merAligner: A Fully Parallel Sequence Aligner

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*—Aligning a set of query sequences to a set of target sequences is an important task in bioinformatics. In this work we present merAligner, a highly parallel sequence aligner that implements a seed–and–extend algorithm and employs parallelism in all of its components. MerAligner relies on a high performance distributed hash table (seed index) and uses one-sided communication capabilities of the Unified Parallel C to facilitate a fine-grained parallelism. We leverage communication optimizations at the construction of the distributed hash table and software caching schemes to reduce communication during the aligning phase. Additionally, merAligner preprocesses the target sequences to extract properties enabling exact sequence matching with minimal communication. Finally, we efficiently parallelize the I/O intensive phases and implement an effective load balancing scheme. Results show that merAligner exhibits efficient scaling up to thousands of cores on a Cray XC30 supercomputer using real human and wheat genome data while significantly outperforming existing parallel alignment tools.**

I. Introduction

Recent advances in sequencing technology have made the redundant sampling of genomes extremely cost effective. Such a sampling consists mostly of short reads with low error rates that can generally be aligned to a reference genome in a straightforward way. However, the increasing depth of coverage makes the alignment of the reads to a reference sequence a computationally expensive task, requiring high degrees of parallelism for efficient execution.

The community has therefore developed several approaches for parallelizing the alignment of multiple reads (*queries*) to a set of reference sequences (*targets*). Such a class of mapping methods include the seed–and–extend algorithms (e.g. BLAST [1]). In this paradigm, the reference sequences are first indexed by constructing a *seed index*, and this data structure is then used to locate candidate query–to–target alignments by extracting seeds from the queries and performing seed index lookups. Finally, an extension algorithm is applied to extend a found seed, where a local alignment is returned as the result.

In some applications of this methodology, the reference genome is known a priori, thus allowing an off-line seed index construction that can then be exploited for multiple read data sets. This scenario allows for straightforward parallelization, where the seed index is replicated across a set of computational nodes, which can then independently and concurrently align their subset of the reads. Indeed, there are existing frameworks (e.g. pMap [2]) that automate the process of (1) index replication, (2) distribution of reads across nodes and (3) local alignment computation.

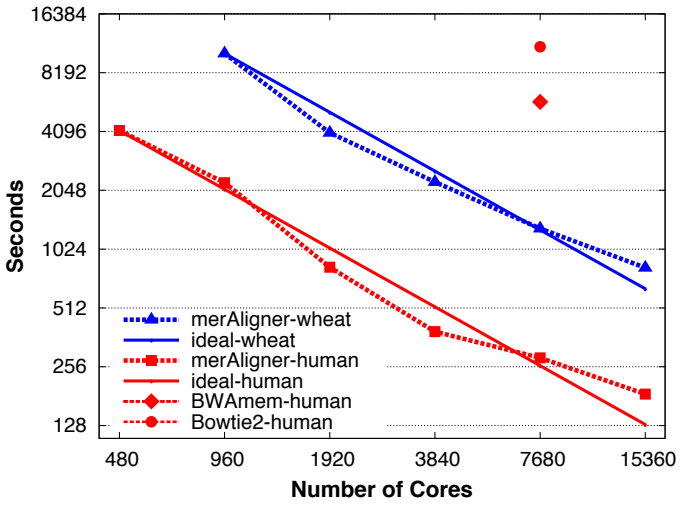

Fig. 1: End-to-end strong scaling of merAligner on Cray XC30 for the human and the wheat genome. The plotted curves exhibit the performance of merAligner while the single data points show the performance of BWA-mem and Bowtie2 used in pMap parallel framework.

However, this approach can suffer from two major limitations. First, the seed index of very large genomes (e.g. wheat [3], pine [4]) may exceed the memory capacity of a singe node, thereby preventing the use of a simple seed index replication scheme. More significantly, there are important applications where the reference sequence is not known ahead of time, thus obviating the off-line approach, and requiring a high-performance implementation of the seed index construction phase to ensure efficient execution of the end-to-end parallel alignment algorithm. A well-known example of this requirement is present in most de novo genome assembly pipelines.

De novo genome assemblers reconstruct an *unknown* genome from a collection of short reads. Typically these assemblers first process the input reads and generate *contigs*, which are genome sequences significantly longer than the input reads. Next, contigs are oriented and gaps are closed during the *scaffolding* phase. The key first stage of the general scaffolding algorithm is aligning the reads onto the generated contigs [5]. Thus, parallel de novo genome assemblers rely on efficient aligner algorithms, where the seed index construction must be efficiently parallelized and distributed to allow high concurrency solutions for grand-challenge genomes.

The parallel alignment work presented in this paper is mo-

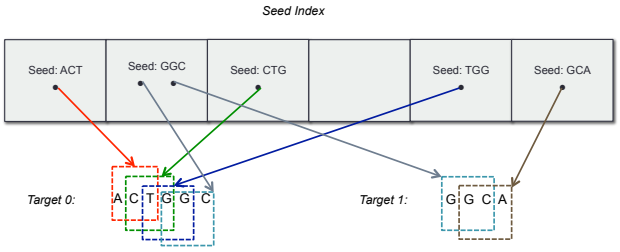

Fig. 2: An example of a seed index data structure that indexes two target sequences. Note that the seed index is distributed and stored in shared memory. Also, the target sequences are stored in shared memory such that any processor can access them. Here the seed GGC is extracted from both target sequences, thus the value in the corresponding hash table entry is a list of pointers to the corresponding sequences. For simplicity we do not show here additional stored information, such as the seed's offsets in the targets.

tivated by our ongoing effort to parallelize the Meraculous [6] genome assembly pipeline [7]. Our study makes numerous contributions including:

- A highly optimized sequence alignment algorithm that parallelizes all its components from end to end, including I/O and seed index construction.
- Scalable seed index implementation that leverages software caching to extend our high-performance *lock-free* and communication-optimized distributed hash table.
- An efficient method to preprocess the target sequences that enables exact sequence matching with minimal communication and computation without sacrificing accuracy.
- Techniques to efficiently parallelize the I/O intensive stages and address load imbalance via randomization.
- Close-to-ideal scaling (with 0.7 - 0.78 parallel efficiency) up to 15K cores on NERSC's Cray XC30 supercomputer, using real data sets from the human and the grand-challenge wheat genomes.
- Comparisons with existing alignment solutions, showing the significant advantage of our end-to-end parallel approach.

Overall, our work shows that efficient utilization of distributed memory architectures enables effective parallelization of sequence alignment in terms of both high scalability and reduced per-node memory requirements. An overview of our efficient end-to-end scalability with performance comparisons to BWAmem [8] and Bowtie2 [9] can be seen in Figure 1; detailed performance analysis is presented in Section VI.

II. The merAligner algorithm

Algorithm 1 describes the parallel algorithm we employ to align a set of query sequences (reads) to a set of target sequences. We choose Unified Parallel C (UPC) [10] for our implementation to reduce programming complexity of global data structures and facilitate a fine-grained parallelism.

A. Extracting Seeds from Target Sequences

First, each processor p_i reads a distinct portion of the target sequences (line 4) and stores them in shared memory such that any other processor can access them. Every target sequence of length L contains $L-k+1$ distinct seeds of length k. The first bases $1\ldots k$ of a target form the first seed, the bases $2\ldots k+1$ form the second seed, etc. We extract seeds from the target sequences and associate with every seed the target from which it was extracted (line 5) – we also keep track of the exact offset of the seed in the target. Note that a given seed s might appear in two or more target sequences.

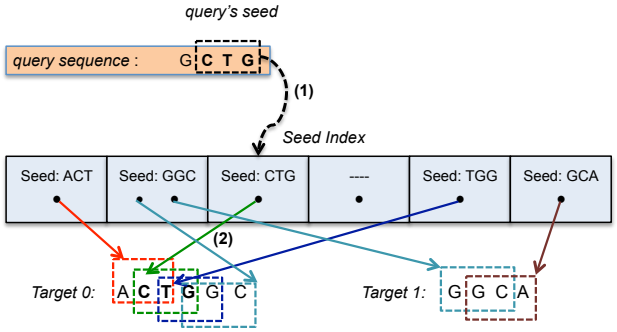

Fig. 3: Locating query-to-target candidate alignments. First the processor extracts a seed from the query sequence (CTG seed). Next the processor looks up the distributed seed index (arrow 1) and finds that a candidate target sequence is Target 0 (arrow 2). Finally, the Smith-Waterman algorithm is executed using as inputs the query and the Target 0 sequences.

B. Indexing Target Sequences

Once the seeds are extracted from the target sequences, they are stored in a global hash table, henceforth referred to as the *seed index* (line 6), where the key is a *seed* and the value is a pointer to the target sequence from which this *seed* has been extracted. If a *seed* is extracted from multiple target sequences, its value in the hash table is a list of pointers to those targets. The seed index is distributed and stored in global shared memory such that any processor can access and lookup any seed. Essentially the seed index data structure provides a mapping from *seed* to *targetSequences* (see Figure 2).

C. Locating Query-to-Target Candidate Alignments

Given a seed s from a query sequence q and an index $seedIndex$, we perform a lookup and locate the candidate target sequences that have $length(s)$ consecutive bases matching with q (line 10). Thus, each one of the query-to-target candidate alignments can be located in $O(1)$ time (see Figure 3).

D. Identifying Alignments via Smith-Waterman

Finally, after locating a candidate target sequence t that has $length(s)$ consecutive bases matching with a query sequence

Algorithm 1 Parallel sequence alignment

```
1:  Input: A set of queries and and a set of targets
2:  Output: Alignments of queries with targets
3:  for all processors p_i in parallel do
4:      targetSeqs ←ReadTargetSequences(targets)
5:      seedsInTargets ←ExtractSeeds(targetSeqs)
6:      seedIndex ←BuildGlobalSeedIndex(seedsInTargets)
7:      myQuerySequences ←ReadQuerySequences(queries)
8:      for each query sequence q ∈ myQuerySequences do
9:          for each seed s ∈ q do
10:             candidateTargets ←LookUp(seedIndex, s)
11:             for each target t ∈ candidateTargets do
12:                 alignmentsSet ←SmithWaterman(t, q)
```

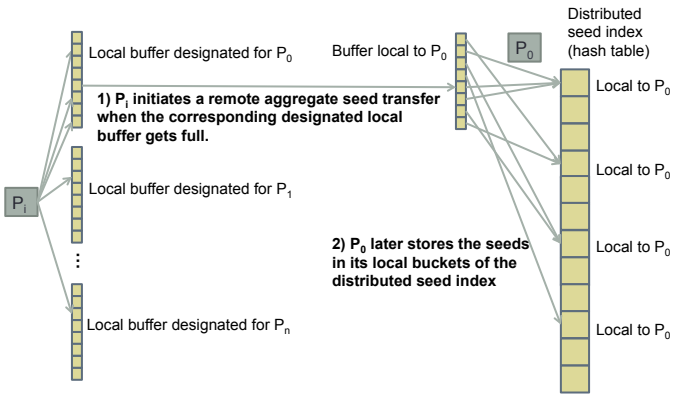

Fig. 4: Communication optimization for the distributed seed index (hash table) construction. In this example, processor p_i performs one remote aggregate transfer to processor p_0 when the local buffer for p_0 gets full. p_0 will store these seeds in its local buckets later by iterating over its local-shared stack.

q (where s is a common seed in both sequences), the Smith-Waterman [11] algorithm is executed with input the sequences t and q in order to perform local sequence alignment (line 12).

III. Distributed Seed Index Optimizations

In order to make this paper self-contained we describe our distributed hash table implementation previously used for contig generation [7], and demonstrate that this idea can be extended to the alignment problem.

A. Distributed Seed Index Construction

A straightforward algorithm for constructing the distributed seed index would process each seed s that a processor p_i encounters in its target sequences by hashing s and invoking a (potentially) remote-node access to the distributed hash table (seed index), in order to store that entry to the appropriate location. Unfortunately this approach suffers from both fine-grained communication and fine-grained locking, necessary to ensure atomic accesses to the buckets. To address this performance deficiency, we mitigate fine-grained communication overhead by leveraging a communication optimization called *aggregating stores*, shown in Figure 4. Here, a processor p_i has $n-1$ local buffers corresponding to the other $n-1$ remote processors, where the size S of each local buffer is a tuning parameter. Every processor hashes one if its targets' seed s and calculates the location in the hash table where s 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 seed entry to the appropriate *local* buffer.

In our implementation, when a local dedicated buffer for processor p_j becomes full, a remote aggregate transfer is initiated to processor p_j. A processor p_j has a pre-allocated *local-shared stack* shared space, where other processors store seed entries destined for that processor. Once all target seeds are computed, each processor iterates over its local-shared stack and stores each seed entry to the appropriate *local* bucket in the distributed hash table, without any communication. Thus, the optimization trades an $S \times (n-1)$ memory

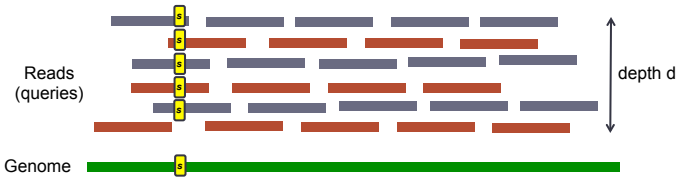

Fig. 5: A genome sampled at some depth of coverage d.

increase for an S-fold reduction in the number of messages relative to the unoptimized approach. At the same time, our optimization solves the problem of fine-grained locking. Since each processor iterates over its local-shared stack and stores the received seeds in the appropriate *local* buckets of the hash table, there in no need for locks, thus allowing the resulting distributed hash table to be *lock free*.

We manipulate local-shared stacks atomically in order to ensure that processors p_i and p_k trying to store seed entries simultaneously at the local-shared stack of another processor p_j do not overwrite the same locations. 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 use the global atomic `atomic_fetchadd()`. Without this aggregating optimization, one would have to access a single remote bucket at a time and consequently would have to obtain one lock at a time. On the other hand, this optimization reduces the number of atomic operations by a factor of S and highlights the advantage of being lock-free.

B. Software Caching Schemes

The nature of the alignment problem enables data reuse in both the seeds and target sequences, allowing us to exploit this insight for a more efficient implementation. High throughput sequencing allows genomes to be sampled redundantly at a depth d, as visualized in Figure 5. Let k be the length of a seed s and L be the read length. Any seed s of the genome (yellow region) is expected to be found $f = d \cdot (1-(k-1)/L)$ times in the read data set (f is the mean of the Poisson distribution of key-frequencies [12]), thus resulting in f lookups for that seed within the distributed seed index. Additionally, targets are in general sequences that are significantly longer than the reads. Thus, multiple reads are expected to be aligned with the same target and a given target t is expected to be reused multiple times in the seed extension procedure.

Given this potential for data reuse, we developed a software cache architecture to reduce communication overhead as shown in Figure 6. In UPC, the address space of every node is logically divided into *private memory* and *shared memory*.

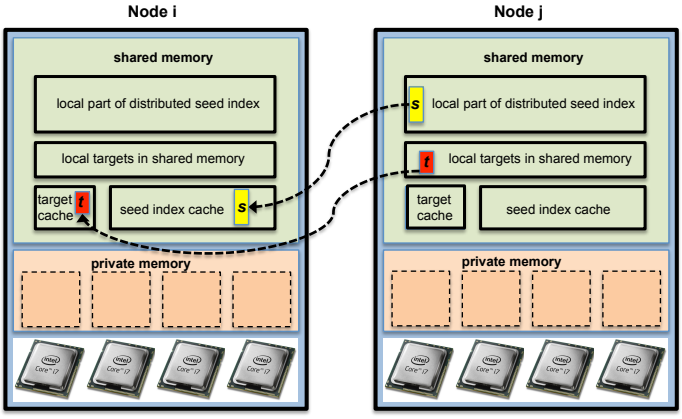

Fig. 6: Software cache architecture for the distributed seed index and the target sequences. Node i has stored in its seed index cache a seed s (yellow block) from Node j. Any lookup for the seed s by processors of Node i will be served by the seed index cache. Similarly, Node i has stored in its target cache a sequence t (red block) from Node j. Any processor of Node i that needs to align a query with respect to t will fetch t from the target cache.

The private memory is thread local and can only be accessed by the UPC thread (which maps to a processor in our case) to which it has affinity. On the other hand, a location of the shared memory can be accessed by *any* UPC thread in the system. It is much faster to access locally stored data than to access shared memory residing on a remote node. Thus, on every node, a portion of the shared memory is dedicated for software caches that can store either remote parts of the distributed seed index (seed index cache) or target sequences owned by remote nodes (target cache). In Figure 6 consider Node i which has stored in its seed index cache a seed s (yellow block) that belongs to the part of the distributed seed index local to Node j. Any lookup for the seed s by processors of Node i will be served by the seed index cache resulting in much faster lookup time than accessing the original yellow block on the remote Node j. Similarly, consider a target sequence t (red block) which has been stored to the target cache of Node i. Processors of Node i that need to align a query in respect to t will fetch t from the target cache and thus avoid the expensive off-node communication.

The expected seed data reuse naturally depends on the seed distribution among processors. As discussed later in Subsection IV-B, for load balancing reasons reads are assigned to processors in a uniformly random fashion. Consider a parallel system with p total processors, with ppn processors per node and a seed s with frequency f in the read data set residing on node i. Following the reasoning in the previous paragraph, there are $f-1$ additional occurrences of that seed s in the read data set or equivalently there are $f-1$ locations in the reads that include that seed.

We can then ask the question: What is the probability that at least one such read is assigned to node i? This problem can be reduced to the well known "bins and balls" experiment. In this case, given $f-1$ balls (remaining occurrences of the seed s) and $m = p/ppn$ bins (nodes), we toss the balls (reads) uniformly at random — the probability that *at least* one of these balls falls in bin i (node i) is $1-(1-1/m)^{f-1}$. Therefore, with probability $1-(1-1/m)^{f-1}$ our approach will perform *at least* one seed index lookup of s resulting in a software cache hit, since there are at least two occurrences of that seed in the same node i. In order to assess the limits of this optimization, consider the case of a read data set with $d = 100$, $L = 100$, $k = 51$, $f = 50$ and a system with $ppn = 24$ cores per node. Figure 7 shows the probability of any seed being reused at least one time given the previous values of f and ppn. Note, however, that this is the behavior in the ideal case of "infinite" cache. In practice, we dedicate a fraction of the nodes' memory for software caching, and tradeoff memory for increased data reuse. For typical experimental values of d and ppn, we expect a significant benefit from our cache optimization strategy, as demonstrated in the experimental results of Section VI.

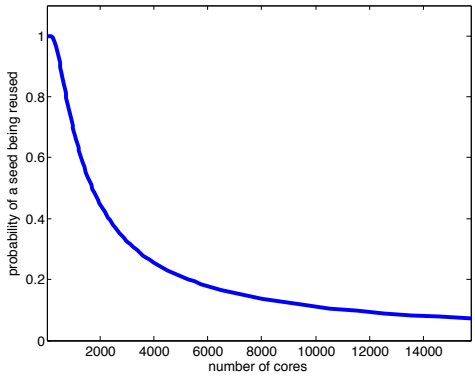

Fig. 7: Probability of any seed being reused as a function of cores. We have picked values of $d = 100$, $L = 100$, $k = 51$, $f = 50$ and ppn=24.

IV. Alignment Optimizations

We now discuss our alignment optimizations and theoretical proofs of expected behavior.

A. Optimizing exact read matches

Here we devise a method to preprocess the target sequences to identify properties enabling exact sequence matching with minimal communication. The property we describe is based on the Lemma 1.

Lemma 1. *Let k be the length of the seeds, q be a query sequence and t be a target sequence where **all** the seeds extracted from t are uniquely located in t (i.e. this seed cannot be found in any other target sequence). Assume that s is a subsequence in q with $length(s) \geq k$ and t is a candidate target to be aligned with q that also includes the subsequence s. Then, q is **uniquely** aligned to t with respect to the subsequence s, in essence there is no other target t' that matches with q in the subsequence s.*

Proof. Since q matches with t in $length(s)$ bases, then all $length(s) - k + 1$ seeds in s belong to both q and t. Since all seeds in t are *uniquely* located in t we conclude that also these $length(s) - k + 1$ seeds are uniquely located in t and therefore there are no other targets that include those seeds. Consequently, there are no other targets that include the subsequence s. □

Consider a subsequence $s \equiv q$, and assume that the first candidate target t_0 is to be aligned with q and it is known that **all** the seeds extracted from t_0 are uniquely located in t_0 (we detail in the subsequent paragraph how to identify such a property). With a fast check we can determine if q and t_0 match in exactly $length(s)$ bases. Given this scenario, then via Lemma 1 with $s \equiv q$ it holds that q is **uniquely** aligned to t_0. Thus it is not necessary to look for more candidate targets and additional seed lookups in the distributed seed index can be avoided. It is thus assured that all possible alignments of q are found (to the set of the targets) by simply performing a single seed lookup — thereby only requiring minimal communication. Further speedups can also be achieved by recognizing that a seed extension algorithm is not necessary in this case, instead a simple and fast string comparison between q and the appropriate location of t_0 can be executed.

We now explain how to identify, efficiently for all target sequences, whether the seeds extracted from t_0 are uniquely located in t_0. During the distributed seed index construction described in Subsection III-A, when a processor adds the received seeds in its local buckets of the hash table, it counts the number of occurrences of each seed — a cheap and local operation. We additionally associate a boolean `single_copy_seeds` flag that is initialized as `true` for all targets. After inserting the seeds into the seed index, a processor p_i can visit all the local seeds and if the count of an encountered seed s' is greater than 1, p_i sets the flags `single_copy_seeds` of the targets that s' was extracted from as `false`. This indicates that those targets do **not** have seeds uniquely located in them. At the end of this step, all the remaining targets with `single_copy_seeds` set to true are guaranteed to have **all** their seeds uniquely located in them.

To maximize the impact of this optimization, we add an additional strategy. Given the seed length k, the longer a target sequence t is, the more probable it is that t contains at least one seed that is not uniquely located in t, thus negating the potential of leveraging the described lookup optimization (even if some reads uniquely match to t). Now consider the case where a target sequence t' has all but one seed a uniquely located in t'. If we fragment t' in two equal-length subsequences t'_1 and t'_2 (that overlap to some degree but have disjoint sets of seeds), then the non-uniquely located seed a in t' should be found (by construction) in either t'_1 or t'_2. Thus the subsequence not containing the seed a consists of uniquely located seeds, thereby enabling our described optimization.

The same reasoning can be applied recursively to address the general case where a target t' contains multiple non-uniquely located seeds. The idea is to fragment the sequence t' into m equal-length subsequences $t'_1, t'_2, ..., t'_m$ that overlap to some degree — however the subsequences have disjoint sets of seeds and the union of their sets of seeds is exactly the set of seeds in the original sequence t'. This approach increases the probability of applying the previous optimization. Note that some additional information must be stored for each one of the subsequences $t'_1, t'_2, ..., t'_m$, to allow quick locating of these subsequences later in the alignment.

B. Load Balancing

Load balancing the queries might initially seem trivial: given n queries and p processors each processor should process n/p queries. Unfortunately, queries may differ in their processing requirements.

For instance, consider a query q' that perfectly aligns with a single target sequence. Let $t_{extractSeed}$ be the required time to extract a seed from a query, $t_{lookupSeed}$ the time to lookup a seed in the seed index, $t_{fetchTarget}$ the time to fetch a target sequence, and $t_{memcmp()}$ the time to perform a `memcmp()` operation on $length(q')$ bytes. Then, the time Algorithm 1 takes (after applying the previous optimization) to process q' is $t_{q'} = t_{extractSeed} + t_{lookupSeed} + t_{fetchTarget} + t_{memcmp()}$. On the other hand, consider a query q'' that can be aligned with C targets. Assume that t_{SW} is the time to execute the Smith-Waterman algorithm. Then, processing q'' takes $t_{q''} = L \cdot (t_{extractSeed} + t_{lookupSeed}) + C \cdot (t_{fetchTarget} + t_{SW})$ time, where $L = length(q'') - length(seed) + 1$. Given $t_{memcmp()} \leq t_{SW}$, it must hold that $t_{q''} \geq min(C, L) \cdot t_{q'}$, thus the processing times of two queries can vary significantly.

Assume that the n queries can be divided in two categories: "fast" and "slow" (depending on their required processing time). The goal is to evenly distribute the slow queries to the available p processors. However, because it is unknown a priori if a query is fast or slow, we implement the following load balancing strategy. Before executing Algorithm 1 the order of the queries is randomly permuted in the input file and each processor is assigned a chunk of n/p consecutive queries from the corresponding file. As proven in Theorem 1, if there are h "slow" queries, p available processors and $p \log p \ll h \leq p$ polylog(p)[1], then with high probability the load imbalance (distance of maximum "slow" load from the average "slow" load h/p) is at most: $2\sqrt{2\frac{h}{p}\log p}$.

Theorem 1. *Let h be the number of "slow" queries and p be the number of available processors and assume that $p \log p \ll h \leq p$ polylog(p). After assigning the h queries randomly to the p processors (or equivalently randomly permuting the order of the queries in the input file) then with high probability the load imbalance (distance of maximum "slow" load from the average "slow" load h/p) is at most: $2\sqrt{2\frac{h}{p}\log p}$.*

Proof. We formulate the process of randomly permuting the order of the queries in the input file as the uniformly random tossing of h balls into p bins. Let M be the random variable that counts the maximum number of balls ("slow" queries) in any bin. It therefore holds that $Pr[M \leq k] = 1 - o(1)$ where $k = \frac{h}{p} + 2\sqrt{2\frac{h}{p}\log p}$ [13], i.e. with high probability the load imbalance (distance of maximum "slow" load from the average "slow" load h/p) is at most: $2\sqrt{2\frac{h}{p}\log p}$. □

C. Restricting the Maximum Alignments per Seed

Even after applying the described load balancing scheme, there may be a few seeds that can be aligned with too many

[1] polylog(p) is some polynomial in log(p)

targets, causing a high processing time for the corresponding queries. Additionally, finding those numerous alignments may not be relevant to many genome alignment applications. Thus, a threshold can be set for the maximum number of alignments per seed, after which the candidate alignment queries are stopped. This threshold determines the sensitivity of our aligner and it can be used to trade off accuracy for speed when appropriate.

V. Additional Optimization

We now describe the additional set of I/O, SIMD, and compression optimizations utilized in our work.

A. Parallel 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 encoding 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. As previously described [7], our work has therefore been leveraging SeqDB [14], a binary format for storing DNA short reads that is implemented on HDF5 [15]. Although SeqDB was not originally designed with parallelism in mind, its HDF5 format allowed us to utilize parallel input data reading using Parallel HDF5 via the MPI-IO with modest modifications, thus enabling a portable solution. Note that the compression from FASTQ to SeqDB is a one-time lossless conversion, where the resulting file is typically 40-50% smaller than the original FASTQ file. While this compression ratio is less compact than other competing formats, SeqDB is significantly faster during the decompress phase. Overall, the SeqDB format is well suited for parallel sequence processing, and we envision it being used more widely by the community.

B. SIMD Optimized Striped Smith-Waterman

MerAligner spends a significant portion of its runtime using the Smith-Waterman (SW) algorithm for seed extension. Due to the critical role of SW, many efforts have been made to accelerate it by taking the advantages of special hardware SIMD (Single Instruction Multiple Data) instructions. In this work we incorporate such an implementation from the Striped Smith-Waterman (SSW) library [16] which has been shown to be orders of magnitude faster than reference implementations of SW in C.

C. DNA Sequence Compression

Given the $\{\texttt{A},\texttt{C},\texttt{G},\texttt{T}\}$ vocabulary of a DNA sequence, only two-bits per base are required for binary representations. We thus use a high-performance compression library that transforms the DNA sequences from text format into a binary format [7]. This approach reduces the memory footprint by $4\times$, while also reducing the bandwidth by $4\times$ for communication events that involve seeds or DNA sequence transfers.

VI. Experimental Results

We now discuss our experimental testbed and runtime results across a variety of input sets, concurrencies, and optimization schemes, as well as comparison with competing approaches.

A. Experimental Testbed

High-concurrency experiments are conducted on Edison, a Cray XC30 supercomputer at NERSC[17]. 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.

Our experiments utilize real data sets for human and wheat genomes. The human data set contains 2.5 billion reads (252 Gbp of sequence) for a member of the CEU HapMap population (identifier NA12878) sequenced by the Broad Institute, and our goal is to align those reads onto a set of target sequences (contigs) that are generated in the Meraculous de novo genome assembly pipeline. The reads are 101 bp in length from a paired-end insert library with mean insert size 238 bp. The wheat data set, contains 2.3 billion reads (477 Gbp of sequence) for the homozygous bread wheat line 'Synthetic W7984' sequenced by the JGI and again we want to align those reads onto a set of target sequences (contigs) that are generated in the Meraculous pipeline. The reads are 100-250 bp in length from 5 paired-end libraries with insert size 240-740 bp. For all experiments the seed length is set to 51, as used in the actual scaffolding step of the Meraculous pipeline.

B. Strong Scaling of End-to-End merAligner

Figure 1 (page 1) shows the merAligner end-to-end strong scaling performance with all optimizations applied. This summarizes the main result of this study, and demonstrates the efficient utilization of distributed memory architectures for enabling scalable high performance sequence alignment. More specifically when scaling from 480 to 15,360 cores the total execution time drops from 4,147 seconds to 185 seconds (a $22\times$ speedup), which translates to 0.7 parallel efficiency at the extreme scale for the human dataset (red curves). At the scale of 15,360 cores our approach performs alignment at 15,499,718 reads/sec. For the larger wheat data set (blue curves), scaling from 960 to 15,360 cores achieves 0.78 parallel efficiency. Note the super-linear speedup in the range of 960 — 7,680 cores, which we speculate is due to reduced congestion on the NIC since the communication is spread to even more nodes while we scale. This will be the subject of future investigation.

C. Anatomy of the Optimizations' Benefits

We now examine the individual effect our optimization schemes, by selectively turning them off and measuring the resulting performance impact.

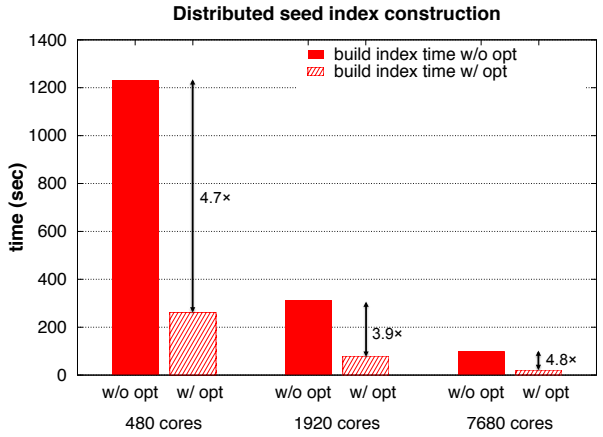

Fig. 8: Distributed seed index construction scaling before and after applying the "aggregating stores" optimization for the human data set.

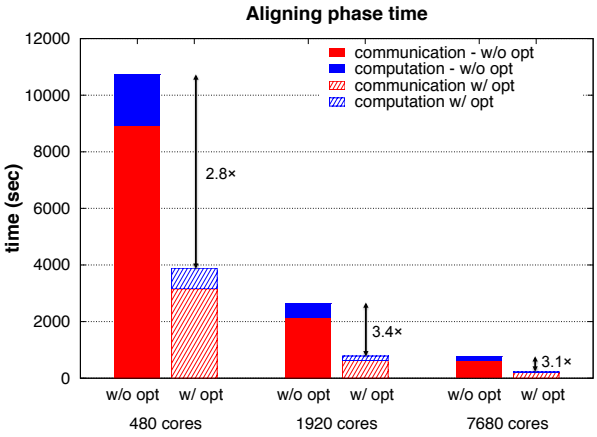

Fig. 10: Impact of "exact matching optimization" on the aligning phase of the human data set.

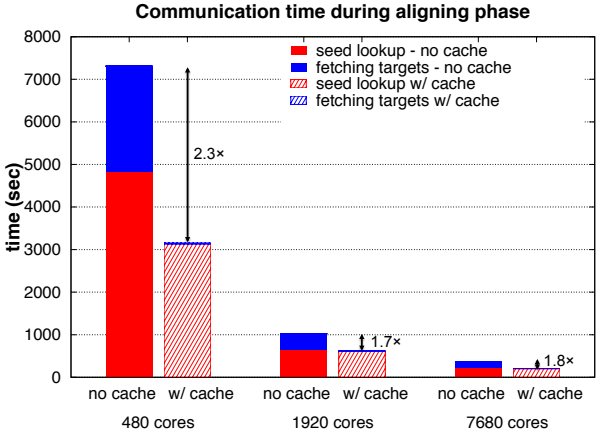

Fig. 9: Impact of software caching on communication for the alignment of the human data set.

1) Distributed Seed Index Construction: Figure 8 illustrates the scaling of the distributed seed index construction before and after applying the "aggregating stores" optimization, given an $S = 1000$. Observe that the reduction in the communication via our optimization dramatically decreases the construction time. At 480 cores the time spent decreases from 1,229 seconds to 262 seconds ($4.7\times$ improvement), and similarly at 7,680 cores we achieve an improvement of $4.8\times$. For the optimized construction phase, increasing concurrency from 480 cores to 7,680 ($16\times$ core increase) results in a near-linear speedup speedup of $12.7\times$. These results show that our algorithm efficiently parallelizes the seed index construction in a distributed memory and enables end-to-end scaling of merAligner. These scalable seed index construction results are in contrast to serial approaches of competing alignment codes as detailed in Subsection VI-D. Also, the algorithm achieves almost perfect load balance in terms of the number of distinct seeds assigned to each processor, thanks to our use of the `djb2` hash function to implement the seed to processor map.

2) Software Caching: In Figure 9 we depict the benefits of software caching on the communication time during the alignment phase (in all experiments 16 GB/node and 6 GB/node are allocated for the seed index and the target cache respectively). The red bars indicate the communication time for the seed lookups and the blue bars represent target sequence fetching overhead. Observe that the target cache is extremely efficient at all concurrencies and it essentially obviates all the communication involved with target sequences. Results also show that the seed index cache is effective at small concurrencies, where lookup time is decreased from 4,839 seconds to 3,130 seconds (~35% reduction) at 480 cores, whereas larger concurrencies see small benefits — validating our analysis in Subsection III-B. Overall, the caching scheme decreases communication overhead by $2.3\times$, $1.7\times$ and $1.8\times$ at concurrencies of 480, 1,920 and 7,680 cores respectively.

3) Exact Read Matching Optimization: Figure 10 shows the significant performance benefits of exact read matching, validating the theoretical analysis of Subsection IV-A. Here the optimization results in runtime improvement of the alignment step by factors of $2.8\times$ and $3.1\times$ for 480 cores and 7,680 cores respectively. Note that these gains come from both decreased communication (since in exact matching just one seed lookup is sufficient) and reduction of computation time (by avoiding Smith-Waterman execution). For example, at 480 cores our approach improves computation by $2.48\times$ and communication by $2.82\times$. Finally we emphasize that ~59% of the aligned reads took advantage of this optimization, thus enabling these impressive performance gains. For the optimized aligning phase, increasing concurrency from 480 cores to 7,680 ($16\times$ core increase) results in a near-linear speedup of $15.9\times$.

4) Load Balancing: In order to assess the effectiveness of the load balancing scheme, we conducted experiments with and without permuting the input read files and measured the maximum, minimum and average computation time as well as alignment times (computation plus communication). Results for 480 cores for the human data set are shown in Table I. Although our load balancing scheme effectively helps reduce the maximum computation time by almost $2.5\times$, the total alignment time is only improved by ~5%. A closer investigation of the original data set reveals that the reads mapping to the same genome region are grouped together. Since some groups of reads did not map to any target, they

Load Balancing	Computation time			Total Alignment time		
	Min	Max	Avg	Min	Max	Avg
Yes	678	**800**	740	2700	**3885**	3277
No	515	**1945**	690	1512	**4092**	2073

TABLE I: Effect of load balancing scheme on the human data set, showing the reduction of the maximum compute time

do not require Smith Waterman execution, and thereby cause an imbalanced computing load. However, this locality made our seed index cache extremely effective and substantially decreased the communication time. Therefore, our load balancing scheme alleviates the computational load imbalance, while making the seed index cache less effective as seen in Table I. Nonetheless, our approach improves the overall execution time. We note that the read grouping in the original data set is not the common case, and thus expect our load balancer to be even more effective in the general case.

D. Comparison with Existing Parallel Aligners

To assess our optimized merAligner in the context of existing solutions, we compare human data performance with BWA-mem [8] and Bowtie2 [9] using the pMap [2] framework. Note that pMap was modified to use the latest versions of the alignment software. Our experiments are configured using 4 instances of 6 threads per Edison node, since it is not possible to run one instance per core due to memory requirements of BWA-mem and Bowtie2 (each node contains 64GB of memory, which is insufficient to hold 24 instances of the seed index). BWA-mem is run with minimum seed length equal to 51 (like merAligner) since alignments with smaller seeds are not applicable to the scaffolding computation. For Bowtie2, we set the minimum seed length to the maximum possible value (31) and we execute the experiment with the `--very-fast` option in order to achieve the best mapping runtime. It is important to highlight that the seed index construction for BWA-mem and Bowtie2 is performed serially. For both cases, pMap partitions the reads to the available instances, then the seed index is loaded into each instance's memory and finally the corresponding instance is called on the set of reads assigned to it.

Table II presents comparative end-to-end performance results at 7,680 cores, and notes which computing phases are performed in serial *(S)* or parallel *(P)*. As expected, the serial seed index construction is a major bottleneck for the competing codes, compared with our parallel merAligner approach. Also, pMap spends a significant amount of time in read partitioning by having a single process sending the appropriate portion of the input read files to the corresponding node (4,305

Aligner	Seed Index Construction	Mapping Time	Total	Speedup
merAligner	21 *(P)*	263 *(P)*	**284 sec**	**1**×
BWA-mem	5,384 *(S)*	421 *(P)*	**5,805 sec**	**20.4**×
Bowtie2	10,916 *(S)*	283 *(P)*	**11,119 sec**	**39.4**×

TABLE II: End-to-end performance comparison between parallel executions of merAligner, BWA-mem and Bowtie2 using 7,680 cores on the human data set (with all times in seconds) — highlighting serial *(S)* or parallel *(P)* implementation of the phases.

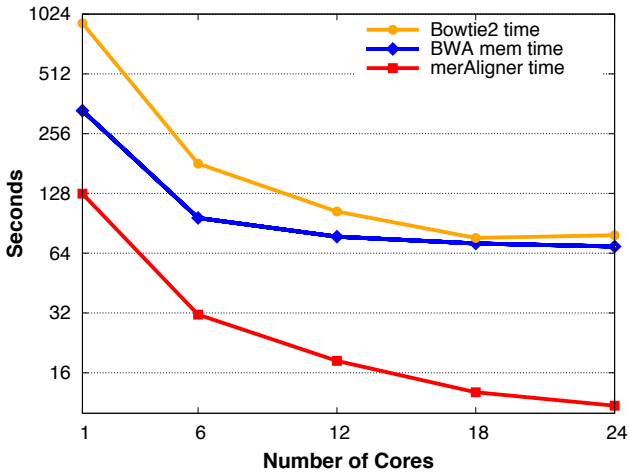

Fig. 11: Shared memory performance of merAligner, BWA-mem and Bowtie2 on a single node of Edison on the *E. coli* data set. At the scale of 24 cores, merAligner is 6.33× faster than BWA mem and 7.2× faster than Bowtie2.

and 3,982 seconds for BWA-mem and Bowtie2 respectively). On the contrary, merAligner does not suffer from this overhead since all processors read *in parallel* the appropriate portions of the input read files. To make though a fair comparison, we exclude the timing of the read partitioning for the cases of BWA-mem and Bowtie2. In total, merAligner is 20.4× and 39.4× faster than the parallel execution of BWA-mem and Bowtie2, respectively.

A complete analysis of the accuracy of the method is outside of the scope of this paper. The algorithm is guaranteed to identify all alignments that share at least one identically matching stretch of at least $length(seed)$ consecutive bases between query and target sequences. Whether such alignments are sufficient is largely an application-dependent question. For the purposes of the Meraculous de novo assembly toolkit, these alignments are precisely those required. Here, we simply report all alignments detected (i.e. without any percent-identity thresholding, using a commonly employed scoring matrix) and find that merAligner successfully aligned 86.3% of the reads, while BWA-mem and Bowtie2 aligned 83.8% and 82.6% of the reads respectively. The accuracy of the detailed alignments is a function of the Smith-Waterman code, and we refer the interested reader to that publication [16].

To further assess merAligner performance, we conduct experiments on the smaller 4.64 Mbp *E. coli* K-12 MG1655 dataset, which allows single node scalability experiments using both BWA-mem and Bowtie2 in parallel mode with threads. The execution time of all three approaches (using a seed length of 19) is shown in Figure 11. Observe that merAligner performance continues to scale using all 24 available cores, while the runtimes of BWA-mem and Bowtie2 stop improving at 18 cores. Overall, merAligner is significantly faster, exceeding BWA-mem and Bowtie2 performance on 24 cores by 6.33× and 7.2× respectively. Subject to the alignment correctness discussion above, we find that merAligner successfully aligned 97.4% of the reads, while BWA-mem and Bowtie2 aligned 96.3% and 95.8% of the reads respectively.

VII. Related work

A thorough survey of sequence aligners is beyond the scope of our work and we refer the reader to [18], [19], [20], [21], [22], [23]. We primarily focus on parallel sequence mapping tools and relevant methods in this section. CUSHAW2 [24], BWA [25], BWA-mem [8], Bowtie2 [9], SNAP [26], SOAP [27] and GSNAP [28] are mapping tools that employ shared memory parallelism during the aligning phase. However, these approaches are more restrictive as they are limited by the concurrency and memory capacity of the shared-memory node. CUSHAW2-GPU [29] and SOAP3-dp [30] are short read aligners that leverage GPU power on a single compute node. pMap [2] is a MPI-based tool used to parallelize existing short sequence mapping tools (like the ones mentioned above) by partitioning the reads and distributing the work among the processors. However, pMap does not leverage any parallelism during the index table construction and therefore a serialization bottleneck is introduced in the mapping pipeline. PBWA [31] employs MPI in order to execute BWA on distributed memory machines, however the index table construction and its replication are serial processes. Also, the sequence distribution is done by a single master process. Therefore, pBWA suffers from the same limitations as pMap. Menon et al. [32] parallelize the genome indexing with MapReduce, however the scaling they obtain is poor.

Bozdag et al. [33] evaluate different methods of distributed memory parallelization of a mapping pipeline. These methods fall basically in three categories: (i) partitioning the reads only, (ii) partitioning the genome (and consequently the index table) and (iii) hybrid method of (i) and (ii) that partitions both reads and the genome (and the index table). One conclusion of this study is that method (i) suffers from the serialized index table construction, method (ii) does not scale in the mapping phase, and regarding method (iii), even though it exhibits improved scalability, its scaling is not close to linear. The main reason is that in the hybrid method, the index table creation is parallelized among subgroups of processes and the reads are also partitioned among subgroups of processes. Therefore, the hybrid method does not exploit the highest possible level of concurrency. Our work does fully parallelize the index table creation and partitions the reads using all available processors.

pFANGS [34] also tries to parallelize both the index table construction and the alignment phase. It distributes the index table among the processors but the processors can not look up the distributed index in arbitrary locations. Therefore, the index lookup tasks are localized first, then an all-to-all personalized communication step is performed, the local lookups take place, and finally the lookup results are redistributed such that they are placed with the relevant queries (this redistribution is done with all-to-all personalized communication). The authors identify that the communication becomes a bottleneck because of the all-to-all communication and therefore they divide the processes in disjoint subgroups where each subgroup works independently by creating its own copy of the index table. However, in this approach the scaling of the index table construction is limited by the size of each subgroup. Orion [35] is an improvement over mpiBLAST [36] and scales the sequence matching with fine-grained parallelization. However, Orion uses mpiBLAST's `mpiformatdb` tool to format and to shard the database and this process is serial.

Our previous study [7] investigated the building of distributed hash tables for the contig construction phase of Meraculous [6]. This work extends the algorithm for our distributed seed index optimizations, which now allows multiple seeds to map onto a given target. Also, in this work we enhance the distributed hash table with software caching support. Finally, Kassens et al [37] employed a PGAS language called UPC++ to parallelize Genome-Wide Association Studies and showed that this programming model is suitable for data-intensive bioinformatics applications.

VIII. Conclusions

This work presents a highly scalable sequence alignment algorithm that effectively parallelizes all computational phases, including seed index construction. Achieving our solution required numerous innovations including software-caching, accelerated exact sequence matching, I/O optimization, and load balancing via randomization. Overall we achieved near perfect scaling on up to 15K cores using real human and wheat data sets, while significantly exceeding the end-to-end performance of existing approaches by factors of 20-39$\times$.

The approach described here was initially developed as part of a UPC adaptation of the Meraculous genome assembler [3], since aligning reads to a reference set of contigs was rate limiting after other stages of the assembler had been parallelized. Other shotgun assemblers have comparable steps, and adaptations of our method (perhaps using different detailed alignment scoring after identifying and extending seed matches) would also likely be useful as modules in other assembly settings. The Striped Smith-Waterman local alignment engine could easily be replaced with any other local alignment software tool; more broadly any seed-and-extend algorithm could be implemented with minor changes to the underlying protocols, including protein-DNA and protein-protein alignments. Importantly, for the case of alignment of read sets from multiple individuals against a reference genome, the cost of building an index can be amortized across the individuals. In contrast, for de novo assembly of a new genome a new index needs to be built for each new assembly. Therefore, the parallelization of index construction is a key element of our approach relative to existing short-read aligners.

While we have focussed on the DNA-alignment problem, and in particular its application in the context of whole-genome shotgun assembly, the work presented here is of sufficient generality that we envision merAligner providing a framework for a generic, distributed hash platform for a variety of applications. In the DNA sequence domain, large genomes (e.g., wheat or pine) may require indices that, unlike human, may not fit in the memory on a single node; our parallel approach avoids this limitation by constructing the index in parallel across distributed memories. Because merAligner

is strongly scalable, it supports the exponential growth and complexity of genome references, in addition to the rapid throughput needed to align massive numbers of genomes.

Other large sequence collections, like the GenBank [38] collection of all known genomic data, are not feasible to index using the methods of existing tools such as BWA and Bowtie; in principle, our approach is scalable to allow short reads from any source to be rapidly aligned against the complete GenBank collection. Extending our approach to other alphabets, one can also use the same methods to align protein sequences (strings of 20 characters, each corresponding to one of the 20 amino acids) against protein datasets. Finally, more general text based queries outside of the biological sequence domain may also be facilitated by our new parallel approach.

ACKNOWLEDGMENTS

Authors from Lawrence Berkeley National Laboratory were supported by the Applied Mathematics and Computer Science Programs of the DOE Office of Advanced Scientific Computing Research and the DOE Office of Biological and Environmental Research under contract number DE-AC02-05CH11231. The first author is also supported by the grant DE-SC0008700. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

REFERENCES

[1] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of molecular biology*, vol. 215, no. 3, pp. 403–410, 1990.

[2] "pMap: Parallel Sequence Mapping Tool," http://bmi.osu.edu/hpc/software/pmap/pmap.html.

[3] J. A. Chapman, M. Mascher, K. Barry, E. Georganas, A. Session, V. Strnadova, J. Jenkins, S. Sehgal, L. Oliker *et al.*, "A whole-genome shotgun approach for assembling and anchoring the hexaploid bread wheat genome," *Genome biology*, vol. 16, no. 1, p. 26, 2015.

[4] A. Zimin, K. A. Stevens, M. W. Crepeau, A. Holtz-Morris, M. Koriabine, G. Marçais, D. Puiu, M. Roberts, J. L. Wegrzyn, P. J. de Jong *et al.*, "Sequencing and assembly of the 22-gb loblolly pine genome," *Genetics*, vol. 196, no. 3, pp. 875–890, 2014.

[5] M. Hunt, C. Newbold, M. Berriman, and T. D. Otto, "A comprehensive evaluation of assembly scaffolding tools," *Genome biology*, vol. 15, no. 3, p. R42, 2014.

[6] 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.

[7] 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.

[8] H. Li, "Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM," *arXiv preprint arXiv:1303.3997*, 2013.

[9] B. Langmead and S. L. Salzberg, "Fast gapped-read alignment with Bowtie 2," *Nature methods*, vol. 9, no. 4, pp. 357–359, 2012.

[10] P. Husbands, C. Iancu, and K. Yelick, "A performance analysis of the Berkeley UPC compiler," in *Proc. of Int. Conf. on Supercomputing (ICS)*. ACM, 2003, pp. 63–73.

[11] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of molecular biology*, vol. 147, no. 1, pp. 195–197, 1981.

[12] B. Liu, Y. Shi, J. Yuan, X. Hu, H. Zhang, N. Li, Z. Li, Y. Chen, D. Mu, and W. Fan, "Estimation of genomic characteristics by analyzing k-mer frequency in de novo genome projects," *arXiv preprint arXiv:1308.2012*, 2013.

[13] M. Raab and A. Steger, "Balls into Bins: A Simple and Tight Analysis," in *Randomization and Approximation Techniques in Computer Science*. Springer, 1998, pp. 159–170.

[14] 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.

[15] M. Folk, A. Cheng, and K. Yates, "HDF5: A file format and I/O library for high performance computing applications," in *Proceedings of Supercomputing*, vol. 99, 1999.

[16] M. Zhao, W.-P. Lee, E. P. Garrison, and G. T. Marth, "SSW Library: An SIMD Smith-Waterman C/C++ library for use in genomic applications," *PloS one*, vol. 8, no. 12, p. e82138, 2013.

[17] "NERSC," https://www.nersc.gov.

[18] A. Hatem, D. Bozdağ, A. E. Toland, and Ü. V. Çatalyürek, "Benchmarking short sequence mapping tools," *BMC bioinformatics*, vol. 14, no. 1, p. 184, 2013.

[19] H. Li and N. Homer, "A survey of sequence alignment algorithms for next-generation sequencing," *Briefings in bioinformatics*, vol. 11, no. 5, pp. 473–483, 2010.

[20] N. A. Fonseca, J. Rung, A. Brazma, and J. C. Marioni, "Tools for mapping high-throughput sequencing data," *Bioinformatics*, p. bts605, 2012.

[21] M. Ruffalo, T. LaFramboise, and M. Koyutürk, "Comparative analysis of algorithms for next-generation sequencing read alignment," *Bioinformatics*, vol. 27, no. 20, pp. 2790–2796, 2011.

[22] M. Holtgrewe, A.-K. Emde, D. Weese, and K. Reinert, "A novel and well-defined benchmarking method for second generation read mapping," *BMC bioinformatics*, vol. 12, no. 1, p. 210, 2011.

[23] S. Schbath, V. Martin, M. Zytnicki, J. Fayolle, V. Loux, and J.-F. Gibrat, "Mapping reads on a genomic sequence: an algorithmic overview and a practical comparative analysis," *Journal of Computational Biology*, vol. 19, no. 6, pp. 796–813, 2012.

[24] Y. Liu and B. Schmidt, "Long read alignment based on maximal exact match seeds," *Bioinformatics*, vol. 28, no. 18, pp. i318–i324, 2012.

[25] H. Li and R. Durbin, "Fast and accurate short read alignment with Burrows–Wheeler transform," *Bioinformatics*, vol. 25, no. 14, pp. 1754–1760, 2009.

[26] M. Zaharia, W. J. Bolosky, K. Curtis, A. Fox, D. Patterson, S. Shenker, I. Stoica, R. M. Karp, and T. Sittler, "Faster and more accurate sequence alignment with SNAP," *arXiv preprint arXiv:1111.5572*, 2011.

[27] R. Li, Y. Li, K. Kristiansen, and J. Wang, "SOAP: short oligonucleotide alignment program," *Bioinformatics*, vol. 24, no. 5, pp. 713–714, 2008.

[28] T. D. Wu and S. Nacu, "Fast and SNP-tolerant detection of complex variants and splicing in short reads," *Bioinformatics*, vol. 26, no. 7, pp. 873–881, 2010.

[29] Y. Liu and B. Schmidt, "Cushaw2-gpu: empowering faster gapped short-read alignment using gpu computing," *Design & Test, IEEE*, vol. 31, no. 1, pp. 31–39, 2014.

[30] R. Luo, T. Wong, J. Zhu, C.-M. Liu, X. Zhu, E. Wu, L.-K. Lee, H. Lin, W. Zhu, D. W. Cheung *et al.*, "Soap3-dp: fast, accurate and sensitive gpu-based short read aligner," *PloS one*, vol. 8, no. 5, p. e65632, 2013.

[31] D. Peters, X. Luo, K. Qiu, and P. Liang, "Speeding up large-scale next generation sequencing data analysis with pBWA," *J Appl Bioinform Comput Biol*, vol. 1, p. 2, 2012.

[32] R. K. Menon, G. P. Bhat, and M. C. Schatz, "Rapid parallel genome indexing with MapReduce," in *Int. workshop on MapReduce and its applications*. ACM, 2011, pp. 51–58.

[33] D. Bozdag, C. C. Barbacioru, and U. V. Catalyurek, "Parallel short sequence mapping for high throughput genome sequencing," in *IPDPS*. IEEE, 2009.

[34] S. Misra, R. Narayanan, W.-k. Liao, A. Choudhary, and S. Lin, "pFANGS: Parallel high speed sequence mapping for next generation 454-roche sequencing reads," in *IPDPSW*. IEEE, 2010, pp. 1–8.

[35] K. Mahadik, S. Chaterji, B. Zhou, M. Kulkarni, and S. Bagchi, "Orion: Scaling genomic sequence matching with fine-grained parallelization," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'14)*, 2014.

[36] A. Darling, L. Carey, and W.-c. Feng, "The design, implementation, and evaluation of mpiBLAST," *Proceedings of ClusterWorld*, vol. 2003, 2003.

[37] J. C. Kassens, J. Gonzalez-Dominguez, L. Wienbrandt, and B. Schmidt, "Upc++ for bioinformatics: A case study using genome-wide association studies," in *Cluster Computing (CLUSTER), 2014 IEEE International Conference on*. IEEE, 2014, pp. 248–256.

[38] D. A. Benson, M. Cavanaugh, K. Clark, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, and E. W. Sayers, "GenBank," *Nucleic acids research*, p. gks1195, 2012.