DOI 10.12694/scpe.v18i1.1233
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 18, Number 1, pp. 51–66. http://www.scpe.org

SEQUENCE SIMILARITY PARALLELIZATION OVER HETEROGENEOUS COMPUTER CLUSTERS USING DATA PARALLEL PROGRAMMING MODEL

MAJID HAJIBABA*, SAED GORGIN*, AND MOHSEN SHARIFI†

Abstract. Sequence similarity, as a special case of data intensive applications, is one of the neediest applications for parallelization. Clustered commodity computers as a cost-effective platform for distributed and parallel processing, can be leveraged to parallelize sequence similarity. However, manually designing and developing parallel programs on commodity computers is a time-consuming, complex and error-prone process. In this paper, we present a sequence similarity parallelization technique using the Apache Storm as a stream processing framework with a data parallel programming model. Storm automatically parallelizes computations via a special user-defined topology that is represented as a directed acyclic graph. The proposed technique collects streams of data from a disk and sends them sequence by sequence to clustered computers for parallel processing. We also present a dispatching policy for balancing the cluster workload and managing the cluster heterogeneity to achieve more than 99 percent parallelism. An alignment-free method, known as n-gram modeling, is used to calculate similarities between the sequences. To show the cost-performance superiority of our method on clustered commodity computers over serial processing in powerful computers, we simply use UniProtKB/SwissProt dataset for evaluation of the performance of sequence similarity as an interesting large-scale Bioinformatics application.

Key words: commodity cluster, parallelization, sequence similarity, Apache Storm

AMS subject classifications. 68W10, 68R10

1. Introduction.

As a result of increases in the size of available genome sequence data, the processing and storage of such data have become one of the major challenges in modern genomic research. Bioinformatics applications such as finding genes in DNA sequences, or aligning similar proteins, all need complex computations and vast amounts of processing power beside massive memory capacity. Therefore, there is high demand for faster algorithms and more powerful computers to reduce the execution time of these applications.

For Bioinformatics applications, the primary form of information are raw DNA and protein sequences. Therefore, one of the first steps toward obtaining information about a new biological sequence is to compare it with a set of known sequences in existing sequence databases [1]. The results of comparisons often suggest functional, structural, or evolutionary analogies between sequences. For a given sequence, searching in sequences of protein databases is one of the most fundamental and important tools for predicting the structural and functional properties of uncharacterized proteins.

The similarity between two sequences (gene or protein) indicates that they may be derived from the same ancestral sequence by an evolution. This evolutionary similarity is called homology and similar sequences are known as homologous [2].

The most common metric showing the similarity between two instances is their distance. Computing distances between sequences in large datasets is a computationally intensive task [1]. To handle the complexity of this task,we can take advantage of parallelization techniques. Pairwise calculation provides a natural target for parallelization because all elements of the distance matrix are independent [1].

Using network-attached workstations as a platform for distributed and parallel computing is becoming more and more popular. This is due to the fact that todays desktop machines are extremely powerful as well as affordable. In contrast, dedicated supercomputers are often very expensive, so most research groups have only limited access to such machines. Commodity computers may be slower than commercial supercomputers, but they are readily accessible and usable for many tasks. Small networks of commodity computers can be used for coarse-grain parallelization of database searches. As a result, in response to the rapid increase in the amount of data generated by the new technologies, such as next-generation sequencing (NGS), researchers can use parallelization strategies and distributed systems on commodity clusters to accelerate the search in such datasets. However, the users of Bioinformatics tools are typically unaware of the advantages of emerging technologies in distributed computing. This is due to the difficulty of converting sequential jobs to distributed

*Department of Electrical Engineering and Information Technology, Iranian Research Organization for Science and Technology, Tehran, Iran (hajibaba.m@irost.org, gorgin@irost.ir).

†School of Computer Engineering, Iran University of Science and Technology, Tehran, Iran (msharifi@iust.ac.ir).

ones and the challenge of understanding distributed technologies well enough to create and manage a distributed environment [3]. Therefore, it can be useful to show how to parallelize Bioinformatics applications that can be followed by Bioinformaticians.

A practical study on using parallel computing to speed up the Bioinformatics problems has different aspects. A rich and complex study of parallel computing techniques and technologies for solving Bioinformatics problems has been explored in [1]. In this paper, we use sequence similarity as a case study of Bioinformatics problems for applying parallelization strategy. But, we focus on alignment-free sequence similarity methods. So, parallelization strategy for alignment-based methods such as BLAST [4, 5] is out of scope of this work. On the other hand, we try to find a software parallel solution to sequence comparison, and hardware parallel solutions are not in our scope. Various parallel architectures for sequence similarity search are compared in [6]. Moreover, some researchers in this scope have used parallel technologies such as GPU to speed up the sequence analysis algorithms [7, 8, 9]. In this paper, we concentrate on having applications that are run both in parallel and in a distributed manner and ignore solutions that only consider running applications in parallel like GPU solutions. We use commodity clustered computers to distribute sequence searches much the same as other researches that have parallelized sequence searches on workstation clusters, such as the work in [10] that uses an alignment-based method. There are also other works that emphasis on parallelization of sequence similarity search [11, 12, 13, 14], but few works use streaming to overcome big data challenges [15, 16, 17]. For example, [15] has proposed a stream-based approach for NGS read alignment based on the IBM InfoSphere Streams computing platform deployed on Amazon EC2.

We also use stream processing for a data and compute-intensive scientific problem (i.e. DNA sequence analysis). But, as a software solution, our strategy is based on a high-level parallel programming paradigm in an open source stream processing framework named Apache Storm. In order to get an efficient use of Storm on a heterogeneous commodity cluster, we have developed a scheduler to balance the workload on processing nodes. The used similarity method, in our work, is based on n-gram that is derived from statistical language modeling that uses the Markov chains representation together with cross-entropy from information theory to get a measure of similarity between n-grams [18]. We use stream processing to overcome computation challenges as well as memory challenges in this type of sequence similarity search method [19, 20].

More precisely, our contributions are as follows:

- We use protein-sequence streaming as a data partitioning strategy and utilize stream processing for coarse-grain search in a large database. Using protein-sequence streaming instead of database partitioning, we can reduce the cost of storage in some solutions such as in cloud computing.
- We build a task dispatching strategy (i.e., a weighted round-robin scheduler) to provide a balanced workload among heterogeneous clustered computers. In this strategy, we dispatch tasks with corresponding data to processing elements. So, each element takes proper sized subsets of the dataset, according to its computing power.
- We deploy Apache Storm for parallelization of a word-based sequence similarity search method in a semi-automatic programmer directed manner.

The rest of this paper is organized as follows. Section 2 outlines the scope of the work, the method used for sequence similarity and the parallel framework used in this paper. Section 3 describes the parallelization concerns of our work and used strategy to overcome these concerns in a real heterogeneous distributed environment. Section 4 presents how we have evaluated the performance and effectiveness of our proposed strategy. Section 5 concludes the paper.

2. Background.

2.1. Sequence comparison. Sequence comparison is used to detect the similarity of proteins for explaining phylogenetic relations between them. Existing sequence comparison methods can be classified into two main categories:

1. Alignment-based methods.
2. Alignment-free methods.

Alignment-based methods [21, 22] use dynamic programming to find the optimal alignment. These methods assign scores to different possible alignments and select the alignment with the highest score. The similarity score between two sequences in alignment-based methods is measured by an alignment algorithm (e.g., BLAST

[23] or FASTA [24]). However, the alignment-based methods have computational limitations on large biological databases and suffer from speed limitation [19, 25].

Alignment-free methods [26, 27] have been developed to alleviate the limitations of alignment-based methods [20]. The similarity score of two sequences in an alignment-free method is measured by various distance metrics, such as Euclidean distance or Kullback-Leibler divergence [28].

The alignment-free methods can be subdivided into three different classes: gene contents, data compression, and word composition [19]. The latter that has received considerable attention counts the number of words shared between a subject and a query sequence. Basically, in word-based methods, an optimal word length k is chosen, and then the similarity of any two sequences is assessed by comparing the frequencies of all subsequences of k adjacent letters (also called n-gram, k-mer, k-tuple or k-word) in sequences. High computing time and memory usage are disadvantages of all these three methods [19, 20]. Nonetheless, word-based methods have received credit in high-throughput classification of genome sequences and are deployed for diverse objectives in genomics such as sequence clustering and word similarity searches [29].

Some of the word-based methods [30, 31], determine the similarity between sequences by using a Markov model to estimate the probability that a sequence is relevant to a given query with respect to their word frequencies. In this paper, we deploy a similar method to calculate similarities between sequences, which is described in Section 2.2.

2.2. Similarity method. Amino acids consist of 20 different symbols that can be considered as the alphabet of a hypothetical language. Protein sequences can also be treated as texts written in this language. With this analogy, we can apply statistical language techniques in biological sequence analysis [18].

The unique order of amino acids in sequences affects the similarity of proteins. These orders can be obtained by n-gram based modeling techniques and used in similarity methods with cross-entropy related measures [18]. In these techniques, a Markov chain is built for n-gram models. For example, in Drosophila gene eyeless sequence [32], amino acids in the range of 30 to 40 are $EAVEASTASH$. Available tokens of this gene in 2-gram model are continuous base pairs with length 2, which are $\{EA, AV, VE, EA, AS, ST, TA, AS, SH\}$, while in the 3-gram model, are continuous base pairs with length 3, which are $\{EAV, AVE, VEA, EAS, AST, STA, TAS, ASH\}$.

The Shannon entropy [33] can be used to show how a sequence is fitted by an n-gram model, which is estimated by equation 2.1 [34].

$$H(x) = -\sum_{w^*} P(w_i^n)\ logP(w_{i+n-1}|w_i^{n-1}) \tag{2.1}$$

In equation 2.1, w_i^n corresponds to $\{w_i, w_{i+1}, \ldots, w_{i+n-1}\}$. The summation is over all feasible combinations of successive w with size n, i.e. $w = \{\{w_1, w_2, \ldots, w_n\}, \{w_2, w_3, \ldots, w_{n+1}\}, \ldots\}$. $P(w_i^n)$ is a joint probability that can be broken down using the Chain Rule to yield equation 2.2.

$$P(w_i^n) = P(w_i)\ P(w_{i+1}|w_i)\ P(w_{i+2}|w_i^2)\ \ldots\ P(w_{i+n-1}|w_i^{n-1}) = \prod_{k=i}^{i+n-1} P(w_k|w_i^{k-i}) \tag{2.2}$$

In equation 2.2, $P(w_{i+n-1}|w_i^{n-1})$ is the conditional probability of the nth word of a n-gram with respect to the previous $n-1$ words. Using the maximum likelihood estimation (MLE) [33], this term can be estimated by using the n-gram frequencies:

$$P(w_{i+n-1}|w_i^{n-1}) = \frac{C(w_i^{n-1}\ w_{i+n-1})}{C(w_i^{n-1})} = \frac{C(w_i^n)}{C(w_i^{n-1})} \tag{2.3}$$

In equation 2.3, $C(w_i^n)$ is the count of w_i^n. For example, in the aforementioned Drosophila gene eyeless sequence [32], the probability of bigram AS is:

$$P(A|S) = \frac{C(AS)}{C(A)} = \frac{2}{3} = 0.66$$

and its bigram model is illustrated in Figure 2.1, after calculating MLE for each token.

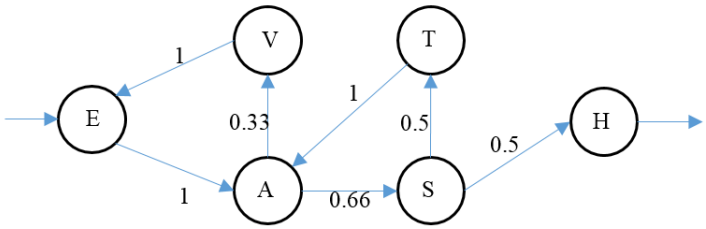

FIG. 2.1. *The bigram model of the sequence of EAVEASTASH*

If equation 2.1 is applied to two different sequences X and Y, the outcomes show which sequence is better fitted by the model, but cannot be used for their direct comparison. Thus, the similarity between two sequences must be represented as equation 2.4.

$$H(x,y) = -\sum_{w^*} P_X(w_i^n)\ logP_Y(w_{i+n-1}|w_i^{n-1}) \tag{2.4}$$

In equation 2.4, $P_X(w_i^n)$ relates to the subject sequence X and $P_Y(w_{i+n-1}|w_i^{n-1})$ relates to the query sequence Y whose model has to be estimated. Variable w_i^n runs over all the words (or n-grams) of the query protein sequence.

Now, if we let S_r be a subject sequence and $S = S_1, S_2, \ldots, S_N$ be the target protein database, then we can compute the similarity between them by using equation 2.4. The first step is the computation of reference score for the subject protein by calculating $H(S_r, S_r)$. Next, each sequence in the target database, $S_i (i = 1, \ldots, N)$ is given as the model sequence for calculating $H(S_r, S_i)$. After these calculations, we can have N dissimilarities that are the absolute differences against the reference score $D(S_r, S_i) = |H(S_r, S_i) - H(S_r, S_r)|$, for all $i = 1, \ldots, N$. Finally, by ordering these N dissimilarity measures, we can simply find the most similar sequences with the lowest distance to the subject sequence.

For protein coding genes, a tuple size of 3 can be a good choice [35] with less sensitivity and computation, but for more accuracy in this paper, we have chosen $n = 4$.

2.3. Apache Storm. Apache Storm is a real-time stream processing framework built by Twitter that is available as an Apache project [36]. After the batch processing became popular, it was realized that in many cases, the turnaround time was unacceptable. Given the needs for low-latency asynchronous stream processing solutions, Storm has tried to address these needs.

Storm is a scalable, fault-tolerant, distributed and real-time platform that provides distributed stream processing in a cluster. The Storm programming paradigm consists of Streams, Spouts, Bolts, and Topology. A Stream is a continuous unlimited sequence of data that can be processed in a parallel and distributed fashion. A Spout acts as a source that Streams come from and a Bolt takes the input Streams, processes them and produces new output Streams. The Spouts and Bolts are organized in a directed acyclic graph (DAG) called a Topology (Figure 2.2). An application developer that uses Storm can define an ideal Topology to describe communications between processing elements in his/her application and declare data dispatching mechanisms between processing elements. In Storm, data placement and distribution are provided by this user-defined Topology. Bolts and Spots in this topology are connected together with stream grouping. Stream grouping defines how Streams should be distributed among the Bolt's tasks. For example, in shuffle grouping, tuples are randomly spread over the Bolt's tasks such that each Bolt gets an equal number of tuples or in all grouping, the Stream is replicated across all tasks of the Bolt. Storm does not use intermediate queues to pass Streams between tasks. Instead, Streams are passed directly between tasks using batch messages at the network level to achieve a good balance between latency and throughput. Storm employs a master-worker computational model wherein the master is the main process that distributes tasks among the workers. Each worker process runs a Java Virtual Machine (JVM), on which it runs one or more executors. An executor is a thread in the system that is generated by a worker process and runs within the workers JVM. Executors are made of one or more tasks. The actual work for a Bolt or a Spout is done with the task. A worker process executes a subsection of a Topology on its own JVM.

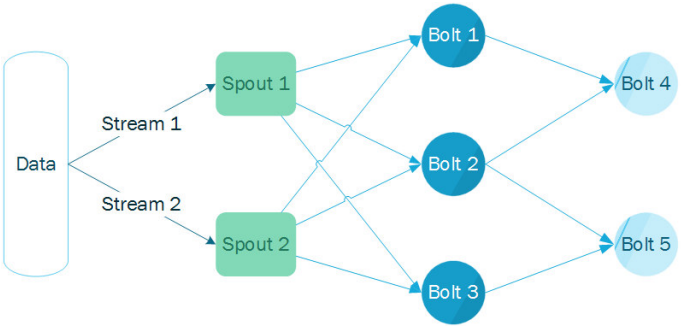

FIG. 2.2. *An example of Storm topology (components with the same color have the same work)*

```
Config conf = new Config();
conf.setNumWorkers(3); // use three worker instances

// define new topology
TopologyBuilder topologyBuilder = new TopologyBuilder();

// create Spout tasks using an object of user defined class of GreenSpout
// with two executors (threads) that should be assigned to execute this bolt
topologyBuilder.setSpout("green-spout", new GreenSpout(), 2);

// create Bolt tasks using an object user defined class of NavyBlueBolt
// with three tasks that should be assigned to execute this bolt in one executor (thread)
topologyBuilder.setBolt("navy-blue-bolt", new NavyBlueBolt(), 1).setNumTasks(3)
        .allGrouping("green-spout");

topologyBuilder.setBolt("blue-bolt", new BlueBolt(), 2).shuffleGrouping("navy-blue-bolt");

StormSubmitter.submitTopology("mytopology", conf, topologyBuilder.createTopology());
```

FIG. 2.3. *An example program in Storm*

The Storm resource manager divides nodes into slots and for each worker node, the user configures how many slots exist on that node. The Storm default scheduler uses a simple round-robin strategy [37]. When a job is submitted to the master, the scheduler counts all available slots on the worker nodes and assigns worker instances to nodes one by one in a round-robin manner. It deploys workers so that each node in a topology has almost an equal number of worker instances. The scheduler gets a job and decomposes it into tasks based on the topology, which can be delivered to slots.

Here are some reasons why Storm has been chosen as our choice for stream processing. Firstly, it is one of the leading open source stream processing tools that guarantees scalable, reliable and fault-tolerant processing of data. Secondly, compared with other leading open source tools such as Apache Spark, Storm processes the incoming streams in real time without any intentional latency. Thirdly, Storm has an easy programming paradigm (in contrast to map-reduce in Spark) and is usable with many programming languages (supports both JVM and non-JVM languages) without requiring any additional new concepts (such as Resilient Distributed Dataset (RDD) in Spark). Figure 2.3 shows a simple example Java code in Storm for the topology represented by Figure 2.2.

At last, but not least, Storm can be deployed on clustered commodity computers with low memory (1GB) and old architecture, while Spark needs high memory capacity for in-memory processing.

3. Distributed computation and parallelization. As mentioned in Section 1, Bioinformatics is confronted with increasingly large datasets. Processing of these datasets takes unacceptably long time on a single computer. Therefore, distributed computing on multiple computers is becoming an attractive approach to achieve shorter execution times [3]. In this section, we describe how to distribute sequence comparison jobs using a stream processing model on commodity computers to process large data in parallel.

Sequence comparison tasks can run on parallel processors using either coarse-grain or fine-grain methods [6]. In a coarse-grain method, sequences are partitioned equally among the processing elements that are appropriate for searches in a large database. So equal sized subsets of the database are assigned to processing elements, one subset to one processing element. Then, each element performs sequence analyses independently on its subset of the database. If there is a small number of sequences, coarse-grain parallelism may not be a good method. In these cases, a fine-grain method can be used to get a better speedup [6]. In our work, in order to evaluate the search in a huge database on a commodity cluster, we use coarse-grain parallelization to minimize the interactions between parallel processes.

Manually developing parallel programs is a time consuming, complex and error-prone process. Up to now, various tools have been available to assist programmers in converting serial programs into parallel ones. Designing and developing parallel programs by these tools requires the use of a parallel programming model such as a message passing or a data-parallel model. In distributed-memory parallel computation, a data-parallel model does not need to pass messages and thus makes the programming easier. A well-known representation of the data-parallel model is a Directed Acyclic Graph (DAG) in which nodes represent application tasks and directed arcs represent inter-task dependencies. One such tool that offers semi-automatic parallelization and follows the data-parallel model in a DAG is the Apache Storm that was described in Section 2.3.

The Storm programming model provides a distributed model for processing of streams on processing elements, wherein each element processes the input stream. Since batch processing is a specialization of the stream processing [38], we can use stream processing frameworks to do batch processing works. In this sense, we can aggregate the streams arriving in a time period and pass the aggregate to a processing element. So, Storm assumes the incoming streams as a batch and a set of batch tasks can be executed in parallel in a stream processing framework. In this manner, each task takes a chain of inputs, processes them using a user-defined function, and produces a result.

For sequence similarity, in this paper, a simple topology is used in Storm. In this topology, a Spout receives data and constructs a sequence from character streams to prepare the data elements in protein-sequence streaming. The Spout sends the result to main processing elements (i.e. Bolts) to compare them with the subject sequence. It sends sequences asynchronously to Bolts via a queue channel, so Bolts have their input data ready for execution. If the queue does not get empty during execution, with a sufficient number of Bolts, we can fully utilize the CPU cycles of the nodes. The subject sequence can be retrieved from a distributed coordination system such as Apache Zookeeper [39]. Sequences are sent from Spout to Bolts with a statically balanced Weighted Round Robin method that is described later (Algorithm 2). This architecture is illustrated in Figure 3.1.

For all schedulers in a distributed system, fairness is a critical feature to be considered. The round robin strategy as the default Storm scheduler in a shared heterogeneous cluster, which has multiple nodes with different hardware configurations, may not be fair. In default scheduling, although the processing is done in parallel, a job waits for its slowest worker to finish.

To balance the workload, we must partition the database into a number of portions according to the number and the power of processors allocated. To do this, our scheduling algorithm gets benchmarking data (as described in Algorithm 1) and then dispatches data elements according to their class weights to balance the workload of the cluster (as described in Algorithm 2).

By using the weights that are taken from benchmarks, the scheduling described in Algorithm 2 minimizes the imbalance ratio in comparison to the default scheduler; the imbalance ratio is the ratio of largest and smallest load on a processing instance, that we measure it in the next section.

In the next section, we compare the performances of the default scheduling and our proposed scheduling, as well as the cost-adjusted-performance of the clustering.

4. Experiments and results. There are various ways to measure the performance of parallel codes. To be rigorous, the speedup obtained by a parallel implementation should be compared with the best available sequential code for that problem. Also of importance is the concept of efficiency, defined as the ratio of speedup to the number of processors, and the computation to communication ratio. But the speedup is not quite calculable due to heterogeneity of clustered computers. Also, sequence-search algorithms can be measured along many dimensions such as execution time (or speed), millions of cell updates per second (MCUPS), deployment cost,

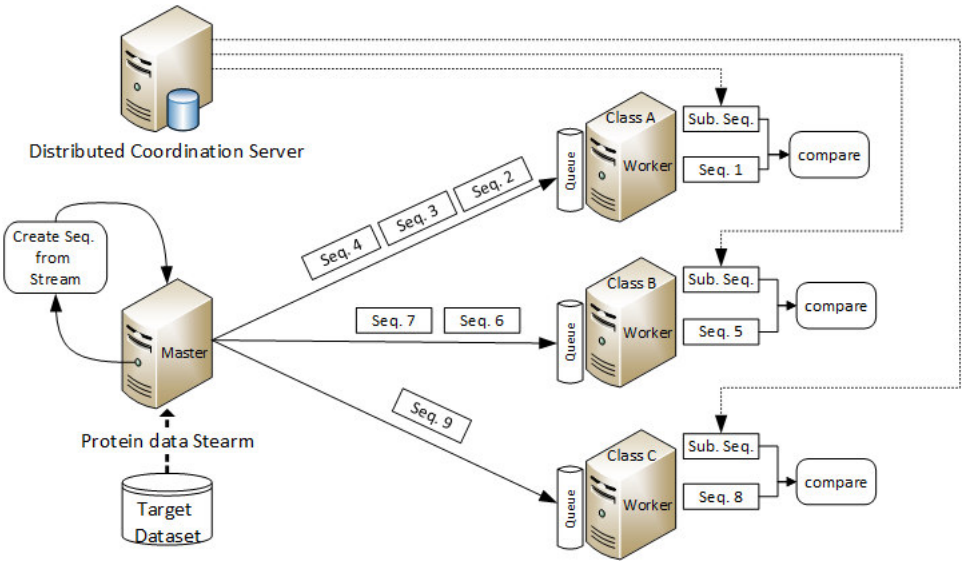

FIG. 3.1. *Sequence similarity distributed computing system overview*

Algorithm 1 Prepare information for choosing tasks

Input: $Benchmark$ is information receive from benchmarking phase
Output: $WorkersMeta$ is a List of workers' metadata

1: Info {
   id, ▷ id of a worker that is running on a node
   $weight$, ▷ class weight of the worker
   $counter$ ▷ the count of tasks must execute on the worker
   };
2: $cluster \leftarrow$ fetch running workers
3: **for each** $worker \in cluster$ **do**
4: $id \leftarrow$ identification number of the worker
5: $weight \leftarrow$ derive the weight of node from $Benchmark$ using GCD
6: $counter \leftarrow weight * factor$ ▷ $factor$ used to round the weight to a proper counter (normally is 10)
7: $tmpInf \leftarrow (Info)[Id, weight, counter]$
8: $WorkersMeta.add(tmpInf)$
9: **end for**
10: **return** $WorkersMeta$

and sensitivity [40]. In this section we measure the imbalance load ratio and the speedup of our solution. At last, we investigate the cost-adjusted-performance that is the product of execution time multiplied by the deployment cost.

For evaluation, the proposed method has been applied to publicly available protein database SWISS-PROT [41] as the target database to compare with the genome HIV-gp120 as a query sequence [42]. The input data (target and query) are all FASTA formatted sequence files. Characteristics of the target dataset and the query sequence are shown in Tables 4.1 and 4.2 respectively.

The studied similarity method has been benchmarked on several testbed systems in an effort to characterize their performance. We first present the performance on a commodity cluster with default Storm scheduler, demonstrating a poor performance caused by heterogeneity. Our comparative study of the performance of our proposed scheduling shows significant improvement in execution time.

Algorithm 2 Choose worker for processing the sequence

Input: $Task$ is a task for comparing two sequence,
$WorkersMeta$ is a List of workers' metadata
Output: $worker$ is the id of a worker to running the $task$

```
Let n be the size of WorkersMeta List
Let workerInf[1...n] be an Array of Info initiated from WorkersMeta
worker ← −1
Label : WorkerSelection
i ← 0
while i < n do
    counter ← workerInf[i].counter
    if counter > 0 then
        worker ← workerInf[i].id
        counter ← counter − 1
        break
    end if
    i ← i + 1
end while
if worker = −1 then
    for all worker ∈ workerInf do
        worker.counter ← worker.weight ∗ factor
    end for
    goto WorkerSelection
end if
return worker                    ▷Send Task to Worker
```

TABLE 4.1
SWISS-PROT dataset characteristics

Characteristic	value
Name	SWISS-PROT
Number of sequences	460,903
Shortest sequence	6
Longest sequence	35,213
Average nucleotides per sequence	374
Total number of nucleotides	172,370,171

Our test cluster had 10 compute nodes, each with 1GB memory and Intel processors with different architectures. Specifications of nodes deployed in the cluster, the execution time for each node to compare subject sequence with previously known target sequences in the dataset, and the number of each node are stated in Table 4.3. Compute nodes ran Apache Storm 0.9.3 under Ubuntu-12.04 to execute 1 or 2 workers per node according to the number of cores.

To evaluate our cluster, lets consider a cluster wherein we are given m workers for scheduling that are indexed by the set $W = w_1, .., w_m$. There are additional given n tasks for scheduling that are indexed by the set $T = t_1, .., t_n$. Let T_i be the set of tasks scheduled on w_i. Then, the load of the worker i (that is w_i) is defined by equation 4.1.

$$\ell_i = \sum_{t_i \in T} C_{i,j} \tag{4.1}$$

In equation 4.1, task t_j takes $C_(i, j)$ units of time if it is scheduled on w_i. The maximum load of scheduling

TABLE 4.2
SHIV-gp120 characteristics

Characteristic	value
Name	HIV1 HXB2 GP120
Length	481
Residues	31-511

TABLE 4.3
Specifications of the cluster nodes

CPU Model	# of Cores	Price[1]	Single Thread Execution Time	# of Nodes
Intel Pentium Processor E2180 (1M Cache, 2.00GHz, FSB 800MHz)	2	$0.5	1h 15m	1
Intel Pentium Processor E2160 (1M Cache, 1.80 GHz, FSB 800MHz)	2	$0.01	2h 10m	2
Intel Pentium 4 Processor HT (3.20GHz, 1M Cache, FSB 800MHz)	1	$2.2	1h 50m	3
Intel Pentium 4 Processor (2.40GHz, 512K Cache, FSB 533MHz)	1	$1.99	3h 30m	3
Intel Celeron D Processor 336 (256K Cache, 2.80GHz, FSB 533MHz)	1	$2.95	4h 20m	1

is called the makespan of the schedule and is given by equation 4.2.

$$\ell_{max} = \max_{i \in \{1,\ldots,m\}} \ell_i \tag{4.2}$$

In our test case, lets assume that T is the set of sequence comparison tasks that is submitted to the master. Each task is composed of two subtasks t_a and t_b. t_a is the task of constructing a sequence from Stream and t_b is the comparison of the sequence with the subject sequence. Since t_a is a lightweight task with respect to t_b, i.e. $t_b \gg t_a$, and t_a always runs on a fixed node (Spout node), we can ignore it from calculations. In addition, if we assume all sequences have the same length, so all tasks would be identical. In our dataset, the difference between sequence lengths is not significant, so for simplicity in calculations, we can assume that all tasks are identical. Sequence comparison tasks have also no data dependencies and are independent of each other in a non-preemptive manner.

Our cluster of computers was composed of 7 one core PCs and 3 double core PCs. To achieve a high level of parallelism, we used a fixed number of workers on each node according to its number of cores. Since we set cores per node as computing resources, we had 13 workers. One worker was used as Spout to collect and dispatch data sequentially. Therefore, twelve ($m = 12$) workers $W = w_1, w_2, \ldots, w_12$ were left for sequence comparison in parallel. Scheduling was carried out at worker level and each worker used First-In, First-Out (FIFO) method for performing the sequence comparison tasks.

By benchmarking, we estimated the time taken by each worker on a node to perform each task. The nodes in the cluster were classified by system benchmarking before submitting a topology. In the benchmarking phase, we ran several sequence comparisons on each node to get a task execution time. A task execution time was measured based on the number of sequences processed in a time period. The benchmarking results are stated in Table 4.4.

In this work, the workers are classified in p classes as $G = \{w_{g_1}, w_{g_2}, \ldots, w_{g_p}\}$ wherein g_i is an index for class i. So, the *makespan* will be $\ell_{max} = \max_{i \in \{1,\ldots,p\}} \ell_{g_i}$. Wherein $\ell_{g_i} = \sum_{t_j \in T_{g_i}} C_{g_i,j}$ denotes the load of

[1]These prices are taken from online marketplaces such as Amazon and eBay

TABLE 4.4
Execution time and performance on each type of node in the cluster testbed

Class (p)	**# of Slots**	**Job Completion Time**	**Sequences**	**Task execution time per worker (core)**	**Performance** ($\frac{sequence}{second}$)
1	2	75m = 4500s	460,903	0.0097	102
2	3	110m = 6600s	460,903	0.0143	70
3	4	130m = 7800s	460,903	0.0169	59
4	3	210m = 12600s	460,903	0.0273	36
5	1	260m = 15600s	460,903	0.0338	29

w_{g_i}, in which, task t_j takes $C_{g_i,j}$ units of time if scheduled on a worker with class i, and T_{g_i} denotes the set of tasks scheduled on w_{g_i}.

The Storm default scheduler uses a simple round-robin strategy that grants the same number of tasks ($\frac{sizeofT}{m}$) to each worker. In our test case, each task is a sequence comparison from the dataset with the subject sequence that is 460903 tasks of sequence comparison. Therefore, the *makespan* of the default scheduler at Storm is the following:

$$\ell_{max} = \sum_{t_j \in T_{g_5}} C_{g_5,j} = \frac{460903}{12} \times C_{g_5,j} = 38409 \times 0.0338 \approx 1,298$$

in which,

$$T_{g_5} = \{t_k, t_{k+12}, t_{k+(2\times12)}, \ldots, t_{k+(38408\times12)}\} \mid 1 \leq k \leq 12, C_{g_5,j} \approx 0.0338$$

To efficiently use a parallel computer system, a balanced workload among the processors is required. To compare the effectiveness of balancing the workload, the percentage of load imbalance (PLIB) [43] is defined by equation 4.3.

$$PLIB = \frac{\ell_{max} - \ell_{min}}{\ell_{max}} \times 100 \tag{4.3}$$

$PLIB$ is the percentage of the overall processing time that the first finished processor must wait for the last processor to finish. This number also indicates the degree of parallelism. For example, if $PLIB$ is less than one, we achieve over a 99 percent degree of parallelism. Therefore, a computational method with a lower $PLIB$ is more efficient than another one with a higher $PLIB$. The workload is perfectly balanced if $PLIB$ is equal to zero [43].

For default scheduler, $PLIB$ is obtained as follows:

$$\ell_{min} = \sum_{t_j \in T_{g_1}} C_{g_1,j} = \frac{460903}{12} \times C_{g_1,j} = 38409 \times 0.0097 \approx 373$$

$$PLIB = \frac{\ell_{max} - \ell_{min}}{\ell_{max}} \times 100 = \frac{1298 - 373}{1298} \times 100 \approx 71$$

Therefore, in this case, we have less than 29 percent parallelism. In this work, the goal is to minimize the *makespan*, i.e. the completion time of the last tasks in the schedule. For this goal, we have to achieve ideal workload balance, where $PLIB = 0$. In our proposed scheduling method, each worker gets tasks according to its computation power (or weight). This means that $\ell_1 \approx \ell_2 \approx \ldots \approx \ell_{12}$.

To obtain the weight of each worker from the benchmark we calculate the greatest common divisor (GCD) of task execution times on nodes and set weights accordingly. Assuming that the GCD of task execution times

is d, then we can set the weight of workers as in equation 4.4.

$$Weight_{g_i} = \frac{\sum_{k=1}^{p} \frac{\text{execution time of } w_{g_k}}{d}}{\frac{\text{execution time of } w_{g_i}}{d}} \tag{4.4}$$

In our example $d = 13$ and we have

$$\sum_{k=1}^{5} \frac{\text{execution time of } w_{g_k}}{d} = \frac{\text{execution time of } w_{g_1}}{d} + \frac{\text{execution time of } w_{g_2}}{d} + \ldots + \frac{\text{execution time of } w_{g_5}}{d} = 7.5 + 11 + 13 + 21 + 26 = 78.5$$

So,

$$Weight_{g_1} = 10.4, Weight_{g_2} = 7.1, Weight_{g_3} = 6, Weight_{g_4} = 3.7, Weight_{g_5} = 3$$

In our test case, Storm selects one slot of a node in class 3 as master process. So we have

$$T_{g_i} = \frac{460903}{10.4 \times 2 + 7.1 \times 3 + 6 \times 3 + 3.7 \times 3 + 3} = \frac{460903}{74.2}$$

$$\ell_{g_1} = \sum_{t_j \in T_{g_1}} C_{g_1,j} = 6212 \times 10.4 \times 0.0097 \approx 627$$

$$\ell_{g_2} = \sum_{t_j \in T_{g_2}} C_{g_2,j} = 6212 \times 7.1 \times 0.0143 \approx 631$$

$$\ell_{g_3} = \sum_{t_j \in T_{g_3}} C_{g_3,j} = 6212 \times 6 \times 0.0169 \approx 630$$

$$\ell_{g_4} = \sum_{t_j \in T_{g_4}} C_{g_4,j} = 6212 \times 3.7 \times 0.0273 \approx 627$$

$$\ell_{g_5} = \sum_{t_j \in T_{g_5}} C_{g_5,j} = 6212 \times 3 \times 0.0338 \approx 630$$

$$\ell_{max} = \max \ell_{g_i} \approx 631$$

Thus, we expect that our scheduler should perform twice better than the default scheduler (based on their ℓ_{max}, i.e. 631 compared to 1298), which is also shown by our experimental results in Figure 4.1.

Figure 4.1 shows the turnaround times under the default scheduler and our proposed scheduling method according to the class of nodes in the cluster; the turnaround time includes time to load workers, submission time of tasks to workers, scheduling time, synchronization time, inter-node communication time, and delays in acknowledging.

Experimental results in Figure 4.1 show that our proposed method produces a smaller $PLIB$ than the default scheduler.

$$PLIB = \frac{\ell_{max} - \ell_{min}}{\ell_{max}} \times 100 = \frac{631 - 627}{631} \times 100 \approx 0.6$$

Our scheduling method also achieved more than 99 percent degree of parallelism.

Program speedup is another important measure of the performance of a parallel program. By conventional definition of the speedup, if machine M_1 can solve problem P in time T_1 and machine M_2 can solve the same problem in time $T2$, the speedup of machine M_1 over machine M_2 on problem P is represented by equation 4.5.

$$S_P = \frac{T_1}{T_2} \tag{4.5}$$

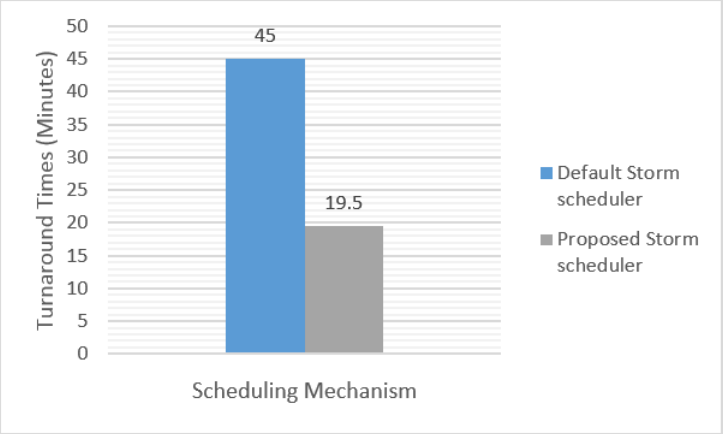

FIG. 4.1. *Comparison of turnaround times for each class of physical nodes and the whole cluster*

If machine M_2 is composed of m identical instances of the base processor of M_1, then we would expect m processors to solve the solution m times faster than a single processor. But this expectation is used in homogeneous clusters. For simple calculation of speedup in heterogeneous clusters, based on [44], if our cluster is a set $H = \{M_1, M_2, \ldots, M_m\}$ of m distinct machines, the speedup of a heterogeneous cluster C on problem P is defined by equation 4.6.

$$S_P = \frac{\min\{T_P^{M_1}, T_P^{M_2}, \ldots, T_P^{M_m}\}}{T_P^C} \tag{4.6}$$

In equation 4.6, $T_P^{M_k}$ is the execution time on machine M_k to solve problem P. In other words, the fastest machine is selected for comparison to the cluster. For our cluster with $m = 13$ based on equation 4.6 the speedup is:

$$S_P = \frac{\min\{4500, 6600, 7800, 12600, 15600\}}{1170} = \frac{4500}{1170} = 3.8$$

But this is not a good measure of speedup for our heterogeneous system due to the difference in nodes performance (slowest machine is 3.5 times slower than the fastest machine). An extension of the relative speedup for heterogeneous systems has been presented in [45], which is proved to be an appropriate tool for the evaluation of the speedup of parallel discrete event simulation in heterogeneous execution environments. So, accordingly, the relative speedup in this paper is calculated by equation 4.7.

$$r_P = \frac{N_e}{T_P^C \times P_c} \tag{4.7}$$

In equation 4.7, N_e is the number of events in the system. The cumulative performance of the heterogeneous system can this be defined by equation 4.8.

$$P_c = \sum_{i=1}^{NT} P_i \times N_i \tag{4.8}$$

Let us denote the number of CPU types in a heterogeneous system by NT, the number and the performance of CPU cores available from type i by N_i and P_i, respectively. Since our problem is near to discrete event simulation, the relative speedup or efficiency of our cluster will be:

$$P_c = 2 \times 102 + 3 \times 70 + 4 \times 59 + 3 \times 36 + 1 \times 29 = 787$$

TABLE 4.5
Cost and performance of the proposed method with other systems

System	# of Cores	Cost	Performance (turnaround time)
Intel Core i7-3770 Processor (8M Cache, up to 3.90GHz)	4	$344.99	26.5m
Intel Pentium G620 (3M Cache, 2.60GHz)	2	$78.12	40m
INTEL Xeon E5430 (12M Cache, 2.66GHz)	4	$40.0	48m
Proposed technique over given cluster	13	$16.04	19.5m

$$r_P = \frac{460903}{787 \times 1170} \approx \%50$$

This value is not bad at all for such heterogeneous clusters [45] with low network bandwidths, especially when the speedup is limited by the serial section (i.e. Spout) of the program.

Scalability becomes the basic need for distributed systems. Because of some problems such as heterogeneity of our cluster and lack of more nodes, we cannot practically test the scalability of our solution. However, we try to measure the scalability of our solutions compared to other similar works that scale well.

A similar work to ours is a streaming approach in a distributed architecture for scaling up natural language processing methods [46] which also leverages Storm. This work describes a series of experiments carried out with the goal of analyzing the scaling capabilities of the language processing. It shows that the use of Storm is effective and efficient for scalable distributed language processing across multiple machines when processing texts on a medium and large scale. Likewise, as mentioned earlier, the used similarity method in this work is a form of natural language processing [31] and implemented using Storm in a heterogeneous distributed environment. So, we can conclude our solution is scalable in nature and is efficient in the same environment.

Storm was originally aimed for processing twitter streams at scale [36]. The authors in [47] propose an event detection approach using Apache Storm to detect events within social streams like Twitter that can scale to thousands of posts every second. It is really a big challenge to analyze the bulk amount of tweets to get relevant and different patterns of information on a timely manner. We observed that the sequence similarity algorithm is an embarrassingly parallel problem that is highly parallel and can be deployed on a distributed system. We can consider sequences as events and similarity of a sequence with the reference as a rule to detect events. It has one Spout that reads events from document (dataset for our case) and a pipeline topology like ours. So, through experimentation on a large Twitter dataset by mentioned paper, we can conclude that our approach can scale to the equivalent size of data.

As a last example, the authors in [48], have addressed the problem of data stream clustering by developing scalable distributed algorithms that run on the Apache Storm. They demonstrate experimentally that they are able to gain close to linear scalability up to the number of physical machines. Likewise, our used similarity method was applied to the clustering sequences, based on Markov chains representation known as n-gram in statistical language modeling [18]. We used this sequence clustering strategy in a stream processing manner on Storm. Therefore, with an optimistic attitude, we can conclude that our solution is scalable at the same level as this work. However, there are some other works that can be used to inference the scalability of our solution [49, 50, 51].

In order to evaluate the performance-per-cost, we just use CPU costs as an approximation of system cost. In our test case, the relation between computer system prices and CPU prices is perfect. The modern and more expensive CPUs need modern and more expensive motherboards, RAMs and also need more power consumption. So we ignore these costs and simply compare CPU costs. The cost of the total cluster and other systems based on the lowest prices (obtained from retail marketplaces) are shown in Table 4.5.

The "cost-adjusted-performance" of the mentioned cluster running Storm with proposed methods, using the multiplication of the runtime and the cost of the system, is compared with other systems in Figure 4.2. The

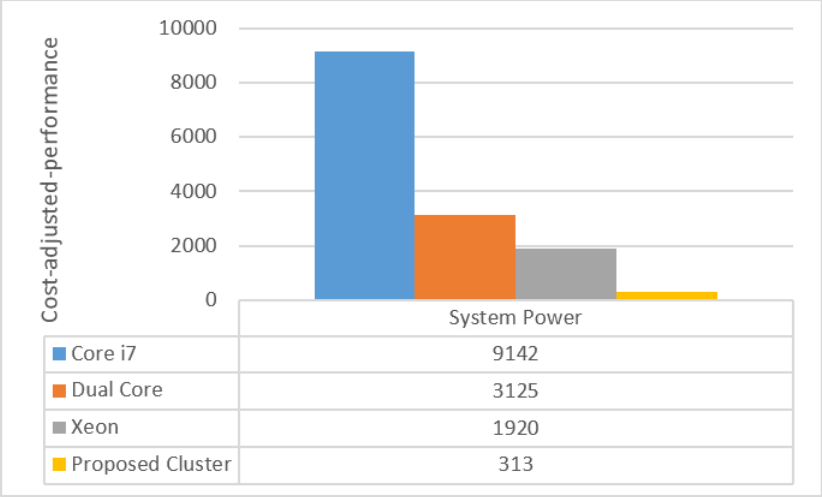

FIG. 4.2. *Comparison of Cost-adjusted-performance of our testbed with proposed method and other types of testbeds*

results show that a commodity cluster with our proposed scheduling method has a better performance-per-cost than other systems.

Although the proposed cluster has more cores, but it demonstrates the lowest cost-adjusted-performance (or highest performance-per-cost) compared to other single systems. The relative importance of equipment cost makes traditional server solutions less appealing for solving our problem because they increase performance, but decrease the performance-per-cost.

5. Conclusions and future works. Clustered commodity computers have already experienced enormous attention and used as a cost-effective platform for distributed and parallel processing of data/compute-intensive applications, compared to more expensive dedicated supercomputers. In Bioinformatics, searching in a database is the most widely used technique and is one of the most common targets to use parallelization in Bioinformatics. In this work, for parallelize sequence similarity methods, we use Storm as a stream processing framework. The Storm framework automatically provides some specifications for parallel computing, such as the scheduling, synchronization, and inter-machine communication. But other specifications, especially partitioning the input data and balancing the workload should be explicitly programmed using a given strategy such as the strategy we have proposed in this paper. Streaming can be a good choice instead of database partitioning when the cost of storage and/or bandwidth is important.

In a perfect world of parallelization, for n processors, the runtime should be n times faster than doing the same work on one processor. However, in a real world, the performance of a parallelization is decreased due to the interaction between processes and synchronization. Moreover, the heterogeneity affects the workload balancing; dividing the jobs equally across machines in a heterogeneous cluster leads to load imbalance. So, efficient scheduling of applications based on the characteristics of computing nodes can yield significant speedups. In this paper, we tried to alleviate these obstacles by running coarse-grain CPU-intensive tasks on loosely-coupled workers with the same workload according to their power.

In the first step to cut down parallelization costs, we diminish the process interactions via reducing the number of tasks. In this work, we try to minimize process interaction by creating coarsely-grained tasks. In the second step after minimizing interactions with proper granularity, communications will be addressed. In a distributed memory fashion, inter-process communications is performed by sending messages over the network. This message-passing paradigm forces programmers to deal with issues such as the location of the data, the method of the communication, and the communication latencies. However, by using a data-parallel model as an explicit programming paradigm for distributed-memory parallel computation that is followed by Storm, the programmer is free from details of message passing. On the other hand, in heterogeneous environments, the speedup suffers from unpredictable behavior that is due to diversities in the performance of CPUs. Getting

a good performance in a heterogeneous distributed system by distributing tasks and dispatching data in such manner that heterogeneity does not affect the speedup is a challenging problem. In this paper, we use an approach that distributes tasks and balances the workload among nodes and provides twofold speedup on a heterogeneous distributed environment.

It should be noted that parallelization strategies presented here would also benefit other commonly used Bioinformatics tools. The alignment-based methods such as BLAST can also be parallelized using our proposed method by streaming the query sequences. Consequently, other algorithms in computational biology may take advantage of performance improvement by this method on cluster based implementations. By running parallel applications on large commodity clusters with tens of nodes, researchers can cheaply perform analyses with acceptable execution time. We are currently studying the potential parallelism of the algorithm, especially in balancing the workload and also the programming paradigms of parallel systems such as MapReduce. In our future work we are going to consider sequence lengths in calculating loads on each node and use a dynamic workload balancer to avoid benchmarking.

REFERENCES

[1] A. Y. Zomaya, *Parallel Computing for Bioinformatics and Computational Biology*, WileySeries on Parallel and Distributed Computing, Wiley-Interscience, 2005.

[2] G. A. Petsko and D. Ringe, *From Sequence to Function: Case Studies in Structural and Functional Genomics*, Primers in biology, New Science Press Sunderland, MA,London, 2004.

[3] A. Matsunaga, M. Tsugawa, and J. Fortes, *Cloudblast: Combining mapreduce and vir-tualization on distributed resources for bioinformatics applications*, in 2008 IEEE FourthInternational Conference on eScience, Dec 2008, pp. 222-229.

[4] A. E. Darling, L. Carey, and W. Feng, *The Design, Implementation, and Evaluation of mpiBLAST*, in ClusterWorld Conference, San Jose, California, June 2003.

[5] H. Lin, X. Ma, P. Chandramohan, A. Geist, and N. Samatova, *Efficient data access for parallel blast*, in 19th IEEE International Parallel and Distributed Processing Symposium, April 2005, pp. 72b72b.

[6] R. Hughey, *Parallel hardware for sequence comparison and alignment*, CABIOS, 12(1996), pp. 473-479.

[7] E. F. d. O. Sandes, G. Miranda, A. C. M. A. d. Melo, X. Martorell, and E. Ayguad, *Cudalign 3.0: Parallel biological sequence comparison in large gpu clusters*, in 2014 14thIEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, May 2014,pp. 160-169.

[8] M. C. Schatz, C. Trapnell, A. L. Delcher, and A. Varshney, *High-throughput sequence alignment using graphics processing units*, BMC Bioinformatics, 8(2007), p. 474.

[9] H. Jiang and N. Ganesan, *CUDAMPF: a multi-tiered parallel framework for ac-celerating protein sequence search in HMMER on cuda-enabled GPU*, BMC Bioinformatics,17 (2016), p. 106.

[10] R. C. Braun, K. T. Pedretti, T. L. Casavant, T. E. Scheetz, C. L. Birkett, and C. A.Roberts, *Parallelization of local blast service on workstation clusters*, Future Gener. Comput. Syst., 17 (2001), pp. 745-754.

[11] A. S. Deshpande, D. S. Richards, and W. R. Pearson, *A platform for biological sequence comparison on parallel computers*, Computer Applications in the Biosciences, 7(1991), pp. 237-247.

[12] T. Rognes, *Paralign: a parallel sequence alignment algorithm for rapid and sensitive database searches*, Nucleic Acids Research, 29 (2001), p. 1647.

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

[14] M. C. Schatz, *Cloudburst: highly sensitive read mapping with mapreduce*, Bioinformatics, 25 (2009), p. 1363.

[15] R. Kienzler, R. Bruggmann, A. Ranganathan, and N. Tatbul, *Large-Scale DNA Sequence Analysis in the Cloud: A Stream-Based Approach*, Springer Berlin, Berlin, Heidelberg, 2012, pp. 467–476.

[16] B. Wolf, P. Kuonen, and T. Dandekar, *Multilevel parallelism in sequence alignment using a streaming approach*, Nesus 2015 workshop, (2015).

[17] S. A. Issa, R. Kienzler, M. El-Kalioby, P. J. Tonellato, D. Wall, R. Bruggmann, and M. Abouelhoda, *Streaming support for data intensive cloud-based sequence analysis*, BioMed Research International, (2013), p. 16.

[18] A. B. Marta and N. Robu, *A study of sequence clustering on proteins primary structure usinga statistical method*, Acta Polytechnica Hungarica, 3 (2006), pp. 1727.

[19] G. Lu, S. Zhang, and X. Fang, *An improved string composition method for sequence comparison*, BMC Bioinformatics, 9 (2008), p. S15.

[20] C. Yu, R. L. He, and S. S.-T. Yau, *Protein sequence comparison basedon k-string dictionary*, Gene, 529 (2013), pp. 250–256.

[21] S. F. Altschul, T. L. Madden, A. A. Schaffer, 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 Res, 25 (1997), pp. 3389–3402.

[22] D. H. Huson and C. Xie, *A poor mans blastx high-throughput metagenomic protein database search using pauda*, Bioinformatics, 30 (2013), p. 38.

[23] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, *Basic local alignment search tool*, Journal of Molecular Biology, 215 (1990), pp. 403–410.

[24] W. R. Pearson and D. J. Lipman, *Improved tools for biological sequence comparison*, in Proceedings of the National Academy of Sciences of the United States of America, 85 (1988), pp. 2444–2448.
[25] C. Yu, M. Deng, and S. S.-T. Yau, *DNA sequence comparison by a novel probabilistic method*, Information Sciences, 181 (2011), pp. 1484–1492.
[26] K. Song, J. Ren, Z. Zhai, X. Liu, M. Deng, and F. Sun, *Alignment-free sequence comparison based on next-generation sequencing reads*, Journal of Computational Biology, 20 (2013),pp. 64–79.
[27] C. A. Leimeister, M. Boden, S. Horwege, S. Lindner, and B. Morgenstern, *Fast alignment-free sequence comparison using spaced-word frequencies*, Bioinformatics, 30 (2014), pp. 1991–1999.
[28] M. N. Davies, A. D. Secker, A. A. Freitas, J. Timmis, E. Clark, and D. R. Flower, *Alignment-independent techniques for protein classification*, Current Proteomics, 5 (2008), pp. 217–223.
[29] T. Z. DeSantis, K. Keller, U. Karaoz, A. V. Alekseyenko, N. N. Singh, E. L. Brodie, Z. Pei, G. L. Andersen, and N. Larsen, *Simrank: Rapid and sensitive general-purpose k-mer search tool*, BMC Ecology, 11 (2011),p. 11.
[30] L. Fu, B. Niu, Z. Zhu, S. Wu, and W. Li, *Cd-hit: accelerated for clustering the next-generation sequencing data*, Bioinformatics, 28 (2012), p. 3150.
[31] A. B. Marta, I. Pitas, and K. Lyroudia, *Statistical Method of Context Evaluation for Biological Sequence Similarity*, Springer US, Boston, MA, 2006, pp. 99–108.
[32] *UniProtKB, Eyeless protein*, www.uniprot.org/uniprot/O96791.
[33] C. D. Manning and H. Schutze, *Foundations of Statistical Natural Language Processing*, MIT Press, Cambridge, MA, USA, 1999.
[34] D. H. V. Uytsely and D. V. Compernolle, *Entropy-based context selection in variable-length n-gram language models*, in IEEE BENELUX Signal Processing Symposium, 1988.
[35] K. Yang and L. Zhang, *Performance comparison of gene family clustering methods with expert curated gene family data set in Arabidopsis thaliana*, Planta, 228 (2008), pp. 439–447
[36] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy, *Storm@twitter*, in Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, SIGMOD14, New York, NY, USA, 2014, ACM, pp. 147–156.
[37] M. Rychly, P. koda, and P. mr, *Scheduling decisions in stream processing on heterogeneous clusters*, in 2014 Eighth International Conference on Complex, Intelligent and Software Intensive Systems, July 2014, pp. 614–619.
[38] J. Kreps, *I Heart Logs: Event Data, Stream Processing, and Data Integration*, OReilly Media,2014.
[39] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, *Zookeeper: Wait-free coordination for internet-scale systems*, in Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference, USENIXATC10, Berkeley, CA, USA, 2010, USENIX Association, pp. 11–11.
[40] A. M. Aji and W. Feng, *Optimizing performance, cost, and sensitivity in pairwise sequence search on a cluster of PlayStations*, in 8th IEEE International Conference on BioInformatics and BioEngineering, BIBE, Athens, 2008.
[41] E. Boutet, D. Lieberherr, M. Tognolli, M. Schneider and A. Bairoch, *UniProtKB/Swiss-Prot*, Methods in Molecular Biology, 406 (2007), pp. 89–112.
[42] *gp120 - ENV Glycoprotein 120*, http://www.bioafrica.net/proteomics/ENV-GP120prot.html.
[43] T. K. Yap, O. Frieder and R. L. Martino *Parallel Computation in Biological Sequence Analysis*, IEEE Transactions on Parallel and Distributed Systems, 9(1998), pp. 283–294.
[44] V. Donaldson, F. Berman and R. Paturi, *Program Speedup in a Heterogeneous Computing Network*, Journal of Parallel and Distributed Computing, 21(1994), pp. 316–322.
[45] G. Lencse and I. Derka, *Testing the Speedup of Parallel Discrete Event Simulation in Heterogeneous Execution Environments*, in Proc. ISC'2013, 11th Annu. Industrial Simulation Conf., Ghent, Belgium, 2013.
[46] R. Agerri, X. Artola, Z. B. G. Rigau and A. Soroa, *Big data for Natural Language Processing: A streaming approach*, Knowledge-Based Systems, 79(2015), pp. 36–42.
[47] R. McCreadie, C. Macdonald, I. Ounis, M. Osborne and S. Petrovic, *Scalable Distributed Event Detection for Twitter*, in IEEE International Conference on Big Data, Silicon Valley, CA, USA, 2013.
[48] P. Karunaratne, S. Karunasekera and A. Harwood, *Distributed stream clustering using micro-clusters on Apache Storm*, Journal of Parallel and Distributed Computing, 2016.
[49] X. Gao, E. Ferrara and J. Qiu, *Parallel clustering of high-dimensional social media data streams*, 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 2015.
[50] W. Inoubli, S. Aridhi, H. Mezni and A. Jung, *Big Data Frameworks: A Comparative Study*, CoRR, abs/1610.09962(2016).
[51] M. A. Lopez, A. Lobato and O. C. M. B. Duarte, *A performance comparison of Open-Source stream processing platforms*, in IEEE Global Communications Conference (Globecom), Washington, USA, 2016.

Edited by: Frédéric Loulergue
Received: August 29, 2016
Accepted: March 6, 2017