SparkLeBLAST: Scalable Parallelization of BLAST Sequence Alignment Using Spark

Karim Youssef
Department of Computer Science
Virginia Tech
Blacksburg, VA, USA
karimy@cs.vt.edu

Wu-chun Feng
Department of Computer Science
Virginia Tech
Blacksburg, VA, USA
feng@cs.vt.edu

***Abstract*—The exponential growth of genomic data presents challenges in analyzing and computing on such biological data at scale. While NCBI's BLAST is a widely used pairwise sequence alignment tool, it does *not* scale to large datasets that are hundreds of gigabytes (GB) in size. To address this scalability problem, mpiBLAST emerged and became widely used, enabling scaling to 65,536 processes. However, mpiBLAST suffers from being tightly coupled with a specific implementation of BLAST, rendering it difficult to upgrade with the ever-evolving NCBI BLAST code. To address this shortcoming, recent parallel BLAST tools, such as SparkBLAST, consist of wrappers that are decoupled from the BLAST code but suffer from poor scalability with large sequence databases. Thus, there does *not* exist any parallel BLAST tool that can simultaneously address the issues of performance, scalability, programmability, and upgradability. To address this void, we propose SparkLeBLAST, a parallel BLAST tool that leverages our performance modeling and the Spark framework to deliver the performance and scalability of mpiBLAST and the ease of programming and upgradability of SparkBLAST, respectively. Ultimately, SparkLeBLAST delivers a 10x speedup relative to the state-of-the-art SparkBLAST and nearly a 2x speedup relative to the latest version of mpiBLAST.**

***Index Terms*—scalable genome analysis, BLAST, Spark, distributed computing, parallel computing, bioinformatics, sequence alignment, mpiBLAST, SparkBLAST**

I. Introduction

Advances in genome sequencing technology [1] have led to an exponential growth in the size of genomic databases. On the one hand, this wealth of data has created a plethora of opportunities for groundbreaking research in multiple fields, including cancer research and evolutionary biology. On the other hand, it has created a "big data" bottleneck for high-performance computing (HPC) systems that perform such large-scale genomic analysis. This, in turn, provides the motivation for continued research into *scalable genomic analysis* tools that are easy to develop, maintain, and upgrade while also being easy to use.

The Basic Local Alignment Search Tools (BLAST) is a widely used pairwise sequence alignment algorithm. Pairwise sequence alignment is a fundamental building block in the genomic analysis that finds similarities between biological sequences. BLAST uses a heuristic approach to identify regions of similarity between pairs of sequences. The heuristic nature of BLAST makes it significantly faster than exact sequence alignment algorithms such as Smith-Waterman [2] and Needleman-Wunsch [3].

The input to a BLAST search is a set of query sequences and a database of known sequences. The query size typically runs in the range of hundreds of megabytes (MB), while a single genomic database can be as large as hundreds of gigabytes (GB), three orders of magnitude larger than the typical query size. The output for every pair of query and database sequences consists of a description of the regions of similarity, the similarity score, and the statistical significance of the similarity.

BLAST is implemented and maintained by the National Center for Biotechnology Information (NCBI) [4]. Despite the performance advantage of BLAST over the locally optimal Smith-Waterman algorithm, it suffers from being bottlenecked on large, "big data" genomic databases. Specifically, the performance of BLAST on pairwise sequence alignment is limited by a computer system's memory wall [5] and I/O wall. For example, a typical BLAST search can take more than *two weeks* to complete a sequential BLAST search for a genome against a large genomic database. However, with the advances in parallel and distributed computing back in the 2000s, a parallel BLAST tool like mpiBLAST can complete the search in one hour [6].

As a consequence, mpiBLAST rapidly garnered significant popularity and has been widely used in many biological studies due to its high scalability and accuracy [6], [7]. Specifically, mpiBLAST easily scales up to 65,536 processors [8]; however, due to the tight coupling with a specific version of NCBI BLAST, mpiBLAST requires significant effort and expertise to maintain and upgrade. For instance, the latest release of mpiBLAST, which wraps around NCBI BLAST in a semi-custom way, was in 2012; since then, NCBI has released multiple updated versions of the BLAST code, but mpiBLAST has *not* been updated in tandem to the changes in the NCBI BLAST code, thus arguably obsolescing mpiBLAST. Despite that, the latest version of mpiBLAST [7] continues to be the most scalable among all parallel BLAST tools, including [9], [10], and [11].

We acknowledge Virginia Tech's Advanced Research Computing (ARC) and the Biocomplexity Institute (BI) for providing the computing resources for our research as well as Sajal Dash and Sarunya Pumma for providing constructive feedback to improve our work.

SparkBLAST [12], a more recent parallel BLAST tool, uses Spark [13] to parallelize pairwise sequence alignment. SparkBLAST leverages the programmability of Spark to parallelize NCBI BLAST by using a wrapper that consists of only 10 lines of code. As a consequence, SparkBLAST is highly maintainable and upgradable with minimal effort. However, SparkBLAST only parallelizes BLAST by using a technique called *query segmentation*, an embarrassingly parallel approach that partitions a query file into individual queries to run in parallel across compute nodes while having to replicate the database on each node. Query segmentation has two major limitations. First, it incurs significant I/O overhead for databases that are larger than the system memory of a single compute node. Second, parallelizing via query segmentation results in a significant load imbalance between parallel workers. Based on previous studies, the BLAST search times of different sequences of the same genome can vary significantly [14]. A similar tool to SparkBLAST is CloudBLAST [15], a parallel BLAST tool based on Hadoop [16] instead of Spark. However, CloudBLAST suffers from the same limitations as SparkBLAST does.

Other tools include PCJ-BLAST [17], HPC-BLAST [18], and the data-parallel BLAST for commodity clusters [19]. PCJ-BLAST uses query segmentation with static load balancing, hence lacking scalability with large databases. On the other hand, HPC-BLAST and the data-parallel BLAST use *database segmentation*; however, they still suffer from poor scalability due to centralized output processing.

In this paper, we present SparkLeBLAST, a parallel BLAST tool that leverages our performance model to deliver the performance and scalability of mpiBLAST with the programmability and upgradability of SparkBLAST. Our main contributions are summarized below:

- SparkLeBLAST, an open-source[1] parallel BLAST tool based on Spark that delivers performance, scalability, programmability, and maintainability.
- A performance model of distributed parallel BLAST that estimates its execution time based on input size and basic characteristics of the system architecture.
- Performance that is up to 10x faster than SparkBLAST and nearly 2x faster than the latest version of mpiBLAST using a large-scale biological database, i.e., the non-redundant protein sequence dataset (nr) [20].

The rest of this paper is organized as follows: Section II describes and validates our performance model of parallel BLAST for different data partitioning approaches. Section III describes the design and implementation of SparkLeBLAST. Section IV evaluates the performance of SparkLeBLAST, compared to SparkBLAST and mpiBLAST. Section V concludes the paper and highlights future directions.

II. Parallel BLAST Performance Model

The main design goals of SparkLeBLAST are to leverage its programmability, upgradability and fault tolerance while also scaling to large databases that do *not* fit in the memory of a single compute node. The intuitive solution is to partition the database such that each partition fits in memory.

Database segmentation would address two main issues incurred by SparkBLAST's query segmentation approach. The first issue is load imbalance, which is caused by variation in the execution time of each query sequence; the second issue is the I/O paging overhead for databases that do not fit in the memory of a single node. However, database segmentation comes at the expense of additional overhead to merge and sort the final output.

Previous studies, including [21], [7], [22], and [23], have extensively analyzed the performance of data-parallel BLAST for different partitioning schemes. The most comprehensive study was performed by Lin et al. [7], which analyzed the trade-off between load balance, I/O, and communication. This study yielded the sophisticated implementation of mpiBLAST v1.6 that enabled hierarchical partitioning with a wide range of configurations and some heuristics that guided the choice of the optimal configuration. However, Lin et al. emphasized that the optimal configuration is highly system and workload dependent. Since the characteristics of computer clusters, e.g. I/O performance, network bandwidth, memory size, evolve continuously, the heuristics provided by Lin et al. need to be revisited over and over. To address this problem, we devised a simplified model for the performance of data-parallel BLAST that estimates the total execution time as a function of input data and basic characteristics of distributed computing clusters.

We begin by describing the main components and parameters of our performance model. After that, we describe and validate the performance model for parallel query segmentation and database segmentation. We then highlight new insights gleaned from our model to conclude this section.

A. Model Parameters

BLAST is sequence similarity algorithm that uses a heuristic approach to identify regions of local similarity between pairs of sequences. The asymptotic upper bound of running a BLAST search on two sequences q and d is $O(|q||d|)$, where $|q|$ denotes the number of characters in sequence q. Hence, for a query $Q = \{q_1, q_2, ..., q_n\}$ and a database $D = \{d_1, d_2, ..., d_m\}$, the asymptotic upper bound execution time is $O(|Q||D|)$, where $|Q|$ is the total number of characters in the query, i.e. $|Q| = \sum_{i=1}^{n} |q_i|$, and likewise for $|D|$ [24].

The heuristic nature of BLAST prunes the search space to a small fraction of the total $|Q||D|$. Thus, a simplified expression for the total execution time of a sequential BLAST search can be expressed as follows:

$$T_{search_{sequential}} = c_s * t_c * |Q| * |D| \quad (1)$$

where c_s denotes the fraction of explored search space and t_c denotes the time per comparison of a pair of characters. We explain how to estimate the values of c_s and t_c later in this section.

At a high level, the total execution time of the data-parallel BLAST can be expressed as follows:

[1] https://github.com/vtsynergy/SparkLeBLAST

$$T_{parallel} = T_{search} + T_{I/O} + T_{comm} \quad (2)$$

where T_{search} denotes the (maximum) search time of a pair of sequences, $T_{I/O}$ denotes the I/O time needed to transfer data between secondary memory (e.g., disk) and main memory, and T_{comm} denotes the communication time necessary for query splitting, broadcasting, and then merging of final results.

To estimate each of the terms in Equation (2), we need to identify the input size, number of cores, number of nodes, memory size, file system characteristics, network bandwidth, and many other parameters. Table I provides a description of these parameters that are needed for our performance model.

TABLE I: Parameters of parallel BLAST performance model

Term	Description
$\lvert Q\rvert$	total number of characters in query
$\lvert D\rvert$	total number of characters in DB
k	number of query sequences
P	total number of processors
N	number of nodes
M	memory per node in bytes
B	disk block size in bytes
P_B	number of blocks transferred in parallel from disk to memory
c_s	average number of aligned characters in a BLAST search
t_c	average time per comparison (seconds/pair of chars)
t_b	average time per block transfer (seconds/block)
B_w	interconnect bandwidth

B. Performance Model

With query segmentation, the query file is partitioned into P segments, where each segment is processed by a single processor (or equivalently, P/N segments per node), and the database is replicated onto each node. Because the computations of an individual query sequence are independent of computations of other query sequences, query segmentation is embarrassingly parallel. It does not incur any communication overhead other than concatenating the output files of all partitions. However, query segmentation suffers from two main drawbacks. First, query segmentation creates a load imbalance between parallel workers because the search time of different sequences differs significantly. Second, query segmentation suffers from significant paging overhead when the database does not fit in the memory of a single compute node.

To model the load imbalance in search time, we express the search time of query segmentation as follows in Equation (3):

$$T_{search_{qs}} = c_s t_c \max_i(|Q_i|)|D| \quad (3)$$

For large query files where the number of sequences is much larger than the number of processors available, it is possible to partition the query file such that every partition has a balanced mix of short and long sequences (approximately $\frac{Q}{P}$ characters per processor). However, if the number of sequences is not much larger than P, the distribution of the average length per partition will be similar to the distribution of the lengths per sequence, resulting in a significant load imbalance.

As for I/O time, because the database is replicated onto each node, the I/O time complexity of query segmentation, in terms of the number of blocks that are transferred from secondary memory to main memory, is $\Omega(\frac{N|D|}{B})$ in the best case. When the database does *not* fit in memory (i.e., $|D| > M$), this complexity becomes much higher. Specifically, the worst case would be that $\frac{|D|-M}{B*P_B}$ blocks would need to be swapped k times between main memory and secondary memory. An approximation of $T_{I/O}$ for query segmentation when the database does *not* fit in memory can be expressed as follows:

$$T_{I/O_{qs}} = t_b(\frac{NM + k(|D| + \frac{|Q|}{N} - M)}{B_{min}(P_B, P)}) \quad (4)$$

Equation (4) estimates the total number of disk blocks transferred from disk to memory and multiplies this total by the time per block transfer. NM represents the aggregate memory of the cluster, assuming that the database replicas fill the system memory at the beginning of execution. The term $k(|D| + \frac{|Q|}{N} - M)$ represents an approximation of the number of blocks that will be swapped during execution as memory becomes full, where $\frac{|Q|}{N}$ represents the approximate length of query segments per node. (For simplicity, we do not consider the intermediate output of BLAST, which is expected to further increase the I/O overhead.)

Finally, the T_{comm} component consists of the query split and broadcast at the start of the job and the concatenation of the final output partitions at the end. Preliminary experiments show that this component is constant and insignificant (i.e., less than 1%) for up to 1024 processors.

We validate our model for query segmentation by estimating the execution time from Equations (3) and (4) and comparing it with actual execution time of SparkBLAST, when running on the Cascades supercomputer at Virginia Tech, which comes equipped with a high-bandwidth interconnect of 100 Gbps and a GPFS file system. A detailed description of Cascades is provided in Section IV. The input data used in the model validation consists of the non-redundant protein database [20] and a random sample of 1000 query sequences from the same database.

We evaluate the values of c_s and t_c empirically by running and profiling the sequential NCBI BLAST using the data outlined above. The output of NCBI BLAST shows the number of aligned characters, which is usually much less than $|Q| * |D|$. The value of c_s represents an average of the ratio of aligned characters to the total number of characters (6.36×10^{-8} from our experiment); the value of t_c represents an average for the total search time divided by the total number of aligned characters (5.1×10^{-4} from our experiment). We obtain the values of t_b, B, and P_B from the GPFS performance sheet and file system configurations on Cascades. Figure 1 shows the actual and predicted execution times.

Database segmentation, on the other hand, works by partitioning the database among compute nodes and replicating the query on each node. The goal of database segmentation is to fit the database in the aggregate memory of the computing cluster. That is, if there are enough resources, each database partition (or segment) fits in the memory of a single compute node, which eliminates the paging overhead incurred by query

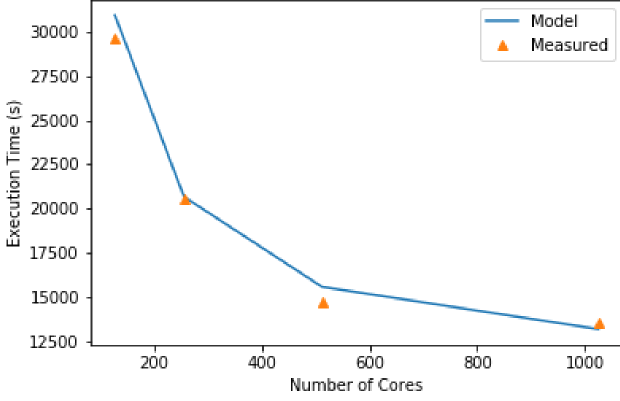

Fig. 1: *Estimated (Model) vs. Actual (SparkBLAST) execution times on Cascades. The average prediction error is 3.29%, while the maximum prediction error is 5.61%.*

segmentation. Database segmentation also yields more load balance than query segmentation, since the computations of the longest query sequences are distributed among more than one processor. However, these optimizations comes at the cost of communication overhead to merge and sort the final output of each query sequence.

The search time for database segmentation can be modeled in a way similar to that of query segmentation. It can be represented as follows:

$$T_{search_{dbs}} = c_s t_c |Q| \max_i(|D_i|) \tag{5}$$

However, since the number of sequences in the database is usually much larger than that of the query, more load balance would be guaranteed for a larger number of partitions. Accordingly, the search time for database segmentation can be approximated to:

$$T_{search_{dbs}} = c_s t_c |Q| \frac{|D_i|}{P} \tag{6}$$

This assumption was validated by previous work. For instance, [7] showed that the "search" component of the old mpiBLAST which used database segmentation scaled nearly linearly.

As for $T_{I/O}$, if there exist enough resources such that the database partitions fit in the memory of a single node, the I/O complexity for reading the database will not exceed $O(|D|)$.

The communication time in case of database segmentation consists of the time taken to merge and sort the final output. The merge step includes exchanging data between all compute nodes in order to collect the results of each query sequence or set of sequences in one node. This data exchange time would increase with the number of nodes as it causes interconnection network contention. The merge overhead depends on the output size, number of processors, and interconnect bandwidth. It is challenging to theoretically estimate the merge time as it is hard to estimate the output size as a function of input size [7]. Accordingly, we conducted an empirical analysis to estimate this overhead. We implemented the logic of processing the output of BLAST in the Spark framework. Our implementation processes the output by grouping results by query ID then sorting all the results in parallel. Details of the implementation are provided in section III. We first ran a BLAST job and collected its output. We then partitioned the output as if it was generated by a parallel BLAST job. And finally, we ran our Spark output processing code and measured the wall-clock time.

We measured the execution time of the merge step from 32 up to 1000 processors on Cascades supercomputer. We used these results to fit a linear regression model to estimate T_{comm} from the number of processors. The parameters of the regression model were 0.018 (slope), 7.87 (intercept), and 0.955 (R^2). Using these results along with equation 6, we estimated the performance of database segmentation in Spark. Figure 2 shows the estimated execution time versus the number of cores on a log-log scale. Our model estimates that such system would scale for up to 8192 cores before the communication overhead would start affecting scalability.

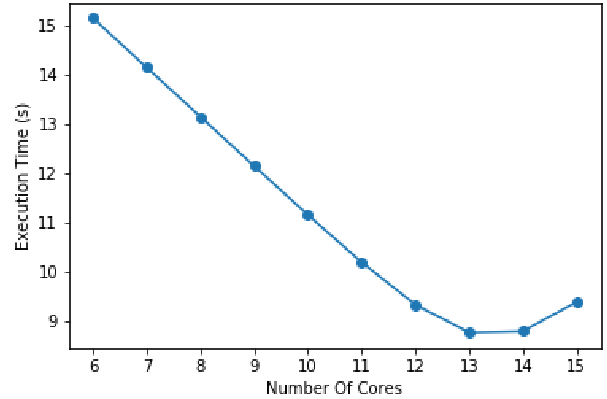

Fig. 2: *Estimated execution time for database segmentation (log-log scale). The merge time starts to affect scalability after 8192 cores.*

C. Insights

The main insights gained from modeling the performance of parallel BLAST are the trade-offs between I/O overhead, load balance, and communication overhead. There are two main insights gleaned from our model:

1) By modeling system and workload characteristics, we automate identifying near-optimal partitioning of input data.
2) We showed that on a fairly modern supercomputer, the number of processors at which communication overhead starts slowing down the performance of database segmentation is pushed further away from what was identified by previous studies including [7].

III. Design And Implementation of SparkLeBLAST

The main design goal of is to combine the scalability of mpiBLAST with the upgradability of SparkBLAST. The scalability of mpiBLAST is due to database and query segmentation in a way that optimizes I/O, load balance, and communication between parallel workers. However, mpiBLAST was developed by modifying multiple components of a specific version of NCBI BLAST code, which makes it costly to upgrade with the ever-evolving NCBI BLAST code. On the other hand, SparkBLAST is a highly upgradable system that consists of a Spark wrapper around unchanged NCBI BLAST code; however, query segmentation limits its scalability with databases that do not fit in the memory of a single compute node.

To achieve upgradability and scalability, we combined the design of SparkBLAST, in terms of building a Spark wrapper around unchanged NCBI BLAST code, with database segmentation. Moreover, we revisited the design of mpiBLAST and leveraged our performance model to identify the tradeoff point between communication, I/O, and load balance. Our performance model estimated that database segmentation scales up to 8192 cores on a supercomputer with a high-bandwidth interconnect of 100 Gbps and a parallel file system. Accordingly, the current version of SparkLeBLAST adopted database segmentation for parallelization.

In order to scale beyond 8192 cores, the next version of SparkLeBLAST will realize hybrid segmentation. SparkLeBLAST 2.0 will leverage the performance model to estimate the maximum scalability of database segmentation on a given system, and then running a coordinator task that will partition the query and launch two or more SparkLeBLAST jobs on each query partition. This design is similar to mpiBLAST v1.6; however, it does not impose a maximum limit on the number of database segments and uses the model to identify the optimal partitioning as a function of the system and the workload characteristics.

The rest of this section describes the software architecture and implementation of SparkLeBLAST. It also describes its workflow and usage. The section organization is as follows. First, we provide a brief background of the Spark programming model. After that, we describe the main system components, workflow, and implementation of each component of SparkLeBLAST. Finally, we describe the usage of SparkLeBLAST on a supercomputer.

A. Spark Programming Model

The motivation behind choosing the Spark framework is its efficient in-memory data processing, programmability, and fault tolerance. Spark abstracts distributed memory computing through the concept of Resilient Distributed Datasets (RDDs) [25]. An RDD is an in-memory immutable distributed collection of objects. The Spark programming model defines computations on RDDs in terms of transformations (e.g., map) and actions (e.g., collect). Transformations perform computations on RDDs and produce new RDDs, while actions produce and write the final output. One of the RDD transformations supported by Spark is called *pipe()*. Spark supports the calling of an external process (e.g., a script) via an API called *pipe()*, which passes the entries of an RDD object to the external process, and passes its output back to the Spark code as a new RDD. SparkBLAST used the *pipe()* API to call the NCBI BLAST binary as an external process. In SparkLeBLAST, we used the same approach with a small adaptation to work with database segmentation. A detailed description of how *pipe()* was used in SparkLeBLAST implementation is provided later in this section.

A Spark program typically runs on a Spark cluster. A Spark cluster consists of a master node and a set of worker nodes. The master node runs a *driver* program that coordinates tasks between workers, while each worker runs an *executor* program in parallel that performs computations on one or more segments of the input dataset. Spark features multi-node parallelism using multiple parallel worker nodes, as well as single node parallelism using multiple parallel tasks on the CPU cores of a single worker. Spark also features a fault-tolerance mechanism that efficiently reschedules failed tasks.

B. SparkLeBLAST Implementation

SparkLeBLAST consists of two main components, *SparkLeMakeDB*, and *SparkLeBLASTSearch*. *SparkLeMakeDB* takes as input a BLAST database in FASTA or FASTQ format, partitions the database into P segments where P is a user-defined parameter, formats each segment into NCBI BLAST format, then stores the formatted segments. *SparkLeBLASTSearch* takes as input a query file in FASTA or FASTQ format, as well as a formatted and partitioned BLAST database. It runs parallel NCBI BLAST search by distributing database segments among parallel workers, replicating the query to each worker, then calling NCBI BLAST binary using *pipe()* on each worker, with the database segment and the entire query as inputs. We provide a more detailed description of the two components below.

1) SparkLeMakeDB: The *SparkLeMaketDB* program partitions the database into segments and formats each segment using the *makedb* command from NCBI BLAST tool. This step is required once per database, and the formatted database can then be reused for subsequent BLAST searches. For this reason, we implemented *SparkLeMaketDB* as a separate program that only does the partitioning and formatting and should be called once before starting a BLAST search on a new database. The formatted and partitioned database could then be reused for subsequent searches with different queries. The input to *SparkLeMaketDB* consists of a path to a sequence database in FASTA or FASTQ format, and a *num_segments* argument that specifies the required number of segments. A preliminary experiment indicated that the optimal number of segments is equal to the number of available processors. An experiment with finer-grained segmentation was performed with dynamic load balancing in Spark but showed poor performance. The implementation of *SparkLeMaketDB* consists of three steps. First, reading the sequence database into an RDD and partitioning it according to the specified *num_segments*

argument. This step is implemented using the *newAPIHadoopFile()* Spark API that takes as arguments the path to the FASTA or FASTQ database as well as the number of segments. Next, the *RDD.pipe()* API is called which pipes each partition to the NCBI BLAST *makedb* command (also called *formatdb* in older versions of BLAST). The output of *makedb* is returned by *pipe()* as a new RDD of formatted database segments. Formatted segments are then written in a persistent storage location specified as an input argument to the *SparkLeMaketDB* program. It is worth noting that the performance evaluation of SparkLeBLAST does not take into consideration the time taken by this step as it is a one time cost.

2) SparkLeBLASTSearch: The *SparkLeBLASTSearch* program performs a parallel BLAST search on the query file and the database segments by invoking NCBI BLAST in parallel using the *pipe()* API. Figure 3 describes the procedure of parallel BLAST search in SparkLeBLAST. This approach is similar to SparkBLAST. However, in SparkBLAST, the query file is first read by Spark, then the query segments are passed to the NCBI BLAST binary via *pipe()*. SparkBLAST's approach creates two steps of I/O, which could affect performance for large segments. In order to avoid this extra overhead in SparkLeBLAST, the driver program of *SparkLeBLASTSearch* initializes a list, *segmentsIDs = ["part-0000", "part-00001", ..., "part-P"]*, where *P* is the number of database segments. Each segment ID is then concatenated with the root path of the partitioned database. An RDD is then generated from the *segmentsIDs* list, which is distributed across worker nodes. The segments IDs RDD is then piped to the external NCBI BLAST binary command along with the path to the query file. This step runs on every task on every worker node in parallel. The invoked NCBI BLAST code then reads its assigned partition, performs the BLAST search, and returns the results as a new RDD to *SparkLeBLASTSearch*.

After all of the workers complete the BLAST search, the output of every query sequence need to be grouped and sorted by alignment score and e-value score (a measure of statistical significance of the alignment). In SparkLeBLAST, merging and sorting final output is performed in parallel using Spark's *reduceByKey()* and *sortBy()* APIs. The *reduceByKey()* API groups all values with the same key into a single worker, where the key is the query ID. After that, the *SortBy()* API sorts the grouped output of every query in parallel by alignment score then by e-value score, where each worker sorts its assigned group of query sequences. Figure 4 describes the merging process. As will be shown in the results section, the merging process is relatively fast and only represents a tiny fraction of the total execution time. It is also worth noting that Spark writes the final output into separate blocks in parallel, which makes the I/O overhead for writing the final output insignificant. Parallel merge and parallel I/O contribute significantly to the scalability of SparkLeBLAST by minimizing the bottlenecks of output merging and writing.

3) Correctness of Search Results: One challenge of parallelizing BLAST using database segmentation is the correct computation of the e-value scores. The e-value score is a statistical significance score that is a function of the entire lengths of the query and the database [24]. The NCBI BLAST code computes the lengths of the query and the database on the fly, which leads to an incorrect e-value score with database segmentation. Hence, the total length of the database needs to be known before the start of the BLAST search. The NCBI BLAST code provides the option of supplying the *effective length of the database* as a program argument. When provided, the code skips the computation of the length and uses the effective length instead. To get the total length of the database before launching the SparkLeBLAST search, we implemented a *MapReduce* job at the beginning of *SparkLeMakeDB* program that counts the number of sequences and the number of characters of the target database and saves them into a database specification file. The *SparkLeBLASTSearch* program then reads the database specification file and provides the total lengths to the NCBI BLAST code via the *pipe()* call as a command argument. This technique results in e-value scores that exactly match the output of sequential NCBI BLAST.

C. SparkLeBLAST Usage on a Supercomputer

SparkLeBLAST can run on any cluster that has Spark installed. In this section, we describe the usage of SparkLeBLAST on a Linux supercomputer governed by SLURM workload manager [26]. The workflow of running SparkLeBLAST on such a system consists of three steps. Deploying a Spark cluster, running *SparkLeMakeDB*, then running *SparkLeBLASTSearch*. It is worth noting that these same steps can be used to run SparkLeBLAST on other types of clusters, with the only difference of how the Spark cluster is deployed. SparkLeBLAST requires the installation of any version of the NCBI BLAST code. The version tested and used in our experiments is 2.2.21. This section provides a high-level overview of the usage and workflow of SparkLeBLAST. SparkLeBLAST code is shipped with a detailed step-by-step user guide for installation and usage.

1) Deploying a Spark Cluster Using SLURM: Most of the supercomputers at research institutions are shared between multiple research groups, where resources are managed by a workload manager like SLURM [26]. In such a setup, a user gains access to computing resources by submitting a job request that specifies the needed resources (e.g., cores, memory, and time). Since the resources are temporary, SparkLeBLAST requires the deployment of a Spark cluster via a SLURM job each time a user intends to use it. SparkLeBLAST code provides a script for deploying a Spark cluster using SLURM. The script is called *start_spark_slurm.sh* and was adapted from [27]. This script takes as input the resources needed for the Spark cluster (e.g., number of nodes, number of cores per node, memory, and time). The script deploys a Spark cluster and saves the address of the master node in a location specified by the user as input.

2) Running SparkLeMakeDB: This step assumes that a Spark cluster is already deployed. A script named *SparkLeMakeDB.sh* launches the Spark job that partitions and formats the BLAST database. This script takes as input the

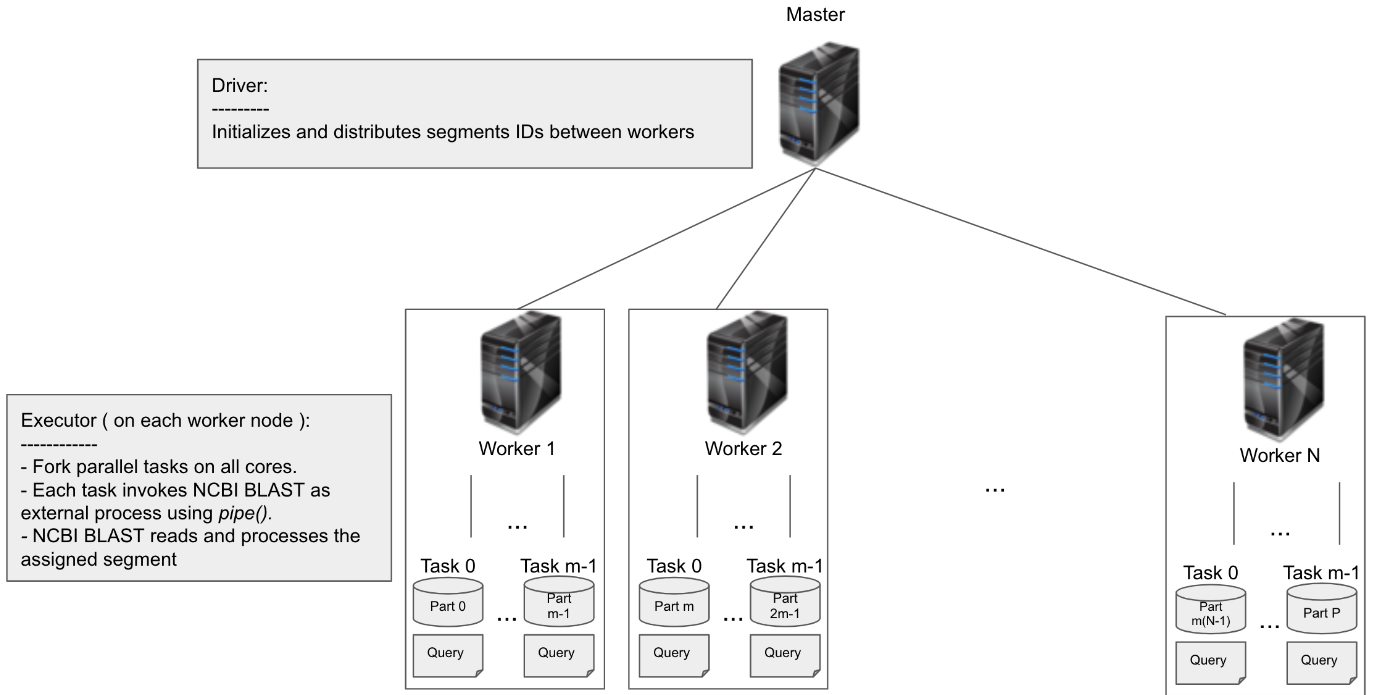

Fig. 3: *SparkLeBLAST parallel BLAST search using database segmentation. The master node runs the driver program that assigns database segments to workers. Every worker runs the executor program on its allocated database segment(s). The executor program spawns multiple parallel tasks, where every task performs NCBI BLAST search on its allocated segment.*

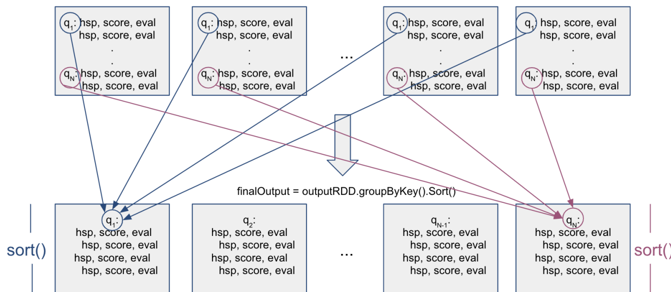

Fig. 4: *For each query sequence, the output consists of a High-scoring Segment Pair (HSP), an alignment score, and a statistical significance score (e-value). Results of every query sequence are grouped and sorted by score and e-values in parallel.*

address of the Spark Master node, the path to a BLAST database in FASTA or FASTQ format, the number of segments, and any NCBI BLAST command-line option. The output of *SparkLeMakeDB* is a directory containing the partitioned and formatted database. The output path is passed to the script as a command-line option.

3) Running SparkLeBLASTSearch: This step assumes that there is a Spark cluster deployed and that there is a BLAST database formatted and partitioned using *SparkLeMakeDB.sh*. To run a Parallel BLAST search, a script named *SparkLeBLASTSearch.sh* is used that takes as input the address of a Spark Master node, the paths to the query and the partitioned and formatted database, and any NCBI BLAST command-line options. The output of *SparkLeBLASTSearch* follows the same output formats of NCBI BLAST and is stored in a file specified by the user.

IV. Performance Evaluation

A. Hardware Platform Description

The performance evaluation of SparkLeBLAST was conducted on two clusters with different characteristics that are described below:

1) BlueRidge: BlueRidge is a Cray CS300 cluster of 408 nodes. Each node consists of two octa-core Intel Sandy Bridge CPUs and 64 GB of memory. For storage, the cluster is connected with a shared Lustre file system. BlueRidge is shared amongst multiple research groups. The maximum allowed cores per user is 1040 cores. In the experiments described below, 1024 cores were used for parallel BLAST search, and the rest of the cores were used to run the master node for SparkBLAST and SparkLeBLAST, and the super-master + master nodes for mpiBLAST [7].

2) Cascades: Cascades consists of 236 nodes. Each node is equipped with two 16-cores Intel Broadwell CPUs and 128 GB of memory. The storage consists of a shared GPFS file system. Cascades is equipped with a 100 Gbps Infiniband interconnect between nodes, and a 10 Gbps Ethernet interconnect with the file system servers. The maximum allowed number of cores per user is 1024. Our experiments demonstrated scaling for up to 1000 cores.

B. Data Description

The performance of SparkLeBLAST was evaluated using actual genome data. The query used is a bacterial genome called Geobacter metallireducens [28], which consists of 3,592 sequences, and a total of 1,233,413 base pairs (i.e., characters). The total size of the query is 1.5 MB. The database consisted of the non-redundant protein database (nr) [20]. The total size of the formatted nr database is 130 GB. The nr database consists of 175,193,827 sequences and a total of 63,927,145,901 base pairs. It is worth noting that the experiments of SparkBLAST inspired the use of the query above. On the other hand, the use of the nr protein database was motivated by the fact that it is the largest open-access protein database to date.

C. Spark Configurations

Spark was configured to run one node as the master and all other nodes as workers. The number of cores per worker was set to the total number of cores per node. The memory per worker was set to be the maximum available memory on each node (124.6GB on Cascades). The number of cores reported in figures 5 and 6 represent the sum of the number of cores of all workers. Spark's event log was enabled in order to collect profiling information (e.g., the execution time of each task and each stage per task).

D. Results

As described in section III, the main performance goals of SparkLeBLAST are to eliminate non-search overhead (i.e., I/O and communication) and to achieve more load balance between parallel executors. Accordingly, the performance of SparkLeBLAST is evaluated based on two metrics; total execution time and imbalance percentage [29]. The total execution time is broken down into BLAST search time and non-search time. The number of cores is varied, ranging from 128 up to 1024 cores on BlueRidge and 1000 cores on Cascades to evaluate scaling.

SparkLeBLAST was compared with SparkBLAST and mpiBLAST. As described in section III, SparkBLAST adopts query segmentation to parallelize BLAST. It was assumed that the performance of SparkBLAST would suffer when the size of the database is larger than the size of the main memory of a compute node. The goal of comparing SparkBLAST with SparkLeBLAST was to quantify the I/O overhead incurred by query segmentation and how it would be mitigated by using database segmentation instead. This comparison also analyzed the merging overhead of database segmentation compared to the I/O overhead of query segmentation. This analysis showed that the latter is insignificant compared with the former. SparkBLAST is an open-source tool. SparkBLAST code was used without modifications except for using a different version of NCBI BLAST to match the version on top of which mpiBLAST was built.

The latest version of mpiBLAST (mpiBLAST-1.6) was built around NCBI BLAST version 2.2.21. While SparkLeBLAST and SparkBLAST support any version of NCBI BLAST, version 2.2.21 was used in order to avoid any performance variations caused by different versions. The mpiBLAST tool supports a wide variety of configurations in terms of data partitioning, ranging from query segmentation, to query and database segmentation, to only database segmentation. According to our performance model, on modern HPC clusters, database segmentation yielded the best performance. Hence, we configured mpiBLAST to use the maximum allowed number of database segments.

Scalability of SparkLeBLAST, SparkBLAST, and mpiBLAST is shown in Figure 5. It was observed that SparkLeBLAST is the fastest among the three tools. On 128 nodes of BlueRidge, SparkLeBLAST is 2.5 times faster than SparkBLAST and 1.85 times faster than mpiBLAST. By increasing the number of cores, it was observed that mpiBLAST and SparkLeBLAST are scaling nearly linearly, while SparkBLAST scales sub-linearly for up to 512 processors, then incurs a slowdown afterward. On 1024 processors, the difference between SparkLeBLAST and mpiBLAST became smaller with an advantage for SparkLeBLAST being 1.5 times faster. SparkBLAST incurred a slowdown on 1024 nodes where SparkLeBLAST was 11 times faster. On Cascades, though still significant, it was observed that the differences are less significant compared with BlueRidge. SparkLeBLAST ran 3.5 times faster than SparkBLAST and 1.35 times faster than mpiBLAST. These observations support the insights that were drawn from our performance model. Query segmentation suffers from significant I/O overhead that leads to poor scalability, while database segmentation scales nearly linearly. A further analysis was conducted to separate the search time from non-search time. Figure 6 shows the execution time breakdown into NCBI BLAST time and non-search time.

It was observed that for both SparkLeBLAST and mpiBLAST, the NCBI BLAST time is the dominating component. On BlueRidge, The average percentage of non-search time is 16.2% for SparkLeBLAST and 6% for mpiBLAST. For SparkBLAST, the non-search time is dominating the execution time, with an average percentage of 54.18%. While on Cascades, the average percentages of non-search time are 15.5%, 7.4%, and 48.5%, respectively.

Another factor that significantly contributes to the performance of parallel BLAST is the load balance between parallel workers. The imbalance percentage metric [29] was used to evaluate and compare the load balance in SparkLeBLAST, SparkBLAST, and mpiBLAST. The definition of imbalance percentage is as follows. In a parallel application with n workers, given the maximum worker time t_{max} and the average worker time t_{avg}, the imbalance percentage is equal to $\frac{t_{max}-t_{avg}}{t_{max}} * \frac{n}{n-1}$. The value of the imbalance percentage ranges from 0 to 1. A parallel application with a perfectly balanced workload will yield an imbalance percentage equal to 0 (i.e, lower is better). Figure 7 shows the imbalance percentage for the three tools run on 128, 256, 512, and 1024 cores on BlueRidge. SparkLeBLAST maintained the lowest imbalance percentage of all the three systems. Also, that the imbalance percentage of SparkLeBLAST is nearly constant.

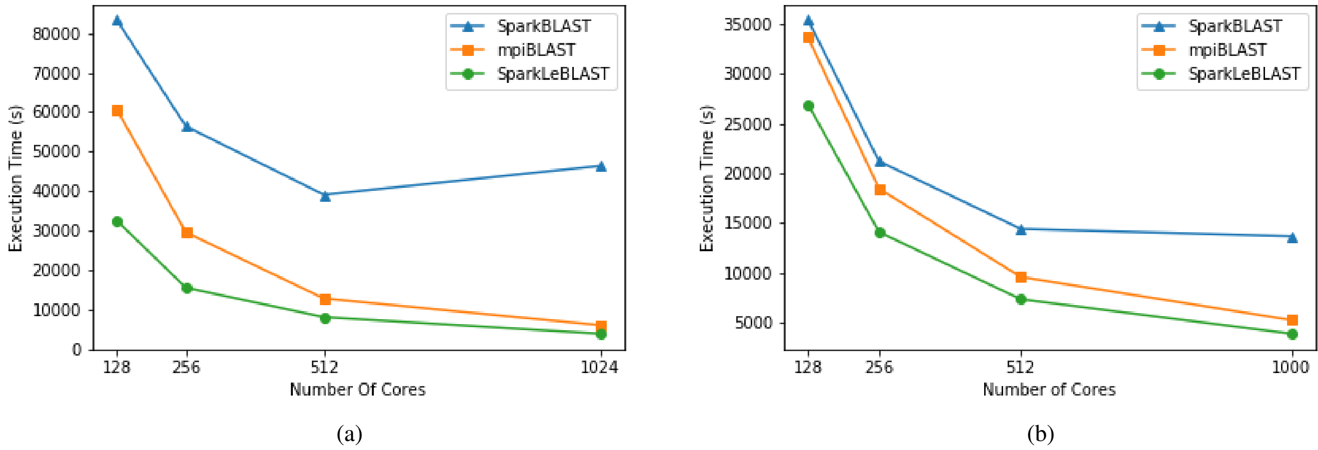

Fig. 5: Scalability of SparkLeBLAST, SparkBLAST, and mpiBLAST on BlueRidge (a), and Cascades (b).

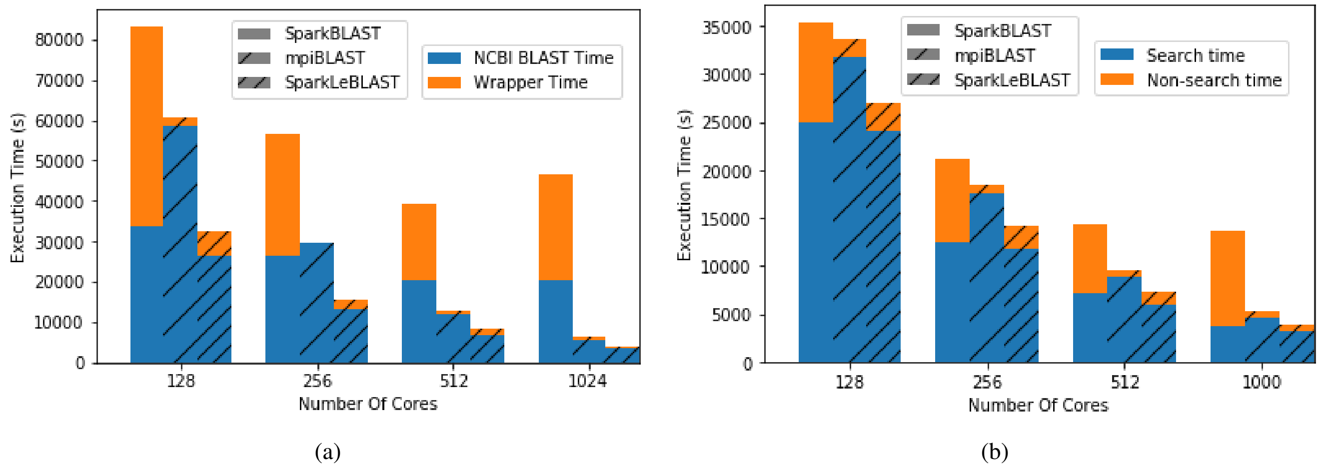

Fig. 6: Execution time breakdown into Search and Non-Search time for SparkLeBLAST, SparkBLAST, and mpiBLAST on BlueRidge (a), and Cascades (b).

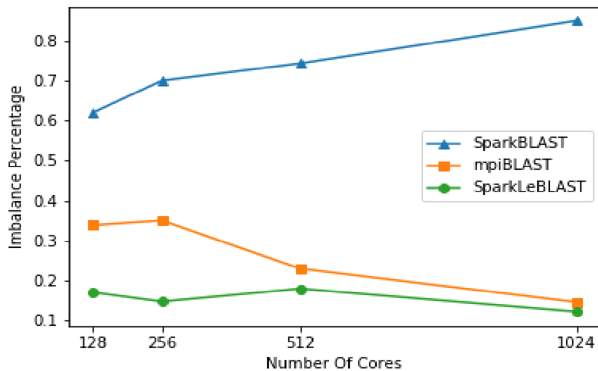

Fig. 7: *Imbalance percentage (lower is better) for SparkLeBLAST, SparkBLAST, and mpiBLAST, running on 128, 256, 512, 1024 cores of BlueRidge cluster.*

These observations suggest that database segmentation in parallel BLAST yields the most balanced workload. As for mpiBLAST, the load imbalance decreased when more cores were added. A suggested explanation of the decrease in load imbalance is that for a higher number of cores, there was more database segmentation and hence more load balance. For SparkBLAST, when more cores were added, there was a significant increase in imbalance percentage. This imbalance is due to the significant variance between BLAST search times of different query sequences. With a larger number of query partitions, there will be a small number of significantly slow workers that will bound the total execution time. At some point, there will be no significant gain in performance from adding more processors as slow workers will bound the speedup.

Finally, it is worth noting that the performance benefits of SparkLeBLAST were achieved with a significantly lower programming effort compared to mpiBLAST. The Spark wrap-

per in SparkLeBLAST consists of around 50 source lines of code, including the code to launch parallel workers, and the code for final output merging. On the other hand, the MPI code that parallelizes BLAST in mpiBLAST consists of over a thousand source lines of code. Also, the cost of upgrading SparkLeBLAST to work with newer versions of NCBI BLAST is zero lines of code, compared to nearly a thousand lines of code to upgrade mpiBLAST.

V. Conclusion and Future Work

This paper presents SparkLeBLAST, an efficient parallelization of NCBI BLAST sequence alignment using Spark. SparkLeBLAST combines the scalability of mpiBLAST, the most scalable parallel BLAST tool to date, with the upgradability of more recent tools such as SparkBLAST. SparkLeBLAST leverages our performance modeling, which demonstrated that on modern clusters, database segmentation with parallel output processing is estimated to scale for up to 8192 cores as it minimizes I/O overhead and increases load balance.

Performance evaluation on two different clusters demonstrated that SparkLeBLAST runs up to 10 times faster than SparkBLAST, and nearly 2 times faster than mpiBLAST for up to 1024 cores. In terms of upgradability and maintainability, SparkLeBLAST is completely decoupled from NCBI BLAST, leading to a zero cost of upgrading to newer versions, compared to a thousand line of code for mpiBLAST.

Subsequent future work includes (1) realizing hybrid (i.e., query and database) segmentation and leveraging the performance model to automatically predict the optimal partitioning, (2) generalizing SparkLeBLAST approach to other sequence alignment tools such as DIAMOND [30], and (3) extending the performance model as well as SparkLeBLAST to efficiently leverage heterogeneous computing systems (e.g., CPUs and GPUs) in order to achieve the maximum attainable speedup for sequence alignment.

References

[1] M. L. Metzker, "Sequencing technologies—the next generation," *Nature reviews genetics*, vol. 11, no. 1, p. 31, 2010.

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

[3] 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.

[4] D. Benson, M. Boguski, D. J. Lipman, and J. Ostell, "The national center for biotechnology information," *Genomics*, vol. 6, no. 2, pp. 389–391, 1990.

[5] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, Mar. 1995. [Online]. Available: http://doi.acm.org/10.1145/216585.216588

[6] A. E. Darling, L. Carey, and W. C. Feng, "The design, implementation, and evaluation of mpiblast," Los Alamos National Laboratory, Tech. Rep., 2003.

[7] H. Lin, X. Ma, W. Feng, and N. F. Samatova, "Coordinating computation and i/o in massively parallel sequence search," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 4, pp. 529–543, 2011.

[8] H. Lin, P. Balaji, R. Poole, C. Sosa, X. Ma, and W.-c. Feng, "Massively Parallel Genomic Sequence Search on the Blue Gene/P Architecture," in *ACM/IEEE SC—08: The International Conference on High-Performance Computing, Networking, Storage, and Analysis*, Austin, Texas, November 2008.

[9] R. D. Bjornson, A. Sherman, S. B. Weston, N. Willard, and J. Wing, "Turboblast (r): A parallel implementation of blast built on the turbohub," in *ipdps*. IEEE, 2002, p. 0183.

[10] J. D. Grant, R. L. Dunbrack, F. J. Manion, and M. F. Ochs, "Beoblast: distributed blast and psi-blast on a beowulf cluster," *Bioinformatics*, vol. 18, no. 5, pp. 765–766, 2002.

[11] C. Oehmen and J. Nieplocha, "Scalablast: A scalable implementation of blast for high-performance data-intensive bioinformatics analysis," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 8, pp. 740–749, 2006.

[12] M. R. de Castro, C. dos Santos Tostes, A. M. Dávila, H. Senger, and F. A. da Silva, "Sparkblast: scalable blast processing using in-memory operations," *BMC bioinformatics*, vol. 18, no. 1, p. 318, 2017.

[13] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[14] 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'06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. IEEE, 2006, pp. 22–22.

[15] 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.

[16] T. White, *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[17] M. Nowicki, D. Bzhalava, and P. Bała, "Massively parallel sequence alignment with blast through work distribution implemented using pcj library," in *International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 2017, pp. 503–512.

[18] S. Sawyer, M. Horton, C. Burdyshaw, G. Brook, and B. Rekapalli, "Hpc-blast: Distributed blast for modern hpc clusters." in *Proceedings of 11th International Conference*, vol. 60, 2019, pp. 1–14.

[19] M. Hajibaba, M. Sharifi, and S. Gorgin, "Data-parallel computational model for next generation sequencing on commodity clusters," in *International Conference on Parallel Computing Technologies*. Springer, 2019, pp. 273–288.

[20] "The non-redundant protein database," http://www.bioinf.org.uk/teaching/gradtrain/blast.html.

[21] O. Thorsen, B. Smith, C. P. Sosa, K. Jiang, H. Lin, A. Peters, and W.-c. Feng, "Parallel genomic sequence-search on a massively parallel system," in *Proceedings of the 4th international conference on Computing frontiers*. ACM, 2007, pp. 59–68.

[22] R. L. de Carvalho Costa and S. Lifschitz, "Database allocation strategies for parallel blast evaluation on clusters," *Distributed and Parallel Databases*, vol. 13, no. 1, pp. 99–127, 2003.

[23] J. C. Correa and G. P. Silva, "Parallel blast analysis and performance evaluation." in *BICoB*, 2011, pp. 211–218.

[24] 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.

[25] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.

[26] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 2003, pp. 44–60.

[27] A. Chu, B. Penneton, B. Sammuli, F.-A. Fortin, I. Lee, J. Asplund, M. Pathirage, N. Petrillo *et al.*, "Magpie," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2013.

[28] M. Aklujkar, J. Krushkal, G. DiBartolo, A. Lapidus, M. L. Land, and D. R. Lovley, "The genome sequence of geobacter metallireducens: features of metabolism, physiology and regulation common and dissimilar to geobacter sulfurreducens," *BMC microbiology*, vol. 9, no. 1, p. 109, 2009.

[29] L. DeRose, B. Homer, and D. Johnson, "Detecting application load imbalance on high end massively parallel systems," in *European Conference on Parallel Processing*. Springer, 2007, pp. 150–159.

[30] B. Buchfink, C. Xie, and D. H. Huson, "Fast and sensitive protein alignment using diamond," *Nature methods*, vol. 12, no. 1, p. 59, 2015.