# Distributed de novo assembler for large-scale long-read datasets

Sayan Goswami
*Department of Computer Science*
*Louisiana State University Shreveport*
*Sayan.Goswami@lsus.edu*

Kisung Lee
*School of Electrical Engineering*
*& Computer Science*
*Louisiana State University*
*klee76@lsu.edu*

Seung-Jong Park
*School of Electrical Engineering*
*& Computer Science, Center for*
*Computation & Technology*
*Louisiana State University*
*sjpark@cct.lsu.edu*

***Abstract*—Third-generation DNA sequencing technologies such as single-molecule real-time sequencing (SMRT) and nanopore sequencing have the potential to fill the gaps in the existing genome databases since the raw sequences produced by these machines are much longer than those of previous generations and therefore result in more contiguous assemblies. However, these long reads have a high error rate, which makes the assembly process computationally challenging. Moreover, since existing long-read assemblers are designed to run on a single machine, they either take days to complete or run out of memory on even moderate-sized datasets. In this paper, we present a distributed long-read assembler that can assemble large-scale noisy sequence datasets on thousands of cores, resulting in orders of magnitude faster assembly times. By effectively using the map-reduce computation model with a distributed hash-map, both built using a high-performance active messaging middleware, we can assemble a PacBio human genome dataset with 139 billion base-pairs (about 130 GB) in about 33 minutes (using 2,560 cores) compared to more than 38 hours (using 28 cores) with the current state-of-the-art assembler.**

***Index Terms*—genome assembly, third-generation sequences, long reads, big data, map-reduce, high-performance computing**

## 1. Introduction

The study of an organism's genome (i.e., the entire set of its DNA) has a variety of applications. Notable use-cases are treatments of cancer using precision medicine [1], monitoring food contamination [2], [3], outbreaks of food- and water-borne diseases [4], and drug resistance in bacteria [5]. Extracting the DNA information consists of two stages - DNA sequencing and sequence assembly. The process by which nucleotides of a DNA sequence are parsed into text is called sequencing, and the devices used to do so are called sequencers. Existing sequencers cannot process entire genomes without significant loss of accuracy so they extract small regions called *reads* from pseudo-random positions in the DNA. With a high enough sampling depth, these reads will likely have overlaps between them. The process by which these overlapping reads are aligned and merged to recreate the original sequence is known as *de novo sequence assembly* (henceforth referred to as assembly).

The main challenge in assembly is that the positions in the original DNA sequence from where the reads were sampled are not known, and thus rebuilding it is equivalent to solving the shortest common superstring problem. Hence programs known as assemblers are used to find the overlaps between all pairs of reads, encode them in a graph, and use it to estimate the original sequence. Ideally, if reads have perfect or near-perfect accuracy, the overlaps are found by indexing them using FM-Indexes, de Bruijn graphs, etc. However, reads produced by the third-generation sequencers (also called long-read sequencers) contain several base-level mismatches/substitutions, insertions, and deletions with error rates varying between 15-30% [6]. This renders exact matching ineffective in finding overlaps so long-read assemblers use computationally expensive algorithms like Smith-Waterman to find gapped alignments between sequences [7]–[9]. An example of the types of errors in two overlapping noisy reads is shown in Fig. 1.

Despite the higher error rate, third-generation sequences are 2-3 orders of magnitude longer and known to provide higher-quality assemblies owing to a better repeat-resolution [9]–[11]. Therefore, these datasets can be useful in filling the gaps (unknown regions) in the reference genomes of organisms. For instance, the recently assembled results from Oxford Nanopore Technologies of the human genome report closure of 12 gaps, each longer than 50 kilobases [12]. Moreover, advances in single-molecule sequencing technology have reduced costs, increased throughput, and produced much more portable machines.

In contrast, long-read assemblers are still lagging compared to their previous generation (NGS) counterparts. Despite using costly inexact-matching algorithms, they run on a single node and, in some phases, use one thread. Therefore, they often have long execution times, especially for large genome datasets, which can be several hundreds of GBs in size. Moreover, they are prone to running out of memory unless executed on machines with large main memories. To the best of our knowledge, no end-to-end distributed long-read assembler exists, which means current assemblers often take days as they are limited by the number of cores, the size

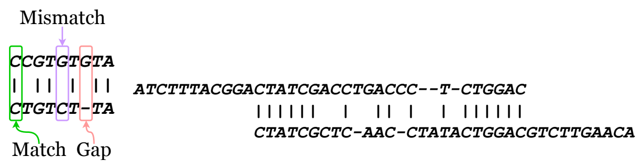

Mismatch
CCGTGTGTA
CTGTCT-TA
Match
Gap
ATCTTTACGGACTATCGACCTGACCC--T-CTGGAC
CTATCGCTC-AAC-CTATACTGGACGTCTTGAACA

Figure 1: Types of errors in two overlapping sequences

of main memory, and the I/O bandwidth of a single machine. We believe that this is the biggest roadblock against the widespread adoption of third-generation sequencing.

In this paper, we present Hiper3ga [1] (High-Performance 3rd-Generation Assembler), a distributed long-read assembler that can assemble large datasets faster by scaling-up on multiple machines. To the best of our knowledge, this is the first fully distributed long-read assembler reported in the literature. This paper makes the following contributions. 1) Hiper3ga distributes and parallelizes the computation in the Minimap+Miniasm [11], which is reportedly the fastest long-read assembler. Hiper3ga can scale up to hundreds of nodes and thousands of cores and thus efficiently assemble large-scale datasets that are otherwise difficult or time-consuming to process using a single node. 2) Hiper3ga builds a high-performance map-reduce framework and a distributed multi-map using GASNet's [14] RDMA-based active messaging primitives over high-throughput communication conduits. Although we evaluate Hiper3ga using Infiniband networks, it can run seamlessly on other conduits such as Cray Aries, Gemini, and IBM's Torus owing to GASNet's conduit transparency. 3) We evaluate Hiper3ga using several real-world datasets, ranging in size from about 3 GB to 139 GB, on a variety of cluster configurations and report the assembly of a PacBio human genome dataset in about 33 minutes on 128 nodes, which is about 71× faster than its single node ancestor.

The rest of the paper is structured as follows. In Section 2, we provide a background of the assembly problem and summarize the related works before stating the motivation for this work. Next, in Section 3, we describe the methodology and implementation details of Hiper3ga. Lastly, we evaluate our work on various datasets against the state-of-the-art long-read assembler in Section 4 and conclude this paper in Section 5.

## 2. Background and Related Work

Given a set of sequences $\mathcal{S}$ ($|\mathcal{S}| = r$) over the alphabet $\Sigma = \{A, C, T, G\}$, our goal is to find a set of alignments $\mathcal{A} \subset \mathcal{S} \times \mathcal{S} \times \mathbb{N}^4$, where each $a_i \in \mathcal{A}$ is a 6-tuple $(s_i, s_j, b_i, b_j, e_i, e_j)$ that corresponds to an overlap between the sequences $s_i$ and $s_j$ starting from positions $b_i$ and $b_j$ respectively and ending at positions $e_i$ and $e_j$ respectively. For each $s_i \in \mathcal{S}$, we define a list $\mathcal{K}_i$ of *k-mers* such that a $k$-mer $k_{ij} \in \mathcal{K}_i$ is the $k$-length substring of $s_i$ starting from position $j$. If two sequences $s_i$ and $s_j$ have an overlap longer than $k$, then we have $\mathcal{K}_i \cap \mathcal{K}_j \neq \emptyset$.

1. The code is available at https://github.com/sayangoswami/hiper3ga.git

$k$-mers are used in the seed-and-extend method [15] of finding overlaps where, for any two sequences $s_i$ and $s_j$, we find $k$-mers common to both $\mathcal{K}_i$ and $\mathcal{K}_j$ and align the sequences such that these common $k$-mers (known as seeds) are also aligned. In order to perform efficient all-vs-all alignment, all $k$-mers are indexed in a fashion that allows us to find the list of all sequences and the corresponding positions where any given $k$-mer occurs. Therefore, all $k$-mers in $\mathcal{K} =< \mathcal{K}_1, \mathcal{K}_2, \cdots, \mathcal{K}_r >$ along with their locations are indexed in the form of 3-tuples $(k_{ij}, i, j)$.

To reduce memory usage, most approaches index only those $k$-mers that are representative of the sequence. Such a set of representative $k$-mers can be chosen in a variety of ways. BLAST [16], which is one of the most popular tools for sequence alignment, uses $k$-mers at fixed intervals (i.e., every $p^{\text{th}}$ $k$-mer) of the reference sequence and indexes them on a hash table. On receiving a query sequence, it generates all $k$-mers from it and searches for them in the hash table of reference $k$-mers. If there are several matches in a relatively small window, it deduces that the query and reference sequences are aligned in that window. On the other hand, MHAP [7] uses a set of $m$ distinct hash functions $\{h_1, h_2, \cdots, h_m\}$ to choose $m$ representative $k$-mers $\{k^r_{i\,1}, k^r_{i\,2}, \cdots, k^r_{i\,m}\}$ from each sequence $s_i \in \mathcal{S}$ such that $k^r_{i\,j} = \text{argmin}_{x \in \mathcal{K}_i} h_j(x)$. These $k$-mers are used to create a min-hash sketch for each sequence, and the similarity between two sequences is detected by approximating their Jaccard coefficients using their respective sketches. While this method is memory efficient, it only provides a cluster of similar sequences instead of a set of candidate seeds/anchors where these sequences are aligned to each other.

Another method of selecting anchor $k$-mers is by using minimizers [17]. In this method, a $k$-mer $k_{ij}$ is only considered for the role of an anchor if $s_i$ contains a region of $w$ consecutive $k$-mers $\mathcal{K}_i{}' \subset \mathcal{K}_i$ for which $k_{ij} = \text{argmin}_{y \in \mathcal{K}_i{}'} f(y)$, where $f$ is a hash function. This technique is similar to that used in MHAP, but while MHAP picks a fixed number of candidates from the entire sequence, this selects one candidate from each fixed-size window of the sequence. This method is also more efficient than BLAST since it queries a fraction of all the $k$-mers in a sequence. The minimizers are then used as seeds to align sequences even if they do not have regions that exactly match by virtue of having a large number of minimizers in common. This process is depicted in Fig. 2 where we generate 4-length minimizers (in dark) using lexicographical ordering on a window size 2 (i.e., two $k$-mers), and use the common minimizers between two overlapping noisy reads (in colored) as anchors to find regions of alignment.

Most long-read assemblers can be categorized into direct, hierarchical, and hybrid [18]. Direct assemblers try to assemble erroneous reads in a single overlap-consensus pass without trying to correct them. Examples of such assemblers include Celera [19] and the Minimap+Miniasm [11] pipeline, which uses minimizers to find multiple-sequence alignment (MSA). Once the erroneous reads are aligned, these assemblers merge the overlapped regions of the reads to create longer contiguous sequences known as *contigs*.

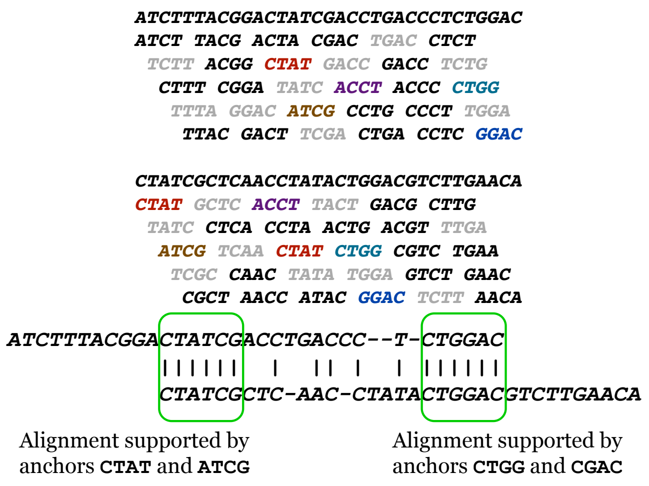

ATCTTTACGGACTATCGACCTGACCCTCTGGAC
ATCT TACG ACTA CGAC TGAC CTCT
TCTT ACGG CTAT GACC GACC TCTG
CTTT CGGA TATC ACCT ACCC CTGG
TTTA GGAC ATCG CCTG CCCT TGGA
TTAC GACT TCGA CTGA CCTC GGAC
CTATCGCTCAACCTATACTGGACGTCTTGAACA
CTAT GCTC ACCT TACT GACG CTTG
TATC CTCA CCTA ACTG ACGT TTGA
ATCG TCAA CTAT CTGG CGTC TGAA
TCGC CAAC TATA TGGA GTCT GAAC
CGCT AACC ATAC GGAC TCTT AACA
ATCTTTACGGACTATCGACCTGACCC--T-CTGGAC
CTATCGCTC-AAC-CTATACTGGACGTCTTGAACA
Alignment supported by anchors CTAT and ATCG
Alignment supported by anchors CTGG and CGAC

Figure 2: Alignment of noisy reads using minimizers

Finally, in the polishing phase, the matches, mismatches, and gaps in contigs are corrected using partial order alignment tools such as RACON [20].

On the other hand, some assemblers attempt to align shorter reads to the longer reads in the same dataset and use a consensus of shorter reads to correct the long ones. The alignment and consensus steps are typically performed multiple times to improve the quality of base calls, and so we can obtain contigs from high-quality long reads. Such assemblers are called hierarchical (or self-correction) assemblers and include HGAP [21], Falcon [22], and Canu [9] that uses the min-hash technique mentioned before.

Finally, the third type known as hybrid assemblers works on the same principle as hierarchical assemblers, but instead of using shorter reads from the same library, they use high-quality short reads from other datasets (e.g., second-generation sequencing) to correct the long reads. Examples of hybrid assemblers include HybridSPAdes [23], Cerulean [24], and dbg2olc [24].

## 2.1. Motivation and Goal

Both the hierarchical and hybrid strategies involve a computationally expensive base-level error-correction step consisting of multiple-sequence alignment and subsequent consensus generation. These require dynamic programming solutions such as Smith-Waterman which have a quadratic time complexity. Therefore, both these approaches require long execution times. On the other hand, direct approaches are faster, but their assemblies are noisy and must be corrected to make them usable. One way to do that is to re-align the noisy contigs with the original reads and then polish them using a consensus-generation tool. Indeed, this approach has been studied where Minimap was used to first align reads to each other and subsequently to the contigs produced by Miniasm. The contigs when polished by RACON produced high-quality assemblies an order of magnitude quicker than existing state-of-the-art assemblers [20].

Despite the rising popularity of third-generation sequencing and advances in long-read assembly, to the best of our knowledge, there is no distributed assembler available today. This poses a two-pronged problem for sequencing the larger datasets. Existing assemblers processing these datasets not only take days to finish but often run out of memory even on big-memory machines. Thus, even with access to a supercomputing cluster or other distributed computing infrastructure, practitioners may be unable to assemble and use long-read sequence datasets. Our goal in this work is to alleviate these issues by building a distributed long-read aligner and assembler. We extend the Minimap+Miniasm approach because not only is it fast and amenable to parallelization, but it can also produce high-quality contigs with the help of RACON.

## 3. Methodology

Hiper3ga is designed to run on a cluster of $n$ nodes and assumes the existence of a distributed filesystem (DFS) that is globally accessible to all nodes in the cluster. Each node $i$ runs a process $p_i$ (we use the terms *node* and *process* interchangeably) that (a) reads the input sequences from the DFS, (b) finds alignments among all sequences belonging to itself and its peers, (c) builds a graph using those alignments, (d) generates contigs, and (e) writes the contigs back to the DFS. Moreover, each node has local storage, which is used to temporarily store the sequence alignments before they are used to generate contigs.

Although the processes at each node are ranked, they work in a peer-to-peer as opposed to a manager-worker model during most of the execution. However, in certain scenarios involving a gather-scatter communication pattern, the process ranked 0 ($p_0$) acts as a manager, aggregating data from its peers and broadcasting the results. The process at each node runs multiple worker threads and has a master thread that is responsible for synchronizing them. The assembly pipeline consists of four main stages - 1) generating minimizers, 2) indexing minimizers and the locations from which they were generated, 3) aligning sequences, and 4) generating contigs. Fig. 3 depicts an architectural overview of the computation involved in these stages.

## 3.1. Overview of GASNet

In Hiper3ga, the spawning of processes on the nodes and all subsequent communication and synchronization between them is handled by GASNet [14] communication middleware that provides active message (AM) primitives using Remote Direct Memory Access (RDMA). It also provides an extended API for remote memory-copy (gets and puts) and a bootstrapping interface that can be used to launch jobs on clusters. During the bootstrap process, each node registers a user-specified amount of memory for communication. The address of this registered memory, henceforth called GASNet-attached segment, and the size of the memory segment for each node is made known to all other nodes.

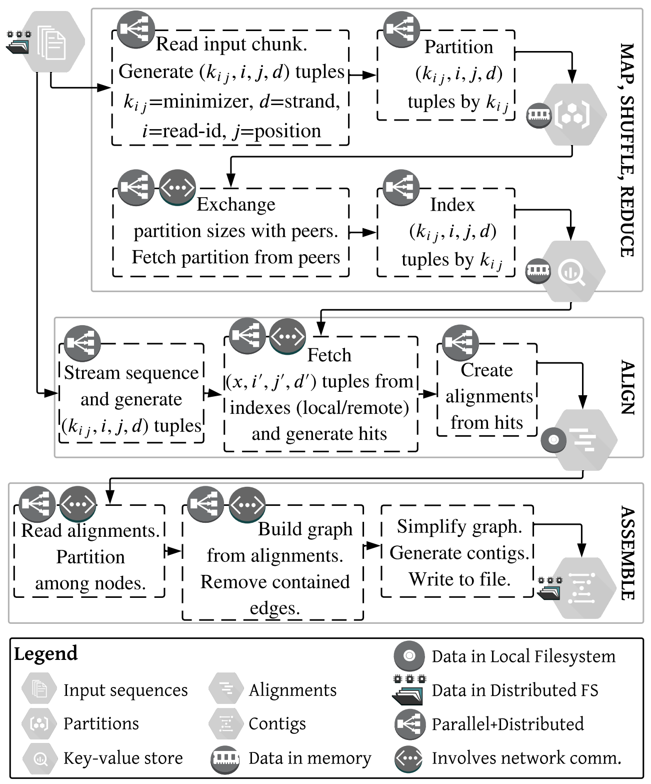


Figure 3: Overview of Hiper3ga

An AM in GASNet works as follows. Node A sends an AM with a handler function, a payload, and arguments to node B. Upon reaching B, the AM invokes the handler using the payload and arguments and optionally sends a reply to A with another AM. Depending on how the payload is copied to the destination node, AMs can be categorized as *MediumAM* or *LongAM*. *LongAM*s have larger payloads (up to 2GB on Infiniband networks) which must be copied directly to the destination node's GASNet-attached segment whereas *MediumAM*s have shorter payloads (less than 4KB on Infiniband) which are copied to memory allocated by the AM on the destination node. In case of remote-gets, the source data must lie in the GASNet-attached segment of the remote node, whereas in case of remote-puts, the entire destination region size must lie within the peer's attached segment. Unlike the longAMs, the size of a remote copy payload is limited only by the size of the registered memory segment.

## 3.2. Map: Generate Minimizers

In this phase, the input file is split into equal-sized chunks which are read from the DFS in parallel by the nodes. Each node in turn splits a chunk among multiple threads which obtain the line offsets in the file. These offsets are sent to the master which uses them to rearrange the chunk-boundaries such that a read is contained in its entirety within a single chunk. Next, each node re-streams the input chunks and sends it along with the lengths and offsets of the sequences in the chunk to a thread-pool for processing. The threads in the thread-pool work on the sequences in parallel and for each sequence, a thread generates a list of 4-tuples $(k_{ij}, i, j, d)$ where $k_{ij}$ is the minimizer obtained from sequence $s_i$ at position $j$ on strand $d \in \{0, 1\}$ (original or Watson-Crick complement). After a chunk is processed, the tuples are bit-packed in parallel and stored in blocks before moving on to the next chunk. We have observed that this reduces the data to about 55% of its original size. Fig. 4 shows the data generated by two nodes from two sequences that were previously introduced in Fig. 2. The tuples highlighted in red are scheduled to be transferred to the other node in the subsequent phase. Note that the strand information is omitted for brevity.

| **Node 0** | **Node 1** |
|---|---|
| **Input (Read-id=0):** ATCTTTACGGACTATCGACCTGACCCTCTGGAC | **Input (Read-id=1):** CTATCGCTCAACCTATACTGGACGTCTTGAACA |
| **Output:** | **Output:** |
| ATCT-(0,0) CTTT-(0,2) TTAC-(0,4) | CTAT-(1,0) ATCG-(1,2) CGCT-(1,4) |
| TACG-(0,5) ACGG-(0,6) CGGA-(0,7) | CTCA-(1,6) CAAC-(1,8) AACC-(1,9) |
| GACT-(0,9) ACTA-(0,10) CTAT-(0,11) | ACCT-(1,10) CCTA-(1,11) CTAT-(1,12) |
| ATCG-(0,13) CGAC-(0,15) ACCT-(0,17) | ATAC-(1,14) ACTG-(1,16) CTGG-(1,17) |
| CCTG-(0,18) CTGA-(0,19) GACC-(0,21) | GGAC-(1,19) GACG-(1,20) ACGT-(1,21) |
| ACCC-(0,22) CCCT-(0,23) CCTC-(0,24) | CGTC-(1,22) GTCT-(1,23) CTTG-(1,25) |
| CTCT-(0,25) CTGG-(0,27) GGAC-(0,29) | TGAA-(1,27) GAAC-(1,28) AACA-(1,29) |

Figure 4: Generating (minimizer,read-id,position)-tuples from reads.

## 3.3. Shuffle: Distribute minimizers among nodes

During the map phase, different nodes may generate the same minimizer from different sequences. These minimizers and their corresponding addresses composed of sequence-ids, positions, and strands must be aggregated so that they can be used as anchors when aligning their corresponding sequences. The first step in this process is to map the minimizers into one of $n$ nodes based on some function $g_1$ and the second step is to distribute the minimizer tuples such that for every minimizer $k_j$ residing in node $i$, $g_1(k_j) = i$. In figure 4, the minimizers are mapped between nodes 0 and 1 based on their last bases. Minimizers ending with bases A and T are mapped to node 0 and those ending with C and G are mapped to node 1. Fig. 5 shows the minimizers after they have been transferred to the nodes to which they were mapped. The partitioning and all-to-all transfers of minimizer tuples are performed as follows.

**3.3.1. Partition.** In this step, each bit-packed data block generated at node $p_i$ during the map phase is assigned a worker thread from $p_i$'s thread pool. Each worker thread $j$ consumes the blocks assigned to it one by one and appends the output to $n$ lists $\mathcal{L}_{j_1}, \mathcal{L}_{j_2}, \cdots, \mathcal{L}_{j_n}$, one for each node in the cluster. For any block, a thread unpacks it and splits the tuples in it into $n$ partitions by performing a radix sort on $g_1(x)$ for every minimizer $x$ in the data block and obtaining the partition offsets. These partitions are bit-packed and appended into $n$ lists maintained by the thread. At the end of this step, node $p_i$ ends up with $t \times n$ lists where each list

| Node 0 | Node 1 |
|---|---|
| **Locally generated:** ATCT-(0,0) CTTT-(0,2) CGGA-(0,7) GACT-(0,9) ACTA-(0,10) CTAT-(0,11) ACCT-(0,17) CTGA-(0,19) CCCT-(0,23) CTCT-(0,25) | **Locally generated:** ATCG-(1,2) CAAC-(1,8) AACC-(1,9) ATAC-(1,14) ACTG-(1,16) CTGG-(1,17) GGAC-(1,19) GACG-(1,20) CGTC-(1,22) CTTG-(1,25) GAAC-(1,28) |
| **Shuffled:** CTAT-(1,0) CGCT-(1,4) CTCA-(1,6) ACCT-(1,10) CCTA-(1,11) CTAT-(1,12) ACGT-(1,21) GTCT-(1,23) TGAA-(1,27) AACA-(1,29) | **Shuffled:** TACG-(0,5) TTAC-(0,4) ACGG-(0,6) ATCG-(0,13) CGAC-(0,15) CCTG-(0,18) GACC-(0,21) ACCC-(0,22) CCTC-(0,24) CTGG-(0,27) GGAC-(0,29) |

Figure 5: Shuffling data between nodes based on minimizers

$\mathcal{L}_{j\,k}, j \in [1,t], k \in [1,n]$ is composed of bit-packed blocks of tuples generated at node $i$ but destined for node $k$.

**3.3.2. Transfer.** Once all the nodes have finished partitioning the minimizers based on their destination nodes, Hiper3ga commences the data transfer between all node-pairs using a one-to-one personalized communication with E-cube routing [27]. In an $n$-node cluster, the transfer is achieved in $n-1$ rounds with inter-node synchronizations between them. At round $q$, each node $i \in [1,n]$ pairs up with node $j = i \oplus q$ and exchanges data with it as follows.

In the beginning, each node $i$ creates a local list $\mathcal{O}_i$ to store the shuffled data, and the address of this list is broadcast to every other node in the cluster. Subsequently, at every round, for every compressed data block in the list $\mathcal{L}_{xj} \forall x \in [1,t]$, node $i$ asynchronously copies the block to node $j$'s GASNet-attached segment using extended APIs. If it cannot send any more data either due to lack of space in node $j$'s segment or because there are no more blocks to send, node $i$ waits for all asynchronous transfers to finish and sends a *MediumAM* to node $j$ with the offsets of the transferred data. The *MediumAM* on reaching node $j$ invokes its handler which uses the message offsets in the payload to copy data from node $j$'s attached segment to the output list $\mathcal{O}_j$ and reports back to node $i$ with an acknowledgement. Node $i$ continues this process until all data blocks are copied to node $j$. After all remote transfers are over, each node $i$ moves the remaining data (i.e., partitions belonging to itself) to the list $\mathcal{O}_i$ and finishes. Ultimately, each node $i$ ends up with a single list $\mathcal{O}_i$ of bit-packed data-blocks where every block contains tuples where the minimizer $x$ satisfies $g_1(x) = i$.

## 3.4. Reduce: Index Minimizers

After the shuffle phase, the minimizer tuples at each node are stored in a local index using minimizers as keys. The value corresponding to a key (minimizer) in the index is a list of (sequence-id, position, strand) tuples from which the it was generated. The index is composed of multiple shards, each of which is an independent hash table and can be updated in parallel to others. The process of distributing tuples among shards is similar to that in section 3.3.1 and involves mapping a minimizer $x$ to shard $q$ using a function $g_2$ such that $g_2(x) = q$. Since tuples are partitioned and shuffled on the basis of their minimizers, the ones with the same minimizer $x$ will always reside in the same node $j = g_1(x)$ and can be obtained by other nodes from the index at node $j$. Similarly, the shard in which a key-value pair resides can be obtained from the value of $g_2(x)$. Once the tuples are distributed, each shard is processed by a thread that sorts them by their minimizers. Next, for each shard the number of unique minimizers is calculated, a hash table is pre-allocated, and the key-value pairs are inserted into it.

The index at each node together with GASNet's communication primitives is used to simulate a read-only distributed hash-map. A client searching for a minimizer first finds the node responsible for it and subsequently the shard to which the minimizer belongs. Next, it sends an AM with the key (minimizer) to the destination node, which then invokes the handler which in turn gets the corresponding value from the index and sends it back to the caller. Note that in practice, the requests are sent in batches in order to improve the network throughput. Fig. 6 shows the indexed minimizers for the example in Fig. 4. The minimizers highlighted in different colors are the ones that have more than one occurrence.

| Node 0 | Node 1 |
|---|---|
| AACA-{(1,29)} ACCT-{(0,17)(1,10)} ACGT-{(1,21)} ACTA-{(0,10)} ATCT-{(0,0)} CCCT-{(0,23)} CCTA-{(1,11)} CGCT-{(1,4)} CGGA-{(0,7)} CTAT-{(0,11)(1,0)(1,12)} CTCA-{(1,6)} CTCT-{(0,25)} CTGA-{(0,19)} CTTT-{(0,2)} GACT-{(0,9)} GTCT-{(1,23)} TACG-{(0,5)} TGAA-{(1,27)} | AACC-{(1,9)} ACCC-{(0,22)} ACGG-{(0,6)} ACTG-{(1,16)} ATAC-{(1,14)} ATCG-{(1,2)(0,13)} CAAC-{(1,8)} CCTC-{(0,24)} CCTG-{(0,18)} CGAC-{(0,15)} CGTC-{(1,22)} CTGG-{(1,17)(0,27)} CTTG-{(1,25)} GAAC-{(1,28)} GACC-{(0,21)} GACG-{(1,20)} GGAC-{(1,19)(0,29)} TTAC-{(0,4)} |

Figure 6: Indexing read-id and position tuples by minimizers.

## 3.5. Aligning Sequences

In this phase, each node streams chunks of input sequences (like they do in the map phase), which are processed in parallel by multiple threads. For each query sequence $s_i$, the thread responsible for it generates a list of its minimizer tuples $(x, i, j, d)$ where $x$ is the minimizer hash value, $i$ is the sequence-id, $j$ is the position, and $d$ is the strand in $s_i$ from which $x$ was obtained. For each tuple in the list, it then checks for occurrences (hits) of its minimizer in other sequences (i.e., for tuples of the form $(x, i', j', d')$) in the distributed hash-map. A hit is stored as a 4-tuple $(i', y, a, j')$ where $i'$ is the id of the target sequence, $y$ is the strand relative to $d$ where the minimizer appears ($y = d \oplus d'$), $j'$ is the position in sequence $i'$ where the minimizer appears, and $a$ is the relative alignment of sequence $i'$ with sequence $i$ ($j' - j$ if $y = 0$, $j + j'$ otherwise). These tuples are then sorted by their sequence-ids followed by their relative alignments. This sorted list is subsequently used to find out if a target sequence $s_{i'}$ has multiple hits with query sequence $s_i$ and where those hits are located relative $s_i$.

Like Minimap, the hits between two sequences are clustered using Hough transformation [28] in order to use them as anchors in the alignment of the two sequences. Informally, a hit between sequences $s_i$ and $s_{i'}$ in positions $j$

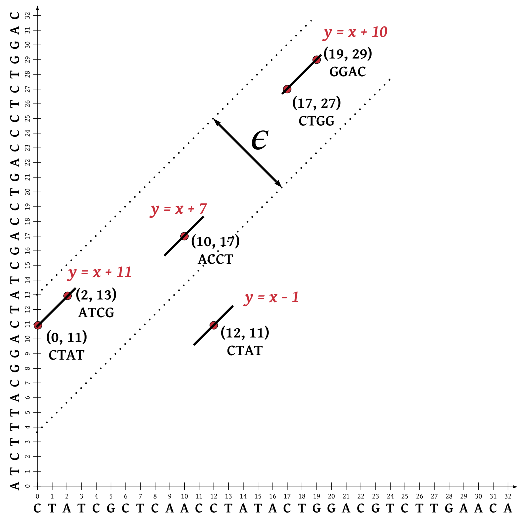


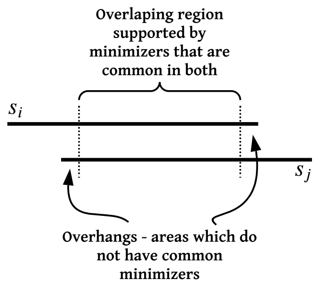
Figure 7: Binning common minimizers using Hough Transform

and $j'$ can be represented as a point $(j, j')$ in a 2-dimensional plane. If $s_i$ and $s_{i'}$ have a significant overlap with each other, they will produce several hits, all of which will be represented as points in the same plane. Furthermore, if the overlapping region is error-free (i.e., both $s_i$ and $s_{i'}$ contain the same substring $o$), all the hits generated from this region will lie on a straight line. This line will be inclined at a 45° angle since for each unique minimizer $x$ generated from $o$, the distance between positions $j$ and $j'$ where $x$ appears in sequences $s_i$ and $s_{i'}$ will be constant.

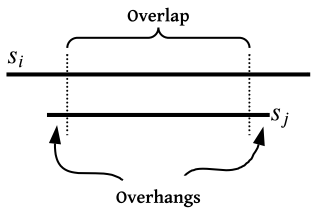
However, if there are errors the hits will not form an exact straight line, but will be clustered around one (approximately collinear). Two minimizer hits $(i', y_1, a_1, j'_1)$ and $(i', y_2, a_2, j'_2)$ between query sequence $s_i$ and target sequence $s_{i'}$ are said to be approximately colinear within a bandwidth $\epsilon$ (500bp by default as suggested by Minimap) if $|a_1 - a_2| < \epsilon$. Once the collinear hits are obtained, Hough Transformation is used to find a straight line such that a majority of the minimizers lie on it. Since the slope is 1, the line takes the form $y = x + c \implies c = y - x$. For each hit, we can calculate the intercept $c$ and use the value supported by the majority of the points. This line is used as a crude alignment of the sequences. Fig. 7 shows the common minimizers generated from the reads in Fig. 2, which are represented in a 2D plane. The lines $y = x + 11$ and $y = x + 10$ are supported by two minimizers each, whereas the lines $y = x + 7$ and $y = x - 1$ are supported by one minimizer each. The hit at $(12, 11)$ is not collinear with the others and is not considered in the Hough Transform.

## 3.6. Assembly

The assembly phase can be split into two sub-phases - graph building and graph simplification. The first step is to distribute the set of alignments created in the previous phase between the different nodes. This is performed using the shuffle method explained in section 3.3. Once the alignments are shuffled among nodes, they are further partitioned among the threads so that they can be processed in parallel. The inter-node and inter-thread partitioning is done based on some function $g_3$ which is applied to the query read-ids of alignments. Specifically, an alignment $a$ is mapped to a partition $i$ if $g_3(q_a) = i$ where $q_a$ is the query read-id of $a$.


Overlaping region supported by minimizers that are common in both
$s_i$
$s_j$
Overhangs - areas which do not have common minimizers

(a) Overlap and overhangs between two sequences



(b) One sequence contained in the other

Figure 8: Types of overlaps

Each thread sorts the list of alignments pairs by their query read-ids and for each unique query, it classifies the alignments with other reads (targets) into five categories -
1.) Internal overlaps - if the target read has regions that are aligned to the query and there are regions on either side that are not aligned (overhangs) such that they are longer than some threshold, the overlap is classified as internal. In other words, the overlap is considered to have arisen from repeats rather than from sequencing the same region of the genome. Internal overlaps are ignored since they do not contribute to read extension. In figure 8a, if the overhangs are larger than some threshold, it is considered an internal overlap.
2.) Query read contained - if the query read is completely contained in one of the target reads (i.e., the overlapping region of the query spans almost the entire read, and the overhang regions are small enough), the query read is marked for deletion. The thread processing the alignment uses an atomic bit-vector to set the bit corresponding to the query read-id to 1. In figure 8b, read $s_j$ will be considered to be contained in read $s_i$ provided $s_j$'s overhanging regions are smaller than some threshold.
3.) Target read contained - like in the previous case, if the target is contained in the query, it is marked for deletion.
4.) Query-suffix target-prefix overlap - if the suffix of the query read has a significant overlap with the prefix of a target read and the overhangs are shorter than some threshold, a (query, target, overlap length) tuple is created which corresponds to a weighted edge in the assembly graph.
5.) Query-prefix target-suffix overlap - if the prefix of the query read has a significant overlap with the suffix of the target read, we create a (target, query, overlap length) triple.

**3.6.1. Aggregating deleted reads.** Once the hits are classified by all nodes, and the contained reads are marked for deletion, the atomic bit-vectors from all nodes are gathered by the master node, aggregated, and scattered back to the peers. The gather-scatter process works as follows. Each node copies its bit-vector into the GASNet-attached segment and performs a barrier synchronization. Node $p_0$ performs a remote-memory-get of $r$ bits ($r$ is the number of reads) from each peer and performs a bitwise-OR of the elements in its local vector with those of the remote vector. Once all data from all nodes are aggregated, $p_0$ performs remote-memory-puts to all peers, followed by a barrier synchronization.

**3.6.2. Deleting edges connecting deleted reads.** When the nodes get the updated set of deleted reads, they remove any suffix-prefix overlaps where either the suffix or the prefix or both belong to one of the deleted reads. Since most of the popular third-generation sequencers have a long-tail distribution of read lengths (as shown later in Section 4.1), it is highly likely that most of the smaller reads will be contained within a few longer ones. Hence, removal of contained reads significantly reduces the number of edges in the final graph by many folds. This makes it possible for downstream phases to process the graph on a single machine and avoid inter-node communication and synchronization.

**3.6.3. Creating an assembly graph.** After edges originating from and/or incident upon contained reads are removed, all nodes send their remaining edge list to the master node. Each peer copies the number of edges and the list of edges into their GASNet-attached segment, which is fetched by the master using remote-memory-gets. Next, the master creates a graph and sanitizes it using two rules - (1) if two vertices (reads) have more than one edges (overlaps), it chooses the one with the highest weight (longest overlap) and deletes the rest of the edges; (2) if an edge exists from vertex $u$ to $v$ but not from $v'$ to $u'$ (i.e., the Watson-Crick complements of $u$ and $v$ respectively), the edge from $u$ to $v$ is deleted.

Once the graph is made consistent, its edges are transitively reduced whenever possible. For any 3 vertices $u$,$v$ and $w$, if there exist edges $u \rightarrow v$, $v \rightarrow w$ and $u \rightarrow w$, we can remove the edge $u \rightarrow w$ and its complement edge $u' \rightarrow w'$ without affecting the connectivity of the graph. Moreover, in order to handle repeats, if some vertex $v$ has multiple outgoing edges $E_v$, and there exists a subset of edges $E'_v \in E_v$ with weight less than some fraction (.5 by default) of the weight of the heaviest edge in $E_v$, all edges in $E'_v$ are removed [11]. Both the steps mentioned above help reduce the number of branches in the graph and simplify it for the subsequent unitig-generation step.

**3.6.4. Simplifying graph and generating unitigs.** The assembly graph $G$ is used to create a set $U$ of *unitigs* where each $u \in U$ is a sequence generated by traversing a path in $G$ where each vertex in the path has at most one incoming edge and at most one outgoing edge. Unitigs are created by concatenating the reads corresponding to the vertices in the path and merging their overlapping regions. Intuitively, each unitig is formed by merging two or more overlapping reads in the original dataset. In the ideal scenario, we would have a single unitig representing the genome from which the original reads in the dataset were sequenced. However, it is possible that subgraphs within the graph are not reachable from one another due to low coverage regions formed as a result of sequencer bias. Moreover, the graph often contains vertices that have more than one incoming edge (joins) or outgoing edge (forks) which results in an ambiguity in path selection during unitig generation.

Some forks and joins can form due to sequencing errors and can be categorized as either *tips* or *bubbles*. A tip is a short path consisting of at most 4 vertices by default (the last vertex in the path has no outgoing edge), and originating from some vertex with out-degree greater than 1. A bubble is comprised of two or more paths between two vertices $u$ and $v$ such that each path is an unitig and shorter than some threshold (50k bases by default). Bubbles are popped by keeping the longest path between the source and sink vertices and deleting all others. The tip-removal, bubble-popping, and simplification phases are performed iteratively until the graph cannot be simplified any further.

Note that each unitig generated in this method is represented as a list $p$ of tuples, where the $i^{\text{th}}$ tuple $(j_i, l_i)$ consists of $j_i$, the read-id and $l_i$, the length of the prefix of read $j_i$. However, the unitig in this form is not usable for downstream applications. Instead, each $(j_i, l_i)$ tuple in each path must be replaced by the actual $l_i$-length prefix of the read $j_i$ and concatenated in order of their occurrence in the path [30]. In order to do this using a single disk pass, the lengths and offsets of all paths are calculated and for each tuple in the path, we calculate and store the prefix length and its offset within the path. Next, the reads along with the lists of their prefix-lengths and offsets are indexed using the read-id as the key. Finally, the reads are streamed from disk, and for each read, we place the prefixes at the appropriate offsets. In an edge from read $u$ to read $v$, the left overhang is simply replaced by the prefix of $u$, and the right overhang is replaced by the prefix of $v$ since it is assumed that base-level errors will be corrected in the polishing phase. When all reads are processed, the resulting strings are then written back to the filesystem.

## 4. Evaluation

In this section, we report the experimental results of Hiper3ga for a wide range of real-world sequence datasets across several cluster configurations.

### 4.1. Datasets and Testbed

TABLE 1: Overview of datasets

| Name | Accn# | # Reads | # Bases | Longest | Avg. |
|---|---|---|---|---|---|
| Oyster | SRR5986177 | 163473 | 2.697 B | 119258 | 16498 |
| Salmonella | SRR7415636 | 737652 | 4.970 B | 81671 | 6737 |
| Banana | SRR7013757 | 896215 | 9.221 B | 76620 | 10288 |
| ValleyOak | SRR6300394 | 6343902 | 17.605 B | 67712 | 2775 |
| Beaver | SRR5173103 | 2387048 | 35.240 B | 19999 | 14762 |
| KissingBug | SRR8466736 | 8228076 | 69.384 B | 127888 | 8432 |
| Human | ERR2808248 | 7875243 | 138.976 B | 115188 | 17647 |

TABLE 2: Comparison of assembly times
*Miniasm failed on 64 GB RAM and was run on a fat node with 256 GB RAM

| Dataset | Minimap+Miniasm | Hiper3ga | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 1 node | 1 node | 2 nodes | 4 nodes | 8 nodes | 16 nodes | 32 nodes | 64 nodes | 128 nodes |
| Human* | 1d 14h 35m 37s | | | | | | 2h 15m 28s | 57m 5s | 32m 39s |
| KissingBug* | 12h 23m 15s | | | | | 53m 26s | 33m 42s | 16m 41s | 9m 46s |
| Beaver | 3h 4m 32s | | | | 27m 20s | 13m 58s | 9m 5s | 4m 47s | 3m 5s |
| ValleyOak | 1h 0m 39s | | | 16m 24s | 8m 51s | 4m 59s | 3m 1s | 2m 14s | 1m 20s |
| Banana | 17m 7s | 12m 49s | 9m 38s | 4m 24s | 2m 31s | 1m 31s | 1m 0s | 30s | 21s |
| Salmonella | 18m 50s | 8m 57s | 5m 53s | 3m 1s | 1m 33s | 1m 4s | 40s | 24s | 18s |
| Oyster | 2m 19s | 2m 32s | 1m 45s | 54s | 31s | 18s | 13s | 10s | 8s |

To evaluate our assembly approach, we use publicly available *PacBio* datasets of several species and of various sizes. The sources and an overview of the characteristics of the datasets are shown in Table 1. The smallest one is the Oyster genome containing about 2.7 billion bases in about 163 thousand reads whereas the largest dataset is the Human genome with about 139 billion bases in 7.88 million reads. The datasets are chosen such that the number of bases roughly double in size from one to the next larger. For all the datasets, the read lengths have a long-tail distribution with a few very long reads and a large number of shorter reads. These shorter reads are generally contained within the longer ones and can be used for subsequent consensus generation. All experiments for both Hiper3ga and Minimap were conducted with a $k$-mer size of 15 and a minimizer-window size of 5 as suggested by Minimap.

To compare the execution times Hiper3ga against Minimap and Miniasm, we use up to 128 nodes (the maximum allowed for a job) of the QB2 [2] supercomputing cluster. Each node in QB2 has two 2.8GHz 10-Core Intel Xeon E5-2680v2 processors, 64 GB RAM, and a 500 GB local hard disk drive. The nodes are connected by a 56 Gigabit/sec Infiniband network and have access to a 2.8 PB Lustre distributed filesystem. To test Miniasm on the larger datasets, we use big-memory nodes in SuperMIC [3] cluster which are composed of 256 GB RAM and two 2.6GHz 14-Core Intel Xeon E5-2690v4 processors.

## 4.2. Comparison of Assembly Times

Table 2 provides a comparative analysis between Minimap+Miniasm and Hiper3ga with respect to all-vs-all sequence alignment and subsequent assembly. Even on the smallest dataset (Oyster), Hiper3ga's performance on a single node is comparable to that of Minimap+Miniasm whereas, in 2 and 4 nodes, it achieves a 1.32x and 2.57x speedup respectively. On the Salmonella dataset, Hiper3ga performs significantly better even on a single node achieving more than 2x speedup. This is due to two reasons - 1) Miniasm is a single-threaded process while Hiper3ga improves upon it by using a fully parallel (and distributed) computation model and 2) Minimap performs several disk I/O passes over the data if it is too large to fit in memory. Specifically, Minimap maintains two file pointers to the input data, one of which is used to read chunks of data, generate minimizers and index them, while the other is used to stream all reads and align them to the indexed minimizers in the chunk. In case of Salmonella, Minimap requires two disk passes whereas Hiper3ga is able to generate and index all minimizers in the memory of a single node by compressing the intermediate data to about half its size. The same applies in case of the Banana dataset where Hiper3ga is 1.34x faster on a single node. Moreover, the assembly phase for the Salmonella dataset takes an uncharacteristically long time (as can be seen in figure 9) which puts Miniasm at a disadvantage compared to Hiper3ga.

In case of the smaller datasets, the speedup saturates in the larger cluster configurations since the inter-node communication costs dominate the total execution time. However, the time difference is starker on larger datasets using more cluster nodes. For instance, on the KissingBug and Human genomes processed using 128 nodes, Hiper3ga achieves a 76.1x and 70.9x speedup respectively compared to Minimap+Miniasm running on 1 node. As mentioned before, this comes down to the Minimap's high disk access cost (it requires 35 disk passes to process the Human dataset) and Miniasm's lack of parallel processing capability. Moreover, because it is limited by the memory capacity of a single node, Miniasm runs out of memory for the KissingBug and Human genome datasets on a thin (64 GB RAM) node and had to be run on a fat node with 256 GB memory. Note that Hiper3ga is not evaluated on the larger datasets on a smaller number of nodes because the number of minimizers is too large to index in memory.

## 4.3. Comparison of Assembly Contiguity

Table 3 shows a comparison of assembly contiguity between Hiper3ga and Miniasm. Along with metrics like the number of unitigs, their total length, and the length of the longest contig, we also report the N50 values where an N50 value of $x$ signifies that each of the longest contigs that together cover more than or equal to 50% of the total assembly size are longer than $x$. It is evident from Table 3 that Hiper3ga's assembly contiguity is comparable to that of Miniasm. The differences that occur in the results are due to the difference in how Minimap and Hiper3ga discard the most frequently occurring minimizers from the alignment

2. http://www.hpc.lsu.edu/resources/hpc/system.php?system=QB2
3. http://www.hpc.lsu.edu/resources/hpc/system.php?system=SuperMIC

TABLE 3: Comparison of assembly contiguity. H3 = Hiper3ga, MM = Miniasm.

| | #Unitigs | | Total Length | | Longest | | N50 | | Median | |
|---|---|---|---|---|---|---|---|---|---|---|
| | H3 | MM | H3 | MM | H3 | MM | H3 | MM | H3 | MM |
| Oyster | 11 | 7 | 167.4 Kbp | 93.1 Kbp | 37.5 Kbp | 25.2 Kbp | 21.8 Kbp | 16.3 Kbp | 16.1 Kbp | 15.2 Kbp |
| Salmonella | 2 | 4 | 4.93 Mbp | 5.01 Mbp | 4.877 Mbp | 4.92 Mbp | 4.877 Mbp | 4.92 Mbp | 2.464 Mbp | 38.36 Kbp |
| Banana | 4205 | 4245 | 220.486 Mbp | 236.415 Mbp | 482.2 Kbp | 552.67 Kbp | 61.6 Kbp | 65.31 Kbp | 42.6 Kbp | 45.1 Kbp |
| ValleyOak | 6759 | 7066 | 541.882 Mbp | 561.158 Mbp | 553.6 Kbp | 629.9 Kbp | 109.2 Kbp | 108.441 Kbp | 61.6 Kbp | 61.1 Kbp |
| Beaver | 34253 | 33902 | 1.911 Gbp | 1.965 Gbp | 565.4 Kbp | 596.4 Kbp | 68.0 Kbp | 72.2 Kbp | 43.7 Kbp | 44.9 Kbp |
| KissingBug | 2114 | 2044 | 679.146 Mbp | 677.180 Mbp | 6.525 Mbp | 5.255 Mbp | 995.5 Kbp | 1.077 Mbp | 78.1 Kbp | 78.1 Kbp |
| Human | 3818 | 3601 | 2.966 Gbp | 2.962 Mbp | 22.127 Mbp | 22.131 Mbp | 3.750 Mbp | 3.925 Mbp | 88.4 Kbp | 90.7 Kbp |

process. Specifically, while Minimap discards 0.1% of the most frequently occurring minimizers from each chunk of input, Hiper3ga does so using 0.1% of the most frequent minimizers in the entire dataset.

## 4.4. Assembly Scalability

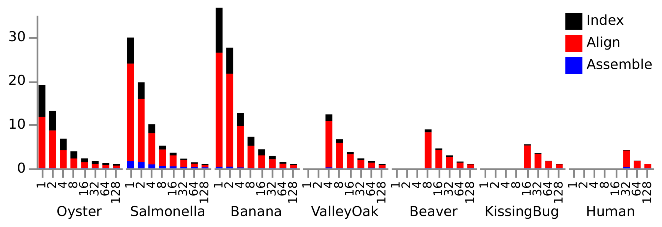


Figure 9: Strong scalability of Hiper3ga from 1 to 128 nodes

Fig. 9 shows the scalability of Hiper3ga on different datasets using various cluster configurations. To show the relative performance of Hiper3ga on the different configurations, we normalize the execution times for each dataset by scaling the 128-node execution time to 1. It can be seen that for the smallest dataset (Oyster), the execution is sped up by a factor of 1.72x-1.94x till 16 nodes after which it decreases to about 1.3x-1.4x when the inter-node communication dominates the execution time. For slightly larger datasets such as Salmonella and Banana, a 1.5x-2.0x speedup is sustained till 64 nodes. In case of the three largest datasets that we tested (Beaver, KissingBug, and Human), the 1.5x-2.0x speedup can be observed till 128 nodes. Note that the speedup obtained on 2 nodes from 1 node in case of Oyster, Salmonella, and Banana is slightly reduced (1.3x-1.5x) due to the introduction of internode communication.

In some cases, we can see a 2x or more speedup while doubling the number of nodes. This is because Hiper3ga allocates large contiguous chunks of memory that act as buffers to reduce the number of messages between nodes by increasing the message size, and the buffer sizes are larger with fewer nodes, resulting in stalls if the memory is too fragmented. These stalls sometimes offset the higher communication costs incurred on larger cluster configurations. Instances of this scenario can be seen in the Human and KissingBug genomes where the speedups achieved going from 32 nodes to 64 nodes are 2.37x and 2.02x respectively.

Fig. 9 also shows the contribution of each phase towards the total execution time. The Index phase (which in this figure also includes the Map and Shuffle phases) is the most scalable part of the pipeline because it requires a minimal number of communication steps. Although the Shuffle phase involves a lot of data transfer, the communication pattern is characterized by small numbers of very large payloads which GASNet handles efficiently, often reaching peak throughputs of 2 to 3 gigabytes per second (16-24 Gbps). Evidently, the bulk of the execution time is taken by the Align phase which is not only the most compute-intensive but is also the most communication-intensive step in the pipeline. The assembly phase is the least expensive in the pipeline and executes in a matter of seconds.

## 5. Conclusion

In this paper, we present Hiper3ga, a distributed long-read assembler, inspired by Minimap+Miniasm, which can assemble large datasets generated from third-generation sequences on multiple nodes in a matter of minutes. Our experimental results demonstrate that Hiper3ga can scale up to hundreds of nodes and provide about two orders of magnitude performance improvements over the state-of-the-art long-read assembler. Moreover, Hiper3ga's contigs are comparable in contiguity to Miniasm. As future work, we plan to improve contig quality by including the base-level error correction phase using a distributed Partial-Order-Alignment-based consensus generation with hierarchical or hybrid strategies. Moreover, we are working on reducing the execution time even further by using the recently released GASNet-Ex communication middleware, which significantly improves upon the GASNet-1 system that we use in Hiper3ga.

## Acknowledgements

This work was partially funded by NIH grants (P20GM103458-10, P30GM110760-03, P20GM103424), NSF grants (IBSS-L-1620451, SCC-1737557, RAPID-1762600), and LA Board of Regents grants (LEQSF(2016-19)-RD-A-08 and ITRS).

## References

[1] C. McNeil, “Nci-match launch highlights new trial design in precision-medicine era,” *JNCI: Journal of the National Cancer Institute*, vol. 107, no. 7, 2015.

[2] M. W. Allard, Y. Luo, E. Strain, J. Pettengill, R. Timme, C. Wang, C. Li, C. E. Keys, J. Zheng, R. Stones *et al.*, “On the evolutionary history, population genetics and diversity among isolates of salmonella enteritidis pfge pattern jegx01. 0004,” *PloS one*, vol. 8, no. 1, p. e55254, 2013.

[3] E. K. Lienau, E. Strain, C. Wang, J. Zheng, A. R. Ottesen, C. E. Keys, T. S. Hammack, S. M. Musser, E. W. Brown, M. W. Allard *et al.*, “Identification of a salmonellosis outbreak by means of molecular sequencing,” *New England Journal of Medicine*, vol. 364, no. 10, pp. 981–982, 2011.

[4] B. R. Jackson, C. Tarr, E. Strain, K. A. Jackson, A. Conrad, H. Carleton, L. S. Katz, S. Stroika, L. H. Gould, R. K. Mody *et al.*, “Implementation of nationwide real-time whole-genome sequencing to enhance listeriosis outbreak detection and investigation,” *Reviews of Infectious Diseases*, vol. 63, no. 3, pp. 380–386, 2016.

[5] S. R. Lockhart, K. A. Etienne, S. Vallabhaneni, J. Farooqi, A. Chowdhary, N. P. Govender, A. L. Colombo, B. Calvo, C. A. Cuomo, C. A. Desjardins *et al.*, “Simultaneous emergence of multidrug-resistant candida auris on 3 continents confirmed by whole-genome sequencing and epidemiological analyses,” *Clinical Infectious Diseases*, vol. 64, no. 2, pp. 134–140, 2016.

[6] Z. S. Ma, L. Li, C. Ye, M. Peng, and Y.-P. Zhang, “Hybrid assembly of ultra-long nanopore reads augmented with 10x-genomics contigs: Demonstrated with a human genome,” *Genomics*, 2018.

[7] K. Berlin, S. Koren, C.-S. Chin, J. P. Drake, J. M. Landolin, and A. M. Phillippy, “Assembling large genomes with single-molecule sequencing and locality-sensitive hashing,” *Nature biotechnology*, vol. 33, no. 6, p. 623, 2015.

[8] C. Ye, C. M. Hill, S. Wu, J. Ruan, and Z. S. Ma, “Dbg2olc: efficient assembly of large genomes using long erroneous reads of the third generation sequencing technologies,” *Scientific reports*, vol. 6, p. 31900, 2016.

[9] S. Koren, B. P. Walenz, K. Berlin, J. R. Miller, N. H. Bergman, and A. M. Phillippy, “Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation,” *Genome research*, vol. 27, no. 5, pp. 722–736, 2017.

[10] M. Chakraborty, J. G. Baldwin-Brown, A. D. Long, and J. Emerson, “Contiguous and accurate de novo assembly of metazoan genomes with modest long read coverage,” *Nucleic acids research*, vol. 44, no. 19, pp. e147–e147, 2016.

[11] H. Li, “Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences,” *Bioinformatics*, vol. 32, no. 14, pp. 2103–2110, 2016.

[12] M. Jain, S. Koren, K. H. Miga, J. Quick, A. C. Rand, T. A. Sasani, J. R. Tyson, A. D. Beggs, A. T. Dilthey, I. T. Fiddes *et al.*, “Nanopore sequencing and assembly of a human genome with ultra-long reads,” *Nature biotechnology*, vol. 36, no. 4, p. 338, 2018.

[13] S. Goswami, A. K. Das, R. Płatania, K. Lee, and S.-J. Park, “Lazer: Distributed memory-efficient assembly of large-scale genomes,” in *2016 IEEE International Conference on Big Data (Big Data)*. IEEE, 2016, pp. 1171–1181.

[14] D. Bonachea and P. Hargrove, “Gasnet specification, v1.8.1,” GASNet Specification, v1.8.1. Lawrence Berkeley National Laboratory. Report: LBNL-2001064, 2017, http://dx.doi.org/10.2172/1398512 Retrieved from https://escholarship.org/uc/item/03b5g0q4.

[15] N. Ahmed, K. Bertels, and Z. Al-Ars, “A comparison of seed-and-extend techniques in modern dna read alignment algorithms,” in *2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. IEEE, 2016, pp. 1421–1428.

[16] S. F. Altschul, T. L. Madden, A. A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman, “Gapped blast and psi-blast: a new generation of protein database search programs,” *Nucleic acids research*, vol. 25, no. 17, pp. 3389–3402, 1997.

[17] M. Roberts, W. Hayes, B. R. Hunt, S. M. Mount, and J. A. Yorke, “Reducing storage requirements for biological sequence comparison,” *Bioinformatics*, vol. 20, no. 18, pp. 3363–3369, 2004.

[18] S. Koren and A. M. Phillippy, “One chromosome, one contig: complete microbial genomes from long-read sequencing and assembly,” *Current opinion in microbiology*, vol. 23, pp. 110–120, 2015.

[19] E. W. Myers, G. G. Sutton, A. L. Delcher, I. M. Dew, D. P. Fasulo, M. J. Flanigan, S. A. Kravitz, C. M. Mobarry, K. H. Reinert, K. A. Remington *et al.*, “A whole-genome assembly of drosophila,” *Science*, vol. 287, no. 5461, pp. 2196–2204, 2000.

[20] R. Vaser, I. Sović, N. Nagarajan, and M. Šikić, “Fast and accurate de novo genome assembly from long uncorrected reads,” *Genome research*, vol. 27, no. 5, pp. 737–746, 2017.

[21] C.-S. Chin, D. H. Alexander, P. Marks, A. A. Klammer, J. Drake, C. Heiner, A. Clum, A. Copeland, J. Huddleston, E. E. Eichler *et al.*, “Nonhybrid, finished microbial genome assemblies from long-read smrt sequencing data,” *Nature methods*, vol. 10, no. 6, p. 563, 2013.

[22] C.-S. Chin, P. Peluso, F. J. Sedlazeck, M. Nattestad, G. T. Concepcion, A. Clum, C. Dunn, R. O'Malley, R. Figueroa-Balderas, A. Morales-Cruz *et al.*, “Phased diploid genome assembly with single-molecule real-time sequencing,” *Nature methods*, vol. 13, no. 12, p. 1050, 2016.

[23] D. Antipov, A. Korobeynikov, J. S. McLean, and P. A. Pevzner, “hybridspades: an algorithm for hybrid assembly of short and long reads,” *Bioinformatics*, vol. 32, no. 7, pp. 1009–1015, 2015.

[24] V. Deshpande, E. D. Fung, S. Pham, and V. Bafna, “Cerulean: A hybrid assembly using high throughput short and long reads,” in *International workshop on algorithms in bioinformatics*. Springer, 2013, pp. 349–363.

[25] A. K. Das, S. Goswami, K. Lee, and S.-J. Park, “A hybrid and scalable error correction algorithm for indel and substitution errors of long reads,” *BMC genomics*, vol. 20, no. 11, pp. 1–15, 2019.

[26] S. Goswami, A. Pokhrel, K. Lee, L. Liu, Q. Zhang, and Y. Zhou, “Graphmap: scalable iterative graph processing using nosql,” *The Journal of Supercomputing*, pp. 1–29, 2019.

[27] Dally and Seitz, “Deadlock-free message routing in multiprocessor interconnection networks,” *IEEE Transactions on Computers*, vol. C-36, no. 5, pp. 547–553, 1987.

[28] I. Sović, M. Šikić, A. Wilm, S. N. Fenlon, S. Chen, and N. Nagarajan, “Fast and sensitive mapping of nanopore sequencing reads with graphmap,” *Nature communications*, vol. 7, p. 11307, 2016.

[29] P. Koppa, A. Das, S. Goswami, R. Platania, and S. Park, “Giga: Giraph-based genome assembler for gigabase scale genomes,” in *Proceedings of the 8th International Conference on Bioinformatics and Computational Biology (BICOB 2016)*, 2016, pp. 55–63.

[30] S. Goswami, K. Lee, S. Shams, and S.-J. Park, “Gpu-accelerated large-scale genome assembly,” in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2018, pp. 814–824.