Cal-DisKS

Calvin-Berkeley Distributed k-mer Sketcher
Elizabeth Koning
Advised by Dr. Joel Adams
Calvin University 2020

Introduction

Computational biology involves immense volumes of data in sequences of DNA. A common question about the sequences is, “how similar are two or more sequences?” k-mers are a widespread tool to answer the question. k-mers are segments of length k that are in the larger sequence. Comparing which k-mers are in two different sequences can measure the similarity between sequences.

A way to address the issue of the large number of k-mers that are involved in these comparisons is to use MinHash. This approach creates “sketches” made of subsets of the k-mers in a sequence. The sketches save memory use and compute time.

This project builds on existing work on k-mer sketchers to develop a parallelized, distributed implementation of a k-mer sketcher. Using parallel I/O and k-mer selection prediction, the project focuses on creating an efficient and usable system.

As for the project's title, “Cal” is an abbreviation for both the University of California, Berkeley, and Calvin University. The project is a senior project at Calvin in collaboration with a lab at Berkeley. The Berkeley advisors are Kathy Yelick and Aydın Buluç. “DisKS” stands for “distributed k-mer sketcher.”

Background

Computational Biology and k-mers

A major aspect of computational biology is analyzing genetic sequences. A genetic sequence might be a complete genome, but it is often shorter sequences of DNA or RNA characters. Obtaining a complete genome of an organism requires piecing together shorter components that are possible for sequencing technology to read. These shorter segments are called “reads.” Assembling a genome is a computationally intensive process. Multiple reads over the same potion of the genome is necessary because of read errors and the assembly process.

Metagenomics analyzes the genetics of multiple organisms together. Instead of isolating the DNA of a single species and sequencing, metagenomics takes a sample of organic material from the environment, and sequences it together with all of the different species’ DNA. This can offer insights into the community in that environment.

A major issue in computational biology and genetics is the massive amounts of data involved in the problems. The complete human genome has 3 billion base pairs, which requires at least 750

megabytes to store. The data to assemble the genome is larger, and metagenomic samples are larger still. Metagenomics data sets frequently have 3-10 terabytes of data. As the technologies to produce the data improve, the amount of data available increases as well, so the volume is only growing.

One common strategy used throughout computational biology to address the amount of genetic data is to break the sequences into smaller chunks, called k-mers. A k-mer is a sequence of k bases from the longer string of characters. The k-mers can be stored as a set or multi-set. Then, the sequences can be analyzed based on which k-mers they contain, and sequences can be compared based on their k-mer content.

K-mers are often used to answer the question, "how similar are two DNA sequences?" There are two major applications of comparing sequences with k-mers. In the first, the goal is alignment. This is typical for genome assembly and focuses on finding a very accurate identification of if and how the sequences overlap. In this first case, k-mers could be used to identify sequences that are likely to overlap, and then a nucleotide-level comparison can be performed. In the second, the goal is to estimate similarity. In this case, an approximation rather than a precise calculation can be used, which is a much less intensive computation.

In both cases, there is still the problem of having many k-mers. While k-mers are useful for efficient comparison, they do not decrease the amount of memory required. There are very many k-mers involved, as there is a k-mer for every sequence of k bases in the original data set.

The Cal-DisKS project focuses on the second case, where the goal is an estimation. In order to more efficiently estimate the similarity between sequences, it uses the MinHash technique.

MinHash Technique

MinHash is a technique that "sketches" the set of k-mers in a set. This limits the memory needed, and makes set comparison much faster. To do this, it creates a set that is a subset of the k-mers in the sequence, which can be thousands of times smaller than the original representation. Yet the sketch or subset is still a useful approximation. In selecting a size for the sketch, there is a trade-off between maximizing precision and minimizing memory. While the error is based on the side of the sketch rather than the size of the data set, a large sketch has lower error, and a smaller sketch requires less memory and compute time.

Selecting appropriate k-mers for the sketch is a key aspect in achieving an accurate representation of the overall set. Choosing random k-mers would result in a high level of error due to sketches of different sets not necessarily selecting their shared elements. Choosing the lexicographic smallest k-mers would be biased, and likely not offer an accurate representation either. However, choosing the minimum hash values of the k-mers can offer an accurate selection. The hash values can be spread across the entire range of k-mers.

The sketch is created by hashing each of the k-mers and storing a defined number of k-mers with the smallest hash values.

Figure 1 shows this selection process. The large shaded circles are the two sets, from two different files. Then, the small circles represent each of the k-mers in these data sets. The filled circles are those which are kept in the sketch representation. Here, the size of the set is five, so each of the sets contains five elements. This enables computation of the percent identity between two sketches. With each of the hash values of the approximation in sets, the intersection can be found.

In the computation, the process reads each of the k-mers and adds it to the set. When the set capacity is reached, then it discards the maximum values to include the smaller values.

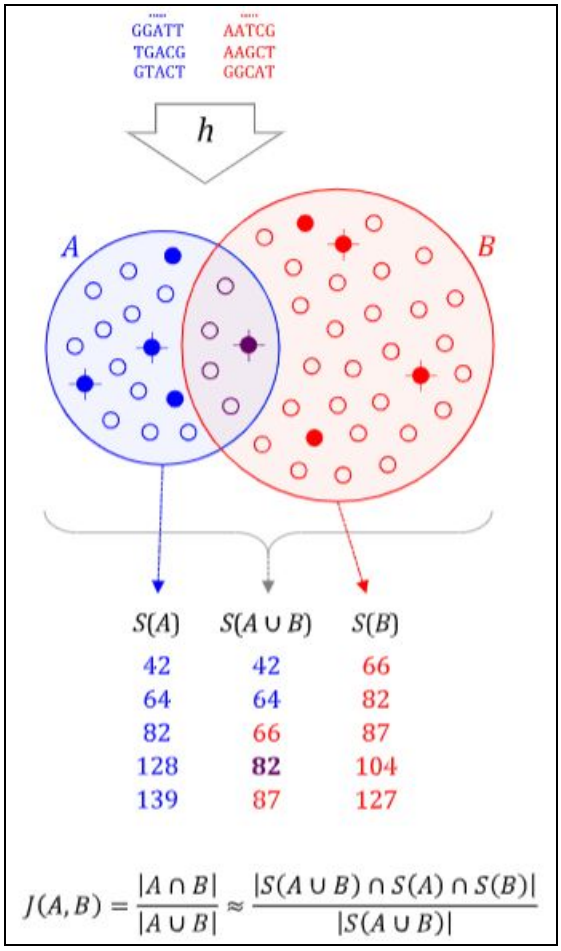

Figure 1: MinHash structures and set comparison.[1]

Distributed Computing

Distributed computing enables using distributed computing systems to coordinate multiple computers throughout a supercomputer or a networked system. Cal-DisKS has been developed in order to work well on a supercomputer with a parallel file system, so that all the processes can access the same file to read simultaneously.

A supercomputer is composed of many computing nodes that have their own local data, but that can send messages between the local memory in order to coordinate their calculations.

MPI, an acronym for Message Passing Interface, is a standard for message passing on distributed systems. MPI allows for various different types of communication. The most simple are sends and receives, where individual processes can send data, whether that be integers, doubles, or characters, to another individual process. The communication used in Cal-DisKS is gather. The gather communication pattern sends arrays from each of the nodes to a single root process. That process then accepts its own

[1] Ondov.

local array and the local arrays of the other processes as one global array, with a length of the number of process times the number of elements per process.

Distributed computing, in general, breaks down problems into smaller sections that can be sectioned into smaller portions for the different computing nodes to handle. Cal-DisKS focuses on data parallelism. This allows different nodes to handle a chunk of the data and them assemble it into the single sketch of the dataset.

Design

The initial goal of the Cal-DisKS project was to modify an implementation of MinHash to work in a distributed system by using BCL (Berkeley Container Library). There are a few different implementations of MinHash that have been published, but none of them have a distributed computing component.

BCL is a library created at Berkeley by Ben Brock.[2] It provides data structures that can be used with a variety of distributed computing environments, including MPI and UPC++. Its structures include many basic data structures, including queues and hash tables. BCL was chosen as the initial approach for the distributed requirement of the project because the writer of the library is one of the advisors, and was available to add to the library for this project.

For the sketching portion of the project, an existing implementation of MinHash needed to be selected from the published versions.

The implementation that was finally selected was sketch.[3] This implementation is a header-only C++ library of sketch data structures. The interface requires individual elements to be added to the sketch object, but the elements can be of any type that can be included in a C++ set. It does not read from a file to select the k-mers. The class that is used for this project is RangeMinHash, which stores the k-mer hash values in a C++ standard library set, and manages the size of the set by removing elements from the end as necessary. Another reason that sketch was selected was because one of the authors recently joined LBNL and was available for questions on this project.

Another implementation that was seriously considered was Mash. This implementation has very thorough I/O with multiple different file types. At one point in development, Mash was selected as the implementation to work with, but the dependencies for the project were not able to be installed on Cori or the Calvin lab computers, so sketch was used instead. A second concern about using Mash was with the I/O. While the interface Mash uses to sketch from a file is very convenient, it also writes the sketch out to a file. Other implementations store the sketch in memory. This means that there is added I/O time for sketching and then comparing sets.

Two other implementations were also considered and not used. An implementation from the Kingsford lab was also considered. However, the project included code in C++, Python, and Perl, and was challenging to run. Because of the parallelization goal, the Perl and Python code meant that this code was not a good fit for the project, as it would not be compatible with C++ MPI. The other was jaccard-ctf and also had compilation problems.

---

[2] Brock.
[3] Baker.

Because of the collaboration with Berkeley, the supercomputer the project was designed to use, adn primarily tested on, Cori, a NERSC supercomputer at Lawrence Berkeley National Lab. However, it should work as well on other supercomputers.

Implementation

Along with including distributed computing in the project, Cal-DisKS assessed the inclusion of a threshold to pre-filter the k-mers as they are added to the set. As k-mers are added to the MinHash set, most of them are discarded. Before adding any of the k-mers to the sets, Cal-DisKS predicts where the cutoff of the hash values will be after all of the k-mers have been added. This avoids storing many values that will never be needed.

The calculation is:

$$threshold = \frac{desired\ \#\ of\ values}{unique\ values} * max\ hash\ value - min\ hash\ value + 1$$

The maximum and minimum hash values are defined by the range of the hash function. The desired number of k-mers is selected by the user of the application. However, the number of unique values is not given. Based on the size of the data file, the total number of k-mers is known, but there are many repeated k-mers in the file.

In order to make use of the threshold, calculating the expected number of unique k-mers is necessary.

The calculation for the expected number of unique k-mers is:[4]

$$\mathrm{E}(N_k) = n\,\frac{n^k - (n-1)^k}{n^k} = \sum_{i=0}^{k-1} (-1)^i \binom{k}{i+1} \frac{1}{n^i} = n - n\frac{(n-1)^k}{n^k}$$

where n is the universe of k-mers (all different k-mers that could be in the data set), and k is the number of k-mers in input the file (from the size of the file).

However, this calculation requires very large numbers. N is the number of k-mers that exist in the universe of k-mers, not just in the file. This means that for very short k-mers of length 7, n is 47 = 16,384. With a typical k-mer of k = 21, n is 421. While C++ offers an unsigned long long for large values, it is only 64 bits, which has a maximum value of 264. The values in the expected value calculation quickly grow beyond this maximum. In order to address this issue, the project used a BigInt class, which accommodates large values by adding digits as necessary. Though this is slower than using the unsigned type, the calculation is only done once.

This calculation is not cheap, so it is not included in the final version. With calculating the fraction to the power of k, it is O(log(k)). The calculation of unique k-mers is not parallelized and is not a quick calculation, due to the large values and the use of BigInt. In practice, this takes significantly longer

[4] Did.

than the time required to read through the file. Because of the structure of the parallel I/O, there was no gain from calculating the threshold.

Parallel I/O

The other contribution of Cal-DisKS is the use of parallel I/O for the sketcher. Though many computational biology applications use parallel I/O, the sketcher implementations referenced are only designed to work with shared memory. While sketch does use SIMD parallelism and is largely thread safe, it does not have a distributed computing component.

In the Cal-DisKS implementation, in order to read the file of genetic sequences, the processes divide the file. If the application is being run on a supercomputer, then each node that is being used reads a chunk of the file as the raw sequences. It then breaks the sequence into k-mers. It uses these k-mers and the threshold calculation to create a local sketch. This local sketch is stored on the individual node and is smaller than the target size of the global sketch. After each local sketch is created, they are sent to process 0, which adds the sketches into a single sketch that becomes the global sketch of the data set.

Once this sketch is created, it can be compared to other sketches to describe the similarity between this set of sequences and another set of sequences.

MPI parallel I/O offers speedup with parallel distributed file systems. Cori uses a Lustre file system, so using parallel I/O is a good fit. Because Calvin's Borg does not have a distributed file system, but instead a single file server node, the project would run on Borg, but would not benefit from the parallelism in the same way. A solution that would be more fitting for Borg would be to have the main process do all of the I/O and send values to the other nodes to process the data. However, this implementation would need to be tested to evaluate whether the additional sends were more or less expensive than the computations that each node could perform.

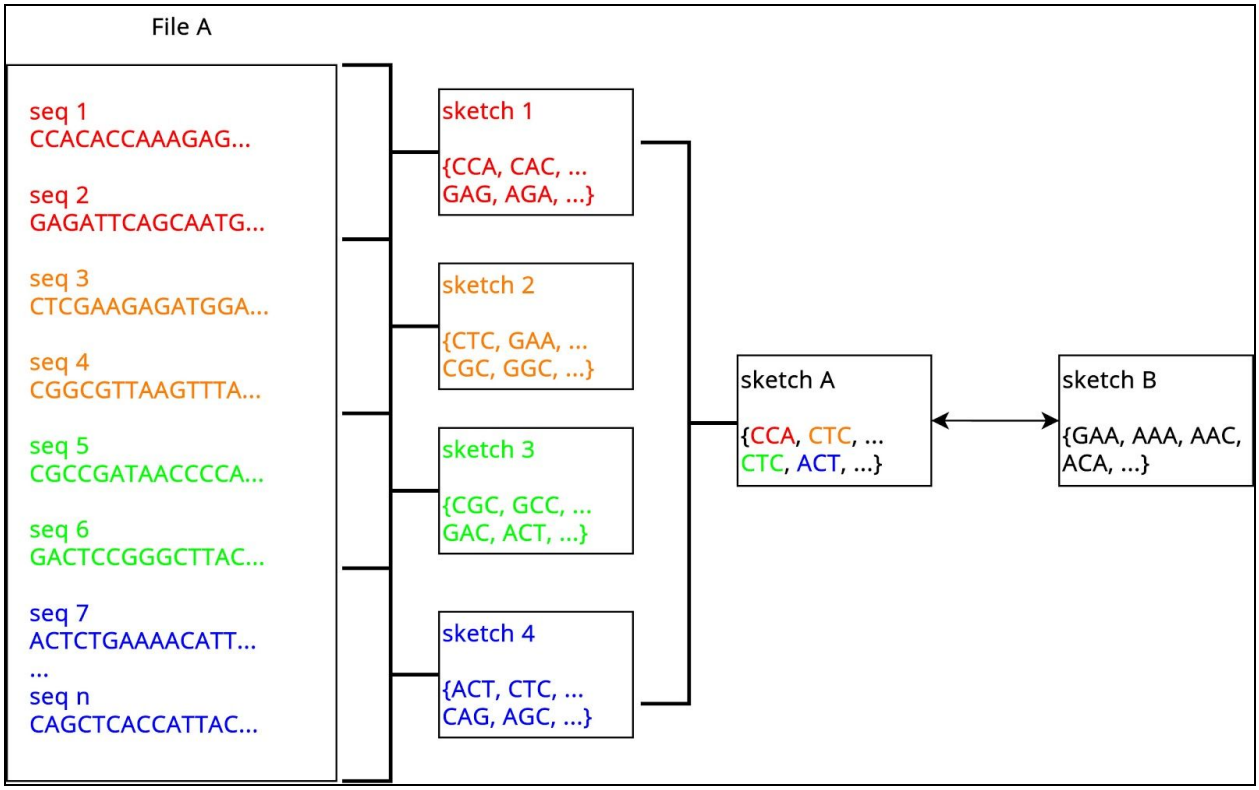

Figure 2: Parallel I/O and sketch comparison.

Results

The final version of Cal-DisKS uses MPI for parallel I/O, and combines the local sketches into a single global sketch. That sketch can then be compared to other sketches created in the same way, using different input files.

The parallel I/O component was more effective than the threshold addition, as each aspect was implemented. The parallel I/O is expected to be highly effective on Cori and other systems that have parallel file systems. It would be less efficient on Calvin's Borg, which manages the files through a single file server node.

There is not currently timing data for the software. To state the efficiency of the project, testing on different sizes of datasets and different numbers of processes would be necessary. The efficiency hinges on the parallel file system and whether the size of the dataset calls for the overhead from MPI.

Acknowledgements

Thank you to all the advisors at Berkeley and Calvin who made this project possible.
Berkeley Advisors:
Giulia Guidi
Ben Brock

Dr. Aydın Buluç
Dr. Kathy Yelick

Calvin Advisor:

Dr. Joel Adams

Reference

Daniel Baker, sketch, GitHub, url: https://github.com/dnbaker/sketch.

Benjamin Brock, Aydın Buluç, and Katherine Yelick. 2019. BCL: A Cross-Platform Distributed Data Structures Library. In Proceedings of the 48th International Conference on Parallel Processing (ICPP 2019). Association for Computing Machinery, New York, NY, USA, Article 102, 1–10. DOI:https://doi.org/10.1145/3337821.3337912

Did (https://math.stackexchange.com/users/6179/did). Finding expected number of distinct values selected from a set of integers. Mathematics Stack Exchange. URL:https://math.stackexchange.com/q/72229 (version: 2017-07-20).

Guillaume Marçais, Dan DeBlasio, Prashant Pandey, Carl Kingsford, Locality-sensitive hashing for the edit distance, Bioinformatics, Volume 35, Issue 14, July 2019, Pages i127–i135, https://doi.org/10.1093/bioinformatics/btz354

Langmead BT and Baker D. Genomic sketching with HyperLogLog [version 1; not peer reviewed]. F1000Research 2019, 8:1866 (slides) (https://doi.org/10.7490/f1000research.1117605.1)

Rowe, W.P.M. When the levee breaks: a practical guide to sketching algorithms for processing the flood of genomic data. Genome Biol 20, 199 (2019) doi:10.1186/s13059-019-1809-x

Ondov, B.D., Treangen, T.J., Melsted, P. et al. Mash: fast genome and metagenome distance estimation using MinHash. Genome Biol 17, 132 (2016) doi:10.1186/s13059-016-0997-x

Appendix: Code Implementation

This appendix supplies the key files that were added to the original project. This does not include edits to existing files, such as mh.h or supplementary files where the majority of the code is from other sources. The complete project code can be found at: github.com/kodingkoning/sketch/

```
/* caldiskstest.cpp sketches two files using sketch and MPI
* Elizabeth Koning, Spring 2020
* for Senior Project at Calvin University.
*/
#include "mh.h"
#include <random>
#include <mpi.h>
#include "bcl/bcl.hpp"
#include "mpiParallelIO.cpp"

using namespace sketch;
using namespace common;
using namespace mh;

int main(int argc, char *argv[]) {
    int id;
    if (argc < 3) {
        fprintf(stderr, "\n*** Usage: caldiskstest <inputFile1>
<inputFile2>\n\n");
        exit(1);
    }
    std::string fastq_file1 = argv[1];
    std::string fastq_file2 = argv[2];
    std::string dir = get_current_dir_name();
    std::string filename1 = dir+"/" + fastq_file1;
    std::string filename2 = dir + "/" + fastq_file2;

    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    RangeMinHash<uint64_t> sketch(LOCAL_SKETCH_SIZE);
    sketchFromFile(filename1, sketch);

    RangeMinHash<uint64_t> sketch2(LOCAL_SKETCH_SIZE);
    sketchFromFile(filename2, sketch2);
```

```
    vector<uint64_t> a = sketch2.mh2vec();
    vector<uint64_t> b = sketch.mh2vec();

    if (id == 0) {
      auto s1 = sketch.cfinalize();
      auto s2 = sketch2.cfinalize();
      double similarity = s1.jaccard_index(s2);
      std::cout << "* similarity between sketches = " << similarity <<
std::endl;
    }

    MPI_Finalize();
    return 0;
}

/* mpiParallelIO.cpp handles the parallel I/O for Cal-DisKS
* Elizabeth Koning, Spring 2020
* for Senior Project at Calvin University.
*/

#include <stdio.h>       /* I/O stuff */
#include <stdlib.h>      /* calloc, etc. */
#include <mpi.h>   /* MPI calls */
#include <string.h>      /* strlen() */
#include <stdbool.h> /* bool */
#include <sys/stat.h>
#include <iostream>
#include "mh.h"
#include "calcThreshold.cpp"

using namespace sketch;

int readFile(const char *fileName, int k, RangeMinHash<uint64_t>&
localSketch, int nProcs, int id);
void readArray(const char * fileName, char ** a, int * n);
void parallelReadArray(const char * fileName, char ** a, int * n, int id, int
nProcs);
void scatterArray(char ** a, char ** allA, int * total, int * n, int id, int
nProcs);
void sketchKmers(char* a, int numValues, int k, RangeMinHash<uint64_t> &
kmerSketch);
```

```
void combineSketches(RangeMinHash<uint64_t> & localSketch,
RangeMinHash<uint64_t> & globalSketch, int nProcs, int id);

void sketchFromFile(std::string filename, RangeMinHash<uint64_t>&
globalSketch) {
    int k = 21; // k = 21 is the default for Mash. It should not go above 32
because it must be represented by an 64 bit unsigned int.
      int nProcs, id;
    double startTime, totalTime, threshTime, sketchTime, gatherTime;

      MPI_Comm_rank(MPI_COMM_WORLD, &id);
      MPI_Comm_size(MPI_COMM_WORLD, &nProcs);

    startTime = MPI_Wtime();

      // TODO : would be better to only calculate this on one process
    // BigInt threshold = find_threshold(filename, k, SKETCH_SIZE);
    threshTime = MPI_Wtime();

    RangeMinHash<uint64_t> localSketch(LOCAL_SKETCH_SIZE);

      //std::cout << "Created sketch objects..." << std::endl;

    readFile(filename.c_str(), k, localSketch, nProcs, id);
      //std::cout << "Read FASTQ file and made local sketches..." <<
std::endl;

    sketchTime = MPI_Wtime();

    combineSketches(localSketch, globalSketch, nProcs, id);
      //std::cout << "Combined the sketches..." << std::endl;

    gatherTime = MPI_Wtime();

    totalTime = MPI_Wtime() - startTime;

    if (id == 0) {
            std::cout << "For file " << filename << ": " << std::endl;
      std::cout << " * Threshold calculation time = " << (threshTime -
startTime) << std::endl;
      std::cout << " * Local sketching time = \t" << (sketchTime -
threshTime) << std::endl;
```

```
      std::cout << " * Sketch combine time = \t" << (gatherTime - sketchTime)
<< std::endl;
      std::cout << " * Total Cal_DisKS time = \t" << (totalTime) <<
std::endl;
    }
}

int readFile(const char *fileName, int k, RangeMinHash<uint64_t>&
localSketch, int nProcs, int id)
{
      int allCount, localCount;
      char *a;

      parallelReadArray(fileName, &a, &localCount, id, nProcs);

      sketchKmers(a, localCount, k, localSketch);

      free(a);
      return 0;
}

/* parallelReadArray fills an array with values from a file.
 * Receive: fileName, a char*
 *          a, the address of a pointer to an array,
 *          n, the address of an int,
 *          id, an int id of the current process,
 *          nProcs, an int number of MPI processes.
 * PRE: fileName contains k-mers, and may contain other characters.
 * POST: a points to a dynamically allocated array
 *    containing file size / nProcs values from fileName.
 */
void parallelReadArray(const char *fileName, char **a, int *n, int id, int
nProcs)
{
      int count, howMany, offset, chunkSize, remainder, headerLen, numLen,
chunkChars;
      const int DEFAULT_BUF_LEN = 10;
      int error;
      MPI_File file;
      MPI_Status status;
      MPI_Offset fileSize;
      char *buffer;
```

```
	// open MPI file for parallel I/O
	error = MPI_File_open(MPI_COMM_WORLD, fileName, MPI_MODE_RDONLY,
MPI_INFO_NULL, &file);
	if (error != MPI_SUCCESS)
	{
		fprintf(stderr, "\n*** Unable to open input file '%s'\n\n",
fileName);
	}

	// get the size of the file
	error = MPI_File_get_size(file, &fileSize);
	if (error != MPI_SUCCESS)
	{
		fprintf(stderr, "\n*** Unable to get size of file '%s'\n\n",
fileName);
	}

	// find size of each process's chunk
	chunkSize = fileSize / nProcs;
	offset = id * chunkSize; // TODO: add room on either end to handle not
losing k-mers between reads
	remainder = chunkSize % nProcs;
	if (remainder && id == nProcs - 1)
	{
		chunkSize += remainder;
	}

	buffer = (char *)calloc(chunkSize + 1, sizeof(char));
	if (buffer == NULL)
	{
		fprintf(stderr, "\n** Unable to allocate %d-length array",
chunkSize);
	}
	MPI_File_read_at(file, offset, buffer, chunkSize, MPI_CHAR, &status);
	// buffer will contain all of the chars
	// TODO: add buffer room at the ends so that we don't lose some of the
k-mers

	MPI_File_close(&file);

	*n = chunkSize;
	*a = buffer;
}
```

```
/* complemenntbase() and reversecomplement() come from BELLA code
 */
char
complementbase(char n) {
	switch(n)
	{
	case 'A':
		return 'T';
	case 'T':
		return 'A';
	case 'G':
		return 'C';
	case 'C':
		return 'G';
	}
	assert(false);
	return ' ';
}

std::string
reversecomplement(const std::string& seq) {

	std::string cpyseq = seq;
	std::reverse(cpyseq.begin(), cpyseq.end());

	std::transform(
		std::begin(cpyseq),
		std::end  (cpyseq),
		std::begin(cpyseq),
	complementbase);

	return cpyseq;
}

// modified from 32 bit version at from
https://github.com/Ensembl/treebest/blob/master/common/hash_com.h
inline uint64_t kmer_int(const char *s) {
	uint64_t h = 0;
	for ( ; *s; s++)
		h = (h << 5) - h + *s;
	return h;
}
```

```
/* sketchKmers adds the kmers in the data read to a Minhash sketch
 * Receive: a, a pointer to the head of an array;
 *                numValues, the number of chars in the array;
 *                k, the number of bases in a k-mer;
 *                kmerSketch, the empty sketch to fill with k-mers;
 * Postcondition: kmerSketch is filled with k-mers from a.
 */
void sketchKmers(char* a, int numValues, int k, RangeMinHash<uint64_t> &
kmerSketch) {
      std::string kmer = "";
      for(int i = 0; i < numValues; i++) {
            if(a[i] == 'A' || a[i] == 'T' || a[i] == 'C' || a[i]== 'G') {
                  if(kmer.length() < k) {
                        kmer.push_back(a[i]);
                  } else {
                        // TODO: check against threshold for the hash values
(will need to send the hash value to the sketch for confirmation)
                        std::string twin = reversecomplement(kmer);
                        if (twin < kmer) {
                              kmerSketch.add(kmer_int(twin.c_str()));
                        } else {
                              kmerSketch.add(kmer_int(kmer.c_str()));
                        }
                        kmer = kmer.substr(1, k-1) + a[i];
                  }
            } else {
                  kmer = "";
            }
      }
}

/* combineSketches adds the kmers in the data read to a Minhash sketch
 * Receive: localSketch, the local sketch for easy MPI process;
 *                globalSketch, the global sketch for process 0 to gather the
hash values;
 *                nProcs, the number of MPI processes;
 *                id, the id of the current MPI process;
 * Postcondition: globalSketch for process 0 has the minimum values from the
local sketches.
 */
```

```
void combineSketches(RangeMinHash<uint64_t> & localSketch,
RangeMinHash<uint64_t> & globalSketch, int nProcs, int id) {
	unsigned num_vals = localSketch.size();
	uint64_t * local_data = localSketch.mh2vec().data();
	uint64_t * global_data = NULL;
	if(id == 0) {
		global_data = new uint64_t[num_vals*nProcs];
	}

	MPI_Gather(local_data, num_vals,  MPI_UNSIGNED_LONG_LONG, global_data,
num_vals, MPI_UNSIGNED_LONG_LONG, 0, MPI_COMM_WORLD);

	if(id == 0) {
		for (unsigned i = 0; i < num_vals*nProcs; i++) {
			globalSketch.addh(global_data[i]);
		}
	}

	delete [] global_data;
}


/* calcThreshold.cpp calculates the MinHash predicted threshold for Cal-DisKS
* Elizabeth Koning, Spring 2020
* for Senior Project at Calvin University.
*/
#include <sys/stat.h>
#include <math.h>
#include "mh.h"
#include "include/sketch/BigInt/BigInt.h"
#include <iostream>

const int BASES = 4;
const int SKETCH_SIZE = 150;
const int LOCAL_SKETCH_SIZE = 150;

double expected_unique_dbl(double k, double n) {
    // kmers is the number of picks, which is the number of kmers in the
files
    // n is the number of unique kmers that could be chosen

    // return n - pow(n-1, k)*pow(n, 1-k);
    return n - n*pow(n-1, k)/pow(n, k);
```

```
    // return n * (1 - pow((n-1)/n, kmers));

    double result = 1;
    for(double i = 0; i < k-1; i += 1) {
        result = 1 + (1 - 1/n)*result;
    }
    return result;
}

// based on answer here:
https://math.stackexchange.com/questions/72223/finding-expected-number-of-dis
tinct-values-selected-from-a-set-of-integers
BigInt expected_unique(const BigInt& kmers, const BigInt& n) {
    // kmers is the number of picks, which is the number of kmers in the
files
    // n is the number of unique kmers that could be chosen
    return n - n*power(n-1, kmers)/power(n, kmers); // tested
}

BigInt find_threshold(std::string file, int k, int sketch_size) {
    // size of file
    struct stat sb;
    if(stat(file.c_str(), &sb) == -1) {
        perror("stat");
        exit(EXIT_FAILURE);
    }
    long long size = sb.st_size; // size is in bytes, which is equal to chars
    long long bases = size / 2;
    std::fprintf(stderr, "Size of %s is %lld bytes, or about %lld bases\n",
file.c_str(), size, bases);

    // number of k-mers in the file. this is assuming that the characters
used
    //      for the names of the reads cancels out for the k-mers lost at the
    //      ends of the reads. Future work could improve this calculation.
    long long kmers = bases - k + 1;

    // n is the number of possible k-mers with k bases
    BigInt n = BigInt(pow(BASES, k));

    // number of unique k-mers expected in the file
    BigInt unique_kmers = expected_unique(kmers, n);
```

```
    // TODO: make expected_unique more efficent for practical reasons

    // NOTE: this assumes that max = 2^64 and min = 0,
    //      which is the case for WangHash used in sketch.
    if (sketch_size > unique_kmers) {
        std::cout << "NOTICE: the size of the sketch is greater than the
number of expected unique kmers." << std::endl;
    }
    BigInt threshold = sketch_size / unique_kmers * pow(2, 64) + 1 ;//
(std::numeric_limits<uint64_t>::max) - (std::numeric_limits<uint64_t>::min) +
1;

    return threshold;
}
```