diBELLA: Distributed Long Read to Long Read Alignment

Marquita Ellis[1,2], Giulia Guidi[1,2], Aydın Buluç[1,2], Leonid Oliker[2], Katherine Yelick[1,2]
[1]University of California at Berkeley [2]Lawrence Berkeley National Laboratory
{mellis,gguidi,abuluc,loliker,yelick}@lbl.gov

ABSTRACT

We present a parallel algorithm and scalable implementation for genome analysis, specifically the problem of finding overlaps and alignments for data from "third generation" long read sequencers [29]. While long sequences of DNA offer enormous advantages for biological analysis and insight, current long read sequencing instruments have high error rates and therefore require different approaches to analysis than their short read counterparts. Our work focuses on an efficient distributed-memory parallelization of an accurate single-node algorithm for overlapping and aligning long reads. We achieve scalability of this irregular algorithm by addressing the competing issues of increasing parallelism, minimizing communication, constraining the memory footprint, and ensuring good load balance. The resulting application, diBELLA, is the first distributed memory overlapper and aligner specifically designed for long reads and parallel scalability. We describe and present analyses for high level design trade-offs and conduct an extensive empirical analysis that compares performance characteristics across state-of-the-art HPC systems as well as a commercial cloud architectures, highlighting the advantages of state-of-the-art network technologies.

KEYWORDS

genomics, bioinformatics, high performance computing, performance analysis, distributed data structures, cloud computing

ACM Reference Format:
Marquita Ellis[1,2], Giulia Guidi[1,2], Aydın Buluç[1,2], Leonid Oliker[2], Katherine Yelick[1,2]. 2019. diBELLA: Distributed Long Read to Long Read Alignment. In *48th International Conference on Parallel Processing (ICPP 2019), August 5–8, 2019, Kyoto, Japan.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3337821.3337919

1 INTRODUCTION

The improved quality, cost, and throughput of DNA sequencing technologies over the past decades has shifted the primary biological challenge from measuring the genome to analyzing the explosion in genomic data, which has far exceeded the growth in computing capabilities. Yet some of the most complex algorithms for genome analysis are typically run on shared memory machines, limiting parallel scalability, and can run for days or even weeks on large data sets. Here we present a parallel algorithm and implementation for one such problem involving the latest sequencing technologies and a variety of parallel platforms.

Because DNA sequencing technologies are unable to read the whole genome in a single run, they return a large amount of short DNA fragments, called *reads*. A read set contains redundant information as each region of the genome is sequenced multiple times (referred to as *depth*, or *coverage*) to account for sequencer errors. These reads are typically assembled together to form longer genomic regions. Current sequencing technologies can be divided in two main categories based on the read length: "short-read" and "long-read" sequencers. Short-read technologies have very low error rates (well under 1%) but the reads are only 100 to 300 base pairs and they cannot resolve repeated regions of the genome longer than those reads [28, 31]. Long-read technologies, including Pacific Biosciences and Oxford Nanopore, generate reads with an average length over $10,000$ base pairs (bps), but they have error rates from 5% to 35%.

One of the biggest challenges for the analysis of sequencing data is *de novo* assembly [36], which is the process of eliminating errors and assembling a more complete version of the genome. This is especially important for plants, animals, and microbial species in which no previously assembled high quality reference genome exists. The different error rates between short and long reads lead to different approaches to assembly. For long reads, the first step is typically to find pairs of reads that overlap and resolve their differences (due to errors) by computing the *alignments*, i.e., the edits required to make the overlapping regions identical [6, 7, 16, 20, 24]. The read-to-read alignment computation is not limited to genome assembly, and is widely used in various comparisons across or within genomic data sets to identify regions of similarity caused by structural, functional or evolutionary relationships [26]. Consequently, highly parallel long-read to long-read alignment would significantly improve the efficiency of these techniques, and enable analysis at unprecedented scale.

In this paper, we focus on this computationally challenging problem of finding overlapping reads and computing their alignment. We introduce diBELLA, the first long-read parallel distributed-memory overlapper and aligner. diBELLA uses the methods in BELLA [14], an accurate and efficient single node overlapper and aligner that takes advantage of the statistical properties of the underlying data, including error rate and read length to efficiently and accurately compute overlaps. BELLA is based on a seed-and-extend approach, common to other aligners [2], which finds read pairs that are likely to overlap using a near-linear time algorithm and then performing alignments on those pairs. BELLA parses each read into all fixed-length substrings called *k-mers* (also called seeds in this context), hashing those *k-mers* and then finding pairs with at least one common *k-mer*. Unlike short read aligners or those that align to a well-established reference, the high error rate in long reads means that BELLA's *k-mers* must be fairly short (17-mers are typical); this in turn means that some *k-mers* will appear many times

ACM acknowledges that this contribution was authored or co-authored by an employee, contractor, or affiliate of the United States government. As such, the United States government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for government purposes only.
ICPP 2019, August 5–8, 2019, Kyoto, Japan
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6295-5/19/08…$15.00
https://doi.org/10.1145/3337821.3337919

in the underlying genome and can therefore create multiple extraneous overlaps. diBELLA adopts the innovations from BELLA and parallelizes the seed-and-extend approach by storing *k-mers* in a distributed hash tables, using that to compute read pairs with a common seed, and then distributing the read pairs for load balanced pairwise alignment.

diBELLA takes advantage of distributed-memory on high performance computing (HPC) systems as well as commercial cloud environments. Significant challenges of diBELLA's parallelization include addressing irregular communication, load imbalance, distributed data structures (such as Bloom filters and hash tables), memory utilization, and file I/O overheads. We demonstrate our scalable solution and detailed performance analysis, across four different parallel architectures, with significantly different architectural design tradeoffs. In addition, we present communication bounds in terms of input data (genome) and expected characteristics from real data sets. Our work not only provides a distributed-memory solution for one of the most computationally expensive pieces of the analysis of third-generation sequencing data, it also provides an alternative workload for future architectural developments.

Following some background on the alignment problem in Section 2- 3, we give a high-level overview of the diBELLA pipeline in Section 4 and then each of the parallel stages in Sections 6–9. In each case, we describe the parallelism opportunities and load balancing challenges with respect to the computation and communication patterns and data volumes. We also show scaling numbers for each stage of the pipeline on the architectures and experimental settings detailed in Section 5. The architectures include AWS and 3 Cray HPC systems (Edison and Cori at NERSC, and Titan at OLCF). We discuss the overall pipeline performance in Section 10, and conclude with a review of related work in Section 11, and a summary of our conclusions in Section 12.

2 READ-TO-READ ALIGNMENT

diBELLA computes read-to-read alignment on long-read data to detect overlapping sequences. Formally, a *pairwise alignment* of sequences s and t over an alphabet Σ is defined as the pair (s', t') such that $s', t' \in \Sigma \cup \{-\}$ and the following properties hold:

(1) $|s'| = |t'|$
(2) $\forall_{i=1}^{|s'|}, s'_i \neq - \; OR \; t'_i \neq -$
(3) $\forall_{i=1}^{|s'|}, s'_i \neq - \; AND \; t'_i \neq - \implies s_i = t_i$.
(4) Deleting all "−" from s' yields s, and deleting all "−" from t' yields t.

Equivalently, we can fix one sequence, s, and edit t via insertions and deletions of characters to match s. One is generally interested in only high quality alignments as defined by some scoring scheme that rewards matches and penalizes mismatches, insertions, and deletions. Finding an *optimal* alignment is attainable via a dynamic programming algorithm such as Smith-Waterman and is an $O(|s| \cdot |t|)$ computation [32].

Pairwise alignment can be extended to sets: given sets of sequences S and T, find the best alignment of all $s \in S$ to all $t \in T$. As a step in *de novo* genome assembly, S and T would both correspond to the same set of reads and the pairwise alignment of these two sets would therefore find reads that overlap with each other. Done naively, set alignment requires $O(|S| \cdot |T| \cdot L^2)$ operations for sequences of length L, which becomes intractable for large data sets. However, there are two main improvements possible when performing read-to-read alignment that focus on only high quality outcomes. First, in place of full dynamic programming for pairwise alignment, one can search only for solutions with a limited number of mismatches (banded Smith-Waterman) and terminate early when the alignment score drops significantly (x-drop) [37]. This makes pairwise alignment linear in L. diBELLA performs each pairwise alignment on a single node using an x-drop implementation from the SeqAn library [9]. The second improvement involves efficiently finding sequences in S and T that are likely to match before computing the expensive pairwise alignment. This is accomplished by finding pairs of reads in the input sets that share at least one identical substring.

Each read in S, T is parsed into substrings of fixed length k, *k-mers*, which overlap by $k-1$ characters and are stored in a hash table. Figure 1 illustrates this idea by showing three shared *4-mers* in a given pair of sequences. Given that long-read data contains errors, the choice of the k-mer length is crucial to maximize the detection of *true* overlapping sequences while minimizing the number of attempted pairwise alignments. Quantitatively analyzed in [14], k should be short enough to identify at least one correct shared *k-mer* between two overlapping sequences, but long enough to minimize the number of repeated *k-mers* in the genome, which could lead to either spurious alignments or redundant information. For example, given the two reads in the example in Figure 1, a k-mer length of 5 would fail to find an overlap. Based on the error rate and depth of a given data set, BELLA and diBELLA compute the optimal k-mer length to ensure that a pair of overlapping reads will have with high probability at least one correct k-mer in common. A typical *k-mer* length for long read data sets is 17-mers based on extensive analysis in [14], whereas it is common to use 51-mers for short read aligners. Note that not all *k-mers* are useful for detecting overlaps. *k-mers* that occur only a single time across S and T, called *singletons*, are ignored as they are likely erroneous. Even if it wasn't an error, a singleton cannot be used to detect an overlap between two strings since it only occurs in one string. Conversely, *k-mers* that occur with very high frequency across the data set are likely from repeated regions of the underlying genome, and can lead to unnecessary or incorrect alignments. diBELLA therefore eliminates high frequency *k-mers* over a threshold m, which is calculated via the approach presented in BELLA [14], using the error rate and other characteristics of the input data set. The *k-mers* that remain after this filtering, we refer to as *retained k-mers* and will be used to detect the overlapping reads on which pairwise alignment is performed. This *k-mer* filtering is specifically for the alignment of long reads to long reads with their high error rates and will affect our parallelization strategy.

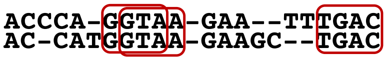
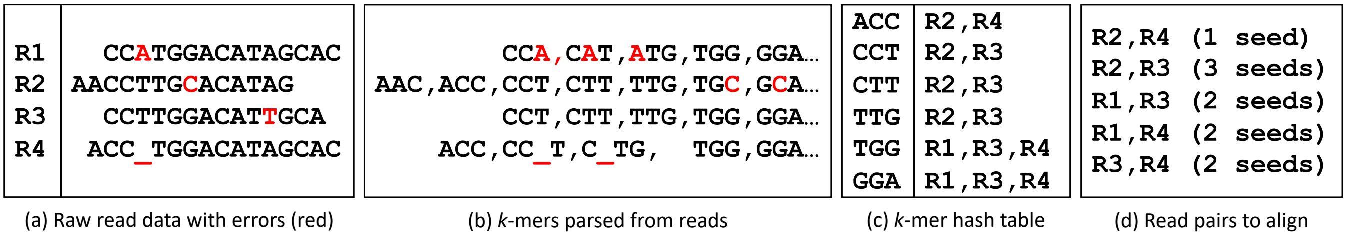


Figure 1: Pairwise alignment of two sequences with 3 common *k-mers* of length 4.



R1	CCATGGACATAGCAC
R2	AACCTTGCACATAG
R3	CCTTGGACATTGCA
R4	ACC_TGGACATAGCAC

(a) Raw read data with errors (red)

```
           CCA,CAT,ATG,TGG,GGA...
AAC,ACC,CCT,CTT,TTG,TGC,GCA...
           CCT,CTT,TTG,TGG,GGA...
       ACC,CC_T,C_TG,   TGG,GGA...
```

(b) *k*-mers parsed from reads

ACC	R2,R4
CCT	R2,R3
CTT	R2,R3
TTG	R2,R3
TGG	R1,R3,R4
GGA	R1,R3,R4

(c) *k*-mer hash table

```
R2,R4 (1 seed)
R2,R3 (3 seeds)
R1,R3 (2 seeds)
R1,R4 (2 seeds)
R3,R4 (2 seeds)
```

(d) Read pairs to align

Figure 2: Overview of diBELLA's pipeline, using $k = 3$ as example: (a) raw input data, (b) k-mer extraction, (c) k-mer hash table and associated read list, and (d) read pair alignment using the seed-and-extend paradigm.

3 COMPUTATIONAL COST

To approximate the computational cost, we first note that the size of the long read input data set N from a given genome is determined by two variables, the size of the underlying genome G and the average depth of per base coverage d (equation 1).

$$N = G \cdot d \tag{1}$$

If L is the average length of sequences in the input, then the size of the read set is $R = G \cdot d/L$. The computation extracts *k-mers* starting at every location in each read of the input set. Thus a read of length L has $L - k + 1$ *k-mers* (although not necessary unique ones). For long-read data, L is generally in the range $1,000 - 100,000$ and k is in the range $11 - 21$ so we approximate the number of *k-mer*'s per read as L. The number of the *k-mers* parsed from the input (i.e., the *bag* of *k-mers*, which may have duplicates) is thus approximately $G \cdot d$.

$$\frac{G \cdot d \cdot (L - k + 1)}{L} \approx G \cdot d \tag{2}$$

Therefore, the total volume of *k-mers* from the input is $k \cdot G \cdot d$ characters. Each *k-mer* character from the four letter alphabet $\{A, C, T, G\}$ can be represented with 2 bits. To support varying values of k with efficient memory storage and alignment, we provide compile time parameters for the *k-mer* representation (typically set to 32 bits or the nearest larger power of two). In general, we avoid storing the entire *k-mer* bag in memory at once, unless sufficient distributed memory resources happen to be available. For example, the *k-mer* bag size of two PacBio *E. coli* data sets, with 30x and 100x coverage respectively, is nearly 3 billion and 11 billion *k-mers*.

diBELLA operates predominately on the much smaller set of distinct *k-mers*, although we retain some information about each instance of a *k-mer*, such as its locations within different reads. We further reduce the size of the retained *k-mer* set (as described in Section 2) by eliminating singletons and high-frequency *k-mers*. Assuming a properly chosen value of k, the set of filtered *k-mers* is approximately the size of the final assembled genome G.

4 diBELLA OVERVIEW

Our distributed-memory diBELLA design is a multi-stage parallel pipeline. *k-mers* are first extracted from files of reads and filtered by frequency, as described in Section 2. Each processor manages a subset of the reads and a subset of the *k-mers*. Note that there is no inherent locality in the order of the reads from the input files (called FASTQ) with respect to their overlap. The first phase builds a distributed Bloom filter[11] to identify and eliminate most singleton *k-mers*. The second phase builds a hash table of non-singleton *k-mers* (as approximated by the Bloom filter), and further filters *k-mers* exceeding the high occurrence threshold, m (see Section 2). The remaining hash table represents a graph with reads (represented by identifiers) as vertices and reliable *k-mers* as edges. That is, two vertices (long reads) are connected if they share a common *k-mer* that was retained after filtering. The next stage forms all pairs of read IDs that share a retained *k-mer* and tracks their location within the reads. The final stage performs alignment on these read pairs using the shared *k-mer* as the starting position (seed) for pairwise alignment.

Our distributed memory design is a four-stage pipeline, with an example shown in Figure 2 :

(1) Extract *k-mers* from files of reads and store in a distributed Bloom filter to eliminate singleton *k-mers*. Initialize the hash table with non-filtered *k-mers*.
(2) Extract *k-mers* and their location metadata from the files again. Insert into the distributed hash table only if the *k-mer* is already resident. After this is done, remove singleton *k-mers* that were missed by the Bloom filter and those that exceed the high occurrence threshold, m.
(3) For each *k-mer* in the hash table, take the associated list of read IDs (and positions) and form all pairs of reads, assigning each pair to one processor.
(4) Redistribute and replicate reads (the original strings) to match read-pair distribution and perform pairwise alignment on each pair locally.

The algorithm makes two passes over the data in order to not store all the parsed *k-mers* in main memory; diBELLA executes in a streaming fashion with a subset of input data at a time to limit the memory consumption.

The Bloom filter, hash table, and list of read pairs are all distributed across the nodes, and the predominate communication pattern, common to each stage, is irregular all-to-all exchanges. The first two stages exchange *k-mers* for counting and for initializing the hash table with *k-mers* and respective source locations. The *k-mers* are mapped to processors uniformly at random via hashing, such that each processor will own roughly the same number of distinct *k-mers*, as in [11]. Further details for these stages follow in Sections 6-7. The third phase consolidates read-pairings (*overlaps*), and their lists of shared *k-mer* positions, which represent alignment tasks. The details of the parallelization and task redistribution are provided in Section 8. The final stage computes all pairwise alignments. Because the pairwise alignments require the full reads, any non-local reads are requested and received by the respective processor. This last stage is described in detail in Section 9. Overall, our design employs Bulk Synchronous Processing [34] throughout, with the communication implemented via MPI Alltoall and Alltoallv functions. Note that a load imbalance can result from the data characteristics, including highly repetitive genome regions. The current diBELLA implementation makes particular design choices for data layout, communication aggregation, and synchronization, and we evaluate their effectiveness through extensive cross-platform performance analysis while identifying opportunities for future optimizations within the general framework. The specific techniques for *k-mer* length selection, filtering and local alignment are based on those in BELLA, but the parallelization approach is applicable to this general style of long read aligner based on *k-mer* filtering and hashing.

5 EXPERIMENTAL SETUP

Our experiments were conducted on four computing platforms, which include HPC systems with varying balance points between communication and computation, as well as a commodity AWS cluster. This gives us performance insights into tradeoffs between extremes of network capabilities. Evaluated platforms include the Cori Cray XC40 and Edison Cray XC30 supercomputers at NERSC, the Cray XK7 MPP at the Oak Ridge National Lab, and an Amazon Web Services (AWS) c3.8xlarge cluster. Details about each architecture are presented in Table 1. Titan has GPUs and CPUs on each node, but we use only the CPUs with total 16 (integer) cores per node. AWS does not reveal specifics about the underlying node architecture or interconnect topology, other than an expected 10 Gigabit injection bandwidth. Based on our measurements, the AWS node has similar performance to a Titan CPU node. Both data sets are small enough to fit in the memory of a single node, and in all experiments, MPI Ranks are pinned to cores.

To highlight cross-network performance and communication bottlenecks, most of our experiments use an input data set and runtime parameters that result in low computational intensity. This data is from *E. coli* bacteria with a depth of 30×, which consists of 16, 890 long reads from the from *Escherichia coli* MG1655 strain, resulting in a 266 MB input file; it has been sequenced using PacBio RS II P5-C3 technology and it has an average read length of 9, 958 bp. The second data set, *E. coli* 100×, was sequenced using PacBio RS II P4-C2 and uses a depth of 100. It consists of 91, 394 long reads from the same strain with an average read length of 6, 934 bps, resulting in a 929 MB input file. diBELLA's overlap detection step identifies 2.27M read pairs for the first data set and 24.87M for the second one.

Computational intensity is most affected by the number of alignments performed for each pair of reads, since each pair might share varying numbers of seeds. Some of these seeds reflect a shifted version the same overlapping region, whereas others may be independent (and ultimately incorrect) overlaps. We use three different options to provide a range of computational intensity. At the two extremes, the *one-seed* option computes pairwise alignment on exactly one seed per pair, while the *all-seed* option computes pairwise alignment on all the available seeds separated by at least the *k-mer* length. As an intermediate point we consider only seeds separated by 1, 000 bps. The analysis associated with the design of BELLA [14], shows that even 1,000 bp separation can be used without significantly impacting quality.

Both data sets are sufficiently small that the working set size fits on a single node across the platforms in our comparison. This choice enables us to show the performance impact of intra-node to inter-node communication on the overall pipeline performance and highlight scaling bottlenecks, and to explore strong scaling on a modest number of nodes, important for comparison with AWS.

6 BLOOM FILTER CONSTRUCTION

Given that singleton *k-mers* constitute the majority of the *k-mer* data set, retaining them is memory inefficient since it would require storage k times larger than the input size. Therefore the goal of this stage is to build a distributed Bloom filter to identify (with high probability) singleton *k-mers*, which can be ignored. It also enables the initialization of a distributed hash table containing the unfiltered *k-mers*. Briefly, a Bloom filter is an array of bits that uses multiple hash functions on each element to set bits in the array. Due to collisions, a value may not be in the array even if its hash bits are set, but a value with at least one zero is guaranteed to be absent from the set [4] (i.e. it may allow false positives, but does not contain false negatives). We follow the methodology of the HipMer short read assembler [13] for this stage, but note that the Bloom filter is even more effective for long reads due to their higher error rate — up to 98% of *k-mers* from long reads are singletons vs. 60 − 85% for short reads. Minimizing the Bloom filter false positive rate depends on the (unknown *a priori*) cardinality of the *k-mer* set. In our experiments thus far, we have not encountered a case where approximating the *k-mer* cardinality using equation 2 and typical ratios of singleton *k-mers* to all *k-mers* across data sets did not provide a sufficiently accurate estimate, such that the more expensive HyperLogLog algorithm in HipMer was required. However, we suspect that for extremely large (tens of trillions of base pairs) and repetitive genomes that we may encounter the same issues that led to this optimization in HipMer.

As mentioned, the input reads are distributed roughly uniformly over the processors using parallel I/O, but there are is no locality inherent in the input files. Each rank in parallel parses its reads into *k-mers*, hashes the *k-mers*, and eventually sends them to a processor indicated by the hash function. The hash function ensures that each rank is assigned roughly the same number of *k-mers*. On the remote node, the received *k-mers* are inserted into the local Bloom filter partition. If a *k-mer* was already present, it is also inserted into the local hash table partition. Although all $G \cdot d$ *k-mers* are to be computed, this process is performed in stages since only a subset of *k-mers* may fit in memory at one time. The Bloom filter construction communicates nearly all (roughly $(P-1)/P$) of the *k-mer* instances to other processors in a series of bulk synchronous phases. The total number of phases depends on the size of the input, and the irregular all-to-all exchange is implemented with MPI Alltoall and Alltoallv. After the hash table is initialized with *k-mer* keys, the Bloom filter is freed.

Figure 3, shows strong scaling performance (including communication) of the Bloom filter phase across the four platforms in our study, measured in millions of *k-mers* processed per second.

Table 1: Evaluated platforms. *128 byte Get message latency in microseconds. †Using the optimal number of cores per node. ‡Measured over approx. 2K cores or maximum (128 for ethernet cluster). §MB/s with 8K message sizes. $^\alpha$CPU nodes only.

Processor	Cori I Cray XC40 Intel Xeon (Haswell)	Edison Cray XC30 Intel Xeon (Ivy Bridge)	Titan Cray XK7$^\alpha$ AMD Opteron 16-Core
Freq (GHz)	2.3	2.4	2.2
Cores/Node	32	24	16
Intranode LAT*†	2.7	0.8	1.1
BW/Node†‡§	113.0	436.2	99.2
Memory (GB)	128	64	32
Network and Topology	Aries Dragonfly	Aries Dragonfly	Gemini 3D Torus

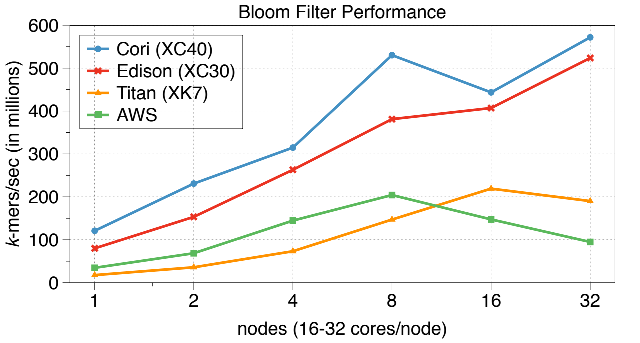


Figure 3: Bloom Filter cross-architecture performance in millions of *k-mers* processed / sec, given E.coli 30x one-seed.

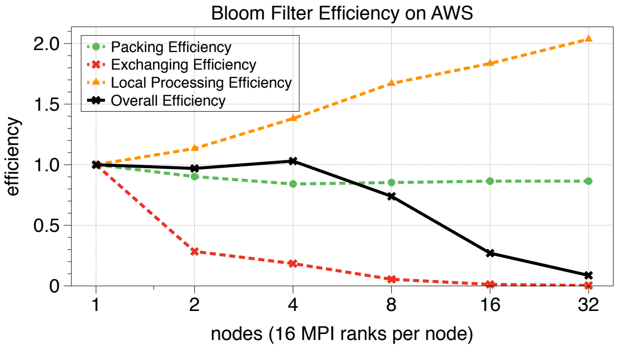


Figure 4: Bloom Filter efficiency on AWS within a 32 node placement group, 1 MPI Rank per core, 16 per node, strong scaling with E.coli 30x one-seed.

Note that on Titan (Cray XK7), 1 MPI Rank is assigned to each *Integer Core*/L1 Cache, and the GPU are not utilized. Each node of Titan contains 16 *Integer Cores*, the overall computational peak of which is significantly lower than Cori and Edison (which contain 32 and 24 more powerful cores per node, respectively). Titan's *k-mer* processing rate is most similar to the AWS cluster (which contains 16 cores per node), and surpasses AWS performance only when communication becomes the dominant bottleneck at 16-32 nodes.

Figure 4 presents a detailed breakdown of the strong scaling efficiency on AWS. Note that the *Local Processing* (hashing and storing *k-mers*) speeds up superlinearly, since more of the input fits in cache for this strong scaling experiment. On the other hand, the *Exchanging* efficiency, computed relative to the single (intra) node communication, degrades significantly with increased concurrency, and eventually overwhelms the overall runtime. More detailed measurements (not shown) reveal that some of the poor scaling in *Exchanging* is only in the first call to MPI's Alltoallv routine. The overhead is assumably from the MPI implementation's internal data structure initialization, related to process coordination and communication buffers setup for subsequent calls.

7 HASH TABLE CONSTRUCTION

In order to identify reads with at least one common *k-mer*, the next phase builds a hash table of *k-mers* and the lists of all read ID (RID) and locations at which they appeared. In this stage all reads are again parsed into *k-mers*, hashed, and sent to the processor owning that *k-mer*, and if the *k-mer* key exists in the hash table (not a singleton), it is inserted with its RID and location and its count incremented. The same strategy for load balancing *k-mers* as in the Bloom filter construction stage (Section 6) is employed here; the *k-mers* are hashed to the same distributed memory location that they were in the previous stage. At the end of this process, the local hash table partitions are traversed independently in parallel to remove any *k-mers* that occur more times than the maximum frequency, and any false-positive singletons. The remaining *k-mers* are referred to as *retained k-mers*. This extra RID and location information makes the hash table different than other tools intended for de Bruijn graph construction of short reads (such as HipMer [13]) or those used to analyze read data directly by counting *k-mers* (such as Jellyfish [25]). The communication is again done in a memory-limited set of bulk-synchronous phases, where the irregular all-to-all exchange of *k-mers* and associated data is implemented with MPI Alltoall and Alltoallv. Note that while the communication volume of this stage is 2.5x larger than the Bloom filter stage, the amount of computation is also higher due to the RID and location handling, as well as the hash table traversal. This difference is apparent in the strong scaling performance comparison between Bloom filtering in Figure 3 and hash table construction in Figure 5. Although the trends are similar across stages and platforms, the computation rate of the hash table stage is roughly double that of the Bloom filter stage. Once again improved cache behavior results in superlinear speed up for this strong scaling computation.

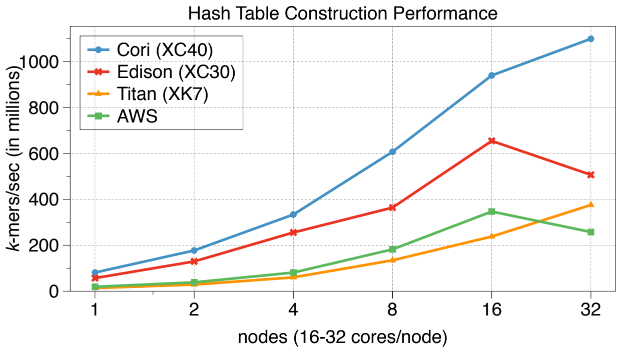


Figure 5: Hash Table stage, cross-architecture performance in millions of *k-mers*/second given E.coli 30x one-seed.

8 OVERLAP

Once the distributed hash table is computed, which maps reliable *k-mers* to source locations (RIDs and positions), the overlap computation is straightforward. Rather than constructing the matrix explicitly as in BELLA, we avoid the associated overhead and compute overlaps directly from hash table partitions, independently in parallel in diBELLA. Further exploration of the associated design tradeoffs is part of ongoing work. Algorithm 1 illustrates this simple, direct computation of set of all pairs of reads represented by identifiers (r_a, r_b), where r_a and r_b share reliable *k-mer* (s). Each *k-mer* "contributes" to the discovery of $[2, m(m-1)/2]$ read pairs where m is the maximum frequency of *reliable k-mers* in diBELLA, or simply the maximum frequency of retained *k-mers* in general. Each of these represents an alignment task for the next stage. However, the owner of the *k-mer* matching (r_a, r_b) may not be the owner of either involved read. To maximize locality in the alignment stage (minimize the movement of reads) each task is buffered for the owner of r_a or r_b (which may be the same owner), according to the simple odd-even heuristic in Algorithm 1. Recall, reads in the input are unordered and partitioned uniformly. The hash table values (RID lists) are also unordered. Hence, for fairly uniform distributions of reliable *k-mers* in the input, we expect this heuristic to roughly balance the number of alignment tasks assigned to each processor. Load balancing by number of tasks is however imperfect, since individual pairwise alignment tasks may have different costs in the alignment stage. The computational impact of various features, such as read lengths and *k-mer* similarity, could be used for estimating the cost changes with the pairwise alignment kernel. We leave further analysis of the relationship between the choice of pairwise alignment kernel and overall load balancing to future work. Our expectations of the general load balancing strategy are discussed further with empirical results in the context of the alignment stage description, Section 9. The final steps of the overlap stage are the irregular all-to-all communication of buffered tasks, implemented with MPI_Alltoallv, and the (optional) output of the overlaps.

Neither the number of overlapping read pairs nor the number of retained *k-mers* common to each can be determined for a given workload until runtime. However, we provide generalizable bounds on the computation and communication from a few basic observations. Recall from Section 2 that the total number of *k-mers* parsed from the input is one order of magnitude larger than the number of characters in the input. However, only a small fraction of these are stored, those that are distinct, and with frequency in [2, m]. Let the fraction of retained *k-mers* to the total number of (input) *k-mers* (K_{input}) be ι_{input}, and the fraction of retained *k-mers* to the size of the *k-mer* set, $|K_{set}|$ be ι_{set}. The importance of the distinction is that $|K_{set}|$ cannot be known until the end of the *k-mer* analysis stage, whereas K_{input} is known *a priori*. Further, $|K_{set}| \leq K_{input}$, and thus $\iota_{set} \geq \iota input$. In our cross-genome analysis, $\iota_{set} \in [0.04, 0.12]$. This analysis is useful for estimating the overlap computation and communication costs, and applicable beyond our particular implementation.

Algorithm 1: Parallel (SPMD) hash table traversal

```
Result: All pairs of reads sharing at least 1 retained k-mer in
        hash table partition, H, and corresponding k-mer
        positions (elided) are composed into alignment tasks.
        Each task, with read identifiers $(r_a, r_b)$, is stored in a
        message buffer for the owner of $r_a$ or $r_b$.
for each k-mer key $k_{hash}$ in hash table H do
    for $i = 0$ to $m-2$ do
        for $j=i+1$ to $m-1$ do
            $(r_a, r_b)$ = task(H[$k_{hash}$][i] , H[$k_{hash}$][j],...)
            if $r_a\%2 = 0$ AND $r_a > r_b + 1$ then
                buffer[owner($r_a$)]← $(r_a, r_b)$
            else if $r_a\%2 \neq 0$ AND $r_a < r_b + 1$ then
                buffer[owner($r_a$)]← $(r_a, r_b)$
            else
                buffer[owner($r_b$)]← $(r_a, r_b)$
            end
        end
    end
end
```

An upper bound on the total (global) number of overlaps follows in Equation (3). The lower bound (Equation (4)) follows from the fact that retained *k-mers* must occur in at least two distinct reads (identifying at least one overlap) or they are discarded. The parallel computational complexity of Algorithm 1's overlap detection (with P parallel processors) is shown in Equation (5), which assumes constant-time storage of read pair identifiers. The hidden constant in Equation (5) is halved by exploiting asymmetry.

$$O(\iota_{set} \times K_{set} \times m^2) < O(\iota_{input} \times K_{input} \times m^2) \quad (3)$$

$$O(\iota_{set} \times K_{set}) < O(\iota_{input} \times K_{input}) \quad (4)$$

$$O(\frac{\iota_{set} K_{set} m^2}{P}) \quad (5)$$

Ignoring the constant for the size of the overlap representation (a pair of read identifiers and positions in our case), the aggregate communication volume is also bounded above by Equation 3, and below by Equation 4.

As a last computational step, after the overlaps are computed and communicated (and lists of common *k-mers* consolidated), the

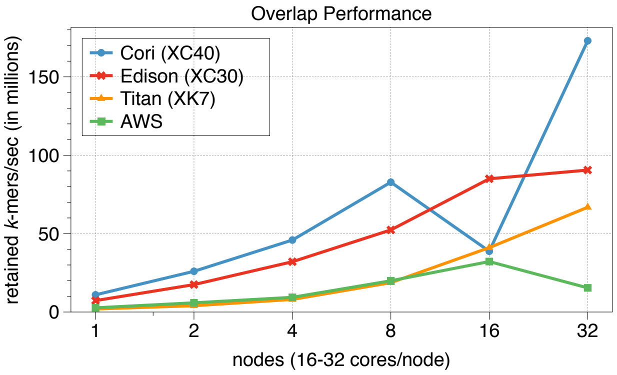


Figure 6: Cross-architecture Overlap stage performance in millions of retained *k-mers*/second given E.coli 30x one-seed.

lists may be filtered further depending on certain runtime parameters. That is, some subset of all *k-mers* per overlapping read pair will be used to seed the alignment in the next stage; the subset is determined by the shared retained *k-mers* total (simply all may be used) and also by certain runtime parameters which can be thought of as "exploration" constraints. These include the minimum distance between seeds, and the maximum number of seeds to explore per overlap. A discussion of these settings in relation to alignment accuracy versus computational cost is presented in the BELLA analysis[14]. In general, increasing the number of seeds to explore per overlap increases computational cost of the alignment stage (not necessarily linearly), depending on the pairwise alignment kernel employed. We present results varying the number of seeds Section 9.

Strong scaling results for the overlap stage are shown in Figure 6. These are presented across our evaluated platforms in terms of millions of retained *k*-mers processed per second, and show similar computational behavior as the previous stages. One unexpected feature of this graph is the dip Cori's performance trend at 16 nodes, due to an unexpected spike in the communication exchange time that does not continue to 32 nodes. The absolute time is short enough to have been caused by interference in the network, but the spike nonetheless brings the Cori performance down to Titan and AWS's at 16 nodes.

9 ALIGNMENT

The *k-mer* load balancing strategy described in sections 6-7 enables uniform *k-mer* load balancing and complete parallel overlap detection. The rebalancing of overlapping reads, however, is left for the alignment stage. Recall, that the input reads are not ordered, and our algorithm partitions them as uniformly as possible at the beginning of the computation (by the read size in memory). Only *k-mers* and read identifiers (not the actual reads) are communicated in the initial stages of the pipeline. After the overlap stage communication, each overlap identifier together with the associated list of share *k-mer* positions, are stored in the appropriate owners location. Note, the *k-mer* positions are retained rather than recomputed because they are the locations of (globally) rare *k-mers* (see Section 2). Computing the alignment of any overlapping pair of reads, however, requires both of the respective input reads.

The properties of the overlap graph underpin the communication design of our application. The size of the retained *k-mer* set determines the size (and sparsity) of the overlap graph. From our filtering steps, we expect this graph to be sparse; from empirical observations across data sets, the filtering typically reduces the *k-mer* set size by 85-98%.

To effectively maximize locality and bandwidth utilization under these conditions, we first explore the performance of a bulk synchronous exchange implemented via MPI_Alltoallv. Note that once the reads are communicated, the alignment computation can proceed independently in parallel. We expect that speedups from the subsequently embarrassingly parallel alignment computations (which are quadratic for exact pairwise alignment and at least linear in the length of the long reads for approximate alignments) will compensate for inefficiencies in the communication to some workload-dependent degree of parallelism.

Each overlap shares an unknown *a priori* number of retained *k-mers* that are used as alignment seeds. Anywhere between one or all of these seeds will be explored in application runs, depending on the user's objectives and runtime settings. Figure 7 shows performance (alignments per second) across our evaluated platforms machines using the (computationally worst-case) one seed per alignment. Here, the number and speed of the cores per node determine the relative performance ranking (see Table 1), with Cori's 32 cores/node clearly surpassing the other systems.

The load balancing strategy described in Sections 8-9 produces near perfect load balancing in terms of the number of alignments computed per parallel process, but imperfect load balancing in terms of time to exchange and compute all alignments. Figure 8 shows the latter load imbalance, calculated as maximum per rank alignment stage times over average times across ranks (1.0 is perfect). There are two reasons for this load imbalance in terms of compute and exchange costs: (1) reads have different lengths, which effect both the exchange time and the pairwise alignment time, (2) the x-drop algorithm returns much faster when the two sequences are divergent because it does not compute the same number of cell updates. A smarter read-to-processor assignment could optimize for variable read lengths, eliminating the exchange imbalance. However, the imbalance due to x-drop can not be optimized statically as it is not known before the alignment is performed. To mitigate the impact of (2), one would need dynamic load balancing, which is known to be high-overhead in distributed memory architectures. The load imbalance in terms of the number of alignments performed per processor is less than 0.002% across all machines and scales. Future work should consider not only the number of alignments per processor but other kernel-dependent characteristics affecting the cost of each pairwise alignment.

10 PERFORMANCE ANALYSIS

The performance rates on each stage show similar results across machines, with the more powerful Haswell CPU nodes and network on Cori (XC40) giving superior overall performance. As expected, all-to-all style communication scales poorly on all networks, with

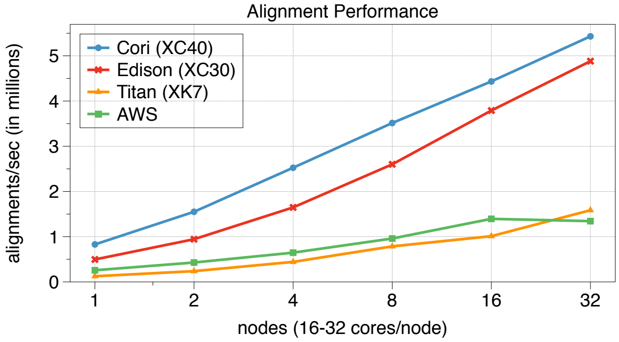


Figure 7: Cross-architecture Alignment stage strong scaling in millions of alignments / second given E.coli 30x one-seed.

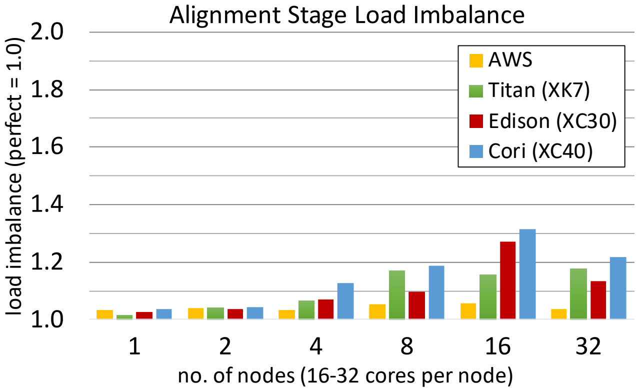


Figure 8: Alignment stage load imbalance on E.coli 30x one-seed, calculated using maximum over average stage times across ranks (1.0 is perfect).

but especially the commodity AWS network. Somewhat more surprising is the high level of superlinear speedup on some stages once the data fits in cache or other memory hierarchy level. The question for overall performance is how these two effects trade off against one another and how the stages balance out. Figure 11 shows diBELLA's overall pipeline efficiency on Cori, varying workloads and computational intensity. Two data sets are used, *E. coli* 30x and *E. coli* 100x, and 3 seed constraints, one-seed, all seeds separated by 1Kbps, and all seeds separated by $k = 17$ bps. Clearly, increasing the computational intensity with larger inputs and seed counts does not alone determine overall efficiency. While the computational efficiency increases with higher computational intensity, the overall efficiency is significantly impacted by the degrading efficiency of exchanges. Figures 9 and 10 respectively show runtime breakdown by stages on Cori for *E. coli* 30× exploring 1 seed per overlapping pair of reads, and *E. coli* 100×, using all seeds with a minimum of 1Kbps-distant from each other. These represent two extremes in terms of computational intensity, the former being minimal, the latter being much higher, but still a realistic point of comparison for the same input genome. The communication time is broken out for each stage. The stages are fairly evenly balanced, although alignment is more expensive computationally than the others (and

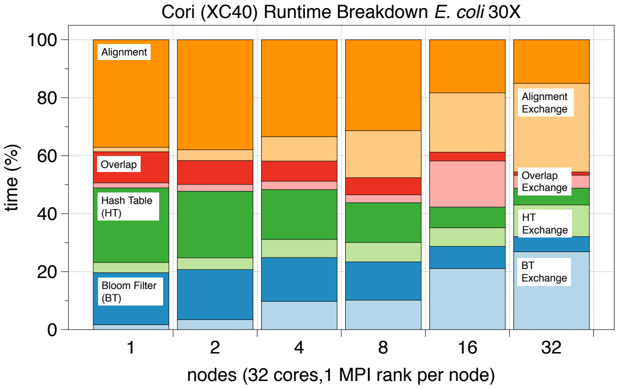


Figure 9: Runtime breakdown on Cray XC40 with minimum computational-intensity workload (E.coli 30x single seed).

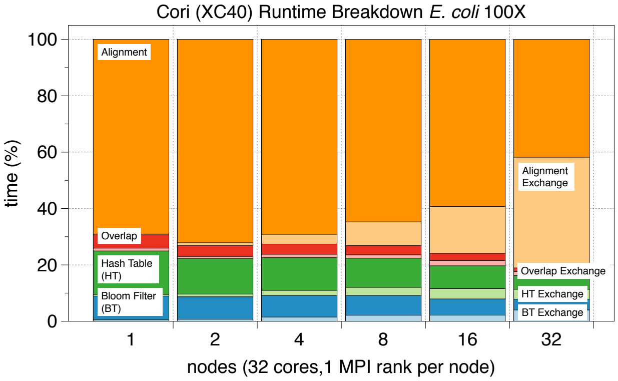


Figure 10: Runtime breakdown on Cray XC40 with higher computational-intensity workload (E.coli 100x, all seeds).

dominates to 32 nodes in the more computationally intense workload). Focusing on Figure 9, the communication time in the Bloom Filter stage is surprisingly higher than in the Hash Table stage where the volume is 2.5x higher, and the communication pattern and number of messages is identical. Further investigation revealed that the problem is the first call to the MPI Alltoallv routine, which is almost twice as expensive the first time as the second, so the Hash Table stage benefitted from whatever internal data structure and communication initialization happened in the Bloom Filter stage. This effect was visible to varying degrees on all 4 platforms. This kind of behavior is most noticeable for workloads with lowest computational intensity.

To further drill down on network and processor balance, Figure 12 shows the efficiency across all 3 HPC networks over the overall pipeline and the exchange time on each. From an efficiency standpoint, the Cray XK7 using only the CPU features on each node gives the best network balance for this problem, even though the network is an older generation than on the XC30 and XC40.

From a performance standpoint, the higher speed processor and network on Cori (XC40), while not as well balanced for efficiency, outperforms the other on the full diBELLA pipeline, Figure 13. Here

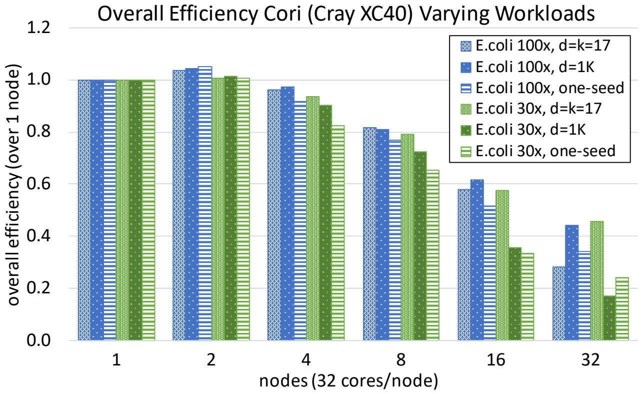


Figure 11: Overall efficiency on Cray XC40 over 2 data sets, E.coli 30x and E.coli 100x, varying seed constraints (1 seed, all separated by 1K, and all separated by k=17 characters).

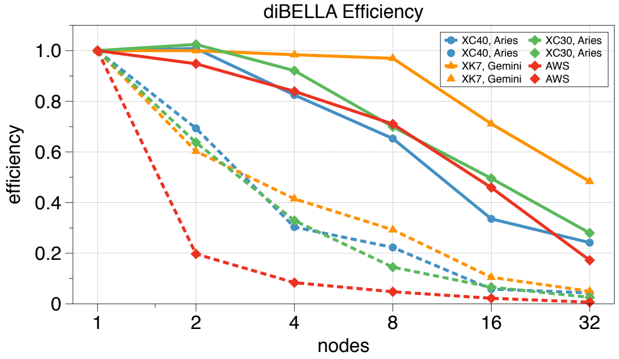


Figure 12: DiBELLA cross-architecture overall (solid) and exchange (dashed) efficiency over minimally compute intensive workload (E.coli 30x one-seed).

we measure performance as alignments per second, where the total number of alignments is fixed for a given input configuration. The performance anomaly at 16 nodes for Cori, which was seen in the Alignment stage, is apparent here in the overall performance, probably due to a performance issue in MPI implementation. With that exception and a drop of performance on AWS at 32 nodes, all of the systems show increasing performance on increased node counts. Recall, that our standard problem used here (*E. coli* at 30x coverage with only 1 seed used per read pair for alignment) was specifically chosen as the low end of computational intensity, and so highlights scaling limits of the machines.

11 RELATED WORK

Nowadays, existing distributed memory alignment codes target the alignment of a read set against a fixed, modest-sized reference sequence such as the human genome [15], where the reference can be replicated across nodes in advance. Conversely, diBELLA computes read-to-read pairwise alignment rather than read-to-reference alignment, and distributes data across nodes for each step of the pipeline. Our implementation is more similar (in spirit) to the end-to-end parallelization in MerAligner [12]. However, diBELLA addresses long read data characteristics, and accordingly, uses a different parallelization and data layout approach. MerAligner aligns short reads to *contigs*, sequences composed of error-free *k-mers*, in order to find connections between *contigs*. Long reads are not only 2-3 orders of magnitude longer than short reads, but also contain errors up to 35% (versus < 1% for short reads). Hence, appropriate *k-mer* lengths for long read overlap and alignment are an order of magnitude shorter. These features combined dramatically increase the size of the *k-mer* data set. Further, in MerAligner, the cardinality of the *contig* set is reduced significantly over the size of the input (see [11] and [10]); whereas in long-read-to-long-read alignment, a potentially all-to-all comparison of input reads may be performed. In summary, these differences result, for the long read case, in significantly higher communication, computation, and memory consumption rates, and a fundamental difference in pairwise alignment kernel and parameter choices (relating both to quality and computational cost). While exploring a PGAS design for this application similar to MerAligner's is part of ongoing work, we do not provide a direct comparison since they target significantly different problems.

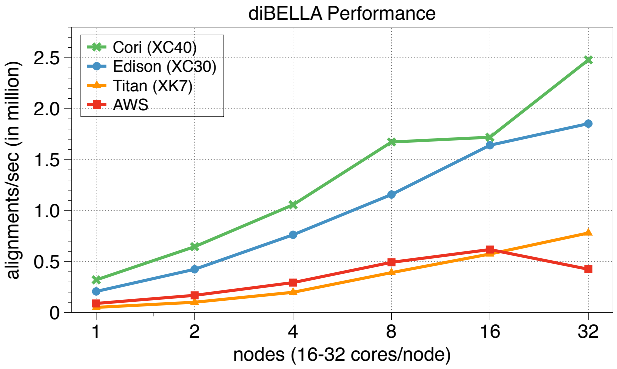


Figure 13: DiBELLA cross-architecture strong scaling as millions of alignments per second given E.coli 30x one-seed.

Table 2: Single node, 64 thread runtime (s) comparison (excluding I/O) on Cori Haswell w/ 128 GB RAM. Reported DALIGNER times also exclude all pre- and post- processing.

	E.coli 30x (sample)	*E.coli* 30x	*E.coli* 100x
diBELLA	12.74	65.72	79.45
DALIGNER	7.31	52.04	63.70

In addition to BELLA, which is the method that our distributed algorithm is based on, other state-of-the-art long-read to long-read aligners include BLASR [5], MHAP [3], Minimap2 [22], MECAT [35], and DALIGNER [27]. BLASR uses BWT and FM index to identify short *k-mer* matches and then groups *k-mer* matches within a given inter-*k-mer* distance. Grouped *k-mers* are ranked based on a function of the *k-mer* frequency and highly scored groups are kept for downstream analysis. MHAP, Minimap2, MECAT, and DALIGNER use *k-mer* matches for identifying candidate overlapping pairs, similarly to diBELLA. MHAP computes an approximate overlap detection performing sketching on the *k-mer* set using minHash. Compact sketches are used to estimate the similarity between sequences. Minimap2 uses minimizers rather than all possible *k-mers* to obtain a compact representation of the original read. Collinear *k-mers* on a read are chained together and used for finding possible matches with other sequences.

Like BELLA and diBELLA, MECAT and DALIGNER do not use approximate representations. MECAT divides reads into blocks and scans for identical *k-mers* which are used to calculate a distance difference factor first between neighboring *k-mers* hits, and then between neighboring blocks. DALIGNER computes a *k-mer* sorting based on the position within a sequence and then uses a merge-sort to detect common *k-mers* between sequences. DALIGNER supports problem sizes exceeding single node resources through a scripting frontend, that divides work into a series of independent execution steps. This script-generated-script can be executed directly (serially) on a single node, or the user can modify it to run independent batch jobs, as individual node resources become available in a distributed setting. For example, if the data is divided into blocks, $B_1, ...B_n$ then DALIGNER can be executed independently by aligning B_1 to itself as one job, B_2 to itself and B_1 as another, and so on. This approach addresses the memory limitations, but it is not scalable. DALIGNER's distributed memory approach reads B_1 from disk n times and the amount of work varies significantly across nodes. Given these differences, we do not provide a direct multi-node comparison, however for completeness, we provide a single node runtime comparison of diBELLA and DALIGNER in Table 2. We exclude I/O time from each measurement, since each tool handles both input and output in significantly different ways. For DALIGNER, we additionally exclude all preprocessing (initializing the database and splitting it into blocks) and post processing (all commands for verifying and merging results) time, and implicitly, the time required of DALIGNER users to extract human-readable results of interest from the database. Table 2 shows that diBELLA's single node runtime is competitive with DALIGNER's across these data sets, even excluding DALIGNER's I/O, preprocessing, and postprocessing.

Of the alternatives, BELLA is the latest (and ongoing) work, with a comprehensive quality comparison to all the above [14]. BELLA's quality is competitive, especially excelling in comparisons where the "ground truth" is known. Further, BELLA offers a computationally efficient approach, yielding *consistently* high accuracy across data sets, and a well-explained and supported methodology. For these reasons, we chose BELLA as the basis for our distributed memory algorithm. The quality produced by diBELLA is at least that of BELLA (see [14] for quality comparisons over data sets also used in this study), and higher when using less restricted sets of seeds than [14].

De novo genome assembly depending on long-read alignment is becoming a crucial step in bioinformatics. Current available long-read *de novo* assemblers are Canu [20], which uses MHAP as long-read aligner, Miniasm [21] which uses Minimap2, and HINGE [17] and FALCON [7], which use DALIGNER. Flye [18] uses the longest jump-subpath approach [23] to compute alignments. From a hardware acceleration standpoint, there has been increasing interest in the long read alignment problem as well, as evidenced by recent work [33][1]. Though we do not explore it in this work, we leave it as a promising future direction.

As mentioned in Section 2, long-read to long-read alignment requires filtering out part of the *k-mers* in order to avoid either spurious alignments or performing unnecessary computation. The parallel *k-mers* analysis in diBELLA is built upon that of HipMer [13]. The Bloom filter stage is identical, while the hash table implementation is different. For each *k-mer*, HipMer stores only the two neighboring bases in the original read it was extracted from (which have to be unique, so there is a single entry per *k-mer*). diBELLA instead needs to communicate and store information about the read where the *k-mer* originated and the location of in which each *k-mer instance* appeared. Both HipMer and diBELLA remove singleton *k-mers*, but diBELLA also removes those *k-mers* whose occurrence exceeds the high occurrence threshold, m. The hash tables also represent different objects. The HipMer hash table represents a de Bruijn graph with *k-mer* vertices, and their connections are computed by adding the *k-mer* extensions and shifting. The graph is broken at points where there is no confidence in the most likely extension. The high error rates in long reads would make such a graph very fragmented. DiBELLA's hash table represents a read graph with read vertices connected to each other by shared *k-mers*, the *k-mers* are then extended to overlapping regions through pairwise alignment. This graph representation, often known as the *overlap graph* in the literature, is more robust to sequencing errors and thus more suitable for long-read data. T. Pan et al's work [30] provides interesting optimizations to HipMer and others' distributed memory *k-mers* analysis design, but focuses on the general problem of counting *k-mers* in distributed memory, and not also the construction and computation on the read overlap graph.

12 CONCLUSIONS

We presented diBELLA, a long read to long read aligner for distributed memory platforms that deals with the unique problem of aligning noisy reads to each other, making it possible to analyze data sets that are too large for a single shared memory and or making heroic computations routine. Alignment is a key step in long read assembly and other analysis problems, and often the dominant computation. diBELLA avoids the expensive all-to-all alignment by looking for short, error-free seeds (*k-mers*) and using those to identify potentially overlapping reads. We believe this is the first implementation of such a long read-to-read aligner designed for distributed memory. In addition to the independent work of performing pairwise alignment on many reads, our implementation takes advantage of global information across the input data set, such as the total count of each *k-mer* used for filtering errors and the distribution of overlaps to distribute load. We performed a thorough performance analysis on 3 leading HPC platforms as well as one commodity cloud offering, showing good parallel performance of our approach, especially for realistic scenarios that perform multiple alignments per pair of input reads. While the HPC systems offer superior performance to the cloud, all of them benefit from the multi-node parallelization. The application is dominated by irregular all-to-all style communication and the study reveals some of the performance anomalies on particular systems, as well as general scaling issues at larger node counts.

We believe that in addition to being a useful tool for bioinformatics, either standalone or as part of a larger pipeline, diBELLA also represents an important parallel workload to understand and drive the design of future HPC system and communication libraries, and a basis for future optimizations in single node optimization for pairwise alignment while retaining the efficiency sparse nature of the interactions.

ACKNOWLEDGMENTS

This work was supported in part by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration and by the National Science Foundation. This research used resources of the National Energy Research Scientific Computing Center (NERSC) under contract No. DE-AC02- 05CH11231, and the Oak Ridge Leadership Computing Facility under DE-AC05-00OR22725. all supported by the Office of Science of the U.S. Department of Energy. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred. AWS Cloud Credits were provided through Amazon Web Services and PI Benjamin Brock.

REFERENCES

[1] Mohammed Alser, Hasan Hassan, Akash Kumar, Onur Mutlu, and Can Alkan. 2019. Shouji: A Fast and Efficient Pre-Alignment Filter for Sequence Alignment. *Bioinformatics* (2019).

[2] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. 1990. Basic Local Alignment Search Tool. *Journal of Molecular Biology* 215, 3 (1990), 403–410.

[3] Konstantin Berlin, Sergey Koren, Chen-Shan Chin, James P Drake, Jane M Landolin, and Adam M Phillippy. 2015. Assembling large genomes with single-molecule sequencing and locality-sensitive hashing. *Nature biotechnology* 33, 6 (2015), 623–630.

[4] Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (1970), 422–âĂŞ426. https://doi.org/10.1145/362686.362692

[5] Mark J. Chaisson and Glenn Tesler. 2012. Mapping single molecule sequencing reads using basic local alignment with successive refinement (BLASR): application and theory. *BMC Bioinformatics* 13, 1 (19 Sep 2012), 238. https://doi.org/10.1186/1471-2105-13-238

[6] Chen Shan Chin, David H. Alexander, Patrick Marks, Aaron A. Klammer, James Drake, Cheryl Heiner, Alicia Clum, Alex Copeland, John Huddleston, Evan E. Eichler, Stephen W. Turner, and Jonas Korlach. 2013. Nonhybrid, finished microbial genome assemblies from long-read SMRT sequencing data. *PLoS Medicine* 10, 6 (4 2013), 563–569. https://doi.org/10.1038/nmeth.2474

[7] Chen-Shan Chin, Paul Peluso, Fritz J Sedlazeck, Maria Nattestad, Gregory T Concepcion, Alicia Clum, Christopher Dunn, Ronan O'Malley, Rosa Figueroa-Balderas, Abraham Morales-Cruz, et al. 2016. Phased diploid genome assembly with single-molecule real-time sequencing. *Nature methods* 13, 12 (2016), 1050.

[8] Chen-Shan Chin, Paul Peluso, Fritz J Sedlazeck, Maria Nattestad, Gregory T Concepcion, Alicia Clum, Christopher Dunn, Ronan O'Malley, Rosa Figueroa-Balderas, Abraham Morales-Cruz, Grant R Cramer, Massimo Delledonne, Chongyuan Luo, Joseph R Ecker, Dario Cantu, David R Rank, and Michael C Schatz. 2016. Phased diploid genome assembly with single-molecule real-time sequencing. *Nature Methods* 13 (17 10 2016), 1050 EP –. http://dx.doi.org/10.1038/nmeth.4035

[9] Andreas Döring, David Weese, Tobias Rausch, and Knut Reinert. 2008. SeqAn an efficient, generic C++ library for sequence analysis. *BMC bioinformatics* 9, 1 (2008), 11.

[10] Marquita Ellis, Evangelos Georganas, Rob Egan, Steven Hofmeyr, Aydın Buluç, Brandon Cook, Leonid Oliker, and Katherine Yelick. 2017. *23rd International European Conference on Parallel and Distributed Computing (Euro-Par 2017).*

[11] Evangelos Georganas. 2016. *Scalable Parallel Algorithms for Genome Analysis.* Ph.D. Dissertation. EECS Department, University of California, Berkeley.

[12] Evangelos Georganas, Aydın Buluç, Jarrod Chapman, Leonid Oliker, Daniel Rokhsar, and Katherine Yelick. 2015. MerAligner: A fully parallel sequence aligner. In *2015 IEEE International Parallel and Distributed Processing Symposium.* IEEE, Hyderabad, India, 561–570.

[13] Evangelos Georganas, Aydın Buluç, Jarrod Chapman, Steven Hofmeyr, Chaitanya Aluru, Rob Egan, Leonid Oliker, Daniel Rokhsar, and Katherine Yelick. 2015. HipMer: An Extreme-Scale De Novo Genome Assembler. *27th ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC 2015)* (November 2015).

[14] Giulia Guidi, Marquita Ellis, Daniel Rokhsar, Katherine Yelick, and Aydın Buluç. 2018. BELLA: Berkeley Efficient Long-Read to Long-Read Aligner and Overlapper. *bioRxiv* (2018), 464420.

[15] Runxin Guo, Yi Zhao, Quan Zou, Xiaodong Fang, and Shaoliang Peng. 2018. Bioinformatics applications on apache spark. *GigaScience* 7, 8 (2018), giy098.

[16] Vasanthan Jayakumar and Yasubumi Sakakibara. 2017. Comprehensive evaluation of non-hybrid genome assembly tools for third-generation PacBio long-read sequence data. *Briefings in Bioinformatics* (2017), bbx147. https://doi.org/10.1093/bib/bbx147

[17] Govinda M Kamath, Ilan Shomorony, Fei Xia, Thomas A Courtade, and N Tse David. 2017. HINGE: long-read assembly achieves optimal repeat resolution. *Genome research* 27, 5 (2017), 747–756.

[18] Mikhail Kolmogorov, Jeffrey Yuan, Yu Lin, and Pavel A Pevzner. 2019. Assembly of long, error-prone reads using repeat graphs. *Nature biotechnology* (2019), 1.

[19] Sergey Koren, Brian P Walenz, Konstantin Berlin, Jason R Miller, Nicholas H Bergman, and Adam M Phillippy. 2017. Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation. *Genome Research* 27, 5 (05 2017), 722–736. https://doi.org/10.1101/gr.215087.116

[20] Sergey Koren, Brian P Walenz, Konstantin Berlin, Jason R Miller, Nicholas H Bergman, and Adam M Phillippy. 2017. Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation. *Genome research* 27, 5 (2017), 722–736.

[21] Heng Li. 2016. Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences. *Bioinformatics* 32, 14 (2016), 2103–2110.

[22] Heng Li. 2018. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics* 34, 18 (2018), 3094–3100.

[23] Yu Lin, Jeffrey Yuan, Mikhail Kolmogorov, Max W Shen, Mark Chaisson, and Pavel A Pevzner. 2016. Assembly of long error-prone reads using de Bruijn graphs. *Proceedings of the National Academy of Sciences* 113, 52 (2016), E8396–E8405.

[24] Nicholas James Loman, Joshua Quick, and Jared T Simpson. 2015. A complete bacterial genome assembled de novo using only nanopore sequencing data. *bioRxiv* (2015). https://doi.org/10.1101/015552 arXiv:https://www.biorxiv.org/content/early/2015/02/20/015552.full.pdf

[25] Guillaume Marçais and Carl Kingsford. 2011. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics* 27, 6 (2011), 764–770.

[26] David W Mount. 2004. Sequence and genome analysis. *Bioinformatics: Cold Spring Harbour Laboratory Press: Cold Spring Harbour* 2 (2004).

[27] Gene Myers. 2014. Efficient local alignment discovery amongst noisy long reads. In *International Workshop on Algorithms in Bioinformatics.* Springer, 52–67.

[28] Niranjan Nagarajan and Mihai Pop. 2009. Parametric complexity of sequence assembly: theory and applications to next generation sequencing. *Journal of computational biology* 16, 7 (2009), 897–908.

[29] Thomas P. Niedringhaus, Denitsa Milanova, Matthew B. Kerby, Michael P. Snyder, and Annelise E. Barron. 2011. Landscape of Next-Generation Sequencing Technologies. *Analytical Chemistry* 83, 12 (2011), 4327–4341. https://doi.org/10.1021/ac2010857 arXiv:https://doi.org/10.1021/ac2010857 PMID: 21612267.

[30] Tony C Pan, Sanchit Misra, and Srinivas Aluru. 2018. Optimizing high performance distributed memory parallel hash tables for DNA k-mer counting. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis.* IEEE, 135–147.

[31] Adam M Phillippy, Michael C Schatz, and Mihai Pop. 2008. Genome assembly forensics: finding the elusive mis-assembly. *Genome biology* 9, 3 (2008), R55.

[32] Temple F. Smith and Michael S. Waterman. [n. d.]. Identification of Common Molecular Subsequences. *Journal of Molecular Biology* 147, 1 ([n. d.]), 195âĂŞ197. https://doi.org/10.1016/0022-2836(81)90087-5

[33] Yatish Turakhia, Gill Bejerano, and William J Dally. 2018. Darwin: A genomics co-processor provides up to 15,000 x acceleration on long read assembly. In *ACM SIGPLAN Notices*, Vol. 53. ACM, 199–213.

[34] Leslie G. Valiant. 1990. A Bridging Model for Parallel Computation. *Commun. ACM* 33, 8 (Aug. 1990), 103–111. https://doi.org/10.1145/79173.79181

[35] Chuan-Le Xiao, Ying Chen, Shang-Qian Xie, Kai-Ning Chen, Yan Wang, Yue Han, Feng Luo, and Zhi Xie. 2017. MECAT: fast mapping, error correction, and de novo assembly for single-molecule sequencing reads. *nature methods* 14, 11 (2017), 1072.

[36] Wenyu Zhang, Jiajia Chen, Yang Yang, Yifei Tang, Jing Shang, and Bairong Shen. 2011. A practical comparison of de novo genome assembly software tools for next-generation sequencing technologies. *PloS one* 6, 3 (2011), e17915.

[37] Zheng Zhang, Scott Schwartz, Lukas Wagner, and Webb Miller. 2000. A greedy algorithm for aligning DNA sequences. *Journal of Computational biology* 7, 1-2 (2000), 203–214.