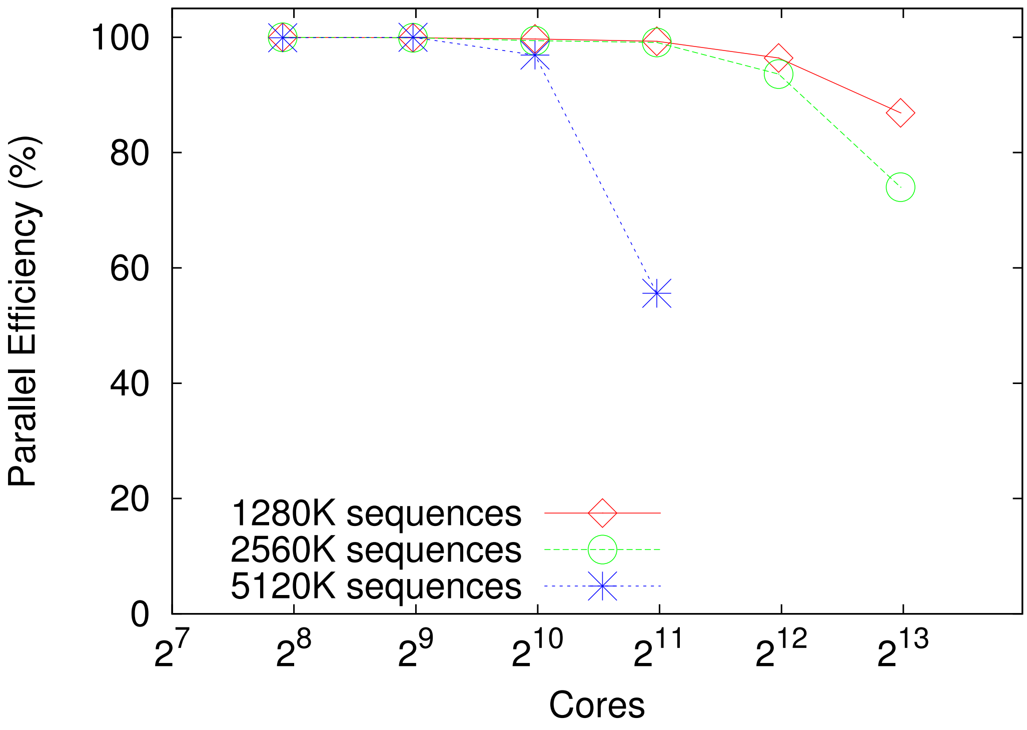


Figure 4.5: Parallel efficiency with input sizes of 1280K, 2560K, and 5120K.

more capable hardware) of 125 seconds at 2K cores for the 2560K dataset is nearly $64\times$ faster than the previously reported 7975 seconds.

Since our goal was to be able to process sequences at the same rates as they are generated on current sequencing equipment, we report our PSAPS results in Figure 4.6. We see from the figure that we did not achieve the same rate of sequence production outlined in Section 1.1 however our best PSAPS rate is over 2×10^6. This is also less than our preliminary work reports at 2.4×10^7 PSAPS. However, our preliminary work, without using a suffix tree filter, was performing imprecise and likely unnecessary work as evidenced by our faster running times when using the suffix tree filter. Our approach, therefore, has better throughput even with fewer PSAPS.

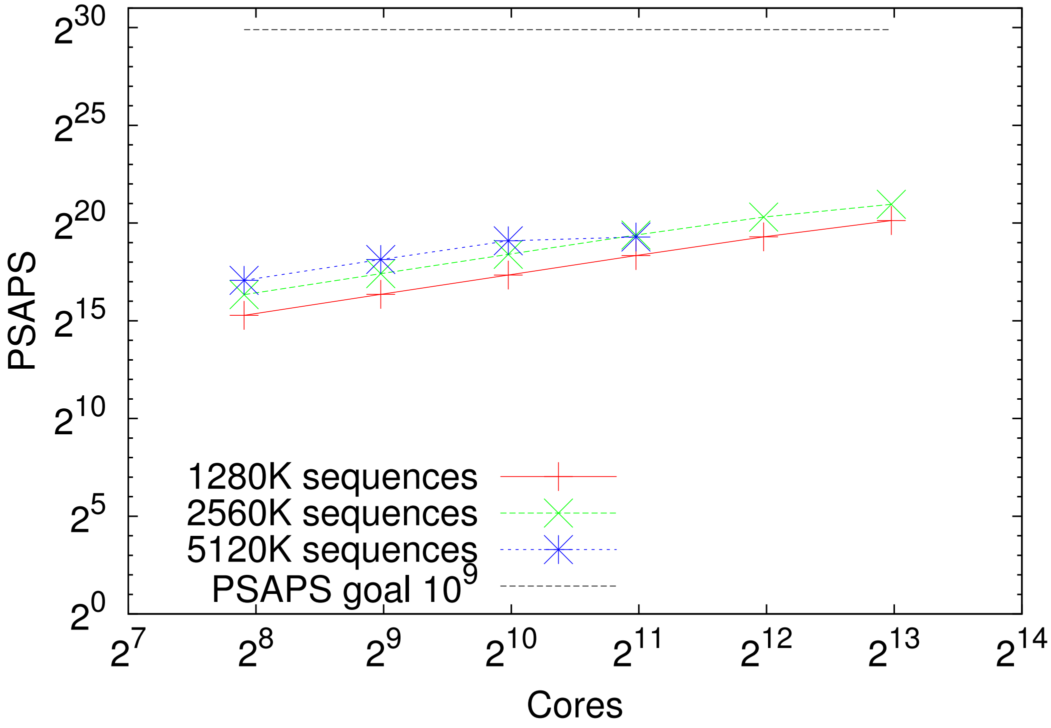


Figure 4.6: PSAPS (Pairwise Sequence Alignments Per Sec) performance with input sizes of 1280K, 2560K, and 5120K.

CHAPTER FIVE

EXTENSIONS

The approach presented and evaluated in Chapters 3 and 4 did not utilize a vectorized implementation of sequence alignments. This was due to the lack of an implementation providing the necessary statistical results as described in 2.1.3. We describe in Section 5.1 a new vectorized implementation of sequence alignment in light of existing approaches. Our implementations of all approaches also optionally compute the required statistics for homology detection.

Since sequence alignment is often the most significant portion of the workload, addressing the performance of pairwise alignment might mean readdressing the rest of the framework. On current and future vectorized hardware, the speed at which an alignment can be performed will increase 10x-20x at which point the cost of stealing an alignment task for load balancing might outweigh the cost of performing the alignment.

Both problems are treated in the sections that follow.

5.1 Vectorized Sequence Alignments using Prefix Scan

Vectorization is an effective way to improve the performance of many kinds of applications via replacing a batch of scalar instructions by vector (SIMD) instructions. In addition, with respect to power consumption, vectorization is considered *free* because it needs relatively little extra hardware support, like SIMD extensions.

Since the 1990s, when Streaming SIMD Extension (SSE) were introduced as part of the x86, they have been widely used in many areas. Recent years have seen SIMD widths expand. Sandy Bridge doubled the SSE SIMD width from 128-bit to 256-bit with new intrinsics called AVX. Moreover, the latest Xeon Phi Coprocessor is equipped with a 512-bit Vector Processing Unit (VPU) with new intrinsics called AVX-512 that can process

16 floating point operations with the same type concurrently. Considering the upcoming Knights Landing CPU with an even more powerful SIMD instructions set, vectorization will provide us more benefits for many applications and algorithms.

There have been many efforts focusing on the vectorization area, especially for dense matrix algorithms. In recent years, many irregular applications have been mapped to various SIMD architectures. However, there are few works considering the effect of increasing SIMD widths on the design and implementation of existing SIMD algorithms. Starting from this viewpoint, we perform a careful study on multiple SIMD sequence alignment algorithms.

In addition to providing a new SIMD implementation of a parallel scan-based sequence alignment algorithms, the objective of this extension is to understand the impacts of widening vector registers on a broad class of sequence alignment algorithms in light of their workload characteristics and parameter ranges.

The order in which we presented the vectorized approaches in Figure 2.2 corresponds generally to their relative performance. Table 5.1 briefly lists the relative performance improvement of each approach. For this analysis, we compare every sequence to each other using a small but representative protein sequence dataset. We implemented each vectorization technique shown here using the SSE4.1 ISA, splitting the 128-bit vector register into eight 16-bit integers. The results of each vector implementation were validated against the scalar result. The first two approaches, namely Blocked and Diagonal, are improved over the scalar implementation, while the Striped approach performed significantly better. For this reason, we only consider Striped and our new Scan implementation for the remainder of this extension. These results reaffirm similar findings from Farrar (Farrar, 2007) for Striped.

Approach	Scalar	Blocked	Diagonal	Striped
Time (s)	70.5	10.6	9.9	4.7
Speedup	1.0	6.6	7.2	15.1

Table 5.1: Relative performance of each vector implementation approach. For this study, we used a small but representative protein sequence dataset and aligned every protein to each other protein. The vector implementations used the SSE4.1 ISA, splitting the 128-bit vector register into 8 16-bit integers. The codes were run using a single thread of execution on a Haswell CPU running at 2.3 Ghz.

5.1.1 Algorithmic Comparison of Striped and Scan

Prior to our experimental evaluation of the Striped and Scan approaches to vectorizing sequence alignment, it is important to understand the algorithmic differences between these approaches. First, we look at the new recurrences for Scan and discuss how to optimally implement them using vectors. This is followed by an analysis of each algorithm's computational complexity.

There are two known formulations for linearizing the data dependencies within the sequence alignment recurrences by using parallel prefix (scan) computation. The approach was first described by Aluru et al. (Aluru et al., 2003), however the formulation by Khajeh-Saeed et al. (Khajeh-Saeed et al., 2010) is simpler though it requires a lengthy proof to confirm its equivalence to the original problem. For comparison with the description in Section 2.1.2, equations from (Khajeh-Saeed et al., 2010) are repeated here in Equations 5.1 through 5.4. Note that this recurrence, computing column by column, initially ignores the influence of the column maximum $D_{i,j}$ and calculates a temporary variable $\widetilde{T}_{i,j}$.

$$I_{i,j} = max(I_{i,j-1}, T_{i,j-1} + G_{\text{open}}) + G_{\text{ext}} \tag{5.1}$$

$$\widetilde{T}_{i,j} = max(T_{i-1,j-1} + W_{i,j}, I_{i,j}) \tag{5.2}$$

$$\widetilde{D}_{i,j} = max_{1<k<j}(\widetilde{T}_{i-k,j} - kG_{\text{ext}}) \tag{5.3}$$

$$T_{i,j} = max(\widetilde{T}_{i,j}, \widetilde{D}_{i,j} + G_{\text{open}}) \tag{5.4}$$

The parallel scan approach is the focus of our implementation. Ideally, the parallel scan would be implemented as described in Blelloch (Blelloch, 1990), mapping a balanced binary tree over the values and using an upsweep followed by a downsweep and applying the associative operator at each node. This is indeed the approach taken by Khajeh-Saeed et al. in (Khajeh-Saeed et al., 2010) though the implementation is written for a GPU. The optimal time complexity of this operation is $\mathcal{O}(n/p + \lg(n))$.

Unfortunately, such operations are not efficient to implement using SIMD vectors. Instead, the parallel scan is implemented in two passes. The first pass has each vector element p compute its portion of the scan in n/p iterations, where n is the number of cells in one column of the DP table (equal to the length of the query sequence). Next, a “horizontal” scan is performed on the resulting vector in $p-1$ operations. Though horizontal operations were added starting in SSE3, our scan requires a combination of addition and maximum rather than just addition or subtraction. Further, the latency and throughput of the horizontal operations are large relative to our approach of shifting the vector $p-1$ times. After the horizontal scan is performed, the resulting vector is shifted to prepare it for the second pass, where it becomes the initial conditions. The second pass is performed in n/p iterations. Instead of the ideal time complexity for the parallel prefix scan, we are left with a time complexity of $\mathcal{O}(n/p + p)$. The pseudocode for the Scan implementation appears in Algorithm 4.

Algorithm 4 Pseudocode for Scan

```
Align_Scan($s_1[1 \dots m], s_2[1 \dots n]$)
Create striped query profile
$L \leftarrow (m + p - 1)/p$                        ▷ number of vector epochs
for each column $j$ along database sequence do
    for each vector epoch $i$ in 1 ... $L$ do
        Load query profile
        Compute and store $I$
        Compute and store $\widetilde{T}$
        Compute initial pass of $\widetilde{D}$
    Local prefix scan of $\widetilde{D}$ result
    for each vector epoch $i$ in 1 ... $L$ do
        Compute second pass of $\widetilde{D}$
        Compute and store $T$
```

Algorithm 5 Pseudocode for Striped

```
Align_Striped($s_1[1 \dots m], s_2[1 \dots n]$)
Create striped query profile
$L \leftarrow (m + p - 1)/p$                        ▷ number of vector epochs
for each column $j$ along database sequence do
    Initialize $D$
    Load $T_{j-1}[L]$
    for each vector epoch $i$ in 1 ... $L$ do
        Load query profile
        Compute $S$
        Load $I_{j-1}$
        Compute and store $T$
        Compute and store next $I$
        Compute next $D$
        Load previous $T_{j-1}[i]$ for next iteration
    while any $D > T$ do
        for each vector epoch $i$ in 1 ... $L$ do
            Recompute $T$
            Recompute $D$
            if not any $D > T$ then
                Break
```

Comparing Algorithms 4 and 5, the Scan approach is only superficially similar to

Striped. For example, as in (Farrar, 2007), the Scan implementation is also striped parallel to the query sequence. In addition, both approaches make at least one full pass over each column in the DP table, but this is where the similarities end. The amount of work performed by each differs in two ways. First, the Striped approach calculates three values per cell, while the Scan approach calculates an additional, temporary value. Second, the Striped approach is often able to abort its additional passes over the column if the upper cell values within the current column no longer contribute to the current cell value. However, in the worst case, it may recompute the column $p-1$ times. The Scan approach will iterate over a column exactly twice and performs the horizontal scan of the intermediate vector $p-1$ times for each column.

Summarizing, the time complexity to compute a column of the DP table using the Scan approach is $\mathcal{O}(n/p + p)$, where n is the length of the query sequence and p is the number of lanes, i.e., processing elements. The Striped approach is nearly identical in its computational complexity with $\mathcal{O}(C * n/p)$, where the additional parameter C is the corrective factor. C represents the number of additional passes until the values converge. The corrective factor C is not necessarily a whole number. For example, a column might converge before reaching its end. For Striped to be effective, $0 <= C << (p-1)$, and ideally it would be zero. The detailed evaluation that follows shows that $C \propto p$ and, due to C, each algorithm has its respective strengths.

5.1.2 Workload Characterization

The performance of the sequence alignment algorithms depends, in part, on the length of the input. Therefore, it is important to know the distributions of sequence lengths for any given set of sequences. We observe that the majority of protein sequences tend to be 300 characters or less, which will have a direct impact on later performance studies.

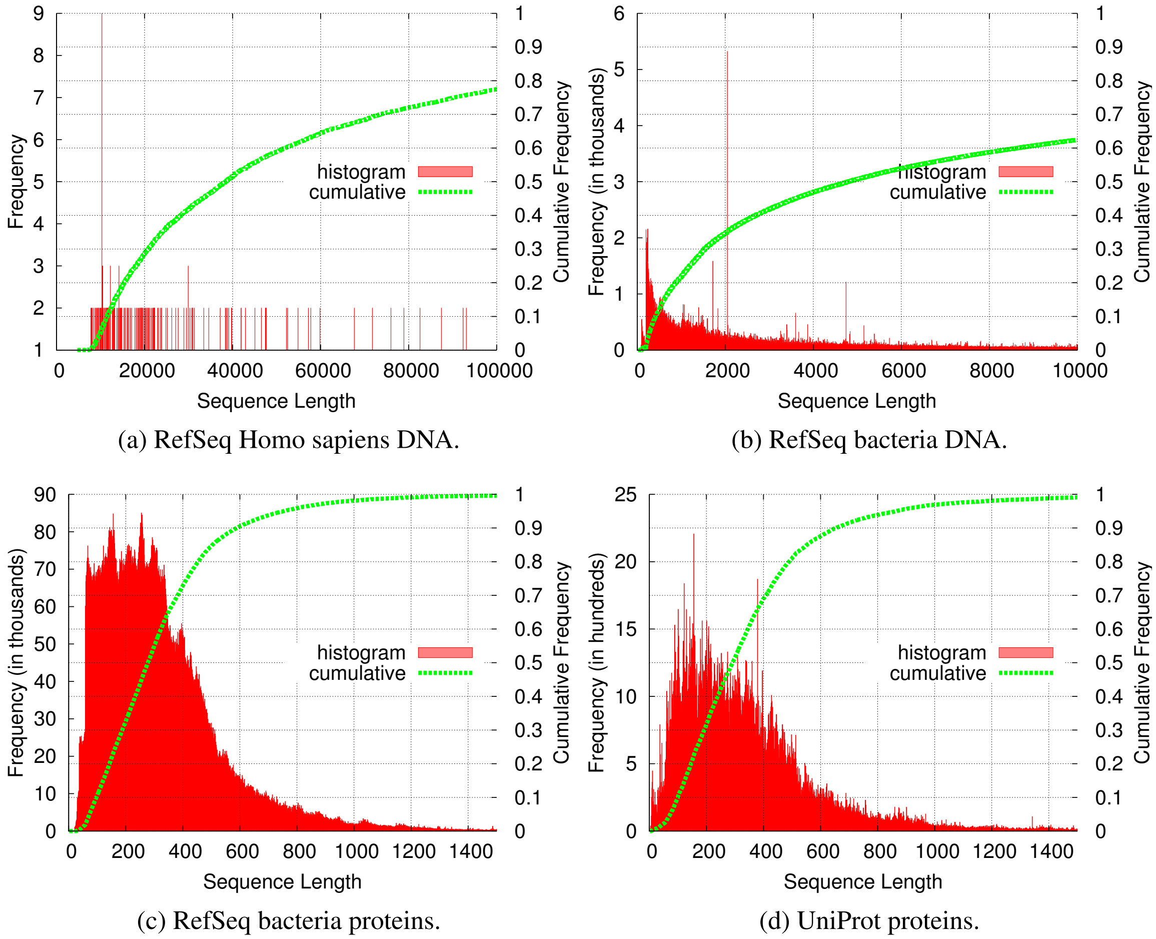
Figure 5.1 characterizes the length distributions of DNA and protein sequences. Genomic DNA sequences tend to vary greatly and can be of significant length. For example, the longest Homo sapiens sequence is 125 Mbp (million base pairs), and the longest genomic bacteria sequence is 14.8 Mbp. Because of such long sequences, Figures 5.1a and 5.1b are truncated before their cumulative frequencies reach 100 percent. Protein sequences tend to be much shorter than DNA sequences. Figures 5.1c and 5.1d show that in two widely used datasets, half of the sequences are length 300 or less. This observation has significant implications for our performance analysis. These four datasets are representative of the various datasets used in other studies.

For many analyses involving the RefSeq bacteria protein dataset, we used a random sampling of 2,000 protein sequences. This dataset is hereafter "bacteria 2K". The sequences within this dataset have an average length of 314 with the longest sequence being 3,206. The frequency of sequence lengths and its cumulative distribution are similar to those found in Figure 5.1c.

For some experiments, we also used the UniProt release mentioned previously (hereafter "UniProt"). In our experiments involving querying a database, UniProt represented our database of sequences. The sequences within this dataset have an average length of 356 with the longest sequence being 35,213. The frequency of sequence lengths and its cumulative distribution appear in Figure 5.1d.

5.1.3 Empirical Characterization

To fully understand the impact of future vector widths on sequence alignments, a number of tests were performed to assess overall algorithm viability. We focus on single-node, single-thread performance to precisely understand the effect of hardware trends within this application domain.



(a) RefSeq Homo sapiens DNA.

(b) RefSeq bacteria DNA.

(c) RefSeq bacteria proteins.

(d) UniProt proteins.

Figure 5.1: Distribution of sequence lengths for all [5,448] RefSeq Homo sapiens DNA (a), all [2,618,768] RefSeq bacteria DNA (b), all [33,119,142] RefSeq bacteria proteins (c), and full [547,964] UniProt protein (d) datasets. Protein datasets are skewed toward shorter sequences, while DNA datasets contain significantly longer sequences. Because of the presence of long sequences, figures (a) and (b) are truncated before their cumulative frequencies reach 100 percent.

Systems and Compilers: The following results were taken on single nodes of two clusters within the PNNL Institutional Computing infrastructure, namely *constance* and *philo*. Constance is based on the Intel Haswell CPU architecture featuring the AVX2 instruction set architecture (ISA). Each node has dual 12-core Intel Haswell E5-2670 v3 CPUs running at 2.3 Ghz with 64 GB 2133 Mhz DDR4 memory per node. The compiler used was Intel ICC 15.0.1 using level three optimization (-O3). The philo cluster consists of nodes with dual 8-core Intel Sandy Bridge E5-2670 CPUs running at 2.6 Ghz with 64 GB of memory. Each philo node has one Intel Xeon Phi 7110P accelerator with 61 cores running at 1.1 Ghz. The compiler used was Intel ICC 13.1.1 using level three optimization (-O3) targeting the MIC architecture (-mmic). The constance cluster was used to test the SSE4.1 and AVX2 ISAs. This was done intentionally to keep the compiler and hardware identical for each of these ISAs to compare the effects of the ISAs rather than the hardware or compiler. The philo cluster was used exclusively to test the performance of the Xeon Phi accelerator that uses the Knights Corner (KNC) ISA.

Scoring Scheme Defaults: Sequence alignments require a scoring scheme as input. The components of the scoring parameters include the substitution matrix, as well as the gap open and gap extension penalties. Unless stated otherwise, all of our experiments use the BLOSUM62 substitution matrix and gap open and extension penalties of -11 and -1, respectively. As with BLOSUM62 or any of the other BLOSUM substitution matrices, we use the default gap open and gap extension penalties as prescribed by the NCBI blastp program.

Datasets: For all analyses, we used sequence datasets from the NCBI Reference Sequence (Tatusova et al., 2014) (RefSeq) database and the Universal Protein Resource (Consortium, 2015) (UniProt) database. The RefSeq project is an ongoing effort to provide a curated, non-redundant collection of sequences, grouped by taxonomy, e.g., fungi, bacteria. UniProt is a comprehensive resource for protein sequence and annotation data. Specifically, from

DP	Method	Lanes	I-refs	D-refs
NW	striped	4	1.3e12	3.7e11
NW	striped	8	9.7e11	2.8e11
NW	striped	16	8.6e11	2.3e11
NW	scan	4	1.6e12	4.8e11
NW	scan	8	8.6e11	2.9e11
NW	scan	16	5.9e11	1.9e11
SG	striped	4	1.1e12	3.5e11
SG	striped	8	7.3e11	2.4e11
SG	striped	16	5.9e11	1.8e11
SG	scan	4	1.6e12	4.8e11
SG	scan	8	8.5e11	2.9e11
SG	scan	16	5.8e11	1.9e11
SW	striped	4	1.3e12	3.4e11
SW	striped	8	7.3e11	2.3e11
SW	striped	16	6.1e11	1.8e11
SW	scan	4	1.8e12	4.7e11
SW	scan	8	9.0e11	2.9e11
SW	scan	16	6.1e11	1.9e11

Table 5.2: Cache analysis of all-to-all sequence alignment for the Bacteria 2K dataset on Haswell. Scan and Striped are using 4, 8, and 16 Lanes. For both scan and striped, instruction and data miss rates for all cache levels were less than 1 percent and are therefore not shown. The primary difference between approaches is indicated by the instruction and data references.

RefSeq we used release 69 which incorporates data available as of January 2, 2015. and from UniProt we used release 2015_02 from 04-Feb-15.

5.1.3.1 Cache Analysis

We performed a cache analysis of the homology detection problem to examine the total instruction counts and cache efficiencies for both Striped and Scan across lane counts, as well as on the Xeon Phi. We used cachegrind to generate reports for the Haswell system and Intel's vtune amplifier for the Xeon Phi system. We found, in general, both Striped and Scan exhibit negligible instruction and data cache miss rates of no more than 1 percent. All measurements, except instruction and data references, are comparable between Scan and Striped, which is why much of that information is omitted from Table 5.2 and Table 5.3.

	NW-Scan	NW-Striped	SG-Scan	SG-Striped	SW-Scan	SW-Striped
Instructions-Retired	6.3e11	9.1e11	6.0e11	6.3e11	6.4e11	6.5e11
CPI-Rate	2.85	2.68	2.72	2.84	2.70	2.72
L1-Misses	2.8e09	1.8e09	2.0e09	1.8e09	2.0e09	1.9e09
L1-Hit-Ratio	0.98	0.99	0.99	0.99	0.99	0.99
Vectorization-Intensity	14.84	13.81	14.82	13.94	14.98	14.10
L1-Comp-to-Data-Acc-Ratio	27.34	29.14	29.79	26.00	32.02	30.39
L2-Comp-to-Data-Acc-Ratio	1731.86	3375.18	2539.59	2324.74	2761.39	2582.76

Table 5.3: Cache analysis of all-to-all sequence alignment for the Bacteria 2K dataset on Xeon Phi. Scan and Striped are using only 16 lanes because the KNC ISA only supports 32-bit integers which restricts this analysis to 16 lanes.

All implementations are extremely cache efficient. This is attributed in part to the use of the striped query profile for both implementations which was already proven by Farrar (Farrar, 2007) to be efficient. The primary reason for the cache efficiencies in our case is the size of the problems being computed. The longest sequence in the bacteria 2K database is 3,206 and easily fits within the cache on both the Haswell CPU and the Xeon Phi CPU. Other cached data includes the values for the DP column being computed. The primary factors affecting performance are the number of instruction and data references.

As expected, the number of instruction and data references decrease as the number of vector lanes increase. However, they decrease more rapidly for Scan than Striped. Striped initially has fewer instructions than Scan when using 4 lanes. By the time 16 lanes are used, Scan has surpassed Striped. Except for the case of NW Striped, where Scan is significantly better, it is not clear whether Scan will continue to outperform Striped for SG and SW.

5.1.3.2 Instruction Mix Analysis

Section 5.1.3.1 described the cumulative instruction counts for the homology detection problem. To understand the lane count trends shown in Table 5.2 and Table 5.3, we ran the same homology detection problem with 16 lanes using Intel's Pin tool (Luk et al., 2005) to capture the instruction mix as shown in Figure 5.2. For space reasons, we omit similar

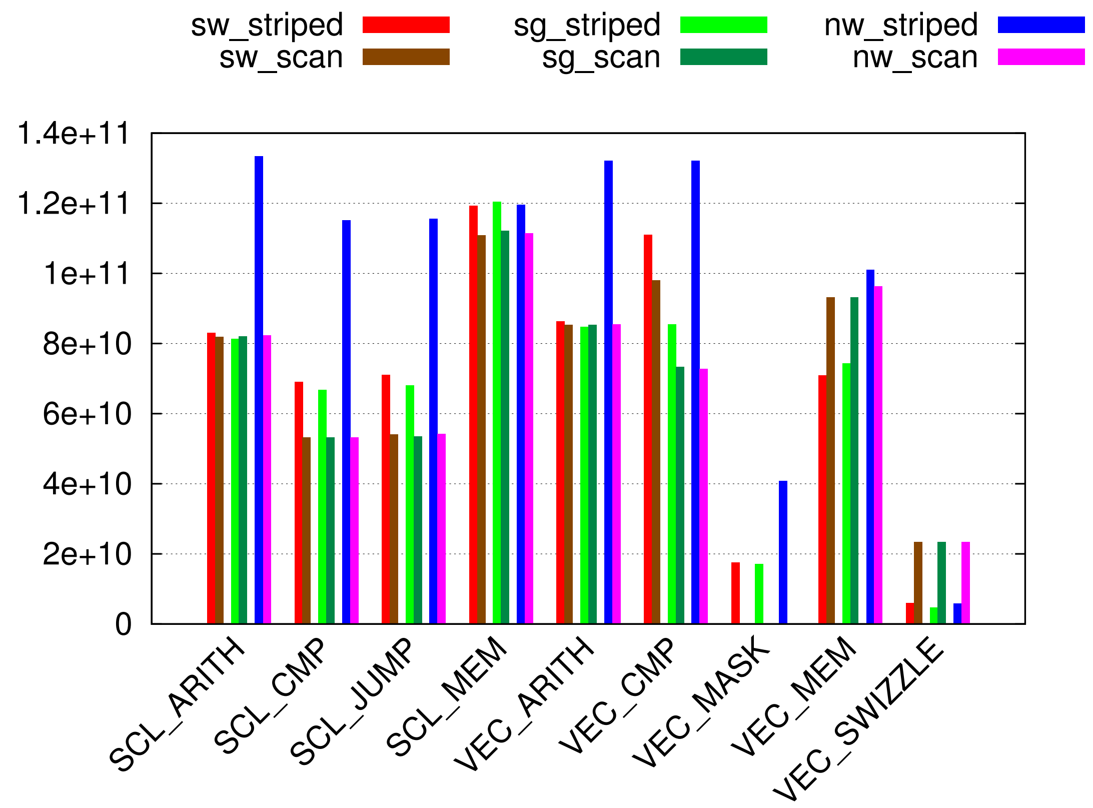


Figure 5.2: Instruction mix for the homology detection problem. For each category of instructions, Scan rarely varies between the three classes of alignments performed, while NW Striped executes more instructions relative to any other case. Striped performs more scalar operations, while Scan performs more vector operations. Scan uses more vector memory and swizzle operations, while Striped is the only one of the two that uses vector mask creation operations.

results for the Xeon Phi which also uses 16 lanes. We observe that for each category of instructions, Scan rarely varies between the three classes of alignments performed. Affirming previous results, NW Striped executes more instructions relative to any other case. We observe that Striped performs more scalar operations, while Scan performs more vector operations. Scan uses more vector memory and swizzle operations, while Striped is the only one of the two that uses vector mask creation operations.

Many of these instruction mix differences can be explained by the algorithmic differences noted in Section 5.1.1. The Striped approach recomputes a column until the values converge. It computes a vector mask and uses additional scalar jumps to check for convergence and break out of the recompute loop. Scan does not compute any vector masks. Assuming the best case, Striped would not need to recompute a column. In such a case, we would expect to see Scan perform more vector arithmetic and comparison instructions because it computes each column twice and computes an additional temporary value per table cell. However, the Striped approach uses more arithmetic and comparison instructions overall. This can only be explained by recomputing the columns a significant number of times. Lastly, Striped performs a few vector swizzle operations before starting a column, while Scan performs more vector swizzle operations because of the $p - 1$ horizontal scan operations performed for each column of an alignment.

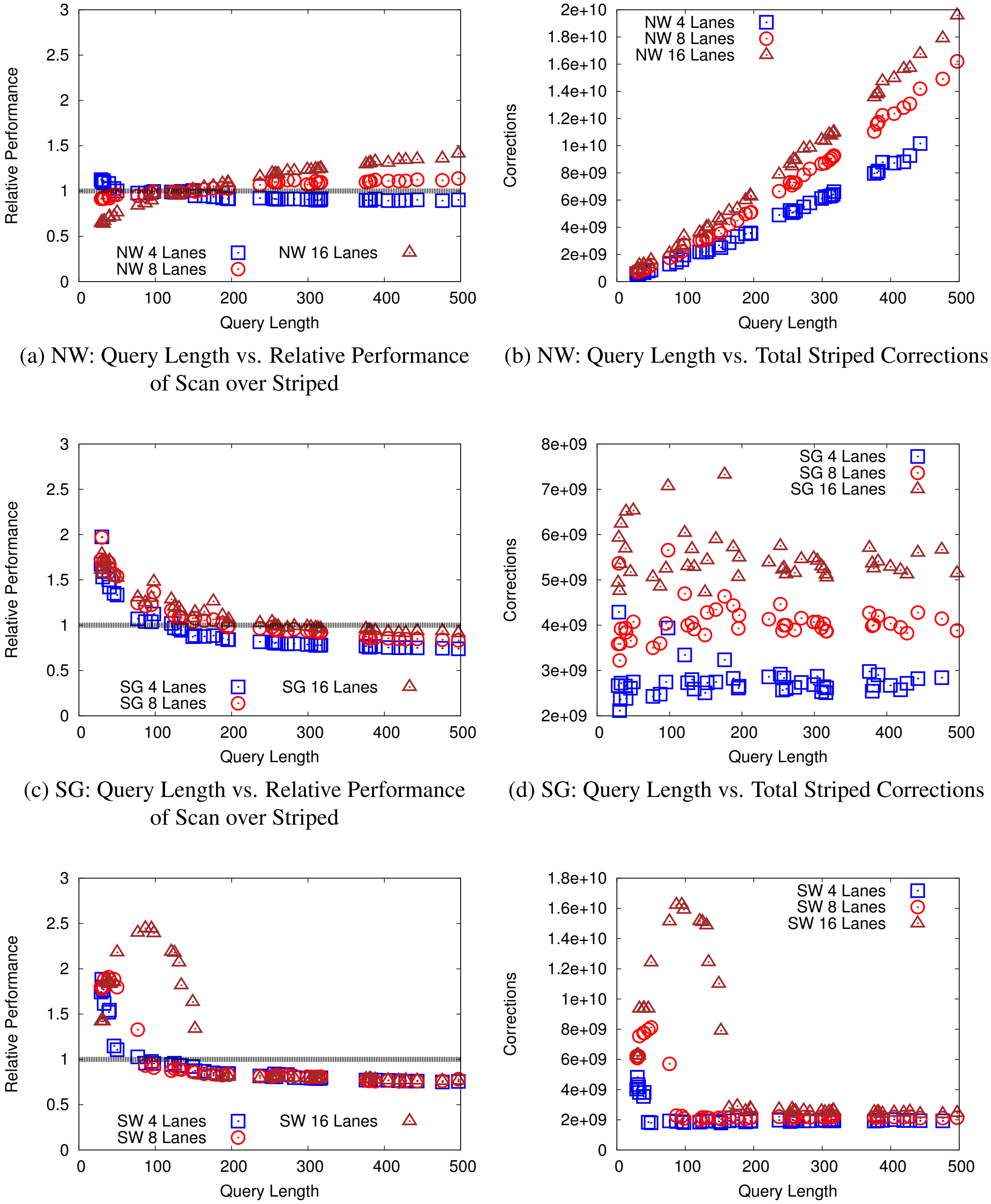
5.1.3.3 Query Length versus Performance

The analyses performed thus far indicate that the performance of Striped and Scan depends on the number of vector lanes applied to the problem. Because the vectors run parallel to the query sequence, the number of vector lanes determines the number of vector epochs based on the lengths of the queries. Therefore, we present the effect of both the query length and number of vector lanes on the relative performance of Striped and Scan.

To that end, we used the bacteria 2K dataset as our query set and performed a database search against the UniProt database. Figures 5.3a through 5.3e show the relative speedup of the Scan approach over the Striped approach as the query lengths increase.

The relative performance of Scan versus Striped shows that both approaches have their merits in light of increasing the number of lanes. Shorter queries perform better for NW Striped, SG Scan, and SW Scan; longer queries perform better for NW Scan, SG Striped, and SW Striped.

The different classes of sequence alignments cross over the relative performance threshold (1.0 on the y-axis) at different points. For NW, the cross over points are for query



(a) NW: Query Length vs. Relative Performance of Scan over Striped

(b) NW: Query Length vs. Total Striped Corrections

(c) SG: Query Length vs. Relative Performance of Scan over Striped

(d) SG: Query Length vs. Total Striped Corrections

(e) SW: Query Length vs. Relative Performance of Scan over Striped

(f) SW: Query Length vs. Total Striped Corrections

Figure 5.3: The relative performance of Scan versus Striped (a-c) shows that both approaches have their merits in light of increasing the number of vector lanes. Shorter queries perform better for NW Striped, SG Scan, and SW Scan. Longer queries perform better for NW Scan, SG Striped, and SW Striped. The reasons for the relative performance differences can be attributed to the number of times the Striped approach must correct the column values before reaching convergence (d-f).

lengths of 149, 149, and 149 for lane counts of 4, 8, and 16, respectively. For SG, the cross overs occur at 121, 188, and 253. For SW, the cross overs occur at 77, 77, and 152. The performance peak at the top of the bubble for SW occurs at 30, 40, and 87. In general, for SG and SW, the cross over points increase with lane counts. For SW, the cross over appears to be jumping dramatically from 8 to 16 lanes. Whether such a dramatic change occurs at 32 lanes needs to be carefully evaluated as new hardware emerges. As for NW, it appears consistently to cross over at query lengths around 150.

The cross over points are particularly concerning in light of many protein datasets being skewed toward shorter sequences. As shown in Figures 5.1c and 5.1d, the majority of protein sequences are 300 amino acids in length or shorter. Therefore, when used as query sequences for database search applications, the point at which SW performance crosses over becomes extremely relevant. Our analysis used, at most, 16 lanes, though 32 lanes will be available in the next wave of CPUs supporting the AVX-512 ISA. Future CPUs and GPUs may continue to adopt even wider vector registers, which, based on these results, is expected to further diminish the return of widening vector registers for this problem domain.

5.1.3.4 Query Length versus Number of Striped Corrections

Combining the results from from the previous analyses, the primary factor influencing Striped performance is the number of corrections that must be made until the column values converge. The number of corrections is further impacted by the number of vector lanes utilized for the Striped computation. This validates the complexity analysis in Section 5.1.1.

For the Striped approach, combining the results from the previous analyses, the primary factor limiting the performance gains afforded by increasing the number of vector lanes is the total number of corrections. Using the same database search application as in

Section 5.1.3.3, Figures 5.3b through 5.3f confirm this observation by showing the plots of query length versus total number of corrections.

The Striped approach, for each column, initializes its D values to zero in the case of SW and to a large negative number in the cases of NW and SG. These are, of course, incorrect values that are later corrected as part of the corrective loop. There is one incorrect value introduced for each vector lane utilized. As the lanes increase, so do the number of incorrect values that can propagate across vector epochs.

The trends displayed show that query length has a direct impact on the number of Striped corrections. For NW, query length is proportional to the number of corrections. In addition, the number of corrections is increasing as lane counts increase. For SG, the number of corrections also increases as lane counts increase, though query length has less of an effect. At shorter query lengths, the number of corrections is less predictable. For SW, there is a clear trend, forming a bubble in the number of corrections relative to the number of lanes. The bubble consistently plateaus when the query length reaches ten times the number of lanes. The peak of the bubbles for SW start at $5E9$ for 4 lanes, then $8E9$ for 8 lanes, and $16E9$ for 16 lanes—roughly doubling as the number of lanes double. Coupled with the computational complexity discussion in Section 5.1.1, this trend will have a severe impact on SW performance as lanes continue to widen. The total number of corrections *increases* as the number of lanes p increase. This implies a correlation between the number of lanes and worst-case performance for the Striped approach.

5.1.3.5 Scoring Criteria Analysis

Having performed a detailed, low-level analysis of Striped and Scan, it remains to be seen whether the observations hold for a user-level analysis. The next experiment is a typical evaluation of the effect that the gap and substitution matrix scoring criteria has on the various implementations. We used the homology detection application with the Bacteria

	Short	Crossover Point			Long
	$<$ Cross	4 Lanes	8 Lanes	16 Lanes	$>$ Cross
NW	Striped	149	149	149	Scan
SG	Scan	121	188	253	Striped
SW	Scan	77	77	152	Striped

Table 5.4: Decision table showing which algorithm should be used given a particular class of sequence alignment and query length.

2K dataset. The substitution matrices used were BLOSUM{45,50,62,80,90} with their corresponding default gap open and extension penalties of $-15 - 2k$, $-13 - 2k$, $-11 - k$, $-10 - k$, and $-10 - k$, respectively. The scoring criteria analysis appears in Figure 5.4.

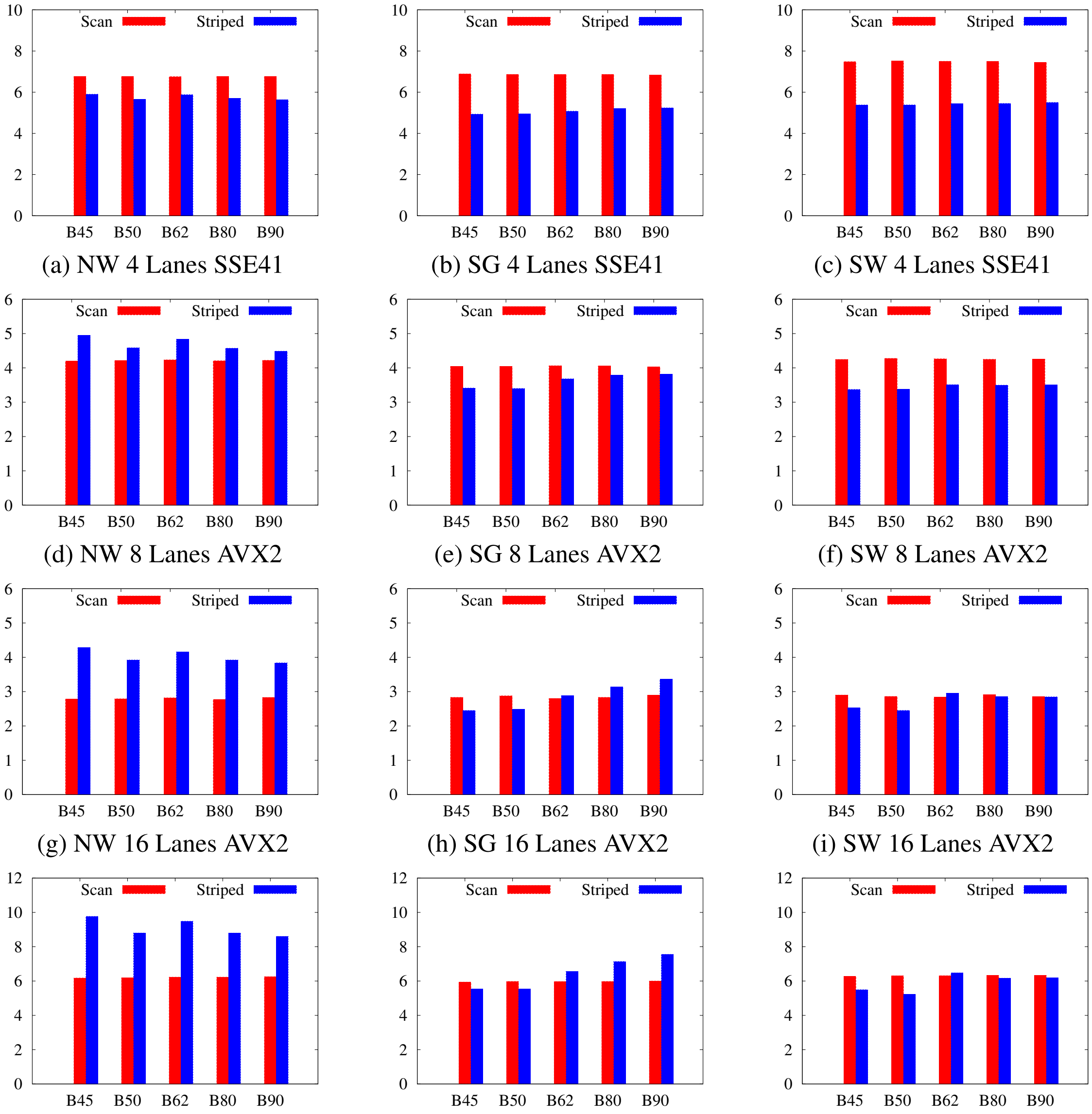
Because the convergence criteria for the column computation in Striped depends on the values of T and D, different substitution matrices and gap penalties affect how quickly the values of T and D diverge—the more divergent, the more corrections must be made. A similar analysis was done by Farrar (Farrar, 2007), though it did not consider NW or SG. Because the Scan approach does not conditionally compute any of its values, the runtimes are stable regardless of the selected substitution matrix or gap penalties.

The Scan approach has stable performance relative to the scoring scheme because it unconditionally makes two passes over each DP table column. The Striped approach varies a moderate amount between selected scoring schemes, generally performing better for smaller gap penalties. As the lane counts increase, the Scan approach eventually overtakes the Striped approach, confirming the results in Section 5.1.3.3.

5.1.3.6 Prescriptive Solutions on Choice of Algorithm

For the particular input datasets we studied, the choice of algorithm to use given a particular class of sequence alignment and query length is summarized in Table 5.4.

We observe that the three algorithms, despite their similarities, exhibit distinct characteristics. Specifically, NW requires a different choice of schemes as compared to SG and SW. In addition, the choice of the schemes is clearly dictated by the input size. Whereas



(a) NW 4 Lanes SSE41 (b) SG 4 Lanes SSE41 (c) SW 4 Lanes SSE41

(d) NW 8 Lanes AVX2 (e) SG 8 Lanes AVX2 (f) SW 8 Lanes AVX2

(g) NW 16 Lanes AVX2 (h) SG 16 Lanes AVX2 (i) SW 16 Lanes AVX2

(j) NW 16 Lanes KNC (k) SG 16 Lanes KNC (l) SW 16 Lanes KNC

Figure 5.4: Total compute times in seconds (Y-axis) for global (NW, left column), semi-global (SG, center column), and local (SW, right column) alignments using the bacteria 2K dataset for a homology detection application. The lane counts increase moving from the first row to the third row, increasing from 4 to 8 and lastly to 16. The fourth row consists of the results for KNC which is also 16 lanes. For each BLOSUM matrix analyzed, the default gap open and extension penalties from NCBI were used as in Section 5.1.3.5. By the time 8 lanes are used, NW Scan consistently outperforms NW Striped. At 16 lanes, Scan begins to outperform Striped for many of the selected scoring schemes.

NW performs better with the Striped implementations for short sequences, SG and SW are faster when using the Scan implementation. The choices are reversed for the long sequences, with NW performing better with Scan and SG/SW performing better with the Striped implementation. The cross-over between what is classified as a short vs a long sequence depended on the SIMD lane width. These widths are shown in columns 3–5 in Figure 5.4. In general, the cross-over points were well within group of shorter sequences in Figure 5.1. Therefore, for longer sequences toward the right of the length distributions in Figure 5.1, the choice of schemes is clear. The cross-over point increased with SIMD lane width for SG and SW. The number of corrections for NW does not vary significantly with lane width, explaining the stability of the cross-over point across SIMD width for NW. In all the cases, widening vector registers makes the parallel scan implementation of the sequence alignment algorithms more attractive.

5.2 Communication-Avoiding Filtering using Tiling

Mansour et al. (Mansour et al., 2011) classify suffix tree construction algorithms into three main categories: in-memory, semi-disk-based, and out-of-core. The in-memory algorithms include McCreight's (McCreight, 1976) and Ukkonen's (Ukkonen, 1995) and are characterized by poor locality of reference and random disk I/Os for inputs larger than main memory. The semi-disk-based algorithms (Hunt et al., 2002; Tian et al., 2005; Phoophakdee and Zaki, 2007) are characterized by better locality of reference but have random disk I/Os and $\mathcal{O}(n^2)$ complexity. Even so, they tend to perform better than the $\mathcal{O}(n)$ in-memory algorithms in practice. The out-of-core algorithms (Ghoting and Makarychev, 2009; Barsky et al., 2009; Mansour et al., 2011) improve further by avoiding random I/Os.

There are only a few fully parallel implementations of suffix tree construction, notably those classified as out-of-core above, while the semi-disk-based algorithms can only be partially parallelized. The semi-disk-based algorithms all share a common two-phase approach of first partitioning the input into smaller subtrees and constructing them separately followed by an expensive merge step which require random I/Os and extensive communication. The out-of-core algorithms avoid the merge step and are therefore more easily parallelizable. However, all algorithms require direct access to the entire input dataset to be available for multiple sequential scans. The best algorithms to date have only been shown to scale modestly; ERa (Mansour et al., 2011) scales to 16 cores with 78% efficiency and PWaveFront (Ghoting and Makarychev, 2009) to over 1,000 cores with 50% parallel efficiency.

If using a suffix tree to filter the $\binom{n}{2}$ pairs of input sequences, a slightly different approach can be used as opposed to building the entire suffix tree first. Both Wu et al (Wu et al., 2012) and Daily et al (Daily et al., 2014) first partition the suffix tree into suffix subtrees similar to the parallel disk-based algorithms. The suffix subtrees can be constructed and processed independently, in parallel. Because they are not concerned with constructing the entire suffix tree, no expensive merge step is required.

Recall from Section 4.4 that the generalized suffix tree filter will generate duplicate pairs which should be removed. In our previous work (Daily et al., 2014), we remove duplicates using a C++ STL set but note an interesting property when processing subtrees in a distributed fashion. A property of suffix trees as stated by Gusfield (Gusfield, 1997, page 98) is that for any internal node with a path-label xa there exists another node with a path-label a. By extension, in the generalized suffix tree, any sequences represented by the leaves below the node for xa will also be represented by leaves below the node for a (the leaves under each node will represent unique suffixes of the GST, but the suffixes will have occurred in the same set of sequences.) If the xa node and a node are processed by difference distributed processes, they will generate similar maximal pairs. The only way to globally remove duplicates is then to exchange duplicate pair information between distributed processes. As a result, we implement a distributed hash table to eliminate duplicates globally between the subtrees being processed. Although the global pair elimination is implemented using an all-to-all communication, we claim the total time spent removing duplicates was no more than one second on average per request. However, the cost of communication will grow as either the input data size grows or as the number of processors increase.

Both (Wu et al., 2012; Daily et al., 2014) use an exact match cutoff heuristic to reduce the number of pairs generated. The assumption is that if two sequences will result in a good alignment score, they will also have a long common substring. For example, they use a cutoff of 7 when processing protein/amino acid sequences. That means every maximal pair generated contains a common substring of at least length 7. However, setting the cutoff to a small value will result in a significant number of maximal pairs. (Daily et al., 2014) reports setting it to 5 caused excessive running times for their 80K sequence dataset.

The growing size of input data sets will eventually necessitate distributing them across the compute cluster. This poses a significant problem as all approaches to date require frequent access to the entire input data set. (Daily et al., 2014) attempts to address this issue by distributing the input data set across a cluster while replicating as much of it as possible on each node to avoid communicating sequences. The resulting strong scaling study showed poor results when the dataset was split across every two nodes in a round-robin fashion.

The trend has been to use the best in-memory approaches until the size of the dataset grows beyond the available resources of a single compute node, at which point either an out-of-core or distributed algorithm is developed. A simple approach using tiling would be to construct the generalized suffix tree for each of $\binom{k}{2}$ partitions. The tree construction cost per partition pair is $\mathcal{O}(n/k)$ time, and since we have $\binom{k}{2}$ different pairs of partitions, the overall time will be $\mathcal{O}(kN)$. This idea was first noted in (Barsky et al., 2009), one of the semi-disk-based suffix tree construction algorithms. However, such an approach would require using a large value of k to allow for efficient utilization of the parallel system. Other, more sophisticated extensions of this simple tiling strategy can be explored taking into account the heterogeneity in task and compute resources, and data locality. An advantage of the tiling approach is that it eliminates the need for duplicate removal. Lastly, since we are now solving many smaller versions of the original problem, we can leverage the best in-memory implementations of the pair filter and sequence alignment algorithms.

CHAPTER SIX

CONCLUSIONS AND FUTURE WORK

We presented a design of a scalable parallel framework which achieves orders of magnitude higher PSAPS performance and at greater scale than contemporary software using the generally applicable all-against-all sequence alignment model. This represents a comprehensive solution to scalable optimal homology detection. This achievement was facilitated using the work stealing dynamic load balancing technique, a one-sided asynchronous PGAS model for data transfer, and a distributed hash table to eliminate duplicate work. Our results demonstrate a promising step towards analyzing biological sequences as fast as they can be generated on contemporary sequencing hardware.

As ongoing hardware advances are made in both memory and the number of compute cores, our solution will continue to remain relevant. As a response to the increasing number of cores, we are beginning to see an increase in the amount of physical memory, as well. The increasing amount of aggregate memory per compute node will allow for continued replication of the entire sequence dataset. As the number of cores increases, the task granularity used for our work stealing-based solution can adjust accordingly to support other models of inter- and intra-node parallelism.

Current and future CPU architectures are trending toward wider vector registers. Therefore, it is imperative that vectorized codes are not adversely affected by these widening trends. This dissertation selected one of the fundamental algorithms from bioinformatics to analyze against these trends. The results were clear: the state-of-the-art implementations based on a striped data layout were inadequate when it comes to realizing the full potential of wider vector registers. At 8 lanes, NW Scan consistently outperforms NW Striped. At 16 lanes, SG and SW Scan outperform Striped for many of the selected scoring schemes. We expect Scan to fully surpass Striped in the next generation of SIMD widths.

We presented a novel SIMD implementation of a parallel scan based algorithm and demonstrate that it overcomes the limitations of the striped scheme. Experimental evaluation demonstrates the three classes of sequence alignment—Needleman-Wunsch, semi-global, Smith Waterman—though very similar in their algorithmic structures, differ widely in their execution times with the Striped and Scan implementations, and in their effective use of wide vector units. We identify the input lengths and vector widths for which one scheme is preferable to the other.

There are a number of promising approaches to further reduce the time-to-solution of homology detection and increase the PSAPS rate. One area for optimization is in reducing the processing time of the worst-case large subtree outliers. Increasing the window size k would produce many more and potentially smaller subtrees. However, due to resource constraints, the window size k cannot simply continue to grow. Using a dynamically sized k is one solution; however k cannot be larger than the minimum match length heuristic provided by the user (in our case it was 7). It may very well be that a real dataset has a frequently occurring substring that is still larger than k.

Our initial results looking at alternative filters and load balancing techniques showed that distributed task counters performed better than work stealing. However, this load balancing approach is only applicable to countable, monotonically increasing enumerated tasks. The extension suggested in Section 5.2 is amenable to all load balancing approaches.

To date, the datasets analyzed contain at most 10M sequences. Our approach, though promising, should be evaluated in light of ever larger datasets. There are additional larger-scale real world applications which might stand to benefit from our approach. For example, the Joint Genome Institute maintains over 100M sequences. A dataset of this size would be a compelling, real-world application.

ATTRIBUTION

Portions of this dissertation proposal were published or have been submitted for publication in the following conference and journal articles.

Workshop Publication:

- Jeff Daily, Sriram Krishnamoorthy, and Ananth Kalyanaraman. Towards scalable optimal sequence homology detection. In High Performance Computing (HiPC), 2012 19th International Conference on, pages 18, 2012. doi: 10.1109/HiPC.2012.6507523.

Journal Publication:

- Jeff Daily, Ananth Kalyanaraman, Sriram Krishnamoorthy, and Abhinav Vishnu. A work stealing based approach for enabling scalable optimal sequence homology detection. Journal of Parallel and Distributed Computing, 2014. ISSN 0743-7315. doi: http://dx.doi.org/10.1016/j.jpdc.2014.08.009. URL http://www.sciencedirect.com/science/article/pii/S0743731514001518.

Conference Submission:

- JA Daily, S Krishnamoorthy, B Ren, and A Kalyanaraman. 2015. On the Impact of Widening Vector Registers on Sequence Alignment. Submitted to the 24th International Conference on Parallel Architectures and Compilation Techniques, San Francisco, CA.

BIBLIOGRAPHY

Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1): 53–86, 2004. ISSN 1570-8667. doi: http://dx.doi.org/10.1016/S1570-8667(03)00065-0. URL `http://www.sciencedirect.com/science/article/pii/S1570866703000650`.

Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3): 403–410, 1990. ISSN 0022-2836. doi: http://dx.doi.org/10.1016/S0022-2836(05)80360-2. URL `http://www.sciencedirect.com/science/article/pii/S0022283605803602`.

Stephen F. Altschul, Thomas L. Madden, Alejandro A. Schffer, Jinghui Zhang, Zheng Zhang, Webb Miller, and David J. Lipman. Gapped blast and psi-blast: a new generation of protein database search programs. *Nucleic Acids Research*, 25(17):3389–3402, 1997. doi: 10.1093/nar/25.17.3389. URL `http://nar.oxfordjournals.org/content/25/17/3389.abstract`.

Srinivas Aluru and Pang Ko. *Lookup Tables, Suffix Trees and Suffix Arrays*, book section 5, page 1104. CRC Press, 2005.

Srinivas Aluru, Natsuhiko Futamura, and Kishan Mehrotra. Parallel biological sequence comparison using prefix computations. *Journal of Parallel and Distributed Computing*, 63(3):264–272, 2003. ISSN 0743-7315. doi: http://dx.doi.org/10.1016/S0743-7315(03)00010-8. URL `http://www.sciencedirect.com/science/article/pii/S0743731503000108`.

AppliedBio. Applied Biosystems by Life Technologies. http://www.appliedbiosystems.com/, 2015. Last date accessed: April 2015.

Marina Barsky, Ulrike Stege, Alex Thomo, and Chris Upton. Suffix trees for very large genomic sequences. In *Proceedings of the 18th ACM conference on Information and knowledge management*, pages 1417–1420, 1646134, 2009. ACM. doi: 10.1145/1645953.1646134.

Alex Bateman, Lachlan Coin, Richard Durbin, Robert D. Finn, Volker Hollich, Sam GriffithsJones, Ajay Khanna, Mhairi Marshall, Simon Moxon, Erik L. L. Sonnhammer, David J. Studholme, Corin Yeats, and Sean R. Eddy. The pfam protein families database. *Nucleic Acids Research*, 32(suppl 1):D138–D141, 2004. doi: 10.1093/nar/gkh121. URL `http://nar.oxfordjournals.org/content/32/suppl_1/D138.abstract`.

Guy E. Blelloch. Prefix sums and their applications. Report, Carnegie Mellon University, 1990.

CancerGenomeAtlas. Cancer genome atlas. http://cancergenome.nih.gov/, 2015. Last date accessed: April 2015.

The UniProt Consortium. Uniprot: A hub for protein information. *Nucleic Acids Research*, 43(D1):D204–D212, 2015. doi: 10.1093/nar/gku989. URL `http://nar.oxfordjournals.org/content/43/D1/D204.abstract`.

Jeff Daily, Sriram Krishnamoorthy, and Ananth Kalyanaraman. Towards scalable optimal sequence homology detection. In *High Performance Computing (HiPC), 2012 19th International Conference on*, pages 1–8, 2012. doi: 10.1109/HiPC.2012.6507523.

Jeff Daily, Ananth Kalyanaraman, Sriram Krishnamoorthy, and Abhinav Vishnu. A work stealing based approach for enabling scalable optimal sequence homology detection. *Journal of Parallel and Distributed Computing*, 2014. ISSN 0743-7315. doi: http://dx.doi.org/10.1016/j.jpdc.2014.08.009. URL `http://www.sciencedirect.com/science/article/pii/S0743731514001518`.

Aaron Darling, Lucas Carey, and Wu-chun Feng. The design, implementation, and evaluation of mpiBLAST. In *4th International Conference on Linux Clusters: The HPC Revolution 2003*, 2003.

Margaret O. Dayhoff, Robert M. Schwartz, and B. C. Orcutt. *A Model of Evolutionary Change in Proteins*, volume 5, pages 345–352. National Biomedical Research Foundation, Washington, D.C., 1978.

James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, 1654113, 2009. ACM. doi: 10.1145/1654059.1654113.

DOEKB. DOE Systems Biology Knowledgebase. http://genomicscience.energy.gov/compbio/, 2015. Last date accessed: April 2015.

Robert C. Edgar. Search and clustering orders of magnitude faster than blast. *Bioinformatics*, 26(19):2460–2461, 2010. doi: 10.1093/bioinformatics/btq461. URL `http://bioinformatics.oxfordjournals.org/content/26/19/2460.abstract`.

S. Emrich, Ananth Kalyanaraman, and Srinivas Aluru. *Algorithms for large-scale clustering and assembly of biological sequence data*, book section 13, page 1104. CRC Press, 2005.

Michael Farrar. Striped smithwaterman speeds database searches six times over other simd implementations. *Bioinformatics*, 23(2):156–161, 2007. doi: 10.1093/bioinformatics/btl582. URL `http://bioinformatics.oxfordjournals.org/content/23/2/156.abstract`.

Amol Ghoting and Konstantin Makarychev. Indexing genomic sequences on the ibm blue gene. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, 1654122, 2009. ACM. doi: 10.1145/1654059.1654122.

Jack A. Gilbert, Folker Meyer, Janet Jansson, Jeff Gordon, Norman Pace, James Tiedje, Ruth Ley, Noah Fierer, Dawn Field, Nikos Kyrpides, Frank-Oliver Glckner, Hans-Peter Klenk, K. Eric Wommack, Elizabeth Glass, Kathryn Docherty, Rachel Gallery, Rick Stevens, and Rob Knight. The earth microbiome project: Meeting report of the 1(st) emp meeting on sample selection and acquisition at argonne national laboratory october 6(th) 2010. *Standards in Genomic Sciences*, 3(3):249–253, 2010. ISSN 1944-3277. doi: 10.4056/aigs.1443528. URL `http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3035312/`. sigs.1443528[PII] 21304728[pmid] Stand Genomic Sci.

Osamu Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162(3):705–708, 1982. ISSN 0022-2836. doi: http://dx.doi.org/10.1016/0022-2836(82)90398-9. URL `http://www.sciencedirect.com/science/article/pii/0022283682903989`.

Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997. ISBN 0-521-58519-8.

Jo Handelsman. Metagenomics: application of genomics to uncultured microorganisms. *Microbiol Mol Biol Rev*, 68(4):669–85, 2004. ISSN 1092-2172 (Print) 1092-2172 (Linking). doi: 10.1128/mmbr.68.4.669-685.2004. Handelsman, Jo Review United States Microbiol Mol Biol Rev. 2004 Dec;68(4):669-85.

HelicosBio. True Single Molecule Sequencing: Helicos BioSciences. http://www.helicosbio.com/, 2015. Last date accessed: April 2015.

Steven Henikoff and Jorja G. Henikoff. Amino acid substitution matrices from protein blocks. *Proceedings of the National Academy of Sciences*, 89(22):10915–10919, 1992. URL `http://www.pnas.org/content/89/22/10915.abstract`.

Ela Hunt, Malcolm P. Atkinson, and Robert W. Irving. Database indexing for large dna and protein sequence collections. *The VLDB Journal*, 11(3):256–271, 2002. ISSN 1066-8888. doi: 10.1007/s007780200064. URL `http://dx.doi.org/10.1007/s007780200064`.

Illumina. Illumina sequencing. http://www.illumina.com/systems.ilmn, 2015. Last date accessed: April 2015.

Ananth Kalyanaraman, Srinivas Aluru, V. Brendel, and Kotharim Suresh. Space and time efficient parallel algorithms and software for est clustering. *Parallel and Distributed Systems, IEEE Transactions on*, 14(12):1209–1221, 2003. ISSN 1045-9219. doi: 10.1109/TPDS.2003.1255634.

Ananth Kalyanaraman, Scott J. Emrich, Patrick S. Schnable, and Srinivas Aluru. Assembling genomes on large-scale parallel computers. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, 2006. doi: 10.1109/IPDPS.2006.1639259.

Ali Khajeh-Saeed, Stephen Poole, and J. Blair Perot. Acceleration of the smithwaterman algorithm using single and multiple graphics processors. *Journal of Computational Physics*, 229(11):4247–4258, 2010. ISSN 0021-9991. doi: http://dx.doi.org/10.1016/j.jcp.2010.02.009. URL `http://www.sciencedirect.com/science/article/pii/S0021999110000823`.

Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 3: Generating All Combinations and Partitions*. Addison-Wesley Professional, 2005. ISBN 0201853949.

Weizhong Li and Adam Godzik. Cd-hit: A fast program for clustering and comparing large sets of protein or nucleotide sequences. *Bioinformatics*, 22(13):1658–1659, 2006. doi: 10.1093/bioinformatics/btl158. URL `http://bioinformatics.oxfordjournals.org/content/22/13/1658.abstract`.

Jonathan Lifflander, Sriram Krishnamoorthy, and Laxmikant V. Kale. Work stealing and persistence-based load balancers for iterative overdecomposed applications. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, pages 137–148, 2287103, 2012. ACM. doi: 10.1145/2287076.2287103.

Heshan Lin, Pavan Balaji, R. Poole, C. Sosa, Xiaosong Ma, and Wu-chun Feng. Massively parallel genomic sequence search on the blue gene/p architecture. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–11, 2008. doi: 10.1109/SC.2008.5222005.

Heshan Lin, Xiaosong Ma, Wuchun Feng, and N. F. Samatova. Coordinating computation and i/o in massively parallel sequence search. *Parallel and Distributed Systems, IEEE Transactions on*, 22(4):529–543, 2011. ISSN 1045-9219. doi: 10.1109/TPDS.2010.101.

Yongchao Liu and B. Schmidt. Swaphi: Smith-waterman protein database search on xeon phi coprocessors. In *Application-specific Systems, Architectures and Processors (ASAP), 2014 IEEE 25th International Conference on*, pages 184–185, 2014. doi: 10.1109/ASAP.2014.6868657.

Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40(6):190–200, 2005. ISSN 0362-1340. doi: 10.1145/1064978.1065034.

Essam Mansour, Amin Allam, Spiros Skiadopoulos, and Panos Kalnis. Era: Efficient serial and parallel suffix tree construction for very long strings. *Proc. VLDB Endow.*, 5(1):49–60, 2011. ISSN 2150-8097. doi: 10.14778/2047485.2047490.

Victor M. Markowitz, Natalia N. Ivanova, Ernest Szeto, Krishna Palaniappan, Ken Chu, Daniel Dalevi, I-Min A. Chen, Yuri Grechkin, Inna Dubchak, Iain Anderson, Athanasios Lykidis, Konstantinos Mavromatis, Philip Hugenholtz, and Nikos C. Kyrpides. Img/m: A data management and analysis system for metagenomes. *Nucleic Acids Research*, 36(suppl 1):D534–D538, 2008. doi: 10.1093/nar/gkm869. URL `http://nar.oxfordjournals.org/content/36/suppl_1/D534.abstract`.

Edward M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM (JACM)*, 23(2):262–272, 1976. ISSN 0004-5411.

Challenges National Research Council Committee on Metagenomics and Applications Functional. *The New Science of Metagenomics: Revealing the Secrets of Our Microbial Planet*. National Academies Press (US) National Academy of Sciences., Washington (DC), 2007. ISBN 978-0-309-10676-4. Book NBK54006 [bookaccession].

NCBI. The National Center for Biotechnology Information. http://www.ncbi.nlm.nih.gov/genbank, 2015. Last date accessed: April 2015.

Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970. ISSN 0022-2836. doi: http://dx.doi.org/10.1016/0022-2836(70)90057-4. URL `http://www.sciencedirect.com/science/article/pii/0022283670900574`.

Chris Oehmen and Jarek Nieplocha. Scalablast: A scalable implementation of blast for high-performance data-intensive bioinformatics analysis. *Parallel and Distributed Systems, IEEE Transactions on*, 17(8):740–749, 2006. ISSN 1045-9219. doi: 10.1109/TPDS.2006.112.

Michael Ott, Jaroslaw Zola, Alexandros Stamatakis, and Srinivas Aluru. Large-scale maximum likelihood-based phylogenetic analysis on the ibm bluegene/l. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–11, 1362628, 2007. ACM. doi: 10.1145/1362622.1362628.

PACBIO. Pacific Biosciences. http://www.pacificbiosciences.com/products/, 2015. Last date accessed: April 2015.

William R. Pearson. Searching protein sequence libraries: Comparison of the sensitivity and selectivity of the smith-waterman and fasta algorithms. *Genomics*, 11 (3):635–650, 1991. ISSN 0888-7543. doi: http://dx.doi.org/10.1016/0888-7543(91)90071-L. URL `http://www.sciencedirect.com/science/article/pii/088875439190071L`.

William R. Pearson and David J. Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences of the United States of America*, 85(8):2444–2448, 1988. ISSN 0027-8424 1091-6490. URL http://www.ncbi.nlm.nih.gov/pmc/articles/PMC280013/. 3162770[pmid] Proc Natl Acad Sci U S A.

PersonalGenomics. Personal Genome Project. http://www.personalgenomes.org/, 2015. Last date accessed: April 2015.

Benjarath Phoophakdee and Mohammed J. Zaki. Genome-scale disk-based suffix tree indexing. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, pages 833–844, 1247572, 2007. ACM. doi: 10.1145/1247480.1247572.

Roche454. 454 Life Sciences - a roche company. http://www.genome-sequencing.com/, 2015. Last date accessed: April 2015.

Torbjorn Rognes. Faster smith-waterman database searches with inter-sequence simd parallelisation. *BMC Bioinformatics*, 12(1):221, 2011. ISSN 1471-2105. URL http://www.biomedcentral.com/1471-2105/12/221.

Torbjrn Rognes and Erling Seeberg. Six-fold speed-up of smithwaterman sequence database searches using parallel processing on common microprocessors. *Bioinformatics*, 16(8):699–706, 2000. doi: 10.1093/bioinformatics/16.8.699. URL http://bioinformatics.oxfordjournals.org/content/16/8/699.abstract.

Souradip Sarkar, Turbo Majumder, Ananth Kalyanaraman, and Partha Pratim Pande. Hardware accelerators for biocomputing: A survey. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pages 3789–3792, 2010. doi: 10.1109/ISCAS.2010.5537736.

Eugene G. Shpaer, Max Robinson, David Yee, James D. Candlin, Robert Mines, and Tim Hunkapiller. Sensitivity and selectivity in protein similarity searches: A comparison of smithwaterman in hardware to blast and fasta. *Genomics*, 38(2):179–191, 1996. ISSN 0888-7543. doi: http://dx.doi.org/10.1006/geno.1996.0614. URL http://www.sciencedirect.com/science/article/pii/S088875439690614X.

Temple F. Smith and Michael S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981. ISSN 0022-2836. doi: http://dx.doi.org/10.1016/0022-2836(81)90087-5. URL http://www.sciencedirect.com/science/article/pii/0022283681900875.

Shulei Sun, Jing Chen, Weizhong Li, Ilkay Altintas, Abel Lin, Steve Peltier, Karen Stocks, Eric E. Allen, Mark Ellisman, Jeffrey Grethe, and John Wooley. Community cyberinfrastructure for advanced microbial ecology research and analysis: The camera resource.

Nucleic Acids Research, 39(Database issue):D546–D551, 2011. ISSN 0305-1048 1362-4962. doi: 10.1093/nar/gkq1102. URL http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3013694/. gkq1102[PII] 21045053[pmid] Nucleic Acids Res.

Roman L. Tatusov, Eugene V. Koonin, and David J. Lipman. A genomic perspective on protein families. *Science*, 278(5338):631–7, 1997. ISSN 0036-8075 (Print) 0036-8075 (Linking). Tatusov, R L Koonin, E V Lipman, D J Review UNITED STATES Science. 1997 Oct 24;278(5338):631-7.

Tatiana Tatusova, Stacy Ciufo, Boris Fedorov, Kathleen ONeill, and Igor Tolstoy. Refseq microbial genomes database: new representation and annotation strategy. *Nucleic Acids Research*, 42(D1):D553–D559, 2014. doi: 10.1093/nar/gkt1274. URL http://nar.oxfordjournals.org/content/42/D1/D553.abstract.

Oystein Thorsen, Brian Smith, Carlos P. Sosa, Karl Jiang, Heshan Lin, Amanda Peters, and Wu-chun Feng. Parallel genomic sequence-search on a massively parallel system. In *Proceedings of the 4th international conference on Computing frontiers*, pages 59–68, 1242542, 2007. ACM. doi: 10.1145/1242531.1242542.

Yuanyuan Tian, Sandeep Tata, RichardA Hankins, and JigneshM Patel. Practical methods for constructing suffix trees. *The VLDB Journal*, 14(3):281–299, 2005. ISSN 1066-8888. doi: 10.1007/s00778-005-0154-8. URL http://dx.doi.org/10.1007/s00778-005-0154-8.

Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995. ISSN 0178-4617. doi: 10.1007/BF01206331. URL http://dx.doi.org/10.1007/BF01206331.

Abhinav Vishnu, Jeff Daily, and Bruce Palmer. Designing scalable pgas communication subsystems on cray gemini interconnect. In *High Performance Computing (HiPC), 2012 19th International Conference on*, pages 1–10, 2012. doi: 10.1109/HiPC.2012.6507506.

Lipeng Wang, Yuandong Chan, Xiaohui Duan, Haidong Lan, Xiangxu Meng, and Weiguo Liu. Xsw: Accelerating biological database search on xeon phi. In *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 950–957, 2014. doi: 10.1109/IPDPSW.2014.108.

Zhong Wang, Mark Gerstein, and Michael Snyder. Rna-seq: a revolutionary tool for transcriptomics. *Nat Rev Genet*, 10(1):57–63, 2009. ISSN 1471-0056. URL http://dx.doi.org/10.1038/nrg2484. 10.1038/nrg2484.

Peter Weiner. Linear pattern matching algorithms. In *Switching and Automata Theory, 1973. SWAT '08. IEEE Conference Record of 14th Annual Symposium on*, pages 1–11, 1973. ISBN 0272-4847. doi: 10.1109/SWAT.1973.13.

A. Wozniak. Using video-oriented instructions to speed up sequence comparison. *Computer Applications in the Biosciences : CABIOS*, 13(2):145–150, 1997. doi: 10.1093/bioinformatics/13.2.145. URL `http://bioinformatics.oxfordjournals.org/content/13/2/145.abstract`.

Changjun Wu and Ananth Kalyanaraman. An efficient parallel approach for identifying protein families in large-scale metagenomic data sets. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–10, 1413406, 2008. IEEE Press.

Changjun Wu, Ananth Kalyanaraman, and William R. Cannon. pgraph: Efficient parallel construction of large-scale protein sequence homology graphs. *Parallel and Distributed Systems, IEEE Transactions on*, 23(10):1923–1933, 2012. ISSN 1045-9219. doi: 10.1109/TPDS.2012.19.

Shibu Yooseph, Granger Sutton, Douglas B. Rusch, Aaron L. Halpern, Shannon J. Williamson, Karin Remington, Jonathan A. Eisen, Karla B. Heidelberg, Gerard Manning, Weizhong Li, Lukasz Jaroszewski, Piotr Cieplak, Christopher S. Miller, Huiying Li, Susan T. Mashiyama, Marcin P. Joachimiak, Christopher van Belle, John-Marc Chandonia, David A. Soergel, Yufeng Zhai, Kannan Natarajan, Shaun Lee, Benjamin J. Raphael, Vineet Bafna, Robert Friedman, Steven E. Brenner, Adam Godzik, David Eisenberg, Jack E. Dixon, Susan S. Taylor, Robert L. Strausberg, Marvin Frazier, and J. Craig Venter. The ¡italic¿sorcerer ii¡/italic¿ global ocean sampling expedition: Expanding the universe of protein families. *PLoS Biol*, 5(3):e16, 2007. doi: 10.1371/journal.pbio.0050016. URL `http://dx.doi.org/10.1371%2Fjournal.pbio.0050016`.