Parallelizing Big De Bruijn Graph Traversal for Genome Assembly on GPU Clusters

Shuang Qiu(✉), Zonghao Feng, and Qiong Luo

The Hong Kong University of Science and Technology,
Clear Water Bay, Hong Kong
{sqiuac,zfengah,luo}@cse.ust.hk

Abstract. De Bruijn graph traversal is a critical step in de novo assemblers. It uses the graph structure to analyze genome sequences and is both memory space intensive and time consuming. To improve the efficiency, we develop ParaGraph, which parallelizes De Bruijn graph traversal on a cluster of GPU-equipped computer nodes. With effective vertex partitioning and fine-grained parallel algorithms, ParaGraph utilizes all cores of each CPU and GPU, all CPUs and GPUs in a computer node, and all computer nodes of a cluster. Our results show that ParaGraph is able to traverse billion-node graphs within three minutes on a cluster of six GPU-equipped computer nodes. It is an order of magnitude faster than the state-of-the-art shared memory based assemblers, and more than five times faster than the current distributed assemblers.

1 Introduction

In genomic analysis pipelines, de novo assembly constructs genome sequences from short DNA fragments, without any reference sequence. Specifically, a De novo assembler first constructs a De Bruijn graph from short DNA fragments. Then it traverses the graph heuristically, searching local shortest paths. Thereafter, it outputs long DNA sequences (called contigs), representing the skeleton of the genome sequence. The performance issues in constructing big De Bruijn graphs are well addressed in recent work [2,7], but traversing big graphs efficiently with a limited number of machines remains an open problem.

Traversing a De Bruijn graph with hundreds of millions to billions of vertices takes hours on a single CPU core, and the memory consumption is tens to hundreds of gigabytes [4]. Parallelization is therefore commonly adopted in existing assemblers to speed up the traversal. In shared memory based parallel assemblers, the performance is limited by the memory size in a single machine [4] or the IO bandwidth for the data transfer [2]. In comparison, scalable distributed assemblers are able to handle big graphs [5,8]. However, the De Bruijn graph partitioning in these assemblers is based on a random strategy, ignoring connections among vertices. Consequently, the communication overhead is high in distributed assemblers. Additionally, all these assemblers are based on the CPU, whereas the GPUs, commonly equipped in the computer nodes, are not utilized.

G. Li et al. (Eds.): DASFAA 2019, LNCS 11448, pp. 466–470, 2019.
https://doi.org/10.1007/978-3-030-18590-9_68

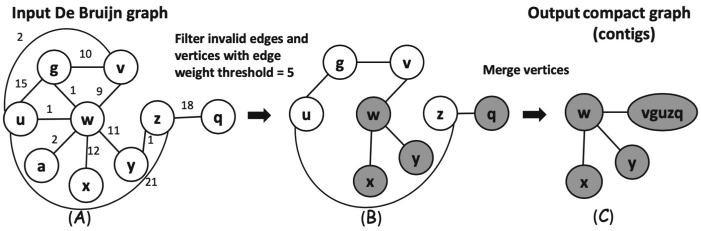

Fig. 1. De Bruijn graph traversal

The major difficulty that hinders the utilization of GPUs in De Bruijn graph traversal is the high divergence and read-write contention among GPU threads. To address this problem, we design algorithms in a vertex-centric manner, and split the vertex merging step into vertex traversing and result gathering steps. This way, graph traversal algorithms are converted into gather and scatter operations, and the costly write contention on many-core processors is resolved. Furthermore, we find that the identifiers of adjacent vertices share a common minimum substring with a high probability, and that this feature can be utilized to reduce the number of edge cuts between subgraphs. Therefore, we utilize a vertex partitioning algorithm [3,7] to reduce the communication overhead in ParaGraph.

2 ParaGraph

We design the workflow of ParaGraph as shown in Fig. 2. ParaGraph takes De Bruijn subgraphs as input, which can be generated from a graph construction tool [7]. In Step 1, we first filter invalid edges and their incident vertices, based on edge weights (as illustrated in Fig. 1(A) and (B)). Since a vertex in the graph is identified with two fixed-length strings, searching a vertex can be implemented as hash table lookup or binary search on sorted vertices. On multiple processors and computer nodes, our graph algorithms are designed with multiple iterations of neighbor updates for vertices. Therefore, we build a new index for vertices and update the index of neighbors of vertices. The efficiency in searching vertices is improved in graph traversal, and the index building overhead is offset by the performance gain. We finally split vertices into the set of linear vertices (vertices of at most two adjacent vertices) and the set of junctions (non-linear vertices).

Based on the results in Step 1, Step 2 traverses the De Bruijn graph and merges linear vertices (as illustrated in Fig. 1(B) and (C)). Specifically, we first

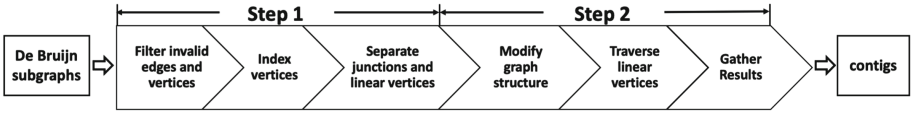

Fig. 2. Processing flow in ParaGraph

modify the graph structure to undirected graphs, which reduces the number of message transfers. To resolve data contention and reduce divergence across threads, we split vertex merging into two steps. First, we traverse linear vertices, and for each vertex v, we record the number of hops to traverse from v to the nearest junction or the nearest leaf (i.e., an end point of connected vertices). If v is in a cycle, we use the smallest vertex ID to identify the cycle and record the distance from v to the smallest vertex. Then we gather the linear vertices based on the recorded distances to the end points, and output contigs.

We design distributed graph processing algorithms, following the push based BSP (Bulk Synchronous Parallel) model [1], in which active vertices send messages to their neighbors, and the neighbors update the associated values based on the received messages. Specifically, we break down a graph algorithm as iterations of compute, communicate, and update operations, and define the compute operators and update operators on vertices. With this abstraction, traversing linear vertices takes $O(logD)$ number of iterations of compute, communicate and update operations, where D is the diameter of the graph.

In distributed graph algorithms, the graph partitioning method is critical in determining the number of messages across computers. We find that utilizing character distribution in vertex identifiers (two fixed-length strings) can reduce the number of edge cuts in partitioning the De Brujin graph. As such, we adopt the P-minimum-substring partitioning [3,7] and prove the following property with uniform and independent distribution assumptions on the start positions of P-minimum-substrings.

With the P-minimum-substring partitioning, the probability that a vertex v and its neighbor u are distributed to the same partition equals $\left(\frac{K-P-1}{K-P}\right)^2$, where K is the string length of the vertex and P is the P-minimum-substring length.

When $(K-P-1)$ is close to $(K-P)$, the majority of adjacent vertices are partitioned into the same subgraph. For example, given $K = 27, P = 11$, this property indicates that more than 85 percent adjacent vertices are located in the same partition.

3 Performance Evaluation

We use two datasets in the evaluation. The dataset *Human Chr14* [4] contains 450 million valid vertices in the De Bruijn graph. The dataset *7 Humans* [6], consisting of seven individual human genomes, contains 2.3 billion valid vertices. Experiments were conducted with one to six computer nodes. Each computer node contains two 2.3 GHz Intel Xeon E5-2670 12-core CPUs, and two Nvidia K80 GPUs. Each K80 consists of two GPUs, each with 12 GB memory. The main memory on each computer node is 128 GB. Each computer node uses Infiniband for network connection.

Overall Performance. We show the overall performance of ParaGraph in Fig. 3. On *Human Chr14*, the overall time of ParaGraph with all CPUs and GPUs is reduced to less than 1/3 of the time with only the CPUs. Moreover,

the running time is reduced to 1/8 of the time with CPUs on a single machine, when ParaGraph runs on six computer nodes. Due to the limit of memory size on a single machine, we run ParaGraph on *7 Humans* on six computer nodes. The running time with both CPUs and GPUs is reduced to 1/3 of the time using only CPUs.

Comparison with Other Assemblers. We compare ParaGraph with the state-of-the-art parallel assemblers. All these CPU-based assemblers use all CPUs on each computer in experiments. Time measurement for each assembler begins at the time the input data is ready in memory and ends at the time the output results are generated in memory. As shown in Table 1, on *Human Chr14*, ParaGraph with CPUs and GPUs is 20 times faster than SOAPdenovo [4] and bcalm2 [2] on a single machine. It is about eight times faster than SWAP2 [5], and six times faster than PPA [8]. On *7 Humans*, only Bcalm2 and ParaGraph are able to run with the available amount of memory. ParaGraph on six computer nodes is 40 times faster than bcalm2 on a single computer.

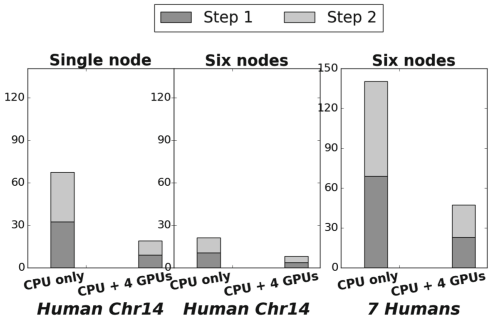

Fig. 3. Overall running time (sec) of ParaGraph

Table 1. Running time (sec) comparison

Software \ Dataset	Human Chr14		7 Humans	
	Single[1]	Multi[2]	Single[1]	Multi[2]
SOAPdenovo	582	OM	OM	OM
bcalm2	485	OM	**2341**	OM
PPA	OM	59	OM	OM
SWAP2	209	68	OM	OM
ParaGraph-CPU	71	22	OM	144
ParaGraph-CPU-GPU	**24**	**9**	OM	**52**

1 Single computer node, 2 Six computer nodes
OM: Out of memory
SOAPdenovo: from tip removing to edge construction. bcalm2: graph compaction within and across buckets, with IO time excluded. SWAP2: graph simplification. PPA: listranking and contig merging, with load and dump time excluded.

4 Conclusion

We propose ParaGraph to parallelize the De Bruijn graph traversal on GPU-equipped clusters. We implement multi-threaded algorithms on each GPU and CPU, use threads to manage message transfers and synchronizations among CPUs and GPUs in a computer node, and finally run concurrent processes on multiple computer nodes. To reduce the overhead in distributed graph traversal, we utilize the identifier distribution features in vertices, such that the majority of messages are within each processor. As a result, ParaGraph is efficient on multiple processors and multiple computer nodes. Source code of ParaGraph is available at https://github.com/ShuangQiuac/UNIPAR, integrated with our

previous work ParaHash [7] to execute the entire workflow of De Bruijn graph construction and traversal.

References

1. Avery, C.: Giraph: large-scale graph processing infrastructure on Hadoop. In: Proceedings of the Hadoop Summit. Santa Clara, vol. 11, pp. 5–9 (2011)
2. Chikhi, R., Limasset, A., Medvedev, P.: Compacting de bruijn graphs from sequencing data quickly and in low memory. Bioinformatics **32**(12), i201–i208 (2016)
3. Li, Y., Kamousi, P., Han, F., Yang, S., Yan, X., Suri, S.: Memory efficient minimum substring partitioning. In: Proceedings of the VLDB Endowment, vol. 6, pp. 169–180. VLDB Endowment (2013)
4. Luo, R., et al.: Soapdenovo2: an empirically improved memory-efficient short-read de novo assembler. Gigascience **1**(1), 18 (2012)
5. Meng, J., Seo, S., Balaji, P., Wei, Y., Wang, B., Feng, S.: Swap-assembler 2: optimization of de novo genome assembler at extreme scale. In: 2016 45th International Conference on Parallel Processing (ICPP), pp. 195–204. IEEE (2016)
6. Minkin, I., Pham, S., Medvedev, P.: Twopaco: an efficient algorithm to build the compacted de bruijn graph from many complete genomes. Bioinformatics **33**(24), 4024–4032 (2016)
7. Qiu, S., Luo, Q.: Parallelizing big de bruijn graph construction on heterogeneous processors. In: 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), pp. 1431–1441. IEEE (2017)
8. Yan, D., Chen, H., Cheng, J., Cai, Z., Shao, B.: Scalable de novo genome assembly using pregel. arXiv preprint arXiv:1801.04453 (2018)