SARVAVID: A Domain Specific Language for Developing Scalable Computational Genomics Applications

Kanak Mahadik[1], Christopher Wright[1], Jinyi Zhang[1], Milind Kulkarni[1], Saurabh Bagchi[1], Somali Chaterji[2]
[1]School of Electrical and Computer Engineering
[2]Department of Computer Science
Purdue University
West Lafayette, IN, USA
{kmahadik, christopherwright, zhan1182, milind, sbagchi, schaterji}@purdue.edu

ABSTRACT

Breakthroughs in gene sequencing technologies have led to an exponential increase in the amount of genomic data. Efficient tools to rapidly process such large quantities of data are critical in the study of gene functions, diseases, evolution, and population variation. These tools are designed in an ad-hoc manner, and require extensive programmer effort to develop and optimize them. Often, such tools are written with the currently available data sizes in mind, and soon start to under perform due to the exponential growth in data. Furthermore, to obtain high-performance, these tools require parallel implementations, adding to the development complexity.

This paper makes an observation that most such 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, the paper presents 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.

To demonstrate the efficacy of SARVAVID, we implement five well-known genomics applications—BLAST, MUMmer, E-MEM, SPAdes, and SGA—using SARVAVID. Our versions of BLAST, MUMmer, and E-MEM show a speedup of 2.4X, 2.5X, and 2.1X respectively compared to hand-optimized implementations when run on a single node, while SPAdes and SGA show the same performance as hand-written code. Moreover, SARVAVID applications scale to 1024 cores using a Hadoop backend.

ICS '16, June 01-03, 2016, Istanbul, Turkey

ISBN 978-1-4503-4361-9/16/05…$15.00

DOI: http://dx.doi.org/10.1145/2925426.2926283

Categories and Subject Descriptors

•Software and its engineering → Domain specific languages; •Computing methodologies → Distributed algorithms;

Keywords

Distributed Systems; Computational Genomics

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 [42, 39]. 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 [26]). For calibration, the smallest human chromosome (chromosome 22) is 49 Mbp long. Existing genomic applications are predominantly written in a monolithic 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

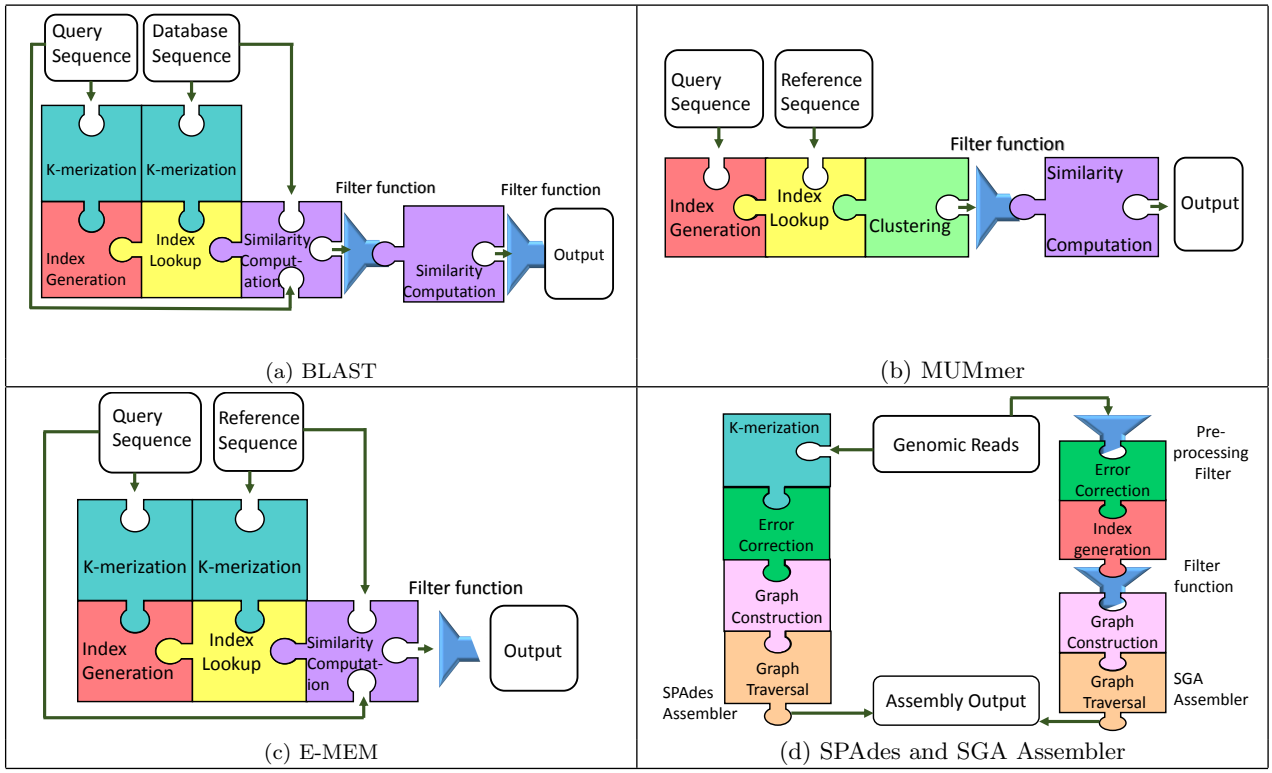

Figure 1: Overview of Genomic Applications in the categories of Local Alignment (BLAST), Whole genome Alignment (MUMmer and E-MEM), and Sequence Assembly (SPAdes and SGA). Common kernels are shaded using the same color.

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 distributed 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 [34, 31, 30, 28, 24] 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 [35, 8]. 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 1 shows that BLAST [2], MUMmer [22], and E-MEM [21] applications conceptually have a common kernel, *similarity computation* to determine similarity among genomic sequences, albeit with different constraints. Apart from these applications, other applications such as BLAT [19], BWT-SW [23], slaMEM [14], essaMEM [41], sparseMEM [20], ALLPaths-LG [15], Bowtie [25] and Arachne [4] 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

[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.

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 [11] 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 [3] and SGA [36] 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.

This paper's contributions are as follows:

- 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 2 describes the different areas of genomic applications. Section 3 details the design of the language and kernels. Section 4 discusses the implementation and various optimizations performed by the compiler. Section 5 details case studies of applications developed using SARVAVID DSL. Section 6 surveys related work, while Section 7 concludes the paper.

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 2 provides a high-level overview of these sub-domains.

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 [37], 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 [19, 23, 6]. 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 [9, 34]. The other approach executes each query in a query-set in parallel against replicated copies of the database [27].

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 [29] 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 [14, 41, 20]. MUMmer uses a suffix tree and E-MEM employs hash table–based indexing, memory optimizations, and multithreading to handle complex genomes.

2.3 Sequence Assembly

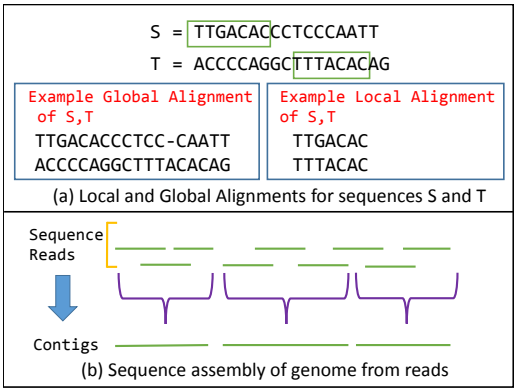

Figure 2: Overview of local alignment, global alignment (a), and sequence assembly(b)

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. 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 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.

3.1 Kernels

Table 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 3.2, 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 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

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

Table 1: Kernels in various genomic applications

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 required 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 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.

3.2 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 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 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 [21] 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 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 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 1(d) namely, **error correction**, **index generation**, **graph construction**, and **graph traversal**.

3.3 Grammar

Figure 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 predefined kernels. Notably, named functions can be passed to

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. $k \in Kernels ::= \texttt{kmerize}(\ldots) \mid \texttt{index\_generation}(\ldots) \mid \ldots$

Figure 3: (Simplified) SARVAVID grammar

kernels, providing higher-order features (*e.g.*, passing a filtering function to a kernel). Table 2 describes the interface for the kernels described in Section 3.1.

```
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);
}
```

Figure 4: BLAST application described in SARVAVID

```
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);
}
```

Figure 5: MUMmer application described in SARVAVID

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 4, 5, and 6. The kernels have been highlighted and a reader may see the similarity with C or C++ syntax, which is by design, as we find that genomics scientists are most comfortable with these languages.

4. COMPILATION FRAMEWORK

```
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);
}
```

Figure 6: E-MEM application described in SARVAVID

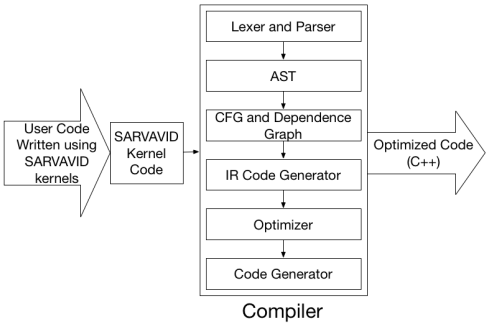

Figure 7: SARVAVID Compilation flow in SARVAVID

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 7 illustrates the overall structure of the compiler framework.

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.

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

Kernel	Interface	Scenarios in which kernel can be used
K-merization	`kmerize`($string$, int, int): $set\langle[int,int]\rangle$	Different k-mer lengths and interval
Index generation	`index_generation`($set\langle\tau\rangle$): $map\langle\tau_1,\tau_2\rangle$	Create a suffix tree, FM-Index, Hash Table
Index lookup	`index_lookup`($map\langle\tau_1,\tau_2\rangle$, τ_3): τ_4	Lookup in a suffix tree, FM-Index, Hash Table
Similarity computation	`similarity_computation`([int,int,int, int,int], $set\langle string\rangle$,$string$,int): [int,int,int,int,int]	Perform exact, ungapped, gapped alignment
Clustering	`clustering`($set\langle\tau\rangle$, e):τ	Perform connectivity based, density based clustering
Graph construction	`graph_construction`($set\langle\tau\rangle$):$map\langle\tau_1,\tau_2\rangle$	Specify different adjacency relationships
Graph traversal	`graph_traversal`($map\langle\tau_1,\tau_2\rangle$): τ	Specify DFS,BFS traversal
Error correction	`error_correction`($set\langle\tau\rangle$, e):τ	Use different functions like BayesHammer,IonHammer

Table 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

```
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

Figure 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

```
StA=k_merize(seqA,k,1);
A=index_generation(StA);
for(kmrb in seqB)
   index_lookup(A,kmrb);
StT=k_merize(seqA,k,1);
T=index_generation(StT);
for(kmrc in seqC)
   index_lookup(T,kmrc);
```

Before optimization

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

After optimization

Figure 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

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 8, seqB is scanned twice for lookup in the indices of seqA and seqB. SARVAVID fuses scanning loops over seqB to reduce cache misses.

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 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.

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 10 SARVAVID moves the loop invariant index generation kernel outside the loop.

```
for(i in num_seq)
  for(j in num_seq && j>i)
    Align(s[i],s[j]);

void Align(Seq s,Seq r)
{
  I=index_generation(s);
  lst=index_lookup(I,r);
  for(h in lst)
    similarity_
    computation(h,s,r);
}
```

Before optimization

```
for(i in num_seq){
  I=index_generation(s[i]);
  for(j in num_seq && j>i)
  {
     lst=
     index_lookup(I,s[j]);
     for(h in lst)
          similarity_
          computation(h,
          s[i],s[j]);
  }
```

After optimization

Figure 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

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,

```
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;
}
```

Figure 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.

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 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.* [26] to implement the partition and aggregate functions. Figure 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.

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 [10]. 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.

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".

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. The input datasets are based on relevant biological problems [5, 17].

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)

Table 3: Applications used in our evaluation and their input datasets.

5.2 Performance Tests

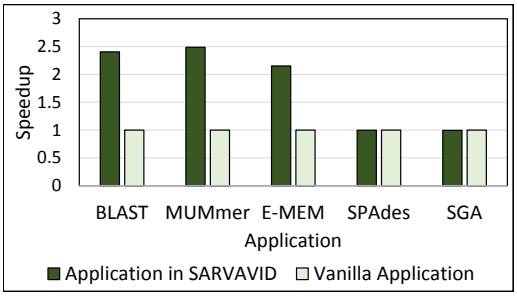

Figure 12: Performance comparison of applications implemented in SARVAVID over original (vanilla) applications. These are all runs on a single node with 16 cores. All except MUMmer are multi-threaded in their original implementations.

In this section we compare the performance of all applications provided with inputs in Table 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 12 shows that SARVAVID versions of applications perform at least as good as the baseline vanilla applications. For 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 various 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 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 13 does not include applications SGA and SPAdes since our compiler's optimizations do not apply to them.

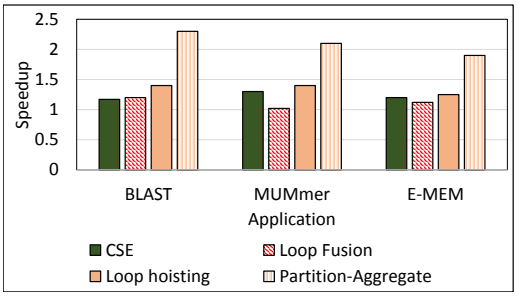

Figure 13: Speedup obtained by CSE, Loop Fusion, LICM, and parallelization (Partition-Aggregate) over a baseline run, *i.e.*, with all optimizations turned off, on a single node

5.3 Scalability

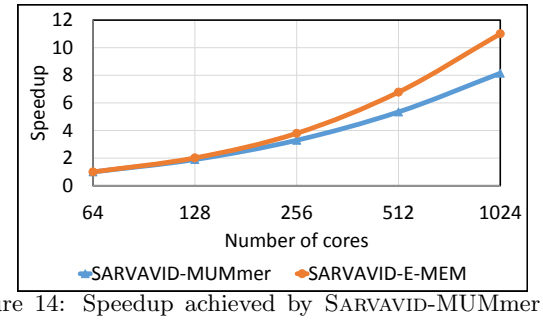

Figure 14: Speedup achieved by SARVAVID-MUMmer and SARVAVID-E-MEM calculated over 64 core runs of SARVAVID-MUMmer and SARVAVID-E-MEM respectively

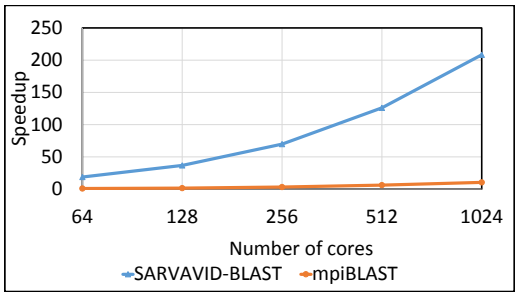

Figure 15: Speedup achieved by SARVAVID-BLAST calculated over 64 core run of mpiBLAST

Currently available versions of MUMmer and E-MEM are single node applications. With SARVAVID, these applications can be automatically scaled to multiple nodes. We 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 14 shows the scalability test. We attribute the lower scalability of MUMmer 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 [9]. mpiBLAST shards the database into multiple non-overlapping segments, and searches each segment independently and in parallel against the query. Figure 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.

5.4 Comparison with library-based approach (SeqAn)

Now we compare performance of applications written in SARVAVID against the well-established computational genomics library, SeqAn [11]. We chose to compare performance of MUMmer written in Seqan and SARVAVID since standard version of MUMmer in Seqan is available [1]. Figure 16 shows the execution time comparison of SeqAn-MUMmer, SARVAVID-MUMmer, and MUMmer over a range of inputs.

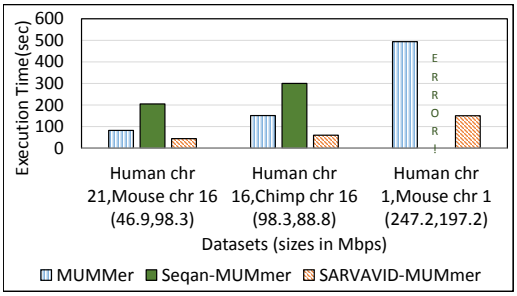

Figure 16: Comparison of execution times for MUMmer implemented in SARVAVID and MUMmer implemented using a popular genomics sequence comparison library called SeqAn. The results are all on a single node with 16 cores

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.

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 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.

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

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

Table 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++

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.

6. RELATED WORK

To address the urgent need for developing new computational genomics applications and tools, many libraries have been proposed [11, 40, 13, 18, 7, 12, 38, 33, 32, 16]. 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 [11] 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 [7] 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 [40] 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 [18] library provides implementations to perform machine learning and data mining tasks such as micro-

array processing and functional ontology while GenomeTools [16] 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++ [13], and PAL [12] 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.

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 paper, 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.

Acknowledgments

The authors would like to thank Amit Sabne for his helpful comments regarding exploiting compiler optimizations and other anonymous reviewers for their feedback. This work was supported in part by NSF grant CCF-1337158 and by the Purdue Research Foundation. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1053575.

8. REFERENCES

[1] Demo mummer. http://docs.seqan.de/seqan/2.0.0/page_DemoMummy.html.

[2] Stephen F Altschul, Warren Gish, Webb Miller, Eugene W Myers, and David J Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, 1990.

[3] Anton Bankevich, Sergey Nurk, Dmitry Antipov, Alexey A Gurevich, Mikhail Dvorkin, Alexander S Kulikov, Valery M Lesin, Sergey I Nikolenko, Son Pham, Andrey D Prjibelski, et al. Spades: a new genome assembly algorithm and its applications to single-cell sequencing. *Journal of Computational Biology*, 19(5):455–477, 2012.

[4] Serafim Batzoglou, David B Jaffe, Ken Stanley, Jonathan Butler, Sante Gnerre, Evan Mauceli, Bonnie Berger, Jill P Mesirov, and Eric S Lander. Arachne: a whole-genome shotgun assembler. *Genome research*, 12(1):177–189, 2002.

[5] Jessica Bolker. Model organisms: There's more to life than rats and flies. *Nature*, 491(7422):31–33, 2012.

[6] Stefan Burkhardt, Andreas Crauser, Paolo Ferragina, Hans-Peter Lenhof, Eric Rivals, and Martin Vingron. q-gram based database searching using a suffix array (quasar). In *Proceedings of the third annual international conference on Computational molecular biology*, pages 77–83. ACM, 1999.

[7] Davin Butt, Andrew J Roger, and Christian Blouin. libcov: A c++ bioinformatic library to manipulate protein structures, sequence alignments and phylogeny. *BMC bioinformatics*, 6(1):138, 2005.

[8] Tracy Craddock, Colin R Harwood, Jennifer Hallinan, and Anil Wipat. e-science: relieving bottlenecks in large-scale genome analyses. *Nature reviews microbiology*, 6(12):948–954, 2008.

[9] Aaron Darling, Lucas Carey, and Wu-chun Feng. The design, implementation, and evaluation of mpiblast. *Proceedings of ClusterWorld*, 2003:13–15, 2003.

[10] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[11] Andreas Döring, David Weese, Tobias Rausch, and Knut Reinert. Seqan an efficient, generic c++ library for sequence analysis. *BMC bioinformatics*, 9(1):11, 2008.

[12] Alexei Drummond and Korbinian Strimmer. Pal: an object-oriented programming library for molecular evolution and phylogenetics. *Bioinformatics*, 17(7):662–663, 2001.

[13] Julien Dutheil, Sylvain Gaillard, Eric Bazin, Sylvain Glémin, Vincent Ranwez, Nicolas Galtier, and Khalid Belkhir. Bio++: a set of c++ libraries for sequence analysis, phylogenetics, molecular evolution and population genetics. *BMC bioinformatics*, 7(1):188, 2006.

[14] Francisco Fernandes and Ana T Freitas. slamem: efficient retrieval of maximal exact matches using a sampled lcp array. *Bioinformatics*, 30(4):464–471, 2014.

[15] Sante Gnerre, Iain MacCallum, Dariusz Przybylski, Filipe J Ribeiro, Joshua N Burton, Bruce J Walker, Ted Sharpe, Giles Hall, Terrance P Shea, Sean Sykes, et al. High-quality draft assemblies of mammalian genomes from massively parallel sequence data. *Proceedings of the National Academy of Sciences*, 108(4):1513–1518, 2011.

[16] Gordon Gremme, Sascha Steinbiss, and Stefan Kurtz. Genometools: a comprehensive software library for efficient processing of structured genome annotations. *Computational Biology and Bioinformatics, IEEE/ACM Transactions on*, 10(3):645–656, 2013.

[17] Leonard Guarente and Cynthia Kenyon. Genetic

pathways that regulate ageing in model organisms. *Nature*, 408(6809):255–262, 2000.

[18] Curtis Huttenhower, Mark Schroeder, Maria D Chikina, and Olga G Troyanskaya. The sleipnir library for computational functional genomics. *Bioinformatics*, 24(13):1559–1561, 2008.

[19] W James Kent. Blatthe blast-like alignment tool. *Genome research*, 12(4):656–664, 2002.

[20] Zia Khan, Joshua S Bloom, Leonid Kruglyak, and Mona Singh. A practical algorithm for finding maximal exact matches in large sequence datasets using sparse suffix arrays. *Bioinformatics*, 25(13):1609–1616, 2009.

[21] Nilesh Khiste and Lucian Ilie. E-mem: efficient computation of maximal exact matches for very large genomes. *Bioinformatics*, 31(4):509–514, 2015.

[22] Stefan Kurtz, Adam Phillippy, Arthur L Delcher, Michael Smoot, Martin Shumway, Corina Antonescu, and Steven L Salzberg. Versatile and open software for comparing large genomes. *Genome biology*, 5(2):R12, 2004.

[23] Tak Wah Lam, Wing-Kin Sung, Siu-Lung Tam, Chi-Kwong Wong, and Siu-Ming Yiu. Compressed indexing and local alignment of dna. *Bioinformatics*, 24(6):791–797, 2008.

[24] Ben Langmead, Kasper D Hansen, Jeffrey T Leek, et al. Cloud-scale rna-sequencing differential expression analysis with myrna. *Genome Biol*, 11(8):R83, 2010.

[25] Ben Langmead and Steven L Salzberg. Fast gapped-read alignment with bowtie 2. *Nature methods*, 9(4):357–359, 2012.

[26] Kanak Mahadik, Somali Chaterji, Bowen Zhou, Milind Kulkarni, and Saurabh Bagchi. Orion: Scaling genomic sequence matching with fine-grained parallelization. In *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*, pages 449–460. IEEE, 2014.

[27] Andréa Matsunaga, Maurício Tsugawa, and José Fortes. Cloudblast: Combining mapreduce and virtualization on distributed resources for bioinformatics applications. In *eScience, 2008. eScience'08. IEEE Fourth International Conference on*, pages 222–229. IEEE, 2008.

[28] Aaron McKenna, Matthew Hanna, Eric Banks, Andrey Sivachenko, Kristian Cibulskis, Andrew Kernytsky, Kiran Garimella, David Altshuler, Stacey Gabriel, Mark Daly, et al. The genome analysis toolkit: a mapreduce framework for analyzing next-generation dna sequencing data. *Genome research*, 20(9):1297–1303, 2010.

[29] Saul B Needleman and Christian D Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970.

[30] Tung Nguyen, Weisong Shi, and Douglas Ruden. Cloudaligner: A fast and full-featured mapreduce based tool for sequence mapping. *BMC research notes*, 4(1):171, 2011.

[31] Henrik Nordberg, Karan Bhatia, Kai Wang, and Zhong Wang. Biopig: a hadoop-based analytic toolkit for large-scale sequence data. *Bioinformatics*, page btt528, 2013.

[32] WR Pitt, Mark A Williams, M Steven, B Sweeney, Alan J. Bleasby, and David S. Moss. The bioinformatics template libraryŮgeneric components for biocomputing. *Bioinformatics*, 17(8):729–737, 2001.

[33] Peter Rice, Ian Longden, Alan Bleasby, et al. Emboss: the european molecular biology open software suite. *Trends in genetics*, 16(6):276–277, 2000.

[34] Michael C Schatz. Cloudburst: highly sensitive read mapping with mapreduce. *Bioinformatics*, 25(11):1363–1369, 2009.

[35] Matthew B Scholz, Chien-Chi Lo, and Patrick SG Chain. Next generation sequencing and bioinformatic bottlenecks: the current state of metagenomic data analysis. *Current opinion in biotechnology*, 23(1):9–15, 2012.

[36] Jared T Simpson and Richard Durbin. Efficient de novo assembly of large genomes using compressed data structures. *Genome research*, 22(3):549–556, 2012.

[37] Temple F Smith and Michael S Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.

[38] Jason E Stajich, David Block, Kris Boulez, Steven E Brenner, Stephen A Chervitz, Chris Dagdigian, Georg Fuellen, James GR Gilbert, Ian Korf, Hilmar Lapp, et al. The bioperl toolkit: Perl modules for the life sciences. *Genome research*, 12(10):1611–1618, 2002.

[39] Zachary D Stephens, Skylar Y Lee, Faraz Faghri, Roy H Campbell, Chengxiang Zhai, Miles J Efron, Ravishankar Iyer, Michael C Schatz, Saurabh Sinha, and Gene E Robinson. Big data: Astronomical or genomical? *PLoS Biol*, 13(7):e1002195, 2015.

[40] 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.

[41] Michaël Vyverman, Bernard De Baets, Veerle Fack, and Peter Dawyndt. essamem: finding maximal exact matches using enhanced sparse suffix arrays. *Bioinformatics*, 29(6):802–804, 2013.

[42] Lauren 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.