MERmaid: A Parallel Genome Assembler for the Cloud

Richard Xia
University of California, Berkeley
Parallel Computing Laboratory
rxia@eecs.berkeley.edu

Albert Kim
University of California, Berkeley
Parallel Computing Laboratory
alkim@eecs.berkeley.edu

Abstract

Modern genome sequencers are capable of producing millions to billions of short reads of DNA. Each new generation of genome sequencers is able to provide an order of magnitude more data than the previous, resulting in an exponential increase in required data processing throughput. The challenge today is to build a software genome assembler that is highly parallel, fast, and inexpensive to run. We present MERmaid, a whole-genome assembler which is highly parallelized and runs efficiently on commodity clusters. Our assembler is based off the Meraculous whole genome assembler, which uses quality information about the input data in place of an explicit error correction step [7]. We analyzed and implemented a number of techniques to lower memory usage, reduce communication bandwidth, and allow for scaling up to hundreds of cores on commodity clusters.

1. Introduction

Over the last decade, DNA sequencing machines have evolved from slow, but accurate machines to massively parallel sequencing platforms capable of producing shorter reads of much greater redundancy. This change in paradigm has prompted computational biologists to develop new algorithms and new software to handle the assembly of these short reads into full genomic data. However, this problem of piecing together sequenced reads into a full genome, known as *de novo* assembly, is much akin to putting together a jigsaw puzzle with millions to billions of very small pieces.

Most software today cannot cope with the vast amounts of data provided by DNA sequencers. The ones that can often require resources at the scale of supercomputers and are often not accessible to the general public. As a result, biologists with access to DNA sequencing machines may not use them often because they are unable to process the sequenced data without the aid of a supercomputer.

However, in recent times, cloud computing platforms have become increasingly more popular and have adequate resources to run complex scientific simulations. Platforms such as Amazon Web Services (AWS)[1] allow any member of the public to run large-scale computation on virtual machines at a reasonable price. As others have argued, the vast resources presented by such platforms present a clear case for cloud computing in genome informatics [17].

In this paper, we present MERmaid, a parallel genome assembler written for cloud computing. While MERmaid follows the general workflow of existing assemblers, especially that of Meraculous [7], it makes several important contributions. Chief among these is accessbility. MERmaid is open-source and was designed to be run on commodity clusters rather than supercomputers. It has been tested and is verified to run on Amazon Elastic Compute Cloud (EC2)[2], which everyone has access to. In the future, an Amazon Machine Image (AMI)[3] containing MERmaid will be released, so that interested parties can simply select our AMI and run our assembler without needing to worry about installing libraries or compiling the source code.

MERmaid was also designed with performance and scalability in mind. In addition to quick, parallel reading of data, MERmaid abides by the philosophy that paging to disk is slow, so it keeps all of its data in memory. MERmaid is able to accomplish this despite the extremely large input data because it takes advantage of the high error rate of the sequencing machines. MERmaid employs low-memory Bloom filters [3] to store data that is likely to be erroneous. Other contributions include the use of locality-sensitive hashing functions to improve the locality of distributed data sets and the exploration various compression schemes for storing data.

The paper is organized in the following manner. Section 2 depicts the current state of assemblers as well as other cutting-edge techniques used in assembly. Section 3 describes MERmaid's architecture and general workflow. Section 4 highlights some of the novel ideas proposed in

[1]`http://aws.amazon.com/`
[2]`http://aws.amazon.com/ec2/`
[3]`https://aws.amazon.com/amis`

MERmaid. In Section 5, we evaluate MERmaid with respect to each design technique. The paper concludes with Section 6, and we describe future work in Section 7.

1.1 Vocabulary

Due to the highly specialized nature of the problem, here we define some commonly-used vocabulary.

Base The most basic component of a DNA sequence. Can take on one of `A`, `C`, `G`, or `T`.

Genome The entirety of an organism's genetic information. Whole-genome assemblers such as MERmaid aim to piece together an individual's genome from sequenced data.

Sequencer A machine that takes an organism's genome and produces short fragments of DNA in the form of reads.

Read A contiguous snippet of the genome that is a sequencer is able to read and produce.

k-mer A substring of a read of size k. We describe the importance of dividing reads into k-mers in Section 2.

Contig A long *contig*uous sequence of bases that results from assembly. Because assembly is not perfect, we often do not obtain a single long sequence of bases representing the whole genome. Instead, we obtain a set of contigs. Longer is better.

2. Related Work

As described in [14], there are there main categories of whole-genome assemblers, of which, the dominant category is the de Bruijn graph assembler. We will focus mainly on de Bruijn graph assemblers, and MERmaid is based on this type of assembler. Assemblers in this category represent the genome as a graph in which the vertices are short reads and the edges connect reads which overlap by $k - 1$ bases. Assembling a genome is equivalent to finding an Eulerian path through the graph. The presence of errors in the sequencing data complicates the process, and all modern de Bruijn graph assemblers use some form of error-correction. Most commonly, assemblers will generate from the input reads shorter sequences of length k, known as k-mers, in a sliding window pattern (Figure 2). This is done to reduce the chance of an error affecting a k-mer and to create a uniform segment length. Velvet [18] is among the earliest of the de Bruijn graph assemblers, and it employs several error-correcting techniques in order to remove spurious edges and to identify "bubbles" in the sequence.

In addition to errors, the massive size of genomes causes performance to be a important issue. ABySS [16] was one of the first attempts at writing a parallelized and distributed whole-genome assembler. ABySS, however, does not attempt to partition k-mers which are likely to be neighbors to the same bin.

More recent assemblers [7, 9] take advantage of the fact that sequencing machines usually report a quality value (QV) for each base, indicating how likely the base contains an error. The use of these richer error models allows assemblers to more quickly and confidently correct errors in the input data.

We base our work on Meraculous [7], a whole-genome assembler which uses quality information associated with the input reads to avoid explicit error-correction steps. One of its strong points is its efficient, low-memory hash table for counting k-mers. The Meraculous codebase is unfortunately somewhat difficult to install and run, as the development team is small and is unable to provide support for others to run it. Although Meraculous is relatively low-memory, it makes excessive use of the filesystem. We made a number of improvements to the Meraculous algorithm in MERmaid to further reduce memory usage and to reduce network I/O traffic, ultimately improving performance and decreasing system requirements.

One very recent assembler, known as SGA [15], takes a different approach to greatly reduce memory usage. Instead of subdividing reads into the smaller k-mers, it uses advanced string edit-distance analysis to detect neighboring reads even with the presence of errors. Its major drawback is that the running time is much greater than any of the de Bruijn graph based algorithms.

We are interested in running whole-genome assemblers on cloud computing platforms, and up until recently there has been no interest in performing genome assembly using cloud computing. Contrail [4] is an assembler which runs on Hadoop. However, it is so recent that there are no publications on Contrail, so it is not possible for us to compare its performance and accuracy.

Melsted et al suggest the use of Bloom filters[3] to efficiently count k-mers [13]. However, in their approach, after they have inserted all the k-mers into the Bloom filter, they read all of the k-mers again in order to accurately count them. We believe that this step is unnecessary as long as one is willing to accept a small rate of false positives. A second drawback with Melsted's implementation is that they must know ahead of time approximately how large the Bloom filter must be. We use a scalable Bloom filter [1] in order to dynamically adjust the capacity.

Minimizing the memory footprint is a popular topic in current research. Conway et al explored the theoretical limits of compression in genome assembly [8] using a succinct

[4]`http://contrail-bio.sf.net/`

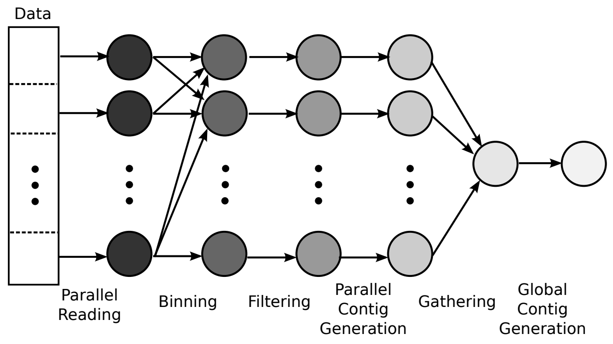

Figure 1. MERmaid's general workflow.

data structure for storing the k-mer count lookup table. We chose to not use their data structure in the interest of time and its effect on performance, but it provides a nice lower bound on how small a k-mer counting structure can be.

3. Architecture

MERmaid's general workflow is described within the following section and by Figure 1. The steps up to the generation of contigs have been implemented. The code is written in C++ and uses the Boost[5] API for OpenMPI[6] to achieve tightly-controlled parallelism. For efficient storage of hash tables, `sparse_hash_map` from the sparsehash[7] library is used. The program is run using only a single command. It takes FASTQ format files as input and writes the final contigs to a FASTA format file.

3.1 Parallel Reading

Since data files from sequencing machines are often extremely large (in the hundreds of GBs for humans), it is imperative to read the input data files in parallel across multiple processes. This first step often proves to be the most time-consuming of the whole process, so in addition to parallelization, we also place the input files on storage with RAID 0 to ensure maximum I/O performance. Anecdotally, we observed that upgrading from a single-disk, simple storage device to RAID 0 device (striped across 4 disks) provided a roughly 2x boost in performance on Amazon EC2. Note that because we use Amazon Elastic Block Store (EBS)[8] volumes, the files are actually being delivered over the network, and the I/O performance also depends on the network bandwidth. For this reason, we focused on using Amazon EC2 High Performance Computing (HPC) Cluster instances[9], which guarantee the greatest amount of network bandwidth (roughly 10Gbit/s).

[5]`http://www.boost.org/`
[6]`http://www.open-mpi.org/`
[7]`https://code.google.com/p/sparsehash/`
[8]`http://aws.amazon.com/ebs/`
[9]`http://aws.amazon.com/hpc-applications/`

```
ACGTACGCGA
ACGTA
 CGTAC
  GTACG
   TACGC
    ACGCG
     CGCGA
```

Figure 2. Generation of 5-mers from a read.

The FASTQ format stores the sequenced reads and the quality scores for these reads in ASCII format. The sequenced reads are represented as sequences of bases (`A`, `C`, `G`, `T`), and there exists a quality score for each base in the read. The quality score is encoded in a single byte and represents the sequencing machine's confidence that the base in the sequence is correct. Note that a base with the value of `N` with quality score 0 may also appear in the case that the sequencing machine is completely unsure; in our current implementation, we ignore any read with `N`s. Typically, the reads generated from a sequencing machine are larger than the k-mers, so a sliding window of size k is used to generate the k-mers, as demonstrated by Figure 2. In addition, for each k-mer, the bases to the left and right of the k-mer (base extensions) and the quality scores of these bases are recorded. For k-mers that are at the edge of a read, an arbitrary base extension with a quality score of 0 is assigned to the side of the k-mer that is at the edge of the read. This information is used later in the construction of the de Bruijn graph.

Once a k-mer has been generated along with all its supplementary information (the base extensions and their quality scores), a hash of the k-mer is used to bin the k-mer. By redistributing the k-mers according to a consistent hashing scheme, we can ensure that every occurrence of each k-mer will belong to a single node, enabling the node to process the k-mer entirely locally. It is worth noting that the generation, sending, and receiving of k-mers are interleaved to maximize throughput by overlapping computation with communication.

Nodes receiving k-mers insert them into a hash table with the k-mer as the key and the supplementary information as the value. However, the number of k-mers that must be stored is often time very large and will not fit in memory if simply stored into a hash table. This is why MERmaid takes advantage of Bloom filters to reduce the storage space for k-mers that are likely to be erroneous. See Section 4.2 for a more detailed explanation of how the Bloom filter is used.

3.2 Parallel Filtering

Like Meraculous, MERmaid avoids an explicit error correction step by filtering the k-mers it collects. To avoid using erroneous k-mers, each node on MERmaid automatically discards any k-mers which do not occur at least d_{min} times (choose d_{min} according to [6]). k-mers that occur only a few times are usually present due to mistakes made by the sequencing machine, so it is acceptable to discard these low-frequency k-mers. k-mers that occur at least d_{min} times are assumed to be "true" k-mers and are saved for the construction the de Bruijn graph.

k-mers are also filtered based on their extensions. For each k-mer, MERmaid tallies up the number of "high quality" extensions on each side. A "high quality" extension refers to a base extension with a quality score of at least Q_{min} (Q_{min} is arbitrarily chosen). If a k-mer has a single "high quality" extension on each side, it is considered valid. All other k-mers are discarded, including k-mers with multiple "high quality" extensions, because they cannot be used during the contig generation step due to ambiguity. After these k-mers are discarded, all the remaining k-mers are guaranteed to have a unique extension in both directions.

3.3 Parallel Contig Generation

Contig Generation After the filtering step, contigs are generated by combining multiple k-mers. For each k-mer, we combine the last $k-1$ bases of the k-mer and its high quality right extension to find next k-mer in the contig. Using this value, we query the hash table for the existence of the next k-mer and also obtain its right extension. Once we have done so, we can repeat the process for the next k-mer, and so on until we no longer find the next k-mer in our hash table of k-mers. Then, we go back to the first k-mer in our contig and repeat the process, this time extending to the left. Once we finish on the lefthand side, we have generated our complete contig, and we move on to an arbitrary k-mer to begin generating the next contig.

In MERmaid, before proceeding with contig generation at the global level (across all k-mers), we first attempt to generate the contigs in parallel at the local level (within each node). Each node iterates over all the k-mers it holds and attempts to generate largest contig it can usingly only the k-mers stored locally. In addition to saving time by generating the contigs in parallel, the consolidation of k-mers into contigs also reduces the amount of network traffic during the global contig generation step, thereby reducing I/O time. Obviously, if the k-mers were binned using a random hashing scheme, the contigs generated would have a very small average length, rendering this step useless. MERmaid's solution is to use a locality-sensitive hash (LSH). The LSH helps to ensure that similar-looking k-mers will be put into the same bin, thereby increasing the average length of the contigs produced during this step. Section 4.3 describes the exact locality-sensitive hashing scheme we used in detail.

3.4 Global Contig Generation

After the parallel contig generation step, the contigs and k-mers of each node are gathered to node 0. Then, contig generation is performed across all gathered contigs and k-mers using the same technique presented in the previous step. The generated contigs are the global contigs. Of these, contigs which do not have a length of at least $2k$ are not deemed useful and are discarded. These resulting contigs are the final contigs of the dataset and are written to the output file using FASTA format.

3.5 Scaffolding

Once the contigs have been generated, some contextual information about the sequencing data can be used to generate scaffolds, which are lists of contigs which are non-overlapping but are a known distance from each other. While MERmaid does have not have this step implemented, it can use the output file from the previous step and the last part of the Meraculous pipeline to generate these scaffolds. The authors of Meraculous claim that the steps up to contig generation take the majority of time in assembly, which is why we concentrated on the steps up to contig generation rather than scaffolding.

4. Design

To keep in line with MERmaid's philosophy of "avoid paging to disk", three key ideas to reduce memory footprint were tested: compression of k-mers, the use of Bloom filters to avoid storing erroenous k-mers, and the use of locality-sensitive hashing for optimal binning.

4.1 Compression of k-mers

Taking inspiration from SGA [15], in which they keep all reads stored in a compressed format using the Burrows-Wheeler transform (BWT) [4], we also attempted to store our k-mers in a compressed state. We experimented with multiple algorithms, including BWT, the move-to-front transform (MTF) [2], run-length encoding, and Huffman coding [10]. As recommend by [12], we attempted to compress the k-mers by first using BWT, then MTF, then RLE, and then Huffman encoding.

We also experimented with the number of bits used to encode a base. For only `A`, `C`, `G`, and `T`, 2 bits are sufficient to encode a base. However, if we wish to encode `N` in the future at least 3 bits are necessary. Therefore, we attempted

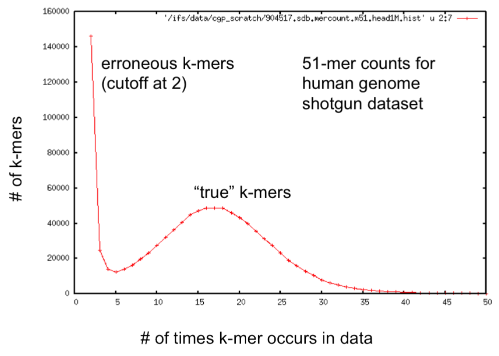

Figure 3. The distribution of k-mers as demonstrated by the frequency of k-mers and the number of unique k-mers which have this frequency. Taken from [7].

our compression techniques on both 2-bit encodings and 3-bit encodings.

The results of attempting to use these compression techniques can be found in Section 5.1.

4.2 Bloom Filter

As mentioned in Section 3.1, simply storing all k-mers is not feasible due to the sheer size of data. As we can see from Figure 3, the majority of k-mers are actually erroneous k-mers which only occur 1 or 2 times. Therefore, what we would like to do is to avoid placing erroneous k-mers into the hash table in the first place.

MERmaid's solution to this is to insert the first occurrence of each k-mer into a Bloom filter. The Bloom filter is an extremely space-efficient data structure, which probabilistically tests for set membership, and is a much better candidate for holding the erroneous k-mers than the hash table. We can check whether an occurrence of a k-mer is the first occurrence simply by testing whether the Bloom filter contains the k-mer. If not, the k-mer is a first occurrence. If so, the k-mer has likely already been inserted into the Bloom filter once and is not the first occurrence. In this case, we should place the k-mer in the hash table. By forcing the first occurrence of each k-mer to be inserted into the Bloom filter rather than the hash table, we ensure that the likely erroneous k-mers, which appear only once, only reside in the Bloom filter, in which space is cheap, thereby reducing the memory footprint of the overall program. Since each process has its own unique set of k-mers, we use a separate Bloom filter for each process and do not need to worry about communicating between different processes when querying the Bloom filters.

One drawback of the Bloom filter is that it is a static data structure. To limit the number of false positives when testing for set membership, the user must first know the number of elements that will be inserted into the Bloom filter. However, because the binning scheme is based on locality-sensitive hashing, it is unclear how many k-mers will be assigned to each node; the binning distribution could be extremely uneven. Therefore, *a priori* knowledge of the number of k-mers that will be inserted into the Bloom filter is not available. To remedy this situation, MERmaid makes use of scalable Bloom filters [1]. The scalable Bloom filter is able to guarantee false positive rates for any number of elements by keeping a list of Bloom filters and adding new Bloom filters to the list if the old ones get too full.

The results from using the Bloom filter in MERmaid can be found in Section 5.2.

4.3 Locality-Sensitive Hashing

The locality-sensitive hashing mechanism is used in MERmaid to maximize the average length of the contigs generated from the parallel contig generation step. To accomplish this, k-mers that extend each other, or similar-looking k-mers, must be placed in the same bin, motivating the use of locality-sensitive hashing.

Shingling The specific hashing technique that we use in MERmaid is called shingling [11]. It works roughly as follows. Given a k-mer, move a sliding window of size j across it. Hash each of the resulting substrings, and take the lowest hash value. If two k-mers are similar, then there should be a large overlap in the sets of substrings generated by the sliding windows. Therefore, for similar k-mers, it is likely that the lowest hash value from the set of substrings will be the same. The user can then simply take the hash value modulo the number of bins to get the correct bin number, and if the similar k-mers have the same lowest hash value, they will have the same bin number.

While shingling is an excellent technique for ensuring that similar k-mers end up in the same bins, it may also be very time-consuming due to the number of hashes it has has to calculate. We explore this tradeoff and the other results of our locality-sensitive hashing in Section 5.3.

5. Evaluation

To evaluate MERmaid, we use the 15.4 megabase genome of the yeast *Pichia stipitis* mentioned in [7]. The input files totaled to 5.6 GB and had 31 million reads. We chose $k = 41$, $d_{min} = 10$, and $Q_{min} = 20$. Each read had length 75, so the these input files generated a total of around 1 billion k-mers. We ran MERmaid on Amazon EC2 Cluster Compute Eight Extra Large instances. These instances have 60.5 GB of memory, 2 eight-core 2.6 GHz CPUs, 10 Gigabit Ethernet, and run 64-bit Ubuntu 11.10. MERmaid

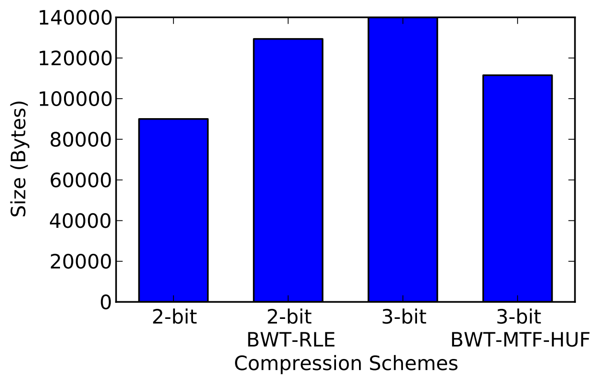

Figure 4. The memory footprint of 10,000 k-mers using different compression schemes. `HUF` **is used for Huffman coding.**

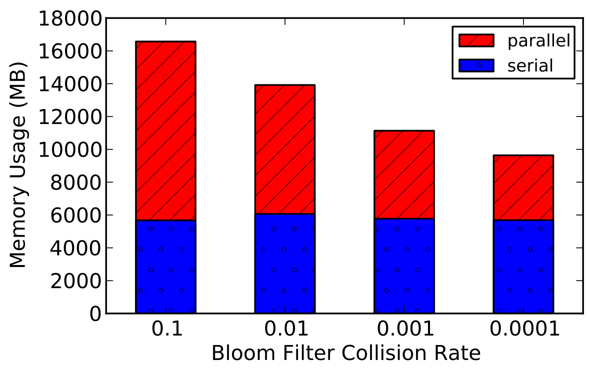

Figure 5. Memory usage of MERmaid across different Bloom filter collision rates. The parallel part denotes steps up to the gathering step. The serial part denotes memory usage during the global contig generation step.

was tested with 8, 16, and 32 cores; in the case of 32 cores, 2 instances were used.

5.1 Compression

Because we were unsure of the advantages of compression, we tested it with random samples of the dataset. Figure 4 shows how much memory it would take to store 10,000 k-mers with or without compression. The compression schemes shown were the compression schemes that proved to be optimal during experimentation. As can be seen, with 2-bit bases, the compression actually increases the memory usage, and with 3-bit bases, there is about a 20% reduction in memory usage with compression. We believe the increase in memory usage for the 2-bit base case is because of the overhead from the variable length encoding used with the compression scheme. Because using only 2 bits per bases is already so efficient, and because there is quite a bit of entropy in genomic data, we believe the optimal encoding scheme for the 2-bit base case is to simply not use compression. In addition, having an even power of 2 is very convenient for multiple reasons. However, if one wishes to take advantage of the extra error information that can be provided with the `N` base in 3-bits, then we can provide a decent amount of savings by using the combination of BWT, MTF, and Huffman encoding.

5.2 Bloom Filter

To test the effectiveness of the Bloom filter, we varied the allowed collision rate for the Bloom filter, and measured the impact on total memory usage, as shown in Figure 5. As expected, decreasing the allowed collision rate increases the amount of storage required by the Bloom filter during the k-mer counting step and reduces the overall memory usage of the program. Furthermore, we can verify that it is indeed the Bloom filter that is lowering the memory usage by the fact that only the parallel portion of memory is shrinking. The serial portion (memory used to gather and generate global contigs) remains the same.

In Figure 6, we see that as we increase the collision rate, the overall time required to generate the global contigs increases. As the collision rate increases, the number of k-mers that get stored in the Bloom filter also increases. To cope with the growth in utilization, the scalable Bloom filter must add new Bloom filters to ensure that the collision rate stays constant. To check for set membership in a scalable Bloom filter, k-mers must check for set membership in each of the internal Bloom filters. As the number of Bloom filters, the number of checks increase. Each of these checks is essentially multiple hash operations, so an increase in running time is expected. Therefore, from Figures 5 and 6, we have shown that we can trade off memory for time and time for memory by varying the Bloom filter collision rate. The user can configure the program based on how large the dataset is.

5.3 Locality-Sensitive Hashing

To measure the effectiveness of the LSH, we decided to look at the distribution of contig lengths after the parallel contig generation at local scale step. We ran MERmaid once without LSH, as shown by Figure 7, and once with LSH, as shown by Figure 8 to see whether there was much of a difference.

As shown by the graphs, there is clearly a difference between turning LSH on and off. Without LSH, the range of contig lengths generated is very narrow, with a maximum of

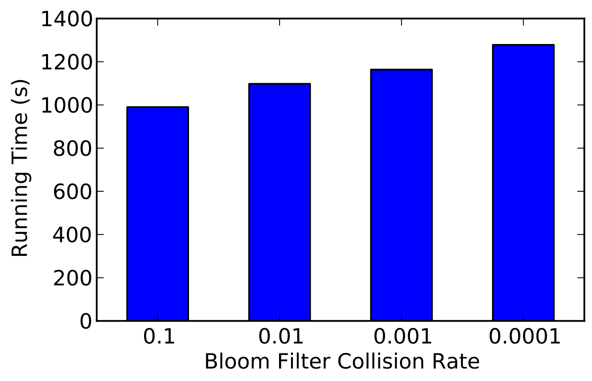

Figure 6. Overall time used to run across different Bloom filter collision rates.

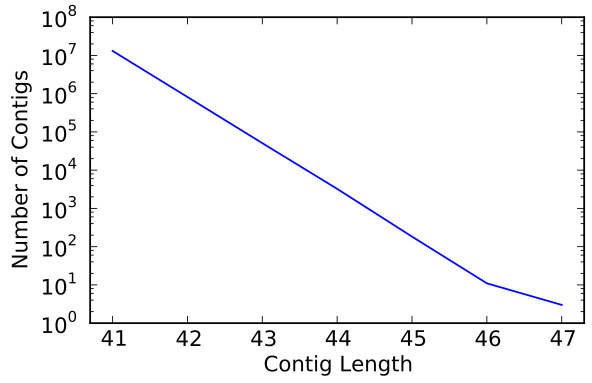

Figure 7. Average contig length after the parallel contig generation at local scale step without LSH. Run on 16 processes.

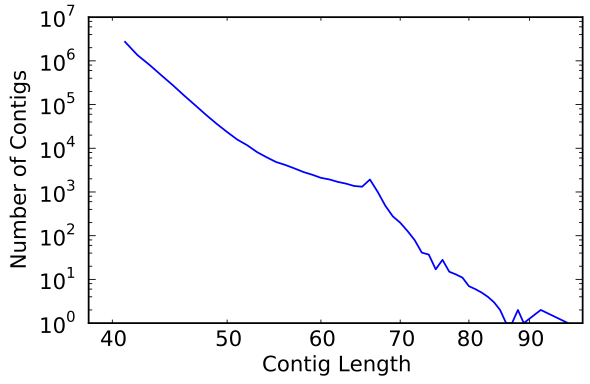

Figure 8. Average contig length after the parallel contig generation at local scale step without LSH. Run on 16 processes.

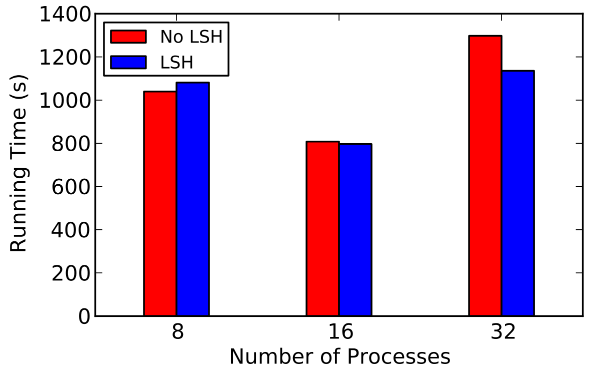

Figure 9. The running time of MERmaid for varying numbers of processes with and without LSH.

only 47. However, with LSH, the range contig lengths increases significantly with a maximum contig length of 97. This clearly shows that there is a clear advantage in using LSH. While it may be the case, that even with LSH, the majority of contigs have lengths in the 40s, there are also quite a few contigs that have longer lengths. This data quite clearly shows that using LSH, should quite significantly reduce the amount of network traffic sent during the gather stage, and should help in reducing the overall running time of the program.

Figure 9 provides further evidence that the combination of LSH and the parallel generation of contigs is a useful step. In most cases, the use of LSH decreases the running time. However, it is interesting to note that the decrease in running time is not extremely noticeable. This is because computing the LSH itself is also a very time consuming operation. Our LSH scheme requires that the hash of several substrings be computed, and this likely increases the running time. From the figures, it seems that the advantages of the LSH outweigh the disadvantages for 16 and 32 processes, while the at 8 processes, the overhead of computing an LSH is not worth it. At lower numbers of processes, the average contig length increases because there are more k-mers per bin, so the LSH may not improve locality enough to justify using.

Another interesting figure to note is Figure 10, which measures the overall memory for varying numbers of processes. Interestingly enough, with LSH enabled, the amount of memory used by MERmaid decreases somewhat significantly. We believe this is because enabling LSH means that contig lengths increase, and so after the gather step, node 0's storage for gathered contigs is a lot smaller than it would be without LSH. This happens because when the different k-

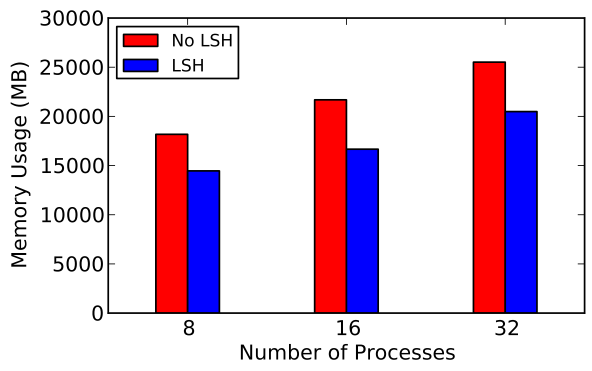

Figure 10. Overall memory usage of MERmaid for varying numbers of processes with and without LSH.

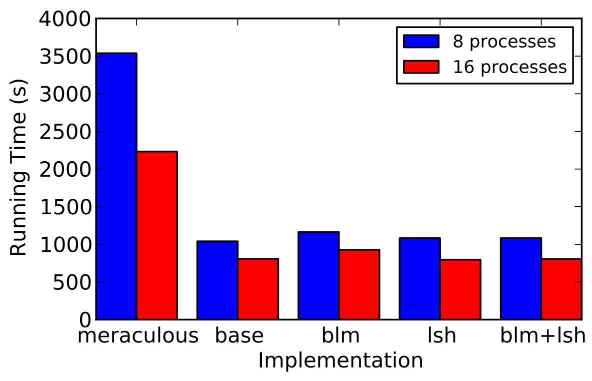

Figure 11. Overall running times for Meraculous and MERmaid under different configurations. `blm` **represents the use of Bloom filters and** `lsh` **represents the use of locality-sensitive hashing.**

mers within a contig are sent separately to node 0, there is redundant space use on all the bases which overlap between them. It is the same phenomenon that causes the initial explosion of data when generating k-mers from input reads. By generating longer contigs before gathering them to node 0, we save a great amount of what would have otherwise been redundant space.

5.4 Overall

In addition to the experiments above, we ran MERmaid in its various configurations to simply look at overall running times and memory usage. We compared these numbers to Meraculous, which was also set up to run on Amazon EC2 using 8 and 16 cores. The results can be seen in Figures 11 and 12.

As can be seen from the figures, MERmaid is clearly an improvement over Meraculous in both running time and memory usage. It seems even the base system of MERmaid is at least twice as faster and twice as memory conservative as Meraculous on both 8 and 16 cores. This shows that MERmaid is an assembler that is able to compete and win against the state-of-the-art assemblers of today.

Across the different configurations of MERmaid, it seems the running time actually stays somewhat consistent. The use of Bloom filters probably does not affect the running time too much, other than hashing overhead, and we've already determined that the hashing overhead of using LSH may be drowning out the savings of the parallel contig generation step. The use of 16 processes over 8 processes clearly show a clear reduction in overall running time as suspected, but is not twice as fast. This is likely due to the global contig generation step which is currently completely serial.

Across the different configurations of MERmaid, it seems the memory usage goes down as expected. The Bloom filter case and the LSH case both show significant reductions in memory, and the combined use of Bloom filter and LSH seem to make MERmaid twice as memory conservative. Increasing the number of processes also seems to increase overall memory usage. We believe this increase in memory usage is due to the overheads associated with creating multiple processes, but this is minor and the average memory usage per process still decreases.

6. Conclusion

As shown by Section 5, MERmaid is a competative assembler with the advantage that it can be run on commodity clusters. We have shown that MERmaid is able to take advantage of the resources on Amazon EC2 to complete the assembly of yeast genome in a reasonable amount of time and cost (only approximately $0.80 when prorated).

We have shown that compression at the base-level is not really worth it due to the overheads and the extreme efficiency of already using only 2 bits per base. We have also shown that the use of a Bloom filter to get rid of erroneous k-mers can significantly decrease the amount of memory required for assemblers. In addition, the use of LSH and a parallel contig generation step may be a useful step to reduce the overall network traffic and memory usage.

Designing an algorithm for use on a cloud computing platform comes with challenges not seen on a supercomputing environment. Memory and network I/O bandwidth is much more precious, so one must always minimize the use of those resources. Some hardware architectural additions that would have made writing distributed algorithms on

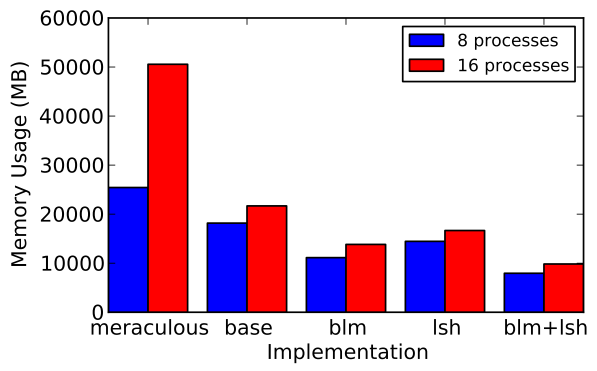

Figure 12. Overall memory usage for Meraculous and MERmaid under different configurations. `blm` **represents the use of Bloom filters and** `lsh` **represents the use of locality-sensitive hashing.**

commodity clusters simpler include finer control over network I/O with respect to topology, cheaper message passing support, and more data-flow-like operations to more dynamically determine execution based on available resources and data.

7. Future Work

There are plenty of opportunities for future work with this project. First and foremost, we would like to move up from yeast genome to human genome. The human genome is about 200 times bigger than the yeast genome, so the just the sheer size difference should bring up some unforeseen challenges. We will at least need to use on the order of dozens of nodes to assemble a human genome, so the internode bandwidth will be more of a restriction than it is currently.

In addition, certain parts of the pipeline are currently locally data-parallel, so we plan to implement GPU versions of those specific kernels. We know from Section 5 that enabling LSH did not significantly reduce overall running time due to the hashing overhead, so we think that a GPU may help to substantially decrease the hashing overhead. Specifically, we are thinking of creating kernels for the k-mer generation step, any hashing of k-mers, and possibly the contig generation step.

One downside of using a GPU is an accelerator is that the Amazon EC2 Cluster Compute instances with GPUs only have two GPUs split between eight general purpose cores, so effectively timesharing the GPUs will likely be a great challenge.

Plans are also underway to develop a preprocessing engine. This engine would sample the input dataset and determine optimal values for various configuration parameters such as d_{min}, k, Q_{min}, and the intial size for the Bloom filter. Ideally, this would be run as a just-in-time (JIT) compilation step runs on a sample of the dataset and could possibly implemented with SEJITS [5] technology.

Other work that we are considering doing include automatic load balancing and fault tolerance. In the case that one bin is nearing its maximum allocation of memory, we would like to split the bin. This is especially important on datasets which we have not run before, and do not know memory requirements for. Assembly of genomic data often takes a long time, and having a job fail can be frustrating. For that purpose, a system for fault tolerance would be very desirable. Currently, if one job fails for us, then we need to rerun the entire step as our checkpoints are only between steps.

References

[1] P. S. Almeida, C. Baquero, N. Preguiça, and D. Hutchison. Scalable bloom filters. *Inf. Process. Lett.*, 101(6):255–261, Mar. 2007.

[2] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei. A locally adaptive data compression scheme. *Commun. ACM*, 29(4):320–330, Apr. 1986.

[3] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[4] M. Burrows and D. J. Wheeler. A Block-sorting Lossless Data Compression Algorithm. 1994.

[5] B. Catanzaro, S. A. Kamil, Y. Lee, K. Asanovi, J. Demmel, K. Keutzer, J. Shalf, K. A. Yelick, and A. Fox. Sejits: Getting productivity and performance with selective embedded jit specialization. (UCB/EECS-2010-23), Mar 2010.

[6] M. Chaisson, P. Pevzner, and H. Tang. Fragment assembly with short reads. *Bioinformatics*, 20(13):2067–2074, 2004.

[7] 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*, 6(8):e23501, 08 2011.

[8] T. C. Conway and A. J. Bromage. Succinct data structures for assembling large genomes. *Bioinformatics*, 27(4):479–486, Feb 2011. [DOI:10.1093/bioinformatics/btq697] [PubMed:21245053].

[9] S. Gnerre, I. Maccallum, D. Przybylski, F. J. Ribeiro, J. N. Burton, B. J. Walker, T. Sharpe, G. Hall, T. P. Shea, S. Sykes, A. M. Berlin, D. Aird, M. Costello, R. Daza, L. Williams, R. Nicol, A. Gnirke, C. Nusbaum, E. S. Lander, and D. B. Jaffe. High-quality draft assemblies of mammalian genomes from massively parallel sequence data. *Proc. Natl. Acad. Sci. U.S.A.*, 108(4):1513–1518, Jan 2011. [PubMed Central:PMC3029755] [DOI:10.1073/pnas.1017351108] [PubMed:21187386].

[10] D. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098 – 1101, sept. 1952.

[11] U. Manber. Finding Similar Files in a Large File System. In *USENIX Technical Conference*, pages 1–10, 1994.

[12] G. Manzini. An analysis of the burrows-wheeler transform. *J. ACM*, 48(3):407–430, May 2001.

[13] P. Melsted and J. K. Pritchard. Efficient counting of k-mers in DNA sequences using a bloom filter. *BMC Bioinformatics*, 12:333, 2011. [PubMed Central:PMC3166945] [DOI:10.1186/1471-2105-12-333] [PubMed:21831268].

[14] J. R. Miller, S. Koren, and G. Sutton. Assembly algorithms for next-generation sequencing data. *Genomics*, 95(6):315–327, Jun 2010. [PubMed Central:PMC2874646] [DOI:10.1016/j.ygeno.2010.03.001] [PubMed:20211242].

[15] J. T. Simpson and R. Durbin. Efficient de novo assembly of large genomes using compressed data structures. *Genome Res.*, 22(3):549–556, Mar 2012. [PubMed Central:PMC3290790] [DOI:10.1101/gr.126953.111] [PubMed:22156294].

[16] J. T. Simpson, K. Wong, S. D. Jackman, J. E. Schein, S. J. Jones, and I. Birol. ABySS: a parallel assembler for short read sequence data. *Genome Res.*, 19(6):1117–1123, Jun 2009. [PubMed Central:PMC2694472] [DOI:10.1101/gr.089532.108] [PubMed:19251739].

[17] L. D. Stein. The case for cloud computing in genome informatics. *Genome Biology*, 11(5):207, 2010.

[18] D. R. Zerbino and E. Birney. Velvet: algorithms for de novo short read assembly using de Bruijn graphs. *Genome Res.*, 18(5):821–829, May 2008. [PubMed Central:PMC2336801] [DOI:10.1101/gr.074492.107] [PubMed:18349386].