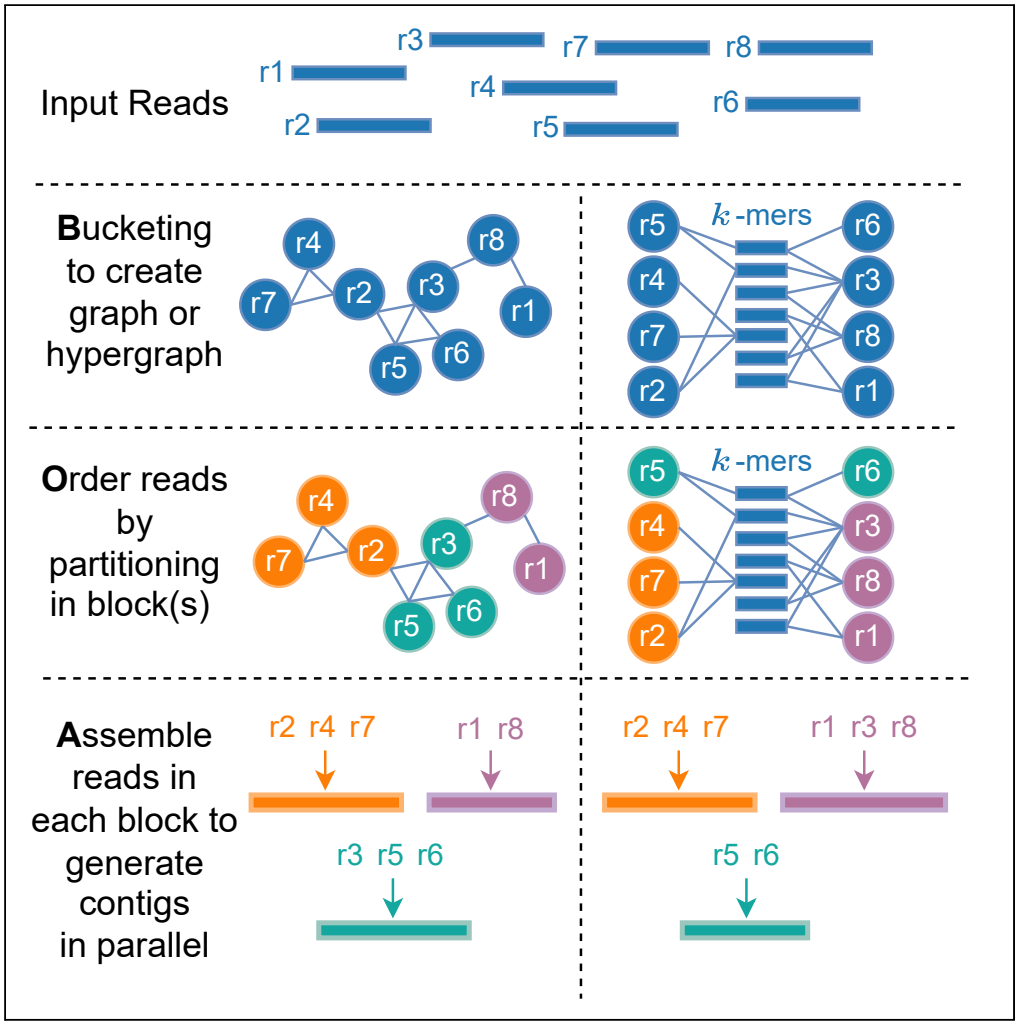
iScience

CellPress
OPEN ACCESS

Article

BOA: A partitioned view of genome assembly



Xiaojing An, Priyanka Ghosh, Patrick Keppler, ..., Aravind Sukumaran Rajam, Ümit V. Çatalyürek, Ananth Kalyanaraman

ananth@wsu.edu

Highlights

A graph/hypergraph partitioning based method to improve assembly quality and runtime

Bucketing and graph/hypergraph partitioning to partition reads into blocks

Each block is then independently assembled using any standalone assembler

Hypergraph variant produces more precise contigs and is faster than state-of-the-art assemblers

An et al., iScience *25*, 105273
November 18, 2022
https://doi.org/10.1016/j.isci.2022.105273

iScience

CellPress
OPEN ACCESS

Article

BOA: A partitioned view of genome assembly

Xiaojing An,[1,7,8] Priyanka Ghosh,[2,7] Patrick Keppler,[3] Sureyya Emre Kurt,[4] Sriram Krishnamoorthy,[6] Ponnuswamy Sadayappan,[4] Aravind Sukumaran Rajam,[3] Ümit V. Çatalyürek,[1,5] and Ananth Kalyanaraman[3,*]

SUMMARY

***De novo* genome assembly is a fundamental problem in computational molecular biology that aims to reconstruct an unknown genome sequence from a set of short DNA sequences (or *reads*) obtained from the genome. The relative ordering of the reads along the target genome is *not* known *a priori*, which is one of the main contributors to the increased complexity of the assembly process. In this article, with the dual objective of improving assembly quality and exposing a high degree of parallelism, we present a partitioning-based approach. Our framework, BOA (bucket-order-assemble), uses a bucketing alongside graph- and hypergraph-based partitioning techniques to produce a partial ordering of the reads. This partial ordering enables us to divide the read set into disjoint blocks that can be independently assembled in parallel using any state-of-the-art serial assembler of choice. Experimental results show that BOA improves both the overall assembly quality and performance.**

INTRODUCTION

In *de novo* genome assembly, the relative ordering and orientation of the input reads along the target genome is not known *a priori*. In fact, it can be argued that one of the primary contributors to the problem complexity is the lack of this information—i.e., if the ordering and orientation of the reads is known at input then the genome assembly problem would reduce to a simpler (albeit less exciting) problem of performing a linear sequence of pairwise alignments between adjacent reads to produce the assembly. However, the DNA sequencers preserve neither the genomic coordinates from where the reads were sequenced nor any significant relative ordering information between the reads (except for paired end read information). Consequently, assembly algorithms are left to infer an ordering and orientation along the course of their respective computations.

Different assembly approaches vary on how much they rely on the read ordering and orientation (henceforth abbreviated as *OO* for simplicity) information, and at what stages of their algorithm they try to infer it. DeBruijngraph assemblers Compeau et al. (2011); Medvedev and Pop (2021); Pevzner et al. (2001), which now represent a dominant segment of modern day short-read assemblers, use an approach that is largely oblivious to OO information. This is because these assemblers use deBruijn graphs that break the reads into shorter fixed-length k-mers at the early stages of the algorithm. Therefore, the information on how the reads are ordered/oriented along the target genome is typically not recoverable until the end of the assembly pipeline (i.e., until after contigs are generated). On the other hand, the more traditional overlap-layout-consensus (OLC) class of assemblers Li et al. (2012); Medvedev and Pop (2021); Pop (2009) are more explicit in trying to infer the OO information in their assembly pipeline—as the overlap phase aligns reads against one another with an intent to arrive at a read layout. And yet, because the overlap phase is also the most time consuming step of the assembly pipeline for the OLC assemblers, the OO information is practically not available until later stages of the assembly.

In this article, we ask the simple question of what if either a total (ideal but not practical) or at least a partial order information can be generated earlier in the assembly computation(In this article, the notion of a *total ordering* is used to imply that the relative ordering between every pair of reads is established; whereas in a *partial order*, the relative ordering is established only for a subset of read pairs). Could that help improve performance and/or assembly quality? If so, what are some of the ways to generate such OO information earlier in the assembly algorithmic stages and what are their assembly efficacies?

[1]School of Computational Science and Engineering, Georgia Institute of Technology, Atlanta, GA 30332, USA

[2]National Center for Biotechnology Information, National Library of Medicine, National Institutes of Health, Bethesda, MD 20894, USA

[3]School of Electrical Engineering and Computer Science, Washington State University, Pullman, WA 99164, USA

[4]School of Computing, University of Utah, Salt Lake City, UT 84112, USA

[5]Amazon Web Services, Seattle, WA 98109, USA

[6]Google, Mountain View, CA 94043, USA

[7]These authors contributed equally

[8]Lead contact

*Correspondence: ananth@wsu.edu

https://doi.org/10.1016/j.isci.2022.105273

Table 1. The inputs used in our experiments

Genome	Size (bp)	No. reads (=$\mathcal{V}$)	No. buckets	No. pins (=$\mathcal{N}$)	No. edges (=$\mathcal{E}$)
C.elegans	100,286,401	100,286,100	409,957,423	6,389,329,498	9,342,286,308
D. melanogaster	143,726,002	142,426,015	555,183,926	8,250,921,240	11,757,427,193
Human chr 7	160,567,423	160,567,400	620,586,298	9,651,040,529	16,009,424,797
Human chr 8	146,259,322	146,259,300	574,127,869	8,923,132,914	13,977,225,241
Human chr 10	134,758,122	134,758,100	527,306,188	8,211,994,915	13,248,263,074
Maize chr 10	152,435,371	152,313,178	469,060,854	5,869,048,129	14,305,585,805
Bettasplendens	456,232,186	394,258,510	1,610,294,923	25,105,195,932	36,509,423,159

Contributions

To address the above questions, we present a parallel assembly framework that uses a graph partitioning-centric approach. Graph partitioning Garey et al. (1974) is a classical optimization problem in graph theory that aims to partition the set of vertices of an input graph into a pre-determined number of partitions in a load balanced manner. The problem has seen decades of research in development and application under numerous contexts including in the parallel processing of graph workloads Hendrickson and Kolda (2000), as well as partitioning assembly graphs Pell et al. (2012) and read datasets Al-Okaily (2016); Jammula et al. (2017).

In this article, we exploit graph partitioning and its properties to produce a partial ordering of reads and in the process also enable parallelization of the assembly workload. More specifically:

- We cast the assembly problem in two forms: a) one that uses graph partitioning, and b) another that uses hypergraph partitioning.
- To enable the application for different types of partitioning, we propose a light-weight bucketing algorithm that bins reads into buckets based on fixed-length exact matches and uses the bins to generate graph/hypergraph representations suitable for partitioning.
- Once bucketed and partitioned, each individual part can be independently assembled. This strategy allows the user to use any standalone (off-the-shelf) assembler of choice. Consequently, we call our assembly framework BOA (stands for bucket-order-assemble). An overview is shown in Figure 1. Two implementations (i.e., concrete instantiations) of this framework are presented and evaluated—one that uses a classical graphpartitioner (ParMETIS Karypis et al. (1997)), Graph-BOA, and another that uses a hypergraph partitioner (Zoltan Devine et al. (2006)), Hyper-BOA.
- To comparatively assess the assembly efficacy of the partitioning-based approach, we also construct a benchmark *Oracle* assembly workflow that uses the correct read ordering available from sequencing simulators.

Experimental results on simulated and real-word datasets demonstrate that our partitioning-based implementations a) improve parallel performance of assembly workloads; and b) improve assembly quality, consistently under several qualitative measures. In fact, on the simulated datasets, the partitioning-based approaches yield results that come closest in terms of quality to the Oracle assemblies produced.

RESULTS

Experimental evaluation was performed on a range of genome inputs—covering model organisms, to human and plant chromosomal DNA—downloaded from NCBI GenBank Duke University School of Medicine (Last date accessed: November 2021). All inputs used are listed in Table 1. Short reads were generated from these reference genomes using the ART sequencing simulator Huang et al. (2012) using an average read length of 100bp, coverage of 100 × , and with paired-end read information. For the Betta genome, the ART sequencing run resulted in 86× coverage. An experiment on a real world data for *D. melanogaster* is presented in Section real world experiment. The QUAST Gurevich et al. (2013) tool was used to assess the quality of the output assemblies.

All our experiments were conducted on the NERSC Cori machine (Cray XC40), where each node has 128GB DDR4 memory and is equipped with dual 16-core 2.3 GHz Intel Haswell processors. The nodes are interconnected with the Cray Aries network using a Dragonfly topology.

The BOA framework is a three-step pipeline:(1) parallel bucketing of input reads; (2) parallel partitioning the reads using either hypergraph partitioning (Hyper-BOA) or graph partitioning (Graph-BOA); and (3) subsequently running a standalone assembler on each part (in parallel). For hypergraph partitioning, we use Devine et al. (2006), and for standard graph partitioning we use ParMETIS Karypis et al. (1997). By default, for all our experiments we used $k = 31$, $l = 8$ and paired-end read information (Hyper-BOA, Graph-BOA).

For the last step of BOA, any standalone assembler can be used. In our experiments, we used MEGAHIT Li et al. (2015), Minia Chikhi and Rizk (2013) and IDBA-UD Peng et al. (2012) as three different options for assembling each block partition in the last step with $k = 31$. Hyper-BOA (minia) refers to the version that uses Minia; Hyper-BOA (idba-ud) uses IDBA-UD; and Hyper-BOA (megahit) uses MEGAHIT.

As baselines for comparing our BOA assemblies, we also generated two other assemblies: (1) The Oracle assembly was generated by: i) first recording the *true and total* read ordering along the genome (i.e., *oracle* ordering) using the read coordinate information from the ART simulator; ii) then trivially block partitioning the oracle ordering of the reads into roughly equal sized blocks (or parts), with the same block size (ρ) used in the partitioning-based approaches; and iii) subsequently running Minia and MEGAHIT on each individual block. (2) In addition, we ran Minia, IDBA-UD and MEGAHIT on the entire read set to enable direct comparison of our partitioning based approach against a (partitioning-free, or $K = 1$) standalone assembler.

Qualitative evaluation

We first present a qualitative evaluation of the BOA framework alongside comparisons to Minia, IDBA-UD, and MEGAHIT standalone assemblies and the Oracle assembly. MEGAHIT and IDBA-UD runs were with paired-end reads, and Minia does not support paired-end reads. Note that the Oracle assembly is not realizable in practice and is used just as a theoretical benchmark for comparison purposes. The Minia, IDBA-UD, and MEGAHIT assemblies are meant to be representative outputs from a typical state-of-the-art standalone assembler. Table 2 shows the results with various qualitative measures including NGA50, N50, largest alignment (in bp), genome coverage (in %), number of misassemblies, and duplication ratio. To enable a fair comparison, we set the number of parts (K) to 400 for both Zoltan and ParMETIS runs.

The results show that Hyper-BOA implementations consistently outperform all other assemblers tested by nearly all the qualitative measures, and for almost all inputs tested. Among the Hyper-BOA implementations, Hyper-BOA (megahit) is the best. Relative to the MEGAHIT standalone assembler, Hyper-BOA (megahit) consistently improves the NGA50 values by an average of 2× and up to 2.5 × ; the N50 values by an average of 1.70× and up to 2.13 × ; whereas the largest alignment length improves 1.47× on average and up to 1.94 × . Hyper-BOA (minia) also improves the assembly quality of its standalone counterpart Minia by similar margins. Intuitively, partitioning can help reduce noise within blocks but there is no guarantee for it as the bucketing step still uses exact matches to group the reads. Repetitive k-mers could still confound the partitioning process. We see the effect of these possibly noisy k-mers in the misassemblies reported by the Hyper-BOA implementations. Yet, the choice of the standalone assembler at the end of the partitioning pipeline provides certain degree of control over these misassemblies, with IDBA-UD typically resulting in fewer missassemblies than the other assemblers.

From Table 2, we also observe that Hyper-BOA results consistently come within 90% or more reach of the quality values produced by the corresponding Oracle assembly. For instance, on average Hyper-BOA (megahit) reaches within 93% of the corresponding Oracle (megahit) NGA50 values, and within 100% of the respective largest alignment values on average. The largest gap is seen in *Human chr 8*, where Hyper-BOA (megahit)'s largest alignment is only 81% of the Oracle's value. Even in this case, however, the Hyper-BOA's largest alignment is considerably larger (1.48 ×) than that of standalone MEGAHIT value.

Of interest, we also note in Table 2 that for two inputs, *Human chr 10* and *C.elegans*, the largest alignment values produced by Hyper-BOA (minia) are marginally better than that of the Oracle values. This can sometimes happen because, after all, the assembly quality is ultimately a function of the block composition that is fed into the final stage of BOA assembly; and the composition between the blocks for Hyper-BOA could

Table 2. Quality metrics for our test inputs across multiple assemblers

Input	Assembler	NGA50	N50	Largest Alignment (bp)	Genome Coverage %	Miss assemblies	Duplication Ratio
C.elegans	Oracle (minia)	11,162	14,172	153,394	91.65	10	1.002
	Oracle (megahit)	11,979	14,189	157,192	91.49		1.005
	Minia	4,155	5,924	75,229	83.26	37	**1.002**
	IDBA-UD	4,387	6,026	75,229	83.14	**0**	**1.002**
	MEGAHIT	4,464	6,276	108,538	83.71	1	**1.002**
	Graph-BOA (minia)	7,829	9,028	143,663	85.83	49	1.013
	Hyper-BOA (minia)	**11,977**	12,715	**158,433**	89.96	19	1.013
	Hyper-BOA (idba-ud)	11,116	**13,404**	**158,433**	89.91	5	1.014
	Hyper-BOA (megahit)	(2.5 ×)11,246	(1.2 ×)12,673	(1.3 ×)143,817	**92.10**	11	1.026
D. melanogaster	Oracle (minia)	41,283	55,104	356,760	88.81	41	1.005
	Oracle (megahit)	46,516	57,037	356,561	88.51	13	1.006
	Minia	13,229	19,551	162,262	78.79	37	**1.002**
	MEGAHIT	16,397	24,312	190,107	78.97	**0**	**1.001**
	Graph-BOA (minia)	19,421	24,136	201,618	83.78	328	1.106
	Hyper-BOA (minia)	38,923	**42,048**	295,288	86.16	299	1.081
	Hyper-BOA (megahit)	(2.4 ×)**40,101**	(1.7 ×)41,729	(1.8 ×)**343,434**	**87.81**	225	1.124
Human chr 7	Oracle (minia)	3,350	4,564	39,858	84.26	40	1.003
	Oracle (megahit)	3,558	4,569	39,858	84.21	40	1.124
	Minia	1,544	2,793	36,845	68.10	88	**1.002**
	IDBA-UD	1,599	2,834	24,503	67.98	**0**	**1.002**
	MEGAHIT	1,638	2,904	36,845	68.95	**0**	**1.002**
	Hyper-BOA (minia)	**4,124**	4,385	39,314	79.54	58	1.008
	Hyper-BOA (idba-ud)	3,285	**4,585**	39,352	79.87	**0**	1.010
	Hyper-BOA (megahit)	(2.0 ×)3,331	(1.5 ×)4,316	(1.2 ×)**43,498**	**83.30**	10	1.018
Human chr 8	Oracle (minia)	3,944	4,869	42,828	88.44	34	1.003
	Oracle (megahit)	4,194	4,883	56,943	88.40	1	1.005
	Minia	1,877	2,784	27,427	74.28	76	**1.002**
	MEGAHIT	1,987	2,893	31,115	75.27	**0**	**1.002**
	Hyper-BOA (minia)	**4,379**	4,569	37,028	86.02	29	1.010
	Hyper-BOA (megahit)	(2.0 ×)4,044	(1.6 ×)**4,604**	(1.5 ×)**46,122**	**88.92**	4	1.020
Human chr 10	Oracle (minia)	3,462	4,392	37,537	87.12	28	1.003
	Oracle (megahit)	3,685	4,395	37,429	87.10	1	1.005
	Minia	1,672	2,654	33,773	71.73	78	**1.002**
	MEGAHIT	1,766	2,755	33,773	72.59	**0**	**1.002**
	Hyper-BOA (minia)	**3,942**	**4,149**	42,959	83.02	41	1.007
	Hyper-BOA (megahit)	(1.9 ×)3,428	(1.5 ×)4,125	(1.3 ×)**44,604**	**86.46**	1	1.017
Maize chr 10	Oracle (minia)	841	3,906	35,657	56.33	4	**1.003**
	Oracle (megahit)	904	3,903	35,657	56.33	0	1.005
	Minia	–	2,058	15,644	17.08	29	**1.003**
	MEGAHIT	–	2,134	15,645	17.34	**0**	**1.003**
	Hyper-BOA (minia)	–	**3,629**	30,306	34.23	178	1.056
	Hyper-BOA (megahit)	–	(1.2 ×)2,559	(2.0 ×)**30,664**	**39.64**	86	1.102

The target number of reads per part (ρ) for Graph-BOA and Hyper-BOA was set to 500K. Also shown in parentheses (×) are the factor of improvements achieved by Hyper-BOA (megahit) over the corresponding standalone MEGAHIT values. Boldface entries are best values.

Table 3. Runtime performance of the different assemblers

Input	Assembler	Parallel Bucketing (sec): max	Parallel Partitioning (sec): max	Assembly (sec): avg	Total time (sec)
C.elegans	Graph-BOA (minia)	51	**180**	150	**381**
	Hyper-BOA (minia)	**33**	536	39	608
	Hyper-BOA (megahit)	**33**	536	**13**	582
	Minia				1,364
	MEGAHIT				2,000
D. melanogaster	Graph-BOA (minia)	81	**195**	51	**327**
	Hyper-BOA (minia)	**57**	867	39	963
	Hyper-BOA (megahit)	**57**	867	**18**	942
	Minia				2,444
	MEGAHIT				2,845
Human chr 7	Hyper-BOA (minia)	70	967	86	1,123
	Hyper-BOA (megahit)	70	967	16	**1,053**
	Minia				2,569
	MEGAHIT				3,377
Human chr 8	Hyper-BOA (minia)	67	826	61	954
	Hyper-BOA (megahit)	67	826	26	**919**
	Minia				2,518
	MEGAHIT				3,134
Human chr 10	Hyper-BOA (minia)	61	844	115	1,020
	Hyper-BOA (megahit)	61	844	18	**923**
	Minia				2,027
	MEGAHIT				2,970
Maize chr 10	Hyper-BOA (minia)	51	745	220	1,016
	Hyper-BOA (megahit)	51	745	19	**815**
	Minia				3,625
	MEGAHIT				3,670

The BOA implementations were run on the NERSC Cori machine with 256 cores (i.e. on 32 nodes with 8 processes per node), while the standalone Minia and *Bettasplendens* baselines run in multithreaded mode on a single node with 32 cores. All times reported are in seconds.

have favored longer growth of the longest contig (relative to the Oracle). NGA50 for Hyper-BOA (minia) is also consistently better than Oracle (minia). Overall, these results show that partitioning helps in closing the gap toward the theoretically achievable peaks in total read order-aware assemblies.

Hyper-BOA versus Graph-BOA

In our results we observed that in general, Hyper-BOA significantly outperforms Graph-BOA. For *C.elegans* and *D. melanogaster*, where both results are available, we see from Table 2 that Hyper-BOA implementations outperform Graph-BOA by all qualitative measures. This is to be expected as the input graph into Graph-BOA, are *not* weighted (see related discussion in Section graph-BOA and hyper-BOA). Note that for the remaining four inputs tested, Graph-BOA could *not* complete because of lack of memory. As described in Section graph-BOA and hyper-BOA, graphs can have a higher memory complexity even with edge duplication reductions shown in Figure 2.

Runtime performance evaluation

Table 3 shows the runtime performance for Hyper-BOA and Graph-BOA implementations, alongside standalone Miniaand MEGAHIT. The bucketing and partitioning steps are parallel, and therefore we report their parallel runtimes. For the assembly step, we report the mean processing time per block partition.

Table 4. Quality and runtime performance for *Bettasplendens* assembly

	NGA50	N50	Largest Alignment (bp)	Genome Coverage %	Missassemblies	Duplication Ratio	Total time Avg. (sec)	Total time Max. (sec)
Oracle (megahit)	5,551	**7,830**	84,290	89.58	1,132	1.005		*
Minia	3,425	5,571	59,787	81.85	878	**1.002**		5,415
MEGAHIT	4,253	5,765	59,789	82.05	**676**	**1.002**		10,313
Graph-BOA (megahit)	4,253	6,516	76,575	84.13	916	1.010	640	**663**
Hyper-BOA (minia)	5,362	7,458	96,553	88.75	1,254	1.012	2,159	3,017
Hyper-BOA (megahit)	5,427	7,474	**101,570**	**89.88**	1,140	1.016	1,791	1,812

Parallel bucketing and partitioning was performed across 512 cores of NERSC Cori (64 nodes × 8 cores per node) with 1024 partitions. The runs for baseline (standalone) Minia and MEGAHIT were executed on a shared memory node with 32 cores. (* indicates that these timings could not be collected in time on the same system.)

The results in Table 3 show that the BOA implementations are significantly faster than the standalone Minia and MEGAHIT executions. For instance, for the MEGAHIT runs, Hyper-BOA (megahit) delivers speedups consistently between 3 and 4× over standalone MEGAHIT. The speedups for the Minia runs are larger.

Large-scale experiment

As one large-scale experiment, we tested our Hyper-BOA (megahit) on the full assembly of the 456*Mbp Bettasplendens* (Siamese fighting fish). Table 4 shows the key results. Consistent with the results on smaller genomes, the Hyper-BOA implementations outperform their respective standalone assemblers—e.g., Hyper-BOA (megahit) yields 1.3× improvement on both NGA50 and N50, 1.7× improvement on largest alignment, and 1.1× improvement in genome coverage over standalone MEGAHIT. Hyper-BOA implementations also significantly reduce time to solution—e.g., it took 2 h 52 min for the standalone MEGAHIT to assemble the *Betta* genome, whereas this only took 30 min for Hyper-BOA (megahit) (i.e., 5.69× speedup).

Real world experiment

We evaluated Hyper-BOA with real world data. More specifically, we ran Hyper-BOA (megahit) and MEGAHIT on a *D. melanogaster* read set (SRA accession SRX13859210) and compared the results. This is an IlluminaHiSeq4000 dataset (average read length 150*bp*), containing 40.4*M* paired-end reads totaling 6.1*Gbp* in size. Similar to previous studies with real world datasets Li et al. (2015); Chikhi and Rizk (2013), we retained only the reads that align to the reference genome. We used minimap2 Li (2018) for the alignment. Following this step, we were left with 31*M* reads totaling 4.6*G* base pairs. The setting of Hyper-BOA (megahit) is the same as the simulated *D. melanogaster* dataset. The results in Table 5 show Hyper-BOA (megahit) generated an assembly comparable to the standalone MEGAHIT in N50 length and largest alignment length, whereas achieving 1.4× improvement in NGA50 length and 7× improvement in runtime performance.

DISCUSSION

We presented a parallel assembly framework named BOA that leverages a graph/hypergraph partitioning-based approach to enforce a partial ordering and orientation of the input reads. Our experiments using three different off-the-shelf assemblers on a variety of inputs, demonstrate that our Hyper-BOA implementations consistently (and significantly) improve both the assembly quality and performance of the standalone assemblers. This work has opened up further research avenues for future exploration including: a)

Table 5. Assembly quality and runtime performance for the real world read set SRA accession SRX13859210

	NGA50	N50	Largest Alignment (bp)	Genome Coverage %	Missassemblies	Duplication Ratio	Total time Avg. (sec)	Total time Max. (sec)
MEGAHIT	1,566	2,651	**82,462**	74.29	**22**	**1.001**		1,498
Hyper-BOA (megahit)	**2,147**	**2,668**	79,365	**78.62**	226	1.124	227	**233**

Parallel bucketing and partitioning was performed across 256 cores of NERSC Cori (32 nodes × 8 cores per node) with 400 partitions. The runs for baseline (standalone) MEGAHIT were executed on a shared memory node with 32 cores.

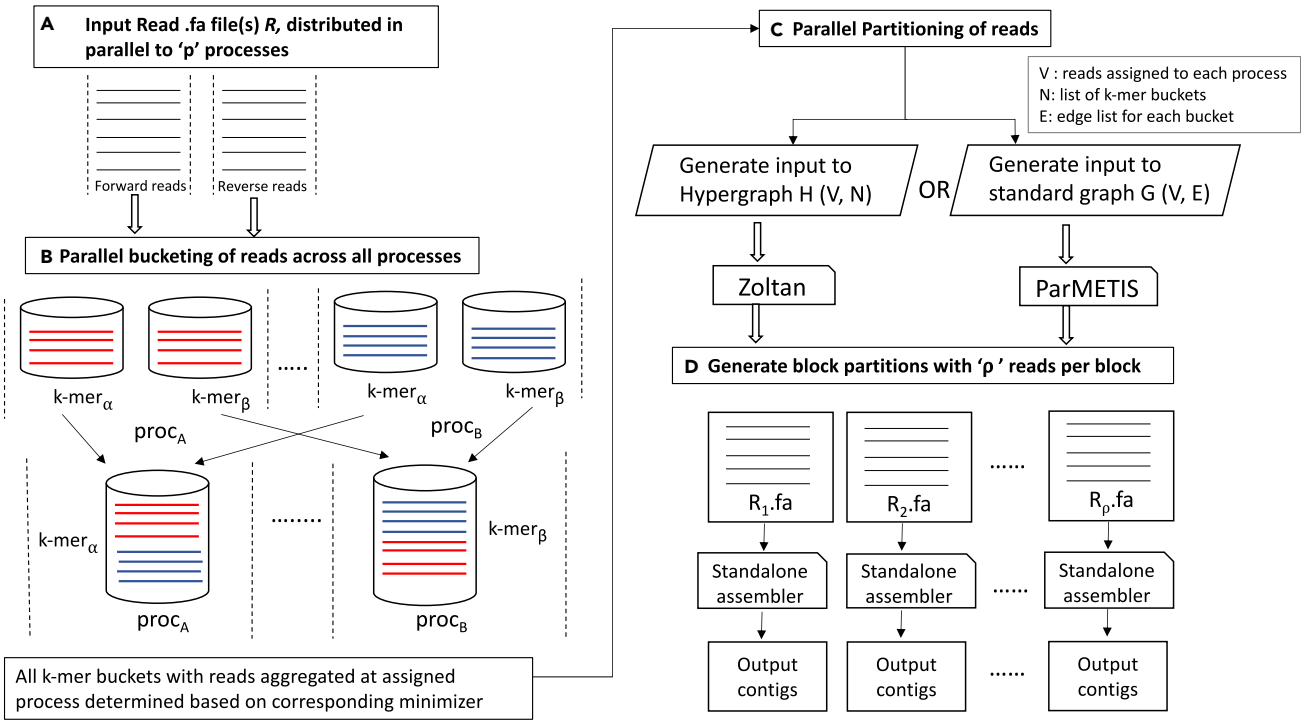


Figure 1. Schematic illustration of the BOA framework

understanding the effect of varying the block (or partition) sizes and modeling that as a space-time-performance quality trade-off problem, b) scaling up to much larger inputs and metagenomic inputs, c) incorporation of long reads as a way to guide the partitioning step, d) extensions of the BOA framework for long read assemblies or hybrid assembly workflows; e) extension of the partitioning-based assembly approach to generate contigs that fall between block boundaries; and f) exploration of alternative partitioning strategies that exploit auxiliary information (e.g., sequence) information.

Limitations of the study

Long reads have become increasing available and have shown to significantly improve assembly quality. As a framework that uses partitioning, BOA can be potentially applied to different read lengths or

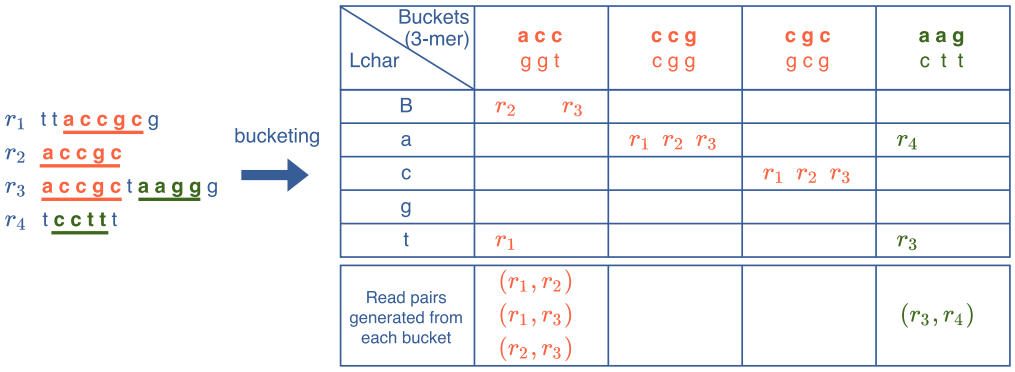


Figure 2. An illustrative example of our pair generation algorithm

On the left are shown four reads and two maximal matches shared among them (shown underlined). Let $k = 3$. The right panel shows a selected subset of buckets relevant to the maximal matches (along each column), and the division of the respective read sets across the different left character sets *Lchar* (along each row). For instance, read r_1 appears in the *Lchar* set for *t* under column *acc* because the *k*-mer*acc* in read r_1 has *t* as its left character. The pairs generated from each bucket are shown in the bottom panel.

technologies. But the original design reported in this article was restricted to short reads, as it is important to first demonstrate the utility of the partitioning idea on the more mature problem of short read assembly. In this regard, there are some non-trivial extensions that have been planned and we believe those extensions (for long reads) would have to be part of a future manuscript.

Another limitation of BOA is the larger memory footprint incurred during the partitioning phase. One of the primary motivations for developing a distributed memory implementation was to be able to scale up the input size by scaling up the available memory in the distributed setting. However, we note that it is also the space required by the graph/hypergraphpartitioner that needs to be factored in while determining memory requirements. To scale to larger inputs on the current evaluation system (64 compute nodes), further optimizations focused on memory will be needed.

Our current implementation does not have the capability of extending the contigs beyond the boundaries of a block, whereas doing so could potentially improve the assembly quality even further. The limitation with the current implementation is because traditional partitioning approaches, by default, generate a disjoint partitioning (of the reads in this case). To grow a contig beyond block boundaries, it will be important to take into account potential overlaps between reads that fall into genomically adjacent partitioned blocks. For this, we would have to sort or at least generate an approximate ordering among the blocks to detect potentially adjacent blocks as per (the unknown) genome. The challenge is to ensure such an ordering is done without introducing a chance/risk of a misassembly. Hence, this is a part of our future work/extension.

STAR★METHODS

Detailed methods are provided in the online version of this paper and include the following:

- KEY RESOURCES TABLE
- RESOURCE AVAILABILITY
  - Lead contact
  - Materials availability
  - Data and code availability
- METHOD DETAILS
  - Preliminaries and notation
  - The BOA assembly framework overview
  - Bucketing algorithm
  - The BOA framework using hypergraph partitioning: Hyper-BOA
  - The BOA framework using graph partitioning: Graph-BOA
  - Graph-BOA and Hyper-BOA
  - Parallelization

ACKNOWLEDGMENTS

The research is supported by U.S. National Science Foundation (awards: CCF 1946752, 1919122, 1919021). This publication describes work performed at the Georgia Institute of Technology and is not associated with Amazon.

AUTHOR CONTRIBUTIONS

Conceptualization, U.V.C. and A.K.; Methodology, Validation, Formal analysis, X.A., P.G., P.K., S.E.K., U.V.C, S.K., P.S., A.S.R., and A.K.; Software, X.A., P.G., P.K., U.V.C., and A.K.; Investigation, X.A., P.G., and P.K.; Resources, A.K.; DataCuration, X.A., P.G., and P.K.; Writing – Original Draft, X.A., P.G., and A.K.; Writing – Review and Editing, X.A., P.G., U.V.C., P.S., and A.K.; Visualization, X.A., and A.K.; Supervision, U.V.C, P.S., and A.K.; Project Administration, A.K.; Funding Acquisition, U.V.C., P.S., and A.K. This author is currently at the National Center for Biotechnology Information (NCBI). The contributions to this work was done during their a_liation with Paci_c Northwest National Laboratory and is not associated with the NCBI.

DECLARATION OF INTERESTS

The authors declare no competing interests.

Received: February 12, 2022
Revised: September 27, 2022
Accepted: September 30, 2022
Published: November 18, 2022

REFERENCES

Al-Okaily, A.A. (2016). Hga: de novo genome assembly method for bacterial genomes using high coverage short sequencing reads. BMC Genom. *17*, 1–11. https://doi.org/10.1186/s12864-016-2515-7.

Chikhi, R., Limasset, A., Jackman, S., Simpson, J.T., and Medvedev, P. (2014). On the representation of de bruijn graphs. In International conference on Research in computational molecular biology, pp. 35–55. https://doi.org/10.1089/cmb.2014.0160.

Chikhi, R., and Rizk, G. (2013). Space-efficient and exact de bruijn graph representation based on a bloom filter. Algorithm Mol. Biol. *8*, 1–9. https://doi.org/10.1186/1748-7188-8-22.

Compeau, P.E.C., Pevzner, P.A., and Tesler, G. (2011). How to apply de bruijn graphs to genome assembly. Nat. Biotechnol. *29*, 987–991. https://doi.org/10.1038/nbt.2023.

Devine, K., Boman, E.G., Heaphy, R., Bisseling, R., and Çatalyürek, U.V. (2006). Parallel hypergraph partitioning for scientific computing. In Proceedings of 20th International Parallel and Distributed Processing Symposium (IPDPS) (IEEE). https://doi.org/10.1109/ipdps.2006.1639359.

Duke University School of Medicine, Last date accessed: November 2021. NCBI GenBank.https://www.ncbi.nlm.nih.gov/genbank/.

Garey, M.R., and Johnson, D.S. (1979). Computers and Intractability*volume 174* (freeman San Francisco). https://doi.org/10.1016/0045-7949(87)90014-9.

Garey, M.R., Johnson, D.S., and Stockmeyer, L. (1974). Some simplified NP-complete problems. In Proceedings of the sixth annual ACM symposium on Theory of computing, pp. 47–63. https://doi.org/10.1016/0304-3975(76)90059-1.

Gurevich, A., Saveliev, V., Vyahhi, N., and Tesler, G. (2013). Quast: quality assessment tool for genome assemblies. Bioinformatics *29*, 1072–1075. https://doi.org/10.1093/bioinformatics/btt086.

Hendrickson, B., and Kolda, T.G. (2000). Graph partitioning models for parallel computing. Parallel Comput. *26*, 1519–1534. https://doi.org/10.1016/s0167-8191(00)00048-x.

Huang, W., Li, L., Myers, J.R., and Marth, G.T. (2012). Art: a next-generation sequencing read simulator. Bioinformatics *28*, 593–594. https://doi.org/10.1371/journal.pone.0090581.

Jammula, N., Chockalingam, S.P., and Aluru, S. (2017). Distributed memory partitioning of high-throughput sequencing datasets for enabling parallel genomics analyses. In Proceedings of the 8th ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics, pp. 417–424. https://doi.org/10.1145/3107411.3107491.

Karypis, G., Schloegel, K., and Kumar, V. (1997). Parmetis: Parallel Graph Partitioning and Sparse Matrix Ordering Library*48*, pp. 71–95. https://doi.org/10.1006/jpdc.1997.1403.

Lengauer, T. (2012). Combinatorial Algorithms for Integrated Circuit Layout*volume 21* (Springer Science & Business Media). https://doi.org/10.7155/jgaa.00447.

Li, Z., Chen, Y., Mu, D., Yuan, J., Shi, Y., Zhang, H., Gan, J., Li, N., Hu, X., Liu, B., et al. (2012). Comparison of the two major classes of assembly algorithms: overlap–layout–consensus and de-bruijn-graph. Brief. Funct. Genom. *11*, 25–37. https://doi.org/10.1108/aa-02-2019-0031.

Li, D., Liu, C.M., Luo, R., Sadakane, K., and Lam, T.W. (2015). Megahit: an ultra-fast single-node solution for large and complex metagenomics assembly via succinct de bruijn graph. Bioinformatics *31*, 1674–1676. https://doi.org/10.1093/bioinformatics/btv033.

Li, H. (2018). Minimap2: pairwise alignment for nucleotide sequences. Bioinformatics *34*, 3094–3100. https://doi.org/10.1093/bioinformatics/bty191.

Medvedev, P., and Pop, M. (2021). What do Eulerian and Hamiltonian cycles have to do with genome assembly? PLoS Comput. Biol. *17*, e1008928. https://doi.org/10.1371/journal.pcbi.1008928.

MPI Forum (2020). MPI: A Message-Passing Interface Standard. 2020 Draft Specification.Technical Report (Univ. of Tennessee). Note: This is a MPI-4 Draft Specification. https://doi.org/10.22443/rms.emc2020.425.

Pell, J., Hintze, A., Canino-Koning, R., Howe, A., Tiedje, J.M., and Brown, C.T. (2012). Scaling metagenome sequence assembly with probabilistic de bruijn graphs. Proc. Natl. Acad. Sci. USA *109*, 13272–13277. https://doi.org/10.1073/pnas.1121464109.

Peng, Y., Leung, H.C.M., Yiu, S.M., and Chin, F.Y.L. (2012). IDBA-UD: a de novo assembler for single-cell and metagenomic sequencing data with highly uneven depth. Bioinformatics *28*, 1420–1428. https://doi.org/10.1093/bioinformatics/bts174.

Pevzner, P.A., Tang, H., and Waterman, M.S. (2001). An Eulerian path approach to dna fragment assembly. Proc. Natl. Acad. Sci. USA *98*, 9748–9753. https://doi.org/10.1073/pnas.171285098.

Pop, M. (2009). Genome assembly reborn: recent computational challenges. Briefings Bioinf. *10*, 354–366. https://doi.org/10.1093/bib/bbp026.

STAR★METHODS

KEY RESOURCES TABLE

REAGENT or RESOURCE	SOURCE	IDENTIFIER
Deposited data		
C. elegans	*C. elegans* Sequencing Consortium	NCBI GenBank assembly accession GCA_000002985.3
D. melanogaster	The FlyBase Consortium/Berkeley Drosophila Genome Project/Celera Genomics	NCBI GenBank assembly accession GCA_000001215.4
Human chr 7	T2T Consortium	NCBI GenBank assembly accession GCA_009914755.2, GenBank sequence CP068271.1
Human chr 8	T2T Consortium	NCBI GenBank assembly accession GCA_009914755.2, GenBank sequence CP068270.1
Human chr 8	T2T Consortium	NCBI GenBank assembly accession GCA_009914755.2, GenBank sequence CP068268.1
Maize chr 10	MaizeGDB	NCBI GenBank assembly accession GCA_902167145.1, GenBank sequence LR618883.1
Bettasplendens	BGI	NCBI GenBank assembly accession GCA_003650155.1
Real world read set	Duke University	NCBI SRA, accession numberSRX13859210
Software and algorithms		
ART_Illumina v 2.8.5	Huang et al. (2012)	RRID:SCR_006538; https://www.niehs.nih.gov/research/resources/assets/docs/artbinmountrainier2016.06.05linux64.tgz
Megahit v 1.2.9	Li et al. (2015)	RRID:SCR_018551; https://github.com/voutcn/megahit/releases/download/v1.2.9/MEGAHIT-1.2.9-Linux-x86_64-static.tar.gz
Minia v 0.0.102	Chikhi and Rizk (2013)	RRID:SCR_004986; https://github.com/GATB/minia/releases/download/v0.0.102/minia-v0.0.102-bin-Linux.tar.gz
IDBA v 1.1.3	Peng et al. (2012)	RRID:SCR_011912; https://github.com/loneknightpy/idba/releases/download/1.1.3/idba-1.1.3.tar.gz
ParMetis v 4.0.3	Karypis et al. (1997)	http://glaros.dtc.umn.edu/gkhome/fetch/sw/parmetis/parmetis-4.0.3.tar.gz
Zoltan v 3.83	Devine et al. (2006)	https://github.com/sandialabs/Zoltan/archive/refs/tags/v3.83.tar.gz
QUAST v 5.1.0rc1	Gurevich et al. (2013)	RRID:SCR_001228; https://github.com/ablab/quast
Minimap2 v 2.24	Li (2018)	RRID:SCR_018550; https://github.com/lh3/minimap2/releases/download/v2.24/minimap2-2.24_x64-linux.tar.bz2
BOA v0	This work	https://github.com/GT-TDAlab/BOA

RESOURCE AVAILABILITY

Lead contact

Further information and requests for resources should be directed to and will be fulfilled by the Lead Contact, Xiaojing An (anxiaojing@gatech.edu).

Materials availability

This study did not generate new unique reagents.

Data and code availability

- The paper analyzes existing, currently available data. The accession identifiers for the datasets are listed in the key resources table.

- BOA is publicly available online from https://github.com/GT-TDAlab/BOA.
- Any additional information required to reanalyze the data reported in this paper is available from the lead contact upon request.

METHOD DETAILS

Preliminaries and notation

Strings and genome assembly

Let s denote an arbitrary string over a fixed alphabet Σ, and let $|s|$ denote the length of the string. Let $s[i, j]$ denote the substring of s starting at index i and ending at index j. As a convention, we index strings from 1, and the i^{th} character of s is denoted by $s[i]$. A k-mer is a (sub)string of length k.

Given a substring $s[i, j]$ of s, we refer to the character immediately preceding the substring in s to be its "left-character" or *lchar* (if one exists). More specifically, $lchar_i = s[i-1]$ if $1 < i \leq |s|$, and if $i = 1$, then $lchar_i = B$, where $B \notin \Sigma$ is used to represent a blank symbol.

The input to genome assembly is a set of n reads (denoted by $\mathcal{R}$). Each read is a string over the alphabet $\Sigma = \{a, c, g, t\}$. We denote the reverse complemented form of a read r as $rc(r)$. If reads are generated with paired-end information, then the two reads of the same pair are assigned consecutive read IDs i and $i+1$, so that the odd read ID corresponds to the forward strand read and the even read ID corresponds to the reverse strand read. We denote the set of all forward (alternatively, reverse) reads as $\mathcal{R}_f$ (alternatively, $\mathcal{R}_r$). Note that $\mathcal{R} = \mathcal{R}_f \cup \mathcal{R}_r$, and $|R_f| = |R_r| = \frac{n}{2}$.

Graph partitioning

A undirected *graph* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is defined by a set of vertices $\mathcal{V}$ and a set of edges $\mathcal{E}$. An edge $e_{i,j}$ is a pair of distinct vertices, i.e., $e_{ij} = \{v_i, v_j\}, v_i \in \mathcal{V}, v_j \in \mathcal{V}$. The degree d_i of a vertex v_i is defined as the number of edges incident to that vertex. Weights and costs can be assigned to vertices and edges. $\mathcal{W}$ is used to represent the weight assignment for vertices, where w_i is the weight for the vertex $v_i \in \mathcal{V}$. $\mathcal{C}$ is the cost assignment for edges, where c_{ij} represents the cost for the edge $e_{ij} \in \mathcal{E}$.

A K-way partition of $\mathcal{G}$, $\Pi = \{\mathcal{P}_1, \cdots \mathcal{P}_K\}$, places each vertex of the graph into a *part*. More concretely, Π is a K-way partition if each part $\mathcal{P}_i$ is a non-empty subset of $\mathcal{V}$, each pair of parts is disjoint, i.e., $\mathcal{P}_i \cap \mathcal{P}_j = \varnothing$ for all $1 \leq i \neq j \leq K$, and the union of all parts recovers $\mathcal{V}$, that is $\cup_{1 \leq i \leq K} \mathcal{P}_i = \mathcal{V}$. For a K-way partition Π, an edge $e_{ij} = \{v_i, v_j\}$ is called external (or *cut*) if $v_i \in \mathcal{P}_a$, $v_j \in \mathcal{P}_b$ with $a \neq b$, otherwise called internal (or *uncut*). $\mathcal{E}_E$ is used to represent the set of all external edges. The cost (or *cutsize*) χ of Π is defined as: A K-way partition, Π, is called balanced if the following holds:

$$\forall i \in \{1, \ldots, K\}, \sum_{v_j \in \mathcal{P}_i} w_j \leq (1 + \varepsilon) W_{avg} \qquad \text{(Equation 1)}$$

where, $W_{avg} = (\sum_{v_j \in \mathcal{V}} w_j)/K$, and ε is a given maximum imbalance ratio.

The graph partitioning problem is defined as follows: given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, vertex weight and edge cost assignments $\mathcal{W}$ and $\mathcal{C}$, a part number requirement K, and the maximum allowed imbalance ratio ε, find a balanced K-way partitioning that minimizes the cost. The graph partitioning problem is known to be NP-hard Garey and Johnson (1979), even for seemingly easier problems such as uniform weighted bi-partitioning Garey et al. (1974).

Hypergraph partitioning

A *hypergraph* $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ contains a set of vertices, $\mathcal{V}$, and a set of nets (hyperedges), $\mathcal{N}$. Hypergraph is a generalization of graph where each hyperedge can connect more than two vertices, i.e., a *net* $n_i \in \mathcal{N}$ is a subset of vertices $\mathcal{V}$. The vertices in a net are called its *pins*, represented by $pins[n_i]$; and the size of the net is the number of its pins. The number of nets incident on v_i represents the vertex degree d_i. Similar with graphs, we use $\mathcal{W}$ and $\mathcal{C}$ as vertex weight and net cost assignments, w_i to represent the weight of a vertex $v_i \in \mathcal{V}$ and c_j to represent the cost of a net $n_j \in \mathcal{N}$.

The K-way partitioning of a hypergraph is similar to that of a standard graph. The main difference comes from the definition of partitioning cost. A net is connected to a part if at least one of its pins is in that part. The connectivity set Λ_j of net n_j is all the parts that the net connects to. The size of Λ_j is denoted λ_j, i.e. $\lambda_j = |\Lambda_j|$. A net n_j is external (or *cut*), if it connects to more than one part, i.e. $\lambda_j > 1$, otherwise, the net is called internal (or *uncut*). The set of all external nets for a partition Π is represented as $\mathcal{N}_E$. There are multiple definitions of cost χ of a partitioning Π, in this work we will use connectivity -1 metric, defined as: The hypergraph partitioning problem is known to be NP-hard as well Lengauer (2012).

The BOA assembly framework overview

The BOA framework hinges on the key idea of block partitioning the reads so that each *block* is expected to contain reads from neighboring regions of the (unknown) target genome. This blocking mechanism is a proxy to obtaining a fully ordered sequence of reads. After block partitioning, each block can be assembled using any standalone assembler of choice, and the combined set of contigs generated across all the blocks represent the final output assembly. This partitioning-based strategy has several advantages:

- The quality of the output assembly can potentially see improvements if the block partitioning of reads is faithful to the origins of the reads along the genome (i.e., reads mapping to neighboring genomic regions are assigned to the same block, while unrelated reads are kept separated across blocks).
- From the performance standpoint, block partitioning can provide significant leverage in controlling the degree of parallelism as each block is independently processed.
- Finally, the BOA framework is oblivious to and allows the use of any standalone assembler of choice downstream. Instead, the framework shifts the focus on keeping related reads together, unrelated reads separate, and keeping the block sizes reasonably small so as to enable fast parallel assemblies.

Figure 1 illustrates the BOA framework with its different components. In what follows, we describe these major components. In particular, we describe two instantiations of the framework—one using classical graph partitioning (Section the BOA framework using graph partitioning: graph-BOA) and another using hypergraph partitioning (Section the BOA framework using hypergraph partitioning: hyper-BOA). Both the initial bucketing step and final assembly step are common to both instantiations.

Bucketing algorithm

Given the set of reads $\mathcal{R}$, the bucketing algorithm computes set of buckets $\mathcal{B}$, where each bucket $b \in \mathcal{B}$ corresponds to a k-mer in $\mathcal{R}$. The bucketing algorithm assigns the reads in $\mathcal{R}$ to at most $|\Sigma|^k$ buckets, for a fixed length $k > 0$. We define a *bucket* for each distinct k-mer present in $\mathcal{R}$. In particular, a read r is assigned to all buckets corresponding to the list of k-mers it contains. Therefore, a bucket is simply a set of read IDs with that k-mer. To account for bidirectionality of reads, we take the lexicographically smaller variant of each k-mer and assign reads to that bucket. This ensures that the read is either present in the bucket corresponding to the k-mer in its direct form or its reverse complemented form (but not both).

Let $\mathcal{B}$ denote the collection of all buckets generated in this process, and b denote an arbitrary member of $\mathcal{B}$. Note that each $b \subseteq \mathcal{R}$. We use $kmer(b)$ to denote the k-mer that defines bucket b. Note that it is possible for buckets to intersect in reads (given that the same read could have multiple distinct k-mers).

The BOA framework using hypergraph partitioning: Hyper-BOA

Hyper-BOA models the multi-way interaction between reads and buckets using a hypergraph. We describe this hypergraph-based model first because it naturally follows from the bucketing step.

Input to Hyper-BOA is the set of buckets $\mathcal{B}$ and output is the *read-bucket* hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$, where reads are represented as vertices, and buckets as nets. Thisstep produces a partitioning Π of $\mathcal{H}$, which is a partitioning of reads. Each bucket $b \in \mathcal{B}$ contains the subset of all reads in $\mathcal{R}$ that share the same k-mer (either in the direct or reverse complemented form). With the hypothesis that this is a necessary—but not sufficient—condition for reads originating in the same region of the target genome, we construct a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ for two possible scenarios.

No paired-end information available

If the input $\mathcal{R}$ does *not* contain paired-end information, then we construct a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ such that $\mathcal{V} = \mathcal{R}$ and $\mathcal{N} = \mathcal{B}$. In other words, we initialize a hypergraph where each read is represented by a vertex and each bucket by a net. The pins of a net correspond to all the reads that are part of the corresponding bucket. Since each vertex is a read and the subsequent assembly workload is not expected to vary with similar sized reads, we assign unit weights to each vertex. One can use a cost function to represent *importance* of a *k*-mer, but for this initial work we simply treat each *k*-mer equally and thus assign unit costs to nets.

Paired-end information available

If the input read set $\mathcal{R}$ *contains* paired-end information, then we construct our *read-bucket* hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ after post-processing the buckets as follows. Recall that for paired-end reads, the two reads of a given pair are assigned consecutive IDs i (odd) and $i+1$ (even). While these two reads of the pair can take part in different sets of buckets, it is desirable to assign these two reads to the same block at the end of partitioning, so that the subsequent assembly step can use the paired-end information. To force this block assignment during partitioning, we *fuse* the two reads into a single vertex in the hypergraph—i.e., the reads i and $i+1$ of a pair are both mapped to the same vertex in the hypergraph $\mathcal{H}$, identified by vertex $\lceil \frac{i}{2} \rceil$ (same as $\lceil \frac{i+1}{2} \rceil$). This can be achieved by simply scanning the list of read IDs in each bucket and renumbering each using the above ceiling function (In our implementation, we actually renumber the read IDs as they are *entered* into their buckets, so that a second pass is unnecessary). Consequently, the new hypergraph $\mathcal{H}$ will contain exactly $\frac{n}{2}$ vertices. The set of nets $\mathcal{N}$ is the updated set of buckets $\mathcal{B}$ with the re-numbered read IDs (as its pins). Each vertex and each net are assigned unit weights.

Partitioning

Once the hypergraph $\mathcal{H}$ is constructed, we call the partition function on $\mathcal{H}$ (described in Section preliminaries and notation) using the Zoltan hypergraph partitioner Devine et al. (2006). Partitioning takes as an input parameter the number of output parts K. However, instead of fixing the number of parts (or equivalently, output *blocks*) arbitrarily, we set a target for the output block size, i.e., for the number of reads per part, denoted by ρ. Intuitively, since each output block is input to a standalone assembler, it is important to keep related reads together so that contigs have a chance to grow long (and not fragment the assembly). However, if the block size becomes too large, then it may not only start including unrelated reads (from far regions of the genome) but also would have a negative impact on the runtime performance. (Note that a single block configuration ($K = 1$) is equivalent to running the standalone assembler on the entire input $\mathcal{R}$.) Therefore, we set a target ρ for the number of reads per block, and using ρ determine K ($\approx \lceil \frac{n}{\rho} \rceil$).

To determine an appropriate ρ, we can use the longest contigs produced by state-of-the-art assemblers as a lower-bound. The idea is to set a target for ρ so that the contigs produced from each block have an opportunity to grow into a contig that is longer than this baseline length. For instance, a block with 100K reads can produce only a contig that is as long as ~100Kbp (assuming 100bp read length and 100× genome sequencing coverage). So if our goal is to surpass this baseline, then the block size has to reflect that—e.g., a constant factor more than that baseline. Setting a high target for ρ as described above is not a guarantee for qualitative improvement, but it provides a chance (to the per-block standalone assemblers). This approach enables empirically calibrating the block size for assembly quality.

One last parameter in partitioning is the balance criterion. To achieve a similar workload across all the individual block assembler runs, we prefer roughly similar sized blocks. However, keeping this very tight might unnecessarily constrain the related reads that will need go into same part. To strike a balance between these two goals we use a balanced constraint of $\varepsilon = 1\%$ (see Equation 1).

The BOA framework using graph partitioning: Graph-BOA

Graph-BOA models the interaction among reads using a graph. Input to Graph-BOA is the set of buckets $\mathcal{B}$ and output is the *read-overlap graph* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where reads are represented as vertices, and edges represent *alignment-free*, *exact match* overlaps between pairs of reads, identified by bucketing phase. This phase produces a partitioning Π of $\mathcal{G}$, which is a partitioning of reads.

Given a set $\mathcal{R}$ of n input reads, we construct a read-overlap graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V} = \mathcal{R}$ and $\{r_i, r_j\} \in \mathcal{E}$ if the two reads r_i and r_j share at least one maximal match (A maximal match is a nonempty exact match

between two strings that cannot be extended in either direction) of length $\geq k$, for some integer constant $k > 0$. In other words, the set of edges $\mathcal{E}$ is generated by enumerating all pairs of reads sharing at least one maximal match α of length $\geq k$. Let $\mathcal{P}$ denote the set of pairs, given by:

$$\mathcal{P} = \{\{r_i, r_j\} \,|\, r_i, r_j \in \mathcal{R},\ i \neq j,\ \text{and } \exists \text{ a maximal match of length } \geq k \text{ between } r_i \text{ and } r_j\}$$

For example, two reads $r_1 = cagcca$ and $r_2 = tgagcc$ share substring $agcc$ as a maximal match, and if $k = 3$ there will exist an edge between the nodes corresponding to r_1 and r_2 in $\mathcal{G}$. The focus on maximal matches is due to the following performance consideration. While buckets are defined based on k-mers, two reads that share a longer exact match of length t could appear in up to $t - k + 1$ distinct buckets. Instead of detecting the same pair $\{r_i, r_j\}$ multiple times in those many buckets, our algorithm detects it only once due to the leftmost common k-mer in the maximal matching. In the above example of r_1 and r_2, the pair is detected due to the leftmost k-mer agc in the maximal match.

Note that once all pairs are generated, each bucket b containing m reads would have effectively contributed $\binom{m}{2}$ pairs to $\mathcal{P}$—i.e., a clique of size m in $\mathcal{G}$. The above maximal match trick is mainly to avoid duplicate detection of any edge in the clique.

Pair generation

To generate $\mathcal{P}$ using all the buckets, our algorithm deploys a two-step strategy as described below. Intuitively, we use the two characters (if present) flanking the maximal match to its left and right. A maximal match is a substring that is both left-maximal (i.e., left characters on both reads mismatch) and right-maximal (i.e., right characters on both reads mismatch). The only exception is when there is no flanking character on any one of the read. In such a case, we use the blank character B for maximality. Note that two reads that have B as their respective left characters are left-maximal. Our algorithm exploits the bucketing information for right maximality and instead checks only for left maximality (while still guaranteeing maximality). The details of the algorithm are described below.

a) *Left-maximality*: Consider the read collection covered by bucket b. For each read $r \in b$, let $\psi(r, b)$ denote the set of suffix positions in read r that have $kmer(b)$ as their prefix; and let $Lchars(r, b) \in \Sigma \cup \{B\}$ denote the set of all characters that immediately precede those suffix positions in r. Using $Lchars(r, b)$, we generate a bit vector L_r of length $|\Sigma| + 1$ as follows:

$$L_r[x] = \begin{cases} 1, & \text{if } x \in Lchars(r, b) \\ 0, & otherwise \end{cases}$$

Example 1. Consider a read $r = accttacc$ and the bucket b for 2-mer ac. Then $\psi(r, b)$ are the suffix positions $\{1, 6\}$ and the corresponding left characters are $r[0] = B$ and $r[5] = t$ (i.e., $Lchars(r, b) = \{B, t\}$). Therefore the bit vector L_r for the bucket corresponding 2-mer ac is $[1, 0, 0, 0, 1]$ for left characters $[B, a, c, g, t]$ respectively.

Remark 1. As noted in §7.3.3, k-mers are indexed by their lexicographically smaller variant to account for bidirectionality. If a given k-mer in a read r is *not* in its lexicographically smaller form, we use character *following* the k-mer in its complemented form, for the purpose of left character lists ($Lchars$). This is to capture the reversal in direction.

Example 2. Consider a read $r = atgcgttg$ and the bucket b for 2-mer tg (or equivalently, its smaller form ca). Then, the $Lchars(r, b) = \{g, B\}$ as they are the corresponding left characters in the reverse complemented form for r. Therefore the bit vector L_r for the bucket corresponding 2-mer tg (or equivalently, ca) is $[1, 0, 0, 1, 0]$ for left characters $[B, a, c, g, t]$ respectively.

b) *Pairing*: Subsequently, we use the L_r-arrays for all reads $r \in b$ to generate the pairs of reads from that bucket b. The set of pairs contributed by bucket b, denoted by $\mathcal{P}(b)$, is given by:

$$\mathcal{P}(b) = \left\{\{r_i, r_j\} \,|\, r_i, r_j \in b,\ \exists\, x \in \Sigma \cup \{B\} \text{ s.t.} L_{r_i}[x] \oplus L_{r_j}[x] = 1 \text{ or } L_{r_i}[B] \vee L_{r_j}[B] = 1\right\}$$

Here, $\oplus$ and $\vee$ are the bitwise XOR and OR operators respectively. Intuitively, a pair of reads is generated at a bucket b only if there exists a pair of suffixes in those reads that differ in their left-characters (thereby guaranteeing left-maximality of the match detected). Note that right-maximality of the match in a pair detected is implicitly guaranteed as the suffixes in those two reads will have to eventually differentiate at some point past the k-mer prefix. Therefore this algorithm is able to report only one instance of a read pair $\{r_i, r_j\}$ for the leftmost matching k-mer of a maximal match.

Example 3. Figure 2 presents an example to illustrate our pair generation algorithm. This example shows two maximal matches (*accgc* and *aagg*) appearing among four reads. As highlighted in the orange, r_1,r_2 and r_3 share the maximum matching *accgc*. This match contains multiple 3-mers: *acc*, *ccg* and *cgc*, and therefore the corresponding reads will appear in all those buckets (shown in orange colored buckets in the table). The rows show the left character lists (*Lchar*) that each read will appear within a given bucket. Our pair generation algorithm will generate pairs from each bucket by performing a cross-product across the different *Lchar* lists. The only exception is the *B* list, where reads appearing in that list are left-maximal and so will yield pairs. Pairs generated from each bucket are shown in the bottom panel. The second maximal match in the example *aagg* (in green), shows a case where the pairing could happen between a read and the reverse complement of another read. In this example, reads r_3 and the reverse complement of r_4 share the maximal match *aagg*, and therefore will be generated from the bucket corresponding to *aag* that is the lexicographically smaller of the two variants (*aag*, *ctt*).

BOA has an optional modification dealing with a specific boundary case where edges may be missed between reads, at a cost of increased memory and runtime. Specifically, consider when:

- The maximum matching α has length > k.
- The leftmost k-mer in α is lexicographically larger than its reverse complement, and.
- The rightmost k-mer is lexicographically smaller than its reverse complement.

In this case, for the k-mers in both ends, the leftmost character recorded for the corresponding k-mer in the read is a character within the maximum matching α. BOA does not look outside of the maximum matching in the read to be able to recognize the end of the maximum matching and generate the read pair.

Example 4. Consider the example of two reads $r_1 = r_2 = ctac$ and $k = 2$. Read r_1 will be in the following buckets: *ag*, *ta*, *ac* with leftmost character as *t*,*c*,*t* respectively. The assignment is the same with r_2. Thus, the baseline algorithm would miss detecting the pair (r_1,r_2).

The algorithm can be easily modified to avoid this boundary case. More specifically, the method can store *both* the leftmost and rightmost characters for each k-mer in the following way: each bucket has two groups of sub-buckets: *a*–,*t*–,*g*–,*c*–,*B*– for leftmost character and $-a$, $-t$, $-g$, $-c$, $-B$ for rightmost character. Edges are generated only within each group of sub-buckets and not among the groups. Note that this solution comes with a slight increase in cost: each length k or greater maximum matching between two reads will produce two of the same read pairs even if the length of the maximum matching is k. For this reason and as BOA is a heuristic, we have not implemented this change in practice but note that in theory we can address it.

Similar to Hyper-BOA, we assign unit weights to vertices and edges, and create one of two different variants of the read-overlap graph $\mathcal{G}$ depending on whether paired-end read information is available or not. More specifically, if paired-end information is available, then we follow the same fuse strategy described underHyper-BOA, by representing both reads of a pair by a single vertex in $\mathcal{V}$ of $\mathcal{G}$. This is achieved by renumbering the read IDs within each input bucket *b*prior to generating pairs.

Partitioning

Once the read-overlap graph $\mathcal{G}$ is constructed, then we call the partition function on $\mathcal{G}$ (described in Section preliminaries and notation) using the ParMETIS graph partitioner Karypis et al. (1997). Here again, we use the number of reads per output block (ρ) as our guide to determine the number of blocks K and set the balanced constraint ε as 1%.

Graph-BOA and Hyper-BOA

There are a few important connections as well as differences between the graph-based approach (Graph-BOA) and hypergraph-based approach (Hyper-BOA) within our BOA framework that are worth noting.

First, from the assembly problem formulation standpoint, Graph-BOA is very similar to the OLC assembler model with the key difference being the "overlaps" in the read-overlap graph are detected using light-weight exact match-based criteria (as described in the bucketing step). Therefore our approach is alignment-free. The read-bucket hypergraphs we construct under Hyper-BOA, are also alignment-free. Furthermore, they can be viewed as a generalization of the read-overlap graphs (from edges to nets; i.e., read pairs to read subsets).

Secondly, from a method standpoint, intuitively both graph and hypergraph approaches try to put reads that are strongly connected to each other into the same part. In hypergraph model, each bucket (i.e., *k*-mer) is uniquely represented by a net. If two reads share multiple *k*-mers, they will be incident in multiple nets, hence representing how strong their connection is. In the graph model, each edge does not represent a unique relation. An edge between two reads might come from different overlaps (or buckets). Hence, one would need an aggregation function to represent that accurately. In our current implementation of Graph-BOA, the edges established between any two reads are unweighted (or equivalently, unit weight). This is in contrast with alignment-based OLC assemblers, which typically use an alignment-based weight along an edge. While edge weights would help guide partitioning decisions, for Graph-BOA there is a tradeoff with performance. One approach to calculate an edge weight between a pair of reads is based on the *length* of maximal matches that led to detection of that edge. However, in our pair generation algorithm, we only *detect* the presence of a maximal match for pairing two reads, without explicitly determining the match itself or its length (as it will become more expensive to compute the matches). An alternative strategy is to count the number of buckets a pair of reads co-occurs to use as the corresponding edge weight. However, this also implies detecting and storing a count for each pair—which could become expensive both for runtime and memory. As a compromise, we have used an unweighted representation for Graph-BOA.

Another point of difference between Hyper-BOA and Graph-BOA is their space and run time complexities. For Hyper-BOA, the *k*-mer based buckets are used to construct the hypergraph. Every bucket with say *m* distinct reads in it, induces a net with *m* pins. Whereas, under Graph-BOA, extra computation is needed to establish pairwise connections between reads as described in Section the BOA framework using graph partitioning: graph-BOA—i.e., every bucket with *m* reads contributes $\binom{m}{2}$ edges. This leads to higher memory usage for Graph-BOA. For example, in case of *C.elegans*, the peak memory usage per MPI rank for Graph-BOA in the bucketing phase is 8.3 GB in comparison to 5.3 GB for Hyper-BOA. While in the partitioning phase, Graph-BOA is much lighter in runtime than Hyper-BOA as shown in Section runtime performance evaluation.

Parallelization

The BOA pipeline is comprised of three phases:

1) *Parallel Bucketing*: In this step, the algorithm first loads the input FASTA file(s) in a distributed manner such that each process receives roughly the same amount of sequence data $\approx \frac{|R|}{p}$, where p is the total number of processes. This is achieved by each process loading a chunk of reads using MPI-IO functions MPIForum (2020), such that no read is split among processes. Each read is assigned a distinct read id. We use MPI_Scan to determine the read offset at each process. Next we generate *k*-mers by sliding a window of length k (k = 31 in our experiments) over each read, as elaborated in the bucketing step (Section bucketing algorithm). For parallel bucketing, an owner process that collects the read IDs for each bucket is assigned. To identify the owner, we use an approach based on *minimizers* Chikhi et al. (2014). In particular, for each *k*-mer bucket, a minimizer of length l (l<k; l = 8 in our experiments) is identified. We use the least frequently occurring *l*-mer within that *k*-mer as the minimizer. Subsequently, a hash function is used to map the minimizer to its owner process. The idea of using minimizers for this assignment step is to increase the probability that adjacent *k*-mers in a read are assigned the same owning process for the corresponding buckets (thereby reducing communication latency). Collective aggregation of the read IDs corresponding to each

bucket is carried out through an MPI_Alltoallv primitive MPIForum (2020). Any bucket with 200 or more distinct reads represented is pruned. This pruning step is to account for over-representation in the buckets corresponding to repetitive regions.

2) *Parallel Partitioning*: In this step at first we generate the input read-overlap graph ($\mathcal{G}$) or read-bucket hypergraph ($\mathcal{H}$), for Graph-BOA or Hyper-BOA, respectively. For Hyper-BOA, we provide Zoltan's hypergraph generating function, a list of all distinct sorted read IDs for each k-mer bucket assigned to a process. For Graph-BOA, each process enumerates edges between pairs of reads sharing at least one maximal match (Section the BOA framework using graph partitioning: graph-BOA) in parallel and then sends the edge lists to the owner processes of the vertices through MPI_Alltoallv. We provide ParMETIS the CSR (Compressed Sparse Row) format graph. We then call the partitioning function, providing as input the generated hypergraph or graph, the number of block partitions K and the balanced constraint ε.

3) *Assembly*: The final phase of the pipeline takes the K partitions generated by the partitioner and launches K concurrent assembly instances using a standalone assembler on each of the K parts (or equivalently, blocks).

Our BOA framework is available at https://github.com/GT-TDAlab/BOA.