A Parallel Connectivity Algorithm for de Bruijn Graphs in Metagenomic Applications

Patrick Flick[*], Chirag Jain[*], Tony Pan[*] and Srinivas Aluru
Georgia Institute of Technology, Atlanta, Georgia, USA
{patrick.flick, cjain, tony.pan}@gatech.edu, aluru@cc.gatech.edu

ABSTRACT

Dramatic advances in DNA sequencing technology have made it possible to study microbial environments by direct sequencing of environmental DNA samples. Yet, due to the huge volume and high data complexity, current de novo assemblers cannot handle large metagenomic datasets or fail to perform assembly with acceptable quality. This paper presents the first parallel solution for decomposing the metagenomic assembly problem without compromising the post-assembly quality. We transform this problem into that of finding weakly connected components in the de Bruijn graph. We propose a novel distributed memory algorithm to identify the connected subgraphs, and present strategies to minimize the communication volume. We demonstrate the scalability of our algorithm on a soil metagenome dataset with 1.8 billion reads. Our approach achieves a runtime of 22 minutes using 1280 Intel Xeon cores for a 421 GB uncompressed FASTQ dataset. Moreover, our solution is generalizable to finding connected components in arbitrary undirected graphs.

Keywords

Metagenomic assembly, soil microbiology, de Bruijn graph, connected component labeling.

1. INTRODUCTION

Metagenomics is an important emerging area in bioinformatics that involves the study of microbial genomes directly obtained from environmental samples. Whereas single genome sequencing extracts short sequences (or *reads*) from samples containing multiple copies of the same genome or multiple cells from the same organism (e.g. bacterial culture, human tissue, tumor cells, etc), metagenomic sequencing samples the aggregate genomes of entire microbial communities. In a complex natural environment, such as soil or human gut, metagenomic samples can potentially contain thousands or even millions of species [8]. High quality analysis of these microbial genomes requires sufficient coverage during sequencing, resulting in huge data volumes. For instance, the grand challenge Iowa corn soil data set sequenced at the Joint Genome Institute contains 1.8 billion sequencing reads [18].

Reconstructing the constituent genomes from metagenomic sequencing reads is still an open problem. Individual reads sequenced by widely used sequencing platforms such as Illumina do not by themselves contain meaningful biological information due to their short lengths (currently about 100-125 base pairs). Prior to analysis, the genomes must be reconstructed partially or completely from the reads. This is done via *resequencing* where reads are mapped to reference genomes, or by *de novo assembly* where genomes are reconstructed by finding sequences of overlapping reads. Resequencing is therefore well suited for detecting variations in genomes, while de novo assembly is useful when reference genomes do not exists or are undesirable. Currently, most short read de novo assemblers utilize *de Bruijn graphs* to encode the overlap information. In a de Bruijn graph, vertices are strings of size k, called *k-mers*, extracted from the source reads. The edges represent $(k-1)$-long suffix-prefix overlaps between k-mers, and correspond to length $k+1$ substrings in a read. A read therefore exists as a path in the de Bruijn graph. Finding a sequence of overlapping reads can then be accomplished by finding an Eulerian path in the graph, visiting each edge exactly once. In practice, assembly using de Bruijn graphs is significantly more complex due to errors in reads, chimeras (two unrelated reads joining together), contaminating sequences, repeats within and across genomes, etc.

The assembly problem becomes highly compute and memory intensive as the size and complexity of the data set increases. In the absence of errors, the size of the de Bruijn graph is equivalent to the number of unique k-mers in the data set, thus is bounded by the number of base pairs in the genome(s). For metagenomic assembly, however, the presence of large number of species substantially increases this bound. Currently, de novo assembly tools designed for single genome would require multiple terabytes of memory to store the graphs built from large metagenomic data sets [20]. For example, Howe *et al.* [12] reported that Velvet [26] failed to assemble a 3.3-billion-read metagenomic data set on a single computer with 500 GB memory.

Recently, Howe *et al.* [12] presented the first successful attempt of the assembly of two large soil metagenomic data

[*]The three authors should be regarded as joint first authors, equally contributing to the work.

SC '15, November 15 - 20, 2015, Austin, TX, USA
© 2015 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-3723-6/15/11...$15.00
DOI: http://dx.doi.org/10.1145/2807591.2807619

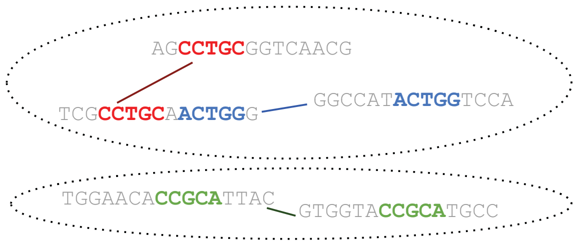

Figure 1: Example of graph connectivity with 5 reads and k-mer size = 5.

sets. They discovered that the high species level heterogeneity in the metagenomic data sets leads to a large number of disjoint connected components in the de Bruijn graph. This property was then exploited to partition the reads into disjoint sets and assemble each of them independently (See Fig. 1 for example). One interesting point reported in this study was the existence of 31 million and 56 million connected components with more than 5 reads in the Iowa corn and prairie soil data sets, respectively. This observation can be leveraged as a way to introduce parallelism to the metagenomic assembly process by way of read partitioning. However, there is one problem that needs to be addressed before reaching that stage: How do we partition a data set with 1.8 billion reads, or equivalently, how do we identify the connected components in a de Bruijn graph with more than 100 billion edges?

None of the previous known methods for computing connected components is viable for such large problem sizes. To address this limitation, we present a novel distributed memory parallel algorithm for connected component labeling, which is scalable to very large sparse graphs and over a thousand cores. The substantial size of the problem being discussed justifies our focus on the distributed memory architecture. To our knowledge, there is no parallel implementation for solving the connected component labeling problem that scales to thousand of cores on distributed memory. Further, we demonstrate the practical significance of this algorithm by partitioning the grand challenge Iowa corn soil metagenome data set. The final output of our algorithm is the input reads segregated into species-level bins which can be assembled independently in parallel with any assembler of choice. To summarize the contributions of this paper:

1. We present a novel distributed connected component labeling algorithm for partitioning a metagenomic data set based on the de Bruijn graph connectivity.

2. We demonstrate the scalability of our algorithm by partitioning one of the largest metagenomic data sets with 1.8 billion reads, or approximately a graph with 135 billion vertices and edges, in 22 minutes.

3. We also discuss the general applicability of our algorithm using the Graph500 benchmark, and compare results against a state of the art parallel BFS implementation which can also be used to identify connected components.

2. RELATED WORK

2.1 Metagenomic assembly

Metagenomic assembly is an actively pursued research problem that is challenging due to unknown composition and number of genomes in a metagenomic sample, potential overlaps between constituent genomes, data complexity and its size. Recently, Nagarajan *et al.* [16] highlighted the need for memory efficient metagenomic assemblers due to the sheer sizes of the data sets. One approach to reduce memory usage is by processing subgraphs of a de Bruijn graph individually. Sequential metagenomic assemblers from Peng *et al.* [21] and Namiki *et al.* [17] partitoned the graph by pruning select edges and vertices to identify species-level subgraphs. Howe *et. al* [12] successfully assembled a large data set sequentially by first finding connected components of a de Bruijn graph using a memory-efficient probabilistic k-mer hash table, the khmer library [14]. However, the probabilistic hash table induces a significant false positive rate during the partitioning and assembly phase. Moreover, based on direct correspondence with the authors, we concluded that sequential connected components labeling for such large data set requires multiple days.

An orthogonal approach utilizes distributed memory systems to circumvent single machine memory limitations. Georganas *et al.* [9] presented impressive scalability results while parallelizing the Meraculous assembler, a single genome assembly tool. However, the applicability of Meraculous or its output quality on metagenomic data sets has not been established. Boisvert *et al.* [4] proposed a distributed metagenome assembler Ray Meta that can assemble a 3-billion-read data set using 1024 processors in 15 hours, 46 minutes.

Our solution incorporates both approaches, by finding connected components of a de Bruijn graph in a distributed environment. The connected components allow us to partition the read set into smaller subsets for assembly independently. Our approach allows users the flexibility to choose any assembler, including parallel assemblers, for the partitioned data set.

2.2 Connected Component Labeling

Connected component labeling for undirected graphs is a well studied problem with applications in many scientific domains. If m is the number of edges and n is the number of vertices in an undirected graph, this problem can be solved sequentially in $O(m + n)$ time via depth or breadth first search.

There have been numerous efforts to parallelize connected components labeling. Hirschberg *et al.* [11] presented a CREW PRAM algorithm that runs in $O(\log^2 n)$ time and does $O(n^2 \log n)$ work, while Shiloach and Vishkin [22] presented an improved version assuming a CRCW PRAM that runs in $O(\log n)$ time using $O(m+n)$ processors. There have been multiple efforts to implement and test the scalability of this algorithm on real-world architectures, but none of them have reported scalability beyond 40 cores. Chiang *et al.* [6] proposed multiple techniques to execute connected component PRAM algorithms using external disks in an I/O efficient manner. They generalized the impractical assumption of concurrent indirect memory accesses made by p processors into as many I/O operations as required for sorting p elements externally.

Bader *el al.* [1] and Patwary *et al.* [19] discussed shared memory multi-threaded parallel implementations for computing spanning forest and connected components on sparse and irregular graphs. Recently, Shun *et al.* [23] reported a work optimal implementation for the same programming model. Cong *et al.* [7] proposed a parallel technique for solving the connectivity problem on a single GPU.

There are also some recent parallel algorithms developed for computing the breadth-first search (BFS) traversal on distributed memory systems [5][25][2]. Even though BFS methods are optimized for traversing a single component for short diameter networks, they can still be utilized for computing connectivity using multiple seed vertices, one from each component. Slota *et al.* [24] presented a hybrid connectivity algorithm on shared memory machines that combines parallel BFS and Shiloach-Vishkin's algorithm. In their algorithm, BFS is used to initially label the largest component before Shiloach-Vishkin's algorithm is used to label the remaining components. Slota *et al.* achieved better performance with the hybrid approach than with BFS or the Shiloach-Vishkin algorithm alone. Recently, there have also been significant algorithmic advances for BFS on distributed memory systems, as reported by Buluc [5], Ueno [25], and Beamer [2].

Previous works on BFS have largely concentrated on and optimized for scale-free graphs, which tend to have a single large component that contains the majority of vertices of the graph. The Graph500 Kronecker generator [15] produces graphs in this category. Metagenomic de Bruijn graphs, however, have an extremely large number of connected components, the largest of which occupies only a small fraction of the total graph [12].

The Shiloach-Vishkin algorithm [22], by computing connectivity from all vertices simultaneously, is better suited for metagenomic de Bruijn graphs. Previous attempts to adapt this algorithm to distributed memory parallel computers [13, 10] did not achieve notable speedups. We propose a new algorithm for computing connectivity of metagenomic de Bruijn graphs on distributed memory systems. We briefly discuss the Shiloach-Vishkin algorithm below as our algorithm draws from some of the ideas behind it.

The Shiloach-Vishkin Algorithm

The Shiloach-Vishkin algorithm was designed assuming a PRAM model. It begins with singleton trees corresponding to each vertex in the graph and maintains this auxiliary structure of rooted directed trees to keep track of the connected components discovered so far during the execution. Within each iteration, there are two phases referred as *shortcutting* and *hooking*. *Shortcutting* involves collapsing the trees using pointer doubling. On the other hand, *hooking* connects two different connected components when they share an edge in the input graph. This algorithm requires $O(\log n)$ iterations each taking constant time. Since this approach uses $O(n+m)$ processors, the total work complexity is $O((m+n)\log n)$.

3. DISTRIBUTED MEMORY CONNECTED COMPONENTS LABELING

3.1 Definitions and Notation

Given an undirected graph $G = (V, E)$ with $n = |V|$ vertices and $m = |E|$ edges, our algorithm identifies its connected components, and labels each vertex $v \in V$ with its corresponding component.

Our algorithm works on an array of tuples $\langle p, q, u\rangle$, where p and q are integers in the range $\{1, ..., |V|\}$. The third element u denotes the corresponding vertex $u \in V$ of the graph. We initialize the array of tuples as follows: for each vertex $u \in V$ we add the tuple $\langle u, u, u\rangle$, and for each edge $\{u, v\} \in E$ we add tuples $\langle v, u, v\rangle$ and $\langle u, v, u\rangle$. The first two elements of the tuples will be updated in each iteration, whereas the last tuple element will never change during the algorithm. The last element is required only for identifying vertices to their final connected components. Since this vertex identifier is not required during the algorithm, we will subsequently simplify the notation and refer to tuples $\langle p, q, u\rangle$ also by $\langle p, q\rangle$.

Let $\mathcal{A}_i$ denote all the tuples in the global array $\mathcal{A}$ in iteration i. We denote the set of unique values of any of the first two entries of all tuples in $\mathcal{A}_i$ by $\mathcal{P}_i$, therefore $\mathcal{P}_i = \{q \mid \langle q, r\rangle \in \mathcal{A}_i \vee \langle r, q\rangle \in \mathcal{A}_i\}$. We refer to the unique values in $\mathcal{P}_i$ as *partitions*, which represent intermediate groupings of tuples that eventually coalesce into connected components. We say that a tuple $\langle p, q\rangle$ has the current partition p and a potential next partition q. Note that this definition of partition does not refer to vertices of the original graph.

We define the *bucket* $\mathcal{B}_i(p)$ of partition p as those tuples which contain p in their first entry: $\mathcal{B}_i(p) = \{\langle q, r\rangle \in \mathcal{A}_i \mid q = p\}$. Further, we define the *candidates* or the potential next partition $\mathcal{C}_i(p)$ of p as the values contained in the second tuple position of the bucket for p: $\mathcal{C}_i(p) = \{q \mid \langle p, q\rangle \in \mathcal{B}_i(p)\}$. We denote the minimum of the candidates of p as $p_{min} = \min_{q\in\mathcal{C}_i(p)} q$. A partition p for which $p_{min} = p$ is called a *root partition*.

We say that two partitions p and q are *connected* at any point in the execution, if either of the tuples $\langle p, q\rangle$ or $\langle q, p\rangle$ exists anywhere in the global array, i.e., if $p \in \mathcal{C}(q)$ or $q \in \mathcal{C}(p)$.

We define the *neighborhood* $\mathcal{N}_i(p)$ of a partition p as all those other partitions q which are connected to p, thus $\mathcal{N}_i(p) = \{q \mid q \in \mathcal{C}_i(p)\}$. We call the number of unique neighboring partitions of p as the size of p's neighborhood.

3.2 Algorithm

Prior to explaining the parallel algorithm, we first describe how and why the algorithm works, assuming that all operations are executed sequentially. We will then describe the parallelization strategies. We show an outline of the sequential algorithm in Algorithm 1.

The overall idea of the algorithm is similar to the *Shiloach-Vishkin* algorithm. Initially, every vertex is its own partition and partitions are connected via the edges of the graph. In each iteration, we join each partition to the unique neighboring partition having the minimum index among its neighborhood, until the partitions converge into the connected components of the graph. In order to resolve long chains and large diameter graphs quickly, we utilize a form of pointer doubling.

As laid out in Section 3.1, we first create an array of tuples, containing one tuple per vertex (self-loop) and two tuples per edge. In every iteration, we first sort the array of tuples $\mathcal{A}$ by the partition id, i.e., by the value of the first entry of each tuple. This enables easy processing of each bucket $\mathcal{B}_i(p)$, since the tuples of a bucket are positioned contiguously in $\mathcal{A}$

ALGORITHM 1: Connected Components Labeling

```
Input: undirected graph $G = (V, E)$
Output: Labeling of Connected Components
// generate tuples, two for each edge,
// and one for each vertex (self loop)
$\mathcal{A}$ = array of tuples $\langle p, q, v\rangle$
for $u \in V$ do $\mathcal{A}$.append($\langle u, u, u\rangle$);
for $\{u, v\} \in E$ do $\mathcal{A}$.append($\langle u, v, u\rangle$, $\langle v, u, v\rangle$);
$i \leftarrow 1$
// initially: $\mathcal{P}_1 = V$
while not terminated do
    sort($\mathcal{A}$ by first element)
    // check for termination
    if all partitions are roots then
        break
    end
    // for each bucket
    for $p \in \mathcal{P}_i$ do
        // determine minimum of candidates
        $p_{min} \leftarrow \min_{q \in \mathcal{C}_i(p)} q$
        if $|\mathcal{B}_i(p)| = 1$ then
            // if single element, update to minimum
            $\langle p, q\rangle \leftarrow \langle min(p, q), min(p, q)\rangle$
        else
            // for all tuples but the minimum ($q = p_{min}$)
            for each $\langle p, q\rangle \in \{\mathcal{B}_i(p) \setminus \langle p, p_{min}\rangle\}$ do
                // update $p$ to new partition $p_{min}$ and reverse
                // update tuples inside array $\mathcal{A}$ as:
                $\langle p, q\rangle \leftarrow \langle q, p_{min}\rangle$
            end
            // add update-request tuple as back reference to $p$
            $\mathcal{A}$.append($\langle p_{min}, p\rangle$)
        end
    end
    $i \leftarrow i + 1$
end
```

after sorting completes. For each bucket, we then determine the minimum candidate value p_{min} using a simple linear scan. Next, we *join* the elements of partition p into partition p_{min} by changing the first entry in each tuple from p to p_{min}. We then reverse the order of first two entries of each tuple in the bucket, such that every initial tuple $\langle p, q\rangle$ of the bucket becomes $\langle q, p_{min}\rangle$. Hence, after the sorting step of the next iteration the candidate partitions receive the updated partition values.

An updated partition p may be referred to in another partition q via $\langle q, p\rangle$ in iteration i. Those tuples which contain p as a second entry will also have to be updated to the new p_{min}. We therefore keep the information that p was updated to p_{min} around for one more iteration by excluding the minimum tuple $\langle p, p_{min}\rangle$ from the update step. This ensures that after the next sort, the bucket for p will contain p's new partition p_{min}. If no tuple referred to p in its second entry, then the next sort will yield only this single tuple for bucket p. In this case we can simply promote the single, left-behind tuple to p_{min}. This is handled as a special case when $|\mathcal{B}_i(p)| = 1$.

On the other hand, if p is part of a longer chain of partitions, we use a form of pointer doubling for quickly collapsing the chain. For this, we append another tuple $\langle p_{min}, p\rangle$ that serves as an update-request. In the next iteration, this tuple will be updated to $\langle p, r_{min}\rangle$, where r_{min} is the minimum neighbor of the partition $r = p_{min}$, and then, at the start of the third iteration, return this information to partition p via sorting. These updates will continue as long as other partitions still refer to p, since the update-request tuple is generated whenever the size of the bucket for p is larger than one.

3.3 Correctness

First, we show that the algorithm correctly identifies connected components, and then go on to show that it terminates properly.

To do so, we use the notion of *connected partitions* as defined in Section 3.1. Connected components can be defined as the transitive closure over the edges of the graph. We claim that the transitive closure over the *connected partitions* is equivalent to the connected components of the original graph at any iteration. In order to prove this, we show that *connected partitions* are initially identical to the graph structure, and that iterations preserve the connectivity.

Claim 1: The initial connectivity as defined by tuples is identical to the graph $G(V, E)$.

Proof: If there is an edge $\{u, v\} \in E$ in the graph, then tuples $\langle v, u, v\rangle$ and $\langle u, v, u\rangle$ are added to $\mathcal{A}$, thus u and v are initialized as *connected partitions*. Conversely, if there is no edge between vertices v and w in G, then their partitions are not connected.

Claim 2: If two partitions p and q are *connected* through a tuple $\langle p, q\rangle$ in any iteration of the algorithm, then they remain *connected* throughout all successive iterations, thus remaining in the same connected component.

Proof: Take the bucket for partition p at the given iteration when it contains, among others, the tuple $\langle p, q\rangle$. Further, let p_{min} be the minimum of the potential next partitions in the bucket for p. By definition, p is *connected* to both q and p_{min}. We distinguish between two cases:

1. Case $q \neq p_{min}$: The partition p is joined into partition p_{min}. In the update step of the bucket, the tuple $\langle p, q\rangle$ will be changed into $\langle q, p_{min}\rangle$. Thus the connection between p and q persists via p's new partition p_{min}.

2. Case $q = p_{min}$: In this case the partition p is joined into partition q. The connection remains further, since in this case tuples $\langle p, p_{min} = q\rangle$ and $\langle p_{min} = q, p\rangle$ will be created.

Thus the algorithm preserves connectivity of partitions with regards to their corresponding connected components. Note further, that new connections are introduced only due to the update step $\langle p, q\rangle \leftarrow \langle q, p_{min}\rangle$, which corresponds to the transitivity since p was connected to both q and p_{min}. Hence, two partitions from two different connected components can never be connected.

Claim 3: If after the sorting step of any iteration, all partitions are roots, then termination is reached and each partition corresponds to a connected component.

Proof: If all partitions are roots, then by the definition of a *root partition*, there are no more connections between any partitions. Due to claim 2, any connected partitions will remain connected throughout all iterations until they are joined into the same partition. If the tuples of two vertices $u', v' \in V$ end up in different *root* partitions, then they cannot be connected in the original graph G, hence they are in different connected components. Therefore each *root* partition must be a separate connected component. Conversely, if two vertices are in different connected components of G, they will end up in different final partitions since at no point are there tuples connecting the connected components.

Thus far, we have shown what a terminating state looks like. We next show that the termination criteria will be reached, i.e., that our algorithm will terminate. Termination is reached for any input graph G, since the number of unique partitions decreases in every step. Consider a partition p, which is the minimum among the partitions of its connected component. If this partition still has neighbors, then p will absorb its neighbor partitions, thus decreasing the number of partitions in each iteration. If p does not have any neighbors remaining, then it is a *root partition* and contains the whole connected component as explained above. Thus termination is reached for this partition.

3.4 Parallel Algorithm

We now describe our parallel implementation of the above algorithm for connected components labeling for metagenomic de Bruijn graphs in a distributed memory environment. Each processor in the environment has its own locally addressable memory space. Remote data is accessible only through well defined communication primitives over the interconnection network. The algorithm consists of three components: data distribution, parallel sort, and bucket update. We designed our algorithm and its components using MPI primitives.

Data Distribution: All data, including the input, intermediate results, and final output, are equally distributed across all available processors. The pipeline begins by generating tuples from the input sequence file. To this end, the file is evenly divided into blocks of size $\frac{N}{\rho}$, where N is the size of the file, and ρ is the number of processors. Parallel I/O is utilized to load each block into its corresponding process, from which we then locally generate the tuples of the form $\langle p, q, u\rangle$ as specified in section 3.1. By the end of this operation, each process contains its equal share of $|\mathcal{A}|/\rho$ tuples.

Parallel Sort: The main step of the algorithm is the sorting of tuples by their first element in order to form the buckets $\mathcal{B}(p)$. Parallel distributed memory sorting has been studied extensively. Blelloch *et al.* [3] give a good review of different methods. We implement a variant of samplesort with regular sampling, where each process first sorts its local array independently, and then picks equally spaced samples. The samples are then again sorted and $\rho-1$ of these samples are used as splitters for distributing data among processors. In a final step, the sorted sequences are merged locally.

Bucket Update: In this step, we need to determine the minimum p_{min} for each bucket $\mathcal{B}(p)$. As a result of the parallel sorting, all tuples $\langle p, q\rangle$ belonging to the same bucket are stored consecutively. However, any bucket might span multiple processors. Thus, we need to take special care of the first and last bucket of each process. All other, internal buckets are processed equivalent to the sequential case. Note that the first and last bucket on a process might be the same. Communicating the minimum of buckets with the previous and next process would require $O(\rho)$ communication steps in the worst case, since large $O(n)$ size partitions can span across $O(\rho)$ processes. We thus use two parallel prefix scan operations with custom operators to achieve independence from the size of partitions, requiring at most $O(\log\rho)$ communication steps in addition to the local linear time processing time.

We first perform an exclusive scan, where each process participates with the minimum tuple from its last bucket. This operation communicates the minimum of buckets from left-to-right. The reduction operator for two tuples is to choose the tuple $\langle p, q\rangle$ with the maximum p, and among those with equal p, the minimum q, i.e., the minimum q of the right- most bucket. In a second reverse exclusive prefix scan, we communicate the minimum from right-to-left. Here, each processor participates with its minimum tuple of its first bucket. The combination operation produces the overall minimum tuple, according to lexicographical ordering of $\langle p, q\rangle$. Given the two results of the scan operations, we can now determine the overall minimum p_{min} for both the first and the last bucket of each processor.

3.5 Excluding Completed Partitions

As the algorithm progresses through iterations, small partitions become completed. A partition is *completed* if it is a root partition which is not connected to any other partition. Since such *completed partitions* will not undergo any further changes during any further iterations, they can be excluded from future sorting and tuples updates.

The property of Claim 3 (see Section 3.3) can be used to detect the global termination. In order to detect whether a single partition is *completed*, we need to take more than the state of the current iteration into account. A given root partition p is completed, i.e., contains an entire connected component, if the size of its neighborhood is zero. This can be checked via a linear scan of bucket $\mathcal{B}(p)$. We also need to check that there are no further connections from other partitions to p. This requires a second iteration, as each tuple $\langle q, p\rangle$ that exists in the current iteration is updated to $\langle p, q_{min}\rangle$, and returned to bucket $\mathcal{B}(p)$ after the sort in the next iteration. We can thus detect termination for a single partition within a total of 2 iterations.

Completed partitions are marked as such and swapped to the end of the local array. All following iterations treat only the first, non-completed part of its local array as the local working set. As a result, the size of the active working set shrinks throughout successive iterations. This optimization yields significant performance gains for our application, since many small connected components are quickly excluded (see Section 4).

3.6 Load Balancing

Although we start with block decomposition of the vector $\mathcal{A}$ in the beginning of the algorithm, exclusion of *completed partitions* introduces an increasing imbalance of the active elements with each iteration. Since we join partitions from larger *ids* to smaller *ids*, a large partition will have smaller final partition ids than small partitions probabilistically. As the sort operation maps large id partitions to higher rank processes, the higher rank processes retain fewer and fewer active tuples over time, while lower rank processes contains growing partitions with small ids. Our experiments in Section 4 study this imbalance of data distribution and its effect on the overall run time. We resolve this problem and further optimize our algorithm by evenly redistributing the active tuples after each global sort. Our experiments show that this optimization yields significant improvement in the total run time.

3.7 Application to de Bruijn Graphs

For the metagenomic assembly problem, our goal is to assemble connected components of the de Bruijn graph in-

Data set	FASTQ Size (GB)	Reads (Million)	Tuples (Billion)	Memory Usage (GB)	Components (Million)	Sequence files
D1	6.2	37	1.71	27.39	29	649.4.815.fastq
D2	20	91	6.34	101.49	53	1424.1.1371.fastq
D3	44	203	14.27	228.35	100	1424.[3-4].1371.fastq
D4	114	531	37.18	594.87	184	1425.[2-6].1367.fastq
D5	210	1063	65.37	1045.98	290	1424.[1-7].1371.fastq 868.[1-5].1053.fastq
D6 (FULL)	421	1810	135.04	2160.65	393	All

Table 1: Sizes of test data sets in terms of file and read sizes, the number of tuples generated, the total memory usage across all processors for storing the tuples, and the count of connected components in the graphs.

dependently using existing assemblers. It is expedient and flexible to provide such assemblers with subsets of reads corresponding to the de Bruijn graph connected components, rather than modifying assemblers to ingest connected components directly from our algorithm. To produce read sets that correspond to de Bruijn graph connected components, we conceptually transform the directed de Bruijn graph to an undirected *read graph*, where each vertex represents a read, and each edge represents a common k-mer between two reads. We claim that this transformation preserves the read connectivity in the read graph compared to the k-mer connectivity in the de Bruijn graph. We note that a read forms a path in the de Bruijn graph, and therefore each read represents a path that is entirely contained in a de Bruijn graph connected component.

Claim 4: Two reads in the same read graph component are mapped to the same de Bruijn graph component, and *vice versa*.

Proof: We prove the forward and reverse cases separately.

1. *If two reads are in the same read graph component, their k-mers are in the same de Bruijn graph component*: There exists a path between the two reads in the read graph component. The k-mers from the reads on this path, including the terminal reads, are connected. Hence they are in the same de Bruijn graph component.

2. *If two reads are in different read graph components, the k-mers from each read are in separate de Bruijn graph components*: Assume the k-mers from these two reads belong to the same de Bruijn graph component, then there exist paths from the k-mers of one read to the k-mers of the second read in the de Bruijn graph component. Since a k-mer path consists of a sequence of substrings of reads, it can also be represented as a sequence of reads. The two reads then belong to the same read graph component due to the existence of a read path, thus contradicting the assumption.

The claim above establishes the equivalence between the connected components in a de Bruijn graph and the corresponding read graph. We can thus compute the disjoint read sets from the read graph, and convert them to de Bruijn graph connected components, which can be performed by the chosen assembler directly.

To construct the read graph, we assign each read a 32 bit unique partition id. For each k-mer in the read r_i, we insert a tuple with the form $\langle r_i, r_i, k_l \rangle$ into the array A. DNA consist of four nucleotides, hence each character requires only 2 bits of memory. Since our target application requires k-mers of size 31, we represent each k-mer by a single 64 bit integer. At this point, each read is in its own partition without any edges connecting the partitions.

As illustrated in Figure 1, two reads sharing the same k-mer belong to the same connected component. We now generate the edges between reads via common k-mers. More formally, if a k-mer k is associated with the tuples $\langle r_i, r_i, k \rangle$, $\langle r_j, r_j, k \rangle \ldots \langle r_k, r_k, k \rangle$, then all the reads $r_i, r_j \ldots r_k$ should belong to the same component. If r_{min} denotes $min\{r_i, r_j \ldots r_k\}$, we modify these tuples to $\langle r_i, r_{min} \rangle$, $\langle r_j, r_{min} \rangle \ldots \langle r_k, r_{min} \rangle$, while preserving the k-mer value k in the third position of the tuples. This step ensures that identical k-mers will start out in the same partition. Note that this step can be realized using a single distributed sort by k-mer and a linear scan. Once the edge relationships among the reads have been established, we can find the connected components and associated read sets using the algorithms described before. Therefore, after the algorithm terminates, each k-mer and corresponding reads will be associated with their containing connected component.

4. EXPERIMENTS AND RESULTS

We used the *CyEnce* cluster located at Iowa State University. Each node of the cluster has two 2GHz 8-Core Intel Xeon E5-2650 CPUs and 128 GB of main memory. The nodes are connected by a QDR (40 Gbps) Infiniband interconnect network. A Lustre parallel file system with 1 MDS and 8 OST servers provides high speed parallel IO for the cluster. We implemented our algorithm using C++ and MPI and run one MPI process per CPU core.

We used the Iowa Continuous Corn Soil Metagenome data set, from the DOE Joint Genome Institute Genome Portal (Project ID: 402461). This data set contains a mix of Illumina reads of lengths 76, 100 and 114. To test the scalability of the proposed algorithm, we created subsets of varying sizes from the complete data set by selectively grouping sequence files of biological samples together. From each read, k-mers of length 31 are generated. The k-mers and their current and next partition ids are stored as 3-tuples, requiring 16 bytes per tuple. The number of reads and the total count of tuples in each data set, as well as the total memory requirement for each, are summarized in Table 1.

The data sets are each stored as a single FASTQ file on the Lustre file system. The FASTQ file is partitioned into

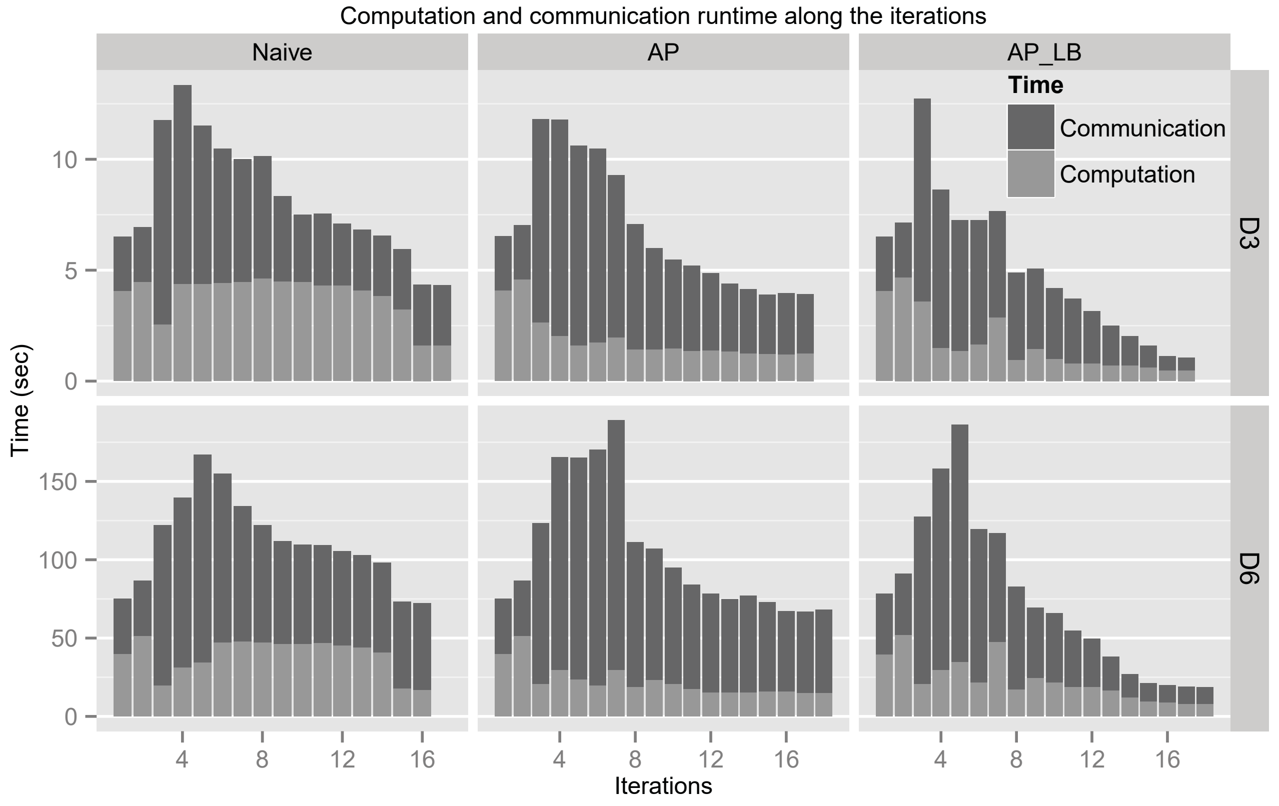

Figure 2: Time spent performing communication and computation in each iteration of the algorithm. Experiments were conducted using data sets D3 and D6 on 1280 cores. The AP algorithm operates on tuples in the active partitions only. The AP_LB algorithm balances work load between iterations of the AP algorithm.

equal-sized blocks, 1 per MPI process. The MPI processes concurrently read the blocks from the file system and generate k-mer tuples in a streaming fashion. The generated tuples are stored in a local vector for each MPI process. In our strong scaling experiments, we observed that the file reading and k-mer tuple construction time decreases nearly linearly with the number of MPI processes. For data set D3 with a FASTQ file of 46.6 GB, the loading time ranges from approximately 18 seconds for 128 processes down to 2.3 seconds for 1024 processes. Since file loading and k-mer tuple generation took only a small fraction of the overall run time, for subsequent discussions we report only the time involved in computing connectivity of the de Bruijn graph represented by the k-mer tuples.

We first examined the performance of the algorithm as it iteratively joins the partitions. These experiments were conducted using each of the three versions of the algorithm: Naive (Section 3.4), active partition only (AP, Section 3.5), and active partition with load balancing (AP_LB, Section 3.6). For each algorithm variant, we performed two sets of experiments with the D3 and D6 data sets using 1280 processes, and measure the time spent on communication and computation for each iteration. Figure 2 shows a stacked bar chart of the time spent on communication and computation for each algorithm variant and data set. The total run times for Naive, AP, and AP_LB for the D3 data set are 140, 117, and 87 seconds, respectively. Howe et al. [12] reported the runtime of 120 hours on the complete data set D6, although the processor specifications were not disclosed. In contrast, the total run times for our Naive, AP, and AP_LB algorithms for the D6 data set are 1840, 1880, and 1350 seconds, respectively. After the second iteration, communication occupies the majority of run time for all the following iterations. Compared to the Naive variant, the AP variant has smaller computation time due to fewer number of tuples involved in the iteration. Load balancing in the AP_LB variant further reduces the per iteration computation and communication times as the work is evenly distributed among the processes.

Processes	Naive	AP	AP_LB
128	771	595	356
256	437	368	216
512	254	220	142
1024	154	139	98
1280	140	117	87

Table 2: Run-time of the three algorithm variants for data set D3, measured in seconds

To confirm our interpretation of these results, we also examined the distribution of active tuples across the processes

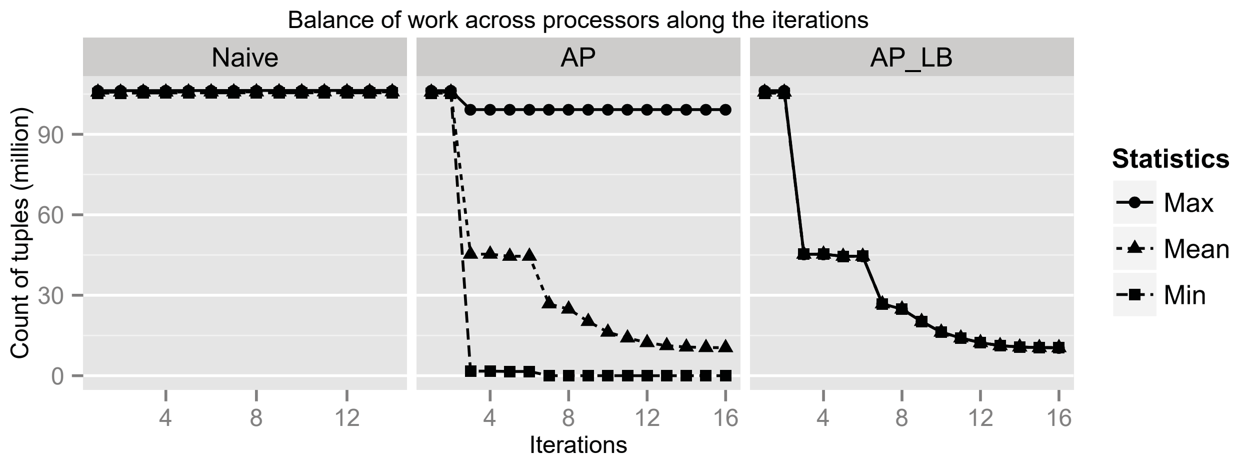

Figure 3: Tuple load balance across processes during each iteration of the three algorithm variants. The experiments were conducted using data set D2 and 60 processes. The graphs illustrate the maximum, average, and minimum count of tuples on all the processes.

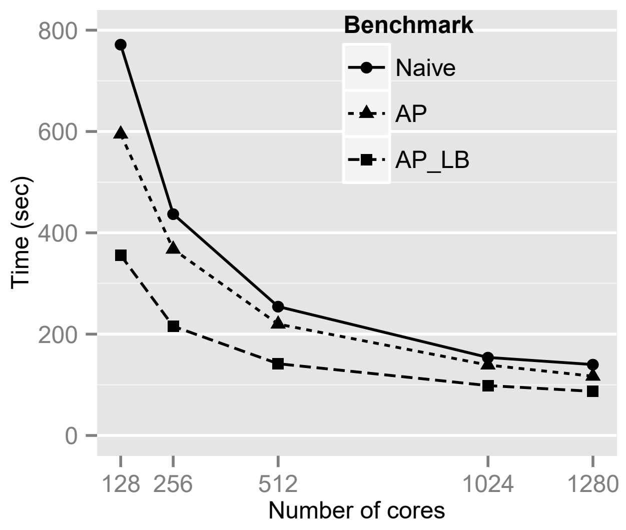

Figure 4: Performance of the three algorithm variants for data set D3, using increasing number of processor cores.

during program execution (Figure 3). We show this result for data set D2 partitioned using 60 cores. We measure the maximum (max), mean, and minimum (min) number of tuples across the 60 cores for each variant of the algorithms. Clearly, the max load is important as it determines the parallel run-time. A smaller separation between the min and max values indicate better load balance. As seen in Figure 3, the max, mean and min graphs are identical for the naive version of our algorithm. The AP variant reduces the size of the total working set with each iteration as illustrated in the decrease in mean value. However, the separation remained relatively constant after the first 3 iterations, indicating imbalance for the remaining large components. With load balancing in the AP_LB variant, we see an even distribution of tuples, as the minimum and maximum count are the same for each iteration. This explains the corresponding reduction in both computation and communication times of AP_LB variant in Figure 2 when compared to those of the AP variant.

We conducted strong scaling experiments using the D3 data set, as its memory requirement is sufficiently low for us to run the experiment using even 128 processes. Our algorithm requires approximately 3 times the size of the input vector in memory due to the use of distributed sample sort, which can cause the data imbalance by approximately 2 times, and MPI_Alltoall requiring separate input and output arrays. Computing using data set D3 on 128 processes, or 8 nodes in the CyEnce cluster, requires 28.5 GB of memory for persistent data, and approximately 57 GB of transient storage per 16-core node. Results of these experiments are shown in Figure 4 and Table 2. We observe that all three variants of our algorithm show scaling consistent with the earlier observation that communication is the dominant factor in our algorithm. The AP variant performs consistently better than the Naive variant through tuple pruning. The AP_LB variant performs consistently better than both Naive and AP variants, by a large margin, through load balancing of computation and MPI message sizes.

4.1 Comparison with Parallel BFS

Recent works have greatly improved the performance of parallel Breadth-First Search (BFS) methods on distributed memory machines. Typically, these methods perform BFS from a given seed vertex, and hence can be used to identify the connected component containing the seed vertex. In this section we compare and contrast our algorithm for connected components labeling with one based on parallel BFS.

For parallel BFS, we chose a recent state-of-the-art implementation by Buluc *et al.* [5]. To facilitate comparison, we adapted this software so as to achieve the same objec-

ALGORITHM 2: Connected Components Labeling With BFS

```
Input: undirected graph $G = (V, E)$
Output: Labeling of Connected Components
// Initialize unvisited vertex list $\mathcal{U}$ from vertices in $V$
$\mathcal{U} \leftarrow V$
// Find each connected component
while $|\mathcal{U}| > 0$ do
    choose a seed $s \in \mathcal{U}$
    // use BFS with the seed to find a connected component $C$
    $C \leftarrow BFS(s)$
    // remove vertices of the component $C$ from unvisited list
    $\mathcal{U} \leftarrow \mathcal{U} \setminus C$
end
```

tive as our algorithm, namely to compute all the connected components in a graph. To do so, parallel BFS is run iteratively, each time selecting a new seed vertex from among the vertices that were not visited during any of the prior BFS iterations (see Algorithm 2). Buluc's BFS implementation requires a distributed vertex array V and produces a corresponding distributed component C. We distribute our unvisited vertex list U using the same decomposition as that for V. The minimum unvisited vertex, by id, among all MPI processes is chosen as the seed vertex for the next BFS iteration. Our implementation of Algorithm 2 uses a hash set implementation (C++11 std::unordered_set) for the unvisited vertex container as it has $O(1)$ complexity for both single element deletion and access to the first element. Since each vertex in V is visited only once by BFS, the total work complexity for tracking the unvisited vertices is $O(|V|)$.

For this experiment, we used *Edison*, NERSC's Cray XC30 system, located at Lawrence Berkeley National Laboratory, due to limited availability of the CyEnce cluster. Each compute node in this system has two 12-core Intel Ivy Bridge processors and 64 GB main memory. The high-speed interconnect uses dragonfly topology and supports 8 GB/sec MPI bandwidth.

We used synthetic graphs for our comparative study. The graph was generated by the *Kronecker* generator using identical parameters as described in the *Graph 500* benchmark [15]. We set the scale value of the graph to 27 and the edge to vertex ratio to 16. As stated in Section 2.2, Kronecker graphs differ from metagenomic de Bruijn graphs and read graphs in that they are short diameter networks and contain a very large component that spans the majority of vertices of the graph. An instance of the generated Kronecker graph has 1 large component with approximately 63 million vertices, 59 components with 3 vertices, and 20814 components with 2 vertices or 1 vertex. In contrast, metagenomic de Bruijn graphs have multiple orders of magnitude more connected components. The smallest de Bruijn graph (D1) contains 29 million components with more diverse sizes.

We experimentally compared the runtime of the AP_LB variant of our algorithm against the BFS-based method. Our algorithm performs better than the BFS-based method by approximately a factor of 2 on varying number of processes from 256 to 4096. Figure 5 and Table 3 show the total runtimes of BFS-based method and our algorithm. Additionally, Table 3 shows the time spent on the BFS routine for the largest component, and the mean time spent for all other components.

Although the absolute time spent by parallel BFS per component is low, the large component count and the need to run parallel BFS for each component sequentially resulted in the cumulative time spent on small components dominating the runtime of the whole execution. This characteristics of the BFS-based method also prevented us from evaluating its performance on our metagenomic data sets due to time and resource limits. Since most of these small partitions are of size 2, choice of seed does not significantly affect the overall runtime for the BFS-based method. In contrast, our AP_LB algorithm simultaneously labels all the connected components. For Kronecker graphs, all the small components were computed within the first three iterations of the algorithm. The AP_LB algorithm terminated after labeling the largest component in the 7th iteration.

Overall these results show that our algorithm performs particularly well on graphs with a large number of connected components. On the other hand, the BFS method offers sig-

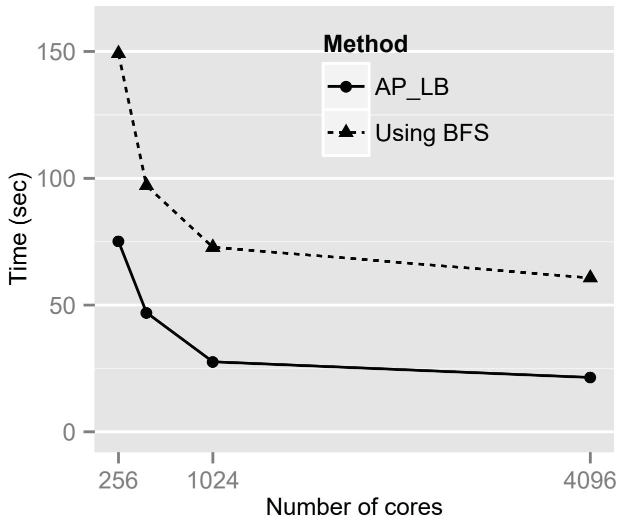

Figure 5: Performance comparison of AP_LB against the BFS-based method using Kronecker graph with scale=27.

Processes	256	484	1024	4096
BFS-based method				
Total time (sec)	149.2	97.1	72.8	60.7
Largest component time (sec)	0.65	0.44	0.29	0.15
Mean time for remaining components (sec)	6.48E-3	4.46E-3	3.31E-3	2.68E-3
AP_LB				
Total time (sec)	75.1	47.0	27.6	21.4

Table 3: Run-time of BFS-based method and AP_LB variant of our algorithm for computing connectivity on Kronecker graph with scale=27

nificantly superior performance when traversing the largest component. We anticipate that a hybrid approach utilizing BFS for relatively large components and our method for remaining small components would offer superior performance for computing connectivity in the Kronecker and many other real-world graph topologies on distributed memory systems.

5. CONCLUSIONS

This work constitutes the first scalable distributed-memory algorithm for computing de Bruijn graph connectivity, and it enabled fast labeling of very large-scale de Bruijn graphs. As a comparison, our largest dataset is derived from an influential work by domain scientists [12] in which about a week of computation time was spent in de Bruijn graph connected components labeling. The motivation for our contribution is to assist such scientific work through the development of novel parallel algorithms and high performance implementations. As an outcome, we are able to label the same de Bruijn graph in 22 minutes on 1280 Xeon cores. Besides the metagenomic de Bruijn graphs, we also compared the performance of our algorithm against parallel BFS for computing connected components on graphs used in the Graph500 benchmark.

Acknowledgments

This research was supported in part by the National Science Foundation under CCF-1360593 and IIS-1416259. The CyEnce cluster used for experiments was supported by the National Science Foundation under CNS-1229081.

6. REFERENCES

[1] D. A. Bader and G. Cong. A fast, parallel spanning tree algorithm for symmetric multiprocessors. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 38. IEEE, 2004.

[2] S. Beamer, A. Buluc, K. Asanovic, and D. Patterson. Distributed memory breadth-first search revisited: Enabling bottom-up search. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1618–1627. IEEE, 2013.

[3] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A comparison of sorting algorithms for the connection machine CM-2. In *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 3–16. ACM, 1991.

[4] S. Boisvert, F. Raymond, É. Godzaridis, F. Laviolette, J. Corbeil, et al. Ray Meta: Scalable de novo metagenome assembly and profiling. *Genome Biol*, 13(12):R122, 2012.

[5] A. Buluç and K. Madduri. Parallel breadth-first search on distributed memory systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 65. ACM, 2011.

[6] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 139–149. Society for Industrial and Applied Mathematics, 1995.

[7] G. Cong and P. Muzio. Fast parallel connected components algorithms on GPUs. In *Euro-Par 2014: Parallel Processing Workshops*, pages 153–164. Springer, 2014.

[8] J. Gans, M. Wolinsky, and J. Dunbar. Computational improvements reveal great bacterial diversity and high metal toxicity in soil. *Science*, 309(5739):1387–1390, 2005.

[9] E. Georganas, A. Buluç, J. Chapman, L. Oliker, D. Rokhsar, and K. Yelick. Parallel de Bruijn graph construction and traversal for de novo genome assembly. In *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*, pages 437–448. IEEE, 2014.

[10] S. Goddard, S. Kumar, and J. F. Prins. Connected components algorithms for mesh-connected parallel computers. *Parallel Algorithms: 3rd DIMACS Implementation Challenge*, 30:43–58, 1994.

[11] D. S. Hirschberg, A. K. Chandra, and D. V. Sarwate. Computing connected components on parallel computers. *Communications of the ACM*, 22(8):461–464, 1979.

[12] A. C. Howe, J. K. Jansson, S. A. Malfatti, S. G. Tringe, J. M. Tiedje, and C. T. Brown. Tackling soil diversity with the assembly of large, complex metagenomes. *Proceedings of the National Academy of Sciences*, 111(13):4904–4909, 2014.

[13] A. Krishnamurthy, S. S. Lumetta, D. E. Culler, and K. Yelick. Connected components on distributed memory machines. In *Parallel Algorithms: 3rd DIMACS Implementation Challenge October 17-19, 1994, volume 30 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 1–21. American Mathematical Society, 1994.

[14] E. McDonald and C. T. Brown. khmer: Working with big data in bioinformatics. *arXiv preprint arXiv:1303.2223*, 2013.

[15] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang. Introducing the Graph 500. *Cray User's Group (CUG)*, 2010.

[16] N. Nagarajan and M. Pop. Sequence assembly demystified. *Nature Reviews Genetics*, 14(3):157–167, 2013.

[17] T. Namiki, T. Hachiya, H. Tanaka, and Y. Sakakibara. MetaVelvet: An extension of Velvet assembler to de novo metagenome assembly from short sequence reads. *Nucleic Acids Research*, 40(20):e155–e155, 2012.

[18] H. Nordberg, M. Cantor, S. Dusheyko, S. Hua, A. Poliakov, I. Shabalov, T. Smirnova, I. V. Grigoriev, and I. Dubchak. The Genome Portal of the Department of Energy Joint Genome Institute: 2014 updates. *Nucleic Acids Research*, 42(D1):D26–D31, 2014.

[19] M. M. A. Patwary, P. Refsnes, and F. Manne. Multi-core spanning forest algorithms using the disjoint-set data structure. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 827–835. IEEE, 2012.

[20] J. Pell, A. Hintze, R. Canino-Koning, A. Howe, J. M.

Tiedje, and C. T. Brown. Scaling metagenome sequence assembly with probabilistic de Bruijn graphs. *Proceedings of the National Academy of Sciences*, 109(33):13272–13277, 2012.

[21] Y. Peng, H. C. Leung, S.-M. Yiu, and F. Y. Chin. Meta-IDBA: A de novo assembler for metagenomic data. *Bioinformatics*, 27(13):i94–i101, 2011.

[22] Y. Shiloach and U. Vishkin. An O(logn) parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57–67, 1982.

[23] J. Shun, L. Dhulipala, and G. Blelloch. A simple and practical linear-work parallel algorithm for connectivity. In *Proceedings of the 26th ACM symposium on Parallelism in Algorithms and Architectures*, pages 143–153. ACM, 2014.

[24] G. M. Slota, S. Rajamanickam, and K. Madduri. BFS and coloring-based parallel algorithms for strongly connected components and related problems. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 550–559. IEEE, 2014.

[25] K. Ueno and T. Suzumura. Highly scalable graph search for the Graph500 benchmark. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, pages 149–160. ACM, 2012.

[26] D. R. Zerbino and E. Birney. Velvet: Algorithms for de novo short read assembly using de Bruijn graphs. *Genome Research*, 18(5):821–829, 2008.