Lazer: Distributed Memory-Efficient Assembly of Large-Scale Genomes

Sayan Goswami[*], Arghya Kusum Das[†], Richard Platania[‡], Kisung Lee[§] and Seung-Jong Park[¶]
School of Electrical Engineering & Computer Science, Center for Computation & Technology,
Louisiana State University
[*] *sgoswami@cct.lsu.edu,* [†] *adas@cct.lsu.edu,* [‡] *rplatania@cct.lsu.edu,* [§] *lee@csc.lsu.edu,* [¶] *sjpark@cct.lsu.edu*

***Abstract*—Genome sequencing technology has witnessed tremendous progress in terms of throughput as well as cost per base pair, resulting in an explosion in the size of data. Consequently, typical sequence assembly tools demand a lot of processing power and memory and are unable to assemble big datasets unless run on hundreds of nodes. In this paper, we present a distributed assembler that achieves both scalability and memory efficiency by using partitioned de Bruijn graphs. By enhancing the memory-to-disk swapping and reducing the network communication in the cluster, we can assemble large sequences such as human genomes (452 GB) on just two nodes in 14.5 hours, and also scale up to 128 nodes in 23 minutes. We also assemble a synthetic wheat genome with 1.1 TB of raw reads on 8 nodes in 18.5 hours and on 128 nodes in 1.25 hours.**

***Keywords*-genome assembly; big data;**

I. Introduction

The last decade has witnessed huge leaps in the advancement of sequencing technology both in terms of throughput and cost per base pair. As a consequence, the sizes of the raw sequencing data are on the rise and the computing resources haven't been able to keep up. Typically, in order to assemble a truly big (order of terabytes) dataset, the state-of-the-art assembly tools need either scaled-up nodes with terabytes of RAM or scaled-out clusters with hundreds of nodes. However, a vast majority of the research community do not have access to such extreme computing resources.

Most of the widely used distributed assemblers are designed with the assumption that the users will have enough resources to deal with all data sizes. However, this is not always the scenario, particularly in case of terabyte-scale datasets where these tools run out of memory on small clusters. Furthermore, the trends in sequencing technology shows an increasing demand for processing power and memory in order for the sequences to be assembled. Consequently, researchers will soon be unable to assemble the larger genomes on the HPC clusters available to them. Therefore, to keep in-tune with the decreasing cost of sequencing, there is a need for assembly tools to be more memory efficient.

Figure 1 categorizes representative assemblers based on memory utilization and scalability. On one hand, we have the first generation assemblers which are multithreaded applications that run on a single node but require terabytes of memory to assemble the larger genomes. On the other hand, we have distributed assemblers that still need huge amounts memory, but can be run on a distributed environment if there isn't enough memory on a single node. At the other end of the spectrum, there are a few assemblers that use succinct data structures to assemble large genomes on single nodes, but are not designed to run on distributed systems. In this paper, we present *Lazer* (large-scale genome assembly on ZeroMQ), a distributed assembly tool designed to have a low memory footprint and yet is capable of scaling up to hundreds of nodes.

Experiments on a human genome dataset (452 GB) demonstrate that Lazer's performance is often better than others in terms of execution times, while having better scalability with a much smaller memory requirement. Moreover, Lazer successfully assembles a 1.1 TB synthetic wheat dataset whereas the other assemblers have failed to execute on it.

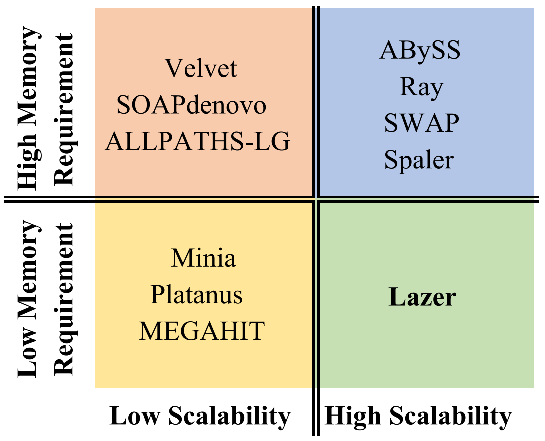


Figure 1. Classification of assemblers wrt. scalability and memory efficiency

The rest of the paper is organized as follows. Section II describes the background and challenges of the de Bruijn graph-based de novo assembly, and Section III discusses the current state of the art in the realm of distributed assemblers. In Sections IV and V, we explain the methodology and implementation details of our proposed assembler. Section VI evaluates our tool on a variety of datasets and cluster configurations followed by a conclusion of our study in Section VII.

II. Background

Genome sequencing is the process of chemically marking (parsing) nucleotides in order to find their exact order in the DNA. Since the state-of-the-art sequencing machines cannot accurately sequence the entire genome in one go, they create multiple clones of the genome, splice them into smaller fragments, and sequence them to produce *short reads*. *De novo assembly* is the process of reconstructing the entire original sequence from short reads, without any backbone sequence (i.e., reference genome) to refer to.

The principal challenge of the assembly process is to find out the overlaps between all pairs of short reads. In order to accomplish this, the short reads are further broken down into smaller overlapping sub-sequences of length k, called *k-mers*. These k-mers are then used to build a de Bruijn graph using the following rules:

1) Each unique k-mer is represented as a vertex in the graph. If the same k-mer is generated more than once from one or more short reads, they must be mapped to the same vertex.
2) An edge exists between two vertices v_i and v_j if their corresponding k-mers, k_i and k_j, have an overlap of $k-1$ characters between them.

The next phase of the assembly is a lossless compression of each vertex chain (i.e., a path of vertices with one in-degree and one out-degree) into a single super-vertex called *contig*. The compression can be performed by traversing the graph starting from the vertices having an unequal in-degree and out-degree (such as forks, joins, etc). Each contig will contain the original sub-sequence(s) from which the participating k-mers were produced. The compressed graph can then be used in a wide variety of contig extension strategies. Figure 2 presents an example of the process described above.

Note that the same k-mer can appear in multiple short reads and there is no way to generate the vertices in the order in which they appear in the graph. This implies that they must be stored in an efficient data structure (such as, an in-memory hash map) in order to facilitate the quick lookup required during traversal. Moreover, for extreme scale parallelism the hash map needs to be globally visible across a distributed computing environment. This global view enables multiple threads across multiple servers to access the entire graph all at once.

III. Related Work

De novo assembly is a widely studied problem so there are many assembly tools to choose from. The computation of older assemblers such as Velvet [1],

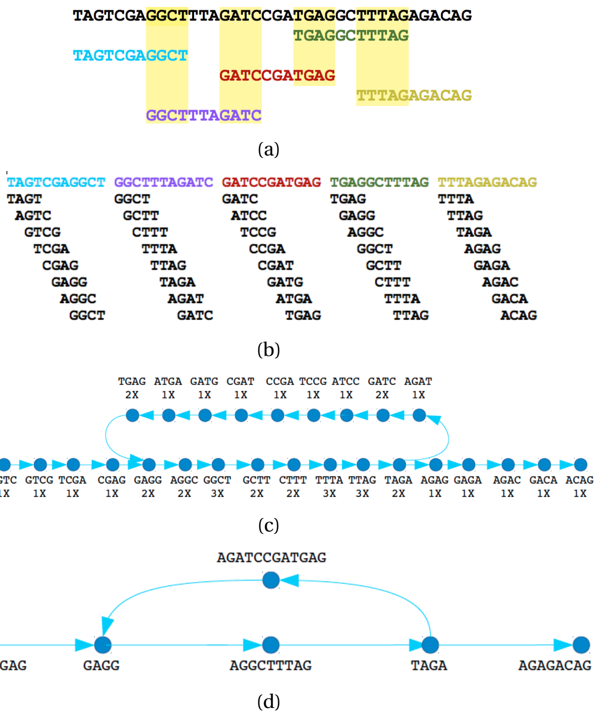

TAGTCGAGGCTTTAGATCCGATGAGGCTTTAGAGACAG
TAGTCGAGGCT
TGAGGCTTTAG
GATCCGATGAG
TTTAGAGACAG
GGCTTTAGATC
(a)
TAGTCGAGGCT GGCTTTAGATC GATCCGATGAG TGAGGCTTTAG TTTAGAGACAG
TAGT AGTC GTCG TCGA CGAG GAGG AGGC GGCT
GGCT GCTT CTTT TTTA TTAG TAGA AGAT GATC
GATC ATCC TCCG CCGA CGAT GATG ATGA TGAG
TGAG GAGG AGGC GGCT GCTT CTTT TTTA TTAG
TTTA TTAG TAGA AGAG GAGA AGAC GACA ACAG
(b)
TGAG 2X ATGA 1X GATG 1X CGAT 1X CCGA 1X TCCG 1X ATCC 1X GATC 2X AGAT 1X
TAGT 1X AGTC 1X GTCG 1X TCGA 1X CGAG 1X GAGG 2X AGGC 2X GGCT 3X GCTT 2X CTTT 2X TTTA 3X TTAG 3X TAGA 2X AGAG 1X GAGA 1X AGAC 1X GACA 1X ACAG 1X
(c)
AGATCCGATGAG
TAGTCGAG
GAGG
AGGCTTTAG
TAGA
AGAGACAG
(d)

Figure 2. The genome assembly pipeline (a) A set of overlapping short reads covering the genome. (b) Generation of *4*-mers from short reads. (c) Creation of de Bruijn graph (the numbers below the k-mers denote their frequency across all reads). Note that k-mer `TAGA` appears in both the second and the fifth short reads. In the second read, the k-mer following `TAGA` is `AGAT`, whereas in the fifth read, the following one is `AGAG`.Therefore, the vertex `TAGA` has two outgoing edges to vertices `AGAT` and `AGAG` (d) Compression of vertex chains into contigs.

ALLPATHS-LG [2], SOAPdenovo [3], etc. are restricted to a single node, and hence these tools cannot address the challenges involved in next-generation sequencing data. Furthermore, these assemblers are severely limited due to inefficient memory management and are incapable of assembling terabytes of data in most practical cases.

To overcome the large memory requirements, Minia [4], Platanus [5], MegaHit [6] make use of several succinct data structures (e.g., Bloom filter, etc.) and encoding schemes and make it possible to assemble large sequences in a single workstation. However, their performance in terms of execution time is still limited by the number of cores available in a single machine. Moreover, given the trends of NGS dataset sizes, it is not clear if these assemblers will be able to accommodate the de Bruijn graphs in the memory of a single machine in the future. Hence, a distributed approach is necessary to alleviate the problems involved in large scale NGS dataset.

Assemblathon2 [7] evaluates various de novo assemblers using a variety of sequence datasets. The only distributed assemblers evaluated in [7] are ABySS [8] and Ray [9], both of which are based on MPI (Message Passing Interface). We have experimentally observed that on large datasets, Ray could not scale up, and ABySS was unable to run at all. PASHA [10] is another distributed assembler, also based on MPI but is severely constrained due to that fact that some of its stages are not distributed. SWAP [11] is better than ABySS, Ray, and PASHA in terms of scalability and execution time, but has a huge memory overhead.

To tackle the scalability problem, assembly tools based on Apache Hadoop or Apache Spark have also been proposed. Contrail [12] was the first de Bruijn graph-based assembler built on Hadoop, followed by CloudBrush [13], which was based on string graphs. However, in case of extremely large datasets their performances deteriorate due the disk-based computation (i.e., huge amount of disk I/O is required) of Hadoop. Some assemblers such as GiGA [14] used a hybrid approach by using Hadoop for building the graph and Apache Giraph for compressing it. Other developments include Spaler [15], which is a de novo assembler built on Spark and GraphX. Although Spaler appears to perform well on datasets up to 100 GB, we could not access its code to evaluate it on the terabyte-scale datasets.

IV. Methodology

Our tool comprises of the first two phases of the assembly pipeline - de Bruijn graph creation and compression. In principle, the graph can be created simply by loading the k-mers into an in-memory hash map, thus implicitly merging unique k-mers to the same vertex. A more memory-efficient option would have been to use a map-reduce-based approach where key-value pairs are partitioned into buckets and then each bucket is reduced by sorting or indexing the keys. However, regardless of the method used to build the graph, it must reside in an in-memory hash table in order to be traversed.

This approach poses a few significant challenges. Firstly, either there must be an enormous amount of memory in a single machine, or the hash map must be distributed. Secondly, in case the hash map is distributed, the lookup time is dominated by the network latency, which is generally in the order of microseconds as opposed to the nanosecond latencies of main memory. This network latency has a significant impact on the total execution time.

We alleviate the problems of scalability and memory efficiency to some extent by partitioning the de Bruijn graph. Our partitioned graph has two advantages:

- It does not require the entire graph to be loaded in memory at once. The vertices within each partition can be indexed, locally compressed, and spilled onto disk until all partitions are processed. Finally, all partitions can be loaded at once and compressed one last time.
- Each partition can be compressed by a thread or a group of threads within a single node in the cluster. This local processing reduces the necessity of synchronization and communication between threads and cluster nodes, thus enhancing the degree of parallelism.

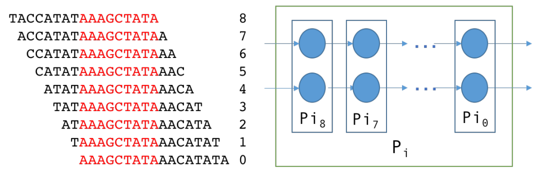
We use minimum substring partitioning (MSP) [16] to distribute the k-mers among the different buckets. In this scheme, a smaller window of size m is slid over each k-mer K_i to generate a set S_i of $k-m+1$ substrings. Since $|\Sigma| = 4$, each substring in S_i can be uniquely mapped to one of 4^m partitions. The partition to which K_i would be mapped is determined by the alphabetically smallest substring in S_i. The motivation behind using this approach instead of an ordinary hash function is that overlapping k-mers (where the prefix of one is the suffix of the another), which will be close to each other in the final de Bruijn graph, will have a better chance of being mapped to the same bucket. In addition, this heuristic allows us to partition the graph while it is being generated from the reads. Therefore, we can reap the obvious benefits of working with smaller sub-graphs without having a separate partitioning phase in the assembly pipeline.

In order to reduce the memory footprint even further, we propose a two-level partitioning scheme. The first level (L1) partition of a k-mer is determined by the minimum substring as explained earlier. The second level (L2) is determined by the position of the minimum substring in the k-mer. Thus, for each L1 partition, there are $k-m+1$ L2 partitions. With this scheme, whenever an L1 partition is locally compressed, at most two consecutive L2 partitions are required to be loaded in memory at once, and the compression proceeds in a lockstep fashion. Figure 3(a) depicts the partitioning scheme described above and 3(b) shows the local compression of L1 partition P_i.

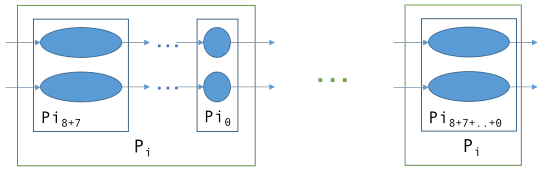
We describe our proposed techniques for the two phases in more detail as follows.

A. De Bruijn graph creation

1) Map phase: In this phase, each mapper thread reads the sequences of nucleotides from files assigned to it. For each sequence of length l, a mapper slides a window of size k to generate $l-k$ intermediate key-value pairs. A key is a byte-encoded k-mer, and the value is a tuple consisting of the two nucleotides at the either end of the window. We call them "intermediate" because multiple short reads may generate the same key whose values must be merged to obtain a final key-value pair. Conceptually, each key represents a vertex of the de Bruijn graph, and its intermediate value represents at most one incoming edge and at most one outgoing edge. K-mers at the leftmost and rightmost end of the sequence are padded with special characters in the value tuple to represent no incoming and outgoing edges respectively. Each key-value



(a)



(b)

Figure 3. The two level partitioning scheme and local compression. (a) In this scenario, the k-mer size is 17 and the substring size is 9. All the overlapping k-mers here have the same minimum substring (AAAGCTATA) and are thus mapped to the same L1 partition. However in each of the k-mers, the substring occurs at different positions and are hence mapped to different L2 partitions. (b) The compression begins with the vertices in L2 partition P_{i_8} as seeds. The vertices in P_{i_7} are loaded in a hash map, merged with the seeds and removed from map. This process continues until the vertices in L2 partitions P_{i_7} through P_{i_0} are merged with those in P_{i_8}

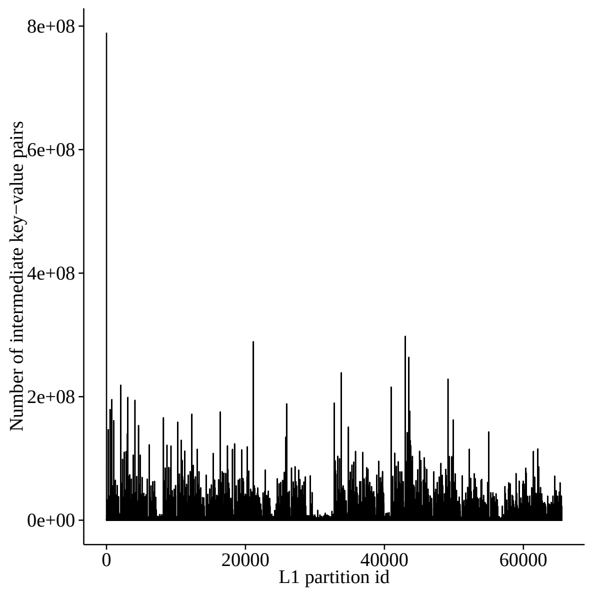


Figure 4. Imbalance of distribution in intermediate data among different partitions

pair is then placed into the appropriate partition based on the minimum substring of the key.

2) Load balancing: Unlike conventional hash functions, the minimum substring partitioning scheme is locality sensitive, and its main purpose is to maximize the probability of a collision for similar strings. Consequently, the distribution of k-mers across different L1 and L2 partitions is severely skewed. Moreover, we have observed that on a number of datasets, almost half of the partitions remain empty. The distribution of intermediate key-value pairs among the non-empty L1 partitions in the Yoruban male dataset can be seen in figure 4

Clearly, there are a few partitions such as partition id 0, that are much larger than the average. In the reduce phase where each node handles a subset of the partitions, if the distribution is not uniform, one of the peer nodes of the cluster might get more than its share of data consequently dominating the total execution time, or worse, running out of memory. To tackle this problem, after all reads are processed, the total number of intermediate key-value pairs for each partition is collected from all the nodes, and the partitions are stored in a max-heap based on their sizes. They are then distributed across the nodes using the Longest Processing Time (LPT) scheme.

3) Reduce phase: In this phase, each reducer thread picks up one of the L1 partitions assigned to it and starts reducing the values corresponding to each unique key in the list. This reduce phase has two objectives:

1) The same k-mer generated from different sequences can have different nucleotides at its ends. These nucleotides which form the incoming and outgoing edges of the vertex represented by the k-mer need to be merged in the final de Bruijn graph. Since the maximum in-degree and out-degree of a vertex is 4, the edges can be encoded in a single byte.
2) The number of times a k-mer (regardless of the outgoing edges) occurs in the entire dataset needs to be tracked. This frequency information is used in subsequent stages to recognize erroneous structures in the graph caused by sequencing errors.

After this phase, all subgraphs within each partition are locally compressed.

B. Graph compression phase

Following reduction and local compression, the vertices are distributed among the nodes for further extension. The vertices that have exactly one incoming edge and at most one outgoing edge are loaded in a distributed hash map. All other vertices are appended to a distributed list and act as seeds for the extension. After the graph is loaded, the workers traverse the linear chains, starting from the vertices in the list. Each worker looks up the hash map in order to get the next vertex in the chain it is working on.

We guarantee that a single path cannot be traversed by more than one thread making our algorithm lock-free and enhancing scalability. Clearly, there can be an imbalance in load distribution between different threads depending on the lengths of the paths. However, we have observed that with real datasets, the imbalance is not severe enough to warrant the overhead of synchronization.

V. Implementation

Our assembler is built on top of ZeroMQ (http://zeromq.org/), which is an embeddable framework for concurrency and communication. It provides a threading library based on Hewitt's *Actor Model* [17] of concurrent computation. The semantics of this model states that an actor is an independent "universal primitive" for concurrency that can only modify its own state and exchange atomic messages with other actors. These concepts differ from the idea of a global state shared by multiple threads, hence actors don't require locks, semaphores or critical sections.

ZeroMQ also provides an intelligent transport layer for low-latency communication and is best suited for small packets. Moreover, it provides uniform message passing semantics between actors (threads), processes, and boxes (cluster nodes). Accordingly, a pair of ZeroMQ sockets can talk to each other over *inproc*, *ipc*, or *tcp* protocols.

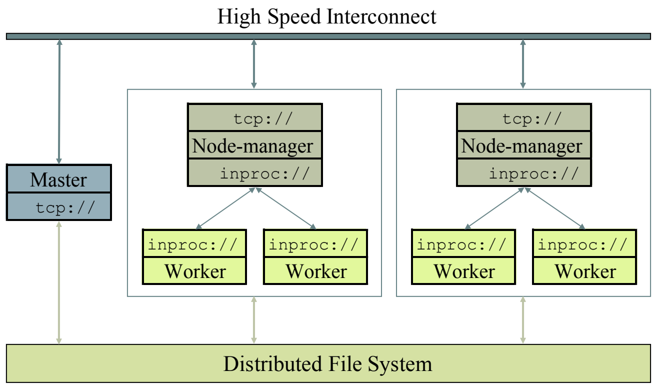


Figure 5. System overview of Lazer

Figure 5 shows the high level system overview of Lazer. It consists of a master process, which is responsible for the distribution of tasks, synchronization between nodes and load-balancing. Moreover, each node runs a slave process consisting of a Node-manager actor which spawns and manages worker actors. Since a manager and its workers reside in the same address space, they can connect to each other via *inproc* protocol. The node-managers connect to the master via tcp. All processes have access to a distributed file system and can send/receive messages to/from each other via a high speed network.

This architecture forms the basis on which all the phases are implemented. The subsections that follow, describe the implementations and actor interactions during each phase in more details.

A. Map Phase of Graph Creation

Figure 6 shows the different actors at play and their communication patterns during the map phase. The

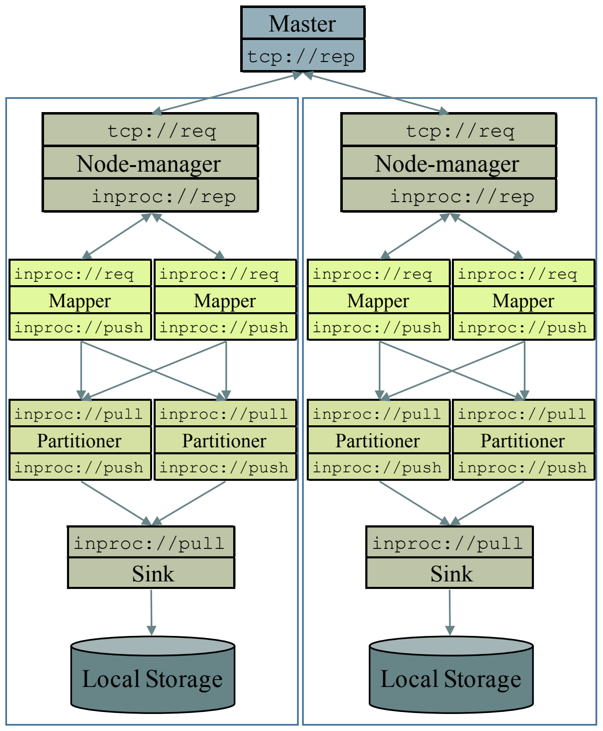


Figure 6. Actor interactions during Map-phase

node-manager spawns three types of workers: mappers, partitioners and sinks. All these actors have the same parent process and can communicate via `inproc`. Mappers ask node-managers for input batches and the request is relayed to the master. On receiving a batch, mapper threads parse the reads into (k+2)-mers (i.e. the k-mer and the nucleotides at its ends). Subsequently, each (k+2)-mer is mapped to one of the available partitions and sent to the partitioner actor responsible for it. The process is described in Algorithm 1.

Algorithm 1 Mapper actor

```
1: Actor MAPPER
2:     while $more\_batches$ do
3:         ZMQ_SEND($manager, GET\_BATCH$)
4:         $read\_batch \leftarrow$ ZMQ_RECV($manager$)
5:         for read $r$ in $read\_batch$ do
6:             $tuple\_array\langle kmer, in, out\rangle T \leftarrow$ PARSE_READ($r$)
7:             for tuple $k2$ in $T$ do
8:                 $\langle P_{L_1}, P_{L_2}\rangle \leftarrow$ MIN_SUBSTR_HASH($k2.kmer$)
9:                 $id \leftarrow P_{L_1} \bmod |partitioners|$
10:                ZMQ_SEND($partitioners[id], k2, P_{L_1}, P_{L_2}$)
11:            end for
12:        end for
13:    end while
14: end Actor
```

Partitions are maintained as in-memory lists and each partitioner is assigned a subset of those lists. The amount of intermediate data generated at each node is often much more than the memory available. Hence, a subset of the data must be spilled onto disks. Here, we answer the following questions pertaining to data spillage:

1) How do we decide what to spill?
2) How is the spilled data stored on disk?

Assuming there are p partitions, each node begins with p empty lists. Partitioners receive intermediate key-value pairs from mappers and append them to appropriate lists, according to their minimum substrings. Data is appended to lists in fixed-size chunks, and the total number of chunks across all partitions in a node is tracked. Whenever a new chunk is created, the total chunk count is compared with a threshold, and if it is larger than the threshold, the chunk is spilled to disk. By configuring the threshold, one can limit the memory-per-node that can be used to store the intermediate data. If there is enough memory, intermediate data isn't spilled onto disk at all, thus removing the I/O bottleneck during reduction.

Since the data must be spilled in a way that allows entire partitions to be read from the disk at once, storing key-value pairs in flat files is not efficient. Instead, we use a disk-based NoSQL database to index the spilled data. If each partition maintains a current chunk count, the unique keys for the spilled blocks can be generated from the partition id and the chunk id. During reduction, all the n_i blocks spilled in a partition P_i can be retrieved by making GET calls to the database using keys ranging from $i.0$ to $i.(n_i - 1)$.

We use RocksDB (http://rocksdb.org/) as the underlying NoSQL database. It is an embeddable persistent key-value store based on log-structured merge trees and can provide a partial ordering of keys based on fixed-length prefixes. Since we dont care how the data is ordered on disk as long as they are grouped by partitions, we can assign the first $log(p)$ bits of keys as the prefix. During reduction of a partition P_i, we can iterate over the key-space with prefix i. Under this scheme, insertion and retrieval perform better than those in databases with total ordering of keys.

Furthermore, RocksDB provides a number of useful features such as multithreaded compactions, thread-safe reads, bloom-filters for caching, and several other parameters that can be configured based on the underlying storage system. All of our experiments were run on spinning disks, but we believe the reduce phase will get a significant performance boost if the database is hosted on SSDs. Although RocksDB supports multi-threaded reads, all writes to the database must be serialized. Hence write access to the database is limited to a single actor (sink) that handles all PUT requests from the partitioners, as shown in Figure 6 and described in Algorithm 2.

Algorithm 2 Partitioner actor

```
 1: Actor PARTITIONER
 2:     $cache \leftarrow lists[|L1|][|L2|]$
 3:     while more_kmers do
 4:         $(k2, P_{L_1}, P_{L_2}) \leftarrow$ ZMQ_RECV($mappers$)
 5:         LIST_APPEND($cache[P_{L_1}][P_{L_2}], k2$)
 6:         if cache_overflow then
 7:             ZMQ_SEND($sink, cache[P_{L_1}][P_{L_2}]$)
 8:         end if
 9:     end while
10: end Actor
```

B. Reduce Phase of Graph Creation

Note that in the map phase, there is no communication between nodes. In other words, if there are p partitions, each of the nodes will initially contain p buckets to hold the intermediate data. This is done in order to achieve a balanced distribution of partitions in the reduce phase. Before the the intermediate data is merged, they must be shuffled around so that for a partition P_i all intermediate data generated within P_i in all the nodes must be moved to the the node responsible for reducing P_i. Figure 7 shows how the reducers fetch intermediate data from the NoSQL database servers located at the different nodes.

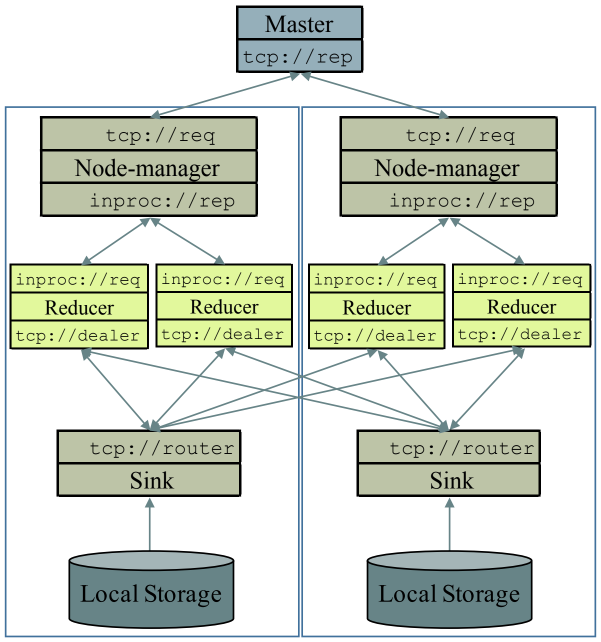


Figure 7. Actor interactions during Reduce-phase

Shuffle: If the intermediate data is fully contained in memory, the performance of the shuffle phase becomes almost entirely dependent on the network throughput. We have experimentally observed that with the high-speed networks available in most HPC clusters, the memory to memory data transfer does not create any bottlenecks in the process (i.e., the processor utilization remains fairly high). However, this changes when the intermediate data is spilled onto disk. In such scenarios, shuffle performance is dictated by the disk read throughput, which is typically much less than the network throughput. Moreover, if there is a single disk per node, as is the case in most HPC clusters, multiple threads reading from the disk increases contention and adversely affects the performance.

The impact due to disk I/O bottleneck is offset to some extent by using a block pre-fetching scheme on the database server side and a scatter-gather approach on the client side. Both of these features are enabled by ZeroMQ's high-speed asynchronous messaging engine. On the client side, each reducer starts with a set containing all the nodes and scatters requests for a chunk of intermediate data. This scheme serves two purposes:

1) Since each reducer sends requests to the servers running on all nodes, the database servers share the load uniformly. Moreover, the fact that each reducer only asks for a small chunk prevents starvation of other reducers, albeit at the cost of a higher number of messages in the network.
2) Since messages are transmitted and received asynchronously by ZeroMQ I/O threads, reducers can immediately start working on the chunks as soon as they arrive. Therefore, the latency is reduced to the time between the transmission of the last request to the arrival for the first response.

On the database side, when a server receives a request for a chunk of partition P_i from a reducer, it checks if any blocks in P_i reside in memory. Otherwise it fetches one from disk and responds with it. As the response is being sent, the server checks if the next block is in memory. If it is not, it pre-fetches one from disk. The pseudo-code for the reducer actor is shown in Algorithm 3

An interesting observation worth mentioning at this point is that on some datasets, a single partition can be so much larger than the others that the reducer handling it might keep on running long after the others are finished. This occurs more often when the program is run on a huge number of cores, where each reducer deals with a very small number number of partitions, most of which have a very small amount of intermediate data, and a select few are massive.

This effect can be seen in figure 10, where partition id 0 clearly stands out from the rest. This is an inherent disadvantage of using the minimum substring partitioning scheme, something that comes as a compromise for the reduced memory footprint. Although the effect of these mega-partitions can be somewhat offset when run in a small cluster, they completely destroy the scalability as the number of nodes is increased.

Algorithm 3 Reducer actor

```
1: Actor REDUCER($P_{L_1}$)
2:     $list \leftarrow NULL$
3:     for $P_{L_2} \leftarrow |L2| to 1$ do
4:         $hashmap \leftarrow$ MERGE($P_{L_1}, P_{L_2}$)
5:         $list \leftarrow$ EXTEND($list, hashmap$)
6:     end for
7: end Actor

8: procedure MERGE($P_{L_1}, P_{L_2}$)
9:     while nodeset_not_empty do
10:        ZMQ_PUBLISH($nodeset, P_{L_1}, P_{L_2}$)
11:        for $i = 1$ to $|nodeset|$ do
12:            $(n, chunk) \leftarrow$ ZMQ_RECV($nodeset$)
13:            if chunk_is_empty then
14:                REMOVEFROM($nodeset, n$)
15:            else
16:                add/merge chunk to hashmap
17:            end if
18:        end for
19:    end while
20: end procedure

21: procedure EXTEND($list, hashmap$)
22:    for all vertex $v$ in list do
23:        if OUT($v$) $= 1$ and $v.next \in hashmap$ then
24:            APPEND($v, hashmap[v.next]$)
25:            ERASE($hashmap, v.next$)
26:        end if
27:    end for
28:    Move all vertices from hashmap to list
29: end procedure
```

Its apparent that the size of the substring chosen for MSP will have a significant impact of the skewness of the distribution. Thus with some trial-and-error, the substring size can be chosen in a way such that the problem can be alleviated. However, these optimum sizes may widely vary across different datasets. Therefore, the substring size must be fixed by the end user for each new experiment, which is a very undesirable side effect.

To relieve the user of this burden, we only process the smaller partitions with a single thread. All other partitions which are larger than a preset threshold are reduced and locally compressed by all available threads within one of the nodes. After running experiments on a number of datasets we have observed 5×10^8 to be a good threshold value (it can be modified by the user if required). The multithreaded algorithm is very similar to the one described in Algorithm 3, barring a few modifications for synchronization, and is therefore omitted from this paper.

C. Graph Compression Phase

During this phase, all partitions of the graph must be loaded into memory for further compression and be globally visible to all nodes in the cluster. Moreover, traversal of the graph demands an $\mathcal{O}(1)$ lookup of vertices, which implies that they must be stored in a distributed in-memory hash-map. Since, at this stage there are no guarantees that adjacent vertices will reside in the same address space, the traversal may require network communication for every other vertex in the chain which imposes strict latency constraints on the hash-map

We observed that the off-the-shelf distributed hash-tables fail to satisfy the low-latency constraints required for graph traversal, and the low memory footprint we are trying to achieve. Moreover, these general-purpose tools come with features such as load-distribution and fault tolerance, which add extra complexity not necessary for our workload.

To get around these issues, we mimic a distributed hash-map, by using an array of independent instances of hash-maps running on each node, and a pre-defined hash-function f_h known to all clients. The servers are oblivious to one another, and the clients maintain connections to all servers in the cluster. For any key-value pair $\langle K, V \rangle$, the server id at which the pair will reside is determined by $f_h(K) \mod n$, where n is the number of server instances. This scheme provides a one-hop access to the hash-map since any client would be able to *GET* a value residing at any node, as long as the key is known to it.

We have chosen Sparsehash (code.google.com/p/sparsehash) as the underlying data structure for the distributed hash-map, since it has the lowest memory footprint. Moreover, due to ZeroMQ's low-latency message transmission and one-hop accesses to each server instance, we have observed very good performance, even with random accesses on billions of keys.

To increase the throughput even further, we employ an asynchronous pipeline of GET requests, where workers simultaneously work on a batch of b paths (simultaneously extend b seeds). At each step, a worker requests b values from the hash-map corresponding to the outgoing edges of paths in the current batch. Once again, since message transmissions are asynchronous, a request returns immediately with a placeholder for a future result. The rationale here is that if the batch size is large enough, by the time the last request is sent, the response for the first one would have arrived, thus hiding the network latency. When a path reaches its end, it is removed from the batch and replaced with a new seed.

VI. Evaluation

A. Overview of the Datasets

To analyze the performance characteristics of Lazer, we use two different datasets: 1) a synthetic wheat (Triticum aestivum) genome dataset of size 1.1TB from Joint Genome Institute [1] and 2) a Yoruban male genome dataset of size 452GB from National Center for Biotechnology Information [2]. During the process of assembly, the first dataset (i.e., the wheat genome) produces more than 4 TB of temporary data, whereas the second one (i.e., the human genome) produces almost 750GB.

B. Overview of the Experimental Testbed

To evaluate the performance of Lazer in a distributed environment, we use QueenBeeII[3] supercomputing cluster located at LSU. Each compute node of QueenBeeII has two 10-core 2.8 GHz E5-2680v2 Xeon processors, 64GB DRAM and 500GB hard disk drive. Although QueenBeeII has a total of 480 nodes with this configuration, only 128 of them are available for running a single job. All the nodes are connected by a 56Gb/sec InfiniBand (FDR) interconnect with 2:1 blocking ratio.

C. Demonstrating the Scalability of Lazer

1) Assembly of Wheat Genome (1.1TB): To demonstrate the scalability of Lazer, we first assemble the relatively larger wheat genome (1.1TB) with different number of nodes in QueenBeeII cluster. Figure 8 shows the execution time of different phases of Lazer while assembling the wheat genome.

Note that the map phase is the most scalable of all phases which is not surprising since this phase is embarrassingly parallel and there is no network communication involved in it. It can also be observed that the compression phase makes up a small fraction of the total execution time and is also scalable even though it involves a distributed key-value store which in-turn is dependent on network latency. We speculate that the asynchronous pipelined accesses to the hash-map and the lock-less traversal algorithm have a significant impact on the scalability of compression.

On the other hand, the reduce phase appears to scale worse than the others due to the large amount of spilled intermediate data that must be shuffled during reduction. Since each node has a single disk, it presents a bottleneck while reading data and prevents a high CPU utilization.

It is worth mentioning here that we could not assemble this dataset using any other assembler even when using the maximum amount resources (i.e., 128 nodes of QueenBeeII cluster) available to us.

[1] https://gold.jgi.doe.gov/projects?id=Gp0039864
[2] http://www.ncbi.nlm.nih.gov/sra/?term=SRA000271
[3] http://www.hpc.lsu.edu/resources/hpc/system.php?system=QB2

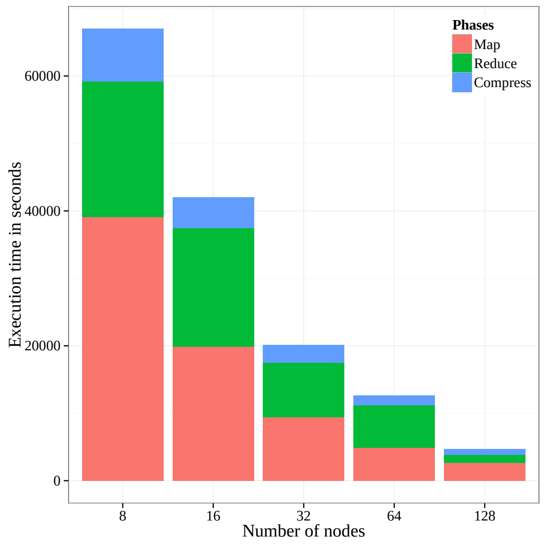


Figure 8. Execution times of Lazer on the synthetic wheat W7984 dataset (1.1 TB): Neither Ray nor SWAP could assemble the 1.1 TB wheat dataset with 128 nodes due to insufficient memory or wall-clock timeout in the cluster

2) Assembly of human genome (452GB): To compare Lazer's overall execution time and scalability with other assemblers, we used a relatively smaller human genome data set (452GB). Figure 9 compares the execution time of Lazer with two other genome assemblers, Ray and Swap while assembling the 452GB human genome. It can be easily observed that Lazer scales almost linearly with increase in number of nodes.

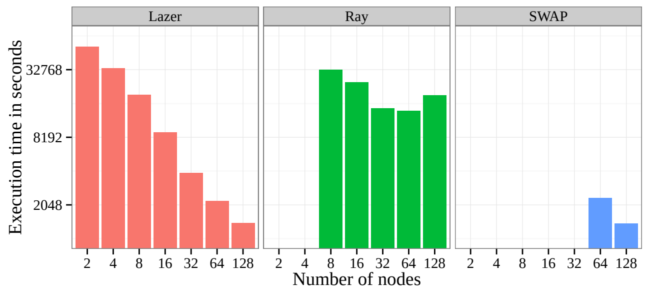


Figure 9. Execution times of Lazer, Swap, and Ray on the Yoruban male dataset (452 GB): Ray cannot complete assembly on 4 nodes; SWAP runs out of memory with 32 nodes. Both axes are in log scale

We were unable to run SWAP on 32 nodes and Ray on 4 nodes due to out-of-memory errors. We observed that Ray's scalability degrades after 32 nodes, and the execution time increases going from 64 to 128 nodes. Lazer, on the other hand, can assemble the dataset on just 2 nodes, and yet shows a near-linear scalability when running on different configurations.

D. Demonstrating Memory-efficiency of Lazer

The memory efficiency of Lazer is evident from the fact that we could not run the 1.1TB wheat genome using the other assemblers even using the maximum amount of resources available to us i.e., 128 nodes of QueenBeeII providing 8TB ($64GB \times 128$ nodes) of aggregated DRAM. On the other hand, Lazer was able to assemble it using only 8 nodes i.e., 512GB ($64GB \times 8$ nodes) of memory.

Figure 9 also demonstrates the memory efficiency of Lazer compared to Ray and Swap [4]. It is apparent that Lazer successfully assembles the human genome using only 2-nodes i.e., 128GB ($64GB \times 2$ nodes), whereas Ray fails to complete using 4-nodes (i.e., $64GB \times 4$ nodes = $256GB$ of DRAM) or higher. SWAP, although better than Ray in terms of execution time, was found to be the least memory efficient in our evaluation. It requires at least 16 times more aggregate memory than Lazer since it runs out on memory with 32 nodes (i.e. $64GB \times 32$ nodes $= 2TB$ of DRAM).

The reason behind the memory efficiency of Lazer is the decrease in the final graph size due to locally compressible graph partitions. As mentioned earlier, there is a significant imbalance in the size of the partitions can easily overwhelm the hardware resources if they are not carefully distributed among the cluster nodes. In Figure 10, we show the distribution of intermediate key-value pairs among peers after the intermediate data is shuffled. It can be observed that the larger partitions are scattered across different nodes in a fairly uniform manner so that no single node is assigned more than one. The low memory requirement can be very significant to bioinformatics researchers who may not have access to large scale compute clusters with hundreds of nodes. In such scenarios, Lazer can provide real time solution for assembling huge NGS data set.

VII. Conclusion

In this paper, we introduced Lazer, a distributed assembly framework that utilizes a smart partitioning scheme for de Bruijn graphs to achieve both scalability and memory efficiency. We have built Lazer on top of the ZeroMQ's actor and intelligent messaging framework to achieve low-latency network communication as well as inter-thread message passing. Experimental results on terabyte-scale datasets show that our framework significantly reduces the memory footprint while ensuring fast assembly. As future work, we plan to add support for reading input data from Hadoop distributed filesystem (HDFS) so that Lazer can be more accommodating to commodity clusters. We also plan to include error removal and scaffolding phases to create a complete assembly pipeline.

[4]In order to have a fair comparison, we only evaluate the contig generation phases of SWAP and Ray with single end reads.

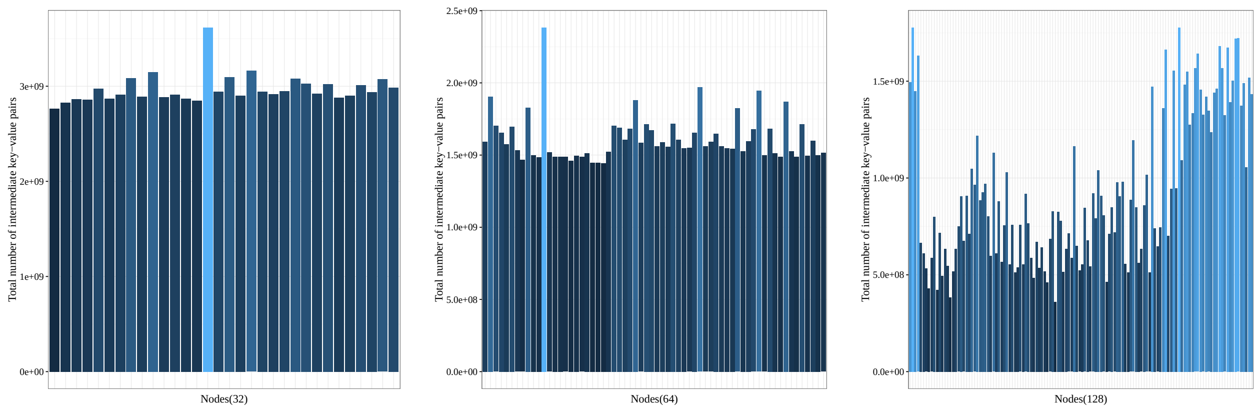


Figure 10. Distribution of intermediate key-value pairs across different nodes on 32, 64 and 128-node cluster configurations

ACKNOWLEDGMENT

This work was supported in part by National Science Foundation under grants MRI-1338051 and CC-NIE-1341008, LA Board of Regents under grant LEQSF(2016-19)-RD-A-08, and IBM Faculty Award.

REFERENCES

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

[2] I. MacCallum, D. Przybylski, S. Gnerre, J. Burton, I. Shlyakhter, A. Gnirke, J. Malek, K. McKernan, S. Ranade, T. P. Shea *et al.*, "Allpaths 2: small genomes assembled accurately and with high continuity from short paired reads," *Genome biology*, vol. 10, no. 10, p. 1, 2009.

[3] Y. Xie, G. Wu, J. Tang, R. Luo, J. Patterson, S. Liu, W. Huang, G. He, S. Gu, S. Li *et al.*, "Soapdenovo-trans: de novo transcriptome assembly with short rna-seq reads," *Bioinformatics*, vol. 30, no. 12, pp. 1660–1666, 2014.

[4] R. Chikhi and G. Rizk, "Space-efficient and exact de bruijn graph representation based on a bloom filter." in *WABI*, ser. Lecture Notes in Computer Science, vol. 7534. Springer, 2012, pp. 236–248.

[5] R. Kajitani, K. Toshimoto, H. Noguchi, A. Toyoda, Y. Ogura, M. Okuno, M. Yabana, M. Harada, E. Nagayasu, H. Maruyama *et al.*, "Efficient de novo assembly of highly heterozygous genomes from whole-genome shotgun short reads," *Genome research*, vol. 24, no. 8, pp. 1384–1395, 2014.

[6] D. Li, C.-M. Liu, R. Luo, K. Sadakane, and T.-W. Lam, "Megahit: an ultra-fast single-node solution for large and complex metagenomics assembly via succinct de bruijn graph," *Bioinformatics*, p. btv033, 2015.

[7] K. R. Bradnam, J. N. Fass, A. Alexandrov, P. Baranay, M. Bechner, I. Birol, S. Boisvert, J. A. Chapman, G. Chapuis, R. Chikhi *et al.*, "Assemblathon 2: evaluating de novo methods of genome assembly in three vertebrate species," *GigaScience*, vol. 2, no. 1, p. 1, 2013.

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

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

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

[11] J. Meng, B. Wang, Y. Wei, S. Feng, and P. Balaji, "Swap-assembler: scalable and efficient genome assembly towards thousands of cores," *BMC bioinformatics*, vol. 15, no. Suppl 9, p. S2, 2014.

[12] M. Schatz, D. Sommer, D. Kelley, and M. Pop, "Contrail: Assembly of large genomes using cloud computing," in *CSHL Biology of Genomes Conference*, 2010.

[13] Y.-J. Chang, C.-C. Chen, J.-M. Ho, and C.-L. Chen, "De novo assembly of high-throughput sequencing data with cloud computing and new operations on string graphs," in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*. IEEE, 2012, pp. 155–161.

[14] P. K. Koppa, A. K. Das, S. Goswami, R. Platania, and S.-J. Park, "Giga: Giraph-based genome assembler for gigabase scale genomes," in *Proceedings of the 8th International Conference on Bioinformatics and Computational Biology (BICOB 2016)*. ISCA, 2016, pp. 55–63.

[15] A. Abu-Doleh and Ü. V. Çatalyürek, "Spaler: Spark and graphx based de novo genome assembler," in *Big Data (Big Data), 2015 IEEE International Conference on*. IEEE, 2015, pp. 1013–1018.

[16] Y. Li, P. Kamousi, F. Han, S. Yang, X. Yan, and S. Suri, "Memory efficient minimum substring partitioning," in *Proceedings of the VLDB Endowment*, vol. 6, no. 3. VLDB Endowment, 2013, pp. 169–180.

[17] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular actor formalism for artificial intelligence," in *Proceedings of the 3rd international joint conference on Artificial intelligence*. Morgan Kaufmann Publishers Inc., 1973, pp. 235–245.

[18] R. Platania, S. Shams, C.-H. Chiu, N. Kim, J. Kim, and S.-J. Park, "Hadoop-based replica exchange over heterogeneous distributed cyberinfrastructures," *Concurrency and Computation: Practice and Experience*, 2016.

[19] A. K. Das, S.-J. Park, J. Hong, and W. Chang, "Evaluating different distributed-cyber-infrastructure for data and compute intensive scientific application," in *Big Data (Big*

Data), 2015 IEEE International Conference on. IEEE, 2015, pp. 134–143.

[20] U. C. Satish, P. Kondikoppa, S.-J. Park, M. Patil, and R. Shah, "Mapreduce based parallel suffix tree construction for human genome," in *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS).* IEEE, 2014, pp. 664–670.