Lazer: A Memory-Efficient Framework for Large-Scale Genome Assembly

Sayan Goswami, Arghya Kusum Das, Richard Platania, Kisung Lee, Seung-Jong Park
Division of Computer Science & Engineering
Louisiana State University

ABSTRACT

Genome sequencing technology has witnessed tremendous progress both in terms of throughput as well as the cost per base pair. However, when it comes to sequence assembly, there still exists a dilemma in the state-of-the-art technology. On one hand, we have a number of distributed assemblers that can utilize several nodes but require massive amounts of memory. On the other hand, there are a few assemblers that can assemble mammalian genomes on a single node but cannot scale up. In this paper, we present a distributed assembler that is both scalable and memory efficient. Using partitioned De Bruijn graphs we enhance memory-to-disk swapping and reduce network communication in the cluster. Experimental results show that our framework can assemble a human genome dataset (452GB) in 14.5 hours using two nodes and 23 minutes using 128 nodes.

1. INTRODUCTION

Genome sequencing is the process of chemically parsing nucleotides to find their exact order in the DNA. Since current sequencing machines cannot sequence the entire genome at one go, they replicate the genome many times over, break it into shorter segments (called short-reads), and then sequence the shorter segments at random. Therefore, there is a high probability that segments from the different replicas will overlap with each other.

De novo assembly is the process of reconstructing the entire genome without referring to any backbone sequences (i.e., only by finding the overlaps). A popular De novo assembly approach is to build a De Bruijn graph from the short-reads and then perform an Eulerian path traversal on it. The advantages of using the De Bruijn graph for genome assembly are twofold: (1) its time complexity is $\mathcal{O}(N)$ where N is the total size of reads, and (2) its space complexity is $\mathcal{O}(G)$ where G is the size of the genome.

The last decade has witnessed huge leaps in the advancement of sequencing technology brought by a class of new sequencers called Next Generation Sequencers (NGS). NGS can sequence very large genomes at a very high throughput and at incredibly low costs. For instance, Illumina HiSeq produces 3×10^9 reads per run at \$0.15 per million bases. Moreover, the coverage of sequencing has also increased by an order of magnitude, thus increasing the total size of reads. Consequently, genome assembly has become more and more demanding in terms of both processing power and memory.

The trends in sequencing technology dictate that the next-generation assemblers should be not only highly scalable but also memory efficient. However, most of the existing assemblers strive to achieve only either one of these two essential requirements. For instance, Minia [3], a standalone assembler that uses bloom filters instead of hash maps, to store the De Bruijn graph, is memory efficient, but cannot be used in a distributed environment. On the other hand, two of the most widely used assemblers, ABySS[7] and Ray[2] are distributed but do not scale well on a large number of nodes. SWAP[6] is another assembler, which is fast and highly scalable, but comes with an enormous memory requirement. In this paper, we present *Lazer* (large-scale genome assembly on ZeroMQ), a scalable distributed assembly framework designed to have a low memory footprint. Experiments on large datasets demonstrate that Lazer's performance is comparable to SWAP while having a much smaller peak memory requirement.

2. BACKGROUND

The principal challenge of the De novo assembly process is to find out the overlaps between all pairs of short-reads. To discover the overlaps, reads are further broken down into smaller overlapping sub-sequences of length k (called k-mers) which 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 De Bruijn Graph. If the same k-mer is generated more than once from one or more short-reads, all replicas are 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.

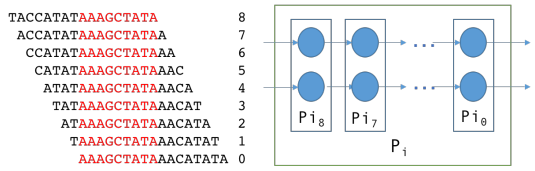
The next phase of the assembly is a lossless compression of all vertex chains (i.e., vertices with one in-degree and one out-degree) into super-vertices called contigs. After the graph is compressed, a variety of contig extension strategies can be applied on it.

3. METHODOLOGY

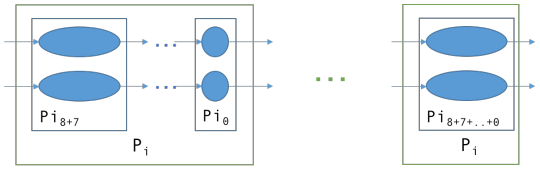
To improve the scalability and memory efficiency of our assembly framework, we partition the De Bruijn graph. By partitioning the graph into smaller units, we do not need to load the entire graph in memory at once during compression. In addition, each partition can be compressed by a thread or a group of threads within a single node in the cluster, enhancing the degree of parallelism. In particular, we use minimum substring partitioning [5] to distribute the k-mers among multiple partitions. In this scheme, a small

window of size m is moved over each k-mer to generate a set S of $k - m + 1$ substrings. The partition to which a k-mer would be mapped is determined by the alphabetically smallest substring in the k-mer. In this scheme, it is more likely that overlapping k-mers (i.e., k-mers in which the prefix of one is the suffix of the another) will be mapped to the same partition, and hence each partition can be locally compressed

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, and the second level (L2) is determined by its position in the k-mer. With this scheme, whenever an L1 partition is locally compressed, at most two of its L2 partitions are required to be loaded in memory at once. Figure 1(a) depicts the partitioning scheme described above.



(a) Two-level partitioning



(b) Local compression

Figure 1: Graph Partitioning and Compressing

The assembly pipeline consists of three phases (map, reduce, and global compression). In the map phase, mapper threads read parts of the input from a distributed file system, parse them into k-mers, and map these k-mers into local partitions. At the beginning of the reduce phase, the total number of intermediate key-value pairs generated within all the L1 partitions across all the nodes is gathered, and the partitions are distributed among the reducers according to the Longest Processing Time scheme. Each reducer in charge of an L1 partition P_i collects all intermediate data in each of the L2 partitions p_j in P_i from all the other nodes and merges the key-value pairs. Subsequently, these L2 partitions are locally compressed as shown in figure 1(b), and loaded into a distributed hash-map. After all the L1 partitions are reduced and compressed locally, a final round of global compression is performed on the graph to produce the contigs.

4. IMPLEMENTATION

Our assembler is built on top of ZeroMQ[1], which is an embeddable framework for concurrency and communication. ZeroMQ provides a threading library based on Hewitt's *Actor Model* [4] of concurrent computation. Unlike the traditional idea of a global state shared by multiple threads, in ZeroMQ each actor can only modify its own state and influence other actors through atomic messages, thus avoiding locks, semaphores, or critical sections. ZeroMQ also provides an intelligent transport layer for low latency communication and is best suited for small packets. This feature is ideal for the fine-grained parallelism required during the graph compression phase of the assembly pipeline.

5. EVALUATION

We performed our experiments on the QueenBeeII cluster with two 10-core 2.8 GHz E5-2680v2 Xeon processors and 64GB memory per node. In addition to having access to a Lustre file system, each node has a local 250GB scratch storage. We evaluated Lazer, Swap, and Ray using the Yoruban male genome dataset (Accession #SRA000271) with 452GB of raw reads on several cluster configurations starting from two nodes to 128 nodes as shown in Figure 2. We were not able to run SWAP on 32 nodes due to out-of-memory errors. Furthermore, we could not run Ray on all the configurations due to cluster wall-clock limitations. We also observed that Ray's performance degrades after 64 nodes.

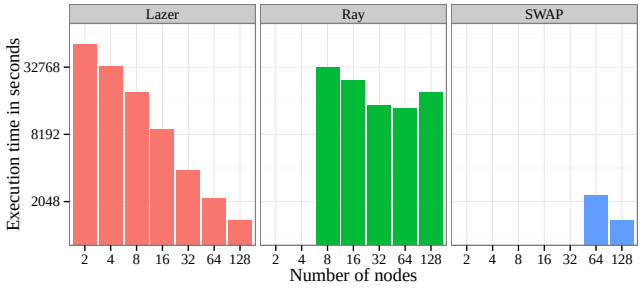


Figure 2: Execution times of Lazer, Swap, and Ray on the Yoruban male dataset (452GB)

6. CONCLUSION

In this paper, we introduced a distributed assembly framework that utilizes a smart partitioning scheme for De Bruijn graphs to achieve both scalability and memory efficiency. Experimental results show that our framework significantly reduces the memory footprint while ensuring fast assembly.

7. REFERENCES

[1] Zeromq. http://zeromq.org/.

[2] S. Boisvert, F. Laviolette, and J. Corbeil. Ray: simultaneous assembly of reads from a mix of high-throughput sequencing technologies. *Journal of Computational Biology*, 17(11), 2010.

[3] R. Chikhi and G. Rizk. Space-efficient and exact de Bruijn graph representation based on a Bloom filter. *Algorithms for Molecular Biology*, 8(1), 2013.

[4] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI*, 1973.

[5] Y. Li, P. Kamousi, F. Han, S. Yang, X. Yan, and S. Suri. Memory efficient minimum substring partitioning. In *VLDB*, volume 6, 2013.

[6] J. Meng, B. Wang, Y. Wei, S. Feng, and P. Balaji. SWAP-Assembler: scalable and efficient genome assembly towards thousands of cores. *BMC bioinformatics*, 15, 2014.

[7] 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*, 2009.