Parallel De Bruijn Graph Construction and Traversal for De Novo Genome Assembly

Evangelos Georganas[†,‡], Aydın Buluç[†], Jarrod Chapman[*]
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 sequence from short, overlapping, and potentially erroneous fragments called reads. We study optimized parallelization of the most time-consuming phases of Meraculous, a state-of-the-art production assembler. First, we present a new parallel algorithm for k-mer analysis, characterized by intensive communication and I/O requirements, and reduce the memory requirements by $6.93\times$. Second, we efficiently parallelize de Bruijn graph construction and traversal, which necessitates a distributed hash table and is a key component of most de novo assemblers. We provide a novel algorithm that leverages one-sided communication capabilities of the Unified Parallel C (UPC) to facilitate the requisite fine-grained parallelism and avoidance of data hazards, while analytically proving its scalability properties. Overall results show unprecedented performance and efficient scaling on up to 15,360 cores of a Cray XC30, on human genome as well as the challenging wheat genome, with performance improvement from *days* to *seconds*.

I. Introduction

Recent advances in sequencing technology have made the redundant sampling of genomes extremely cost-effective. Such a sampling consists mostly of shorts reads with low error rates most of which can be aligned to a reference genome in a straightforward way. De novo genome assemblers, on the other hand, reconstruct an unknown genome from a collection of short reads, without requiring a reference genome. De novo assembly is therefore more powerful and flexible than mapping-based approaches, since it can be applied to novel genomes and can in principle discover previously unknown sequences in species for which reference genomes do not exist. This advantage, however, comes at a cost of significantly increased run time and memory requirements. If de novo assembly could be performed rapidly, it would be preferable to mapping based approaches for large-scale human genome sequencing, and other biomedical model and agricultural species.

Meraculous [1] is a state-of-the-art de novo assembler for short reads, which has demonstrated completeness and accuracy [2]. It is a hybrid assembler that combines aspects of de Bruijn-graph-based assembly with overlap-layout-consensus approaches. In the de Bruijn-based approach short sequence reads are represented as all (overlapping) substrings of length k, referred to as k-mers. Meraculous leverages the base quality scores produced by sequencing instruments to identify those k-mers that (1) occur often enough in the reads that they
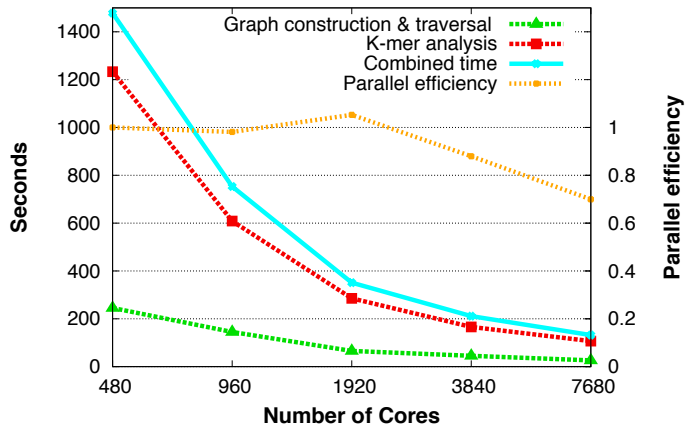
Fig. 1: Performance and strong scaling of our de Bruijn graph construction & traversal and k-mer analysis steps on Cray XC30 for the human genome. The top three timing curves are with respect to the first y-axis (left) whereas the parallel efficiency curve is with respect to the second y-axis (right). The x-axis uses a log scale.

are unlikely to contain an error and (2) have unique high quality extensions in the reads at each end of the k-mer. The result is a possibly disconnected linear 'UU sub-graph' of the full de Bruijn graph representation of the reads that can be efficiently traversed to yield nearly error-free contiguous sequences. The traversal of the 'UU graph' is a significant computational bottleneck that must be performed on a single large memory node and takes on the order of days for human but is prohibitive for wheat.

In this work, we present a highly-parallel implementation of the most challenging phases of the de Bruijn-based whole genome assembly, namely the k-mer analysis step and the de Bruijn graph construction & traversal steps. These steps are described in detail in Section II-A. Our main performance result is summarized in Figure 1, showing unprecedented scalability to several thousand cores at which point we can go from raw sequencing data to contigs in less than three minutes for a human genome. Although this work focuses on Meraculous, our parallelization techniques are generally applicable to other de Bruijn-based assemblers. Overall this study makes numerous contributions in significantly optimizing and parallelizing these code components using novel algorithmic and programming methodologies, including:

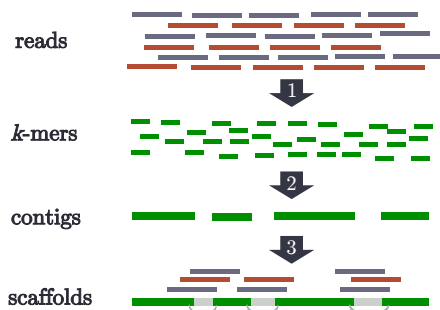• A new parallel algorithm for k-mer analysis. We success-

Fig. 2: Meraculous assembly flow chart.

Fig. 3: A de Bruijn graph with $k = 3$. We highlight with red boxes the UU k-mers and indicate the corresponding contigs.

fully reduced the memory requirements by $6.93\times$ using Bloom filters and probabilistic counting techniques, and attained remarkable scaling up to 15K cores for this I/O- and communication-intensive computation.

- A novel parallel algorithm for de Bruijn graph construction & traversal that leverages the one-sided communication capabilities of the Unified Parallel C (UPC) to facilitate the requisite random access pattern in the distributed hash table.
- Novel communication optimizations at the hash table construction and the design of a lightweight synchronization scheme for the parallel de Bruijn graph traversal.
- Analytical propositions to quantify the expected data hazards within the de Bruijn graph traversal with respect to concurrency.
- Unprecedented performance and scaling on NERSC's Edison system, a Cray XC30, using *the human genome and the grand-challenge wheat genome* to over 15K processors.

Our work shows that efficient utilization of distributed memory architectures is possible for this irregular problem. Hence, it removes the requirement of large memory machines imposed by the original de Bruijn graph construction code (even for modestly sized genomes) by exploiting UPC's PGAS (Partitioned Global Address Space) capabilities. We demonstrate that our implementation is portable and executable on both shared- and distributed-memory architectures without modifications.

II. DE NOVO GENOME ASSEMBLY OVERVIEW

De novo whole genome assembly is the process of reconstructing the genomic sequence of an organism from short reads. This is made possible by sequencing the genome redundantly at some *depth* so that the reads overlap. Overlap-layout-consensus (OLC) [3] and the de Bruijn graph [4] are two types of commonly used techniques for de novo assembly.

The OLC approach is based on the concept of an overlap graph and is more suitable for assembling genomes that are sequenced in longer reads that are based on Sanger technology. In the overlap graph, the vertices are short reads and the edges represent overlap between those reads. There are several drawbacks of the OLC approach for short-read assembly: (1) the resulting overlap graph is linear on the order of the sequencing data, which is larger than the genome itself by a factor of *depth*, (2) the computation of the overlap graph requires all pairwise read-vs-read comparisons, (3) the problem is to find a Hamiltonian path in the resulting graph, a problem known
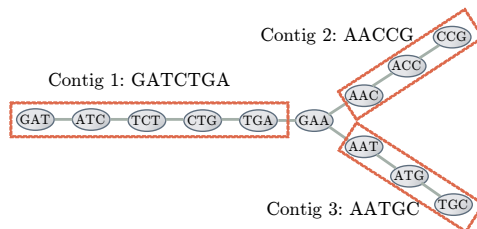
to be NP-complete. By contrast, de Bruijn graph is linear in the order of the genome size, does not require the all pairwise read-vs-read comparison for graph construction, and solves a much simpler Eulerian path problem. In the de Bruijn graph, vertices are k-mers. For those reasons, our work focuses on a de Bruijn graph based assembler Meraculous.

A. De Bruijn Assembly a la Meraculous

The steps of the de Bruijn assembly, as interpreted by Meraculous can be summarized as:

1) **K-mer analysis** (a) *k-mer counting*: The reads are chopped into k-mers that overlap by $k-1$, and a count is kept for each k-mer occurring more than once. The value of k is chosen to balance the need for: (i) uniqueness of the k-mers in the genome, favoring larger k, and (ii) presence of multiple overlapping error-free k-mers in each read, favoring k smaller than $1/error$. (b) *k-mer characterization*: The generated k-mers are preprocessed and only those with unique high-quality forward and backward single-base extensions (henceforth called UU k-mers) are kept.

2) **Contig generation** (a) *De Bruijn graph construction*: The UU k-mers are stored in a hash table where the "key" is the k-mer and the "value" is a two-letter code [ACGT][ACGT] that indicates the unique bases that immediately precede and follow the k-mer in the read dataset. This hash table represents the "UU graph". (b) *De Bruijn graph traversal*: k-mers are selected to seed forward and reverse traversals of the UU graph to produce an initial set of "UU contigs" (henceforth called *contigs*). The resulting contigs are independent of the selection of seed k-mers and each UU k-mer is represented exactly once in the full set of contigs.

3) **Scaffolding and gap closing:** Information from the initial reads is leveraged to orient the contigs, close gaps among them and form the *scaffolds*, which is the result of the de novo assembly.

Figure 2 presents a visualization of these steps where the first arrow depicts k-mer analysis, the second arrow depicts contig generation, and the third arrow depicts scaffolding. We emphasize that this work parallelizes steps 1 and 2 that are the most time-consuming and parallelizing step 3 will be the subject of future work. Figure 3 illustrates an example de Bruijn graph with $k = 3$. Every node in the graph is a k-mer and two nodes that have an overlap of $k - 1$ bases are connected with an edge. In this graph we can identify three contigs since the GAA node does not have a unique forward extension (it has two forward high quality extensions).
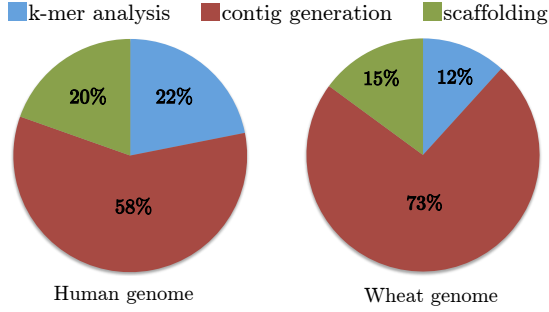
Fig. 4: Percentage of total execution time for each phase of the original Meraculous for (left) human and (right) wheat data set. The original runtime for these genome assemblies are 75.6 and 165.4 hours respectively. Note that these pie charts are just for motivation and only characterize the costs of the steps in Meraculous prior to this work.

B. The Meraculous Assembler

The production version of Meraculous is written in Perl, where some of the phases are readily parallelized but with limited levels of concurrency; however, the Meraculous de Bruijn graph traversal is completely serial due to the fine-grained parallelism and data-hazards required for an effective parallel implementation. Figure 4 shows the percentage breakdown of the original Meraculous code for the human and wheat data sets, using maximum useful concurrency levels for the available parallel regions. Observe that the work described in this paper parallelizes the two most expensive components, contig generation as well as k-mer analysis, which together account for 80% and 85% of the overall runtimes for human and wheat, respectively. For both genomes, the original contig generation step ran serially whereas the k-mer analysis used 104 and 206 nodes (each equipped with 48 GB of memory) of a cluster, respectively for human and wheat. Parallel optimization of scaffolding will be the subject of future work.

The quality of assemblies with Meraculous have been previously evaluated compared with other assemblers, using synthetic data in Assemblathon I [5] and using real vertebrate datasets (bird, fish, snake) in Assemblathon II [2]. Meraculous performed exceptionally well in base-level and scaffolding accuracy, with steadily improving contig sizes. In the recent Assemblathon II, Meraculous achieved first place rankings in as many or more metrics than all but one participating group, with continued outstanding performance in metrics associated with global assembly accuracy. According to this study, there is no distributed memory assembler ranked above Meraculous. This justifies our work on developing a fast distributed memory parallelization of Meraculous to achieve both quality and computational efficiency. For the computationally intensive steps described here, our work is a faithful parallelization of the latest Meraculous algorithm, producing identical results.

III. PARALLEL K-MER ANALYSIS

K-mers are contiguous DNA subsequences of length k that are smaller than the read length, a property that allows the exploitation of overlaps among reads without resorting to expensive all-vs-all comparison. Since both k-mer counting and

---

Algorithm 1 Parallel three-pass k-mer analysis

1: **for** all processors p_i **in parallel do**
2:   **while** there are *reads* to process **do**   ▷ Pass 1
3:     *kmers* ←PARSETOKMERS(*reads*)
4:     *est* ← HYPERLOGLOG(*kmers*)
5:   *globalest* ← PARALLELREDUCE(*est*)
6:   *bfilter* ← BLOOMFILTER(*globalest* /*p*)
7:   **while** there are *reads* to process **do**   ▷ Pass 2
8:     **for** a batch of *reads* **do**   ▷ memory constraint
9:       *kmers* ←PARSETOKMERS(*reads*)
10:       *owners* ←MAPTOPROCESSORS(*kmers*)
11:     ALLTOALLV(*kmers*, *owners*)
12:     **for** each incoming k-mer *km* **do**
13:       **if** *km* exists in *bfilter* **then**
14:         INSERT(*locset*, *km*, *null*)
15:       **else**
16:         INSERT(*bfilter*, *km*)
17:   **while** there are *reads* to process **do**   ▷ Pass 3
18:     **for** a batch of *reads* **do**   ▷ memory constraint
19:       *kmers* ←PARSETOKMERS(*reads*)
20:       *owners* ←MAPTOPROCESSORS(*kmers*)
21:       *exts* ←FETCHEXTENSIONS(*kmers*, *reads*)
22:     ALLTOALLV(*kmers*, *owners*)
23:     ALLTOALLV(*exts*, *owners*)
24:     **for** each incoming k-mer *km* and extension *ext* **do**
25:       **if** *km* exists in *locset* **then**
26:         UPDATE(*locset*, *km*, *ext*)

---

characterization phases have similar computational patterns, we chose to implement them as one step.

Counting the frequencies of each distinct k-mer involves reading file(s) that includes DNA short reads, parsing the reads into k-mers, and keeping a count of each distinct k-mer that occurs more than ϵ times ($\epsilon \approx 1, 2$). The reason for such a cut-off is to eliminate sequencing errors. K-mer characterization additionally requires keeping track of all possible extensions of the k-mer from either side. This is performed by keeping two short integer arrays of length four per k-mer, where each entry in the array keeps track of the number of occurrences of each nucleotide [ACGT] on either end. If a nucleotide on an end appears more times than a threshold t_{hq}, it is characterized as high quality extension.

One of the difficulties with performing k-mer analysis in distributed memory is that the size of the intermediate data (the set of k-mers) is significantly larger than the input, since each read is subsequenced with overlaps of $k - 1$ base pairs. As each process reads a portion of the reads, a deterministic map function maps each k-mer to a processor. This map, $map : k\text{-}mer \rightarrow \{1, \ldots, p\}$, assigns all the occurrences of a particular sequence to the same processor, thus eliminating the need for a global hash table. Because a k-mer can be seen in two different orientations (due to reverse complementarity), the map flips it to the lexicographically smaller orientation before calculating the process number to which it is mapped.

Algorithm 1 lists our parallel k-mer analysis algorithm, which does three streaming passes, as described in detail below. During the I/O steps (lines 2, 7, and 17), each processor reads an equal amount of sequence data. The algorithm achieves almost perfect load balance in terms of the number of distinct k-mers assigned to each processor, thanks to our

use of the strong MurmurHash [6] function to implement the k-mer to processor map. The *locset* is a set data structure that does not allow duplicates, hence potential reinsertion attempts (line 14) are treated as no-ops.

K-mer analysis algorithm uses MPI's irregular personalized communication primitive, ALLTOALLV. Since the k-mer to processor map is uniform, as described before, the input to the ALLTOALLV is evenly distributed in each processor. If h is the maximum data received by any processor and there are p processors, then ALLTOALLV can be implemented with a communication volume of $O(h + p^2)$ per processor [7]. Unfortunately, the data received per processor, h, is not necessarily evenly distributed due to potential presence of high count k-mers. It is most problematic when there are less than p high count k-mers as there is no way to distribute them evenly in that case. As shown in Section VII, this is the case for wheat but not human.

A. Eliminating Erroneous K-mers

The previously mentioned deterministic k-mer to processor mapping allows us to use hash tables that are local to each processor. Even then, memory consumption quickly becomes a problem due to errors because a single nucleotide error creates k erroneous k-mers. It is not uncommon to have over 80% of all distinct k-mers erroneous, depending on the read length and the value of k. We ameliorate this problem using Bloom filters, which were previously used in serial k-mer counters [8].

A Bloom filter [9] is a space-efficient probabilistic data structure used for membership queries. It might have false positives, but no false negatives. If a k-mer was not seen before, the filter can accidentally report it as 'seen'. However, if a k-mer was previously inserted, the Bloom filter will certainly report it as 'seen'. This is suitable for k-mer counting as no real k-mers will be missed. If the Bloom filter reports that a k-mer was seen before, then we insert that k-mer to the final *locset* that does the actual counting. Our novelty is the discovery that localization of k-mers via the deterministic k-mer to processor mapping is *necessary and sufficient* to extend the benefits of Bloom filters to distributed memory.

The false positive rate of a Bloom filter is $Pr(e) = (1 - e^{-hn/m})^h$ for m being the number of distinct elements in the dataset, n the size of the Bloom filter, and h the number of hash functions used. There is an optimal number of hash functions given n and m, which is $h = \ln 2 \cdot (m/n)$. In practice, we achieve approximately 5% false positive rate using only 1-2$\%$ of the memory that would be needed to store the data directly in a hash table (without the Bloom filter). Hence, in a typical dataset where 80% of all k-mers are errors, we are able to filter out 76% of all the k-mers using almost no additional memory. Hence, we can effectively run a given problem size on a quarter of the nodes that would otherwise be required.

B. Estimating the Bloom Filter Size

We have so far ignored that Bloom filters need to know the number of distinct elements expected to perform optimally. While dynamically resizing a Bloom filter is possible, it is

expensive to do so. We therefore use a cardinality estimation algorithm to approximate the number of distinct k-mers. Specifically, we use the Hyperloglog algorithm [10], which achieves less than $1.04/\sqrt{m}$ error for a dataset of m distinct elements. Hyperloglog requires a only several KBs of memory to count trillions of items. The basic idea behind cardinality estimators is hashing each item uniformly and maintaining the minimum hash value. Hyperloglog maintains multiple buckets for increased accuracy and uses the number of trailing zeros in the bitwise representation of each bucket as the estimator.

The observation that leads to minimal communication parallelization of Hyperloglog is as follows. Merging Hyperloglog counts for multiple datasets can be done by keeping the minimum of their final buckets by a parallel reduction. Consequently, the communication volume for this first cardinality estimation pass is *independent of the size of the sequence data*, and is only a function of the Hyperloglog data structure size. In practice, we implement a modified version of the algorithm that uses 64-bit hash values as the original 32-bit hash described in the original study [10] is not able to process our massive datasets.

C. Parallel File I/O

A standard format to represent DNA short reads is the FASTQ format, a text file that includes one read per line with another line of the same length that encodes the quality of each base pair. Unfortunately, there is no scalable way to read a FASTQ file in parallel due to its text-based nature. One commonly used approach is to create many subfiles to be read by different processors. Unfortunately this approach creates problems for data management and can not flexibly be processed by varying numbers of processors. After evaluating numerous binary formats including BAM [11], we concluded that SeqDB [12] is the most suitable format for storing unaligned short-read data that can be read efficiently in parallel.

The key benefit of the binary SeqDB format is its current implementation on HDF5 [13]. Although SeqDB was not designed with parallelism in mind, the fact that an HDF5 file can be read by a Parallel HDF5 code makes it readable in parallel with minimal modifications. We modified the SeqDB code to enable Parallel HDF5 support via the MPI-IO interface. Both HDF5 and Parallel HDF5 are widely used and available by default in many supercomputers. The compression from FASTQ to SeqDB is a one time lossless conversion that can be performed in about an hour and a half for the whole wheat genome (about $1\,\mathrm{TB}$ at disk) using a single compute node. Since our k-mer analysis employs a multi-pass algorithm, the ability to quickly read SeqDB files in parallel becomes an important advantage. The resulting file is typically 40-50$\%$ smaller than the FASTQ file. While this compression ratio is less efficient than some other competing formats, SeqDB is much faster to decompress than alternatives. We envision the sequencing platforms adopting the SeqDB format due to its demonstrated advantages, especially for parallel processing.
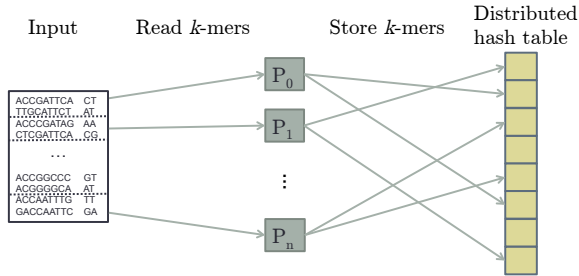
Fig. 5: Parallel de Bruijn graph construction.

IV. PARALLEL DE BRUIJN GRAPH CONSTRUCTION

We now discuss our parallel de Bruijn graph construction implementation. In particular we are working with a linear subgraph of the full de Bruijn graph, as specified by the Meraculous approach. Below, we use "de Bruijn graph" to refer to this linear subgraph of the full de Bruijn graph. In our approach a de Bruijn graph is represented as a hash table with a k-mer as "key" (a k-mer is a node in the graph) and a two-letter code [ACGT][ACGT] as "value" which represents two undirected edges incident to that node. The parallelization relies heavily on a high performance distributed hash table, shown in Figure 5 that inputs a set of k-mers with their corresponding two letter code (high quality extensions). Assuming m initial k-mers and and n processors, each processor will read m/n k-mers, hash the keys and store every entry to the appropriate bucket of the distributed hash table.

Communication optimization: The basic algorithm described in the previous paragraph suffers from fine-grained communication required to store k-mers into the distributed hash table. To mitigate this overhead we leverage a communication optimization called *aggregating stores*, shown in Figure 6. Here, a processor p_i has $n-1$ local buffers corresponding to the rest $n-1$ remote processors, where the size S of each local buffer is a tuning parameter. Every processor hashes a k-mer entry and calculates the location in the hash table where it has to be stored. Instead of incurring a remote access to the distributed hash table, the processor computes the processor ID owning that remote bucket in the hash table and stores the entry to the appropriate *local* buffer.

In this approach, when a local buffer dedicated for processor p_j becomes full, a remote aggregate transfer to the processor p_j is initiated. Each processor p_j has a pre-allocated shared space (henceforth called *local-shared stack*) where other processors can store entries destined for that processor p_j. In Section VI we describe the memory management of the local-shared stacks. Once all k-mers are processed, each processor iterates over its local-shared stack and stores each entry to the appropriate *local* bucket in the distributed hash table, in a communication-free fashion. This optimization trades off $S \times (n-1)$ extra memory for the reduction in the number of messages by a factor of S compared to the unoptimized version. This algorithm also avoids data races by having the owner process do all the inserts into the hash table.
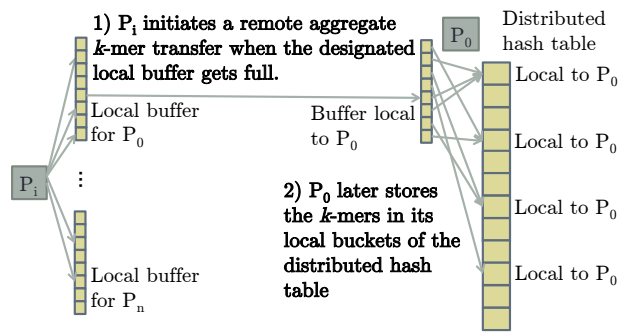

Fig. 6: Communication optimization for the de Bruijn graph construction. In this example, processor p_i performs one remote aggregate transfer to processor p_0 when the local buffer for p_0 gets full. p_0 will store these k-mers in its local buckets later by iterating over its local-shared stack.

V. PARALLEL DE BRUIJN GRAPH TRAVERSAL

Given the de Bruijn graph construction, we now describe our parallel algorithm used to traverse the graph and output the computed set of contigs (defined in Section II-A).

A. Parallel Traversal Without Conflicts

In order to form a contig, a processor p_i chooses a random k-mer from the hash table as seed and creates a new *subcontig* data structure which is represented as a string and 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 using the high quality extensions stored as values in the distributed hash table. To extend a subcontig from its right endpoint, processor p_i uses the $k-1$ last bases and its right high quality extension R for the right-most k-mer in the subcontig. It therefore concatenates the last $k-1$ bases and the extension R to form the next k-mer to be searched in the hash table. Processor p_i performs a lookup for the newly formed k-mer and if it is found successfully, the subcontig is extended to the right by the base R. The same process can be repeated until the lookup in the hash table fails, meaning that there are no more UU k-mers that could extend this subcontig in the right direction. A subcontig can be extended to its left endpoint using an analogous procedure. If processor p_i can not add more bases to either endpoint of the subcontig, then a contig has been formed and is stored accordingly.

B. Lightweight Synchronization Scheme

All processors independently start building subcontigs and no synchronization is required unless two processors pick initial k-mer seeds that eventually belong in the same contig. In this case, the processors have to collaborate and resolve this conflict in order to avoid redundant work. We now explain our lightweight synchronization scheme at the heart of the parallel de Bruijn graph traversal.

1) Basic Data Structures: Before detailing the synchronization scheme, we describe the high-level enhancements required in the data structures. First, the k-mer data structure is enriched with a binary flag used_flag that may take two possible values: UNUSED or USED. Initially this field is set to the UNUSED value. When a processor finds a k-mer in the hash table:

Found UNUSED *k*-mer in *any* direction of the subcontig:
1) *Add forward/backward extension*
2) *Mark k-mer as USED*

Found USED *k*-mers in *both* directions of the subcontig and *both* of the neighboring subcontigs are ACTIVE:
1) *Set own state to ABORTED*
2) *Pick another random seed from the hash table*

Attached in a neighboring subcontig

Picked a *k*-mer as seed and initiated a subcontig

Reached both endpoints of a contig:
1) *Set state to COMPLETE and store contig*
2) *Pick another random seed from hash table*

Found USED *k*-mers in both directions of the subcontig and one of the neighboring subcontigs is ABORTED:
1) *Attach the ABORTED subcontig to the local subcontig*
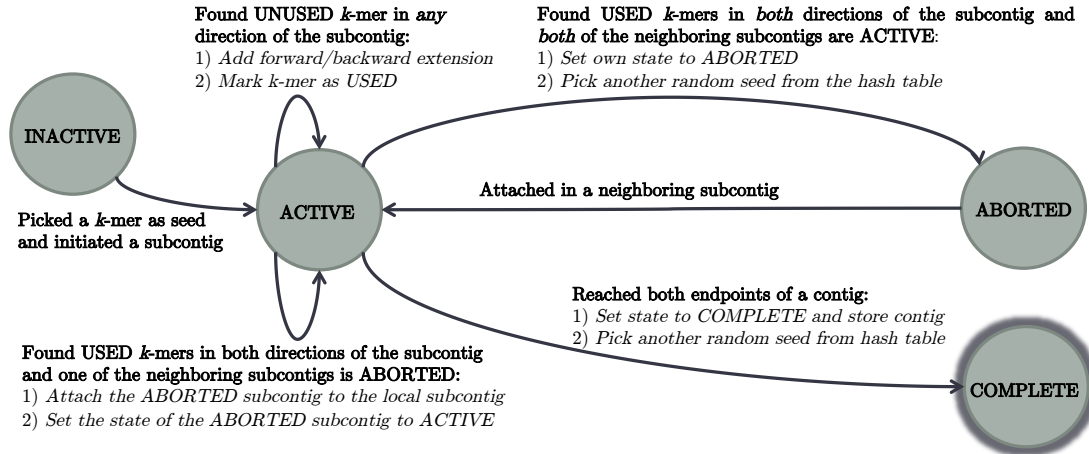2) *Set the state of the ABORTED subcontig to ACTIVE*

Fig. 7: Subcontig's state machine used for synchronization during the parallel de Bruijn graph traversal. With boldface text we indicate the preconditions for the corresponding transition. With italic text we describe the actions that occur during the transition.

- The processor reads the k-mer's used_flag.
- If the value is UNUSED then the processor can infer that no other processors have reached this k-mer and thus can visit it. It therefore sets that k-mer flag to USED and can use the k-mer's information conflict-free.
- If the value is USED then another processor has already added that k-mer to another subcontig.

Note that these actions need to be executed in an atomic fashion to ensure correctness (detailed in Section VI). Thus the k-mer's used_flag is updated atomically using compare_and_swap() remote atomics to avoid races and redundant work. Furthermore, the k-mer data structure has a pointer my_subcontig_ptr to the subcontig's data structure that it currently belongs to. Finally, we enhance the subcontig data structure with a lock s_lock and a flag state that may take one of the following values: INACTIVE, ACTIVE, ABORTED and COMPLETE.

2) Synchronization Algorithm: The synchronization scheme in our parallel de Bruijn graph traversal is based on the subcontig's state machine illustrated in Figure 7. Note that a processor might conflict with at most two processors due to the one dimensional nature of the subcontigs. Therefore, when we mention neighboring subcontigs of a subcontig S we refer the subcontigs that conceptually lie to the right and to the left of S in the eventual contig. Similarly, when we refer to the neighboring processors of p_i we are referring to the processors that are working on the neighboring subcontigs.

Implicitly each (uninitiated) subcontig is in INACTIVE state. When a processor p_i picks a k-mer as seed, it initiates a subcontig, then a subcontig data structure is created and its state is set to ACTIVE, while the the seed's my_subcontig_ptr is set to point to the newly created subcontig. The processor p_i tries to extend the subcontig in both directions using the previously described procedure. When p_i finds UNUSED k-mers in *either* direction of the subcontig, it successfully adds left/right extension bases to the subcontig and marks the visited k-mers' used_flag field as USED. At the same time, p_i updates the visited k-mers' my_subcontig_ptr field to point to the subcontig's data structure. Meanwhile, the subcontig's

state remains ACTIVE.

Assume that processor p_i is unable to extend its current subcontig in *either* direction because it has found USED k-mers in both directions (or it has reached one endpoint of the eventual contig and has found a USED k-mer in the other direction). Processor p_i then attempts to obtain *all* the s_locks of the neighboring subcontigs including its own s_lock in a *total global order* indicated by the the processors' ids — and thus ensuring that deadlocks are avoided. Processor p_i has access to the neighboring subcontigs' data structures (and consequently to the appropriate s_locks) via the my_subcontig_ptr fields of the USED k-mers found in both directions.

After obtaining these s_locks, p_i examines the state fields of the neighboring subcontigs and takes appropriate actions as indicated by the state machine in Figure 7, specifically:

- If both neighboring subcontigs are ACTIVE, then p_i sets the state of its subcontig to ABORTED, releases the obtained s_locks following the inverse order in which they were obtained and picks another random k-mer seed from the distributed hash table to initiate another subcontig.
- If at least one of the neighboring subcontigs is ABORTED then p_i attaches that subcontig to its own. p_i then sets the state of the "aborted" subcontig to ACTIVE as it is now added to an "active" subcontig. Moreover, the my_subcontig_ptr fields of the k-mers in the previously "aborted" subcontig are fixed to point to the p_i's subcontig data structure (our scheme allows us to fix just one intermediate pointer to avoid revisiting all these k-mers). Finally, p_i releases the obtained s_locks following the inverse order in which they were obtained and continues to extend its subcontig.

A processor working on an ACTIVE subcontig A continues computing while other processors might obtain A's s_lock and examine A's state, hence allowing our synchronization scheme to be lightweight. Additionally, a processor "aborts" its subcontig when there are more processors working on the same contig and they will claim the "aborted" subcontig later, hence avoiding redundant work. Note that the subcontigs' state flag doesn't require atomic update since the locking scheme we describe guarantees that at most one processor will

examine and update a subcontig's flag at any point in time.

When p_i reaches both endpoints of the contig, it sets the state of the subcontig to COMPLETE and stores the contig. Afterwards, p_i picks another random k-mer seed from the distributed hash table and initiates another subcontig. When all k-mers in the hash table have been visited the parallel de Bruijn graph traversal is complete.

C. De Bruijn Graph Traversal Synchronization Cost Analysis

Since the synchronization cost of the parallel de Bruijn graph traversal will likely determine parallel efficiency, we present three analytical propositions to quantify the correlation between the expected subcontig conflicts and the concurrency. Proposition 1 gives a lower bound on the total expected conflicts while Proposition 2 provides an upper bound. Finally Proposition 3 demonstrates the expected number of conflicts incurred on the critical path.

Proposition 1. *Let p be the total processors during parallel execution and n the number of contigs to be assembled during the de Bruijn graph traversal. Assuming that the n contigs have the same length and $n \gg p$, the number $f(p)$ of expected conflicts during the traversal increases at least linearly with p, i.e. $f(p) \geq a \cdot p + b$ for some constants a and b.*

Proof. Initially there are n contigs to be assembled and the p available processors pick random k-mers as seeds to start the traversal. A conflict occurs if two (or more) processors pick an initial seed belonging to the same contig. We can formulate this process as tossing p balls into n bins where the tosses are uniformly at random and independent of each other and thus the probability that a ball falls into any given bin is $1/n$. Let's denote $\mathcal{Q}_{i,t}$ the event where ball i falls into bin t (and thus $\Pr[\mathcal{Q}_{i,t}] = 1/n$) and Φ_{ij} be the event that ball i and ball j collide. By using the Bayes rule we can calculate the probability of any two balls i and j falling into one bin (i.e. balls i and j collide):

$$
\begin{aligned}
\Pr[\Phi_{ij}] &= \sum_{t=1}^{n} \Pr[\mathcal{Q}_{j,t}|\mathcal{Q}_{i,t}]\Pr[\mathcal{Q}_{i,t}] = \sum_{t=1}^{n} \frac{1}{n}\Pr[\mathcal{Q}_{i,t}] \\
&= \frac{1}{n}\sum_{t=1}^{n} Pr[\mathcal{Q}_{i,t}] = \frac{1}{n}\sum_{t=1}^{n} \frac{1}{n} = \frac{1}{n}
\end{aligned}
\tag{1}
$$

where we have used the fact that $\Pr[\mathcal{Q}_{j,t}|\mathcal{Q}_{i,t}] = \Pr[\mathcal{Q}_{j,t}] = 1/n$ since the events $\mathcal{Q}_{j,t}$ and $\mathcal{Q}_{i,t}$ are independent.

Let X_{ij} be the indicator random variable of Φ_{ij}, i.e. $X_{ij} = 1$ if Φ_{ij} happens and $X_{ij} = 0$ otherwise. We can consider X_{ij}s as Bernoulli variables and tosses can be considered as a sequence of Bernoulli trials with a probability $1/n$ of success, i.e. $\Pr[X_{ij} = 1] = \Pr[\Phi_{ij}] = 1/n$. If C is defined as the number of conflicts then $C = \sum_{i \neq j} X_{ij}$. So, we can calculate the expected number of conflicts $\mathbb{E}[C]$ as:

$$
\mathbb{E}[C] = \sum_{i \neq j} \mathbb{E}[X_{ij}] = \sum_{i \neq j} \Pr[X_{ij} = 1] = \frac{1}{n}\binom{p}{2}
\tag{2}
$$

Now, consider the traversal as a sequence of steps, where at each step p contigs are calculated, and the traversal consists

of n/p such steps. In the first step, the expected number of conflicts is given by equation 2: $\mathbb{E}[C_1] = \frac{1}{n}\binom{p}{2}$. We assume that as soon as these conflicts are resolved no more conflicts occur at the same step and thus $\mathbb{E}[C_1]$ is a lower bound of the expected number of conflicts in the first step. In the i-th step, there are $n - (i-1)p$ remaining contigs and we can consider the i-th step as tossing p balls into $n - (i-1)p$ bins, thus we get that $\mathbb{E}[C_i] = \frac{1}{n-(i-1)p}\binom{p}{2}$ where C_i is the number of conflicts at step i. A lower bound on the expected number of conflicts for the entire graph traversal is therefore:

$$
\begin{aligned}
\mathbb{E}[C_{total}] &= \mathbb{E}[C_1] + \mathbb{E}[C_2] + \cdots + \mathbb{E}[C_{n/p}] \\
&= \frac{1}{n}\binom{p}{2} + \frac{1}{n-p}\binom{p}{2} + \cdots + \frac{1}{p}\binom{p}{2} \\
&= \binom{p}{2} \cdot \frac{1}{p} \cdot \left(1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n/p}\right) \\
&= \binom{p}{2} \cdot \frac{1}{p} \cdot H_{n/p} = \frac{p-1}{2} \cdot H_{n/p}
\end{aligned}
\tag{3}
$$

where with H_i we denote the i-th partial sum of the diverging harmonic series. For large values of n/p we can write:

$$
H_{n/p} \simeq \ln(n/p) + \gamma = \ln(n) - \ln(p) + \gamma
\tag{4}
$$

where $\gamma \simeq 0.577$ is the Euler-Mascheroni constant. So, the number $f(p)$ of the expected conflicts as a function of p is:

$$
f(p) = \mathbb{E}[C_{total}] = \frac{p-1}{2} \cdot (\ln(n) - \ln(p) + \gamma)
\tag{5}
$$

We have assumed that $n \gg p$. If $n \geq 100 \cdot p$ then:

$$
\begin{aligned}
\ln(n) &\geq \ln(100 \cdot p) = \ln(100) + \ln(p) \geq 4.6 + \ln(p) \\
&\Leftrightarrow \ln(n) - \ln(p) + \gamma \geq 5.177
\end{aligned}
\tag{6}
$$

By combining equation 5 and inequality 6 we get that:

$$
\begin{aligned}
f(p) &= \frac{p-1}{2} \cdot (\ln(n) - \ln(p) + \gamma) \geq \frac{p-1}{2} \cdot 5.177 \\
&\Rightarrow f(p) \geq 2 \cdot p - 2
\end{aligned}
\tag{7}
$$

which completes the proof by setting $a = 2$ and $b = -2$. $\square$

Proposition 2. *Let G be a de Bruijn graph and assume that the result of its traversal consists of a single contig L. Let C_G be the number of conflicts in this traversal. Now assume that some nodes are removed from G resulting in a graph G'. The traversal of G' results in $C_{G'}$ conflicts where $C_{G'} \leq C_G$.*

Proof. By removing a number of nodes from G, the traversal of the new graph G' results in a set of contigs S that span part of the single contig L. More accurately, the only places that the combination of S's members differs from L are those ones that correspond to the removed nodes (k-mers). Now assume a seeding function that incurs C_G conflicts in G. The same seeding function creates at most C_G conflicts in G' because the removed nodes only can prevent a conflict (since a removed node divides a contig in two independent subcontigs). Therefore we get: $C_{G'} \leq C_G$. $\square$

The result of Proposition 2 suggests that the worst case for the number of conflicts in a traversal is one in which the

resulting assembly is a single contig and thus C_G is an upper bound in the number of conflicts. Now, let's assume that the seeds are picked uniformly distanced across the single contig L and that p processors are performing the traversal of G in parallel. Such a traversal of G results in $O(p)$ conflicts. By using the latter fact and the result of Proposition 2 we conclude that an upper bound for the total number of conflicts during a traversal with seeding function resulting in uniformly distanced seeds is $O(p)$. Also, Proposition 1 indicates that the total number of conflicts is $\Omega(p)$. Therefore, under the assumptions made in Propositions 1 and 2, the total number of expected conflicts is $\Theta(p)$.

Let T_{serial} be the serial time for de Bruijn graph traversal, p the number of total processors, $\mathcal{C}_{crit}(p)$ the expected number of conflicts incurred on the critical path and t_{confl} the effective time for resolving a conflict. Then the time for the parallel traversal $T(p)$ can be modeled roughly as:

$$T(p) = \frac{T_{serial}}{p} + \mathcal{C}_{crit}(p) \cdot t_{confl} \qquad (8)$$

The first term of Equation 8 decreases linearly with p and is the useful work, while the second term is the synchronization overhead of the parallel algorithm, and $\mathcal{C}_{crit}(p)$ depends on the total number of conflicts $\mathcal{C}_{total}(p)$ and p. We showed that under some assumptions $\mathcal{C}_{total}(p) = \Theta(p)$ and thus a synchronization scheme that merely leaves $\Theta(p)$ conflicts to be resolved on the critical path is unacceptable because then the second term of equation 8 would increase linearly with p. The synchronization algorithm at subsection V-B is lightweight in a sense that as soon as a conflict is detected, one of the involved processors takes actions according to the state machine and does not prevent others from doing useful work. Proposition 3 shows that the expected number of the conflicts incurred on the critical path is $\mathcal{C}_{crit}(p) = 2 \cdot \mathcal{C}_{total}(p)/p$. The scalability performance results in Section VII confirm that our synchronization overhead does not become a bottleneck even at high concurrency.

Proposition 3. *Let $\mathcal{T}$ be the set of conflicts during the parallel de Bruijn graph traversal with cardinality $|\mathcal{T}| = \mathcal{C}_{total}(p)$. Assuming that the contigs have the same length, the expected number of the conflicts incurred along the critical path $\mathcal{C}_{crit}(p)$ is equal to $2 \cdot \mathcal{C}_{total}(p)/p$.*

Proof. We will use the same formulation of the problem as in Proposition 1. Additionally consider a collision in a particular bin t during a step of the algorithm where there are R bins (equivalently consider a conflict in a particular contig t during a step of the traversal where there are R remaining contigs to be assembled). Let $\mathcal{Q}_{i,t}$ be the same as in Proposition 1, Z_t the indicator variable of the event "there is a collision in bin t" and Φ_{ij}^t the event that balls i and j fall in bin t. Since the events $\mathcal{Q}_{i,t}$ are independent we can write:

$$\Pr[\Phi_{ij}^t] = \Pr[\mathcal{Q}_{i,t} \cap \mathcal{Q}_{j,t}] = \Pr[\mathcal{Q}_{i,t}]\Pr[\mathcal{Q}_{j,t}] = \frac{1}{R}\frac{1}{R} = \frac{1}{R^2} \qquad (9)$$

Using the Bayes theorem we get:

$$\Pr[\Phi_{ij}^t \mid Z_t = 1] = \frac{\Pr[Z_t = 1 \mid \Phi_{ij}^t] \cdot \Pr[\Phi_{ij}^t]}{\Pr[Z_t = 1]}$$
$$\overset{(9)}{=} \frac{\Pr[Z_t = 1 \mid \Phi_{ij}^t] \cdot 1/R^2}{\Pr[Z_t = 1]} = \frac{1/R^2}{\Pr[Z_t = 1]} \qquad (10)$$

since $\Pr[Z_t = 1 \mid \Phi_{ij}^t] = 1$ (i.e. given that balls i and j fall into bin t, $Z_t = 1$ with probability 1 since there is a collision in that bin). Equation 10 dictates that given a collision t, all pairs of balls (i,j) are equally probable to incur that collision t. Thus, if we denote with C_{ij}^t the event where the pair of balls (i,j) incurs a particular collision t, we can write: $\Pr[C_{ij}^t] = \Pr[C_{lm}^t]$ for any pairs (i,j) and (l,m). The total number of pairs of balls is $\binom{p}{2}$ and therefore:

$$\Pr[C_{ij}^t] = \frac{1}{\binom{p}{2}} = \frac{2}{p(p-1)} \qquad (11)$$

Finally, if we denote with $\mathcal{I}_i^t$ the event where ball i incurs the collision t (or equivalently a processor i incurs the conflict t) we can write:

$$\mathcal{I}_i^t = \bigcup_{\substack{j=1 \\ j \neq i}}^{p} C_{ij}^t \qquad (12)$$

where the events C_{ij}^t with i, t fixed and $j \in \{1, ..., p\} - \{i\}$ are mutually exclusive. Therefore:

$$\Pr[\mathcal{I}_i^t] = \Pr[\bigcup_{\substack{j=1 \\ j \neq i}}^{p} C_{ij}^t] = \sum_{\substack{j=1 \\ j \neq i}}^{p} \Pr[C_{ij}^t] \overset{(11)}{=} \sum_{\substack{j=1 \\ j \neq i}}^{p} \frac{2}{p(p-1)}$$
$$= \frac{2}{p(p-1)} \sum_{\substack{j=1 \\ j \neq i}}^{p} 1 = \frac{2}{p(p-1)} \cdot (p-1) = \frac{2}{p} \qquad (13)$$

If we denote with $\mathcal{F}_i$ the number of conflicts that a processor i incurs on the critical path we get:

$$\mathbb{E}[\mathcal{F}_i] = \sum_{t \in \mathcal{T}} \Pr[\mathcal{I}_i^t] \overset{(13)}{=} \sum_{t \in \mathcal{T}} \frac{2}{p} = \frac{2}{p} \sum_{t \in \mathcal{T}} 1 = \frac{2}{p}\mathcal{C}_{total}(p) \quad (14)$$

Equation 14 indicates that the expected number of the conflicts incurred along the critical path is equal to $2 \cdot \mathcal{C}_{total}(p)/p$, which completes the proof. $\qquad \square$

VI. IMPLEMENTATION DETAILS

UPC is a PGAS language that combines single program multiple data (SPMD) programming model with a global address space. A fixed number of UPC threads are created at program startup and every thread runs the same program. Each UPC thread has both private local memory and a partition of the shared space. An object living in the private local memory can be accessed only by its owner-thread, while all threads can access/modify objects in the shared address space. We primarily use the portable, high-performance Berkeley UPC compiler [14]. In particular, the contig generation step is implemented in Berkeley UPC. The k-mer analysis is

implemented using C++ with MPI (due to MPI-IO's integration with HDF5 and the presence of a highly-tuned ALLTOALLV primitive) and linked with the rest of the application thanks to the interoperability of UPC.

Berkeley UPC also has one-sided communication capabilities. A major attribute of this communication model is that information is never required from the remote user code to complete a communication operation. This is extremely beneficial in the context of the parallel de Bruijn graph construction and traversal where the accesses in the distributed hash table are essentially random. Thus, when a processor needs to access a remote bucket in the distributed hash table, no coordination is required in the code with the owner of the remote bucket. This simplifies programming compared with a two-sided communication model. The result is a portable implementation that can be executed on both shared and distributed memory machines without any change, as will be demonstrated in Section VII.

Memory Management: We now describe the mechanism that allows us to implement the "aggregating stores" communication optimization. When processor p_i stores S entries to the local-shared stack of p_j, it needs to locate the position in p_j's stack that these entries should go to. Thus, every local-shared stack is associated with its `stack_ptr` pointer that indicates the current position in the local-shared stack. These `stack_ptr` variables are shared and accessible to all processors. Therefore, if processor p_i is about to store S entries to processor p_j, it (a) reads the current value of p_j's `stack_ptr`, called `cur_pos`, (b) increases the value of p_j's `stack_ptr` by S and (c) stores the S entries in p_j's local-shared stack into the locations `cur_pos` $\cdots$ `cur_pos+S-1` with an aggregate transfer. Steps (a) and (b) need to be executed atomically to avoid data hazards, for which we leverage global atomics `atomic_fetchadd()` provided by Berkeley UPC.

Atomic Flag Updates: In order to ensure atomicity in the parallel de Bruijn graph traversal we utilize a lock-free approach that is based on Berkeley UPC atomics. In particular, to atomically read and update a k-mer's `used_flag` (as it is described in Section V-B), we make use of the `compare_and_swap()` global atomic. Thus, if the `used_flag` field is already USED, then the atomic returns the value USED. Conversely, if the `used_flag` field is UNUSED the atomic sets that field to USED and returns the value UNUSED.

DNA Sequence Compression: Given that the vocabulary in a DNA sequence is the set $\{A, C, G, T\}$ only two-bits per base are required for a binary representation. We thus implement a high-performance compression library that compresses the DNA sequences from text format into a binary format. This approach reduces the memory footprint by $4\times$, while additionally reducing the bandwidth by $4\times$ for communication events that involve k-mers or DNA sequences transfers. Finally we exploit the complementary nature of DNA by not explicitly storing both (redundant) strands. Instead we process one of its strands and can recover the reverse complement in a straightforward way. This design decision complicates the code but offers an extra $2\times$ saving in memory requirements.

VII. EXPERIMENTAL RESULTS

A. Experimental Testbed

High-concurrency experiments are conducted on Edison, a Cray XC30 supercomputer at NERSC. Edison has a peak performance of 2.57 petaflops/sec, with 5576 compute nodes, each equipped with 64 GB RAM and two 12-core 2.4 GHz Intel Ivy Bridge processors for a total of 133,824 compute cores, and interconnected with the Cray Aries network using a Dragonfly topology. We also explore shared-memory using a single 1 TB node of the NERSC Carver system, containing four 8-core 2.0 GHz Intel Nehalem processors.

Our experiments are conducted on real data sets for human and wheat genomes. The 3.2 billion base pair (gigabase, Gb) human genome is assembled from 2.5 billion reads (252 Gbp of sequence) for a member of the CEU HapMap population (identifier NA12878) sequenced by the Broad Institute. The reads are 101 bp in length from a paired-end insert library with mean insert size 238 bp. The 17 Gbp hexaploid wheat genome (*Triticum aestivum L.*), is assembled from 2.3 billion reads (477 Gbp of sequence) for the homozygous bread wheat line 'Synthetic W7984' sequenced by the JGI. The reads are 100-250 bp in length from 5 paired-end libraries with insert size 240-740 bp. This important genome was only recently sequenced for the first time, due to its size and complexity.

B. Parallel k-mer Analysis

The k-mer analysis step of the original Meraculous had 3459 GB overall memory footprint. By contrast, our optimized k-mer analysis implementation only required 499 GB of RAM for the 240 cores run on human genome, a reduction of $6.93\times$. Our code achieves a rate of over 3 billion k-mers analyzed per second for our largest run (15K cores) on the human data, including the I/O cost. Our maximum attained aggregate I/O performance is 18.5 GB/s, out of the theoretical peak of 48 GB/s of Edison. The ability to perform this step so fast allowed us to do previously infeasible exploratory analyses to optimize the assembly as a function of k.

The strong scaling results for the human and wheat data are shown in Figure 8 (left) and (right), respectively. Our algorithm scales efficiently for the human data, all the way from 240 cores to over 15K cores, for both communication and local computation phases. The relatively poor scaling of communication for the wheat is because wheat data is significantly more skewed with about 60 k-mers occurring over 10 million times, and only 403 k-mers occurring over 1 million times each. This creates load imbalance in the ALLTOALLV phase as the receiving processors of those extreme frequency k-mers spend relatively longer time in the collective call. Knowing which k-mers occur in extreme counts would help us solve this problem by treating them specially. Unfortunately, this is partially a chicken-and-egg problem since our goal is to count k-mer frequencies in the first place.
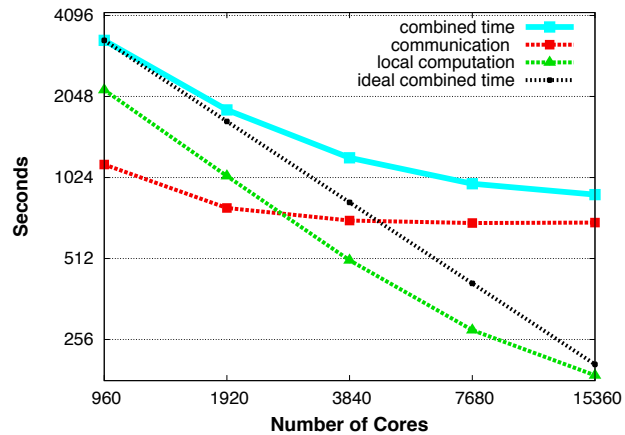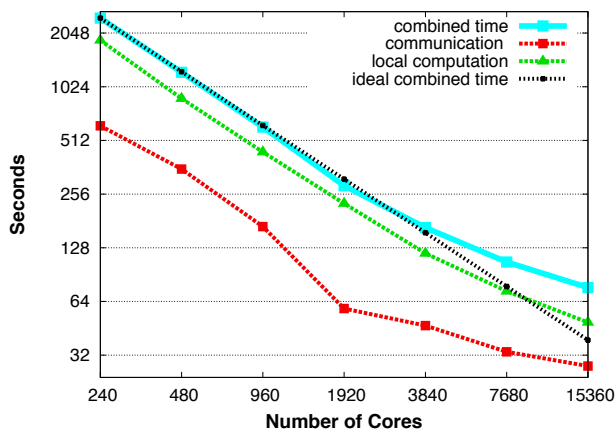
Fig. 8: Strong scaling of the k-mer analysis on Edison for human (left) and wheat (right) genome. Both axes are in log scale. Combined time corresponds to the sum of communication and computation times of the k-mer analysis.
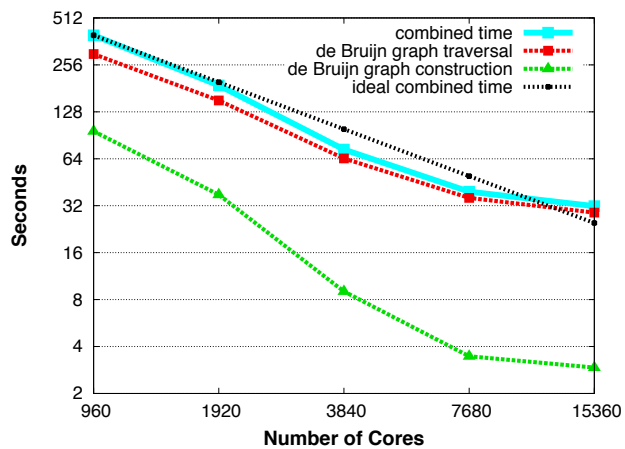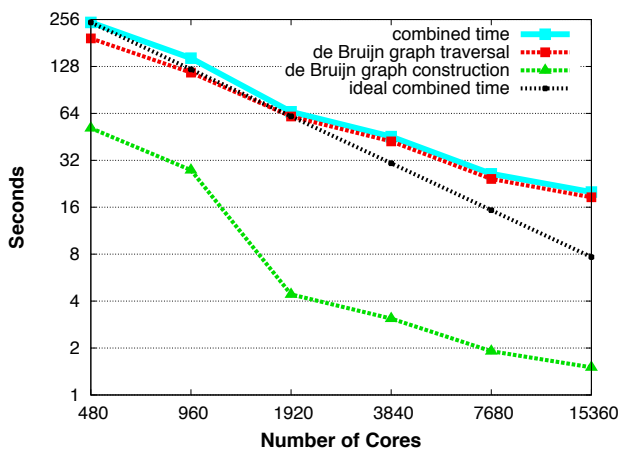

Fig. 9: Strong scaling of the contig generation on Edison for human (left) and wheat (right) genome. Both axes are in log scale. Combined time corresponds to the sum of de Bruijn graph construction and traversal times.

C. Parallel De Bruijn Graph Construction and Traversal

Figure 9 (left) exhibits the strong scaling behavior of our de Bruijn graph construction and traversal using the human data set. We were not able to run this step for less than 480 cores, due to memory limitations. Our implementation shows efficient scaling for both constructing and traversing the de Bruijn graph, attaining a $12.24\times$ relative speedup when increasing concurrency from 480 to 15,360 cores. We highlight that due to our optimized parallelization approach, our 15,360 core execution completes the contig generation for the human genome in ~ 20 *seconds*, compared with the original Meraculous serial code that required ~ 44 *hours* on a 128 GB shared memory node. Figure 9 (right) presents the strong scaling results for the wheat data set. The starting point for the graph is 960 processors, since the memory requirements are significantly larger than the human dataset. We achieve $12.5\times$ relative speedup when increasing concurrency from 960 to 15,360 cores. For this grand-challenge genome, our implementation required only ~ 32 *seconds* on 15,360 cores, whereas single-node memory constraints prevent the original Meraculous version from running this problem. We extrapolate that if enough memory was available, the original Meraculous

code would have required 121 *hours*.

In both data sets, the superlinear speedup in the graph construction can be explained by the *aggregating stores* optimization described in Section IV. As we present strong scaling results, each processor stores fewer k-mers in its hash table as p grows; at the same time, our communication buffers are a constant size S, so fewer communication rounds are needed with larger processor counts. At very large values of p, some buffers may never fill and will therefore be sent using small data transfers; this can also produce faster communication in the underlying communication protocols.

Additionally, for each strong scaling experiment we measured the fraction of time spent to add forward/backward extensions at the subcontigs, because this is the useful workload and the fraction of time spent in the synchronization protocol along the critical path. We observed that although the relative time spent in synchronization mode increases for higher concurrency levels, the majority of the time is always consumed in doing useful work. This empirical behavior validates our analysis in Section V-C and demonstrates that our synchronization scheme is lightweight, enabling our parallel algorithm to scale up to tens of thousands of cores.

D. Shared Memory Results on Carver

Table I presents the performance of our parallel algorithms on the large shared-memory Carver node for the human genome. Again we attain near perfect scaling between 16- and 32-cores ($1.94\times$ relative speedup). The UPC implementation is identical for both our shared- and distributed-memory experiments, highlighting that our optimized code is both portable and highly efficient. Note that the wheat genome exceeded the large memory capacity of Carver, emphasizing the importance of the UPC distributed memory capability.

E. Strong Scaling on Small E. coli Dataset

To assess the performance of our work on a small dataset, we experimented on the 4.64 Mbp *E. coli* K-12 MG1655. Serial execution for k-mer analysis and contig generation on Edison required 1281.4 seconds, while at 96 cores the runtime was reduced by a factor of $92.8\times$ to 13.8 seconds.

F. Comparison with De Novo Parallel Assemblers

To assess the performance of our work relative to other parallel de novo genome assemblers we evaluated the MPI-based Ray [15] (version 2.3.0) and ABySS [16] (version 1.3.6) on Edison. In particular, we identified the steps in each implementation that generate contigs that are analogous to our "UU contigs", starting from the input reads. In Ray, these are "seeds"; in ABySS these are "unitigs". Results show that our approach generates contigs from raw reads (human dataset) in 753 seconds on 960 cores, approximately $27.9\times$ and $64.3\times$ faster than Ray and ABySS respectively. We emphasize that both Ray and ABySS spend significant time in I/O, and suspect that this component is not scaling efficiently as it is lacking an input format amenable to parallelization such as SeqDB. Ray spent 15,760 seconds for I/O and 5,292 seconds for k-mer analysis and contig generation. On the other hand, our algorithm spent 17 seconds for I/O ($924\times$ faster than Ray) and 735 seconds for k-mer analysis and contig generation ($7\times$ faster than Ray). Although our presentation here emphasizes the computational efficiency of the algorithms, we also note that Meraculous performs better, in terms of accuracy, completeness and contiguity/scaffolding, than these other parallel assemblers on test datasets.

G. Improvement in End-to-End Assembly Times

We incorporated our work into the Meraculous assembler by replacing the k-mer analysis and contig generation steps and keeping the original scaffolding step. The result is an end-to-end improvement in Meraculous assembly times of $6.6\times$ for Wheat (from 165.4 hours to 25.06 hours) and $5\times$ for Human (from 75.6 hours to 15.15 hours) when we used 15,360 cores.

Cores	K-mer Analysis	Graph Construction	Graph traversal	Total
16	57487 s	399 s	685 s	58571 s
32	29590 s	202 s	393 s	30185 s

TABLE I: Performance results (in seconds) on Carver for human genome. Wheat data exceeds the 1TB memory capacity of the shared-memory node.

VIII. Related Work

A thorough survey of de novo genome assembly is beyond the scope of our work and we refer the reader to Miller et al. [17]. We primarily refer to parallel assemblers in this section. Ray [15] is a parallel de novo genome assembler that uses its custom distributed compute engine and is built on top of MPI for communication. ABySS [16] is another MPI-based parallel assembler that leverages de Bruijn graphs and implements them via distributed hash table. Neither of these publications describe their parallel algorithm implementations in detail. As shown in Section VII-F our approach outperformed these contig generation steps by $27.9\times$ and $64.3\times$ (respectively) on 960 Edison cores.

PASHA [18] is also de Bruijn-based parallel assembler that utilizes MPI. PASHA is not optimal in its de Bruijn graph traversal since redundant work may be done in the walks; its algorithm introduces a serialization bottleneck and the results do not exhibit good scaling (however PASHA claimed to be $2.25\times$ faster than ABySS). YAGA [19] is a parallel distributed-memory de Bruijn based assembler that is shown to be scalable except for its I/O. Different from YAGA, we refrain from using parallel sorting as it can be an impediment to scaling, especially for the very large genomes examined in our study. In addition, our efficient parallel HDF5 based I/O allows us to achieve end-to-end performance scalability.

Shared-memory assemblers include Pasqual [20] (OpenMP parallelization), which is based on the overlap-layout-consensus approach, while SOAPdenovo [21] (pthreads parallelization) and Velvet [22] (OpenMP parallelization) are based on the de Bruijn graph approach. Jellyfish [23] is a shared-memory k-mer counting software. Khmer [24] is a pre-publication software for k-mer counting that uses Bloom filters but is not distributed. These approaches are more restrictive as they are limited by the concurrency and memory capacity of the shared-memory node. Kmernator [25] is the only distributed k-mer counter but it consumes more memory than our approach since it does not use Bloom filters.

IX. Conclusions and Future Work

We developed and implemented new parallel algorithms for the core of a state-of-the-art de novo genome assembler, demonstrating remarkable scaling up to 15K cores on a human genome and on the more challenging wheat genome. We presented new parallel approaches to successfully attain unprecedented scaling and runtime efficiency for the de Bruijn graph construction & traversal — despite the irregular computational challenges at the heart of this algorithm. To enable programming productivity, high performance, as well as portability across both distributed- and shared-memory platforms, we leveraged UPC's partitioned global address space design for the de Bruijn graph construction & traversal components. We plan to release a complete assembly pipeline for general use once the remaining steps of assembly are adapted.

Recent developments in biomedical sequencing allow a single genomics facility to sequence 18,000 human genomes/year (50/day) at a cost of $\sim$ \$1,000 per genome, with applications to

the genetics of rare and common disease. Cancer is a particularly appealing application, as the disease arises fundamentally from genomic defects in tumor cells, and the identification and characterization of these mutations can guide treatment. This flood of genomic data promises to transform and personalize medicine, but tools for analyzing this data are lagging behind the capacity to generate this data. We estimate that the parallelized version of Meraculous will be able to assemble 50 genomes/day on 960 processors (a modest 40 nodes). Using parallellized Meraculous, the world's biomedical sequencing capacity could be assembled using a fraction of the Edison supercomputer.

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. R. Bradnam, J. N. Fass, A. Alexandrov, P. Baranay, M. Bechner *et al.*, "Assemblathon 2: evaluating de novo methods of genome assembly in three vertebrate species," *GigaScience*, vol. 2, no. 1, pp. 1–31, 2013.

[3] E. W. Myers, G. G. Sutton, A. L. Delcher *et al.*, "A whole-genome assembly of drosophila," *Science*, vol. 287, no. 5461, pp. 2196–2204, 2000.

[4] P. A. Pevzner, H. Tang, and M. S. Waterman, "An Eulerian path approach to DNA fragment assembly," *Proceedings of the National Academy of Sciences*, vol. 98, no. 17, pp. 9748–9753, 2001.

[5] 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.

[6] A. Appleby, "Murmurhash," 2011.

[7] D. A. Bader, D. R. Helman, and J. JáJá, "Practical parallel algorithms for personalized communication and integer sorting," *Journal of Experimental Algorithmics (JEA)*, vol. 1, p. 3, 1996.

[8] P. Melsted and J. K. Pritchard, "Efficient counting of k-mers in DNA sequences using a bloom filter," *BMC bioinformatics*, vol. 12, no. 1, p. 333, 2011.

[9] 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.

[10] P. Flajolet, E. Fusy, O. Gandouet, and F. Meunier, "Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm," *DMTCS Proceedings*, 2008.

[11] SAM Format Specification Working Group *et al.*, "The sam format specification (v1. 4-r985)."

[12] 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.

[13] T. H. Group, "Hierarchical data format version 5, 2000-2013." [Online]. Available: http://www.hdfgroup.org/HDF5

[14] P. Husbands, C. Iancu, and K. Yelick, "A performance analysis of the Berkeley UPC compiler," in *Proc. of International Conference on Supercomputing*, ser. ICS '03. New York, NY, USA: ACM, 2003, pp. 63–73.

[15] 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.

[16] 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.

[17] J. R. Miller, S. Koren, and G. Sutton, "Assembly algorithms for next-generation sequencing data," *Genomics*, vol. 95, no. 6, pp. 315–327, 2010.

[18] 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.

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

[20] X. Liu, P. R. Pande, H. Meyerhenke, and D. A. Bader, "Pasqual: parallel techniques for next generation genome sequence assembly," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 5, pp. 977–986, 2013.

[21] 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.

[22] D. R. Zerbino and E. Birney, "Velvet: algorithms for de novo short read assembly using de Bruijn graphs," *Genome research*, vol. 18, no. 5, pp. 821–829, 2008.

[23] G. Marçais and C. Kingsford, "A fast, lock-free approach for efficient parallel counting of occurrences of k-mers," *Bioinformatics*, vol. 27, no. 6, pp. 764–770, 2011.

[24] M. R. Crusoe, G. Edvenson, J. Fish, A. Howe, L. Irber *et al.*, "khmer k-mer counting & filtering FTW," https://github.com/ctb/khmer, 2014.

[25] R. Egan, "Kmernator: An MPI toolkit for large scale genomic analysis," https://github.com/JGI-Bioinformatics/Kmernator, 2014.