Title

HipMer

Permalink

https://escholarship.org/uc/item/5js586fb

ISBN

978-1-4503-3723-6

Authors

Georganas, Evangelos

Buluç, Aydın

Chapman, Jarrod

et al.

Publication Date

2015-11-15

DOI

10.1145/2807591.2807664

Peer reviewed

HipMer: An Extreme-Scale
De Novo Genome Assembler

Evangelos Georganas[†,‡], Aydın Buluç[†], Jarrod Chapman[*], Steven Hofmeyr[†],
Chaitanya Aluru[‡], Rob Egan[*], Leonid Oliker[†], Daniel Rokhsar[*,¶], Katherine Yelick[†,‡]

[†]*Computational Research Division* / [*]*Joint Genome Institute, Lawrence Berkeley National Laboratory, USA*
[‡]*EECS Department* / [¶]*Molecular and Cell Biology Department, University of California, Berkeley, USA*

ABSTRACT

De novo whole genome assembly reconstructs genomic sequences from short, overlapping, and potentially erroneous DNA segments and is one of the most important computations in modern genomics. This work presents HipMer, the first high-quality end-to-end *de novo* assembler designed for extreme scale analysis, via efficient parallelization of the Meraculous code. First, we significantly improve scalability of parallel k-mer analysis for complex repetitive genomes that exhibit skewed frequency distributions. Next, we optimize the traversal of the de Bruijn graph of k-mers by employing a novel communication-avoiding parallel algorithm in a variety of use-case scenarios. Finally, we parallelize the Meraculous scaffolding modules by leveraging the one-sided communication capabilities of the Unified Parallel C while effectively mitigating load imbalance. Large-scale results on a Cray XC30 using grand-challenge genomes demonstrate efficient performance and scalability on thousands of cores. Overall, our pipeline accelerates Meraculous performance by orders of magnitude, enabling the complete assembly of the human genome in just 8.4 minutes on 15K cores of the Cray XC30, and creating unprecedented capability for extreme-scale genomic analysis.

1. INTRODUCTION

DNA sequencing and assembly are the first necessary steps towards understanding and modifying the behavior of genomes, with promises of transforming and personalizing medicine. Recent advances in next-generation sequencing are exponentially improving the rate and reducing the cost of genome sequencing, creating the potential to assemble an unprecedented number of animal, fungi, plant and environmental genomes. However, state-of-the-art short-read shotgun sequencing technologies make the problem of *de novo* assembly one of the most demanding computational bioinformatics challenges, particularly for human and even larger gigabase-scale genomes, which can now be sequenced at modest cost. *De novo* assemblers reconstruct an un-

known genome from a collection of short reads, and have inherent advantages of discovering variations that may remain undetected when aligning sequencing data to a reference genome; unfortunately *de novo* assembly computational runtimes cannot keep up with the data generation of modern sequencers.

In this work we present HipMer, the first end-to-end high-performance parallelization of the Meraculous *de novo* assembler [1], which has been ranked among the first places in most metrics of the Assemblathon II competition [2]. Meraculous leverages base quality scores from sequencing instruments to identify non-erroneous overlapping substrings of length k (k-mers) with high quality extensions. Meraculous uniquely assembles genome regions into uncontested sequences called *contigs* by constructing and traversing a de Bruijn graph of k-mers, a special type of graph that is used to represent overlaps among k-mers. Contigs are then linked together to create *scaffolds*, sequences of contigs that may contain gaps among them. Finally gaps are filled using localized assemblies based on the original reads.

In our previous work [3] we studied parallel high-performance algorithms for k-mer analysis and contig generation, which form the initial stages of the Meraculous assembly pipeline. The full pipeline that we have now implemented is described in Section 2. We have applied several key optimizations (Section 3) to our previous algorithms. First, we significantly improve scalability of parallel k-mer analysis for complex repetitive genomes that exhibit skewed frequency distributions. Results on the grand-challenge, highly repetitive wheat genome show a performance improvement of up to 2.4× compared with the previous version. Second, we optimize the traversal of the de Bruijn graph of k-mers by employing a novel communication-avoiding parallel algorithm in a variety of practical use-case scenarios. Experimental results demonstrate a speedup of up to 2.8× when processing multiple genomes of the same species by leveraging our algorithmic optimization. Additionally, we increase I/O performance by efficient parallel reading of the input FASTQ files.

Section 4 then describes our efficient implementation and parallelization of scaffolding, enabling the first massively scalable, high quality, complete end-to-end *de novo* assembly pipeline. Experimental large-scale results on the NERSC Edison Cray XC30 using human and the wheat genomes, as well as the massive-scale metagenome of wetland soil samples are presented in Section 5 — demonstrating efficient performance and scalability on thousands of cores. Compared with the original Meraculous code, which has limited
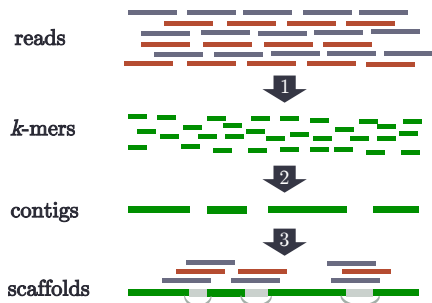
Figure 1: Meraculous assembly flow chart.

scalability, and requires approximately 23.8 *hours* to assemble the human genome, HipMer employs an efficient Unified Parallel C (UPC) implementation, computing the assembly in only 8.4 *minutes* using 15,360 cores of Edison — an overall speedup of approximately 170×. Overall results show that HipMer holds the potential to outperform today's biomedical sequencing capacity via high-concurrency supercomputing assembly, and significantly accelerate the dissemination of computed genomes to the bioinformatics community.

2. MERACULOUS ASSEMBLY PIPELINE

This section gives a high level overview of Meraculous assembler pipeline, which consists of three major stages (see Figure 1):

1) **K-mer analysis:** The input reads are processed to implicitly exclude errors. First, the reads are chopped into *k-mers*, which are overlapping sequences of length k. Then, the k-mers are counted and those that appear fewer times than a threshold are treated as erroneous and discarded. Additionally, for each k-mer we keep track of the two neighboring bases in the original read it was extracted from (henceforth we call these bases *extensions*). The result of k-mer analysis is a set of k-mers and their corresponding extensions that with high probability include no errors.

2) **Contig generation:** The resulting k-mers from the previous step are stored in a de Bruijn graph. The de Bruijn graph is a special type of graph that represents overlaps in sequences. In this context, k-mers are the vertices in the graph, and two k-mers that overlap in $k - 1$ bases whose corresponding extensions are compatible are connected with an undirected edge in the graph (see Figure 2 for a de Bruijn graph example with $k = 3$).

Due to the nature of DNA, the de Bruijn graph is extremely sparse (e.g. the human genome's adjacency matrix that represents the de Bruijn graph is a $3 \cdot 10^9 \times 3 \cdot 10^9$ matrix with between two and eight non-zeros per row for each of the possible extensions). For k-mers where the extensions are defined to be unique in both directions, only rows with two non-zero entries need to be considered. Thus a compact representation can be leveraged via a hash table: A vertex (k-mer) is a key in the hash table and the incident vertices are stored implicitly as a two-letter code [ACGT][ACGT] that indicates the unique bases that immediately precede and follow the k-mer in the read dataset. By combining the key and the two-letter code, the neighboring vertices in the graph can be identified. Via construction and traversal of the underlying de Bruijn graph of k-mers the connected components in the graph are identified, which are linear chains

of k-mers, called *contigs*. Contigs are (with high probability) error-free sequences that are generally longer than the original reads.

3) **Scaffolding and gap closing:** The first step of scaffolding is to map the original reads onto the generated contigs. This mapping provides information about the relative ordering and orientations of the contigs. Once the orientations are determined, it is possible that there are gaps between pairs of contigs. We then further assess the read-to contig mappings and locate the reads that are placed into these gaps. Ultimately, we leverage this information and close the contig gaps. The final result of the assembly is a completed set of *scaffolds*.

3. NOVEL OPTIMIZATIONS

We now describe several novel optimizations developed to improve the behavior of previously studied components [3]: high-frequency k-mer analysis, communication-avoiding de Bruijn graph traversal, and FASTQ read I/O performance.

3.1 Efficient High Frequency k-mers Analysis

During the k-mer generation phase, our previous work [3] relied on Bloom filters [4] to eliminate erroneous k-mers without inserting them into the main data structures, resulting in memory requirement reductions of up to 85% in human and wheat genomes. In that approach, we assigned each k-mer to a particular 'owner' processor, which counts all the occurrences of a given k-mer. In other words, processors did not perform any partial counting. This choice was largely motivated by the need to use Bloom filters for eliminating erroneous k-mers, which requires that all counting for a k-mer is performed by its owner in order to safely call it erroneous,.

For a simple analysis, assume that there are n k-mers and a fraction $0 \leq f \leq 1$ of them occur only once (hence guaranteed to be an error or unreliable at best). Blindly performing partial counting in each processor for all k-mers encountered would require $O(np)$ aggregate space over p processors. Using the owner computes model and Bloom filters reduces this overhead to $O(fn)$. The downside of this approach is that highly complex plant genomes, such as wheat, are extremely repetitive and it is not uncommon to see k-mers that occur millions of times. For example, the (hexaploid) bread wheat lines 'Synthetic W7984' assembled in previous work [5] has about 2,000 k-mers that each occur more than half a million times and about 70 k-mers that occur over 10 million times for k=51. Such high-frequency k-mers create a significant load imbalance problem, as the processors assigned to these high-frequency k-mers require significantly more memory and processing times.

Consequently, we improve our approach by first identifying frequent k-mers (i.e."heavy hitters" in database literature) and treating them specially. In particular, the owner compute method is still used for low-to-medium frequency k-mers but the high frequency k-mers are accumulated locally, followed by a final global reduction. Since an initial pass over the data is already performed to estimate the cardinality (the number of distinct k-mers) and efficiently initialize our Bloom filters, running a streaming algorithm for identifying frequent k-mers during the same pass is essentially free in terms of I/O costs.

We use the counter-based algorithm of Misra and Gries [6] (subsequently reinvented several times [7, 8]). If the true fre-

quency of a k-mer x is $f(x)$, the Misra-Gries algorithm outputs all k-mers with $f(x) \geq 1/\theta$ using $O(\theta)$ space. Furthermore, the reported count $f'(x)$ is a lower bound on the actual count, i.e. $f'(x) \leq f(x)$. Since the items with $f'(x) > 1$ can not be eliminated by the Bloom filter, integration of the Misra-Gries algorithm to the k-mer analysis step does not decrease the efficiency of Bloom filters as long as we only treat k-mers with $f'(x) > 1$ specially.

For parallelization, we take advantage of the fact that the Misra-Gries algorithm creates mergeable summaries [9] and use a high-performance implementation of the algorithm described by Cafaro and Tempesta [10].

3.2 Communication-Avoiding de Bruijn Graph Traversal

We now describe the key components of the de Bruijn graph traversal methodology and our innovative communication-avoiding algorithm for accelerating performance; a more in-depth discussion of de Bruijn graph traversal is described in our previous work [3].

The first step towards efficient parallelization is to store the de Bruijn graph in a distributed hash table. First, a given processor p_i chooses a random k-mer as seed and creates a new subcontig data structure that is represented as a string, where the initial content of the string is the seed k-mer. Processor p_i then attempts to extend the subcontig towards both of its endpoints by using the edges of that k-mer node. In order to explore a neighboring vertex, p_i performs a lookup in the distributed hash table. Processor p_i can also extend the subcontig towards both of its endpoints until the contig construction is completed (or equivalently until the corresonding connected component has been explored). If two processors work on the same connected component (i.e. both selected seed k-mers from the same contig), race conditions are avoided via a lightweight synchronization scheme [3]. Finally, the parallel traversal is terminated when all the connected components in the de Bruijn graph are explored.

Although our de Bruijn graph traversal demonstrates high scalability [3], it inherently suffers from high communication overhead. Because we focus on *de novo* genome assembly (i.e. there is no reference genome), the initial contig construction lacks implicit locality, and it is the goal of the de Bruijn algorithm to explore connectivity and to find the corresponding connected components. Additionally, due to the nature of DNA, the de Bruijn graph is extremely sparse, as previously discussed. Thus, to explore an additional vertex in the graph and expand a subcontig by one base is it necessary to perform a lookup in the distributed hash table, which with high probability will incur a communication event.

Therefore, given a genome of size G, the asymptotic communication cost of the parallel traversal is $O(G)$ messages, or $O(G/p)$ messages if we measure the number of messages along the critical path. Since the messages have size $O(1)$ (each message is just a single k-mer), the asymptotic bandwidth costs are the same. Note that our parallel algorithm is load balanced and communication along the critical path is decreased as the number of processors is increased, thus allowing our traversal algorithm to strong scale efficiently [3].

There are two key insights that underly the development of our communication-avoiding algorithm for the parallel de Bruijn graph traversal:
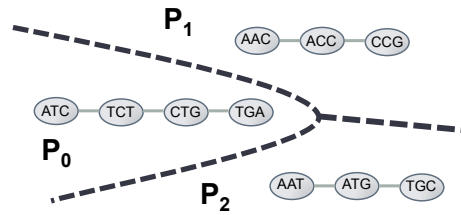

Figure 2: A de Bruin graph of k-mers where $k = 3$. In this graph we can identify three contigs or equivalently three connected components. The dashed lines indicate an optimal partitioning of the graph with $p = 3$. Each processor will work only with local k-mers during the traversal.

Oracle Partitioning.

If given an *oracle partitioning function* that could accurately predict how the k-mers are placed into contigs, the k-mers could be partitioned in such a way that k-mers ultimately belonging to the same contig would be mapped to the same processor. Thus, during the traversal a processor would incur *no* communication, since none of the k-mer accesses (lookups in the distributed hash table) will require communication — all the vertices of the de Bruijn graph that build up a particular contig are local to a single processor (local buckets in the distributed hash table). This idea is similar to graph partitioning, which aims to minimize the number of edges between separated components. An example is given in Figure 2, which shows three processors and three equally-sized sub-graphs that do not share any edges. By assigning the corresponding k-mers to the appropriate processor, all three processors will conduct local lookups in the hash table during the parallel traversal.

Genetic Similarity.

The nucleotide diversity is similiar for given organisms. For example it is estimated that humans differ in only 0.1%–0.4% of DNA base pairs, meaning that the contigs of different humans have a high degree of similarity. This implies that an oracle partitioning derived from one genome will work for others of the same species.

Communication-avoiding parallel algorithm

These insights have enabled us to develop an off-line *oracle partitioning function*. Once the contigs are computed for a given organism, we can apply this oracle partitioning function to *any other* de Bruijn graph (for another member of the same species), to dramatically decrease the communication incurred during the traversal computation. This approach can also be leveraged when searching for the optimal k length for *de novo* assembly. Typically, computational biologists begin the genome assembly process for a given organism with a draft version generated using a reasonable initial k value. Different k lengths are then explored to optimize the quality of the assembly output. Thus we can generate our *oracle partitioning* function during the initial contig generation phase, and use it to significantly reduce communication for subsequent assemblies that explore different k values. Even though the contigs are expected to change for varying k lengths, the new set of contigs will have a high degree of similarity with the first draft assembly, resulting

in significant performance improvements.

The (off-line) algorithm for generating the oracle partitioning function, `oracle_hash()`, given a set of contigs C and a uniform hash function, `uniform_hash()`, consists of two steps:

(1) Iterate over the set of contigs C and assign each contig a processor ID (in a cyclic fashion to ensure load balance) among processors.

(2) Iterate within each contig and extract its k-mers. To each one of these k-mers assign the contig's processor ID and store this information in a compact `oracle_hash_vector`. In particular, given a k-mer A, we store the corresponding processor ID in the position `uniform_hash(A)` of `oracle_hash_vector`. If there is a collision (i.e. another k-mer of another contig has been already stored in this position of the vector), then k-mer A will be stored to a remote processor instead of the correct (local) one. The number of collisions in this `oracle_hash_vector` is approximately the number of communication events that will be incurred during the traversal. Therefore, with a larger `oracle_hash_vector` we can decrease the number of collisions and consequently we can decrease the communication volume. Essentially we can trade off memory requirements and the number of collisions in the `oracle_hash_vector`, thus increasing or decreasing communication overhead according to the size of the available aggregate memory. Note that the algorithm for generating the `oracle_hash_vector` can be trivially parallelized. Nevertheless, since the construction of the `oracle_hash_vector` is an offline process and has to be completed only once, it does not lie on the critical path of the pipeline's execution.

The `oracle_hash()` values are computed during the graph construction and traversal for a given `oracle_hash_vector` as follows:

(1) The processors load the `oracle_hash_vector` in their memory, and the `oracle_hash_vector` is replicated across processors. Alternatively we can replicate the `oracle_hash_vector` on a node basis in order to decrease the per thread memory requirements.

(2) When calling `oracle_hash(A)` at the application level in respect to a k-mer A, the value of `uniform_hash(A)` is computed. Next we lookup in the processor-local `oracle_hash_vector` and retrieve the corresponding processor ID (p_i). Based on the value of p_i, we subsequently reshuffle the original `uniform_hash(A)` value. In particular, if `uniform_hash(A)` was about to be mapped in location (bucket) b of processor p_j (assuming a cyclic distribution of buckets to processors), the return value of `oracle_hash(A)` is adjusted such that it is mapped at location (bucket) b of processor p_i. Observe that uniformity in the distribution of buckets is preserved since all the hash values are shifted in a uniform way to the appropriate buckets. Moreover, the vast majority of the contig's k-mers that will be looked up after k-mer A during the traversal phase, are located in the same processor p_i by construction of the `oracle_hash_vector`. Therefore, if the processors select traversal seeds from local buckets, they will be mostly performing local accesses in the hash table when traversing the de Bruijn graph.

A refinement for practical considerations (e.g. SMP clusters), is working with node IDs instead of processor IDs. In this setting, the algorithm avoids the off-node communication while performing intra-node accesses when generating the contigs, which are orders of magnitude cheaper than the off-node communication overhead.

3.3 Parallel Reading of FASTQ Files

Our previous work relied on the SeqDB [11] binary format, which utilizes the modern HDF5 file format, for fast parallel I/O. Even though SeqDB is a scalable compressed storage medium for sequence data, this required conversion from the FASTQ format prior to the execution, and could thus become an impediment for end users who do not want to change their file formats. The challenge was that neither of the parallel genome assemblers we tested (Ray and Abyss) had a scalable fast FASTQ reader. In this work, we overcome this barrier by implementing a parallel block FASTQ reader.

The reader first samples the FASTQ file in parallel (each processor samples about a million reads concurrently) to estimate the id lengths, which can be variable across reads. The average id length is then used as an estimate to find splitting points across processors. Since a splitting point can be in the middle of a read, a processor p_i fast forwards to the beginning of the next full read, while the previous partial read is processed by the neighboring processing p_{i-1}. The key to high performance is to use MPI-IO functions (in our case `MPI_File_read_at`) with large buffer sizes and parse the buffered data in memory. Using this approach, our method obtains close to the I/O bandwidth achieved by reading SeqDB (up to compression factor differences).

4. SCAFFOLDING PARALLELIZATION

In this section we describe the newly developed parallelizations of the scaffolding's modules.

4.1 Contig Depths and Termination States

Given the exact count (depth) of the k-mers and a set of contigs, we first calculate the average depth of each contig. The parallelization here consists of two steps. First, the k-mers are stored in a distributed hash table where the keys are k-mers and the values are the corresponding counts. For the construction of the distributed hash table we employ a communication optimization, namely "aggregating stores" [3]. This optimization aggregates fine-grained messages and decreases their total number along the critical path, while additionally reducing the synchronization costs in the distributed hash table construction. Next, each processor p_i is assigned $1/p$ of the contigs (p is the total number of available processors) and for every contig, p_i looks up all the contained k-mers and sums up their counts. The average depth of the contig is then calculated as the mean depth of all the k-mers. Note that this step does not require any synchronization, as the distributed hash table is only read after its construction.

In the distributed hash table of k-mers we store their corresponding extensions as their values. The algorithm then iterates in parallel over the contig set and identifies the termination condition for every contig. For example, a contig may have been terminated because it did not find any neighboring k-mers to its endpoints, or if it found a branch in the graph with two high quality neighboring k-mers. The latter is the case that often arises in diploid organisms and in repetitive genome regions. The Meraculous assembler terminates contigs if they do not have unique high quality neighboring k-mers, but this termination state is utilized in the following bubble identification step during scaffolding.
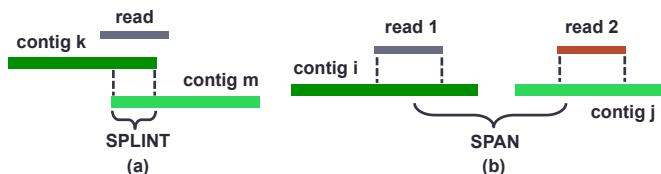
4.2 Identifying Contig Set Bubbles

Figure 3: (a) Contigs k and m constitute a splint. (b) Contigs i and j constitute a span.

In this step, we examine the termination condition of contigs and we detect *bubbles*: contigs that have the same k-mers as extensions of their endpoints (note that this is only applicable in diploid genomes). By leveraging this bubble information and the contigs's end termination state, a graph of contigs is created called the *bubble-contig* graph. This graph is orders of magnitude smaller than the original k-mer de Bruijn graph because the connected components (contigs) of the de Bruijn graph have been contracted to supervertices. We build this bubble-contig graph in parallel by employing a distributed hash table.

Once the bubble-contig graph is built, it is traversed to merge qualifying contigs (e.g. by following one of the paths in the bubbles). We parallelize this traversal using a speculative approach. The processors pick random seeds from the bubble-contig graph and initiate independent traversal. Once an independent traversal is terminated we store the resulting path. However, if multiple processors work on the same path, they abort their traversals and allow a single processor to complete them. In practice, this speculative execution spends most of the time ($\sim 99\%$) in parallel traversals. Given that the bubble-contig graph is orders of magnitude smaller than the original de Bruijn graph, its traversal is extremely fast. The result of this module is a set of contig paths, where every path can be compressed to a single DNA sequence. For simplicity, we call these compressed paths contigs in the description of the following modules.

4.3 Alignment of Reads onto Contigs

In this pipeline phase the goal is to map the original reads onto the generated contigs. This mapping provides information about the relative ordering and orientation of the contigs. In order to efficiently parallelize the read-to-contig alignment, we employ merAligner [12], a scalable, end-to-end parallel sequence aligner. MerAligner implements a seed-and-extend algorithm and fully parallelizes all of its components. Therefore it significantly outperforms other parallel alignment tools that mostly build their lookup tables serially [12].

4.4 Insert Size Estimation of Read Libraries

Given a set of paired-end reads (i.e. the reads come in pairs) and the corresponding read-to-contig alignments from the previous step, we now use full length alignments in which both ends of a pair are placed within a common contig, and calculate the insert size. Sampling the alignments allows us to estimate the insert size of the library, which is used as a parameter in subsequent scaffolding modules. The insert size estimation is parallelized by having p processors build local histograms of distinct sampled alignments and eventually merging these p local histograms to a global one, from which the insert size of the read library is computed.

4.5 Locating Splints and Spans

The next step is to process the alignments and identify *splints*, which are contigs that overlap at their ends. Essentially, if a particular segment of a read aligns to the ends of two different contigs we conclude that these contigs form a splint (see Figure 3 (a)). The parallelization of this step is straightforward: Each of the p processors independently processes $1/p$ of the total read alignments and stores the splints's information.

Additionally, by processing paired-end reads' alignments we identify *spans*, which are pairs of contigs associated with particular "mate" reads via their read-to-contig alignments. For example, consider that the first read of a pair aligns with contig i while the second read of that pair aligns with contig j. It can thus be concluded that contigs i and j form a span (see Figure 3 (b)). Note that we have already calculated the insert size of the read library in previous step and therefore can estimate the gap size among contigs i and j. Again the parallelization of this module is straightforward, where each processor independently assesses $1/p$ of the total read alignments and stores the spans's information.

4.6 Contig Link Generation

Once splints and spans are created, they can be assessed to generate *links* among pairs of contigs. More specifically, if a sufficient number of read alignments supports a particular splint between contig k and contig m, we generate a *splint link* for that pair of contigs. Parallelizing this operation requires a distributed hash table, where the keys are pairs of contigs and values are the splint/overlap information. Each processor is assigned $1/p$ of the splints and stores them in the distributed hash table. Here, we again apply the aggregating stores optimization to minimize the number of messages and the synchronization cost. When all splints have been stored in the distributed hash table, each processor iterates over its local buckets to further assess/count the splint links.

In an analogous way, if a sufficient number of paired-read alignments supports a particular span between contig i and contig j we generate a *span link* for that pair of contigs. The parallelization of this operation relies on a distributed hash table where the keys are pairs of contigs and values are the span/gap information. Again we have each processor assess $1/p$ of the spans and apply the aggregating stores optimization. After the distributed hash table construction is finalized, each processor iterates over its local entries to further assess/count the span links.

4.7 Ordering and Orientation of Contigs

After storing the previously generated links in a distributed hash table where the keys are pairs of contigs and values are the corresponding link information, the data in the links is processed to build *ties* among the contigs by consolidating splint/span links. This step employs parallelism by having each processor work on the local buckets of the links' distributed hash table. After the establishment of ties among the contigs, Meraculous' scaffolder traverses the implicit graph of ties and then locks contigs together in order to form *scaffolds* (see Figure 4). The traversal is done by selecting seeds in order of decreasing length (this heuristic tries to lock together first "long" contigs) and therefore it is inherently serial. We have optimized this component and found that

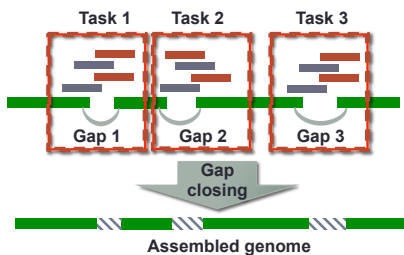Figure 4: A scaffold formed by traversing a sequence of ties.

Figure 5: A set of gaps in a scaffold to be closed.

Figure 6: Strong scaling of k-mer analysis on wheat, showing the effect of the heavy hitters (high frequency k-mers) optimization.

its execution time is insignificant compared to the previous pipeline operations. This behavior is expected since the number of contigs (vertices in the ties's graph) is orders of magnitude smaller than the k-mers (vertices in the de Bruijn graph of the pipeline).

4.8 Gap Closing

The gap closing stage uses the merAligner outputs (i.e. read-to-contig alignments), the scaffolds and the contigs to attempt to assemble reads across gaps between the contigs of scaffolds (see Figure 5). To determine which reads map to which gaps, the alignments are processed in parallel and projected into the gaps. The gaps are divided into subsets and each set is processed by a separate thread, in an embarrassingly parallel phase. Several methods are available for constructing closures, and are used in succession until a closure is found or no more methods are available. The first method used is *spanning*, i.e. finding a read that begins with the end of the contig on one side of the gap, and finishes with the start of the contig on the other. Should spanning fail, an attempt is made to do a traversal (k-mer walk) across the reads from one side of the gap to the other, with iteratively increasing k-mer sizes until the gap is closed. This mini-assembly step is first attempted from the right hand side contig to the left, and if that fails, from left to right. If both traversals fail, the final method used is an attempt to *patch* across the two incomplete traversals, i.e. find an acceptable overlap between the two sequences.

The various closure methods differ in computational intensity, with spanning and patching being orders of magnitude quicker than k-mer walks. Given that it is not clear *a priori* what methods will be successful for closing a gap, the computational time can vary by orders of magnitude from one closure to the next. To prevent load imbalance in the gap closing phase, the gaps are distributed in a Round Robin fashion across all the available threads. This suffices to prevent most imbalance because it breaks up the gaps from a single scaffold, which tend to require similar costs to close.

5. EXPERIMENTAL RESULTS

Parallel performance experiments are conducted on Edison, the Cray XC30 located at NERSC. Edison has a peak performance of 2.57 petaflops/sec, with 5,576 compute nodes, each equipped with 64 GB RAM and two 12-core 2.4GHz
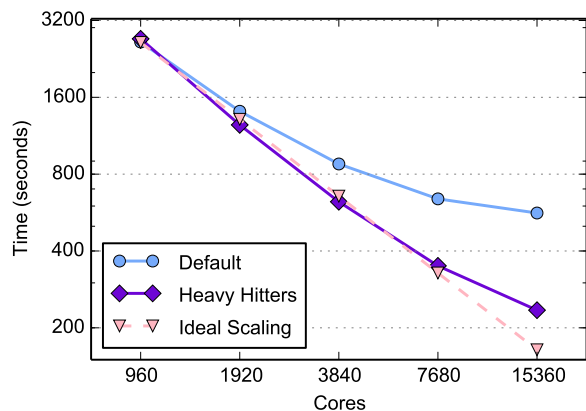
Intel Ivy Bridge processors for a total of 133,824 compute cores, and interconnected with the Cray Aries network using a Dragonfly topology. For our experiments, we use Edison's parallel Lustre /scratch3 file system, which has 144 Object Storage Servers providing 144-way concurrent access to the I/O system with an aggregate peak performance of 72 GB/sec.

To analyze HipMer performance behavior we examine a human genome for a member of the CEU HapMap population (identifier NA12878) sequenced by the Broad Institute. The genome contains 3.2 Gbp (billion base pair) assembled from 2.9 billion reads (290 Gbp of sequence), which are 101 bp in length, from a paired-end insert library with mean insert size 395 bp. Additionally, we examine the grand-challenge hexaploid wheat genome (Triticum aestivum L.) containing 17 Gbp from 2.3 billion reads (477 Gbp of sequence) for the homozygous bread wheat line 'Synthetic W7984'. Wheat reads are 150-250 bp in length from 5 paired-end libraries with insert sizes 240-740 bp. Also, for the scaffolding phase we leveraged (in addition to the previous libraries) two long-insert paired-end DNA libraries with insert sizes 1 kbp and 4.2 kbp. This important genome was only recently sequenced for the first time [13], and requires high-performance analysis due to its size and complexity.

Additionally, we leverage HipMer for k-mer analysis and contig generation of a massive-scale metagenomics dataset, containing wetland soil samples that are a timeseries dataset across several physical sites from the Twitchell Wetland in the San Francisco Bay-Delta. These samples consist of either the rhizome-associated soil of two important wetland plants (cattail and tule) or bulk soil material, and consist of 7.5 billion reads (1.25 Tbase). These data are 4.07× larger than the human data (2637 GB vs 648 GB) and significantly more complex due to the diversity of the metagenomics. Hence, they require extreme-scale computing for effective analysis.

In this section we only present performance results for components with improvements and optimizations (heavy hitters for k-mer analysis, communication-avoiding contig generation), new components (the various scaffolding steps), and end-to-end results for the whole *de novo* assembly pipeline. A detailed analysis of existing results can be found in our previous work [3, 12]. Also, the accuracy of the assemblies generated by HipMer is not reported in this paper since it produces results that are biologically equivalent to the

original Meraculous results. The original Meraculous's accuracy has been exhaustively studied using different metrics and datasets (including the human genome) in the Assemblathon I [14] and II [2] studies. In those studies, Meraculous is also compared against other de novo genome assemblers and found to excel in most of the metrics.

5.1 Optimization Effects on k-mer Analysis

Figure 6 presents the effect of identifying heavy hitters (i.e. high frequency k-mers). As described in Section 3.1, we treat heavy hitters specially by first accumulating their counts and extensions locally, followed by a final global reduction. We do not show the results for human as the heavy hitters optimization does not significantly impact its running time.

We use $\theta = 32,000$ in our experiments, which is the number of slots in the main data structure of the Misra-Gries algorithm. Since the wheat data has approximately 330 billion 51-mers, this choice of θ only guarantees the identification of k-mers with counts above 10 million. In practice, however, the performance was not sensitive to the choice of θ, which was varied between 1K and 64K with negligible (less than 10%) performance difference. At the scale of 15,360 cores, the heavy hitters optimization results in a 2.4× improvement for the wheat data.

In the default version, the percentage of communication increases from 23% in 960 cores to 68% in 15,360 cores. In the optimized version, however, communication increases from 16% to only 22%, meaning that the method is no longer communication bound for the challenging wheat dataset and is showing close to ideal scaling up to 7,680 processors. Scaling beyond that is challenging because the run times include the overhead of reading the FASTQ files, which requires 40-60 seconds (depending on the system load). Since the Edison I/O bandwidth is already saturated by 960 cores, the I/O costs are relatively flat with increasing number of cores and thus impact scalability at the highest concurrency.

5.2 Communication-Avoiding de Bruijn Graph Traversal

To evaluate the effectiveness of our communication-avoiding algorithm we investigate performance results on the human genome data set on Edison using two concurrencies: 480 and 1,920 cores. Table 1 presents the speedup achieved by our communication avoiding algorithm (columns "oracle-1" and "oracle-4") over the basic version without an oracle hash function (column "no-Oracle"). The case labeled "oracle-1" corresponds to an oracle hash function with a per-thread memory requirement of 115 MB, while the "oracle-4" hash functions requires four times more memory, i.e. 461 MB per-thread. Recall that an increase of dedicated memory for the oracle hash function corresponds to more effective communication-avoidance. At 480 cores the communication-avoiding algorithms yields a significant speedup up to 2.8× over the basic algorithm, while at 1,920 cores we achieve an improvement in performance up to 1.9×.

The data presented in Table 2 show that the performance improvements are related to a reduction in communication. As expected, the basic algorithm performs mostly off-node communication during the traversal. In particular, 92.8% of the lookups result in off-node communication at 480 cores, and 97.2% of the lookups yield off-node communication at 1,920 cores. By contrast, even a lightweight oracle hash func-

Cores	Graph traversal time (sec)			Speedup	
	no-Oracle	oracle-1	oracle-4	oracle-1	oracle-4
480	145.8	105.8	52.1	1.4×	2.8×
1,920	46.3	35.9	24.8	1.3×	1.9×

Table 1: Speedup of communication-avoiding parallel de Bruijn graph traversal vs. the basic (no Oracle) algorithm.

Cores	Off-node communication (% of total)			% reduction in off-node comm	
	no-Oracle	oracle-1	oracle-4	oracle-1	oracle-4
480	92.8 %	54.6 %	22.8 %	41.2 %	75.5 %
1,920	97.2 %	54.5 %	23.0 %	44.0 %	76.3 %

Table 2: Reduction in communication time via oracle hash functions.

tion "oracle-1" reduces the off-node communication by 41.2% and 44%, at 480 and 1,920 cores, respectively. By allocating more memory for the oracle hash function ("oracle-4") we can further decrease the off-node communication, by 75.5% and 76.3%. Note that the remainder of this paper only examines the analysis of single individual genomes and cannot effectively leverage the oracle partitioning. Therefore this optimization is turned off during the experiments for the results presented in this paper. However, the oracle partitioning infrastructure is crucial for the scenario of exploring assemblies of multiple genomes of the same species and for the scenario of optimizing an individual assembly by iterating over multiple lengths for the k-mers. These two use-case scenarios of HipMer are the subject of ongoing studies in biology.

5.3 Parallel Scaffolding

Figure 7 (left) illustrates the strong scaling performance for the scaffolding module using the human dataset. This graph depicts three scaffolding component runtimes at each concurrency: (1) the merAligner runtime, the most time consuming module of scaffolding, (2) the time required by the gap closing module and (3) the execution time for the remaining steps of scaffolding. The final component includes computation of contig depths and termination states, identification of bubbles, insert size estimation, location of splints & spans, contig link generation and ordering/orientation of contigs — which have all been parallelized and optimized for the HipMer implementation of our study. Scaling to 7,680 cores and 15,360 cores results in parallel efficiencies (relative to the 480 core baseline) of 0.48 and 0.33, respectively. While merAligner exhibits effective scaling behavior all the way up to 15,360 cores (0.64 efficiency), the scaling of gap closing is more constrained by I/O and only achieves a parallel efficiency of 0.35 at 7,680 cores, and 0.19 at 15,360 cores. Similarly, the remaining modules of scaffolding show scaling up to 7,680 cores, while once again suffering from I/O saturation at the highest (15,360 cores) concurrency.

Figure 7 (right) presents the strong scaling behavior of scaffolding for the larger wheat dataset. We attain parallel efficiencies of 0.61 and 0.37 for 7,680 and 15,360 cores respectively (relative to the 960 core baseline). While merAligner and gap closing exhibit similar scaling to the human test case, the remaining scaffolding steps consume a significantly higher fraction of the overall runtime. There are two main reasons for this behavior. First, the highly repetitive nature
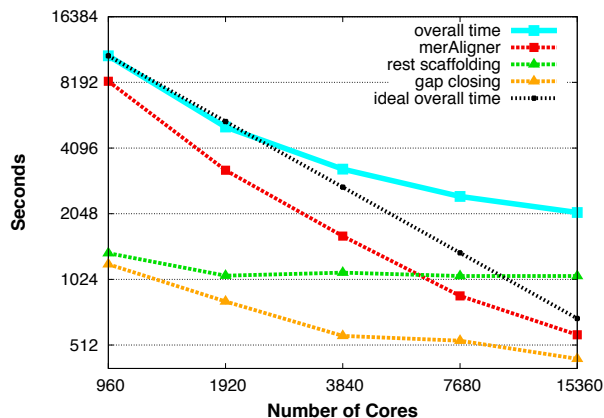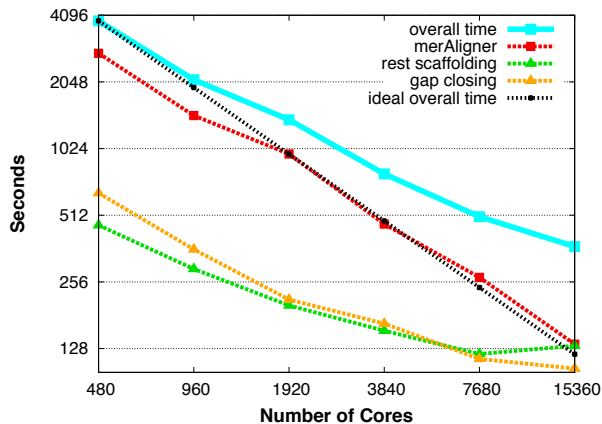
Figure 7: Strong scaling of **scaffolding** for (left) human genome and (right) wheat genome. Both axes are in log scale.

of the wheat genome leads to increased fragmentation of the contig generation compared with the human DNA, resulting in contig graphs that are contracted by a smaller fraction. (We refer to contraction with respect to the size reduction of the k-mer de Bruijn graph when it is simplified to equivalent contig graphs.) Hence, the serial component of the contig ordering/orientation module requires a relatively higher overhead compared with the human data set. Second, the execution of the wheat pipeline as performed in our previous work [5] requires four rounds of scaffolding, resulting in even more overhead within the contig ordering/orientation module.

5.4 Twitchell Wetlands Metagenome Scaling

Metagenomes or environmental sequencing is one of the ways that researchers can investigate the genomes of microbial communities that are otherwise unable to be discovered. Because these datasets, depending on their origin, can contain upwards of 10,000 different species and strains, the assembly requires many terabytes of memory, which sometimes prevents the largest datasets from being assembled at all. Even when assembly is possible, typically 90% of the reads [15] cannot be assembled because the sampling was not high enough to resolve low-abundance organisms, and yet collecting more sequencing data would exacerbate the computing resource requirements.

We show the performance of HipMer's k-mer analysis and contig generation steps on the Twitchell wetlands metagenome dataset, one of the largest and most complex microbial communities available. Since this dataset includes a widely diverse set of organisms, its k-mer count histogram is much flatter and the Bloom filters are less effective. In particular, only 36% of k-mers have a single count (versus 95% for human), which necessitates a much bigger working set

Cores	k-mer analysis	contig generation	file I/O
10K	776.04	47.83	92.81
20K	525.34	31.02	95.42

Table 3: Performance (seconds) and scaling of k-mer analysis and contig generation for metagenomic data. For this table only, we report the time spent in I/O on a separate column to highlight the scaling of non-I/O components.

for the local hash tables that store the k-mers and their extensions. Therefore, we only show results for 10K and 20K cores, which consumed 13.8 TB and 16.3 TB aggregate distributed memory, respectively, for analyzing the k-mers. Since metagenome assembly is not within the original design of the Meraculous algorithms, and single-genome logic may introduce errors in the scaffolding of a metagenome, we will only execute HipMer through the uncontested contig generation, and plan future work to adapt HipMer to properly scaffold a metagenome at scale.

As shown in Table 3, all steps except file I/O show scaling. Since the I/O is well saturated at both concurrencies, the time difference is solely due to system load. Though the accomplishment of generating an assembly for a dataset of this size is an achievement in and of itself, the ability to rapidly generate contigs opens up the possibility of further iterating the metagenome assembly parameter space in a way that was until now unprecedented.

5.5 End-to-End Performance

Figure 8 shows the end-to-end strong scaling performance of HipMer on the human (left) and the wheat (right) data sets. For the human dataset at 15,360 cores we achieve a speedup of 11.9× over our baseline execution (480 cores). At this extreme scale the human genome can be assembled from raw reads in just ~8.4 minutes. On the complex wheat dataset, we achieve a speedup up to 5.9× over the baseline of 960 core execution, allowing us to perform the end-to-end assembly in 39 minutes when leveraging 15,360 cores. In the end-to-end experiments, a significant fraction of the overhead is spent in parallel scaffolding (e.g. 68% for human at 960 cores); k-mer analysis requires significantly less runtime (28% at 960 cores) and contig generation is the least expensive computational component (4% at 960 cores).

5.6 Competing Parallel De Novo Assemblers

To compare the performance of HipMer relative to existing parallel *de novo* end-to-end genome assemblers we evaluated Ray [16, 17] (version 2.3.0) and ABySS [18] (version 1.3.6) on Edison using 960 cores. Both of these assemblers are described in more detail in Section 6. Ray required 10 hours and 46 minutes for an end-to-end run on the Human dataset. ABySS, on the other hand, took 13 hours and 26 minutes solely to get to the end of contig generation. The subse-
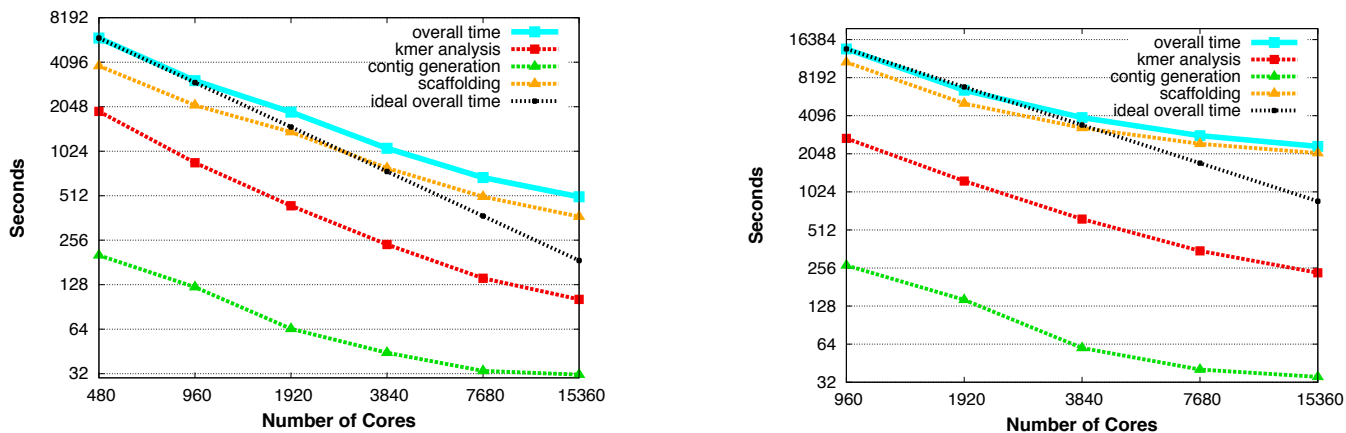
Figure 8: **End-to-end strong scaling** for (left) human genome and (right) wheat genome. Both axes are in log scale.

quent scaffolding steps are not distributed-memory parallel. At this concurrency on Edison, HipMer is approximately 13 times faster than Ray and *at least* 16 times faster than ABySS.

6. RELATED WORK

As there are many *de novo* genome assemblers and assessement of the quality of these is well beyond the scope of this paper, we refer the reader to the work of the Assemblathons I [14] and II [2] as examples of why Meraculous [1] was chosen to be scaled, optimized and re-implemeted as HipMer. For performance comparisons, we primarily refer to parallel assemblers with the potential for strong scaling on large genomes (such as plant, mamalian and metagenomes) using distributed computing or clusters.

Ray [16, 17] is an end-to-end parallel *de novo* genome assembler that utilizes MPI and exhibits strong scaling. It can produce scaffolds directly from raw sequencing reads and produces timing logs for every stage. One drawback of Ray is the lack of parallel I/O support for reading and writing files. As shown in Section 5.6 Ray is approximately 13× slower than HipMer for the human data set on 960 cores.

ABySS [18] was the first *de novo* assembler written in MPI that also exhibits strong scaling. Unfortunately only the first assembly step of contig generation is fully parallalized with MPI and the subsequent scaffolding steps must be performed on a single shared memory node. As shown in Section 5.6 ABySS' contig generation phase is approximately 16× slower than HipMer's entire end-to-end solution for the human data set on 960 cores.

PASHA [19] is another partly MPI based de Bruijn graph assembler, though not all steps are fully parallelized as its algorithm, like ABySS, requires a large memory single node for the last scaffolding stages. The PASHA authors do claim over 2× speedup over ABySS on the same hardware.

YAGA [20] is a parallel distributed-memory that is shown to be scalable except for its I/O, but the authors could not obtain a copy of this software to evaluate. HipMer employs efficient, parallel I/O so is expected to achieve end-to-end performance scalability. Also, the YAGA assembler was designed in an era when the short reads were extremely short and therefore its run-time will be much slower for current high throughput sequencing systems.

SWAP [21] is a relatively new parallelized MPI based de Bruijn assembler that has been shown to assemble contigs for the human genome and performs strong scaling up to about one thousand cores. However, SWAP does not perform any of the scaffolding steps, and is therefore not an end-to-end *de novo* solution. Additionally, the peak memory usage of SWAP is much higher than HipMer, as it does not leverage Bloom filters.

There are several other shared memory assemblers that produce high quality assemblies, including ALLPATHS-LG [22] (pthreads/OpenMP parallel depending on the stage), SOAPdenovo [23] (pthreads), DiscovarDenovo [24] (pthreads) and SPADES [25] (pthreads), but unfortunately each of these requires a large memory node and we were unsuccessful at running these experiments using our datasets on a system containing 512GB of RAM due to lack of memory. This shows the importance of strong scaling distributed memory solutions when assembling large genomes.

7. CONCLUSIONS AND FUTURE WORK

Next-generation short-read sequencing technology has resulted in explosive growth of sequenced DNA. However, *de novo* assembly has been unable to keep pace with the flood of data, due to vast computational requirements and the algorithmic complexity of assembling large-scale genomes and metagenomes. In this work we address this challenge head on by developing HipMer, an end-to-end high performance *de novo* assembler designed to scale to massive concurrencies. Our work is based on the Meraculous assembler, which has been shown to be one of the top *de novo* approaches in recent Assemblathon competitions [2].

We developed several novel algorithmic advancements by leveraging the efficiency and programmability of UPC, including optimized high-frequency k-mer analysis, communication avoiding de Bruijn graph traversal, advanced I/O optimization, and extensive parallelization across the numerous and complex application phases. We emphasize that distributed hash tables lie in the heart of HipMer and the main operations on them are irregular lookups. Therefore our algorithms avoid synchronization and message matching logic that would be imposed by a two-sided communication model and instead employ the asynchronous one-sided communication capabilities of UPC. The global address space is

also convenient for these algorithms, since variables may be directly read and written by any processor.

Overall results show unprecedented performance and scalability, attaining an overall runtime of 8.4 minutes for the human DNA at 15K cores on the Cray XC30, compared with 10.8 hours for Ray and 23.8 hours for the original Meraculous application. Additionally, we explored performance on the grand-challenge wheat genome, which, to date, has been too large and complex for most modern *de novo* assemblers. Our results demonstrated impressive scalability, allowing the completed wheat assembly in just 39 minutes using 15K cores.

Furthermore, we have shown that the distributed memory implementation of HipMer can successfully assemble contigs from one of the largest, most complex and challenging, deeply sequenced metagenome datasets in less than 11 minutes using 20K cores. We have begun to apply the Meraculous algorithm to the new challenge of metagenome assembly and plan to continue to develop the necessary algorithmic changes for scaffolding, variant resolution and clustering, and will then adapt this code into an end-to-end high performance metagenome assembler.

Using our HipMer technology enables — for the first time — assembly throughput to exceed the capability of all the world's sequencers, thus ushering in a new era of genome analysis. Additionally, HipMer makes it possible to improve assembly quality by running tuning parameter sweeps that were previously prohibitively expense. The combination of high performance sequencing and efficient *de novo* assembly is the basis for numerous bioinformatic transformations, including advancement of global food security and a new generation of personalized medicine.

Acknowledgments

References

[1] J. A. Chapman, I. Ho, S. Sunkara, S. Luo, G. P. Schroth, and D. S. Rokhsar, "Meraculous: De novo genome assembly with short paired-end reads," *PLoS ONE*, vol. 6, no. 8, p. e23501, 08 2011.

[2] K. Bradnam1, J. Fass, A. Alexandrov *et al.*, "Assemblathon 2: evaluating de novo methods of genome assembly in three vertebrate species," *GigaScience*, vol. 2, no. 10, 2013.

[3] 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 *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'14)*, 2014.

[4] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[5] J. Chapman, M. Mascher, A. Buluç, K. Barry, E. Georganas, A. Session, V. Strnadova, J. Jenkins, S. Sehgal, L. Oliker, J. Schmutz, K. Yelick, U. Scholz, R. Waugh, J. Poland, G. Muehlbauer, N. Stein, and D. Rokhsar, "A whole-genome shotgun approach for assembling and anchoring the hexaploid bread wheat genome," *Genome Biology*, vol. 16, no. 26, 2015.

[6] J. Misra and D. Gries, "Finding repeated elements," *Science of computer programming*, vol. 2, no. 2, pp. 143–152, 1982.

[7] E. D. Demaine, A. López-Ortiz, and J. I. Munro, "Frequency estimation of internet packet streams with limited space," in *Proceedings of the 10th Annual European Symposium on Algorithms (ESA)*, 2002, pp. 348–360.

[8] R. M. Karp, S. Shenker, and C. H. Papadimitriou, "A simple algorithm for finding frequent elements in streams and bags," *ACM Transactions on Database Systems (TODS)*, vol. 28, no. 1, pp. 51–55, 2003.

[9] P. K. Agarwal, G. Cormode, Z. Huang, J. M. Phillips, Z. Wei, and K. Yi, "Mergeable summaries," *ACM Transactions on Database Systems (TODS)*, vol. 38, no. 4, p. 26, 2013.

[10] M. Cafaro and P. Tempesta, "Finding frequent items in parallel," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 15, pp. 1774–1788, 2011.

[11] M. Howison, "High-throughput compression of FASTQ data with SeqDB," *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, vol. 10, no. 1, pp. 213–218, 2013.

[12] E. Georganas, A. Buluç, J. Chapman, L. Oliker, D. Rokhsar, and K. Yelick, "merAligner: A Fully Parallel Sequence Aligner," in *Proceedings of the IPDPS*, 2015.

[13] K. F. Mayer, J. Rogers, J. Doležel, C. Pozniak, K. Eversole, C. Feuillet, B. Gill, B. Friebe, A. J. Lukaszewski, P. Sourdille *et al.*, "A chromosome-based draft sequence of the hexaploid bread wheat (triticum aestivum) genome," *Science*, vol. 345, no. 6194, 2014.

[14] D. Earl, K. Bradnam, J. St John, A. Darling *et al.*, "Assemblathon 1: a competitive assessment of de novo short read assembly methods." *Genome research*, vol. 21, no. 12, pp. 2224–2241, Dec. 2011.

[15] 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*, vol. 111, no. 13, pp. 4904–4909, 2014.

[16] S. Boisvert, F. Laviolette, and J. Corbeil, "Ray: simultaneous assembly of reads from a mix of high-throughput sequencing technologies," *Journal of Computational Biology*, vol. 17, no. 11, pp. 1519–1533, 2010.

[17] S. Boisvert, F. Raymond, É. Godzaridis, F. Laviolette, J. Corbeil *et al.*, "Ray meta: scalable de novo metagenome assembly and profiling," *Genome Biology*, vol. 13, no. R122, 2012.

[18] J. T. Simpson, K. Wong *et al.*, "Abyss: a parallel assembler for short read sequence data," *Genome research*, vol. 19, no. 6, pp. 1117–1123, 2009.

[19] Y. Liu, B. Schmidt, and D. L. Maskell, "Parallelized short read assembly of large genomes using de Bruijn graphs," *BMC bioinformatics*, vol. 12, no. 1, p. 354, 2011.

[20] B. G. Jackson, M. Regennitter *et al.*, "Parallel de novo assembly of large genomes from high-throughput short reads," in *IPDPS'10*. IEEE, 2010.

[21] J. Meng, B. Wang, Y. Wei, S. Feng, and P. Balaji, "SWAP-assembler: scalable and efficient genome assembly towards thousands of cores," *BMC Bioinformatics*, vol. 15, no. Suppl 9, p. S2, 2014.

[22] S. Gnerre, I. MacCallum, D. Przybylski, F. Ribeiro, J. Burton, B. Walker, T. Sharpe, G. Hall, T. Shea, S. Sykes, A. Berlin, D. Aird, M. Costello, R. Daza, L. Williams, R. Nicol, A. Gnirke, C. Nusbaum, E. Lander, and D. Jaffe, "High-quality draft assemblies of mammalian genomes from massively parallel sequence data," in *Proceedings of the National Academy of Sciences USA*, 2010.

[23] R. Li, H. Zhu *et al.*, "De novo assembly of human genomes with massively parallel short read sequencing," *Genome research*, vol. 20, no. 2, pp. 265–272, 2010.

[24] D. Jaffe, "Discovar: Assemble genomes and find variants," http://www.broadinstitute.org/software/discovar/blog/, 2014.

[25] A. Bankevich, S. Nurk, D. Antipov, A. A. Gurevich, M. Dvorkin, A. S. Kulikov, V. M. Lesin, S. I. Nikolenko, S. Pham, A. D. Prjibelski, A. V. Pyshkin, A. V. Sirotkin, N. Vyahhi, G. Tesler, M. A. Alekseyev, and P. A. Pevzner, "SPAdes: A new genome assembly algorithm and its applications to single-cell sequencing," *J Comput Biol.*, vol. 19, no. 5, pp. 455–477, May 2012.