Parallel *de novo* Assembly of Long Reads with `Ananas`

Frank Austin Nothaft

Abstract

Genome assembly is a critical and expensive computational biology problem. The computational cost of the problem is exacerbated by the difficulty of parallelizing the algorithm. In this paper, we look at the overlap-layout-concensus formulation of the genome assembly problem. We introduce the tool `Ananas`, which demonstrates linear scalability on a small dataset. `Ananas` is built on top of the `ADAM` genomics API, using Apache `Spark` and `GraphX`, and parallelizes all stages of the algorithm.

1 Introduction

One of the canonical computational biology problems is the genome assembly problem. In the *de novo* genome assembly problem, the goal is to sequence "reads" from the currently unknown genome of an organism, and to "assemble" these fragments into long contiguous strings (contigs) that best represent the underlying genome of the organism [16]. There are two canonical graph-based formulations of this problem: we can either use a *de Bruijn* [16] or an *overlap* graph [10] to reconstruct genomic sequences.

In this paper, we look at parallelizing an overlap graph based assembler. We have chosen the overlap graph approach, because although the *de Bruijn* graph based approach is lower cost, overlap graphs are better at resolving the repeat structure of genomes. This is caused by *de Bruijn* graphs collapsing down repeats that are larger than the k-mer size used when creating the graph[1]. Additionally, we are interested in investigating data generated with new sequencing technologies, such as Pacific Biosciences' Single Molecule Real Time sequencing technology (SMRT, [3]) or the Oxford Nanopore technology [2]. These sequencing technologies generate very long reads[2]—these long reads are easier to use for generating high quality overlaps, and have lower asymptotic overlapping cost.

In an overlap graph based assembler, the assembly process can be broken up into three stages, which is known as the OLC process:

1. First, we find the **overlaps** between all reads.
2. Then, we **layout** these reads as a graph, and simplify the graph.
3. Finally, we find the **concensus** sequence that this graph describes.

In the remainder of this paper, we introduce `Ananas`, a parallel/distributed tool that implements the OLC process. `Ananas` is implemented on top of `ADAM` [9, 14], an Apache `Spark`-based [20, 21]

[1] k-mers are the k-letter length substrings that are used as the vertices in a *de Bruijn graph.*

[2] These technologies generate reads with length in excess of > 10 kilobase pairs (kbp). The most common sequencing vendor (Illumina) generates reads with length typically less than ~ 250bp.

API for processing genomic data. We implement a MinHash signature based method for computing read overlaps [1], which is parallelized using `Spark`. We then used the Apache `GraphX` [4, 19] to build an overlap graph from the read overlaps. We reduced the overlap graph into a string graph [11] and walked this graph to emit contigs. Our approach is fully parallel and scales linearly out to four machines on a modest (1.5GB) input dataset. `Ananas` is released as open source software under the Apache 2 license and is available from `https://www.github.com/fnothaft/ananas`.

2 Methods

Although the read overlapping methods are close to embarassingly parallel, the graph traversal methods are not as obviously data parallel. In this section, we will discuss our implementations of both methods.

2.1 Parallel Read Overlapping

When computing the pairwise overlaps of reads, we are trying to identify reads that are highly similar. Although traditional overlapping algorithms tend to be index-based [12], we use a MinHash signature based method which was originally proposed by Berlin et al [1]. While a naïve overlapper will attempt to compute the full $\mathcal{O}(n^2)$ pairwise overlaps, a locality sensitive hashing (LSH) scheme can be used to reduce the number of overlaps computed to $\mathcal{O}(\frac{n^2}{b})$, where n is the number of reads in the dataset and b is the number of buckets used for the LSH [6]. Additionally, using MinHash signatures allows us to decrease the cost of approximately comparing two sequences. Comparing two MinHash signatures has time complexity $\mathcal{O}(s)$, where s is the signature length, while the time complexity of local sequence alignment is $\mathcal{O}(n_1 n_2)$, where n_1, n_2 are the lengths of the two sequences being compared [18].

MinHash signatures are useful as they can be used to rapidly compute approximate similarity measures of two sequences. First, for two sets S_1, S_2, the Jaccard similarity of these two sets is given by equation (1). If the two sets yield length s signatures $S_1 \to M_1, S_2 \to M_2$, their similarity can be approximated by equation (2).

$$\text{Jaccard Similarity} = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|} \tag{1}$$

$$\text{Jaccard Similarity} \sim \frac{\sum_{i=1}^{s} I^{[M_1^i = M_2^i]}}{s} \tag{2}$$

The process for computing the signature of a read is straightforward. First, we generate a length s array R of random integers, where s is the signature length we are using. Then, we "shingle" each element of our input dataset: shingling extracts length n token overlaps, where n is a user provided parameter. The semantics of the shingles depends on the input dataset: for traditional document similarity comparison, n-gram shingles are typically used, where an n-gram is a sequence of n consecutive words and where consecutive n-grams have $n-1$ words in common [6]. In our application, we use k-mers and set $k = 16$, which allows us to bit pack each k-mer into a 32-bit integer[3]. Once we have generated the shingles, we apply a hash function to each shingle that maps

[3]We use *canonical* k-mers. To generate a canonical k-mer, we take the lexicographically first k-mer out of this k-mer and it's reverse complement k-mer. To generate the reverse complement of a k-mer, we reverse the k-mer string and "flip" the bases in the string using the `A` $\leftrightarrow$ `T`, `C` $\leftrightarrow$ `G` mapping. As we will discuss later, this is critical as reads may be sequenced from either strand of a DNA molecule.

$T \mapsto \text{Int}$, where T is the type of the shingle. In our case, we are bitpacking the k-mers into integers, and we are able to just emit the integer value of the bitpacked canonical k-mer as our hash. Given the length $n_i - k + 1$ hash vector H_i of read i (where n_i is the read length and $k = 16$ is the k-mer shingle length) and the length s array R of random integers, we compute the signature M_i of read i using equation (3), where $\oplus$ is the bitwise XOR of two integers.

$$M_i[j] = \min_{h \in H_i} h \oplus R[j], \forall j \in \{0, \ldots, s-1\} \tag{3}$$

To run this process, we start by converting each read sequence into an indexed *de Bruijn* sequence, which we have introduced in [13][4]. This process is embarassingly parallel and is implemented by `map`ping over an input Resilient Distributed Dataset (RDD, see [20]). Simultaneously, we generate an array of hashes for use in generating the MinHash signatures of each sequence and broadcast this array of hashes to each node.

As noted above, once these signatures have been computed, we still need to compute all pairwise alignments. While we do support computing the full cartesian overlap, we rely in practice on the LSH scheme illustrated by Berlin et al [1] and formalized by Leskovec et al [6]. In this approach, we map each signature to b buckets and only estimate overlaps within these buckets. We require that $s \bmod b = 0$. The significance of this is that $p = \frac{b}{s}$ hashes from the MinHash signature of read R_i are used to determine the buckets to send R_i to. To implement this operation, we perform a `flatMap` over all read signatures, and compute b bucket IDs. The bucket IDs are tuples of integers and length p integer arrays. We compute the ID tuple T_i^b for bucket b from read R_i with signature M_i using equation (4).

$$T_i^b = (b, M_i[pb : p(b+1) - 1]) \tag{4}$$

Once we have `flatMap`ped all reads, we `groupBy` each key to collect an array containing all reads that fall into each bucket. We then run a pairwise comparison of all signatures inside of each bucket. It is worth noting that two reads can pairwise map into multiple buckets (i.e., both R_1 and R_2 will map into bucket $(0, [h_0, h_1])$ and bucket $(1, [h_2, h_3])$ if $M_1[0:3] = M_2[0:3]$). This can lead to us emitting duplicate overlaps. To handle this, we only evaluate a pairwise overlap if this is the "earliest" bucket where the two reads could have met. We impose an ordering by looking at the integer in the tuple; in the example given earlier, we would only evaluate the overlap of the two reads in bucket $(0, [h_0, h_1])$.

When we evaluate an overlap, we emit the approximate similarity score given by equation (2). We immediately filter all overlaps where the estimated overlap length is below a user provided threshold (the estimated overlap o_e is given by $s_{1,2} \max n_1, n_2$, where $s_{1,2}$ is the estimated similarity between the two reads and n_1, n_2 are the lengths of the two reads). If the read pair passes the similarity threshold, we then compute the exact overlap length. We do this by finding the size of the intersection of the two indexed *de Bruijn* sequences. Additionally, we compute several other properties:

[4] An indexed *de Bruijn* sequence is a list of k-mers which are tagged with the position where the k-mer occurred in the graph. Since two adjacent k-mers in a sequence trivially satisfy the $k-1$ base condition necessary to place an edge in a *de Bruijn* graph, this list is an equivalent representation of a graph. While we solely use this structure to shingle reads for our hashing scheme, this structure is highly applicable in the local sequence assembly case (see ch. 3 of our other text [13], and we plan to make further use of the datastructure in a future version of `Ananas`.

- Does this overlap switch strands? When we create the canonical k-mers in the *de Bruijn* sequence, we log whether the k-mers were originally canonical or not. If the overlapping k-mers had different original canonicality, we note that the reads are sequenced from different strands.
- At what end of the read does the overlap start from? Does the overlap contain the beginning or end of the read, or is the entire read overlapped by a longer read?

While the approach outlined in this section improves the runtime complexity of overlapping, it comes at the cost of data being duplicated at a factor of b. In §4, we discuss a possible solution to this problem.

2.2 Parallel Graph Simplification and Traversal

Once we have emitted the overlaps, we use these overlaps to create a graph in `GraphX` [4, 19]. From here, we run transitive reduction to simplify the graph, and then walk the graph to emit contigs. The graph operations were complicated because `GraphX` uses an underlying representation that assumes a directed graph. In a two-strand sequencing process, overlap graphs are bidirected.

The transitive reduction algorithm for reducing an overlap graph into a string graph was introduced by Myers [11]. Our goal is to remove all redundant edges while maintaining all true paths by simplifying the graph. While transitive reduction is well defined for a directed, acyclic graph[5] its definition is unclear for a bidirected, possibly cyclic graph. While Myers [11] effectively defines the transitive reduction of an overlap graph to be any graph that has not oversimplified the presence of a sequence variant, we find this definition to be imprecise. We define the following rules for transitively reducing an overlap graph:

- Disregard all reads that are fully overlapped by another read. They can trivially be merged into the longest read that fully contains them.
- Treat all other edges as undirected. If there is an admissible triangle containing reads A, B, C, each node should greedily pick to keep the edge with the longest overlap length at it's start and end. E.g., if A—B has overlap of 500 bp, B—C has overlap of 500 bp, and A—C has overlap of 300 bp, we will choose to keep A—B—C and will eliminate A—C.

Note in the rules above that we only process *admissible* triangles. Specifically, we only process triangles where all read alignments are concordant. Temporarily assuming a single stranded sequencing protocol, a triangle $\triangle ABC$ would be admissible if the A—B overlap began at the end of A and the start of B, the B—C overlap begin at the end of B and the start of C, and the A—C overlap began at the end of A and the start of C. Additionally, if the A—B overlap is longer than the A—C overlap, the B—C overlap must also be longer than the A—C overlap. In our implementation, we keep any edges where a single node votes to keep the edge. Given the restrictions on admissibility above, this is provably correct. In this proof, we assume a single stranded sequencing protocol for simplicity. A directed edge from $A \to B$ implies that there is an overlap from the end of A to the beginning of B.

Proof. First, consider the case where the triangle is inadmissible:

[5] Specifically, at vertex A, with neighbors $\mathcal{N} = B, \ldots, Z$, we preserve reachability if we eliminate all edges from $A \to n, n \in \mathcal{N}$ if there exists a walk $A \to o_1 \to \cdots \to n$.

- If the triangle is inadmissible because the overlap directions are not concordant, this implies that the triangle describes a loop in the graph. Specifically, $A \to B$, $B \to C$, $C \to A$ implies that there is an inversion of the sequence contained in read A. This loop cannot be reduced, but vertices B, C can be merged.
- If the triangle is inadmissible because of the lengths of the overlaps are discordant, this implies that the triangle describes a large allelic divergence, such as an insertion/deletion variant. Since this triangle describes two sequence variants, it cannot be reduced nor can any vertices be merged.

In the case that the triangle is admissible, A, B, and C will all vote for the same edges. A proof of this is trivial. However, for this proof to be valid, our approach must satisfy a graph with multiple triangles. This is true. Observe a graph A—B—C—D—E, A—C—E, B—D. If the simplest reducible graph is A—B—C—D—E, triangles $\triangle ABC$, $\triangle BCD$, and $\triangle CED$ will all be admissible and will vote for A—B—C—D—E. □

While this problem statement is straightforward, it is somewhat difficult to map to the vertex-centric `GraphX` API. To implement this algorithm, each vertex collects its edge attributes, which is then joined against the graph. We then run an iteration of message passing where each vertex sends its edges to all neighbor vertices. At this point, each vertex knows what edges it has, and what edges each neighbor has. We then separate the neighbor edges into overlaps that start at the start/end of the read, and run a tail-recursive algorithm that implements the triangle voting process. Once each node has run its voting process, we run a subgraph operation. This subgraph operation is used to keep only edges who were voted for. No nodes are dropped from the graph.

Once we have run transitive reduction on the graph, we emit contigs. While conceptually we could merge all edges that have unit in/out degree into a larger node and then estimate node copy number [11], we decided not to implement these features for two reasons:

- Merging edges together is only necessary for estimating node copy number, and is difficult to implement in the `GraphX` API, which assumes that graphs are immutable (with the exception of allowing subgraphs to be created).
- The formulation of node copy number presented by Myers [11] requires the development of a `GraphX`-based solver for min cost flow, which was intractible given the project timelines.

To generate contigs, we ran a graph labeling algorithm that is similar to `GraphX`'s implementation of connected components labeling. We started by collecting all edges to each vertex, and sorting these edges into start/end vertices. From here, we ran the following algorithm:

1. If a node has either no start or no end vertices, it chooses to send a message in the initial iteration. This message contains the node ID. This ID is used to label the generated contig.
2. At each iteration, if a node has received a message, it logs the ID of the node that started the message passing in the initial iteration, and the ID of the node that message came from. If the node that is sending a message has any edges on the side *opposite* from where the message came from (e.g., if the message came in from the start of the read, it needs to go out on an edge from the end of the read), we send a message on that edge in the next round.

This algorithm is implemented using `GraphX`'s `Pregel` abstraction [8] and runs until no more messages can be sent. There are several important implementation details:

- If our graph contains a single connected component where all but two edges have in/out degree of one, we will send two messages in the inital iteration. These two messages will describe the same contig. To handle this, when the two messages reach the same node, we pick the message with the lowest starting node ID and relabel all nodes tagged with the previous starting node ID. This is why we store both the starting node ID and the message sender ID.
- In `GraphX`'s implementation of `Pregel`, node updates can only occur on update. This implies that vertex state cannot be mutated when sending messages. To ensure that a node does not send multiple messages to the same vertex, we choose the message recipients for the next iteration when we are processing the received messages. Additionally, we track an iteration count with each message. Since `GraphX` only activates vertices that received a message in the last iteration, we are able to track what messages to send by looking for the labels that have the highest iteration count.

Once we have run this algorithm, we `flatMap` each vertex sequence and key by the starting node ID. We also track the strandedness relative to the starting node, and the iteration in which the node was tagged. We then `groupBy` the key (the starting node ID), sort all subsequences, and reverse complement sequences as necessary. Finally, we use `ADAM` [9, 14] to save all contigs from `Spark` in a parallel manner.

3 Results

We evaluated `Ananas` on Amazon's Elastic Compute 2 (EC2) cloud, using the `r3.2xlarge` instance type. We evaluated `Ananas` on PacBio [3] sequence of the `E. coli` virus[6]. We evaluated on two and four nodes, and saw linear speedup between these two nodes. This can be seen in Figure 1.

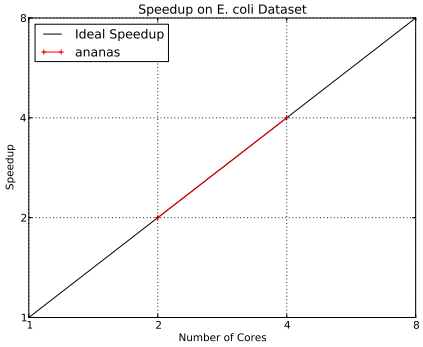

Figure 1: Speedup across nodes

For our speedup experiments, we used an overlap cutoff of 500 bp and 4 buckets. With this configuration, approximately 10% of time was spent on overlapping, 70% was spent on transitive reduction, and 20% was spent on contig generation. Although the speedup is linear, the performance is poor. On four machines, it took approximately 50 minutes to process the *E. coli* dataset, which

[6]This sequence was captured using an RS II with P6/C4 chemistry. The dataset is available from `https://github.com/PacificBiosciences/DevNet/wiki/E.-coli-20kb-Size-Selected-Library-with-P6-C4` and was preprocessed with `SMRT`© `Analysis` to convert to FASTA format.

is worse than expected. However, it is worth noting that while this number is worse than hoped, 50 minutes on four eight core machines represents approximately 27 core hours, which is approximately 5× better than the performance of the error correction step for a hybrid PacBio assembly system on *E. coli* data [5]. In future work, we plan to perform a more thorough evaluation against prior tools.

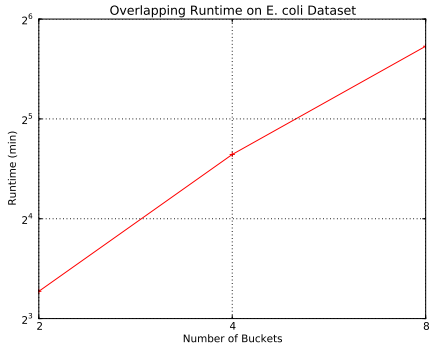

Figure 2: Overlapping performance vs. bucket size

Overlapping performance should scale with the number of buckets in use. Specifically, a two-fold increase in the number of buckets we evaluate leads to a two-fold increase in data (due to replication during the `flatMap`) and a two-fold increase in the number of comparisons performed. To evaluate this, we swept the number of buckets used during overlapping from two to four to eight and performed overlapping on two machines. This is shown in Figure 2. This trend held close to true; while the increase in runtime was slightly greater than two going between two and four buckets, it was slightly less than two when going from four to eight buckets.

4 Discussion

We will continue to improve `Ananas` after this project has completed. We are interested in improving performance via better overlapping schemes, and by improving the cost of transitive reduction. Additionally, we would like to move to a better scheme for generating contigs.

As explained in §2.1, our overlapping performance is impacted by the bucket count, b, used for overlapping. One approach that can be used to further improve overlapping performance is to move to a multi-probe LSH approach [7]. In this scheme, a likelihood function is used to predict the buckets to materialize. This allows us to eliminate "unproductive" buckets, which leads to a reduction in the number of reads duplicated and the number of comparisons performed.

Currently, our runtime is dominated by the transitive reduction phase. We are looking into ways to improve the performance of this phase. Specifically, we think there may be ways to recast the implementation. Although the algorithm we introduce in §2.2 is cast as a triangle-based algorithm, due to the APIs provided by `GraphX`, we implement the algorithm by materializing edges to vertices. We may be able to improve performance by modifying the `GraphX` core.

Finally, we would like `Ananas` to support variable ploidy assembly. While `Ananas` currently assembles contigs assuming that the underlying organism is haploid (there is a single copy of each chromosome, and thus a single path through the graph), this assumption is unrealistic for many organisms [15]. To do this, we would like to call the *copy number* of each segment: Myers [11]

describes a min cost flow based approach for calling the number of paths through each node by reweighting the edges. Although the approach presented by Myers can be efficiently implemented, it is not clearly justified from a theoretical perspective. We propose using a likelihood based formulation for calling edge multiplicity (and thus vertex copy number from in/out degree). An approach similar to `CGAL` [17] could be used; `CGAL` itself cannot be used as it assumes a mate pair[7] structure that is not applicable to long read datasets.

5 Conclusion

In this paper, we have introduced `Ananas`, a parallel/distributed *de novo* assembler for long read genomic data. `Ananas` is built on top of `ADAM` [9, 14], Apache `Spark` [20, 21], and `GraphX` [4, 19]. We have evaluated `Ananas` on *E. coli* data and found that it scales linearly across four machines. Additionally, we have demonstrated a parallel way to implement approximate MinHash-based overlapping, and have provided a formalized algorithm for transitive reduction over overlap graphs. This algorithm can be probably parallelized across all triangles in a graph.

References

[1] K. Berlin, S. Koren, C.-S. Chin, J. Drake, J. M. Landolin, and A. M. Phillippy. Assembling large genomes with single-molecule sequencing and locality sensitive hashing. *bioRxiv*, page 008003, 2014.

[2] J. Clarke, H.-C. Wu, L. Jayasinghe, A. Patel, S. Reid, and H. Bayley. Continuous base identification for single-molecule nanopore DNA sequencing. *Nature nanotechnology*, 4(4):265–270, 2009.

[3] J. Eid, A. Fehr, J. Gray, K. Luong, J. Lyle, G. Otto, P. Peluso, D. Rank, P. Baybayan, B. Bettman, et al. Real-time DNA sequencing from single polymerase molecules. *Science*, 323(5910):133–138, 2009.

[4] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph processing in a distributed dataflow framework. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[5] S. Koren, M. C. Schatz, B. P. Walenz, J. Martin, J. T. Howard, G. Ganapathy, Z. Wang, D. A. Rasko, W. R. McCombie, E. D. Jarvis, et al. Hybrid error correction and de novo assembly of single-molecule sequencing reads. *Nature biotechnology*, 30(7):693–700, 2012.

[6] J. Leskovec, A. Rajaraman, and J. D. Ullman. *Mining of massive datasets*. Cambridge University Press, 2014.

[7] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe LSH: efficient indexing for high-dimensional similarity search. In *Proceedings of the International Conference on Very Large Databases (VLDB '07)*, pages 950–961. VLDB Endowment, 2007.

[8] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*, pages 135–146. ACM, 2010.

[7] In many "short read" sequencing techniques, sequencing libraries are constructed with read pairs. In this paradigm, two reads are generated from a single DNA sequence.

[9] M. Massie, F. Nothaft, C. Hartl, C. Kozanitis, A. Schumacher, A. D. Joseph, and D. A. Patterson. ADAM: Genomics formats and processing patterns for cloud scale computing. Technical report, UCB/EECS-2013-207, EECS Department, University of California, Berkeley, 2013.

[10] E. W. Myers. Toward simplifying and accurately formulating fragment assembly. *Journal of Computational Biology*, 2(2):275–290, 1995.

[11] E. W. Myers. The fragment assembly string graph. *Bioinformatics*, 21(suppl 2):ii79–ii85, 2005.

[12] G. Myers. Efficient local alignment discovery amongst noisy long reads. In *Algorithms in Bioinformatics*, pages 52–67. Springer, 2014.

[13] F. A. Nothaft. Scalable genome resequencing with ADAM and avocado. Master's thesis, Computer Science Division, University of California, Berkeley, May 2015.

[14] F. A. Nothaft, M. Massie, T. Danford, Z. Zhang, U. Laserson, C. Yeksigian, J. Kottalam, A. Ahuja, J. Hammerbacher, M. Linderman, M. J. Franklin, A. D. Joseph, and D. A. Patterson. Rethinking data-intensive science using scalable analytics systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*, 2015.

[15] B. Paten, A. Novak, and D. Haussler. Mapping to a reference genome structure. *arXiv preprint arXiv:1404.5010*, 2014.

[16] P. A. Pevzner, H. Tang, and M. S. Waterman. An Eulerian path approach to DNA fragment assembly. *Proceedings of the National Academy of Sciences*, 98(17):9748–9753, 2001.

[17] A. Rahman and L. Pachter. CGAL: computing genome assembly likelihoods. *Genome Biology*, 14(1):R8, 2013.

[18] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.

[19] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. GraphX: A resilient distributed graph system on Spark. In *First International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM, 2013.

[20] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI '12)*, page 2. USENIX Association, 2012.

[21] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the USENIX Conference on Hot Topics in Cloud Computing (HotCloud '10)*, page 10, 2010.