Distributed Memory Partitioning of High-Throughput Sequencing Datasets for Enabling Parallel Genomics Analyses

Nagakishore Jammula, Sriram P. Chockalingam, and Srinivas Aluru

Georgia Institute of Technology

ACM BCB – August 2017

Presented by: Evan Stene

Outline

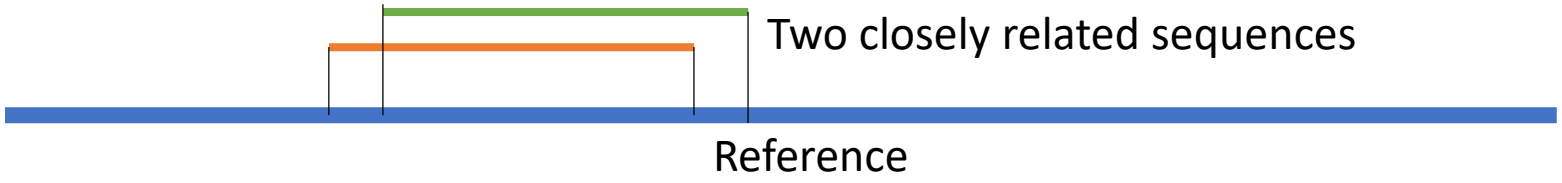
- Introduction
- Background
- Methods
- Results

Introduction

- Large volume of short biological sequence data
- Construction of long sequences from short is time consuming
  - Use existing sequence for reference
  - Compare short sequences to each other
- Distributed computing is a promising direction for speed up
- **Can intelligent partitioning benefit sequence construction?**

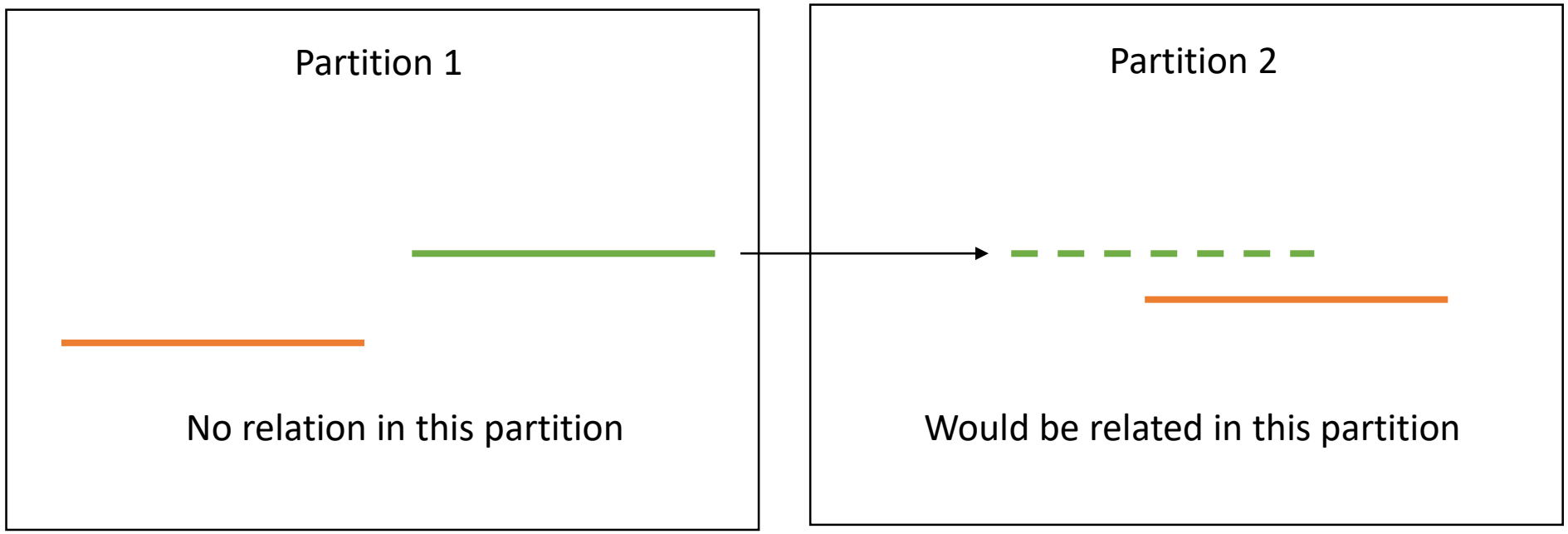
Introduction – Case 1: Alignment

- Assuming a copy of a reference exists in each partition
- Partitioning dataset is simply balancing partition sizes (always true?)
- Reference acts as global coordinate system
- Position on reference will infer relations across partitions



Introduction – Case 2: Assembly

- Ideally, group reads with greatest relation
- How can we calculate overlap quickly?



Introduction – Motivation

- Intelligently partitioning entire dataset can be time consuming (depending on method)
- Partitioning a graph that represents the dataset is a good partition of the dataset as well
- Related works:
  - Pairwise similarity – very time consuming
  - Hash partition of graph – destroys data locality

Background – de Bruijn Graph

- Used to bring down number of comparisons in assembly
- Captures connection and frequency of common subsequences
- Tracing paths through graph recreate sequence dataset

Background – de Bruijn Graph

Input Sequences, k = 3

A A B C D D

A A B C E D

Graph

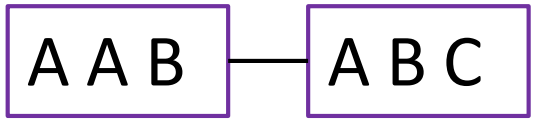
Background – de Bruijn Graph

Input Sequences, k = 3

A A B C D D

A A B C E D

Graph



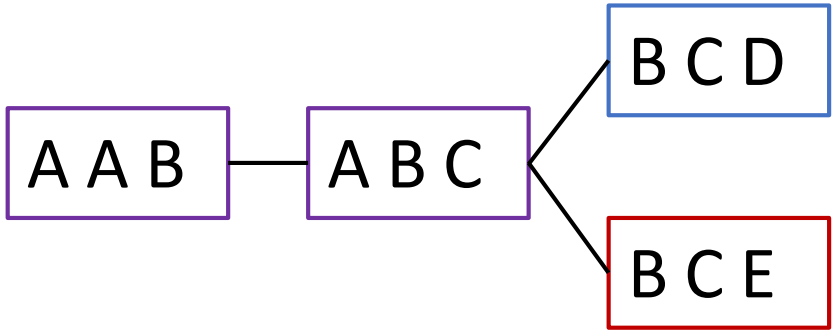
Background – de Bruijn Graph

Input Sequences, k = 3

A A B C D D

A A B C E D

Graph



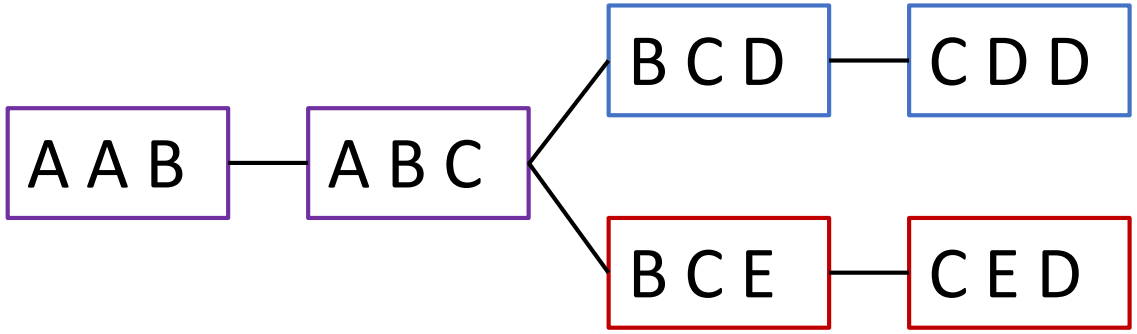
Background – de Bruijn Graph

Input Sequences, k = 3

A A B C D D

A A B C E D

Graph

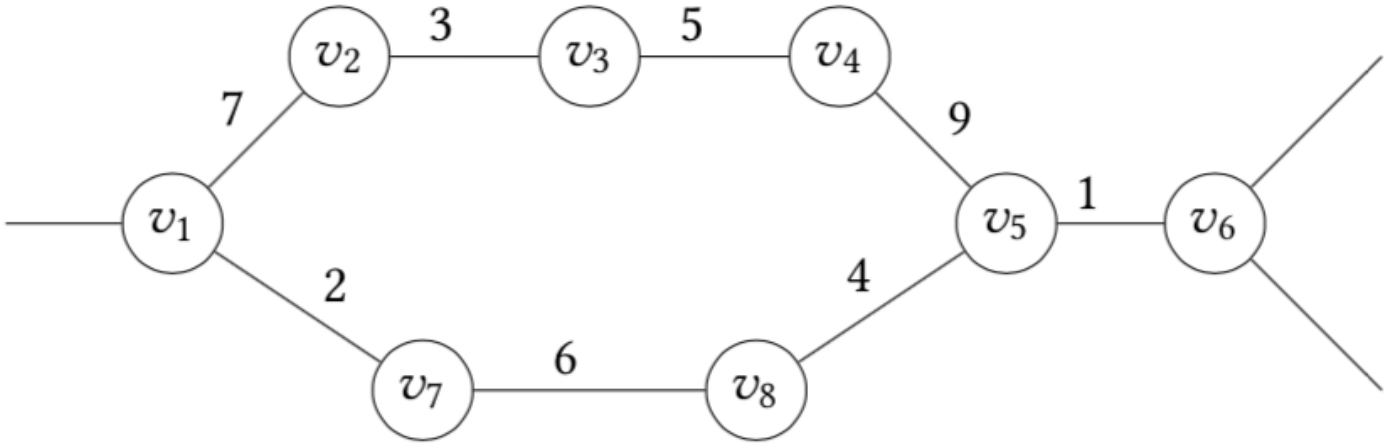


Methods

- Construction
- Compaction
- Graph Partition
- Dataset Partition

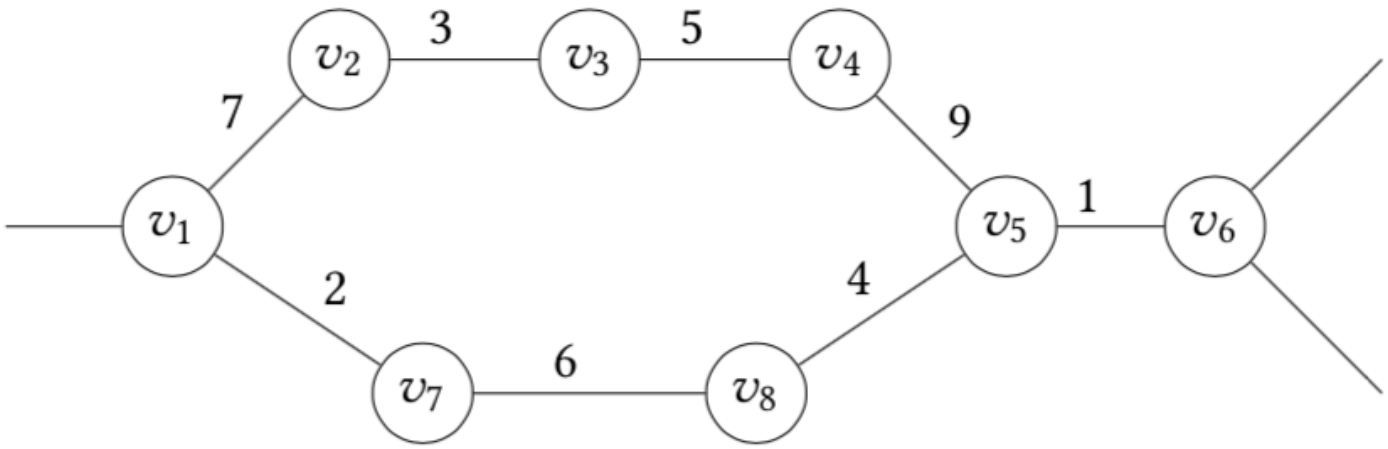
Methods - Construction

- Each vertex has at most 8 neighbors
  - Alphabet of size 4 for DNA
  - 4 edges in, 4 out
- 2 Classes of vertex:
  - Vertices that branch (>1 in/out edges)
  - Vertices in a chain



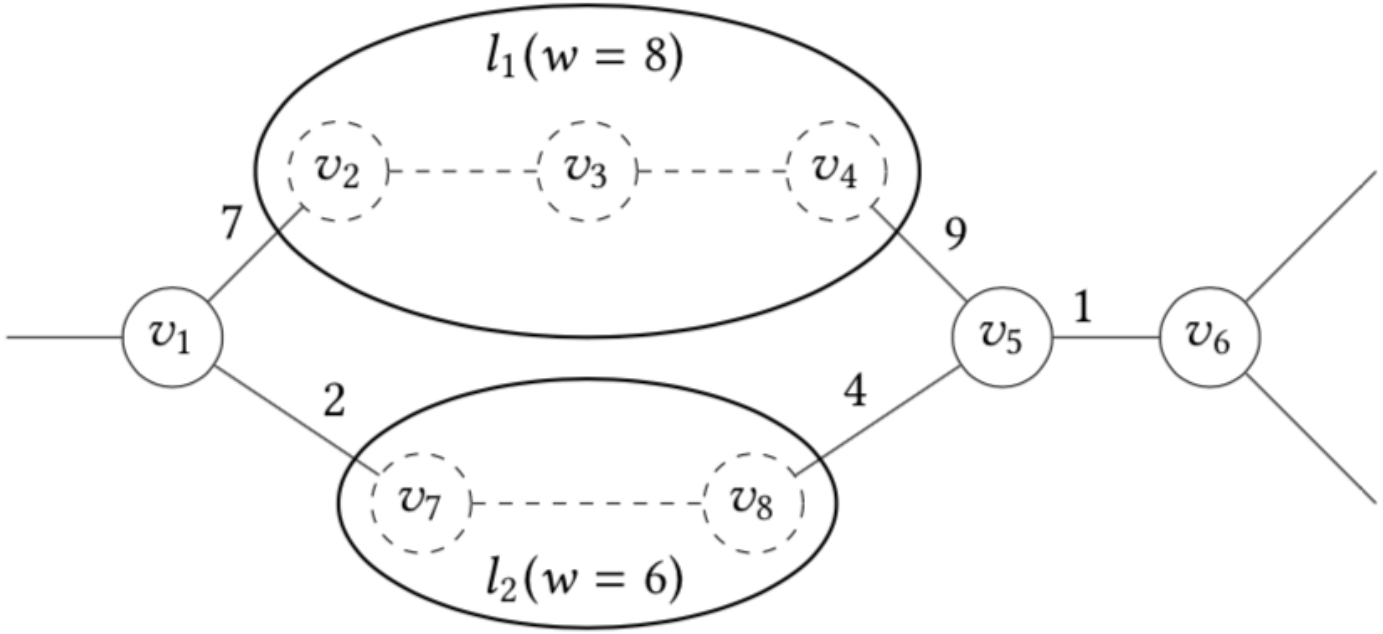
Methods - Construction

- Build hash table of all subsequence of size k in dataset
- For each subsequence check the 8 possible neighbors
- Add 1 to weight of edge for each occurrence
- Trim edges below some threshold



Methods - Compaction

- Essentially connected components
- Combine chain vertices into single node
- Concatenate labels (subsequences) and sum edge weights



Methods – Graph Partition

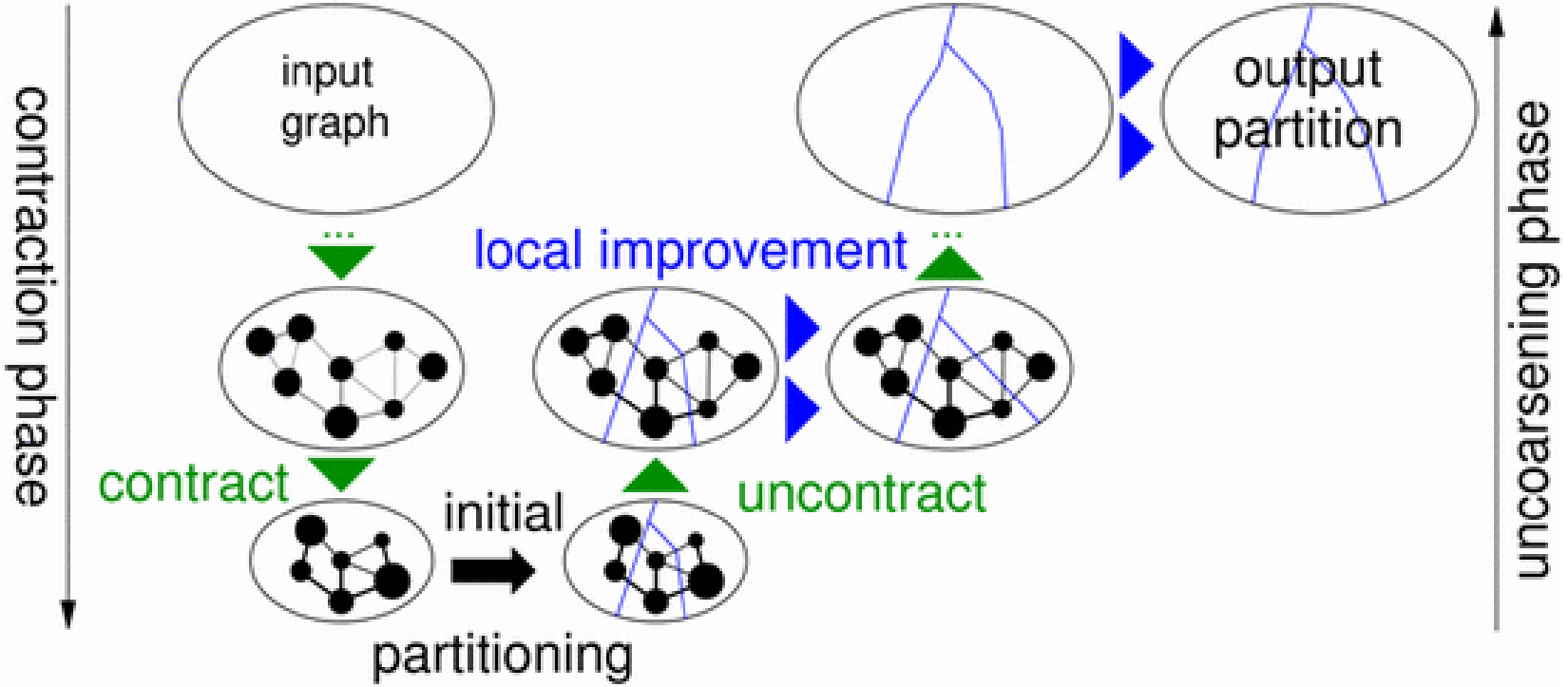
- Optimize two parameters – *min cut* and *balance*
- Cut defined as weight of edges between partitions
- Bound the balance of partitions by some threshold (1 + ε)

$$\frac{\max_{1 \le i \le m} C(V_i)}{\sum_{1 \le i \le m} C(V_i)/m}$$

Balance function
$C(V_i)$ = sum of weights of vertices in V_i
m = total number of partitions

Methods – Graph Partition

- Recursively coarsen graph
- Partition coarsest graph
- Recursively un-coarsen graph, refining cut after each iteration



From [1]

Methods – Dataset Partitioning

- Map partition id to each subsequence of length k
  - Chains will contain multiple subsequences that will need to map
- Build distributed index from mapping
- Assign sequence r to partition id most frequently assigned
  - Sequence r will contain |r| - k + 1 subsequences

Results – Test Environment

- 32 nodes
  - 16 cores
  - 128GB memory
- OpenMPI 1.8.6

Table 1: Datasets used for experimental evaluation

Dataset	Genome length (Giga base-pairs)	Dataset size (Giga bases)	Read length (Bases)
Fish	1.0 Gbp	52.7 Gb	101
Bird	1.2 Gbp	70.7 Gb	101
Snake	1.6 Gbp	84.1 Gb	121

Results - Compaction

Table 2: Reduction in the size of the DBG due to compaction

Dataset	Plain graph (No. of vertices)	Compacted graph (No. of vertices)	Compaction ratio
Fish	733,774,187	16,672,988	44
Bird	1,208,521,390	25,740,770	47
Snake	1,361,026,568	27,199,895	50

Results – Graph Partitioning

Table 3: Quality of de Bruijn graph partitioning

Dataset	Sum of weights of all edges in the graph	Sum of weights of edges cut	*Cut* ratio
Fish	13,593,910,042	19,252,245	1.42×10^{-3}
Bird	22,462,771,436	22,337,839	0.99×10^{-3}
Snake	29,754,489,857	47,197,297	1.59×10^{-3}

Results – Graph Runtimes

Table 4: Runtime in seconds for the *Bird* dataset for de Bruijn graph construction (Algorithm 1), chain labeling (Algorithm 2) and compaction (Algorithm 3).

No. of cores	Algorithm 1 (s)	Algorithm 2 (s)	Algorithm 3 (s)	Total (s)
64	391	790	33	1214
128	159	309	11	479
256	76	180	6	262
512	45	115	3	163

Results – Dataset Quality

Table 5: Read partitioning quality evaluation for all datasets

Dataset	No. overlapping read pairs	`inter-pairs`	Cut ratio
Fish	20,930,646,131	193,013,621	0.92×10^{-2}
Bird	14,302,047,674	304,849,664	2.13×10^{-2}
Snake	6,728,041,314	263,882,542	3.92×10^{-2}

Results – Dataset Runtimes

Table 6: Runtime in seconds for the *Bird* dataset for computing read partitioning from de Bruijn graph partitioning.

No. of cores	64	128	256	512
Runtime (s)	2090	950	456	226

Total runtime for Bird dataset on 512 cores: 11 min

References

[1] Henning Meyerhenke, Peter Sanders, and Christian Schulz. 2015. Parallel Graph Partitioning for Complex Networks. In Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium. 1055–1064.