TECHNIQUES FOR SCALING COMPUTATIONAL GENOMICS APPLICATIONS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Kanak Mahadik

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2017

Purdue University

West Lafayette, Indiana

To my parents.

ACKNOWLEDGMENTS

I moved from picturesque San Francisco to sleepy West Lafayette, in pursuit of a PhD, and my experience has been nothing short of amazing. I have had several opportunities to meet wonderful people during this journey, and I would like to acknowledge them.

I would like to thank my advisors Milind Kulkarni and Saurabh Bagchi for their support and guidance throughout the PhD. My meetings with Milind always used to be exciting, involving discussions ranging from interesting algorithmic insights, high-level presentation challenges, and notorious technical bugs. Saurabh's kind manner and critical discussions helped me to communicate and evaluate research ideas. Both have been mentors to me, and have shaped my research career. I am fortunate to have both of you as my advisors. I would also like to thank my advisory committee members, Samuel Midkiff and Ananth Grama for their invaluable feedback and comments. I would like to thank all the committee members for their help related to finalizing the plan of study and the dissertation.

I would like to thank my parents, my in-laws, and my sister for their support and encouragement throughout this journey. Communicating with them about graduate school and other experiences was always relaxing.

I would also like to thank my lab mates in both PLCL (Parallelism, Languages, and Compilers Lab) and DCSL (Dependable Computing Systems Lab). All our interesting conversations, useful distractions playing racquetball and badminton, and exciting outings have always helped me de-stress and enjoy my work.

Finally, I would like to thank my husband, Amit Sabne for being a friend, a mentor, and a constant source of motivation. His companionship and continuous encouragement have been pivotal in keeping me inspired throughout my PhD.

TABLE OF CONTENTS

Page

Page

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

Figure Page

Figure Page

ABSTRACT

Mahadik, Kanak PhD, Purdue University, August 2017. Techniques for Scaling Computational Genomics Applications. Major Professors: Milind Kulkarni and Saurabh Bagchi.

A revolution in personalized genomics will occur when scientists can sequence genomes of millions of people cost effectively and conclusively understand how genes influence diseases, and develop better drugs and treatments. The announcement by Illumina on sequencing a human genome for $1000 is a stellar attempt to solve the first part of the puzzle. However to provide genetic treatments for diseases such as breast cancer, cystic fibrosis, Huntington's disease, and others requires us to develop tools that can quickly analyze biological sequences and understand their structural and functional properties. Currently, tools are designed in an ad hoc manner, and require extensive programmer effort to develop and optimize them. Existing tools also show poor scalability for the exponentially increasing genomic data generated from continuously enhancing sequencing technologies.

We have taken a holistic approach to enhance the performance and scalability of genomic applications handling large volumes of data of application of techniques at three levels - algorithm, compiler, and data structure. At the algorithm level, we identify opportunities for exploiting parallelism and efficient methods of data distribution. Our technique Orion exploits fine-grained parallelism to scale for long genomic sequences and achieves superior performance and better load balance than state-of-the-art distributed genomic sequence matching tools. ScalaDBG transforms the sequential and computationally intensive process of iterative de Bruijn graph construction to a parallel one. At the compiler level, we develop a domain-specific language, called SARVAVID. SARVAVID provides commonly occurring modules in genomics applications as high-level language constructs and performs domain-specific optimizations well beyond the scope of libraries and generic compilers. At the data structure level, we identify opportunities to exploit cache locality

and software prefetching for enhancing the performance of indexing structures in genomic applications. We apply our approach to the major classes of genomic applications and demonstrate the benefits with relevant genomic datasets.

1. INTRODUCTION

Human body cells consist of about 20,000 genes. Genes are part of a long molecule called DNA (deoxyribonucleic acid). DNA has a double helical structure and it encodes all information necessary to build and maintain the organism. More importantly, DNA contains information used by cells to produce proteins, which facilitate all functions of the human being. DNA is composed of nucleotides, which consists of bases adenine ("A"), guanine ("G"), cytosine ("C"), and thymine ("T"). Thus, DNA is encoded using the character set {A, C, G, T}. The functioning of a gene depends on the number and order of these bases in the DNA. Change in the number and order of bases in specific genes of the human body leads to manifestation of genetic diseases such cancer, sickle cell anemia, diabetes and so on. Perhaps the most important application of genomic analysis is the identification and treatment of these diseases. To develop treatments, we need to first collect and generate the genetic codes for millions of people, and then analyze this data to identify genetic patterns that can be associated with the diseases.

The good news is that the cost of sequencing the human genome has gone down from 3 billion dollars to 1000$, owing to the extraordinary progress in genome sequencing technologies. The Human Genome Project presented the first finished human genome in 2003. In 2014, Illumina announced the $1000 human genome. The cost of sequencing has been plummeting since the introduction of next-generation sequencing technologies and we are now at a juncture where algorithms and processors need to play serious catch-up to keep pace with the rate of sequenced data [1]. The rate of increase in number of human genomes sequenced is doubling every seven months, while Illumina estimates the rate of growth to be doubling every twelve months. Using Moore's Law as a benchmark, we might estimate that computer processors double in speed every 18 months. Thus, sequencing technology is outpacing Moore's law and the performance gap continues to widen. It is clear that we cannot rely on performance improvements of computers to speed up genomic applications.

All genomic analyses pipelines start with obtaining raw human genomic information, followed by reconstructing the genome from this information. Genomic analyses applications are run on this genomic data to gather insights such as in pipelines for variant calling [2], differential gene expression [3], and genetic disease testing. Tremendous improvements in sequencing technologies have created a bottleneck for the stages of genome reconstruction and analysis. The tools and applications responsible for these stages are required to scale up and process the massive datasets generated from the first stage of the pipelines. However, current genomic tools are underperforming, hindering the efforts of researchers analyzing these datasets to obtain greater insights. There is a need for fast and scalable tools for these stages in the pipeline.

Existing genomic tools are designed in an ad hoc manner, and are not easily amenable to automatic optimizations such as reduction in memory footprint or creation of concurrent tasks out of the overall application. The tools are written with the currently available data sizes in mind and consequently underperform due to the exponential growth in data. In addition, to obtain high performance, these tools require parallel implementations, adding to development complexity. In addition, different use cases for genomic applications necessitate the application designer to make sophisticated design decisions, which are difficult for a genomics researcher to grasp.

To solve this problem of developing performant and scalable applications, we take a holistic approach. We develop techniques at three levels - algorithm, compiler, and data structure - to enhance the performance and scalability of genomic applications handling large volumes of data. This is shown in Figure 1.1. At the algorithm level, we identify opportunities for exploiting parallelism and efficient methods of data distribution. Our technique Orion [4] exploits fine-grained parallelism to scale for long genomic sequences and achieves superior performance and better load balance than state-of-the-art distributed genomic sequence matching tools. ScalaDBG transforms the sequential and computationally intensive process of iterative de Bruijn graph construction to a parallel process. At the compiler level, we develop a domain-specific language, called SARVAVID [5]. SARVAVID provides commonly occurring modules in genomics applications as high-level lan-

guage constructs and performs domain-specific optimizations well beyond the scope of libraries and generic compilers. At the data structure level, we identify opportunities to exploit cache locality and software prefetching for enhancing the performance of indexing structures in genomic applications.

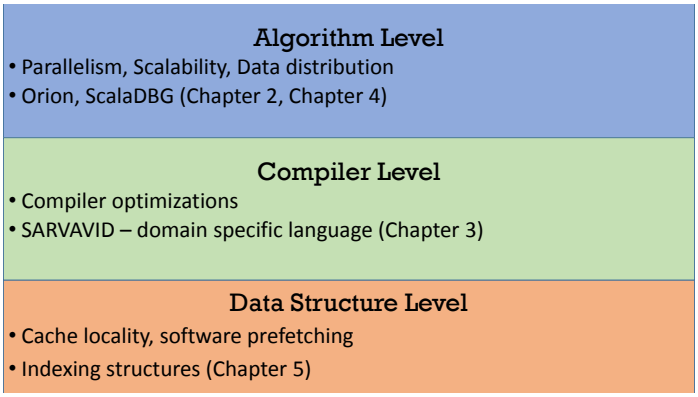

Fig. 1.1.: Techniques for enhancing performance of genomic applications

Three broad classes of genomics applications are local sequence alignment, whole genome alignment (also known as global alignment), and sequence assembly. Local sequence alignment finds regions of similarity between all sub-sequences of genomic sequences. Local alignment tools are generally used to find conserved genes between relatively distant organisms, such as human beings and fruit flies. Whole genome alignment finds mappings between entire genomes. Whole genome alignment tools are generally used to find variations between close organisms, such as between two human genomes. Sequence assembly aligns and merges the sequenced fragments of a genome to reconstruct the entire original genome. Applications falling in these three classes use different heuristics and steps in their workflow, due to the approximate nature of their task. They also use different data structures in their workflow. We apply our approach to these major classes of genomic applications and demonstrate the benefits with relevant genomic datasets.

I have made the following research contributions to tackle above issues :

1. Developed a fine-grained parallelism technique called Orion for sequence alignment. In the space of parallel genomic sequence search, most of the popular software pack-

ages, such as mpiBLAST, use the database segmentation approach, wherein the entire database is sharded and searched on different nodes. However this approach does not scale well with the increasing length of individual query sequences as well as the rapid growth in size of sequence databases. We have proposed a fine-grained parallelism technique called Orion, which divides the input query into an adaptive number of fragments and shards the database. Our technique achieves higher parallelism (and hence speedup) and load balancing than database sharding alone, while maintaining 100% accuracy. This technique is explained in detail in Chapter 2.

2. Developed a domain-specific language called SARVAVID, that provides commonly occurring modules in genomics as high -level language constructs, and a compiler that performs domain-specific optimizations

   We observe that computational genomics tools contain a recurring set of software modules, or kernels. The availability of efficient implementations of such kernels can improve programmer productivity, and provide effective scalability with growing data. To achieve this goal, we have developed a domain-specific language, called SARVAVID, which provides these kernels as language constructs. SARVAVID comes with a compiler that performs domain-specific optimizations, which are beyond the scope of libraries and generic compilers. Furthermore, SARVAVID inherently supports exploitation of parallelism across multiple nodes. SARVAVID beats handwritten implementations of well-known genomics applications in most cases, and is at least almost as fast. This DSL and the optimizations are explained in detail in Chapter 3.

3. Developed ScalaDBG a parallel assembler, which transforms the sequential, compute intensive process of de Bruijn graph construction for a range of k-values, to a parallel process.

   Among several approaches to assembly, the iterative de Bruijn graph assemblers, such as IDBA-UD, generate high-quality assemblies by sequentially iterating from small to large k-values used in graph construction, to solve the branching and frag-

mentation problem. However, this approach is time intensive because the creation of the graphs for increasing k-values proceeds sequentially. We develop a novel mechanism whereby the graph for the higher k-value can be "patched with contigs generated from the graph with the lower k-value. We show that our technique achieves higher parallelism and similar accuracy as IDBA-UD for the assembled genome, for a variety of datasets. Moreover, ScalaDBGs multi-level parallelism allows it to simultaneously scale up and scale out. This technique is explained in detail in Chapter 4.

4. Developed locality-aware indexing structures for sequence alignment

   Indexing structures such as hash tables, FM-index, and tree-based indices are widely used in a number of sequence alignment tools. However, index-search using FM-index and hash table has revealed highly irregular memory access patterns with low data locality and a high cache miss rate. In the absence of optimized structures and implementations for modern processors, these tools are underperforming. Furthermore, these tools are falling short of the throughput requirement imposed by the exponential growth in genomic data. We posit that locality-aware indexing structures are better suited for modern multi-core processors with significantly larger number of cores, memories, caches and compute power. To this end, we develop an optimized tree-based index and FM-index structure with greater opportunities to exploit data locality to boost search performance. This technique is explained in Chapter 5.

2. ORION: SCALING GENOMIC SEQUENCE MATCHING WITH FINE-GRAINED PARALLELIZATION

2.1 Introduction

One of the foundational building blocks of computational biology is *sequence alignment*, looking for similarities between particular DNA, RNA or protein sequences and a database of other sequences. Finding regions of similarity between target sequences and databases helps biologists understand structural, functional and evolutionary relationships between sequences to predict biological function of genes, find evolutionary distance between sequences and do genome assembly by finding common regions and repeats within a genome. For example, finding large overlaps between the DNA sequences of a newly discovered biological specimen (the *query*) and the DNA sequences of known organisms (the *database*) can highlight evolutionary relationships between the organisms.

The classic algorithm for performing sequence alignment, identifying matches between a query and a database of sequences, is the *Basic Local Alignment Search Tool* (BLAST) [6, 7]. BLAST operates by comparing each of the sequences in the input query set against each of the sequences in a database to identify *alignments* that partially or completely overlap. The more similarity there is, the higher the alignment's score. *E-value* ia numerical value that captures the likelihood that the similarity is statistically significant. Alignments with E-value below a certain threshold are output as potential matches by the algorithm. Section 2.2 describes the algorithm in more detail.

The National Center of Biotechnology Information (NCBI) provides public databases of gene sequences that researchers can search using BLAST. `http://blast.ncbi.nlm.nih.gov/` Unfortunately, the explosive growth in the number of biological sequences poses a formidable challenge to the current database searching algorithms. In December 2013, the GenBank database—hosted by NCBI—had about 170 million sequences, and the number

of bases has doubled approximately every 18 months [8, 9]. Given the exponential growth in the size of sequence databases, and the requirement to query longer sequences, current database searching algorithms struggle to provide the alignment and search results in a timely manner. Early parallel BLAST implementations [10, 11] exploited coarse-grained parallelism: individual queries can be processed simultaneously against the same database. However, while such parallelism improves throughput, it does not help an individual researcher with a single query: For example, a BLAST job with a query sequence of 100,000 contiguous fragments (i.e., contigs or overlapping sequenced data reads) BLASTed against the non-redundant (NR) nucleotide database could take 70 days [12]! To provide genomics researchers with reasonable latency for their searches, exploiting additional parallelism has become a necessity.

The most popular open source parallelization of BLAST is mpiBLAST, using, unsurprisingly, MPI to run BLAST in parallel on clusters [13]. mpiBLAST adopts a natural parallelization strategy. Because BLAST compares the input query against each sequence in the database separately, parallelism can be exploited by performing multiple such comparisons concurrently. mpiBLAST thus *shards* the database into multiple pieces each containing a subset of the databases's sequences and distributes the shards across the computational nodes in the cluster. These shards can then be searched independently and simultaneously for alignments with the input query.

Unfortunately, while mpiBLAST can exploit parallelism by sharding large databases, and even by processing multiple input queries in parallel, it has significant limitations for many biological use cases. In *long sequence alignment*, a long input query is matched against a database. Such use cases are becoming increasingly common. With the rapid expansion of next generation sequencing technologies, the number of organisms whose entire genomes are being sequenced has been growing at a rapid pace. Once a genome is sequenced, it is annotated, which involves (among other processes) comparing the newly-sequenced genome, or parts thereof, with that of a closely-related organism or with the expansive NT database, to establish the evolutionary relations of this newly-sequenced

organism. This results in large queries, with the upper bound being the size of the entire genome, which can be millions of nucleotides.

In this scenario, mpiBLAST runs out of parallelization opportunities. There is but one input sequence, so parallelism by processing multiple queries simultaneously is impossible. And increasing the number of database shards to increase parallelism suffers from diminishing returns: even if the database contains enough sequences to profitably create additional shards, additional shards increase scheduling overhead as well as the time required to aggregate the output from each query-shard work unit.

Moreover, mpiBLAST's parallelization strategy can lead to severe load imbalance with large queries, or with queries of very different sizes. If a query sequence is long, or has many matches with a particular database sequence, it will take a long time to process, while a short query sequence, or one with little similarity to a database sequence can be completed much faster. As a result, the execution time of different query-shard work units can vary significantly, a problem that is only exacerbated as queries get longer [14, 15]. Further, it is difficult to predict what the running time for a unit of work will be from simple metrics as the length of the query [15]. Consequently, the static load balancing approach of mpiBLAST tends to create severe load imbalances among the different nodes processing different work units, as we experimentally show in our evaluation.

To address these concerns, we propose *Orion*, a new parallel BLAST implementation that exploits finer-grained parallelism than mpiBLAST, achieving both more parallelism in the face of long sequences as well as better load balance. The key insight behind Orion is that a single, long query sequence need not be matched against a database sequence serially; instead, the query can be *fragmented* into sub-queries (which we call "query fragments"), each of which can be matched against the database independently and in parallel. Figure 2.1 captures the various levels of parallelism inherent in sequence alignment. The early approaches to sequence alignment primarily targeted the lowest level, processing multiple queries in parallel against the entire database, while mpiBLAST exploits the two lowest levels, processing the same query against different database shards simultaneously. Orion exploits all levels of parallelism: inter-query, intra-database, and *intra-query*.

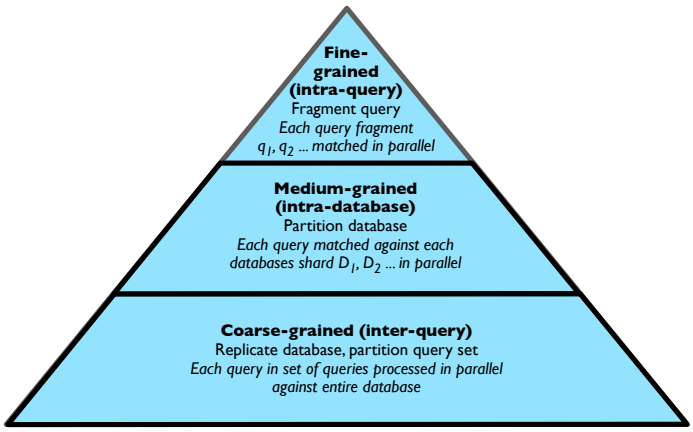

Fig. 2.1.: Parallelism in genomic sequence search. Our solution Orion is the first to exploit opportunity for parallelism at all three levels. mpiBLAST, for example, only uses the lower two levels.

Query fragmentation itself is not a new strategy in the BLAST community. It first arose in recent work that noted that BLAST's performance is severely degraded by cache misses as query size grows, and proposed query fragmentation as a solution [16, 17]. Such strategies either require access to the entire query to compute alignments [16], or require that the query fragments overlap by a substantial amount to avoid missing alignments [17], obviating parallelism or necessitating substantial extra work.

In contrast, in Orion, we limit the size of the overlap by querying the input parameters such as the thresholds in the BLAST algorithm and the penalties due to a mismatch in BLAST, and employ a novel extension and aggregation strategy to avoid missing alignments. My fragmenting strategy is such that practically there is no loss in accuracy, *i.e.* *every* sequence that will be matched successfully in BLAST will also be matched successfully in Orion. However, the overlaps are not so large as to eliminate the scope for intra-query parallelism.

We introduce three chief novelties:

1. We develop an analytical model based on BLAST's scoring formula that identifies the optimal fragmentation strategy, avoiding redundant work.

2. We introduce a speculative extension strategy that allows alignments that may cross query fragment boundaries to be identified.

3. We build an aggregation algorithm that combines full and partial alignments from each fragment to generate a final set of alignments that matches the original sequential algorithm.

We parallelize and implement the algorithm using the Hadoop MapReduce framework, and demonstrate that the algorithm yields better parallelization, performance and load balance than mpiBLAST, while producing the same results. The software package is available through `https://github.com/purdue-dcsl/Orion`

Outline Section 2.2 describes the basic BLAST algorithm, as well as mpiBLAST's parallelization strategy. Section 2.3 details the design of Orion's fragmentation and aggregation algorithms. Section 2.4 discusses the Hadoop implementation of Orion. Section 2.5 compares Orion to both sequential BLAST and mpiBLAST. Section 2.6 surveys related work, while Section 2.7 concludes.

2.2 Background

This section provides background on the general concepts of sequence alignment; BLAST, the most popular algorithm for performing sequence alignment; and mpiBLAST, the most common parallel implementation of BLAST.

2.2.1 Sequence alignment

Sequence search typically examines one or more *query sequences*, Q, against a *database* of reference sequences, D. The sequences might be nucleotide sequences (*e.g.*, genomes of organisms) or peptide sequences (the chains of amino acids that make up a protein). We will focus on nucleotide sequences for the remainder of the document. Each query sequence $q \in Q$ is compared against each database sequence $d \in D$ to determine their similarity. Similarity is determined by looking for long subsequences that are common to both

d and q. High similarity between nucleotide sequences indicate that the same gene might exist in both sequences, or that both sequences have similar biological function. Similarly, regions of little similarity between sequences might indicate that such regions do not have any biological importance (they are "junk DNA").

A nucleotide sequence is represented by a string of bases drawn from $\{A, C, G, T\}$, so finding common sequences between two such strings seems like it can be solved using traditional string-matching algorithms. However, because genomes are constantly mutating, it is often useful to look not for *exact* matches, but merely *good* matches between sequences. Common alterations to genomic sequences include changes of a single base, leading to a *mismatch* between sequences, and insertion or deletion of a single base, leading to a *gap* between sequences. Hence, alignment must consider several scenarios when looking for a good match. Consider the query sequence $q = CACTTGA$ shown in Figure 2.2. There are several possible database sequences that could "match" q, once mismatches and insertions and deletions of bases from the query are taken into account.

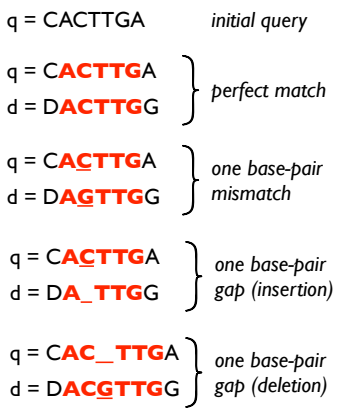

Fig. 2.2.: Query sequence and possible matching database sequences. Matching alignments are shown in bold, red text. Mismatches and gaps (both inserted and deleted bases) are underlined. In the second alignment, a possible match is found by positing that a nucleotide was altered in the database sequence to produce the query sequence. In the third alignment, a possible match is found by positing that a nucleotide was *inserted* into the database sequence to produce the query sequence, while in the fourth alignment, a possible match is found by positing that a nucleotide was *removed* from the database sequence.

Each of the database sequences in Figure 2.2 represent a possible alignment; the only difference is in the "score" given to the alignment: fewer mismatches or gaps produce a higher score. Nevertheless, having a mismatch or gap does not disqualify a particular match: a long alignment with one or two mismatches can produce a higher score than a short alignment with no mismatches. The classic dynamic programming algorithm for computing alignments with gaps and mismatches is Smith-Waterman [18].

2.2.2 BLAST

The basic Smith-Waterman algorithm suffices to find alignments, but it is slow ($O(mn)$ time to find alignments between sequences of length m and n) and has high space overhead ($O(mn)$ space to store the scores in the dynamic programming matrix). Altschul *et al.* designed the Basic Local Alignment Search Tool (BLAST) to perform faster alignments, at the cost of accuracy (potentially missing some alignments) [6]. While the details of BLAST are quite complex, here we provide a high level intuition of BLAST's operation. We describe BLAST in terms of a single query q and database sequence d, though the algorithm ultimately operates on sets of both.

BLAST has three phases: (i) the *k-mer match* phase; (ii) the *ungapped alignment* phase; (iii) the *gapped alignment* phase. In all three phases, BLAST relies on a scoring function that provides a numerical score for the current proposed alignment. In the first phase, BLAST considers every k-length subsequence (called *k-mers*) of q and d and looks for k-mers that appear in both.[1] This step is performed efficiently by creating a lookup table with all k-letter words in q. The algorithm then walks through d and uses the lookup table to see if a k-length subsequence of d matches any part of q. These matches are *seeds* of potential alignments.[2]

[1]When performing nucleotide (DNA or RNA) alignment, only exact k-mer matches are identified; when performing protein alignment, partial matches can be found, with scores based on the particular peptides matched.

[2]Note that this is the phase where inaccuracy relative to Smith-Waterman is introduced, as alignments that do not have a k-mer seed will be missed.

In the second phase, ungapped alignment, each seed is *extended* both to the left and right allowing both perfect matches (corresponding nucleotides in q and d) and mismatches (different nucleotides in q and d). While perfect matches increase the score of the potential alignment, mismatches decrease the score. BLAST tracks the current score of the alignment, s, and the maximum score seen so far for the current seed, s_{max}. If $s_{max} - s$ is greater than some threshold t_x (called the *X-drop* threshold), the second phase terminates, returning the alignment with the peak score for the current seed. If the returned alignment's score s is greater than some threshold t_u (which we call the *ungapped threshold*), the alignment is passed to phase three. As an optimization, if a seed is contained within a previously-found alignment, the seed can be skipped.

In phase three, gapped alignment is performed. The ungapped alignment is extended in both directions, this time allowing insertions and deletions to occur as the alignment is extended. As in the second phase, the maximum score of the alignment s_{max} is tracked, and if the current score s drops below s_{max} by more than t_x, the phase is terminated and the resulting alignment is returned.

Table 2.1.: Parameters and default values for BLAST. There are two default x-drop values, the first for ungapped alignment and the second for gapped alignment. There is no default value for t_u, as the score threshold for significance is dependent on query and database sequence length.

Parameter	**Description**	**Default value**
k	Length of initial seeds	11
t_x	X-drop value	20, 15
t_u	Ungapped alignment threshold	N/A
E	Final reporting threshold	10

After each seed is processed, all the alignments that score above a threshold of statistical significance (called the *E-value*) are sorted and returned to the user. Numerically, the lower the E-value is the better the match is, *i.e.* lesser is the chance that the alignment happened purely by chance. Therefore, if the calculated E-value is *less than* the E-value threshold, is the alignment output to the end user. Table 2.1 summarizes the parameters used in BLAST.

2.2.3 mpiBLAST

Figure 2.1 shows the types of parallelism that arise in BLAST. Most early attempts to parallelize BLAST exploited the coarsest granularity of parallelism: each query q in the set of queries Q is processed independently. The database D of sequences is replicated on each compute node, and queries are then processed simultaneously on each node [10, 11]. Later approaches adopt a more aggressive, finer-grained parallelization strategy: in addition to partitioning the query set Q into individual queries $Q_1, Q_2, \ldots$, the database is partitioned into subsets $D_1, D_2, \ldots$. For clarity, we will refer to partitioning the query set as *segmenting* the query set, and partitioning the database as *sharding* the database. Each pair (Q_i, D_j) represents a work unit, applying one query segment against one database shard. The work units can be processed in parallel, with the results from each query aggregated later. Perhaps the best-known example of this parallelization strategy is mpiBLAST [13].

mpiBLAST follows the master-worker paradigm. Before alignment can start, the master shards the database into disjoint partitions of approximately equal size and places them in shared storage. The master uses a greedy algorithm to assign unprocessed database shards to its workers. Query segments are then handed to each worker. A worker executes the basic BLAST algorithm for the query segment on its database shard(s) and sends the results back to the master. The master ensures that every query segment is processed against every database shard, and also aggregates the results for each query, performing the final sorting to present the queries' alignments. mpiBLAST achieves parallelism by segmenting the queries and sharding the database, and in addition improves performance relative to non-sharing implementations by choosing shard sizes so that each shard fits in a worker node's main memory.

mpiBLAST works well when Q contains many short sequences and D is large, affording it opportunities both to create sufficient parallelism and to provide load balance (by generating far more work units than worker processes). However, in many biological settings, these assumptions do not hold true. For example, it is common to match a single, large query sequence against a small database (*e.g.*, matching a long human DNA

sequence against a database containing genomes for each human chromosome). In such settings, mpiBLAST cannot generate enough work to provide parallelism and load balance. Even if the database is large enough to shard, long queries lead to more variable runtime [14, 15], creating load imbalance problems. Moreover, because mpiBLAST relies on the basic BLAST algorithm at each worker, it suffers from poor performance in the face of long queries [16].

We studied the scalability of mpiBLAST, in terms of length of query sequences handled by performing experiments to search Human genes from the NCBI gene database(`http://www.ncbi.nlm.nih.gov`) over the *Drosophila melanogaster* database. The sequences ranged from 3000bp to 99Megabp (base pairs) in length. We used a small test cluster of 4 nodes and 64 cores to do the experiments. To enable mpiBLAST to fully exploit available parallelism, we made 64 shards of the database. Figure 2.3 shows that performance of mpiBLAST is good at query sequences of length less than 1 Mbp, but starts to worsen at a threshold of 1Mbp. The performance worsens rapidly beyond this threshold of 1Mbp, reaffirming the poor performance of mpiBLAST in the face of long queries as mentioned above.

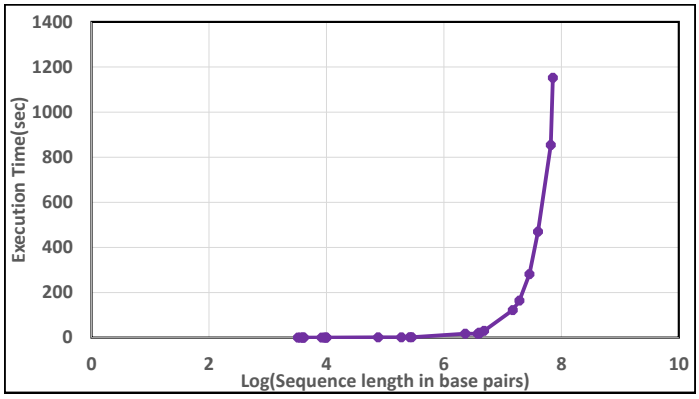

Fig. 2.3.: mpiBLAST behaviour for long sequences

In the next section, we discuss our design of a new parallel BLAST implementation that provides parallelism and load balance even for large queries.

2.3 Design of Orion

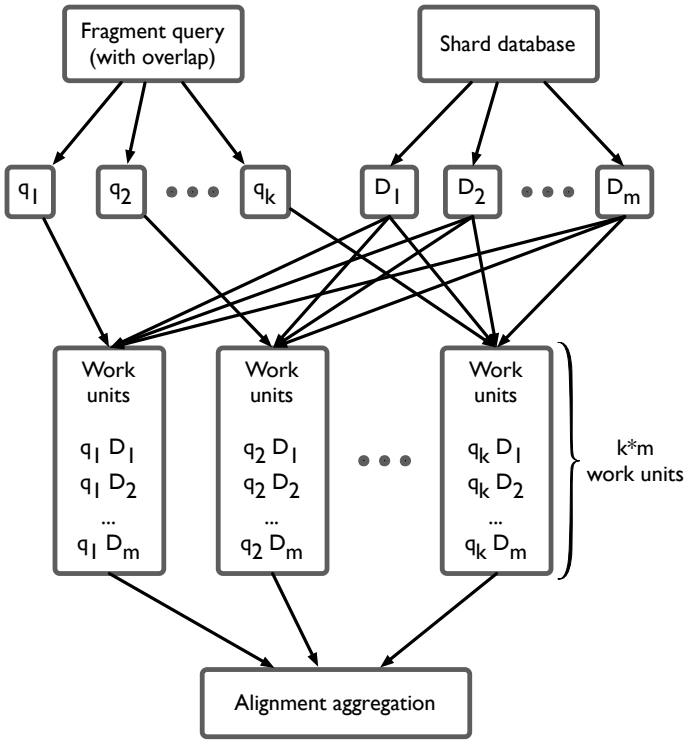

Fig. 2.4.: High Level Architecture of Orion

This section discusses the design of Orion. Implementation-specific details are discussed in Section 2.4. The high-level architecture of Orion is shown in Figure 2.4.

2.3.1 Query fragmentation

As introduced earlier, Orion uses as a fundamental strategy, the fragmentation of a query and matching the fragments in parallel. Continuing with the notation from Section 2.2, we have a query set Q, which comprises individual queries $Q_1, Q_2, \cdots, Q_m$. The entire database is D and it is sharded into disjoint shards $D_1, D_2, \cdots, D_n$. Further, Orion fragments each query Q_i into fragments $Q_{i1}, Q_{i2}, \cdots, Q_{ik}$. Our design creates equal-sized query fragments, by determining the optimal fragment size.

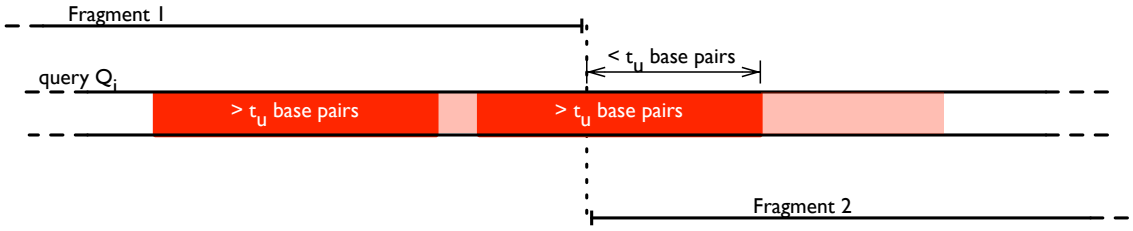

Fig. 2.5.: Example alignment that spans two disjoint query fragments. The alignment is shaded, while the darker shaded regions represent ungapped sub-alignments that would be reported as part of Phase ii of BLAST.

A simple approach to query fragmentation is as follows: for a given query Q_i,match each query fragment against each database shard in parallel, using baseline, sequential BLAST. After all fragments of Q_i have been matched against each database sequence, aggregate the results, combining alignments from neighboring fragments that can be concatenated to form a larger alignment, and report them. Unfortunately, this simple strategy, which assumes that query fragments are independent, is incorrect; if an alignment spans two query fragments, then the *portion* of the alignment that lies in each fragment may not have a high enough score to be reported.

Consider Figure 2.5. It shows an alignment that spans two query fragments with no overlap. The shaded(dark and light) portion of the query represents the alignment that should be reported, while the darker shaded portion represents ungapped sub-alignments that exceed the threshold t_u, introduced in Section 2.2 (it is the number of base pairs that produce a long enough alignment to pass the score threshold). While the search over fragment 1 will return a partial alignment, triggered by the first ungapped sub-alignment, the search over fragment 2 will not return any alignments: the portion of the final alignment that lies in fragment 2 does not have any sufficiently-long ungapped alignments to pass the threshold in phase ii of BLAST.

This situation is not a corner case, rather it is quite common in practice, with the likelihood increasing with decreasing size of each query fragment. The choice of short query fragments is of course appealing from the point of view of increasing the number of work

units and the degree of parallelism. Note, also, that this issue applies not only to the t_u threshold, but also to the two other thresholds in BLAST: the initial k-mer threshold (if a k-mer spans two fragments, it will never be discovered) and the final E-value thresholds. In general, if the overall alignment passes a threshold, but the sub-alignments found on each fragment do not, the alignment will be missed.

Fragment overlap

To overcome the missed alignment problem described above, Orion uses a combination of *overlapping query fragments* and *alignment aggregation*. To see why overlapping fragments can be useful, consider overlapping neighboring query fragments by k nucleotides. By doing so, it is no longer possible to miss a k-mer match. Intuitively, the overlap should be large enough such that the following condition holds.

> If there is a matching sequence between the query and the database, then the partial matches within each query fragment should be able to pass each of the thresholds of the three phases.

How large is large enough will depend on various factors — the lengths of the query and of the database, the thresholds for ungapped and gapped alignments, the E-value threshold, and the word size for the initial k-mer matches. Now there is a downward pressure on the size of overlap. Too much overlap will mean the work of matching will be duplicated in nodes that are processing adjacent query fragments. Some earlier, non-parallel implementations of BLAST have suggested overlapping queries, but typically choose extremely large overlap values to avoid missing alignments [19]

Orion chooses the overlap to be t_u, and can find the whole alignment of Figure 2.5 Fragment 1 sees a partial alignment and Fragment 2 sees a partial alignment, and there is no longer any way to miss any sub-alignments, as shown in Figure 2.6

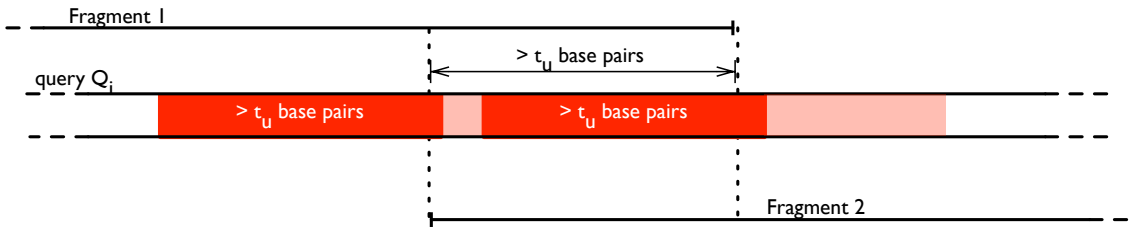

Fig. 2.6.: Alignment with sufficient overlap

2.3.2 Alignment aggregation

In Orion, rather than adopting an *ad hoc* approach to fragment overlap, we use a more disciplined strategy. In particular, note that we can introduce an additional *alignment aggregation* phase to the search process. As Orion processes a single query fragment, if an alignment does not hit a query boundary (*i.e.* the entire alignment fits in a single fragment), it is returned as normal. But if a partial alignment *does* hit a fragment boundary, it *may* be part of a larger alignment that spans two fragments. Hence, Orion returns these alignments as well.

After all of the query fragments have been processed, Orion performs alignment aggregation. Any alignments that lie entirely within a single fragment can be returned as is (note that alignments that lie entirely within the overlap between two fragments will be returned by both fragments). However, any alignments that hit query boundaries must be combined with alignments from the other side of the boundary. Orion "undoes" the overlap between the alignments, merges them together and then reports the result only if the *combined* alignment passes all the score thresholds.

Speculative extension

For the reduction phase to work properly, if a partial alignment hits the fragment boundary, Orion must perform gapped extension even if the partial alignment doesn't meet the ungapped alignment threshold. To see why this is necessary, consider the alignment in Figure 2.7.

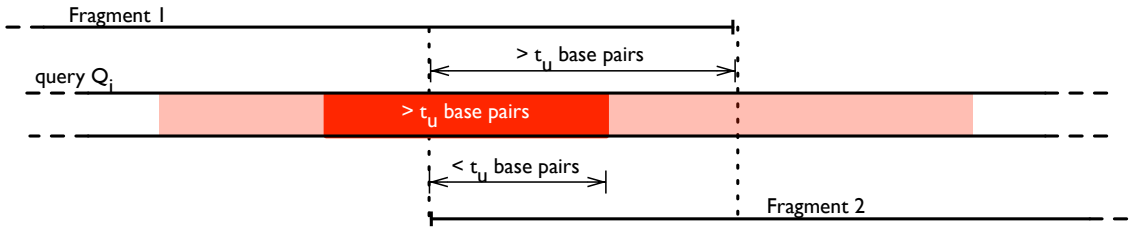

Fig. 2.7.: Alignment with fragment overlap. Fragment 2 must perform gapped extension despite not seeing a high-scoring alignment.

The alignment contains a single ungapped subalignment that exceeds the threshold. This subalignment falls entirely within fragment 1, so fragment 1 proceeds with gapped alignment, finding the lightly shaded portions of the alignment. However, fragment 2 *does not* see enough of the ungapped alignment to trigger gapped extension, and hence the portion of the alignment that lies only in fragment 2 would be missed.

To avoid this problem, Orion performs gapped extension *speculatively*: fragment 2 performs gapped extension for its partial alignment anyway. Because the actual score of the ungapped alignment is not known (as it lies partially in fragment 1), Orion uses a relative scoring metric. Rather than extending the alignment until the score drops to t_x below the maximum score seen so far, Orion starts the scoring at 0, and extends the alignment until the score drops to $-t_x$. This results in slightly longer gapped extensions, but the excess is cleaned up during alignment aggregation.

We note, also, that fragment overlap plays a role in speculative extension. If Orion performs an extension, speculative or otherwise, of a partial alignment that hits a fragment boundary, and the extension is terminated (due to X-dropoff) *within the overlap region*, then the partial alignment does not need to be returned, as the neighboring fragment will be able to see the entire alignment (consider if the lightly shaded portion on the right side of Figure 2.7 did not exist; Fragment 1 would see the entire alignment).

Possible missed alignments

There is one corner case where Orion will miss a query alignment that the baseline BLAST would have found. Such a miss happens due to the query fragmentation of Orion, and despite the overlaps in the query fragments. The inaccuracy arises in the case where an alignment spans two fragments, but the portion of the alignment that lies in one fragment does not contain any k-mer matches. In this case, that fragment will not even initiate the search for an alignment. We expect this case to be extremely rare in practice. Experimentally we find that such a miss never happens in our evaluation, and thus we achieve accuracy of 100%.

2.3.3 Calculating overlap length

So the question arises what should be the ideal overlap length. The overlap must be at least k: smaller overlaps may result in k-mer hits being missed. Increasing overlap length beyond k makes extensions more likely to terminate within fragment boundaries, resulting in less work during alignment aggregation. Nevertheless, making overlaps too large results in redundant work during the search phase.

We choose our overlap size with these criteria in mind. In particular, we choose our overlap size to ensure that ungapped alignments that pass the t_u threshold lie within each fragment. According to [20], the expected value (E-value) of a single distinct alignment may be calculated by the formula

$$E = Kmne^{-\lambda S}$$

where, K and λ are Karlin-Altschul parameters, m and n are the effective lengths of the query sequence and database, respectively, and S is the alignment score. The "effective" lengths are shorter than the actual lengths to account for the fact that an optimal alignment is less likely to start near the edge of a sequence than it is to start away from that edge. We want to calculate the smallest value of S that will cause the calculated E-score to be less than the threshold E-value (the notation we have used for the latter is simply E).

Putting these constraints together (detailed derivation follows that in [21]), we derive the following formula for fragment overlap (L).

$$S_{lb} = \lceil \frac{ln(Kmn/E_{th})}{\lambda} \rceil$$

$$L = max(k, S_{lb}/p) \tag{2.1}$$

where, k is the word size of the initial k-mer match, S_{lb} is the shortest ungapped alignment that still passes the E-value test (*i.e.* calculated E score is exactly equal to E-value and any shorter ungapped alignment will not pass this test). This S_{lb} is then divided by the reward for match of one single bp, p, to come up with the length of the overlap (in terms of bp). To account for the degenerate case where the calculated value of S_{lb}/p is smaller than the length of the initial k-mer match, the max is taken in the final calculation of L.

This choice of L guarantees the following property. Consider two adjacent fragments F_1 and F_2 (Figure 2.5 or Figure 2.7 may serve as a reference). If in the baseline (unfragmented) query, there is a sequence with enough of a match with the database such that $E_{calculated} \leq E$, then, there is enough overlap between F_1 and F_2 such that there will be a sub-sequence in either F_1 or F_2 that will give $E_{calculated}$(sub-sequence) $\leq E$.

2.3.4 Threshold for fragment size

Intuitively, it seems clear that Orion should not fragment a query that is smaller than a certain size. This is due to the fact that there is a certain overhead of fragmentation—divide the query up, send each query fragment to a separate node, and after the parallel matches, aggregate the results of the individual matches to create the final output. These costs must be balanced against the additional scope for parallelization, and (to a second order effect) better load balancing, that results from fragmenting the query. Further, there is a constant cost of running the baseline sequential BLAST.

Orion takes these two factors into account to select a desired query fragment length. The desired query fragment length depends on *both the database and the exact query* simply

because the amount of work that is to be done depends on these two elements. However, for the purpose of calibration, it is clearly infeasible for Orion to determine this desired query fragment length for *every* query for each database it is to be run against. Therefore, we make the practical simplification of performing this calibration once for each database that the matching is going to be performed against. We find that experimentally this simplification is justified with little performance degradation compared to the ideal design choice.

2.4 Implementation

In this section we describe the implementation of Orion.

2.4.1 Sharding the database and fragmenting the query

Orion uses mpiBLAST's `mpiformatdb` tool to format and to shard the database. It divides the database into a specified number of shards, which are approximately equal in size and are then placed on shared storage.

To fragment the query, Orion uses a simple preprocessing step that takes as input the database length, the original query sequence and the desired fragment length of each query. Orion then calculates the overlap length using Equation 2.1, fragments the input query sequence using the fragment length and overlap length parameters, and places the fragmented query sequence on shared storage.

2.4.2 Parallel BLAST search

Orion's parallel BLAST search on each fragment/shard work unit naturally fits into the MapReduce paradigm [22], with each of the fragment/shard search tasks as a "map" task. We use Hadoop streaming to implement the map phase of the parallel blast search. The map tasks run NCBI blastall for every fragment/shard pair with the specified arguments for the program, the database shard, and the query. The outputs are the parsed BLAST results for search of the query over the respective database shard. The parsed output of BLAST

search reports for each alignment the identifier for the database sequence, the offsets of the alignment in the database, the length of the database sequence, the query fragment identifier, query fragment length, offsets of alignment in the query fragment, the sense of the alignment, the E-value, and the number and location of matches, mismatches, and gaps. This information resides in files stored on HDFS. The identifier for the database sequence as the key and the alignment information as the value is fed to the reduce phase.

2.4.3 Aggregation of results

The aggregation phase is the Reduce phase of Orion's Map-Reduce job. It is required to merge overlapping alignments that cross over fragment boundaries and present the alignments as a single alignment as would have been reported by BLAST. The key is the database sequence identifier which divides the space of alignments results. In simple words, it first collects all alignments from all the query fragments that matched a particular database sequence together. It then finds overlapping or adjacent alignments from this set and aggregates them. Finally the set contains all aggregated alignments. The benefit of choosing sequence identifier from the database as the key is that multiple reducers can work in parallel over different database sequences.

2.4.4 Sorting of results to create final output

Orion outputs alignment results in decreasing order of their scores or increasing E-value. Orion samples the score data for a rough approximation of the distribution of the score values, and then different ranges of values are assigned to different reducers to sort in parallel. Finally the merge is done in parallel, since the range of score values for each reducer task is known. The result is the final set of alignments sorted according to E-values, exactly what would be returned by (serial) BLAST.

2.5 Evaluation

In this section we present a performance evaluation of Orion on the Gordon supercomputing system. We first compare the execution times of Orion and mpiBLAST, the most popular open-source parallel implementation of BLAST. We then compare the scalability and the effectiveness of load balancing of the two solutions. We also evaluate the overall speedup for Orion, and do a sensitivity study to determine the relationship between query fragment length and execution time for Orion. We use a biologically relevant comparative genomics problem which searches queries from the human genome over the *Drosophila melanogaster* database, to validate that Orion has performance gains in realistic scenarios, as we detail in Section 2.5.2.

2.5.1 Experimental Setup

We used two cluster to perform the experiments. Our test cluster consisted of 14 nodes, each having two quad-core AMD Opteron 2354 1.1MHz processors and 8 GB of memory. We alse used the Gordon supercomputing system to run our experiments. Gordon is a dedicated XSEDE cluster maintained by the San Diego Supercomputer Center. Each compute node contains two 8-core 2.6 GHz Intel EM64T Xeon E5 (Sandy Bridge) processors and 64 GB of DDR3-1333 memory. We used a cluster of 64 such nodes, each node having 16 cores.

In these experiments the internal BLAST implementation for both mpiBLAST and Orion used default values for E-value, match rewards, mismatches and gap penalty, and the drop off values and all other configurable parameters (see Table 2.1). The overlap length was calculated using Equation 2.1. The relevant parameters for the overlap equation are given in Table 2.2.

We used Hadoop version 1.1.1 and mpiBLAST's latest version-1.6.0 in the experiments. The Hadoop cluster was setup such that one node acted as both the master node and the slave node. All the other nodes were configured as slave nodes. The master node in the Hadoop cluster assumes the role of namenode, secondary namenode and jobtracker. The

Table 2.2.: Parameters required to calculate overlap length

Parameter	Value
Length of Drosophilia database	122,653,977
k	0.711
λ	1.374

slave nodes act as datanodes and tasktrackers. All the nodes in the cluster act as both storage and compute nodes. Each node was configured to run a maximum of 16 map and reduce tasks concurrently, to match the number of cores on the nodes.

2.5.2 Biological relevance of evaluation strategy

With the availability of whole-genome sequences for an increasing number of species, we are now faced with the challenge of decoding the information in these sequences. Comparative genome sequence analysis for multiple species at varying evolutionary distances, often termed phylogenetic footprinting, is a powerful approach for identifying protein coding and functional noncoding sequences. Drosophila or fruit fly has been valuable as a model organism for studying human behavior, development, and diseases, given the parallels between the genomes of humans and these tiny flies. In addition, their short life spans and prolific breeding allows for quick turnaround of large-scale biological experiments. Comparison of the Drosophila genome with the human genome, for example, revealed that approximately 75% of human disease genes have homologs in Drosophila [23]. Motivated by this, in this paper we have used Drosophila as a model reference genomic database for aligning a set of long genomic scaffolds of human chromosomes; scaffolds are assemblies of contigs and gaps reconstructed from the NGS reads. The final goal of the genomic comparisons, as done in this paper, would be to explore the evolutionarily-conserved sequences from Drosophila to humans. For example, ultra-conserved elements (UCEs) are arguably the most constrained sequences in the human genome and the majority of these are outside the protein-coding regions [24]. Thus, one exciting use case for such rapid comparisons of long human chromosomal sequences with other databases (e.g., Drosophila database),

at different evolutionary distances, could be to discover new UCEs present across varying evolutionary distances. Interestingly, single nucleotide polymorphisms (SNPs) in UCEs have been linked to cancer risk, impaired transcription factor binding, and homeobox gene regulation in the central nervous system [25]. Our future efforts will be directed at aligning long or complete cancer genome sequences, from databases such as the Cancer Genome Atlas Network [26], with normal genome sequences to detect the altered sequences driving different types of cancer.

2.5.3 Comparison of Execution Times

In this section we compared the time to completion of a query set for Orion and mpiBLAST. We used human chromosome contigs as our query sequences, and the *Drosophila melanogaster* representing the fruit fly genome as our database. The Drosophila database has an unformatted size of 118MB database and contains 1170 sequences. All the databases were taken from NCBI. Contigs are contiguous sequences that form part of the organism's genome after cleanup has been performed on the raw NGS instrument reads.

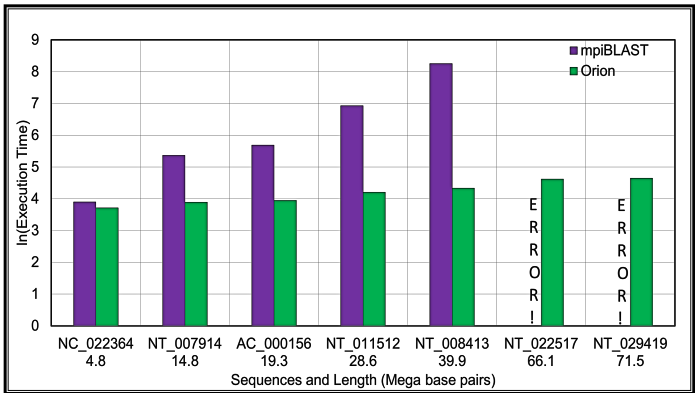

Fig. 2.8.: Execution time comparison of individual queries for Orion and mpiBLAST on test cluster

We first measured the time to completion of the individual queries on the test cluster. This is shown in Figure2.8. For all the sequences, Orion is much faster than mpiBLAST, and the benefit increases with the length of the query sequence. For the query sequence

NT_008413 Orion is 50X faster than mpiBLAST. Also, mpiBLAST ran out of memory, and could not handle sequences longer than 66.1 Mbp on these machines.

We then performed another experiment on the larger cluster of 64 nodes. Orion is aimed at solving the problem of delivering an efficient and low latency genomic sequence search system for long sequences. To validate this we choose a query set that consists of 16 sequences which are genomic contigs and scaffolds randomly selected from different human chromosomes. The query sizes range from 1 Mbp (Mbp=10^6 base pairs) to 71 Mbp. mpiBLAST performance is sensitive to the number of database shards used, and Orion performance too is sensitive to the number of database shards and query fragments. Hence the number of shards chosen for Orion and mpiBLAST, and the fragment size chosen for Orion were such that both Orion and mpiBLAST have optimal performance for the specific configuration of the experimental machine. We performed the experiment by varying the number of cores in the cluster to study the scalability of Orion and mpiBLAST.

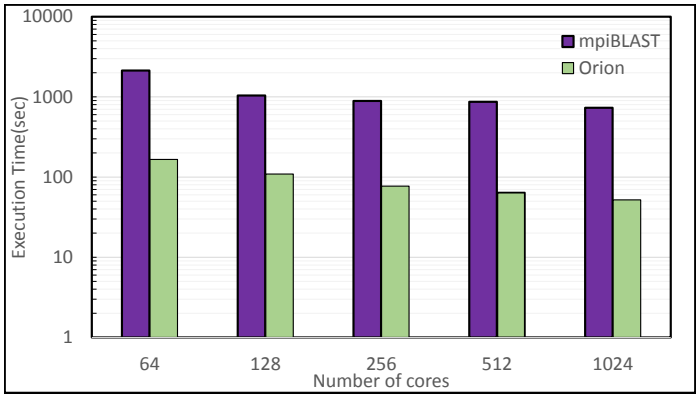

Fig. 2.9.: Execution time comparison of query set for Orion and mpiBLAST on Gordon cluster

Figure 2.9 shows the performance of of mpiBLAST and Orion for the chosen query set. Note the logarithmic scale on the Y-axis. From the figure we see that the performance of Orion is significantly better than mpiBLAST at all configurations of number of cores in the system. As expected, as the number of cores increase the execution time goes down for

both Orion and mpiBLAST. However Orion performs about 12.3X better on an average for the chosen query set.

Now, looking at the performance on individual query sequences within the query set, we noted that Orion is 23X faster than mpiBLAST for the longest (71 Mbp) of the query sequences. we also noted that the gain of Orion over mpiBLAST increases with increase in query sequence length. Further, mpiBLAST could not handle sequences longer than 96 Mbp and terminated with an error message complaining that it required about 2178 Gb of memory for dynamic programming! The vast majority of the human chromosomes are longer than 96 Mbp and thus, with the current state-of-the-art, we would not be able to run a parallel sequence matching for this wide variety of genomic sequences.

It should be noted that while Orion achieved superior performance for the longer queries, it did not miss any alignments reported by mpiBlast, which is the same as alignments reported by BLAST. Thus, the accuracy of Orion remained at *100% for all the query sequences*.

2.5.4 Load Balancing

mpiBLAST's parallelization strategy of segmenting the input query set into individual queries can lead to severe load imbalance among the worker processes. Thus, some processes do the bulk of the work and a majority of the processes terminate quickly. In contrast Orion's query fragmentation strategy divides the entire work into smaller work units, each of which is handled by a Hadoop task. This reduces the variability in load distribution, and enables greater predictability in execution times of each work unit. This ultimately leads to a more efficient use of system resources.

We validate this by comparing the search times of mpiBLAST's processes and Orion's Map and reduce tasks' run times in the 256 core configuration of Experiment 1. Since the running times of the tasks in Orion and the processes in mpiBLAST are not comparable, we use *Coefficient of Variation* (CV) to measure the variability in the run times. CV is defined as Mean/Standard Deviation. Table 2.3 shows that CV for mpiBLAST processes'

run times is higher than Orion's. This shows that Orion achieves better load balance than mpiBLAST.

Table 2.3.: Average, standard deviation (in seconds) and coefficient of variation for processes in mpiBLAST and Map and Reduce Tasks in Orion

Metric	**mpiBLAST**	**Orion**
Average (s)	315.78	2.10
Standard Deviation (s)	182.18	0.25
Coefficient of Variation	0.58	0.24

2.5.5 Scalability Tests

To evaluate the scalability of Orion, we run and profile a sequence search job with long queries over the Drosophila database. The sequences used here are even bigger than the ones used in Experiment 1, we used 32 sequences in the range of 1Mbp-99Mbp, and thus well beyond the usable range of mpiBLAST.

We increase the number of cores in the system from 4 nodes (64 cores) to 64 nodes (1024 cores) and measure the speedup achieved as illustrated in Figure 2.10. As can be seen, Orion scales to 1024 cores at a nearly constant parallel efficiency, *i.e.,* the slope of the speedup curve is almost constant. At 1024 cores, Orion achieves a speedup of 5 times the baseline of 64 cores. This speedup demonstrates that Orion can fully leverage the massive parallelism of today's supercomputing systems while solving important biological problems.

2.5.6 Comparison with Blast+

In this experiment we compared the performance of Orion and BLAST+. BLAST+ is a new suite of BLAST tools that runs on the NCBI servers. It is interesting to compare Orion and BLAST+ since BLAST+ also performs what they call "query splitting" to address the failure of BLAST to run long sequences [16]. BLAST+ is designed to run on standalone

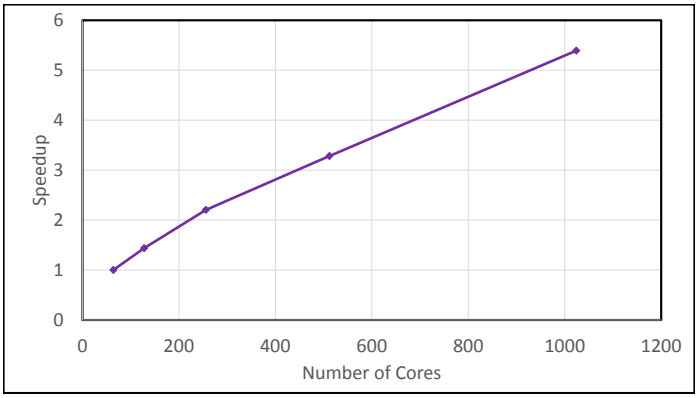

Fig. 2.10.: Speedup for Orion of searching Homo Sapien genomic scaffolds on Drosophila database

Linux/Windows boxes and uses multithreading for enhancing performance. We ran Homo sapiens chromosomal sequences and genomic scaffolds, shown on the X axis in Figure 2.11 as queries over the Drosophila database using Orion and BLAST+. We ran BLAST+ with 16 threads to fully utilize the available cores in the node, and ran Orion with 16 Map and Reduce tasks on a single node. Note that BLAST+ is only capable of running on a single node, which severely limits its applicability for large workloads.

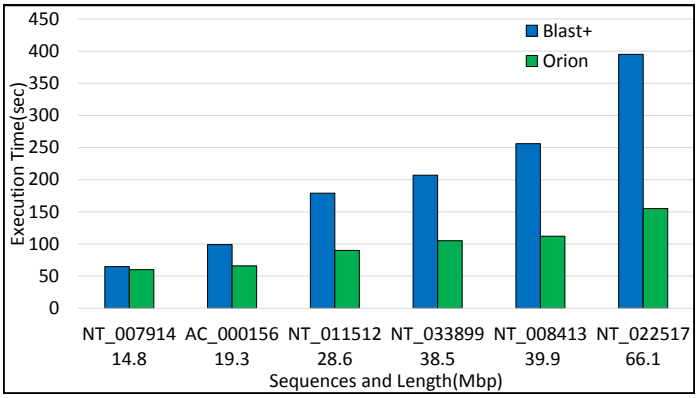

Fig. 2.11.: Comparison of BLAST+ and Orion

As seen in the figure, Orion performs better than BLAST+ for all the sequences with length of more than 10 Mbp. For smaller sequences, BLAST+ performs better than Orion due to the constant overhead of Hadoop job setup and tear down Orion has, which is higher

than the completion time of BLAST+ for the smaller queries. However it should be noted that this is a small constant overhead. Also the performance gains for Orion increase with increasing query sequence length. The performance gain of Orion over BLAST+ can be attributed to the finer level of parallelism of Orion. It exploits both intra-database and intra-query parallelism, while BLAST+ can only exploit intra-query parallelism.

2.5.7 Sensitivity study of Orion for different fragment lengths

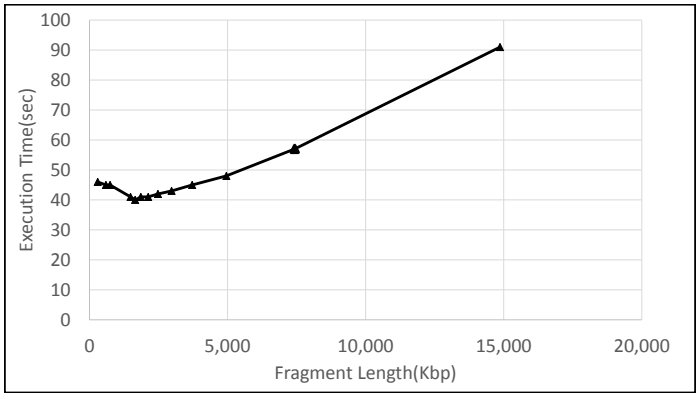

Fig. 2.12.: Sensitivity of Orion to fragment length

Here, we studied the sensitivity of Orion to different fragment lengths and show the results in Figure 2.12. We note that there are competing concerns regarding fragment length. Larger fragments mean less opportunities for alignments to cross boundaries, and thus less work to perform during alignment aggregation. However, as fragments get longer, the scope for parallelism decreases, and if fragments get too long, BLAST (which Orion uses) begins to suffer from poor cache behavior [16]. In addition the number of work units which is given by the number of query fragments times the number of database shards, should be larger than the number of available cores. Hence, we expect there to be a sweet spot in performance. we show this sweet spot for a 14.5M base-pair query against the Drosophila database. The ideal fragment length is 1.6M base pairs. This kind of calibration of Orion can be done once, for each database and then it can be used with the optimal (or near optimal) fragment size determined during the calibration. Note that for small queries, with

size smaller than the optimal fragment length, the sweet spot is never hit and Orion does not benefit from fragmenting the query.

2.5.8 Time distribution for phases of Orion

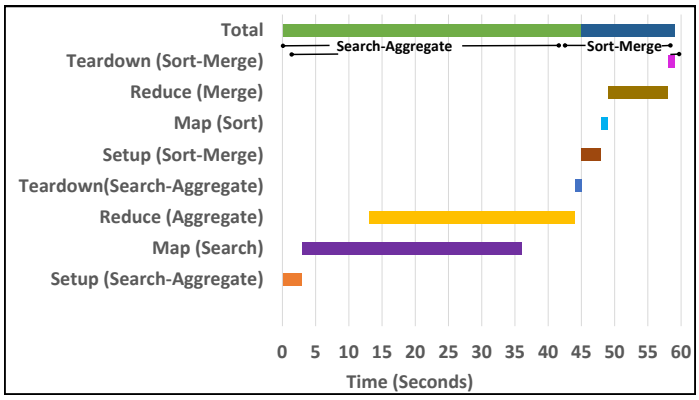

Fig. 2.13.: Timeline of events for Orion

Orion's running time can be decomposed into two primary MapReduce jobs: (1) the Search-Aggregate job and (2) the Sort-Merge job. We profiled Orion to determine how each job contributes to the total execution time of Orion. Because the behavior of Orion varies from fragment to fragment, we present results of aligning a representative input sequence, NT_007914, which has 14.5M base pairs, with the Drosophila database. The experiment was run on a single node, with 16 mappers and 16 reducers.

For each MapReduce job, we profiled the different phases of the job: (1) Setup, (2) Map, (3) Reduce, and (4) Teardown. For each job, and phase within the job, we recorded at what time the job/phase began and ended. The timeline of events is shown in Figure 2.13. The different phases of the two jobs are shown on the Y axis while the X axis shows the timeline. Note that Hadoop takes advantage of pipelining, so portions of each job's Map phase can overlap with its Reduce phase.

Based on these measurements, we see that the execution time is dominated by the Search-Aggregate job. Within this job, the Map phase performs the initial search for alignments and runs in a completely parallel manner, while the Reduce phase aggregates to-

gether the results from the query fragments. The Reduce phase overlaps with the Map phase, as the Reduce phase includes the time to copy results from the mappers to the reducers, and the copying starts while the Map phase is still running. The aggregation itself begins once all the data is available.

The Sort-Merge job also executes as a Map Reduce job, with the Map partitioning the alignments with different range of scores and assigning them to different reducers. The Reduce phase merges all these alignments into the final sorted output. Since the Map phase is completely parallel, the Reduce (Merge) phase dominates the sorting time as expected. The setup and teardown times are significant for jobs with short execution times. This detailed breakdown and timing of all the phases of execution, helps to understand Orion's behaviour and can be used for optimizing the execution times of Orion.

2.5.9 Results on larger databases

Orion consistently outperforms mpiBLAST over other databases as well. We also performed experiments over two larger databases — the Mouse genome database (unformatted size 2.77G) and the NT database (56.5 G) and found similar results. For example, with mpiBLAST, the search of a single query sequence NG_007092 having 2311 Kilo basepairs over the mouse database took 2664 seconds to complete while Orion completed the search in 201 seconds while for the even larger NT database a single query sequence NT_077570, having 263 Kilo base pairs, took almost an hour and a half (5,271.8 seconds), while Orion ran in 15 minutes using the sweet spot for the fragment length determined for query NT_077570. Thus Orion also scales to bigger databases. At these large database sizes the difference in matching times between our solution and the current state-of-art becomes even more significant and impactful.

2.6 Related Work

A vast body of research [27–34] has addressed the parallelization of sequence alignment algorithms based on various parallel programming paradigms in the wake of the mas-

sive data sets generated by next-generation high-throughput sequencing systems. These parallelization methods can be classified into two categories by their approaches to data decomposition. In the first category, where mpiBLAST belongs, the database that contains reference sequences is partitioned into multiple shards and hence a query sequence can be searched simultaneously against different shards by different execution units, *i.e.* processes or threads. Methods in the second category consider a large set of queries and by parallelizing the alignment of multiple queries, reduce the overall finish time. The set of queries are simply split into smaller subsets and the alignment of different subsets are executed in parallel by different execution units. The rest of this section reviews several representative works from both the categories. None of the schemes described here adopt the same query fragmentation strategy as Orion (though some fragment sequences in the database). To our knowledge, Orion's fragmentation strategy is unique among parallel BLAST implementations.

CloudBurst [27] is modeled after the short read-mapping program RMAP [35], but implements the algorithm as a classic MapReduce program to parallelize execution using multiple compute nodes. Like RMAP, CloudBurst takes a seed-and-extend approach, extracting all k-mers in the reference sequence and non-overlapping k-mers in all queries in the map phase, sorting all k-mers by their sequence in the shuffling phase, and finally in the reduce phase identifying k-mers shared between the reference sequence and the queries and extending them into end-to-end alignments allowing for a fixed number of insertions, deletions or mismatches. It is optimized for the alignment of many short queries against long reference sequences. Like the database sharding in mpiBLAST, CloudBurst partitions database sequences into 65 kb chunks with 1kb overlaps to support cross-chunk alignment of queries shorter than 1 kb. The shuffle phase which is essentially an all-to-all communication among all compute nodes imposes a high throughput demand on the network and will eventually become the scalability bottleneck.

CloudBLAST [32] uses the Hadoop MapReduce framework to parallelize the alignment of a set of queries. Similar to our approach that builds on top of the established BLAST implementation, CloudBLAST runs BLAST as map tasks of Hadoop over a distributed

cluster of virtual machines. The set of queries are partitioned into subsets, which is then assigned to map tasks that search the subset of queries over the entire database. Without exploiting the parallelism from database sharding, CloudBLAST suffers poor performance when dealing with large reference databases.

Yang *et al.* [19] also identify the scalability limitation of BLAST for long query sequences and employ the Hadoop framework to speedup the alignment of long sequences. The parallelism in their scheme comes solely from database sharding: they exploit the file segmentation in HDFS to split a large database into 64MB chunks and run a query as parallel map tasks against different chunks of the database. Long reference sequences in the database are also split into fragments of fixed size with overlaps to reduce the possibility that a map task needs to access chunks of the database stored on a remote node.

GPU-BLAST [34] achieves nearly 4X speedup on a 1.15 GHz NVIDIA Fermi GPU over the single-threaded NCBI-BLAST running on an 2.67 GHz Intel Xeon CPU. It takes the database sharding approach by assigning the reference sequences in the database to different GPU threads for parallel alignment. To mitigate the performance penalty from thread divergence, GPU-BLAST also includes a preprocessing step that sorts all reference sequences in the database by their lengths to avoid having threads of the same warp work on reference sequences with significant length differences.

2.7 Conclusions

With the ever increasing importance of gene sequencing and alignment to systems biology, and the corresponding increase in the number, size and variety of queries and genomic databases, it is of paramount importance that computational sequencing algorithms be parallelized efficiently. Prior approaches to parallel BLAST search did not exploit all available parallelism, leading to unacceptably slow performance when performing matches on large query sequences. In this chapter, we have presented Orion, which uses a novel parallelization strategy, fragmenting individual queries into overlapping fragments. Through a careful analysis, we determine how to fragment the queries such that the accuracy of the

final alignments is not reduced. The evaluation with real biological use cases shows that Orion significantly outperforms the most popular parallel BLAST implementation, called mpiBLAST, for large queries. For example, with a large NCBI database called the NT database and a long query corresponding to a human chromosome, Orion shows a 5X improvement in execution time over mpiBLAST. Further, the nature of genome alignment is such that static scheduling does not work well. As a result, mpiBLAST shows significant load imbalance. Orion on the other hand, thanks to using the Hadoop framework, achieves load balancing across all the computational cores.

3. SARVAVID: A DOMAIN SPECIFIC LANGUAGE FOR DEVELOPING SCALABLE COMPUTATIONAL GENOMICS APPLICATIONS

3.1 Introduction

Genomic analysis methods are commonly used in a variety of important areas such as forensics, genetic testing, and gene therapy. Genomic analysis enables DNA fingerprinting, donor matching for organ transplants, and in the treatment for multiple diseases such as cystic fibrosis, Tay-Sachs, and sickle cell anemia. Recent advances in DNA sequencing technologies have enabled the acquisition of enormous volumes of data at a higher quality and throughput. At the current rate, researchers estimate that 100 million to 2 billion human genomes will be sequenced by 2025, a four to five order of magnitude growth in 10 years [36, 37]. The availability of massive datasets and the rapid evolution of sequencing technologies necessitate the development of novel genomic applications and tools.

Three broad classes of genomics applications are local sequence alignment, whole genome alignment (also known as global alignment), and sequence assembly. *Local sequence alignment* finds regions of similarity between genomic sequences. *Whole genome alignment* finds mappings between entire genomes. *Sequence assembly* aligns and merges the sequenced fragments of a genome to reconstruct the entire original genome.

As the amount of genomic data rapidly increases, many applications in these categories are facing severe challenges in scaling up to handle these large datasets. For example, the parallel version of the de facto local sequence alignment application called BLAST suffers from an exponential increase in matching time with the increasing size of the query sequence–the knee of the curve is reached at a query size of only 2 Mega base pairs (Mbp) (Figure 3 in [4]). For calibration, the smallest human chromosome (chromosome 22) is 49 Mbp long. Existing genomic applications are predominantly written in a monolithic

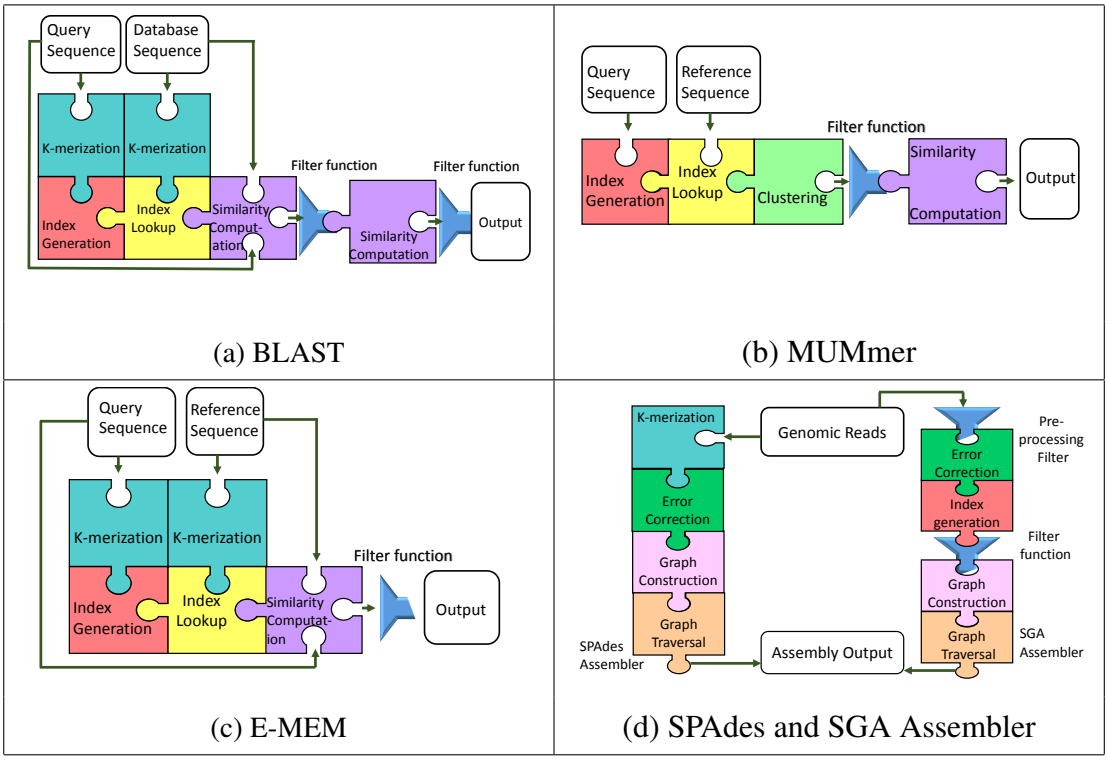

Fig. 3.1.: Overview of Genomic Applications in the categories of Local Alignment (BLAST), Whole genome Alignment (MUMmer and E-MEM), and Sequence Assembly (SPAdes and SGA). Common kernels are shaded using the same color.

manner and therefore are not easily amenable to automatic optimizations such as reduction of the memory footprint or creation of concurrent tasks out of the overall application. When the working set of the application spills out of memory available at a compute node, performance starts to degrade.

Further, different use cases often necessitate the application designer to make sophisticated design decisions. Consider a use case where a reference sequence, RS, is matched against a stream of incoming sequences. The natural choice is to index RS and search through it for other sequences. However, if the index overflows the memory capacity of the node (as will happen for long sequences), the user may partition RS into two or more sub-sequences, distribute their indices to different nodes, and perform the matching in a dis-

tributed manner. The distributed approach may fail to obtain high performance if the data transfer cost dominates. Another common example is the choice of data structures. Multiple sophisticated data structures such as suffix trees, FM-index, and Bloom filters have been developed for use with particular applications, but the correct choice depends on the size and characteristics of the input and the capabilities of the compute node. Such design decisions are often challenging for a genomics researcher to grasp. Furthermore, the monolithic design of these applications make it nearly impossible for a genomics researcher to change the design decisions adopted by the application.

One obvious solution to the challenge of large data sets is to parallelize applications. Unfortunately, for a significant number of important problems in the genomics domain, no parallel tools exist. The available parallel tools cater to local sequence and whole genome alignments [2, 3, 38–40] where a set of input sequences is matched against a set of reference sequences. Developing such tools is a cumbersome task because the developer must employ strategies for load balancing, synchronization, and communication optimization. This productivity bottleneck has been identified by several researchers in the past [41, 42]. This bottleneck, in part, has resulted in exploitation of only the coarser grained parallelism, wherein individual sequences are matched concurrently against the reference sequence, a typical case of embarrassing parallelism. However, for long and complex sequences the individual work unit becomes too large and this coarse-grained parallelization becomes insufficient.

Our key insight is that across a wide variety of computational genomics applications, there are several common building blocks, which we call *kernels*. These building blocks can be put together through a restricted set of programming language constructs (loops, etc.) to create sophisticated genomics applications. To that end, we propose SARVAVID[1], a Domain Specific Language (DSL) for computational genomics applications. Figure 3.1 shows that BLAST [6], MUMmer [43], and E-MEM [44] applications conceptually have a common kernel, *similarity computation* to determine similarity among genomic sequences,

[1]SARVAVID is a Sanskrit word meaning "omniscient". Here, SARVAVID knows the context of a large swath of genomics applications and can consequently execute them efficiently.

albeit with different constraints. Apart from these applications, other applications such as BLAT [45], BWT-SW [46], slaMEM [47], essaMEM [48], sparseMEM [49], ALLPaths-LG [50], Bowtie [51] and Arachne [52] also are composed from the same small set of kernels.

SARVAVID provides these kernels as language constructs, with a high degree of parameterization. With such high-level abstractions, genomics researchers are no longer restricted by the design decisions made by application developers. An entire genomics application can be expressed in a few lines of code using SARVAVID. Genomics researchers do not have to be algorithm/data structure experts. Using kernels to express applications is highly intuitive, allowing users to focus on the scientific piece of the application rather than on software design and implementation concerns such as algorithmic improvements and data structure tuning.

Typically, providing high-level abstractions comes with a penalty—a lack of efficient implementations. For example, library-based approaches for writing genomics applications such as SeqAn [53] provide the expressibility of high-level abstractions, but do not allow optimizations across library calls. Moreover, the use of library calls precludes general-purpose compilers from performing many other common optimizations. Instead, SARVAVID's DSL approach enables domain-specific, kernel-aware optimizations, eliminating the abstraction penalty. Moreover, SARVAVID's high-level specification through kernels enables easy extraction of parallelism from the application, allowing it to scale to a large number of cores. Fundamentally, it is easier for our DSL compiler to understand and implement a data-parallel form of execution where a kernel can be executed concurrently on different partitions of the input data. Where such independent partitions are unavailable, SARVAVID allows users to provide a partition function and a corresponding aggregate function. Thus, finer grained parallelism can be exploited in SARVAVID. Finally, kernelization aids in the reuse of optimized software modules across multiple applications.

We demonstrate the effectiveness of SARVAVID by developing kernelized versions of five popular computational genomics applications—BLAST for local alignment, MUMmer and E-MEM for global alignment, and SPAdes [54] and SGA [55] for assembly. The

developed applications all run at least as fast as existing hand-written baselines, and often significantly faster. SARVAVID versions of BLAST, MUMmer, and E-MEM achieve a speedup of 2.4X, 2.5X, and 2.1X over vanilla BLAST, MUMmer, and E-MEM applications respectively.

We make the following contributions:

- We recognize and present a set of common software building blocks, *kernels*, which recur in a wide variety of genomics applications.
- We create a DSL customized for computational genomics applications that embeds these kernels as language constructs.
- We design a compiler that can perform domain-specific optimizations and automatically generate parallel code, executable on a Hadoop backend.
- We demonstrate the effectiveness of SARVAVID by compactly implementing five popular genomics applications, achieving significant concurrent speedups.

Section 3.2 describes the different areas of genomic applications. Section 3.3 details the design of the language and kernels. Section 3.4 discusses the implementation and various optimizations performed by the compiler. Section 3.5 details case studies of applications developed using SARVAVID DSL. Section 3.6 surveys related work, while Section 3.7 concludes the document.

3.2 Background

Genomic sequences are either nucleotide or protein sequences. A nucleotide sequence is represented by a string of bases drawn from the set $\{A, C, T, G\}$ and a protein sequence is represented by a string drawn from a set of 20 characters. Genomic sequences are constantly evolving, leading to substitution of one base for another and insertion or deletion of a substring of bases. The following section describes how these sequences are processed in the key sub-domains in computational genomics—local sequence alignment, whole

genome alignment, and sequence assembly. Figure 3.2 provides a high-level overview of these sub-domains.

3.2.1 Local Sequence Alignment

Local sequence alignment is a method of arranging genomic sequences to identify *similar* regions between sequences. Local alignment searches for segments of all possible lengths between the two sequences that can be transformed into each other with a minimum number of substitutions, insertions and/or deletions. As genomes evolve, parts of the genomic sequences mutate and similar regions will have a large number of matches and relatively few mismatches and gaps. Regions of similarity help predict structural, functional, and evolutionary characteristics of newly discovered genes.

Local sequence alignment applications search a query set of input sequences against all sequences in a database to find similar subsequences. One such searching technique, the Smith-Waterman (SW) algorithm [18], uses dynamic programming to exhaustively search the database to find the optimal local alignments. However, this technique is extremely slow even for small sequences. Therefore, applications use indexing to speed up the database search and heuristics to prune the search space efficiently [45, 46, 56]. BLAST uses heuristics and hash table based indexing. BLAST heuristics maintain a good balance between accuracy (ability to find matching substrings) and speed, making it the most popular approach for local alignment.

Parallelization methods for local alignment have mainly explored two approaches. The first partitions the database into multiple shards and then searches the query sequence simultaneously against the shards [13, 38]. The other approach executes each query in a query-set in parallel against replicated copies of the database [32].

3.2.2 Whole genome alignment

Whole genome alignment refers to determining the optimal end-to-end alignment between two entire sequences. It provides information about conserved regions or shared

biological features between two species. Global alignment methods start by identifying chunks of exact matches within the sequences and then extending the matches over the remaining regions in the sequence to get the entire alignment. Indexing one sequence speeds up the search for finding the chunks of exact matches. The Needleman-Wunsch algorithm [57] uses dynamic programming to determine the optimal global alignment of two sequences. However, this technique is slow. Instead, applications use special data structures for efficient indexing of the sequences [47–49]. MUMmer uses a suffix tree and E-MEM employs hash table–based indexing, memory optimizations, and multithreading to handle complex genomes.

3.2.3 Sequence Assembly

DNA sequencing technologies are designed to obtain the precise order of nucleotides within a DNA molecule. However, they can only read small fragments of the genome, called *reads*. To mitigate the errors during the sequencing process, the genome is replicated before sequencing. Sequence assembly is the task of putting together these fragments to reconstruct the entire genome sequence. The more challenging variant of this (and one we focus on here) is where a reference genome is not available to guide the assembly; this variant is called *de-novo* assembly.

All assembly approaches assume that highly similar DNA fragments emerge from the same position within a genome. Similar reads are "stitched" together into larger sequences called *contigs*, to uncover the final sequence. To enable this process, assembly methods build a graph from the sequenced reads. Paths in the graph are identified by traversing the graph with different start and end nodes to generate contigs.

3.3 SARVAVID Language Design

SARVAVID DSL provides three main components to the user. First, a set of kernels that provide efficient implementations of commonly recurring software modules in genomics applications. Second, language features that enable easy combination of these kernels

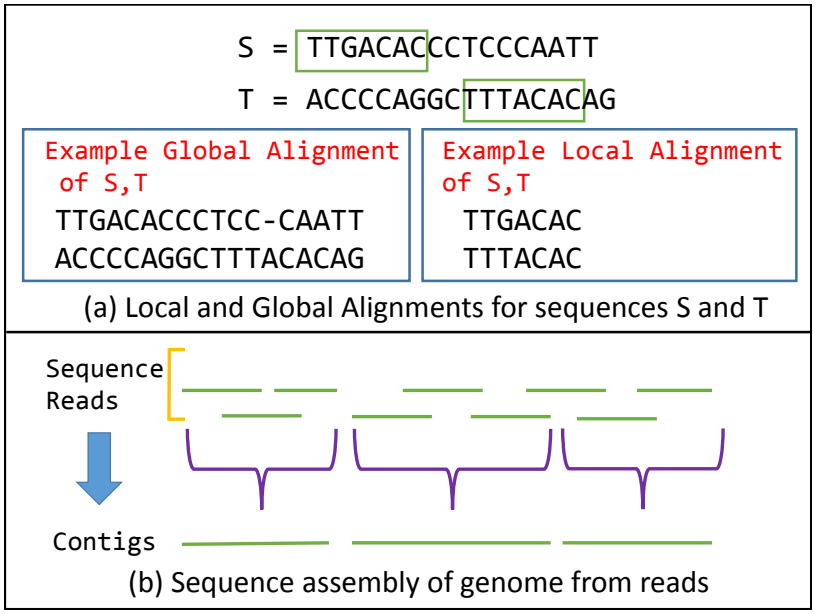

Fig. 3.2.: Overview of local alignment, global alignment (a), and sequence assembly(b)

while keeping the syntax simple. Third, a compiler that generates optimized code by using domain-specific grammar. In this section, we first provide a brief description of the kernels, and then a discussion of how they are used to build popular genomics applications. We then present SARVAVID grammar.

3.3.1 Kernels

Table 3.1 shows genomic applications and kernels that can be used to build them. We defer the description of how these kernels can be used in developing various applications to Subsection 3.3.2, and describe the kernels first.

1. *K-merization*: Genomic applications generally start by sampling subsequences of a string and then seeding further analyses with important subsequences. K-merization is a process that generates k-length subsequences, called *k-mers*, of a given string. Consider a string "ACGTCGA". All *3-mers* of this string are ACG, CGT, GTC, TCG, and CGA. This sampling can also be done at fixed intervals within the string. Thus, ACG and CGA are 3-mers separated by an interval of 4. The k-merization kernel

Table 3.1.: Kernels in various genomic applications

Kernel	BLAST	E-MEM	MUMmer	SPAdes	SGA
K-merization	✓	✓		✓	
Index-generation	✓	✓	✓		✓
Index-lookup	✓	✓	✓		✓
Similarity-computation	✓	✓	✓		
Clustering			✓		
Graph-construction				✓	✓
Graph-traversal				✓	✓
Error-correction				✓	✓

takes a string, length of the subsequences and the interval at which the subsequences are extracted as inputs, and outputs a set of k-mers and the offsets at which they occur in the sequence.

2. *Index generation*: Genomic applications regularly search for patterns in strings; indexing the strings makes this search efficient. The index data structure can be a hash table, suffix tree, suffix array, Burrows-Wheeler, or FM index, depending on factors such as lookup speed and storage requirements. The index generation kernel takes a string and the desired index data structure as inputs, and outputs the resulting index.

3. *Index lookup*: Genomic applications need fast lookup strategies as they attempt to find patterns in strings. Index lookup finds a sequence or a subsequence in an index. It takes the sequence to be searched along with the index structure as inputs, and outputs the index entry if the sequence can be found in the index.

4. *Similarity computation*: Applications try to find sequences that match a protein or DNA sequence, either exactly or by allowing mismatches or gaps. Recall that two genomic sequences are considered to be similar if one string can be transformed

into the other using edit operations: substitution, insertion, or deletion. Each edit operation is associated with a penalty and a match is associated with a reward. The score of the alignment is the sum of rewards minus the sum of penalties. Similarity computation takes the sequences and the score matrix as inputs, and returns the score of the alignment along with the edit operations chosen.

5. *Clustering*: Clustering is used in applications to group similar sequences into gene families or group small matches together to find longer alignments. Clustering uses a rule to group together a set of genomic objects in such a way that objects in the same group are more similar to each other than those in other groups. Clustering takes genomic objects and the clustering function (rule) as inputs, and returns the set of clusters.

6. *Graph construction*: Genomic assembly applications stitch together similar sequenced reads of a genome to obtain the original genome sequence. They use a graph construction module to generate a graph using the sequenced reads. Graph nodes are either the entire reads or subsequences of the reads. Overlapping nodes are connected by an edge. For example, reads that form suffix-prefix pairs are connected by an edge. Thus reads “ACGT”, “CGTC” and reads “TCAC” , “CACT” are connected by an edge. This kernel takes a set of sequences representing nodes, and a function to determine adjacency between nodes as inputs, and outputs the graph in the form of an adjacency list.

7. *Graph traversal*: Assembly applications attempt to find Eulerian paths and Hamiltonian paths in the graph to generate an assembly. All paths between nodes in the graph need to be identified by walking from one node to the other using intermediate nodes; hence, a graph traversal module is employed. Graph traversal takes a map representing graph adjacency list along with the entry and exit nodes as inputs, and outputs paths between them.

8. *Error correction*: DNA sequencing technologies read small fragments of the genome. However, they make errors while reading them. Error correction techniques are re-

quired in assembly applications to correct them. This kernel determines error-prone reads from a read list and removes them. It takes a set of reads and a function used to recognize error prone reads as inputs, and returns the corrected read list.

Extension of kernels This set of kernels can be easily extended. The grammar of SARVAVID presented in Subsection 3.3.3 describes how the DSL developer can design kernels found in other applications that are not currently in the kernel set of SARVAVID. Once they are added to this kernel set, they can be used for application design.

3.3.2 Survey of popular applications and their expression in the form of kernels

Now we survey popular applications and express them using kernels described above. Table 3.1 shows the kernels present in the different applications.

BLAST BLAST is a local alignment tool. BLAST first generates k-length substrings (k-mers) of the query sequence and inserts the k-mers into a hash table. It then generates k-mers of the database and looks them up in the hash table. The matching positions of k-mers in the query and the database forms the *hit-list*. Hits in the hit-list are processed using *ungapped extension*. The hits are extended to the left and to the right by allowing perfect matches and mismatches, with matches increasing and mismatches decreasing the score. All alignments scoring higher than the ungapped alignment threshold are passed for further processing. In the next step, called *gapped extension*, these high-scoring alignments are extended by allowing matches, mismatches, and gaps. Finally, selected high-scoring alignments are presented as outputs. BLAST involves the **k-merization**, **index generation**, **index lookup**, and **similarity computation** kernels.

MUMmer MUMmer is a whole genome alignment tool. MUMmer first identifies anchors or short regions of high similarity between the genomes by building a suffix tree for the reference sequence, then streams the query sequence over it. The matches are clustered and scored to determine the regions that give maximal similarity. The matches in

the clusters are extended to obtain the final global alignment. This workflow can be represented using kernels, as in Figure 3.1(b). Thus, MUMmer can be expressed using the **index generation**, **index lookup**,**clustering**, and **similarity computation** kernels.

E-MEM E-MEM is a more recent algorithm for whole genome alignment and follows a different strategy than MUMmer. E-MEM [44] identifies **M**aximal **E**xact **M**atches (MEM) between two sequences or whole genomes. E-MEM builds a compressed text index of the reference sequence and then searches for subsequences of the query in this index. Identified "hits" are extended to generate the MEMs, *i.e.*, sequences that cannot be extended either way in the query or the reference. This workflow can be represented using kernels, as in Figure 3.1(c). Thus E-MEM is expressed using the **k-merization**, **index generation**, **index lookup**, and **similarity computation** kernels.

SPAdes SPAdes first corrects errors using k-mers of the sequence reads and then builds a graph from the k-mers obtained from the reads. This graph is used to find an Eulerian path for the graph to derive the assembly. SPades is expressed using the **error correction**, **k-merization**,**graph construction**, and **graph traversal** kernels, as shown in Fig 3.1(d).

SGA Assembler SGA Assembler performs assembly by first preprocessing the read data set and removing low-quality reads. It then creates an FM index from the reads and the corrected reads are used to build a string graph. Assembly is obtained by traversing the string graph. SGA uses kernels as shown in Fig 3.1(d) namely, **error correction**, **index generation**, **graph construction**, and **graph traversal**.

3.3.3 Grammar

Figure 3.3 provides a reduced grammar for SARVAVID. Production 1 describes types ($\mathbb{T}$) allowed, while Production 2 describes different collections ($\mathbb{C}$) in SARVAVID. Primitive types such as integers, strings or bools are allowed. Strings are parameterized by the alphabet Σ of the string—for example, nucleotide strings are parameterized by the alphabet

$\{A, C, T, G\}$. SARVAVID also allows *tuples* of multiple types and *collections*: maps, sets, multisets, and sequences (i.e., ordered multisets). Productions 4 and 7 describe functions in SARVAVID. These are user-defined functions that take multiple variables and produce a single variable. SARVAVID also uses this facility to do basic collection manipulation such as retrieving a key's entry from a map, or accessing a particular value of a tuple as shown in Production 8. Production 6 states that the main program is a collection of statements, while Production 9 provides their derivations. Production 10 describes the key feature of SARVAVID. It provides pre-defined kernels. Notably, named functions can be passed to kernels, providing higher-order features (*e.g.*, passing a filtering function to a kernel). Table 3.2 describes the interface for the kernels described in Section 3.3.1.

1. $\tau \in \mathbb{T} ::= c \mid [\tau_1, \tau_2, ...] \mid string_\Sigma \mid int \mid bool$
2. $c \in \mathbb{C} ::= map < \tau_1, \tau_2 > \mid sequence < \tau > \mid set < \tau > \mid multiset < \tau >$
3. $x \in Variable ::= x_1 \mid x_2 \mid \ldots$
4. $f \in Functions ::= f_1 \mid f_2 \mid \ldots$
5. $v \in Values ::= \mathbb{Z} \cup \Sigma^*$
6. $p \in Program ::= f_d^* \; s$
7. $f_d \in FuncDefs ::= f = f_v(\tau_1 : x_1, \tau_2 : x_2, \ldots) : \tau_r$
8. $e \in Exprs ::= v \mid x \mid f(x_1, x_2, \ldots) \mid x[e]$
9. $s \in Stmts ::= s; s \mid \textbf{for}(x_1 \in c : x_2)\{s\} \mid \textbf{for\_key}(x_1 \in map < \tau_1, \tau_2 >: x_2)\{s\} \mid \textbf{for\_value}(x_1 \in map < \tau_1, \tau_2 >: x_2)\{s\} \mid x = f(x_1, x_2, \ldots) \mid x = k$
10. $k \in Kernels ::= \texttt{kmerize}(\ldots) \mid \texttt{index\_generation}(\ldots) \mid \ldots$

Fig. 3.3.: (Simplified) SARVAVID grammar

Programs are translated by the SARVAVID compiler into C++ code. The translated code uses particular instantiations of the kernels. We show example codes for BLAST, MUMmer, and E-MEM applications in SARVAVID in Figures 3.4, 3.5, and 3.6. The kernels have been highlighted and a reader may see the similarity with C or C++ syntax,

Table 3.2.: Kernels in SARVAVID and their associated interfaces. The input arguments are shown before the ": " and the output arguments after it. The scenarios show different possible behaviors of the kernels

Kernel	Interface	Scenarios in which kernel can be used
K-merization	`kmerize`(*string*, *int*, *int*): *set*⟨[*int*,*int*]⟩	Different k-mer lengths and interval
Index generation	`index_generation`(*set*⟨τ⟩): *map*⟨τ_1, τ_2⟩	Create a suffix tree, FM-Index, Hash Table
Index lookup	`index_lookup`(*map*⟨τ_1,τ_2⟩ , τ_3): τ_4	Lookup in a suffix tree, FM-Index, Hash Table
Similarity computation	`similarity_computation`([*int*,*int*,*int*, *int*,*int*], *set*⟨*string*⟩,*string*,*int*): [*int*,*int*,*int*,*int*,*int*]	Perform exact, ungapped, gapped alignment
Clustering	`clustering`(*set*⟨τ⟩, *e*):τ	Perform connectivity based, density based clustering
Graph construction	`graph_construction`(*set*⟨τ⟩):*map*⟨τ_1,τ_2⟩	Specify different adjacency relationships
Graph traversal	`graph_traversal`(*map*⟨τ_1,τ_2⟩): τ	Specify DFS,BFS traversal
Error correction	`error_correction`(*set*⟨τ⟩, *e*):τ	Use different functions like BayesHammer,IonHammer

```
int main(){
   Q_set = k_merize(Q, k, 1);
   lookup_table = index_generation(Q_set);
   DB_set = k_merize(D,k,1);
   for(d_set in DB_set)
      list = index_lookup(lookup_table,d_set);
   for(hit in list){
      al1 = similarity_computation(hit,Q,D,UNGAPPED);
      if(filter(al1, threshold1))
         ungapped_list = insert(al1);
   }
   for(al in ungapped_list){
      al2 = similarity_computation(al,Q,D,GAPPED);
      if(filter(al2,threshold2))
         gapped_list = insert(al2);
   }
   status = print(gapped_list);
}
```

Fig. 3.4.: BLAST application described in SARVAVID

which is by design, as we find that genomics scientists are most comfortable with these languages.

```
int main( )
{
    suffix_tree = index_generation(R);
    full_list = index_lookup(suffix_tree,Q);
    sel_list = filter(full_list,mem_length);
    cluster_list = clustering(sel_list,DIST);
    for (cluster in cluster list){
        for (element in cluster){
            m = extract_offsets(element);
            al = similarity_computation(m,Q,R,DYN);
            match_list = insert(al);
        }
    }
    status = print(match_list);
}
```

Fig. 3.5.: MUMmer application described in SARVAVID

```
int main( )
{
     interval = mem_length - k + 1;
     db_set = k_merize(D, k, interval);
     lt_table = index_generation(db_set);
     q_set = k_merize(Q,k,1);
     list = index_lookup(lt_table,q_set);
     for(h in list){
          m = similarity_computation(h,D,Q,EXACT);
          if(filter(m,mem_length))
               match_list = insert (m);
     }
     status = print(match_list);
}
```

Fig. 3.6.: E-MEM application described in SARVAVID

3.4 Compilation Framework

SARVAVID performs optimizations by building a dependence graph that traces the flow of data from one kernel call to the next, enabling both the determination of which data is

used by which kernels as well as exposing opportunities for parallelism. The compiler then translates the flow using context-dependent rewrite rules for the kernels and then hands the translation over to the domain-specific optimizer. Once the optimizations are complete, the compiler translates the SARVAVID code into C++, which can be further optimized during compilation by the target language compiler. Figure 3.7 illustrates the overall structure of the compiler framework.

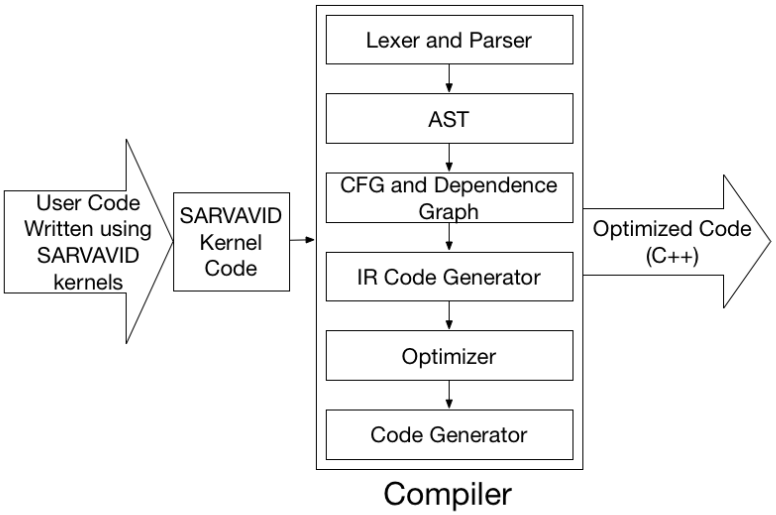

Fig. 3.7.: SARVAVID Compilation flow in SARVAVID

Generic compilers lack domain-specific optimizations as they do not understand the underlying semantics. An expert can perform optimizations manually and beat the best optimizing compilers, mainly because of his domain knowledge. In SARVAVID, high performance is obtained by leveraging domain-specific semantics. Lack of such knowledge limits generic compilers from performing many traditional optimizations, such as loop fusion and loop invariant code motion, as the existence of complicated genomics kernels can preclude the compiler from understanding the necessary dependence structure. Such optimizations are enabled by the SARVAVID compiler. Next we explain some such optimizations enabled by SARVAVID.

```
StA=k_merize(seqA,k,1);
A= index_generation(StA);
for(kmr in seqB)
  index_lookup(A,kmr);

StC=k_merize(seqC,k,1);
C= index_generation(StC);
for(kmr in seqB)
  index_lookup(C,kmr);
```

Before optimization

```
StA=k_merize(seqA,k,1);
A= index_generation(StA);
StC=k_merize(seqC,k,1);
C= index_generation(StC);
for(kmr in seqB)
{
   index_lookup(A,kmr);
   index_lookup(C,kmr);
}
```

After optimization

Fig. 3.8.: seqB is looked up in indices of reference sequences seqA and seqC. SARVAVID understands the kernels index generation and lookup, and knows that the loops over seqB can be fused

3.4.1 Loop Fusion

Loop fusion combines two loops into a single loop. Fusion generally improves performance if the data accessed in both the loops is the same. It does so by increasing locality of reference, resulting in reduced cache misses. Accessing long genomic sequences multiple times at different time instances leads to significant cache misses in applications. SARVAVID's compiler fuses scans over long sequences, and schedules computation when the data is still in cache. In the example shown in Figure 3.8, seqB is scanned twice for lookup in the indices of seqA and seqB. SARVAVID fuses scanning loops over seqB to reduce cache misses.

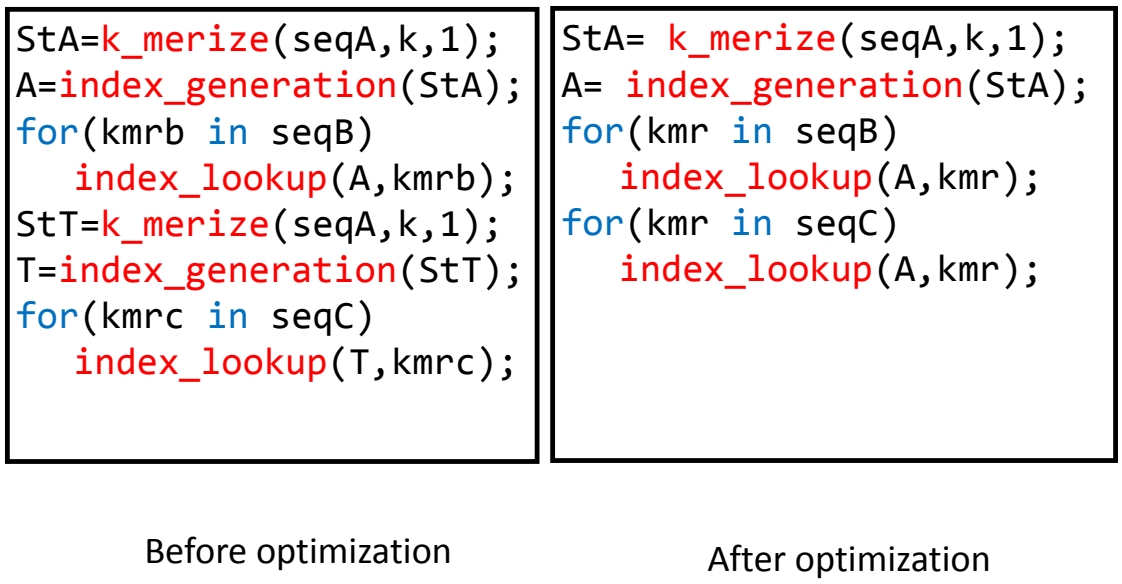

Fig. 3.9.: seqA is compared with seqB and seqC. SARVAVID understands the kernels and reuses index A in the lookup calls for seqC, deleting the second expensive call to regenerate the index for seqA

3.4.2 Common Sub-expression Elimination (CSE)

Common sub-expression elimination searches for instances of identical expressions and examines if they can be replaced by storing the result of the expression once and reusing it. In genomic applications, sequences are indexed for fast search. However, these indexes are recomputed for lookup against different sequences from various function calls. SARVAVID's compiler computes redundant expressions only once, and reuses their result. In the example shown in Figure 3.9, seqA is k_merized twice. After performing CSE on it, the compiler recognizes that the two index generation calls produce the same result, and therefore eliminates one of them as well.

3.4.3 Loop invariant code motion (LICM)

LICM searches for statements in a loop that are invariant in the loop, and hence can be hoisted outside the loop body, resulting in computation reduction. Genomics applications often compare a set of sequences against each other. Since the applications only allow two inputs, the user is forced to call them repeatedly in a loop. The SARVAVID compiler can identify loop invariant kernels in the application and hoist them outside the loop. In Figure 3.10 SARVAVID moves the loop invariant index generation kernel outside the loop.

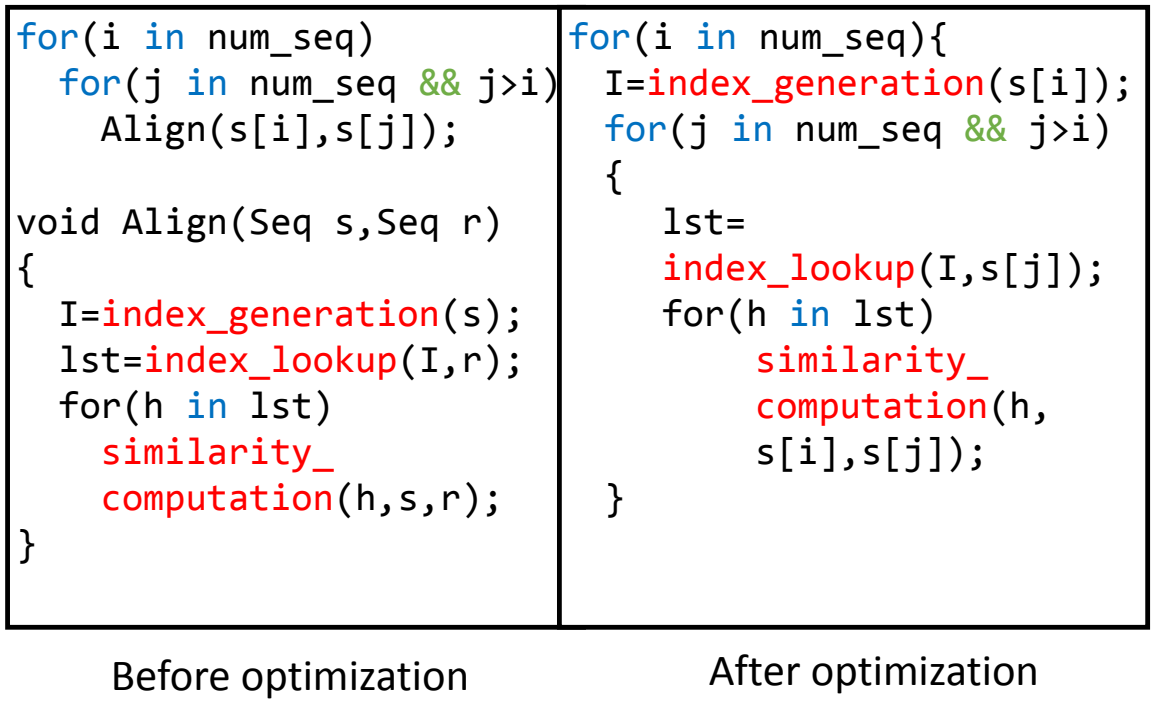

Fig. 3.10.: Sequences in the sequence set are compared against each other. SARVAVID compiler first inlines the kernel code, and then hoists the loop invariant index generation call prior to the loop body, thus saving on expensive calls to the index generation kernel

```
for(Q in query_set)
{
   Sq=k_merize(Q,k,1);
   I=index_generation(Sq);
   for(kmr in R)
   {
     L=index_lookup(I,kmr);
     for(h in L)
       h=similarity_
         computation(h,Q,
         R,UNGAPPED);
    }
}
```

```
Partition(String R, Int F , Int
O)
{
  Partition R into fragments of
  length F with required
  overlap  O;
}
Aggregate(Alignment A[])
{
   Aggregate overlapping
   alignments for fragments of R
   matching the same query
   sequence;
}
```

Fig. 3.11.: Individual query sequences can be aligned in parallel by partitioning the query set. The reference sequence can be partitioned and processed using the partition and aggregate functions.

3.4.4 Partition-Aggregate Functions

If iterations over elements in a collection are independent, SARVAVID can partition the collection into its individual elements automatically. For example, if there is a set of queries where each query needs to be aligned against a reference sequence, then each query can be processed independently and in parallel. In addition to this embarrassing parallelism, SARVAVID can explore more subtle opportunities for data parallelism. Partition-Aggregate function pairs define how arbitrary input datasets can be split and how results of the individual splits can be combined to give the final output of the application. Kernels are executed concurrently over the splits.

Consider a simple **for** loop aligning several `query` sequences in a set against a single reference sequence `R`, as shown in Figure 3.11. In addition to parallelizing the **for** loop, SARVAVID can also *partition* the `R` sequence itself, and perform similarity computations for each subsequence separately. This exposes finer grained parallelism for long sequences. To account for the fact that alignments could cross partition boundaries, the partitions of `R` overlap one another. The resulting alignments can then be combined using a custom aggregation function that eliminates duplicates and splices together overlapping alignments.

We use the strategy proposed by Mahadik *et al.* [4] to implement the partition and aggregate functions. Figure 3.11 provides the pseudo code for partition and aggregate functions. SARVAVID invokes the partition functions and runs them on the Hadoop distributed infrastructure as Map tasks. SARVAVID then invokes the aggregation function on the results of all the partitions as Reduce tasks.

3.4.5 SARVAVID Runtime

SARVAVID's runtime component orchestrates the execution of input datasets for an application. SARVAVID compiler translates the application expressed in the form of kernels, and applies relevant optimizations. The optimized C++ code is handed over to *gcc* to create an executable. SARVAVID then prepares the input datasets, with the supplied partition function, if any, and keeps them on a shared storage. SARVAVID framework also compiles the supplied aggregate function to an executable and all executables are placed on the same shared storage. SARVAVID's parallelization strategy of running computation concurrently on multiple partitions of the input data fits very well with Hadoop's MapReduce paradigm [22]. Hence, we use Hadoop streaming to create and run Map/Reduce jobs with the created executables. A preprocessing script, which is part of the SARVAVID framework, generates a listing to run a specific executable with the correct inputs and arguments.

Consider a reference R and query Q as inputs for a local/global alignment application, and two partitions each of the query and reference are requested by the partition function. The sequences are partitioned to create R1 and R2, and Q1 and Q2. They are placed on the Hadoop Distributed File System(HDFS), accessible to all nodes in the cluster. The preprocessing script creates a listing to execute the local/global alignment on every partition of the reference and query (*i.e.*, R1 with Q1, R1 with Q2, R2 with Q1, and R2 with Q2). SARVAVID submits a MapReduce job with this listing as the input. Each item in the listing creates a separate map task. Thus, 4 map tasks are spawned for the job, and they run the specified executable with the requested reference and query arguments. The map tasks run in parallel on available cores in the cluster, and their outputs are fed to the reduce tasks. The

reduce task runs the aggregation script on its input to create the final output. This output is fetched from the HDFS and presented to the user.

3.5 Evaluation

In this section we demonstrate the performance and productivity benefits of using SARVAVID to develop sequence analysis applications. We select applications used extensively by the genomics community in the categories of local alignment, global alignment and sequence assembly, namely, BLAST, MUMmer, E-MEM, SPAdes, and SGA. For terminology, we use the term baseline-"X" or simply "X" to denote the vanilla application and "X" written in SARVAVID with SARVAVID-"X".

3.5.1 Experimental Setup and Data Sets

We performed our experiments on an Intel Xeon Infiniband cluster. Each node had two 8-core 2.6 GHz Intel EM64T Xeon E5 processors and 64 GB of main memory. The nodes were connected with QDR (40 Gbit) Infiniband. We used up to 64 identical nodes, translating to 1024 cores. We used the latest open source versions of all applications - BLAST (2.2.26), MUMmer (3.23), E-MEM (dated 2014), SPAdes (3.5.0), and SGA (0.10.14). Each node in the Hadoop cluster was configured to run a maximum of 16 Map tasks and 16 Reduce tasks to match the number of cores on the nodes. We tabulate the different inputs used in our experiments in Table 3.3. The input datasets are based on relevant biological problems [58, 59].

3.5.2 Performance Tests

In this section we compare the performance of all applications provided with inputs in Table 3.3. Applications BLAST, E-MEM, SPAdes and SGA are multi-threaded and can therefore utilize multiple cores on a single node, while MUMmer runs on a single core. Hence, we first compare performance on a single node. Figure 3.12 shows that SARVAVID versions of applications perform at least as good as the baseline vanilla applications. For

Table 3.3.: Applications used in our evaluation and their input datasets.

Application	**Input**
BLAST	Fruit fly (127 MB), Mouse (2.9 GB)(reference)
	Human contigs (1-25 MB)(queries)
MUMmer	Mouse (2.9 GB) , Chimpanzee (3.3 GB),
	Human(3.1GB), Fruit fly species (127-234 MB)
E-MEM	*T durum* (3.3 GB), *T aestivum* (4.6 GB)
	T monococcum (1.3 GB), *T strongfield* (3.3GB)
	Fruit fly species (127-234 MB)
SPAdes	E-coli paired reads (2.2 GB),
	C elegans reads (4.9 GB),
SGA	E-coli reads (2.2 GB),
	C elegans reads (4.9 GB)

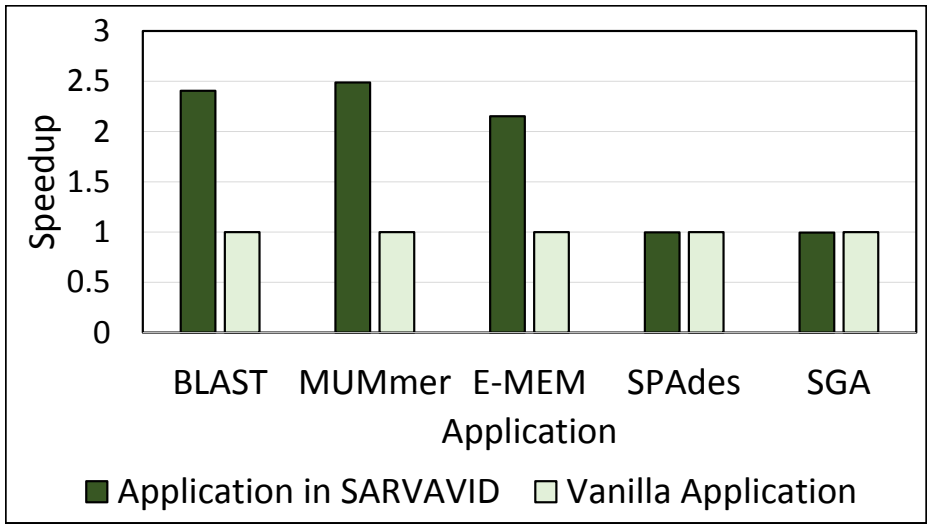

Fig. 3.12.: Performance comparison of applications implemented in SARVAVID over original (vanilla) applications. These are all runs on a single node with 16 cores. All except MUMmer are multi-threaded in their original implementations.

applications BLAST, MUMmer, and E-MEM, SARVAVID performs better than the baseline applications by 2.4X, 2.5X, and, 2.1X respectively. These benefits are a result of var-

ious optimizations performed in SARVAVID. SGA and SPAdes are not amenable to these optimizations, and hence achieve performance equal to the baseline.

To understand how each individual optimization contributes to the achieved speedup we compare the speedup achieved by a program version where only that optimization was enabled over a program version where all optimizations were disabled. Figure 3.13 shows the results, all for a single node (which has 16 cores). CSE achieves an average speedup of 1.22X. Loop fusion achieves an average speedup of 1.11X. Loop hoisting achieves an average speedup of 1.35X. Fine-grain parallelism exploitation using partition-aggregate functions obtains an average speedup of 2.1X, even on a single node. Figure 3.13 does not include applications SGA and SPAdes since our compiler's optimizations do not apply to them.

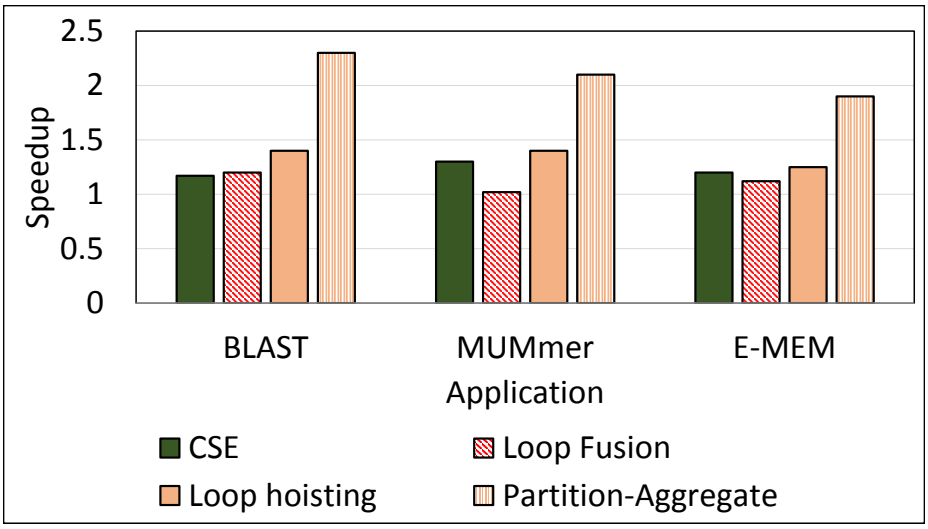

Fig. 3.13.: Speedup obtained by CSE, Loop Fusion, LICM, and parallelization (Partition-Aggregate) over a baseline run, *i.e.*, with all optimizations turned off, on a single node

3.5.3 Scalability

Currently available versions of MUMmer and E-MEM are single node applications. With SARVAVID, these applications can be automatically scaled to multiple nodes. We

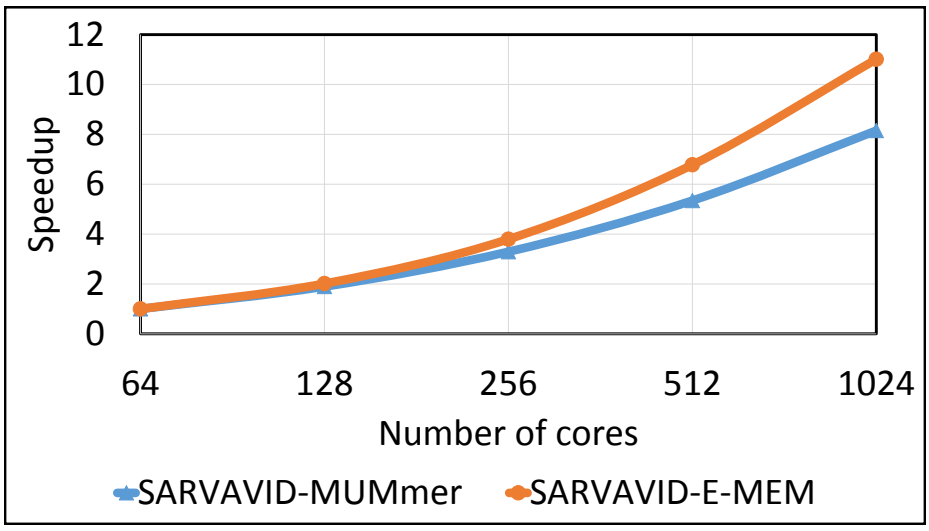

Fig. 3.14.: Speedup achieved by SARVAVID-MUMmer and SARVAVID-E-MEM calculated over 64 core runs of SARVAVID-MUMmer and SARVAVID-E-MEM respectively

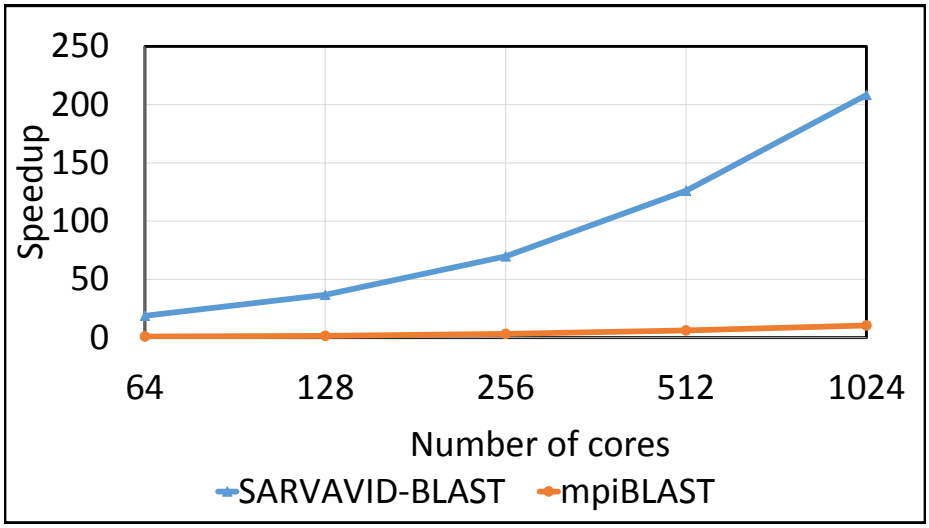

Fig. 3.15.: Speedup achieved by SARVAVID-BLAST calculated over 64 core run of mpiBLAST

increased the number of cores in the system for SARVAVID-MUMmer and SARVAVID-E-MEM applications from 64 (4 nodes) to 1024 (64 nodes) and measured the speedup achieved using as baseline SARVAVID-MUMmer and SARVAVID-E-MEM running on 64 cores. Fig 3.14 shows the scalability test. We attribute the lower scalability of MUM-

mer compared to E-MEM to its clustering and post-processing workload, which are both sequential.

For the scalability comparison of BLAST, we use an existing, highly popular parallel version for BLAST, called mpiBLAST [13]. mpiBLAST shards the database into multiple non-overlapping segments, and searches each segment independently and in parallel against the query. Figure 3.15 shows that on 1024 cores, SARVAVID-BLAST achieved a speedup of 19.8X over mpiBLAST. mpiBLAST exploits only the coarser grain parallelism, unlike SARVAVID, where both the coarse and fine grained parallelisms are exploited. The scalability is attributed to the partition and the aggregate functions that make the corresponding parts of SARVAVID applications run as Map-Reduce tasks on a standard Hadoop backend.

3.5.4 Comparison with library-based approach (SeqAn)

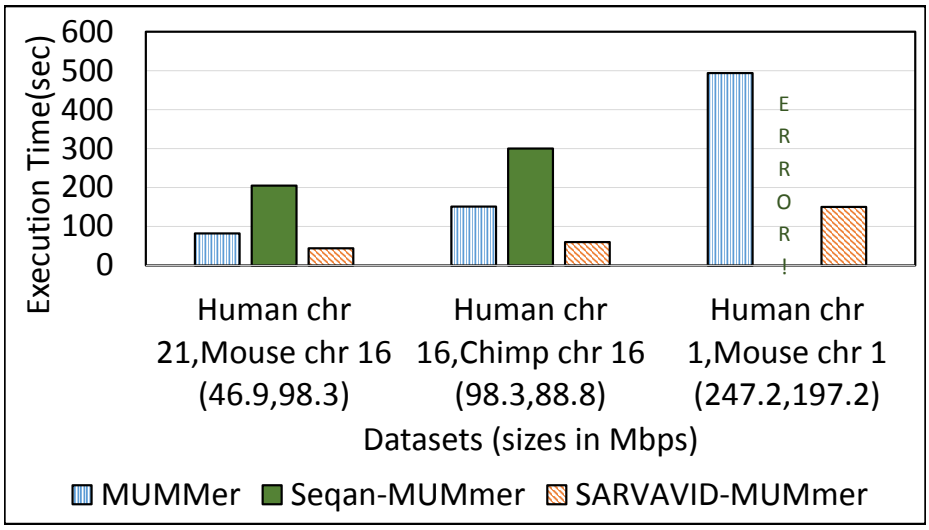

Fig. 3.16.: Comparison of execution times for MUMmer implemented in SARVAVID and MUMmer implemented using a popular genomics sequence comparison library called SeqAn. The results are all on a single node with 16 cores

Now we compare performance of applications written in SARVAVID against the well-established computational genomics library, SeqAn [53]. We chose to compare perfor-

mance of MUMmer written in Seqan and SARVAVID since standard version of MUMmer in Seqan is available [60]. Figure 3.16 shows the execution time comparison of SeqAn-MUMmer, SARVAVID-MUMmer, and MUMmer over a range of inputs.

We used chromosomes from Human, Mouse and Chimp genomes as query and reference sequences. we varied reference size from 46.9 Mbp to 247.2 Mbp and query size from 88.8 Mbp to 197.2 Mbp. SARVAVID- MUMmer is faster than MUMmer written in SeqAn and MUMmer for all input sizes. Seqan-MUMmer terminated with error for the last input dataset. MUMmer written in SARVAVID partitions the query and reference sequence and executes in parallel on all 16 cores of the node. It also exploits optimizations provided by SARVAVID compiler to achieve speedup. SeqAn-MUMmer is slower for all input sizes than even the vanilla MUMmer, since it builds a suffix tree for both sequences while MUMmer builds a suffix tree just for the reference sequence, and streams the query sequence. SARVAVID-MUMmer's kernel implementation is similar to vanilla MUMmer. The lines of code required to implement MUMmer in SARVAVID is 36 and in SeqAn it is 40. Thus, SARVAVID- MUMmer code is as concise as the most popular library implementation.

3.5.5 Ease of developing new applications

As we have seen so far, most genomics applications are built with the same set of kernels. However, currently, to develop a new application, the developer must write it from scratch. We now show the ease of developing new applications achieved using SARVAVID. Once such kernels are available, the application developer's task boils down to using the kernels in the right order and with the right language constructs. Table 3.4 shows the lines of code (LOC) required to implement the application in a general purpose language (C, C++) (for this we used the existing standard implementations) and in SARVAVID. Only the executed lines of code from these application packages were counted; the supporting libraries were skipped. For SARVAVID, we also count the LOC for the user-provided

partition-aggregate tasks. we see that SARVAVID allows the application to be expressed in significantly fewer lines, thus hinting that there are productivity benefits from using it.

Table 3.4.: Applications and their Lines-of-Code when implemented in SARVAVID and their original source. A lower LOC for applications written in SARVAVID indicate ease of development compared to developing the application in C or C++

Application (Source Language)	**Original LOC**	**Kernel SARVAVID LOC**	**Partition-Aggregate Functions LOC**	**Total SARVAVID LOC**
BLAST(C)	6815	47	315	362
MUMmer(C++)	5746	36	218	254
E-MEM (C++ & OpenMP)	550	40	218	258
SPAdes (C++,Python)	5985	70	-	70
SGA (C++,Python)	10475	92	-	92

3.5.6 Case study - Extending existing applications

Alignment applications generally take two inputs, a query sequence and a reference sequence. However, there is often a need to compare multiple genomic sequences, for example, to determine evolutionary distances between species. In such cases, alignment applications are simply executed in a pairwise all-against-all fashion to determine the degree of similarity between the species. To determine distances between different fruit fly species such as *D melanogaster* (171 MB), *D ananassae* (234 MB), *D simulans* (127 MB), *D mojavensis* (197 MB), and *D pseudoobscura* (155 MB), one would need to run MUMmer once for every pair of species *i.e.* C(5, 2) = 10 times. In SARVAVID, we can simply take the original SARVAVID-MUMmer implementation and wrap it in a doubly-nested loop to perform the pairwise comparisons. SARVAVID's compiler optimizations such as loop hoisting and CSE can then optimize the program, delivering an implementation akin to a hand-optimized version. The SARVAVID version using a doubly-nested loop runs 2.4X faster than pairwise MUMmer. This study demonstrates the power of SARVAVID: applications can be easily developed, extended, and optimized.

3.6 Related Work

To address the urgent need for developing new computational genomics applications and tools, many libraries have been proposed [53, 61–69]. These libraries provide ready-to-use implementations for common tasks in the domain. However, since these library-based approaches rely on general-purpose compilers, they cannot exploit optimizations across library calls. To our knowledge, there has been no work on developing DSLs for computational genomics applications. Below we review some of these libraries.

SeqAn [53] is an open-source C++ library of algorithms and data structures for sequence analysis. Similar to our approach, SeqAn identifies algorithmic components used commonly across genomic tools and packages them in a library. Libcov [64] library is a collection of C++ classes that provides high level functions for computational genomics. Libcov uses STL containers for ease of integration. Unlike SARVAVID, SeqAn and Libcov provide no opportunities for compiler optimizations or automatic parallelization.

The NCBI C++ toolkit [61] provides low-level libraries to implement sequence alignment applications using multithreading. The low-level nature of these libraries restrict their use to skilled developers, and limit their effectiveness in the hands of domain scientists; providing efficient applications often requires carefully composing these libraries and correctly making myriad design decisions. SARVAVID provides higher-level abstractions, making it easier to write genomics applications.

The Sleipnir [63] library provides implementations to perform machine learning and data mining tasks such as micro-array processing and functional ontology while GenomeTools [69] provides efficient implementations for structured genome annotations. These libraries provide no efficient datatypes / objects for genomic applications, so are not directly relevant to the domain SARVAVID targets.

Libraries such Sleipnir, Bio++ [62], and PAL [65] provide support for parallelization using multithreading on a single node. However, they do not offer support to scale beyond that. With the increasing biological data, it is imperative to speedup the development of

distributed genomics applications. SARVAVID's distributed runtime allows programs to scale across multiple nodes and to larger inputs.

3.7 Conclusion

Next generation sequencing is generating huge, heterogeneous, and complex datasets. To process this enormous amount of data, we need to develop efficient genomics applications rapidly. In addition, applications need to be parallelized efficiently. SARVAVID allow users to naturally express their application using the DSL *kernels*. This expedites the development of new genomics applications to handle new instrumentation technologies and higher data volumes, along with the evolution of existing ones. In addition, the SARVAVID compiler performs domain-specific optimizations, beyond the scope of libraries and generic compilers, and generates efficient implementations that can scale to multiple nodes. In this chapter, we have presented SARVAVID and showed implementations of five popular genomics applications, BLAST, MUMmer, E-MEM, SPAdes and SGA in it. We have identified and abstracted multiple building blocks, *kernels* that are common between multiple genomic algorithms. SARVAVID versions are not only able to match the performance of handwritten implementations, but are often much faster, with a speedup of 2.4X, 2.5X, and 2.1X over vanilla BLAST, MUMmer, and E-MEM applications respectively. Further, SARVAVID versions of BLAST, MUMmer, and E-MEM scale to 1024 cores with speedups of 11.1X, 8.3X and 11X respectively compared to 64 core runs of SARVAVID-BLAST, SARVAVID-MUMmer and SARVAVID- E-MEM.

4. SCALABLE GENOMIC ASSEMBLY THROUGH PARALLEL *DE BRUIJN* GRAPH CONSTRUCTION FOR MULTIPLE K-MERS

4.1 Introduction

The process of sequence assembly merges together sequence fragments or *reads* generated by sequencing machines for reconstructing the genome. It is a fundamental step in all kinds of genomic analyses. *De novo* assembly is the process of reconstructing the genome sequence without using a reference genome. It is a crucial process especially for new or uncharacterized species, and is computationally complex due to the large volume of reads, errors introduced by sequencing machines, repeating patterns in the original genome, and uneven sampling of the reference genome. With the rapid advancement of sequencing technologies projected to generate 2^{40} exabytes of data by 2025 just for the human genomes [37], there is a dire need for *de novo* assembly algorithms to play catch-up.

De Bruijn Graph based genome assembly

Currently the most popular de novo genome assembly method is the *de Bruijn Graph* (DBG) method [70], used by assemblers such as Velvet [71], ABySS [72], and ALLPATHS-LG [50]. This method constructs a *de Bruijn* graph G from the sequenced reads. A DBG is a directed graph whose vertices are length-k substrings, or *k-mers* of the reads. Two vertices u and v in the graph G are connected if they are consecutive k-mers in a read and the last $(k-1)$ nucleotides of the k-mer represented by u are the same as the first $(k-1)$ nucleotides of the k-mer represented by v [73].

The goal of assembly applications is to get long contiguous genomic sequences called *contigs* from the sequenced read datasets. Contigs are constructed by traversing the DBG to identify *maximal paths* in the graph. All vertices in any maximal path have in-degree and out-degree equal to 1, except for the vertices at the start and end of the paths. Scaffolding

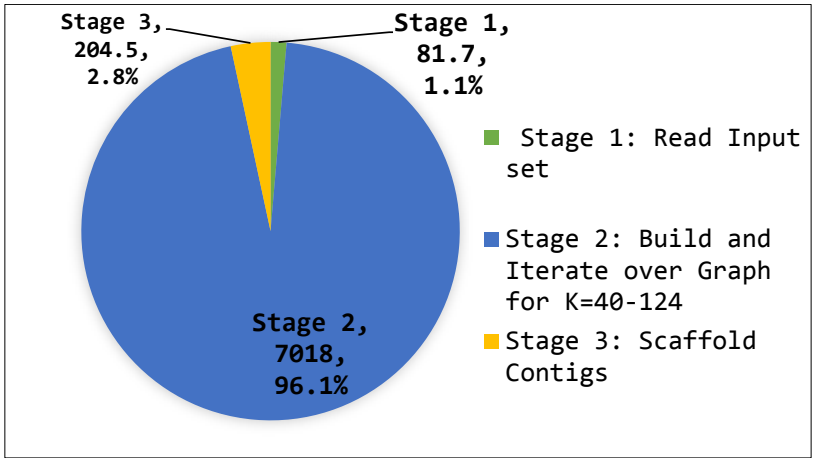

Fig. 4.1.: Distribution % of major stages in IDBA-UD, the time taken for each stage is provided in seconds for CAMI metagenomic dataset with 33 million paired-end reads of length 150, insert size 5kbp, with 8 k-values ranging from 40 – 124 with a step of 12.

techniques are further used to determine the relative order and orientation of these contigs to get even longer assembled regions called *scaffolds*.

Characteristics of the DBG structure are highly dependent on the value of k selected for graph construction. If the value of k is small, many false positive vertices and edges are introduced in the DBG due to repeats or erroneous reads. A vertex connecting to multiple other vertices introduces branches in the DBG and terminates the maximal path it belongs to, resulting in smaller-sized contigs. A large k-value, on the other hand, can differentiate among smaller repeats of length less than its length, and hence, reduce the number of branches. However, due to low or non-uniform sampling, some k-mers responsible for vertices and edges in the DBG are missed. This leads to a fragmented DBG with dead-end paths if reads covering consecutive k-mers are missing. The severity of the gap problem increases with increasing values of k. Thus, selecting the correct value for the k-parameter in DBG algorithms is crucial as it provides a balance between this *branching* and *fragmentation* problem.

Using multiple k-values

To achieve better assemblies, IDBA (Iterative DBG Assembler) [73], IDBA-UD [74],

SOAPdenovo2 [75], and SPAdes [54] use multiple k-values, iteratively, to assemble the sequenced reads. Essentially, a DBG with a small k-value can be traversed to infer what longer sequences *might look like*, allowing a DBG with a larger k-value to incorporate this additional information to "patch up" gaps. SPAdes follow an interative assembly graph construction process using multiple k-values to construct a multi-sized DBG. IDBA iterates from small to large values of k, maintaining an accumulated DBG to carry useful information forward as it moves on to higher k-values. These iterative approaches produce DBGs that generate superior assemblies than a DBG built with a single (large or small) k-value [54, 73–75]. IDBA-UD is an improved version of IDBA, and in the rest of the paper we only refer to IDBA-UD.

Motivation for ScalaDBG

Unfortunately, while these solutions can leverage multiple k-values to generate higher quality assemblies, the time taken for assembly increases linearly with the number of different k-values being used. Since IDBA-UD is memory-efficient and faster than SPAdes we chose to focus on IDBA-UD [76]. Figure 4.1 shows the time taken by IDBA-UD to execute the key stages of assembly: Reading the sequence file (stage 1), processing with multiple k-values—first building the graph and then iterating over the graph with $k = 40 - 124$ with a step of 12 to generate the contigs (stage 2), and then, scaffolding(stage 3) to get the final assembly. We invoked IDBA-UD on a metagenomic dataset part of the CAMI benchmark with 33 million paired-end reads of length 150, insert size 5kbp with the k-value range of $40 - 124$ and a step size of 12. We used 1 Intel Xeon E5-2670 2.6 GHz node with 16 cores and 32 GB memory for the experiment. We found from Figure 4.1 that the stage 2 of iterative graph-construction process, comprising of initial graph-build construction and iteration takes up 96.1% of the total workflow time. Clearly, the iterative graph-construction step dominates the total time taken for assembly.

Delving further in this experiment, we determine the relationship between the k-values, quality of the assembly, and the time taken for graph construction.The most common metric to assess assembly quality is *N50*. N50 is defined as the length of the smallest contig above which 50% of an assembly would be represented (or smallest scaffold if it is applied after

Table 4.1.: Relationship between k-values, Quality of Assembly, and Runtime for IDBA-UD running CAMI medium complexity metagenomic dataset

k-value range (count)	**step size**	**Total Time (sec)**	**N50**	**(%) improvement in N50**	**(%) increase in execution time**
40-124 (2)	84	3165	3547	-	-
40-124 (4)	28	4954	8255	132	56.5
40-124 (8)	12	7100	10729	202	124.3

scaffold construction). A higher N50 number indicates a better assembly. We used the same k-value range$(40-124)$ and dataset for IDBA-UD as in the above experiment, and varied the step size, to get 3 different configurations with 2,4, and 8 k-values shown in Table 4.1. As the number of k values in the range set is increased, the quality of the assembly improves. However, the total time taken also increases proportionately, *i.e.*, linearly in proportion with the number of different k values being used. This is an undesirable consequence of iterating through increasing number of k-values *sequentially*. Thus, although the N50 value of the assembly, when iterating using k-values of 40-124 with a step size of 12 is $3X$ that of using k-values 40 and 124, it takes 124.3% higher time for constructing the final graph with these k-values.

Our solution: ScalaDBG

To address this concern, we propose *ScalaDBG*, a new parallel assembly algorithm, which parallelizes stage 2 of Figure 4.1. The key insight behind ScalaDBG is that the graphs for multiple k-values need not be constructed serially. Instead, each graph construction can be done independently and in parallel. Accumulating the graph for the higher k-value, such as in IDBA-UD, introduces an apparent dependency on the graph with lower k-values. We remove this dependency, by introducing a *patching technique*, which can patch the higher k-valued graph (k_2) with contigs from the lower k-valued graph (k_1). Crucially, the first stage of construction of the k_1 graph and the k_2 graph can proceed in parallel and the relatively shorter stage of patching the k_2 graph with the contigs from the k_1 graph happens

subsequently. Thus, the gaps in the higher k-valued graph are *cemented* from the contigs of the lower k-valued graph, with the branches of the lower k-valued graph removed.

For a long chain of k-values, ScalaDBG first performs graph construction in parallel. For each pair of graphs, the higher k-valued graph is patched using the lower k-valued graph. Thus, ScalaDBG applies a divide-and-conquer approach, which can be visualized as in Figure 4.5. Multiple patch processes occur in parallel, *e.g.*, the patching of graph G_{k2} with contigs from graph G_{k1} proceeds in parallel with the patching of graph G_{k4} with contigs from graph G_{k3}(refer Figure 4.5). Each pair of adjacent graphs is patched to generate a single graph. This process is repeated until there is only a single graph. Thus, patching proceeds according to the *tree reduction* parallel pattern. In this way, a long chain of k-values can be broken down, with both construction and patching, happening in parallel. Thus, as the list of k-values grows longer, ScalaDBG identifies greater scope for parallel execution. ScalaDBG is the first assembler to parallelize multi-k value de Bruijn graph construction. We show in our evaluation that there is no statistical difference in the assembly quality of ScalaDBG and IDBA-UD.

The genomic analysis pipeline starts with generation of genomic data from sequencing machines, followed by the kernels of genome assembly and subsequent analysis of the assembled sequences. Tremendous improvements in sequencing machines have made the kernels of genome assembly and analysis the bottleneck for extracting knowledge from raw genomic datasets. A high latency genome assembly kernel negatively affects the subsequent analysis kernels and overall performance of the pipeline. Hence ScalaDBG targets to speeds up this module using cheaper compute nodes that are easily accessible to the community in the form of a public cloud. The scalability of IDBA-UD, the most popular iterative DBG assembler, is severely limited by the memory and compute capacity of server nodes, which are very costly to upgrade. By leveraging relatively lower capacity nodes having modest hardware, ScalaDBG speeds up assembly for a much lower cost.

Several steps within each graph construction stage, such as k-mer counting, indexing, and lookup, occur in parallel. We borrow IDBA-UD's strategy and implementation for graph construction for a *single* k-value. Thus, ScalaDBG exploits parallelism at 2 levels—

a coarse-grained level and a fine-grained level—by constructing graphs with multiple k-values, in parallel, while also parallelizing the individual graph-construction processes. This strategy of ScalaDBG enables us to leverage the *hybrid MPI-Open-MP parallel programming model*. While each MPI process can independently perform graph construction on different nodes in a cluster, Open-MP threads can exploit all cores on a single node. Thus, ScalaDBG can leverage the power of a robust server, with multiple processors and a large RAM, for *vertical scaling or scale-up* as well as a cluster with multiple nodes, for *horizontal scaling or scale-out*.

We make the following technical contributions in this paper:

1. We break the dependency in DBG creation for multiple k-values—from a purely serial process to one where the most time-consuming part (the DBG creation for individual k-values) is parallelized. This innovation can be applied out-of-the-box to most DBG-based assemblers.

2. We develop a divide-and-conquer strategy for handling a long chain of k-values, which improves the quality of assembly, while efficiently utilizing all available machines in a cluster and all available cores on a machine.

3. We develop a software package ScalaDBG that uses OpenMP for scale-up within one server and MPI for scale-out across multiple servers. The software package is available through `https://github.com/purdue-dcsl/Scaladbg`.

Key experimental results

IDBA-UD assembles the CAMI metagenomic dataset for a k-value range of $40 - 124$, with a step of 12 in 2.2 hours on a single node. With ScalaDBG, we assemble the same dataset in 1.1 hours on 8 such nodes, which is 2X faster.

Outline: Section 4.2 decribes IDBA-UD, and provides examples of using multiple k values in graph construction, Section 4.3 discusses details of ScalaDBG's strategy for breaking dependencies in build and reduction tree, Section 4.5 discusses implementation details, Section 4.6 evaluates ScalaDBG, and Section 4.8 concludes.

4.2 Background

In this section, we provide background on the creation of DBGs using multiple k values and an overview of IDBA-UD.

4.2.1 Using Multiple k-values in Iterative Graph Assembly

Figure 4.2 shows the effect of using a small k ($k = 3$), and a larger k ($k = 4$) during DBG construction. Figure 4.2 (a) shows the graph constructed from read set with $k = 3$. The vertices are consecutive 3-mers of the read set. They are connected to each other if they have a 2-mer overlap. This graph has branching at vertex ACG due to repeating region in the genome *ACGT* and *ACGA*. The contig set generated by identifying maximal paths in the graph is {*AATGCCGT,ACGAA,ACGT,CGTACG*}. As the value of k is increased to 4, the branch disappears as the higher k-value can now distinguish between the repeat region in *ACGT* and *ACGA*. However, some reads such as *CCGTA* and *GTACG* are are not sampled from the genome sequence and so vertices and edges in the graph are missed. Hence *GTAC* and *TACG* cannot be connected. Thus, Figure 4.2 (b) with $k = 4$ has gaps in it. In general, DBG built using a lower k value has multiple branches, and DBG built using a higher k value has gaps. The contig set generated for $k = 4$ is {*TACGTAC, TACGAA, AATGCCGT*}. If we can take the graph built with $k = 4$ and augment it with the contigs from the $k = 3$ graph, then it is conceptually possible to arrive at the graph shown in Figure 4.2(c). In Figure 4.2(c), an edge is added between circled vertices *GTAC* and *TACG* due to presence of the substring *GTACG* in the contig set obtained using $k = 3$. After adding the edge, and traversing the composite graph, contigs longer than those created from both $k = 3$ and $k = 4$ are obtained : {*TACGTAC***G**,*TACGAA,AATGCCGT*}.

4.2.2 IDBA-UD

IDBA-UD is a de Bruijn graph assembler. It iterates on a range of k values from $k = k_{min}$ to $k = k_{max}$, with a step-wise increment of s. It maintains an accumulated de Bruijn graph H_k at each step. In the first step a de Bruijn graph $G_{k_{min}}$ is generated from the input reads.

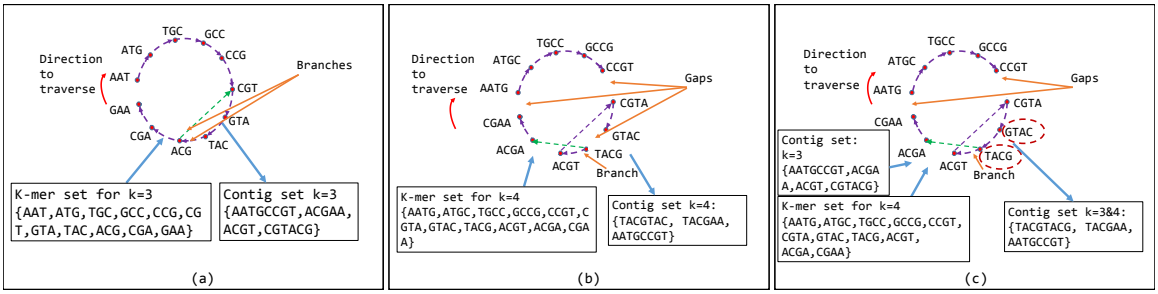

Fig. 4.2.: Desired Genome Sequence : **AATGCCGTACGTACGAA**, Read Set : *AATGC, ATGCC, GCCGT, TGCCG, CGTAC, TACGT, ACGTA, TACGA, ACGAA* De Bruijn Graph for $k = 3$ (sub-figure (a)) and $k = 4$ (sub-figure (b)). The final graph (sub-figure (c)) can be created by filling in some of the gaps in the $k = 4$ graph with contigs from the $k = 3$ graph. The vertices for which new edge is added (sub-figure (a)) are circled. Traversing this final graph results in the final contig set.

For $k = k_{min}$, H_k is equivalent to $G_{k_{min}}$. At any step, contigs for graph H_k are generated by considering all maximal paths in graph H_k. All vertices in any maximal path have an in-degree and out-degree equal to 1 except the vertices at the start and end of the path. All reads from the input set that are substrings of these contigs are removed, reducing the input size at each step. A read of length r generates $r - k + 1$ vertices. As k is increased each read will introduce fewer vertices. This reduction in input read set coupled with the fact that there are fewer vertices for larger k values, makes subsequent graph constructions less time consuming.

The inputs to the next step, where $k = k_{min} + s$ consist of the graph H_k, the remaining reads, and the contigs from H_k. Each path of length s in H_k is converted to a vertex. All such vertices are connected by an edge if the corresponding $(k + s + 1)$-mer exists in either the remaining reads or the contigs of H_k. This process is repeated for each subsequent iteration until $k = k_{max}$ is reached. Note that in this algorithm at iteration i, graph $H_{k_{min}+i*s}$ depends on graph $H_{k_{min}+(i-1)*s}$ obtained at iteration $i-1$, the reduced read set, and the contigs obtained at the iteration $i - 1$.

This dependency forces IDBA-UD to operate sequentially on the chain of k-values, no matter how long the chain is. This is the crux of the problem that we address through ScalaDBG.

IDBA-UD also uses existing graph simplification strategies such as dead end removal and merging bubbles to get longer final contigs. In its dead end removal phase, IDBA-UD removes short simple paths leading to dead ends. In the bubble merging phase, several similar sequences are merged into one sequence. A *bubble* is formed when several similar paths have the same start and end vertex [71]. These bubbles can be formed due to errors in reads, generating similar paths with few differences. Bubble merging and dead end removal phases throw away vertices and edges from the graphs for simplification. At the end, scaffolding techniques are applied to the contigs of H_{kmax} to get the final set of contigs.

4.3 Design of ScalaDBG

In this section we describe the design details of ScalaDBG, considering the build phase (building a DBG) and the patch phase (patching a partial DBG with contigs from a lower k-value DBG). We also outline the scheduling algorithm used in ScalaDBG to maximize utilization of nodes in a cluster.

4.3.1 Build Phase

To simplify the exposition, we describe our protocol first using just two different k values, $k1$ and $k2$, with $k1 < k2$. Figure 4.3 shows the stages of ScalaDBG for the two k values. In the build phase, DBGs are built for each k value in parallel, and the build module generates G_{k1} and G_{k2}. The graph construction is the most time consuming phase of the entire pipeline, with the construction of G_{k1} taking the most time.

Vertices in G_{k1} and G_{k2} are all $k1$-mers and $k2$-mers obtained from the original input read set I, respectively. Vertices and edges are established in G_{k1} and G_{k2} as per the definition of DBG. We denote the number of vertices in G_{k1} and G_{k2} as $|G_{k1}|$ and $|G_{k2}|$ respectively. Graph G_{k1} will typically be larger in size, in terms of vertices and edges, *i.e.*,

$|G_{k1}| > |G_{k2}|$, since $k1 < k2$. Importantly, the creation of the DBGs for the two different k-values proceeds in parallel, unlike in all prior protocols. Now G_{k1} will have a higher number of branches than G_{k2}, while G_{k2} will have a higher number of gaps than G_{k1}. We generate contigs C_{k1} from G_{k1} by finding maximal paths according to standard practice (we use the IDBA algorithm specifically) but we do *not* yet proceed to generate C_{k2} from G_{k2}.

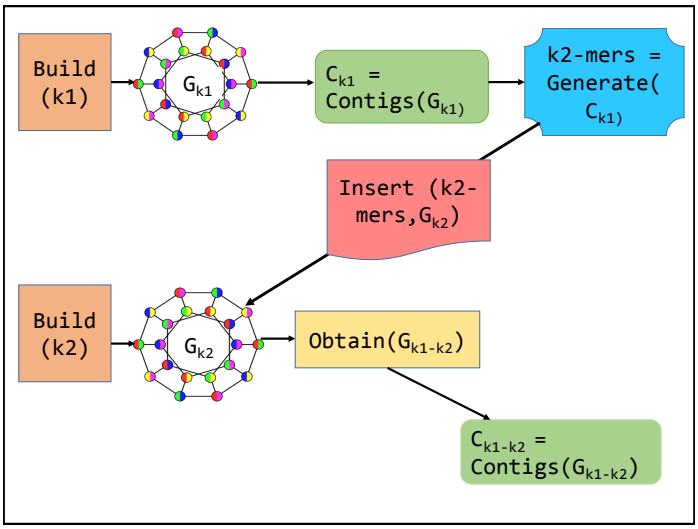

Fig. 4.3.: High Level Architecture Diagram of ScalaDBG. This shows the graph construction with only two different k values, $k1$ and $k2$ with $k1 < k2$. The graph G_{k2} is "patched" with contigs from G_{k1} to generate the combined graph G_{k1-k2}, which gives the final set of contigs. Different modules in ScalaDBG are highlighted by different colors.

4.3.2 Patch Phase

We cannot simply create contigs from graph G_{k2} because it will have gaps relative to the graph G_{k1}. Therefore, our idea is to *patch* graph G_{k2}, *i.e.*, fill the gaps in the graph by bringing in more vertices and connecting the new vertices plus the existing vertices with additional edges. *The fundamental insight that we have is that contigs C_{k1} have the information to close some of these gaps.* We process C_{k1} by generating $k2$-mers from sequences in C_{k1}, *i.e.*, we generate all subsequences of length $k2$ from all contigs in the

C_{k1} set. These $k2$-mers are inserted into the G_{k2} graph as vertices. The introduction of new vertices in the graph can result in the introduction of new edges. Two vertices u and v in the graph G_{k2} are connected by an edge if the last $(k2-1)$ nucleotides of the $k2$-mer represented by u are the same as the first $(k2-1)$ nucleotides of the $k2$-mer represented by v, and u and v are consecutive k-mers in the contig set C_{k1} or read set. The resulting graph is denoted as G_{k1-k2}. Thus, G_{k1-k2} is the aggregated graph obtained by filling gaps of G_{k2} using C_{k1}. The final contig set is generated from G_{k1-k2}.

Example for ScalaDBG

Consider an example to illustrate the build and patch phases of ScalaDBG with k values of 3 and 4. The graphs G_3 and G_4 are obtained by $k_1 = 3$, and $k_2 = 4$ and are shown in Figure 4.2(a) and (b). In ScalaDBG, during the build phase, G_3 and G_4 are constructed in parallel. Contig set C_3 obtained from graph G_3 is {*AATGCCGT,ACGAA,ACGT,CGTACG*}. In the Patch phase of ScalaDBG, graph G_4 is patched with contigs in C_3. 4-mers obtained from C_3 are {*AATG, ATGC, TGCC, GCCG, CCGT, ACGA, CGAA, ACGT, CGTA, GTAC, TACG*}. Of these 4-mers, if two consecutive 4-mers are substrings of a sequence in the contig set, they are inserted as vertices in G_4 and are connected by an edge. Vertices $GTAC$ and $TACG$ are connected by an edge in G_4 to get G_{3-4}. G_{3-4} is shown in Figure 4.2 (c). Final contig set C_{3-4} generated from G_{3-4} is $\{TACGTACG, TACGAA, AATGCCGT\}$. This is the same graph that we arrive to in Section 4.2.

4.3.3 Patching Multiple *k* Values in Parallel

When the number of k values to be iterated over is greater than 3, *ScalaDBG has two options while patching*. It can adopt either *a serial method* shown in Figure 4.4, or *a parallel method* shown in Figure 4.5. Figure 4.4 shows the serial patching process when there are four different k values, $k1,\ k2,\ k3,\ k4$, and $k1 < k2 < k3 < k4$. Initially, graphs for each of the 4 k values are generated in parallel. In the serial variant of ScalaDBG, the graph associated with the lowest k value, $k = k1$ is assembled, and contigs are generated from the graph G_{k1}. The obtained contigs are used to patch the graph associated with the next

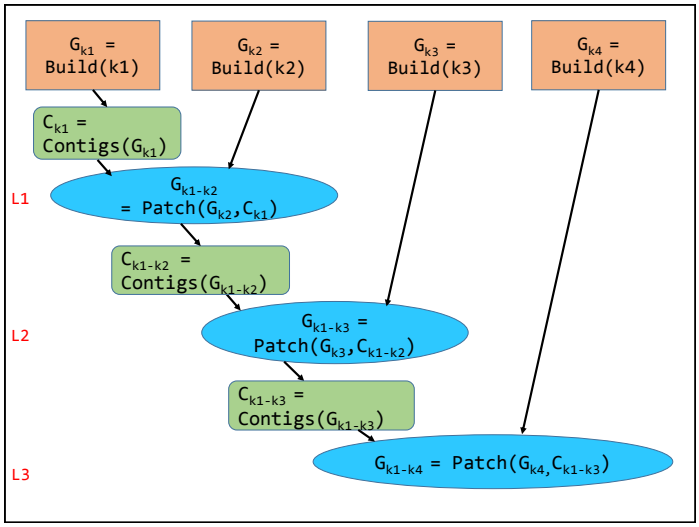

Fig. 4.4.: Schematic for ScalaDBG using serial patching, called ScalaDBG-SP

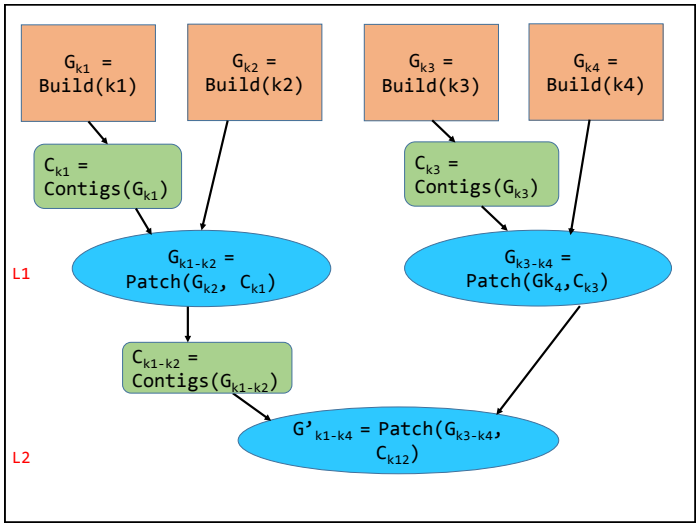

Fig. 4.5.: Schematic for ScalaDBG using parallel patching, called ScalaDBG-PP.

higher k value ($k2$). Contigs are generated using the higher k valued graph. This process is repeated, serially for each increasing higher k value, until the graph associated with the highest k value in the chain is patched and assembled. The final set of contigs is generated from this final patched graph. Thus, the graph building for each separate k value occurs in parallel but the patching and generating contigs occurs serially. The advantage of the serial

patching method is its simplicity, owing to the simple communication patterns between processes operating over the different k values. However, as the number of different k values increases, the number of serialized patching steps grows *linearly* with it. The serialized patching process starts dominating the total time of the workflow for ScalaDBG. To resolve this bottleneck, in the patch phase, ScalaDBG in the parallel mode allows multiple patch operations to occur in parallel.

The insight behind our parallel method is that multiple patch operations, which are independent of each other, can proceed in parallel. Thus, the patching process proceeds like a reduction tree. Figure 4.5 shows the parallel method of patching, where multiple patch processes occur in parallel, *e.g.*, the patching of graph G_{k2} with contigs from graph G_{k1} proceeds in parallel with the patching of graph G_{k4} with contigs from graph G_{k3}. Each pair of adjacent graphs is patched to generate a single graph. This process is repeated until there is only a single graph. Thus, patching proceeds according to the *tree reduction* parallel pattern. In this way, a long chain of k-values can be broken down, with both graph construction and patching happening in parallel. Thus, as the list of k values grows longer, ScalaDBG identifies greater scope for parallel construction while making use of more compute nodes. In the parallel patch method, the number of serialized patching steps grows only *logarithmically* with the number of different k values.

In the rest of the sections, we refer to ScalaDBG using serial patching as *ScalaDBG-SerialPatch* (or ScalaDBG-SP), and ScalaDBG using parallel patching as *ScalaDBG-ParallelPatch* (or ScalaDBG-PP). Between the serial and parallel patch methods, different pairs of graphs are merged with each other. Hence, the final contigs generated by the two methods may differ. In general, the assembly quality of the serial method is higher since the difference between k values associated with adjacent graphs is smaller and it has been shown that small jumps in the k-values leads to better quality aggregated DBGs [73].

4.3.4 Efficient scheduling of multiple k-values

ScalaDBG includes a scheduler that can run a single assembly comprising any number of k-values on any number of computational nodes. The ScalaDBG scheduler creates a greedy schedule to maximize the utilization of nodes in the cluster. We discuss the operation of the scheduler for the tree reduction pattern of ScalaDBG-PP (ScalaDBG-SP is a simpler form of the schedule and is omitted here for space). The generated schedule consists of a set of *rounds* as shown in Figure 4.6. Each round is the assignment of a *task* to a node, where a task means the node builds a graph or patches an existing graph with contigs from another graph, or the node is idle. The rounds continue until the final graph is obtained.

The scheduler uses the following observation in creating the schedule: building graph for $k1$ will take longer than building graph for $k2$, where $k1 < k2$ because $|G_{k1}| > |G_{k2}|$. Hence the node processing $k2$ will get done earlier, and asynchronously send graph G_{k2} to the node building graph G_{k1}, thus hiding the latency of the communication. This node is then free to take up the next task, either creating a new graph or patching an existing graph. The scheduler consistently overlaps computation with communication in this manner. Similarly, nodes patching higher k-valued graphs will finish their tasks earlier than nodes patching lower k valued graphs. Hence these nodes will asynchronously send the patched graphs, and start building the graph for the next k value. The order of patching dictates the node that will receive a graph. A node that receives a graph always performs patching as the next task, while a node that sends a graph starts building a new graph for the next k value. The tasks are assigned such that the number of idle nodes in any round is reduced. The amount of work done in each round is not necessarily the same across the nodes, *e.g.*, in round 1, node 4 does less work than node 3, which does less work than node 2, which does less work that node 1.

We use the following example to explain the ScalaDBG scheduler. In this example, there are 4 available nodes ($n1, n2, n3, n4$) and 8 input k values, $k1 - k8$, with $k1 < k2 <$

... < $k7$ < $k8$. We explain the processing done at each node during each round with the representation shown in Figure 4.6.

In round 1, all nodes build new graphs from the k values $k1 - k4$. Once done with the graph construction, node $n2$ sends graph G_{k2} to node $n1$ and node $n4$ sends graph G_{k4} to node $n3$. In round 2, node $n1$ does the patching to generate graph G_{k1-k2} and similarly node $n3$ does a patching while nodes $n2$ and $n4$ build new graphs. This way the different rounds continue till the final graph G_{k1-k8} is assembled in node $n1$. Note that in the later rounds, some of the nodes become idle as there are no more tasks to schedule. In this

	Nodes			
Round	***N1***	***N2***	***N3***	***N4***
1	G_{k1} = Build(k1) C_{k1} = Contigs(G_{k1})	G_{k2} = Build(k2) Send(G_{k2}) → N1	G_{k3} = Build(k3) C_{k3} = Contigs(G_{k3})	G_{k4} =Build(k4) Send(G_{k4}) → N3
2	G_{k1-k2} = Patch(C_{k1},G_{k2}) C_{k1-k2} = Contigs(G_{k1-k2})	G_{k5} = Build(k5) C_{k5} = Contigs(G_{k5})	G_{k3-k4} = Patch(C_{k3},G_{k4}) Send(G_{k3-k4}) → N1	G_{k6} = Build(k6) Send(G_{k6}) → N2
3	G_{k1-k4} = Patch(C_{k1-k2}, G_{k3-k4}) C_{k1-k4} = Contigs(G_{k1-k4})	G_{k5-k6}= Patch(C_{k5},G_{k6}) Send(G_{k5-k6}) → N1	G_{k7} = Build(k7) C_{k7} = Contigs(G_{k7})	G_{k8} = Build(k8) Send(G_{k8}) → N3
4	G_{k1-k6} = Patch(C_{k1-k4}, G_{k5-k6}) C_{k1-k6} = Contigs(G_{k1-k6})		G_{k7-k8} = Patch(C_{k7}, G_{k8}) Send(G_{k7-k8}) → N1	
5	G_{k1-k8} = Patch(G_{k1-k6}, G_{k7-k8}) C_{k1-k8} = Contigs(G_{k1-k8})			

Fig. 4.6.: Schedule created by the ScalaDBG Scheduler for 8 k-values and 4 nodes. Different computational nodes in the cluster execute different tasks in each round of the workflow.

manner, arbitrary number of k values are scheduled to run on a set of nodes by ScalaDBG' s scheduler.

4.4 Correctness of ScalaDBG Methodology

In this section, we first establish the equivalence of graphs obtained by the build and serial patching process of ScalaDBG and the iterative build process of IDBA-UD. We then

discuss the implications of the parallel patching and the graph simplication procedures on the output of ScalaDBG.

Theorem 4.4.1 (Equivalence between graph obtained in IDBA-UD and ScalaDBG- SP) *For a fixed iteration set of k-values starting from k = kmin to k = kmax, the final graph obtained by ScalaDBG-SP and IDBA-UD is identical.*

Proof We use the principle of Mathematical Induction to establish the equivalence of the resulting graph in ScalaDBG- SP and IDBA-UD.

Initial Step Let R represent the initial read set input to ScalaDBG- SP and IDBA-UD. When $kmax = kmin$, the final graph obtained after the first iteration is the final graph. There is no patching involved. ScalaDBG-SP and IDBA-UD generate $kmin - 1$-mers from R and use the same build procedure to generate graph G_{kmin} which is the final graph.

Inductive Step For $kmax = K$, where $K > kmin$, the graph obtained using ScalaDBG-SP is identical to IDBA-UD. We denote this graph by G_K. We must prove, the statement is true for $kmax = K + 1$.

The graph obtained by ScalaDBG-SP after constructing graphs from $k = kmin$ to $k = K$ in parallel, followed by serial patching is G_K. ScalaDBG-SP patches the graph G_{K+1} using contigs generated fromG_K to get final graph P_{K+1}. IDBA-UD generates G_K at the end of $k = K$ iteration. After the next $K + 1$ iteration, it generates H_{K+1} as the final graph. We need to show that $P_{K+1} = H_{K+1}$.

In IDBA-UD, to construct H_{K+1} from G_K , first potential contigs in G_K are constructed by identifying *maximal paths* $v_1, v_2, \cdot, \cdot, \cdot, v_p$. All vertices in a maximal path have in-degree and out-degree equal to 1 except v_1 and v_p which may have in-degree 0 and out-degree 0, respectively. Note that a path of p vertices represents a potential contig of length $K + p - 1$. Let the contig set of G_K be denoted by C_G_K. Let R_K represent the read set of IDBA-UD at beginning of iteration $K + 1$. All reads in R_K that can be represented by potential contigs in C_G_K i.e. those reads that are substrings of a contigs in set C_G_K are removed. Let this new read set be denoted by R_{K+1}. Thus,
$\{R_{K+1} = R_K - r\}, \forall r \in R_K$, r is a substring of a contig in C_G_K. In the construction of H_{K+1},

only the reads in R_{K+1} and the potential contigs of G_K stored in C_G_K are considered. H_{K+1} consists of vertices formed using edges in G_K where each edge (vi, vj) in G_K is converted into a vertex, representing a $(K + 1)$-mer, if the $(K + 1)$-mer is a substring of a contig in C_G_K.

ScalaDBG-SP starts with the original read set R and generates all $(K + 1)$-mers of the reads. Graph G_{K+1} is built using R. Contig set C_G_K (same graph will generate same contig set, inductive step) is built using G_K. Then ScalaDBG-SP extracts $K + 1$-mers from each contig in the set C_G_K and inserts it into G_{K+1}. Vertices and edges in H_{K+1} in IDBA-UD obtained by upgrading vertices in G_K are $(K + 1)$-mers of contigs in C_G_K. Hence ScalaDBG-SP is guaranteed to insert these vertices and edges as we create $(K + 1)$-mers from the contigs. All vertices and edges in P_{K+1} are formed using the original read set R and the contigs of G_K. Now R_{K+1} is a proper subset of R. So all vertices and edges inserted using R_{K+1} will be inserted by R. Hence all vertices and edges formed using $R_{K+1} + C\_G_K$ in H_{K+1} will be formed using $R + C\_G_K$ in P_{K+1}. Thus P_{K+1}, formed by patching together G_K and G_{K+1} is identical to H_{K+1}.

From *Initial Step*, *Inductive Step* and principle of mathematical induction, IDBA-UD and ScalaDBG- SP generate identical graphs.

■

4.4.1 Implications of ScalaDBG's methods

Although the process of build and serial patch process of ScalaDBG essentially generates a graph identical to IDBA-UD, the assembly metrics of ScalaDBG- PP, ScalaDBG-SP and IDBA-UD differ. There are primarily two reasons for the differences : i) the out of order patching in ScalaDBG-PP and ii) the graph simplification procedures (bubble merging and dead end removal).

The order in which graphs built with different k values is significantly different for ScalaDBG- PP than ScalaDBG-SP and IDBA-UD. While IDBA-UD iterates over the k values in an increasing order, with a maximum of *step-size* difference between the itera-

tions, ScalaDBG-PP follows tree-style reduction to patch the graphs. Hence the difference in k values of two patched graphs will be higher, and the order of patching will also be different.

The difference in output of ScalaDBG-SP, ScalaDBG-PP and IDBA-UD is also due to the graph simplification procedures applied before generating the contigs. The bubble merging and dead-end removal phases remove incorrect vertices and edges in the graph based on their multiplicity information. The graphs obtained by ScalaDBG and IDBA-UD workflows, although identical, have different multiplicity information for their vertices and edges. Hence the graph simplification procedures generate different contig sets with different assembly metrics for ScalaDBG-SP, ScalaDBG-PP and IDBA-UD. However, in our evaluation section, we will show that the difference is not statistically significant.

4.4.2 Generality of ScalaDBG's methods

ScalaDBG breaks down the kernels of building the graph, patching the graph with contig set, and generation of contigs. Hence, the technique can be used to combine the contigs of a single k-value assembler such as Velvet. We use an example to explain this application shown in Figure 4.7. There are four different k values, $k1,\ k2,\ k3,\ k4$, $k1 < k2 < k3 < k4$. Initially a single k-value assembler is run for each of these k-values in parallel to generate the contig sets. The obtained contig sets are used to patch the graph associated with the contig set of the next higher k value ($k2$). Graphs G_{k2} and G_{k4} are built from the contig sets C_{k2} and C_{k4} respectively, according to the definition of DBG, considering the contig set as the input read set. Then the graphs G_{k2} and G_{k4} are patched using contig sets C_{k1} and C_{k3} respectively. After this stage, the method follows the standard parallel patch workflow of ScalaDBG. Note that the technique can also be applied using serial patch.

4.5 Implementation

We implement ScalaDBG using OpenMP (Version 4.0) (for parallelism within a node) and MPI (MVAPICH 2.2) (for parallelism across nodes in a cluster), compiled with GCC

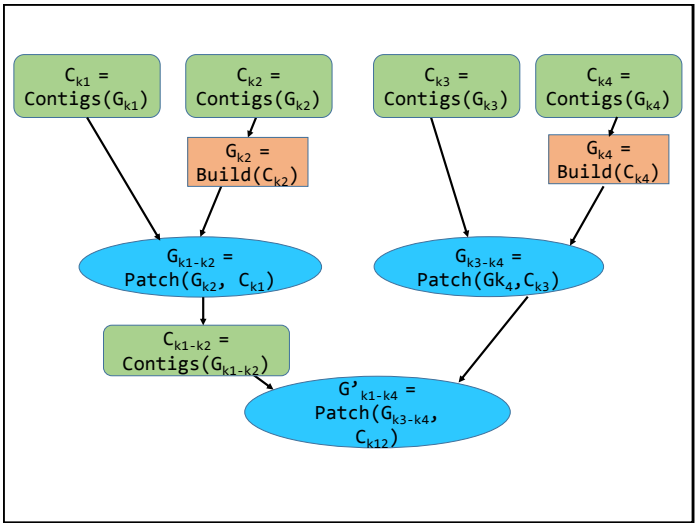

Fig. 4.7.: General assembler used in conjunction with ScalaDBG's technique.

version 4.9.3. The relevant IDBA-UD configuration parameters are: *option.mink*: minimum k-mer size, *option.maxk*: maximum k-mer size and *option.step*: increment between each k-mer graph built. We refer to each MPI process by its rank. N MPI processes have ranks from 0 to $N-1$. MPI process with rank 0 is referred to as the Master. The pseudocode for ScalaDBG- SP is shown in Algorithm 1 and ScalaDBG- PP is shown in Algorithm 2.

In Algorithm 1, the work is split up into n chunks, where $n = number\_of\_kmers$. Each rank will compute the *kmer_size* it will work on, then read in the input files and build a DBG. If the rank is not the Master, it will serialize and send its DBG to the Master. The Master will receive all other graphs and patch them in serial order, as shown at a high level in Figure 4.4. Intermediate graph representations are written and read from the Lustre Parallel File System.

In Algorithm 2, the process is the same as Algorithm 1, up through building a DBG. For the next stage we split up the ranks with half receiving while the other half sends as in Figure 4.5. After a rank sends its DBG, it is no longer used in the patching. This stage is repeated until there is only 1 rank that receives, which will always be the Master. The Master will then complete the last assembly and perform contig generation.

Algorithm 1 ScalaDBG with serial patching

1: **for** *All ranks* **do**
2: *read input_file(s), using multiple threads* ▷ We borrow IDBA implementation here.
3: $kmer\_size \leftarrow rank * option.step + option.mink$
4: **if** $kmer\_size \leq option.maxk$ **then**
5: *Build DBG using multiple threads*
6: **end if**
7: **end for**
8: **if** $rank \neq MASTER$ **then**
9: *Serialize DBG and Send to* Master
10: **else**
11: Master *Receive DBG* G_i
12: Master *generate Contigs of own DBG* C_{self} ▷ MASTER does this in parallel
13: **end if**
14: **if** $rank = MASTER$ **then**
15: **while** $kmer\_size < option.maxk$ **do**
16: $kmer\_size+ = option.step$
17: *Patch received DBG* G_i *with contigs* C_{self} *to get* G_a
18: $C_{self} \leftarrow$ *contigs of* G_a
19: **end while**
20: **end if**

Algorithm 2 ScalaDBG with parallel patching

1: **for** $Allranks$ **do**
2: $read\ input\_file(s)$, $using\ multiple\ threads$ ▷ We borrow IDBA implementation here.
3: $kmer\_size \leftarrow rank * option.step + option.mink$
4: **if** $kmer\_size \leq option.maxk$ **then**
5: $Build\ DBG\ using\ multiple\ threads$
6: $level \leftarrow 1$
7: **while** $(2^{level} < number\_of\_kmers)$ and $(rank\%2^{level} == 0)$ **do**
8: **if** $(rank + 2^{level-1} < number\_of\_kmers$ **then**
9: $Receive\ DBG\ from\ rank + 2^{level-1})$ ▷ Receive and Assemble done in parallel
10: $level++$
11: **if** $rank \neq highest\_receiving\_rank$ **then**
12: $Assemble\ our\ DBG$
13: **end if**
14: $Patch\ received\ DBG\ G_i\ with\ contigs\ C_{kmer\_size}\ to\ get\ G_a$
15: **end if**
16: **end while**
17: **if** $rank \neq MASTER$ **then**
18: $Serialize\ DBG\ and\ Send\ to\ rank - 2^{level-1}$ ▷ $level$ will be different for each rank
19: **else**
20: $Patch\ last\ DBG\ G_i\ with\ contigs\ C_{kmer\_size}\ to\ get\ G_a$
21: **end if**
22: **end if**
23: **end for**

4.6 Evaluation

In this section we evaluate ScalaDBG in comparison to IDBA-UD in terms of the time taken to perform the assembly and the quality of the assembly. We used two different types of sequencing read sets - metagenomic and single cell sequencing, as they are typically assembled by the genomics community using the iterative de Bruijn graph approach.

4.6.1 Evaluation Setup and Data Sets

We performed our experiments on an Intel Xeon Infiniband cluster. Each node had Intel Xeon E5-2670, 2.6 GHz with 16 cores per node, and 32 GB of memory. The nodes were connected with QDR Infiniband. We used the latest version of IDBA-UD (1.1.1) [74]. We used the read sets listed in Table 4.2. We obtained the *S. aureus* and SAR 324 single-cell datasets from [77]. The metagenomics datasets were obtained from the CAMI benchmark [78]. The number of nodes in the cluster are equal to the number of different k-values in the configuration for ScalaDBG, while IDBA-UD can only run on a single node. ScalaDBG outputs contigs for a given dataset. Existing scaffolding techniques can be applied to these output contigs to get the final assembly. We only focus on evaluating the performance and accuracy metrics of the assembled contigs in the following experiments for IDBA-UD and ScalaDBG since our contribution is confined until that stage.

Table 4.2.: Read Sets used in the Experiments. PE denotes Paired End reads

Name	Read Set Type	Read Length	# of reads	Characteristics
RM1	Real, Metagenomic	150 bp	33140480	PE,Insert size:5 kbp
RM2	Real, Metagenomic	150 bp	33128228	PE,Insert size:5 kbp
SC-*E.coli*	Real, Single Cell	100 bp	23,818,596	PE,Insert size:266 bp
SC -*S. aureus*	Real, Single Cell	100 bp	66,997,488	PE,Insert size:214bp
SC-SAR324	Real, Single Cell	100 bp	55,733,218	PE,Insert size:180bp

4.6.2 Relevance of Datasets

The field of metagenomics encompasses the sequencing and analysis of the total microbial DNA, sampled directly from the environment. This culture-independent analysis of the metagenome, *i.e.*, of the genetic material from this microbial potpourri, has transformed the study of microbial communities, affording the ability to study greater-than-99% of unculturable prokaryotes, present in various environments. In recent years, DNA sequencing has become significantly more affordable and widespread, and the realized potential of exploring the microbiome in different ecosystems, including in the human gut, has strongly motivated research in metagenomics [79].

A single cell is the ultimate denomination in a multicellular organism. For example, the human body consists of roughly 37.2 trillion cells living in harmony. However, in cancer, this harmonious equilibrium is lost and this is where even one single cell can wreak havoc by evolving into a malignant tumor mass, wherein the lineages diverge and form distinct populations giving rise to what is known as clonal diversity. While in the past, technological limitations required micrograms of input tissue mass resulting in an *average* signal being emanated from a complex mass of heterogeneous cell types, single-cell sequencing (SCS) methods can now revolutionize the understanding of cancer biology, affording insights into the role of rare cells in the evolution of cancer. In the case of metagenomic and single cell sequencing datasets, sequencing depths of different regions of a genome, or genomes from different organisms are exceedingly uneven.Hence multiple k-values are required for accurately assembling the datasets. So we evaluate ScalaDBG and IDBA-UD using these relevant datasets.

4.6.3 Performance Tests

Figures 4.8, 4.9, 4.10, 4.11, and 4.12 show the time taken by IDBA, ScalaDBG-SP, and ScalaDBG-PP to generate contigs from the 5 different read sets mentioned in Table 4.2. The performance test brings out the effect of different k-values on performance. For the metagenomic datasets we changed the step size to get three different configurations. We

used step sizes of $28, 12$, and 6 in the range $40 - 124$ (we give step sizes in reverse order because this corresponds to an increasing number of k-values). For the single cell datasets, we used step sizes of $14, 6$, and 3 in the range of $29 - 71$. The lower and upper bounds of the range is lower for the single cell dataset since its reads are shorter in length. We also ran SAR324 in the range of $20 - 50$ with step of $10, 5$, and 2 to get $4, 7, 16$ k values respectively.

The first 3 configurations have successively higher number of k-values: 4, 8,and 15, and are meant to evaluate the effect on quality of assembly and running time as the number of k values is increased. In addition the range extremes are held constant to obtain the information obtained from the two extreme k values. We ran ScalaDBG by matching the number of nodes in the cluster with the number of distinct k-values in the configuration to maximize scaling out performance for ScalaDBG. IDBA-UD on the other hand can only run on a single node. We report the overall execution times for both IDBA and ScalaDBG for generating the final contigs from the input readset.

We see that the speedup for ScalaDBG-PP and ScalaDBG-SP over IDBA increases with increase in the number of k values, for all the read sets. Speedup of ScalaDBG completely depends on the specific k values chosen. For the SAR324 dataset in the range of 20-50 with step size of 2, speedup of ScalaDBG- PP is 6.7X and speedup of ScalaDBG-SP is 3.1X over IDBA-UD. Of all the remaining readsets and configurations, ScalaDBG-PP achieves a maximum speedup of 3.3X for the SC-SAR324 readset in the $\{29 - 71\}$, step size 3 configuration. ScalaDBG- SP achieves a maximum speedup of 1.6X for RM1,RM2, SC-*S.aureus*, and SC-SAR324 readsets in the configurations processing 15 k values. For all the datasets and configurations, ScalaDBG is faster than IDBA-UD. Further, ScalaDBG-PP is faster than ScalaDBG-SP since it has higher parallelism during assembly for the patching process. Speedup of ScalaDBG-SP and ScalaDBG-PP over IDBA is higher for the larger readsets of RM1, RM2, SAR 324, and *S. aureus*.

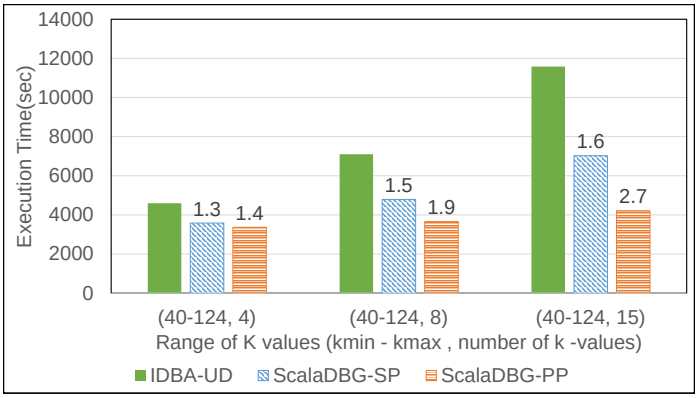

Fig. 4.8.: Time taken by IDBA-UD, ScalaDBG-SP, ScalaDBG-PP on RM1 data set. ScalaDBG runs on a cluster using the number of nodes equal to the number of *k*-values.

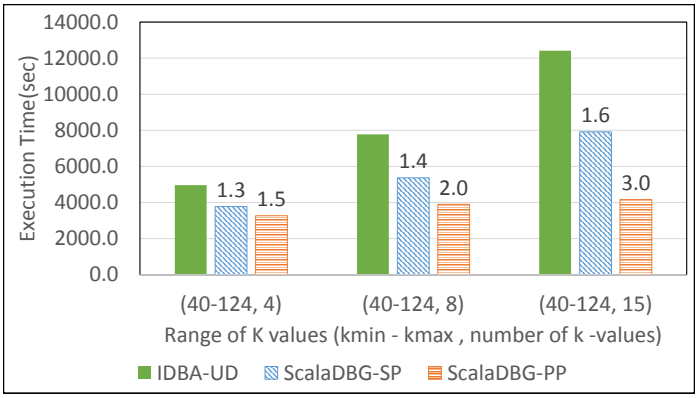

Fig. 4.9.: Time taken by IDBA, ScalaDBG-SP, ScalaDBG-PP on RM2 data set.

4.6.4 Accuracy

Tables 4.5 and 4.4 show the accuracy metrics for assembling the datasets in Table 4.2 for the above performance tests. For SAR324 we could not obtain its reference genome, so Table 4.4 denotes the coverage as NA. For the metagenomic datasets, we only reported number of contigs, N50 and max contig length, since the reference assemblies contain multiple genomes in 4.5. Differences in the accuracy arise as a result of the invocations of graph simplification and out of order patching. The results demonstrate that ScalaDBG and IDBA have comparable accuracy metrics in all cases. We performed the T-test and determined

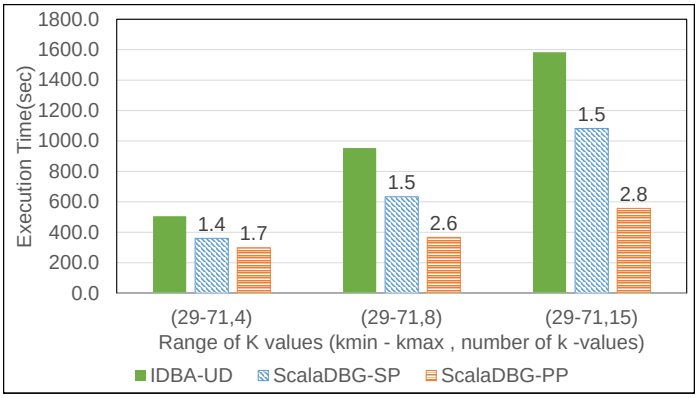

Fig. 4.10.: Time taken by IDBA, ScalaDBG-PP, ScalaDBG-PP for completing assembly on the SC-*E. coli* dataset. Speed up w.r.t IDBA-UD running on the same *k* value configuration is shown.

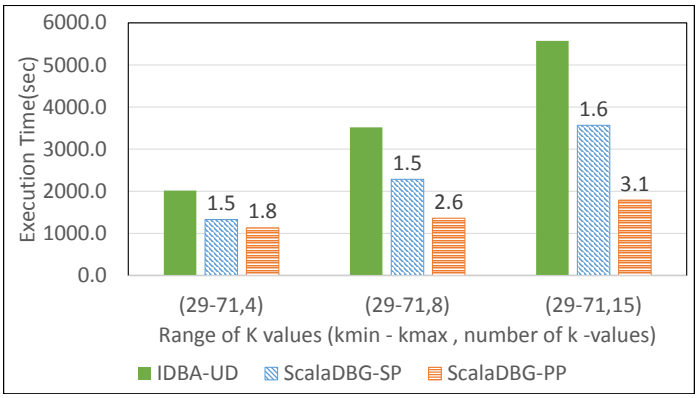

Fig. 4.11.: Time taken by IDBA, ScalaDBG-PP, ScalaDBG-PP for completing assembly on the SC-*S.aureus* dataset.

that the differences in the assembly metrics obtained for ScalaDBG-SP, ScalaDBG-PP and IDBA-UD are not statistically significant.

4.6.5 Time distribution for phases of ScalaDBG

ScalaDBG's running time can be decomposed into three major stages: (1) the de Bruijn graph construction, (2) graph patching, and (3) contig generation. We profiled ScalaDBG

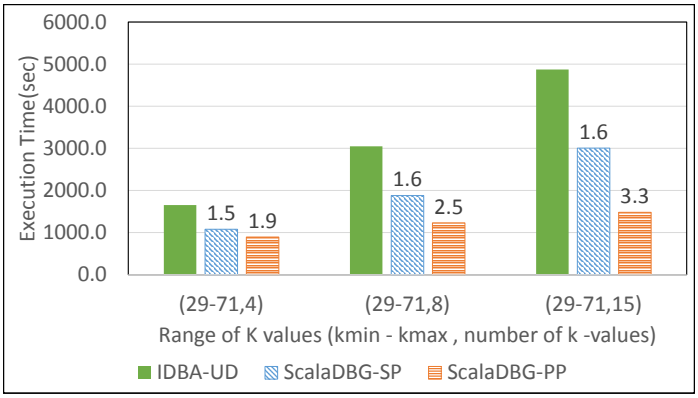

Fig. 4.12.: Time taken by IDBA, ScalaDBG-PP, ScalaDBG-PP for completing assembly on the SC-SAR324 dataset.

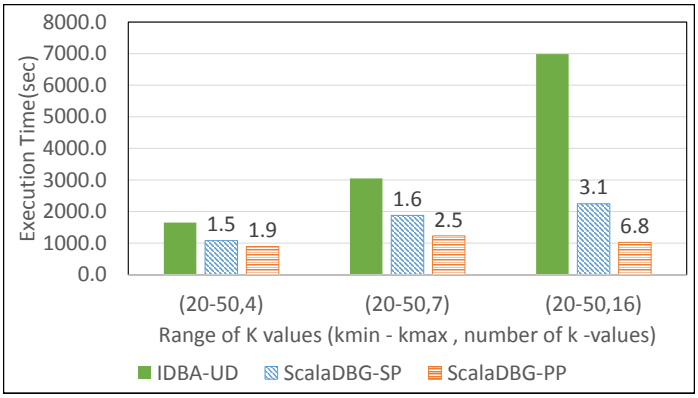

Fig. 4.13.: Time taken by IDBA, ScalaDBG-PP, ScalaDBG-PP for completing assembly on the SC-SAR324 dataset for range(20-50)

to determine how each job contributes to the total execution time of ScalaDBG. Because the behavior of ScalaDBG varies from specific configuration of k-values and dataset, we present results of assembling the SAR 324 dataset for k-value range of {20-50} with a step size of 2. The experiment was run on a 16 nodes.

ScalaDBG workflow starts by reading the input read set and terminates after generating the contigs from the read set. The timeline of events is shown in Figure 4.14 and.Figure 4.15 The different phases of the jobs are shown on the Y axis while the X axis shows the time-line. Note that in ScalaDBG-SP of Figure 4.14, only the construction processes execute in

Table 4.3.: Accuracy Comparison for Performance Tests on RM1 and RM2 datasets

Assembler	# Contigs	N50 (bp)	Max Contig	# Contigs	N50 (bp)	Max Contig Length
	RM1 k=40-124,4			**RM2 k=40-124,4**		
IDBA-UD	96291	8255	641885	123807	2251	572031
ScalaDBG-SP	94958	8104	641902	122427	2281	571953
ScalaDBG-PP	95519	7629	497722	123037	2249	444176
	RM1 k=40-124,8			**RM2 k=40-124,8**		
IDBA-UD	95633	10729	772713	121911	2457	563546
ScalaDBG-SP	96849	10183	772927	121955	2582	573903
ScalaDBG-PP	98018	7962	497730	121772	2408	444517
	RM1 k=40-124,15			**RM2 k=40-124,15**		
IDBA-UD	95640	11453	772928	121720	2504	563546
ScalaDBG-SP	99857	10182	641918	119814	2679	573903
ScalaDBG-PP	99951	7906	641988	121568	2472	444518

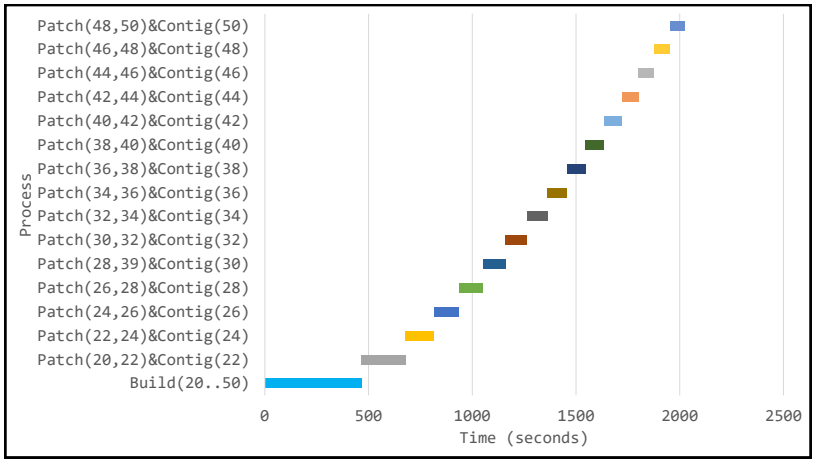

Fig. 4.14.: Timeline for processes of ScalaDBG-SP for SAR 324 dataset, k-value range{20-50}, step size 2

parallel, while in Figure 4.15 in ScalaDBG-PP the patch and contig generation processes execute in parallel.

The detailed break down of the time taken by ScalaDBG for each of the contig generation and patching steps is shown in the Figures. The patch process takes less time compared to the contig generation process for the dataset. For ScalaDBG- PP the contig generation and patch processes execute concurrently when the processes are independent.

Table 4.4.: Accuracy Comparison for Performance Tests on SC-*E. coli*, SC- *S. aureus* datasets

Assembler	# Contigs	N50 (bp)	Max Contig Length	Coverage	NGA50 (bp)	# missasemblies
	SC-*E.coli* k=40-124,4					
IDBA-UD	504	41996	133040	93.004	41009	4
ScalaDBG-SP	506	43834	133040	93.086	41309	4
ScalaDBG-PP	333	46016	140917	93.072	41996	3
	SC-*E.coli* k=40-124,8					
IDBA-UD	503	42834	140971	93.045	41309	4
ScalaDBG-SP	504	43834	133040	93.064	41996	3
ScalaDBG-PP	333	46016	140917	93.086	42289	4
	SC-*E.coli* k=40-124,15					
IDBA-UD	507	42289	133040	93.101	41009	6
ScalaDBG-SP	512	46016	140971	93.093	42289	6
ScalaDBG-PP	333	46016	140917	93.078	42289	4
	SC-*S.aureus* k=40-124,4					
IDBA-UD	400	24855	126604	98.121	26379	3
ScalaDBG-SP	377	24855	126604	98.189	26379	3
ScalaDBG-PP	370	24855	126604	98.201	26379	3
	SC-*S.aureus* k=40-124,8					
IDBA-UD	412	24855	126604	98.081	26379	3
ScalaDBG-SP	384	24855	126604	98.176	26379	3
ScalaDBG-PP	373	24855	126604	98.205	26379	3
	SC-*S.aureus* k=40-124,15					
IDBA-UD	413	24855	126604	98.068	26379	3
ScalaDBG-SP	393	24855	126604	98.167	26379	3
ScalaDBG-PP	374	24855	126604	98.205	26379	3

For ScalaDBG-SP all the processes of patching and contig generation are serialized. This detailed profile can be used to optimize ScalaDBG further.

The processes of ScalaDBG and IDBA-UD differ in the work they do. ScalaDBG-SP and ScalaDBG-PP do not update the input read set at each iteration. Instead, each build process in ScalaDBG starts with the original read set. IDBA-UD updates the read set at each iteration. However, if the reduction in the input read set is not significant at each iteration, then the overhead of updating the read set for IDBA-UD starts to dominate. Especially for lower k-values excessive branching can lead to less reduction in the read set, increasing

Table 4.5.: Accuracy Comparison for Performance Tests on SC-SAR 324 datasets

Assembler	# Contigs	N50 (bp)	Max Contig Length	Coverage	NGA50 (bp)	# missasemblies
	SC-SAR324 k=29-71,4					
IDBA-UD	733	61419	202281	NA	NA	NA
ScalaDBG-SP	709	64747	202281	NA	NA	NA
ScalaDBG-PP	705	62374	202281	NA	NA	NA
	SC-SAR324 k=29-71,8					
IDBA-UD	742	60700	202281	NA	NA	NA
ScalaDBG-SP	710	64747	202281	NA	NA	NA
ScalaDBG-PP	703	63904	202281	NA	NA	NA
	SC-SAR324 k=29-71,15					
IDBA-UD	747	60700	202281	NA	NA	NA
ScalaDBG-SP	723	64747	202281	NA	NA	NA
ScalaDBG-PP	712	64795	202281	NA	NA	NA
	SC-SAR324 k=20-50,4					
IDBA-UD	1082	32119	131087	NA	NA	NA
ScalaDBG-SP	1085	38257	131546	NA	NA	NA
ScalaDBG-PP	1080	38257	131546	NA	NA	NA
	SC-SAR324 k=20-50,7					
IDBA-UD	1088	33192	131087	NA	NA	NA
ScalaDBG-SP	1087	38257	131546	NA	NA	NA
ScalaDBG-PP	1078	38257	131546	NA	NA	NA
	SC-SAR324 k=20-50,16					
IDBA-UD	7740	22977	131087	NA	NA	NA
ScalaDBG-SP	8342	24254	131041	NA	NA	NA
ScalaDBG-PP	8118	24254	131041	NA	NA	NA

the overhead for IDBA-UD. In addition, the patch and contig generation processes execute only logarithmic number of times in ScalaDBG-PP as compared to IDBA-UD and ScalaDBG-SP.

If we serialize the execution time of the parallel processes in ScalaDBG-SP and ScalaDBG-PP shown in Figure 4.14 and Figure 4.15, the serial execution time for ScalaDBG-SP is 6455 seconds and ScalaDBG- PP is 6457 seconds. The execution time for IDBA-UD is 6897 seconds. Out of this total time 86% of the overall execution time can be parallelized. Hence the maximum speedup for ScalaDBG-PP is 6.8X. ScalaDBG-SP performs the patch and contig generation serially, hence its speedup drops to 3.1X. For this dataset and k-value

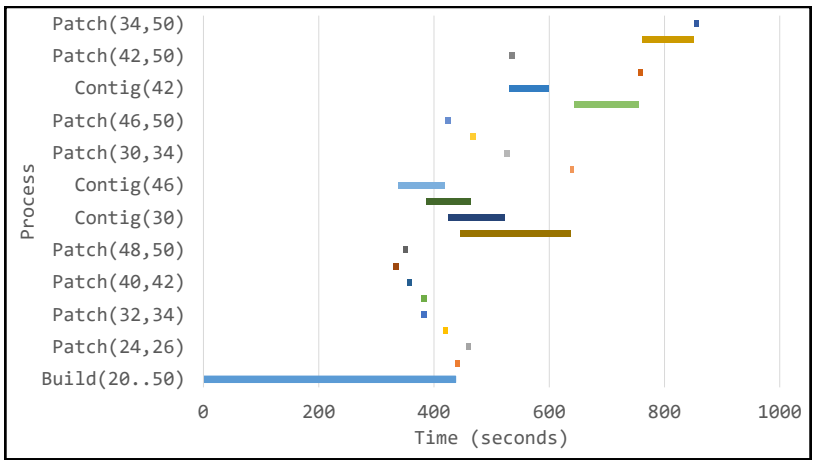

Fig. 4.15.: Timeline for processes of ScalaDBG-PP for SAR 324 dataset, k-value range{20-50}, step size 2

range configuration, the additional work done by ScalaDBG is offset by the work done by IDBA-UD in updation of the read set.

4.6.6 Comparison with distributed assembler Abyss

In this experiment we compared the performance and the assembly metrics for ScalaDBG and Abyss [72]. Abyss assembler uses MPI to parallelize the execution of de Bruijn graph construction on multiple nodes. It builds the de Bruijn graph using a single k-value. We ran ScalaDBG on 4 nodes using k-value range 20-50 with a step size of 10. We ran Abyss using a median value of k=35 on 4 nodes. Both ScalaDBG and Abyss use all cores on all the nodes. As shown in Table 4.6 ScalaDBG has better performance than Abyss for the chosen configuration. Since it uses multiple k-values, namely, 20,30,40, and 50 as opposed to Abyss which just uses k-value 35, ScalaDBG also has higher N50 and maximum contig length value as compared to Abyss.

Table 4.6.: Accuracy and Performance comparison on SC-*SAR 324* datasets for ScalaDBG-PP and Abyss

Assembler	Execution Time(sec)	N50 (bp)	Max Contig Length
Abyss	2240	37486	131365
ScalaDBG	892	38257	131546

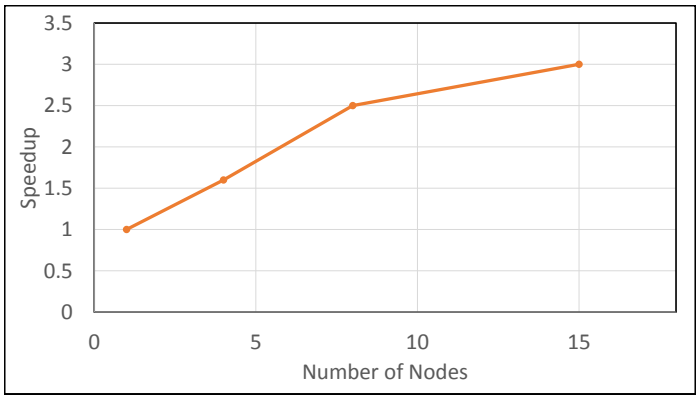

Fig. 4.16.: Scaling Results for ScalaDBG, speedup shown w.r.t ScalaDBG running on 1 node, RM2 dataset, k-value range {40-124} step size 6

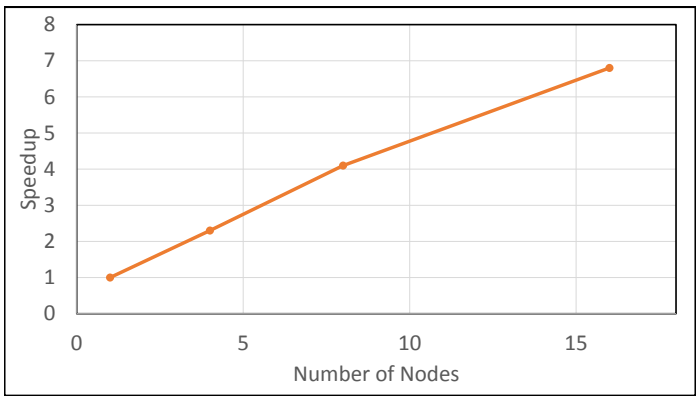

Fig. 4.17.: Scaling Results for ScalaDBG, speedup shown w.r.t ScalaDBG running on 1 node, SAR324 dataset, k-value range {20-50} step size 2

4.6.7 Scalability Tests

To evaluate the scaling out for ScalaDBG, we used RM2 and SAR324 data sets. Figure 4.16 and Figure 4.17 shows the *k*-values range and the number of nodes used in the cluster. As can be seen, ScalaDBG achieves a speedup of 6.8X for SAR324 and 3X for the RM2 dataset , compared to the baseline version running on 1 node. The reduction in efficiency for ScalaDBG-PP is due to load imbalances in the parallel reduction tree . The speedup demonstrates that ScalaDBG can scale out in a cluster and leverage the power of all its cores.

4.7 Related Work

Efficient de-novo assembly applications have been proposed to deal with the tremendous increase in genomic sequences [50, 54, 55, 71–75, 80–84]. These assembly applications are either limited to scaling up on a single node, or cannot use multiple *k* values during the process of assembly. To our knowledge, there has been no previous work on parallelizing de bruijn graph construction for multiple K-mers on multiple nodes in a cluster.

Ray [80], ABySS [72], PASHA [81], and HipMer [82] can parallelize the assembly process for a single *k* value on multiple nodes in a cluster. Flick *et al.* [85] propose a method to find weakly connected components of the de Bruijn graph that can be compressed in parallel. However, these approaches do not generate good quality assemblies for metagenomic and single cell datasets with uneven sequencing depths, unlike ScalaDBG since they work on a single k-value.

SGA [55], Velvet [71], SOAPdenovo [75], ALLPATHS-LG [50] can parallelize the assembly process on multiple cores of a single node. Metagenomic assemblers such as Meta-velvet [83] also do not use multiple k values in assembly. IDBA, IDBA-UD, and SPAdes can utilize multiple *k* values. They are limited to scale on a single node. ScalaDBG can scale up, scale out, and also process multiple k-values.

4.8 Conclusion

Faster and cheaper sequencing technologies have led to a massive increase in the amount of sequencing data. Efficient assembly algorithms are key to uncovering knowledge within the data and make possible medical breakthroughs based on single cell and metagenomic datasets. Existing iterative methods of debruijn graph construction such as IDBA-UD, generate longer contigs but are completely sequential and suffer from significantly longer graph construction times. In this paper we presented a technique ScalaDBG that breaks this serial process of graph construction into a parallel process. Our technique is general, and can be easily extended to other DBG-based assembly algorithms.

5. INDEXING DATA STRUCTURES

5.1 Introduction

The cost of sequencing DNA has plummeted since the introduction of next generation sequencing technologies, with recent advances bringing the cost of sequencing a single human at 30-fold coverage to around $1,000 [86]. This has led to several large sequencing projects such as the 1000 Genomes project [87], the Exome sequencing project (ESP) [88], and The Cancer Genome Atlas (TCGA), http://cancergenome.nih.gov/, [89]. These projects allow us to detect and characterize genomic variation by alignment of whole genomes with each other, and gain insights for evolutionary trends, disease identification and treatment. Hence we need to facilitate rapid comparisons of entire genomes. However, the generation of the massive quantities of genomic data at a higher throughput have created scalability bottlenecks for whole genome alignment tools.

Whole genome alignment or global alignment is the process of end-to-end comparison of two closely related species or two organisms of the same species and is a nontrivial computational task. Exact search is a crucial kernel of popular whole genome alignment tools such as MUMmer [43, 90], VMatch [91], backwardMEM [92], sparseMEM [49], slaMEM [47], essaMEM [48], and most recently E-MEM [44]. The whole genome alignment workflow begins by first an exact comparison of the two genome sequences being aligned to identify *anchors* or regions of exact matches that cannot be extened to either side without producing a mismatch (called as MEMs) between the two sequences. The matches are then clustered and non-exact region between the matches is done using dynamic programming. In this workflow, the time taken for exact matching dominates the overall execution time. For the comparison between fruitfly species *D. melanogaster* and *D. ananassae* MUMmer spends 80% time on exact matching, and for whole genome alignment between *D. williston* and *D.persimilis* exact matching takes 78% of the overall time.

Since MEM computation is a challenging problem, backwardMEM, sparseMEM, slaMEM, essaMEM, and E-MEM just perform exact match computation.

Exact search to compute the MEMs is performed by indexing one of the two genomic sequences, and then searching the other sequence by performing a lookup. The time taken by exact search is clearly driven by the design of the underlying index structure. The above tools use indexing data structures such as representations of suffix trees (MUMmer), suffix arrays(Vmatch, backwardMEM, sparseMEM, essaMEM), lookup or hash tables (LUT)(E-MEM), and Burrows Wheeler transform (BWT)-based FM-index (slaMEM) to index the genome sequence.

These tools come from an era when multicore processors were rare and memory and caches were much smaller. It made more sense formerly to focus on improving sequential performance and lowering the overall memory requirement of the tools. So tools such as essaMEM, sparseMEM, slaMEM, and E-MEM focus on reducing the memory footprint using sparse suffix array structures, hash tables, and the space economical FM-index.

This conventional wisdom of employing just space-efficient structures for superior performance, however, does not blend well with current multicore and manycore processors. In this changed scenario, it is more important to exploit opportunities for locality, parallelization, prefetching, and vectorization for performance gains. Recent work on performance analysis of FM-index-based search and hash table- based search has found that it has irregular memory access with poor locality between accesses [93, 94]. Hence, in the current form genome indexing structures are unsuitable for modern multicore processors and fares poorly on them.

Due to the changed hardware context, there is a need to revisit the data structures and algorithms used for exact search to identify the ones with better data locality. We observe that tree-based indices provide more opportunities to exploit data locality in search. In this paper, we adopt the prefix Directed Acyclic Word Graph (DAWG) for genomic indexing. DAWG is a tree-based index. The key reason for better suitability of the prefix DAWG for search is that at any node, there are at most four possible outgoing edges A/C/G/T. Therefore, we can arrange the adjacent nodes of a node to be spatially closer. During

search, the nodes to be traversed next are restricted to the subgraph of the current node. This property results in relatively better spatial locality compared to the FM-index and hash table or lookup table, since the next node is often reachable within a few hops. Furthermore, we found a property of a genome indexed as a DAWG wherein, the branching is higher at upper levels than at lower levels. Hence, reads having same initial paths are highly likely to have similar paths at later levels. Thus, if reads traversing the same initial path are grouped and processed consecutively, there will be high data reuse. These reasons lead to greater opportunities to exploit locality in search for prefix DAWG. We also optimize the FM-index-based search to exploit data locality.

We survey state-of-the-art global alignment tools and find that exact matching through index search is the top time-consuming function in whole genome alignment applications. We implement optimized versions of exact matching using index search in prefix DAWG and FM-index. We exploit techniques such as multithreading, software prefetching, and tiling. For alignment of large genomic sequences E-MEM is both space and time efficient in comparison to all the other tools. Hence, we compare the performance of our optimized structures for MEM computation with E-MEM.

5.2 Optimizing the indexing data structures

We develop optimized exact match implementation based on index search for the prefix DAWG and FM-index for MEM computation. The prefix DAWG structure lends itself naturally to exploit data locality optimizations. Memory accesses for the FM-index are optimized through careful study of their control flow.

For both FM-index and prefix DAWG we recognize and employ strategies such as software prefetching and overlapping computation with data load to reduce number of cache misses and improve locality. We also bin queries that may access similar regions in the structures and execute them consecutively. We also utilize parallelizing strategies to leverage all the cores in modern multicore processors. We apply our strategies to develop optimized prefix DAWG and FM-index structures with better cache locality and performance.

5.3 Evaluation

In this section we demonstrate the performance gains of using an optimized implementation of FM-index and prefix DAWG for MEM computation over the E-MEM implementation.

We used one server node of Intel(R) Xeon(R) CPU E5-4650 2.70GHz having 32 cores. It has 198GB of main memory, and L2 and L3 caches of 256KB and 20MB respectively. We used the fruitfly species of Drosophila melanogaster and Drosophila ananassae to perform whole genome alignment. *D. ananassae* was used as reference sequence and its index was computed. *D.melanogaster* was used as the query sequence. We include the time taken to compute MEMs of length 50 using E-MEM, and optimized FM-index and prefix DAWG data structure.

Table 5.1.: Execution Time taken for MEM computation

Data Structure	**Time Taken(sec)**
E-MEM	17.9
FM-Index	1.2
Prefix DAWG	0.9

As seen in the table, the optimized prefix DAWG implementation is 19.9X faster than E-MEM's implementation for MEM computation. The optimized FM-index implementation is 14.9X faster than E-MEM's implementation for MEM computation. The optimized prefix DAWG implementation is 1.3X faster than the optimized FM-index implementation. Hence developing and employing cache efficient index structures in whole genome alignment applications can significantly enhance their performance.

5.4 Conclusion

Efficient whole genome alignment tools are required to match the throughput of next generation sequences, and urgently process the data to identify genetic variations responsible for human diseases. Exact search in MEM computation is a crucial module in all global

alignment tools. It is enabled by indexing structures such as FM-index, prefix DAWG, and hash or lookup table. Through our study we find that employing cache efficient index structures in whole genome alignment applications can significantly enhance their performance. Such optimized structures are highly relevant for leveraging today's modern processors for performance and scalability.

6. CONCLUSION

The explosive growth in genomic datasets can enable interesting applications such as personalized medicine,where medical procedures can be tailored to individual patients based on their proclivity to diseases based on their genome. However, to enable researchers to gain these insights, we need scalable genomics applications and tools that can analyze these massive genomic datasets.

We have developed a holistic approach that provides techniques at three levels - algorithm, compiler, and data structure - to develop efficient and scalable genomic applications. At the algorithm level, we have developed a fine-grained parallelism technique, called Orion which divides the input genomic sequence into an adaptive number of fragments with optimal overlap. This technique achieves higher speedup, parallelism and load balancing than current state-of-the-art tools. We have also analyzed the iterative de Bruijn graph based genome assembly applications, and parallelized the most compute intensive phase of iterative de Bruijn graph construction in ScalaDBG. At the compiler level, we have developed a domain-specific language, called SARVAVID, that makes developing computational genomics applications easier for the genomics researcher. SARVAVID framework provides commonly recurring software modules in computational genomics applications as language constructs. The availability of efficient implementations of such constructs improves programmer productivity, and provide effective scalability with growing data. The DSL compiler performs domain-specific optimizations, which are beyond the scope of libraries and generic compilers. In addition, SARVAVID supports exploitation of parallelism across multiple nodes At the data structure level, we have recognized opportunities to develop indexing data structures with better data locality and optimize them to leverage modern hardware.

To bring applications such as developing targeted drug therapies in practice, continuous exploration of more novel techniques to speed up genomic applications and tools is needed.

We believe our techniques are crucial in development of the required scalable genomic genomic tools.

REFERENCES

REFERENCES

[1] S. D. Kahn *et al.*, "On the future of genomic data," *Science(Washington)*, vol. 331, no. 6018, pp. 728–729, 2011.

[2] A. McKenna, M. Hanna, E. Banks, A. Sivachenko, K. Cibulskis, A. Kernytsky, K. Garimella, D. Altshuler, S. Gabriel, M. Daly *et al.*, "The genome analysis toolkit: a mapreduce framework for analyzing next-generation dna sequencing data," *Genome research*, vol. 20, no. 9, pp. 1297–1303, 2010.

[3] B. Langmead, K. D. Hansen, J. T. Leek *et al.*, "Cloud-scale rna-sequencing differential expression analysis with myrna," *Genome Biol*, vol. 11, no. 8, p. R83, 2010.

[4] K. Mahadik, S. Chaterji, B. Zhou, M. Kulkarni, and S. Bagchi, "Orion: Scaling genomic sequence matching with fine-grained parallelization," in *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*. IEEE, 2014, pp. 449–460.

[5] K. Mahadik, C. Wright, J. Zhang, M. Kulkarni, S. Bagchi, and S. Chaterji, "Sarvavid: A domain specific language for developing scalable computational genomics applications," in *Proceedings of the 2016 International Conference on Supercomputing*. ACM, 2016, p. 34.

[6] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of molecular biology*, vol. 215, no. 3, pp. 403–410, 1990.

[7] S. F. Altschul, T. L. Madden, A. A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman, "Gapped blast and psi-blast: a new generation of protein database search programs," *Nucleic acids research*, vol. 25, no. 17, pp. 3389–3402, 1997.

[8] D. A. Benson, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, and E. W. Sayers, "Genbank," *Nucleic acids research*, vol. 38, no. suppl 1, pp. D46–D51, 2010.

[9] I. Mizrachi, "Genbank," *National Center for Biotechnology Information (US)*, 2013.

[10] R. Braun, K. T. Pedretti, T. L. Casavant, T. E. Scheetz, C. L. Birkett, and C. A. Roberts, "Parallelization of local blast service on workstation clusters," *Future Generation Computer Systems*, vol. 17, no. 6, pp. 745–754, 2001.

[11] N. Camp, H. Cofer, and R. Gomperts, "High throughput BLAST," Silicon Graphics, Inc., Tech. Rep., 1998.

[12] P. Wit, M. H. Pespeni, J. T. Ladner, D. J. Barshis, F. Seneca, H. Jaris, N. O. Therkildsen, M. Morikawa, and S. R. Palumbi, "The simple fool's guide to population genomics via rna-seq: an introduction to high-throughput sequencing data analysis," *Molecular ecology resources*, vol. 12, no. 6, pp. 1058–1067, 2012.

[13] A. Darling, L. Carey, and W.-c. Feng, "The design, implementation, and evaluation of mpiblast," *Proceedings of ClusterWorld*, vol. 2003, 2003.

[14] S. Schwartz, W. J. Kent, A. Smit, Z. Zhang, R. Baertsch, R. C. Hardison, D. Haussler, and W. Miller, "Human–mouse alignments with blastz," *Genome research*, vol. 13, no. 1, pp. 103–107, 2003.

[15] M. K. Gardner, W.-c. Feng, J. Archuleta, H. Lin, and X. Ma, "Parallel genomic sequence-searching on an ad-hoc grid: experiences, lessons learned, and implications," in *SC 2006 Conference, Proceedings of the ACM/IEEE*. IEEE, 2006, pp. 22–22.

[16] C. Camacho, G. Coulouris, V. Avagyan, N. Ma, J. Papadopoulos, K. Bealer, and T. Madden, "Blast+: architecture and applications," *BMC bioinformatics*, vol. 10, no. 1, p. 421, 2009.

[17] C. Francis, "Fragblast," http://www.clarkfrancis.com/blast/fragblast.html.

[18] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of molecular biology*, vol. 147, no. 1, pp. 195–197, 1981.

[19] X.-l. Yang, Y.-l. Liu, C.-f. Yuan, and Y.-h. Huang, "Parallelization of blast with mapreduce for long sequence alignment," in *Parallel Architectures, Algorithms and Programming (PAAP), 2011 Fourth International Symposium on*. IEEE, 2011, pp. 241–246.

[20] S. Karlin and S. F. Altschul, "Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes," *Proceedings of the National Academy of Sciences*, vol. 87, no. 6, pp. 2264–2268, 1990.

[21] E. Michael Gertz, "BLAST Scoring Parameters," ftp://ftp.cbi.edu.cn/pub/software/blast/documents/developer/scoring.pdf.

[22] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[23] E. Bier, "Drosophila, the golden bug, emerges as a tool for human genetics," *Nature Reviews Genetics*, vol. 6, no. 1, pp. 9–23, 2005.

[24] I. V. Makunin, V. V. Shloma, S. J. Stephen, M. Pheasant, and S. N. Belyakin, "Comparison of ultra-conserved elements in drosophilids and vertebrates," *PloS one*, vol. 8, no. 12, p. e82362, 2013.

[25] T. Ryu, L. Seridi, and T. Ravasi, "The evolution of ultraconserved elements with different phylogenetic origins," *BMC evolutionary biology*, vol. 12, no. 1, p. 236, 2012.

[26] R. McLendon, A. Friedman, D. Bigner, E. G. Van Meir, D. J. Brat, G. M. Mastrogianakis, J. J. Olson, T. Mikkelsen, N. Lehman, K. Aldape *et al.*, "Comprehensive genomic characterization defines human glioblastoma genes and core pathways," *Nature*, vol. 455, no. 7216, pp. 1061–1068, 2008.

[27] M. C. Schatz, "Cloudburst: highly sensitive read mapping with mapreduce," *Bioinformatics*, vol. 25, no. 11, pp. 1363–1369, 2009. [Online]. Available: http://bioinformatics.oxfordjournals.org/content/25/11/1363.abstract

[28] B. Langmead, K. Hansen, and J. Leek, "Cloud-scale rna-sequencing differential expression analysis with myrna," *Genome Biology*, vol. 11, no. 8, p. R83, 2010. [Online]. Available: http://genomebiology.com/content/11/8/R83

[29] H. Nordberg, K. Bhatia, K. Wang, and Z. Wang, "Biopig: a hadoop-based analytic toolkit for large-scale sequence data," *Bioinformatics*, vol. 29, no. 23, pp. 3014–3019, 2013. [Online]. Available: http://bioinformatics.oxfordjournals.org/content/29/23/3014.abstract

[30] A. McKenna, M. Hanna, E. Banks, A. Sivachenko, K. Cibulskis, A. Kernytsky, K. Garimella, D. Altshuler, S. Gabriel, M. Daly, and M. A. DePristo, "The genome analysis toolkit: A mapreduce framework for analyzing next-generation dna sequencing data," *Genome Research*, vol. 20, no. 9, pp. 1297–1303, 2010. [Online]. Available: http://genome.cshlp.org/content/20/9/1297.abstract

[31] T. Nguyen, W. Shi, and D. Ruden, "Cloudaligner: A fast and full-featured mapreduce based tool for sequence mapping," *BMC Research Notes*, vol. 4, no. 1, p. 171, 2011. [Online]. Available: http://www.biomedcentral.com/1756-0500/4/171

[32] A. Matsunaga, M. Tsugawa, and J. Fortes, "Cloudblast: Combining mapreduce and virtualization on distributed resources for bioinformatics applications," in *eScience, 2008. eScience'08. IEEE Fourth International Conference on*. IEEE, 2008, pp. 222–229.

[33] L. Pireddu, S. Leo, and G. Zanetti, "Mapreducing a genomic sequencing workflow," in *Proceedings of the Second International Workshop on MapReduce and Its Applications*, ser. MapReduce '11. New York, NY, USA: ACM, 2011, pp. 67–74. [Online]. Available: http://doi.acm.org/10.1145/1996092.1996106

[34] P. D. Vouzis and N. V. Sahinidis, "Gpu-blast: Using graphics processors to accelerate protein sequence alignment," *Bioinformatics*, 2010. [Online]. Available: http://bioinformatics.oxfordjournals.org/content/early/2010/11/17/bioinformatics.btq644.abstract

[35] A. Smith, Z. Xuan, and M. Zhang, "Using quality scores and longer reads improves accuracy of solexa read mapping," *BMC Bioinformatics*, vol. 9, no. 1, p. 128, 2008. [Online]. Available: http://www.biomedcentral.com/1471-2105/9/128

[36] L. J. Young, "Genomic data growing faster than twitter and youtube," http://spectrum.ieee.org/tech-talk/biomedical/diagnostics/the-human-os-is-at-the-top-of-big-data, July 2015.

[37] Z. D. Stephens, S. Y. Lee, F. Faghri, R. H. Campbell, C. Zhai, M. J. Efron, R. Iyer, M. C. Schatz, S. Sinha, and G. E. Robinson, "Big data: Astronomical or genomical?" *PLoS Biol*, vol. 13, no. 7, p. e1002195, 2015.

[38] M. C. Schatz, "Cloudburst: highly sensitive read mapping with mapreduce," *Bioinformatics*, vol. 25, no. 11, pp. 1363–1369, 2009.

[39] H. Nordberg, K. Bhatia, K. Wang, and Z. Wang, "Biopig: a hadoop-based analytic toolkit for large-scale sequence data," *Bioinformatics*, p. btt528, 2013.

[40] T. Nguyen, W. Shi, and D. Ruden, "Cloudaligner: A fast and full-featured mapreduce based tool for sequence mapping," *BMC research notes*, vol. 4, no. 1, p. 171, 2011.

[41] M. B. Scholz, C.-C. Lo, and P. S. Chain, "Next generation sequencing and bioinformatic bottlenecks: the current state of metagenomic data analysis," *Current opinion in biotechnology*, vol. 23, no. 1, pp. 9–15, 2012.

[42] T. Craddock, C. R. Harwood, J. Hallinan, and A. Wipat, "e-science: relieving bottlenecks in large-scale genome analyses," *Nature reviews microbiology*, vol. 6, no. 12, pp. 948–954, 2008.

[43] S. Kurtz, A. Phillippy, A. L. Delcher, M. Smoot, M. Shumway, C. Antonescu, and S. L. Salzberg, "Versatile and open software for comparing large genomes," *Genome biology*, vol. 5, no. 2, p. R12, 2004.

[44] N. Khiste and L. Ilie, "E-mem: efficient computation of maximal exact matches for very large genomes," *Bioinformatics*, vol. 31, no. 4, pp. 509–514, 2015.

[45] W. J. Kent, "Blatthe blast-like alignment tool," *Genome research*, vol. 12, no. 4, pp. 656–664, 2002.

[46] T. W. Lam, W.-K. Sung, S.-L. Tam, C.-K. Wong, and S.-M. Yiu, "Compressed indexing and local alignment of dna," *Bioinformatics*, vol. 24, no. 6, pp. 791–797, 2008.

[47] F. Fernandes and A. T. Freitas, "slamem: efficient retrieval of maximal exact matches using a sampled lcp array," *Bioinformatics*, vol. 30, no. 4, pp. 464–471, 2014.

[48] M. Vyverman, B. De Baets, V. Fack, and P. Dawyndt, "essamem: finding maximal exact matches using enhanced sparse suffix arrays," *Bioinformatics*, vol. 29, no. 6, pp. 802–804, 2013.

[49] Z. Khan, J. S. Bloom, L. Kruglyak, and M. Singh, "A practical algorithm for finding maximal exact matches in large sequence datasets using sparse suffix arrays," *Bioinformatics*, vol. 25, no. 13, pp. 1609–1616, 2009.

[50] S. Gnerre, I. MacCallum, D. Przybylski, F. J. Ribeiro, J. N. Burton, B. J. Walker, T. Sharpe, G. Hall, T. P. Shea, S. Sykes *et al.*, "High-quality draft assemblies of mammalian genomes from massively parallel sequence data," *Proceedings of the National Academy of Sciences*, vol. 108, no. 4, pp. 1513–1518, 2011.

[51] B. Langmead and S. L. Salzberg, "Fast gapped-read alignment with bowtie 2," *Nature methods*, vol. 9, no. 4, pp. 357–359, 2012.

[52] S. Batzoglou, D. B. Jaffe, K. Stanley, J. Butler, S. Gnerre, E. Mauceli, B. Berger, J. P. Mesirov, and E. S. Lander, "Arachne: a whole-genome shotgun assembler," *Genome research*, vol. 12, no. 1, pp. 177–189, 2002.

[53] A. Döring, D. Weese, T. Rausch, and K. Reinert, "Seqan an efficient, generic c++ library for sequence analysis," *BMC bioinformatics*, vol. 9, no. 1, p. 11, 2008.

[54] A. Bankevich, S. Nurk, D. Antipov, A. A. Gurevich, M. Dvorkin, A. S. Kulikov, V. M. Lesin, S. I. Nikolenko, S. Pham, A. D. Prjibelski *et al.*, "Spades: a new genome assembly algorithm and its applications to single-cell sequencing," *Journal of Computational Biology*, vol. 19, no. 5, pp. 455–477, 2012.

[55] J. T. Simpson and R. Durbin, "Efficient de novo assembly of large genomes using compressed data structures," *Genome research*, vol. 22, no. 3, pp. 549–556, 2012.

[56] S. Burkhardt, A. Crauser, P. Ferragina, H.-P. Lenhof, E. Rivals, and M. Vingron, "q-gram based database searching using a suffix array (quasar)," in *Proceedings of the third annual international conference on Computational molecular biology*. ACM, 1999, pp. 77–83.

[57] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of molecular biology*, vol. 48, no. 3, pp. 443–453, 1970.

[58] J. Bolker, "Model organisms: There's more to life than rats and flies," *Nature*, vol. 491, no. 7422, pp. 31–33, 2012.

[59] L. Guarente and C. Kenyon, "Genetic pathways that regulate ageing in model organisms," *Nature*, vol. 408, no. 6809, pp. 255–262, 2000.

[60] "Demo mummer," http://docs.seqan.de/seqan/2.0.0/page_DemoMummy.html.

[61] D. Vakatov, K. Siyan, and J. Ostell, "The ncbi c++ toolkit [internet]," *National Library of Medicine (US), National Center for Biotechnology Information, Bethesda (MD)*, 2003.

[62] J. Dutheil, S. Gaillard, E. Bazin, S. Glémin, V. Ranwez, N. Galtier, and K. Belkhir, "Bio++: a set of c++ libraries for sequence analysis, phylogenetics, molecular evolution and population genetics," *BMC bioinformatics*, vol. 7, no. 1, p. 188, 2006.

[63] C. Huttenhower, M. Schroeder, M. D. Chikina, and O. G. Troyanskaya, "The sleipnir library for computational functional genomics," *Bioinformatics*, vol. 24, no. 13, pp. 1559–1561, 2008.

[64] D. Butt, A. J. Roger, and C. Blouin, "libcov: A c++ bioinformatic library to manipulate protein structures, sequence alignments and phylogeny," *BMC bioinformatics*, vol. 6, no. 1, p. 138, 2005.

[65] A. Drummond and K. Strimmer, "Pal: an object-oriented programming library for molecular evolution and phylogenetics," *Bioinformatics*, vol. 17, no. 7, pp. 662–663, 2001.

[66] J. E. Stajich, D. Block, K. Boulez, S. E. Brenner, S. A. Chervitz, C. Dagdigian, G. Fuellen, J. G. Gilbert, I. Korf, H. Lapp *et al.*, "The bioperl toolkit: Perl modules for the life sciences," *Genome research*, vol. 12, no. 10, pp. 1611–1618, 2002.

[67] P. Rice, I. Longden, A. Bleasby *et al.*, "Emboss: the european molecular biology open software suite," *Trends in genetics*, vol. 16, no. 6, pp. 276–277, 2000.

[68] W. Pitt, M. A. Williams, M. Steven, B. Sweeney, A. J. Bleasby, and D. S. Moss, "The bioinformatics template librarygeneric components for biocomputing," *Bioinformatics*, vol. 17, no. 8, pp. 729–737, 2001.

[69] G. Gremme, S. Steinbiss, and S. Kurtz, "Genometools: a comprehensive software library for efficient processing of structured genome annotations," *Computational Biology and Bioinformatics, IEEE/ACM Transactions on*, vol. 10, no. 3, pp. 645–656, 2013.

[70] P. E. Compeau, P. A. Pevzner, and G. Tesler, "How to apply de bruijn graphs to genome assembly," *Nature biotechnology*, vol. 29, no. 11, pp. 987–991, 2011.

[71] D. R. Zerbino and E. Birney, "Velvet: algorithms for de novo short read assembly using de bruijn graphs," *Genome research*, vol. 18, no. 5, pp. 821–829, 2008.

[72] J. T. Simpson, K. Wong, S. D. Jackman, J. E. Schein, S. J. Jones, and I. Birol, "Abyss: a parallel assembler for short read sequence data," *Genome research*, vol. 19, no. 6, pp. 1117–1123, 2009.

[73] Y. Peng, H. C. Leung, S.-M. Yiu, and F. Y. Chin, "Idba–a practical iterative de bruijn graph de novo assembler," in *Research in Computational Molecular Biology*. Springer, 2010, pp. 426–440.

[74] ——, "Idba-ud: a de novo assembler for single-cell and metagenomic sequencing data with highly uneven depth," *Bioinformatics*, vol. 28, no. 11, pp. 1420–1428, 2012.

[75] R. Luo, B. Liu, Y. Xie, Z. Li, W. Huang, J. Yuan, G. He, Y. Chen, Q. Pan, Y. Liu *et al.*, "Soapdenovo2: an empirically improved memory-efficient short-read de novo assembler," *Gigascience*, vol. 1, no. 1, p. 18, 2012.

[76] M. M. Abbas, Q. M. Malluhi, and P. Balakrishnan, "Assessment of de novo assemblers for draft genomes: a case study with fungal genomes," *BMC genomics*, vol. 15, no. 9, p. S10, 2014.

[77] U. of California at San Diego, "Single cell data sets," http://bix.ucsd.edu/projects/singlecell/nbt_data.html.

[78] A. Sczyrba, P. Hofmann, P. Belmann, D. Koslicki, S. Janssen, J. Droege, I. Gregor, S. Majda, J. Fiedler, E. Dahms *et al.*, "Critical assessment of metagenome interpretation- a benchmark of computational metagenomics software," *bioRxiv*, p. 099127, 2017.

[79] T. Thomas, J. Gilbert, and F. Meyer, "Metagenomics-a guide from sampling to data analysis," *Microbial informatics and experimentation*, vol. 2, no. 1, p. 3, 2012.

[80] S. Boisvert, F. Laviolette, and J. Corbeil, "Ray: simultaneous assembly of reads from a mix of high-throughput sequencing technologies," *Journal of computational biology*, vol. 17, no. 11, pp. 1519–1533, 2010.

[81] Y. Liu, B. Schmidt, and D. L. Maskell, "Parallelized short read assembly of large genomes using de bruijn graphs," *BMC bioinformatics*, vol. 12, no. 1, p. 354, 2011.

[82] E. Georganas, A. Buluç, J. Chapman, S. Hofmeyr, C. Aluru, R. Egan, L. Oliker, D. Rokhsar, and K. Yelick, "Hipmer: an extreme-scale de novo genome assembler," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2015, p. 14.

[83] T. Namiki, T. Hachiya, H. Tanaka, and Y. Sakakibara, "Metavelvet: an extension of velvet assembler to de novo metagenome assembly from short sequence reads," *Nucleic acids research*, vol. 40, no. 20, pp. e155–e155, 2012.

[84] Y. Peng, H. C. Leung, S.-M. Yiu, and F. Y. Chin, "Meta-idba: a de novo assembler for metagenomic data," *Bioinformatics*, vol. 27, no. 13, pp. i94–i101, 2011.

[85] P. Flick, C. Jain, T. Pan, and S. Aluru, "A parallel connectivity algorithm for de bruijn graphs in metagenomic applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2015, p. 15.

[86] E. C. Hayden, "The $1,000 genome," *Nature*, vol. 507, no. 7492, p. 294, 2014.

[87] . G. P. Consortium *et al.*, "An integrated map of genetic variation from 1,092 human genomes," *Nature*, vol. 491, no. 7422, pp. 56–65, 2012.

[88] W. Fu, T. D. OConnor, G. Jun, H. M. Kang, G. Abecasis, S. M. Leal, S. Gabriel, M. J. Rieder, D. Altshuler, J. Shendure *et al.*, "Analysis of 6,515 exomes reveals the recent origin of most human protein-coding variants," *Nature*, vol. 493, no. 7431, pp. 216–220, 2013.

[89] J. N. Weinstein, E. A. Collisson, G. B. Mills, K. R. M. Shaw, B. A. Ozenberger, K. Ellrott, I. Shmulevich, C. Sander, J. M. Stuart, C. G. A. R. Network *et al.*, "The cancer genome atlas pan-cancer analysis project," *Nature genetics*, vol. 45, no. 10, pp. 1113–1120, 2013.

[90] A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White, and S. L. Salzberg, "Alignment of whole genomes," *Nucleic acids research*, vol. 27, no. 11, pp. 2369–2376, 1999.

[91] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch, "Replacing suffix trees with enhanced suffix arrays," *Journal of Discrete Algorithms*, vol. 2, no. 1, pp. 53–86, 2004.

[92] M. I. Abouelhoda and E. Ohlebusch, "Chaining algorithms for multiple genome comparison," *Journal of Discrete Algorithms*, vol. 3, no. 2, pp. 321–341, 2005.

[93] J. Zhang, H. Lin, P. Balaji, and W.-c. Feng, "Optimizing burrows-wheeler transform-based sequence alignment on multicore architectures," in *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*. IEEE, 2013, pp. 377–384.

[94] W. Wang, W. Tang, L. Li, G. Tan, P. Zhang, and N. Sun, "Investigating memory optimization of hash-index for next generation sequencing on multi-core architecture," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*. IEEE, 2012, pp. 665–674.

VITA

VITA

Kanak Mahadik is a PhD student in Electrical and Computer Engineering, Purdue University, West Lafayette. She is advised by Prof. Milind Kulkarni and Prof. Saurabh Bagchi. Kanak received her B.E. in Computer Engineering from Pune University, India in 2010 and M.S. in Computer and Information Technology from Purdue University, West Lafayette in 2012. After working on performance optimizations at Salesforce.com, Kanak resumed graduate school in 2013 as a Ross Graduate Fellow. Her current research is on developing computational genomics applications and making them run on very large datasets, at very large scales. She is passionate about enabling personalized genomics through computational achievements and making a difference to the human condition, closely collaborating with biologists and clinicians.