K-mulus: A Database-Clustering Approach to Protein BLAST in the Cloud

Carl H. Albach† Sebastian G. Angel† Christopher M. Hill† Mihai Pop

Department of Computer Science
University of Maryland, College Park
College Park, MD 20741
{calbach, sga001, cmhill, mpop}@umiacs.umd.edu

ABSTRACT

With the increased availability of next-generation sequencing technologies, researchers are gathering more data than they are able to process and analyze. One of the most widely performed analyses is identifying regions of similarity between DNA or protein sequences using the Basic Local Alignment Search Tool, or BLAST. Due to the large scale of sequencing data produced, parallel implementations of BLAST are needed to process the data in a timely manner. In this paper we present K-mulus, an application that performs distributed BLAST queries via Hadoop and MapReduce and aims to generate speedups by clustering the database. Clustering the sequence database reduces the search space for a given query, and allows K-mulus to easily parallelize the remaining work. Our results show that while K-mulus offers a significant theoretical speedup, in practice the distribution of protein sequences prevents this potential from being realized. We show K-mulus's potential with a comparison against standard BLAST using a simulated dataset that is clusterable into disjoint clusters. Furthermore, analysis of K-mulus's poor performance using NCBI's Non-Redundant (nr) database provides insight into the limitations of clustering and indexing a protein database.

General Terms

Bioinformatics, High-Performance Computing

Keywords

BLAST, Cloud Computing, MapReduce, Hadoop, Bioinformatics

1. BACKGROUND

Identifying regions of similarity between DNA or protein sequences is one of the most widely studied problems in bioinformatics. These similarities can be the result of functional, structural, or evolutionary relationships between the sequences. As a result, many tools have been developed with the intention of efficiently searching for these similarities. The most widely used application is the Basic Local Alignment Search Tool, or BLAST[3].

With the increased availability of next-generation sequencing technologies, researchers are gathering more data than ever before. This large influx of data has become a major issue as researchers have a difficult time processing and analyzing it. For this reason, optimizing the performance of BLAST and developing new alignment tools has been a well researched topic over the past few years. Take the example of environmental sequencing projects, in which the bio-diversity of various environments, including the human microbiome, is analyzed and characterized to generate on the order of several terabytes of data[2]. One common way in which biologists use these massive quantities of data is by running protein BLAST on large sets of unprocessed, repetitive reads to identify putative genes[9, 14, 16]. Unfortunately, performing this task in a timely manner while dealing with terabytes of data far exceeds the capabilities of most existing BLAST implementations.

As a result of this trend, large sequencing projects require the utilization of high-performance and distributed systems. However, most researchers do not have access to these computer clusters, due to their high cost and maintenance requirements. Fortunately, cloud computing offers a solution to this problem, allowing researchers to run their jobs on demand without the need of owning or managing any large infrastructure. The speedup offered by running large jobs in parallel, as well as the opportunities for novel reduction of the BLAST protein search space [13, 19], are the motivating factors that led us to devote this paper to effectively parallelizing protein BLAST (blastx and blastp).

One of the original algorithms that BLAST uses is 'seed and extend' alignment. This approach requires that there be at least one k-mer (sequence sub-string of length k) match between query and database sequence before running the expensive alignment algorithm between them [3]. Using this rule, BLAST can bypass any database sequence which does not share any common k-mers with the query. Using this heuristic, we can design a distributed version of BLAST using the MapReduce model. One aspect of BLAST which we hoped to take advantage of was database indexing of k-mers. While some versions of BLAST have adopted database k-mer indexing for DNA databases, it seems that this approach has not been feasibly scaled to protein databases[13]. For

†Authors contributed equally.

this reason, BLAST iterates through nearly every database sequence to find k-mer hits. K-mulus attempts to optimize this process by using lightweight database indexing to allow query iteration to bypass certain partitions of the database.

In this paper we present K-mulus, an application which distributes queries over many database clusters and produces output which is equivalent to that of BLAST. K-mulus aims to achieve speedups by generating database clusters which enable the parallelization of BLAST and reduction of the search space. After the database is partitioned by cluster, a lightweight k-mer index is generated for each partition. During a query, the k-mers of the query sequences can be quickly compared against these indices to determine if the cluster contains potential matches. In this way, a query sequence need only be compared against a subset of the original database, thereby reducing the search space. Note that the speed of K-mulus is completely dependent on cluster variance; if a query sequence matches every partition index, there is no reduction in search space.

mpiBLAST [5] is a popular distributed version of BLAST, which can yield super-linear speed-up over running BLAST on a single node. *mpiBLAST* works by segmenting the database into equal-sized chunks and distributing these chunks among the available nodes. All nodes then proceed to search the entirety of the query set against all chunks of the database. The results from individual nodes are later aggregated. Although K-mulus and *mpiBLAST*'s both divide the database into smaller chunks, *mpiBLAST* chunks the data arbitrarily, while K-mulus does so according to similarity-based clustering.

Another parallel implementation of BLAST is *CloudBLAST*[11]. *CloudBLAST* uses the MapReduce paradigm with Hadoop to parallelize BLAST. Whereas *mpiBLAST* segments the database, CloudBLAST's parallelization approach involves segmenting the queries.

K-mulus differs from the previous parallel implementations of BLAST by using similarity-based clustering of the database along with the creation of a database index for each cluster. There is a large potential advantage to our approach. K-mulus identifies each cluster by a center which allows us to determine whether or not a query will find a match in the corresponding segment of the database. This optimization has the potential to dramatically reduce the search space for each individual query, thus eliminating unnecessary searches.

There are a few advantages of using the MapReduce [6] framework over other existing parallel processing frameworks. The entirety of the framework rests in two simple methods, a mapper and a reducer. During the mapper, input is distributed among all nodes that were assigned to the task of mapping. Each node processes the input in a particular way according to the developer's specifications, and outputs a key-value pair. Once all nodes have finished outputting all their key-value pairs, all the values for a given key are aggregated into a list, and are sent to the reducer. During the reduce phase, the (key, list of values) pairs are received. This list of values is used to compute the final result according to the application's needs. For instance, if a developer wanted to count the number of occurrences of each of the words in a body of text, they could design a MapReduce application that would do this in parallel with little to no effort. Initially, they would feed the body of text as the input. Individual words from the file would be distributed evenly among all available nodes. The mappers would output key-value pairs of the form (word, 1). Finally, when all mappers have finished, the reducers will each get a key, and a list of values. In this case, the list of values is exclusively composed of 1s. The reducer would then aggregate all the entries in the list, and output the final key-value pair, which corresponds to (word, count).

Another advantage of the MapReduce framework is that it abstracts the communication between nodes, thus allowing software developers to run their jobs in parallel over potentially thousands of processors. Although this makes it simple to program, without direct control of the communication, it may be inefficient.

Lastly, MapReduce has become a de-facto standard for distributed processing, thus the code written will be very portable.

The MapReduce framework is tightly integrated with a distributed file system allowing it to coordinate computation with the location of the data. Google Distributed File System (GFS) [7] is one such implementation. Files in GFS are partitioned into smaller chunks and stored distributively among a cluster of computers. A master node is responsible for storing the metadata, such as chunk locations per file and their replicas, and granting access to said chunks. MapReduce attempts to schedule computation at the nodes storing the input chunks. However, given that GFS is proprietary, Hadoop[4], an open-source distributed file system has it's own spin on MapReduce. Applications built using the Hadoop framework only need to specify the map and reduce functions. Hadoop has become the de-facto standard for cloud computing, a model of parallel computing where infrastructure is treated as a service. The end users, or application developers, need not worry about the maintenance or cluster configuration. An example of a popular cloud computing provider is Amazon's Elastic Compute Cloud (EC2)[1]. K-mulus is implemented using the MapReduce framework so it can be easily integrated with both public and private clouds.

2. DESIGN

Our model is divided into two sections. First, we present a methodology for dividing the existing database into smaller clusters and generating indices for each cluster. This is the preprocessing step for K-mulus. Second, we consider the process by which a query is executed in K-mulus.

In order to cluster the database, we used a collection of clustering algorithms: k-means[8], k-medoid[18], and our own MapReduce efficient hierarchical clustering. We also keep track of the centers for each cluster as they play the crucial role of identifying membership to a cluster. Fig 1. shows a high level view of our pre-processing model. We further explain the details involved in clustering the existing database in Section 3, as well as our preliminary results in Section 4.

After the database has been clustered, we compare the input

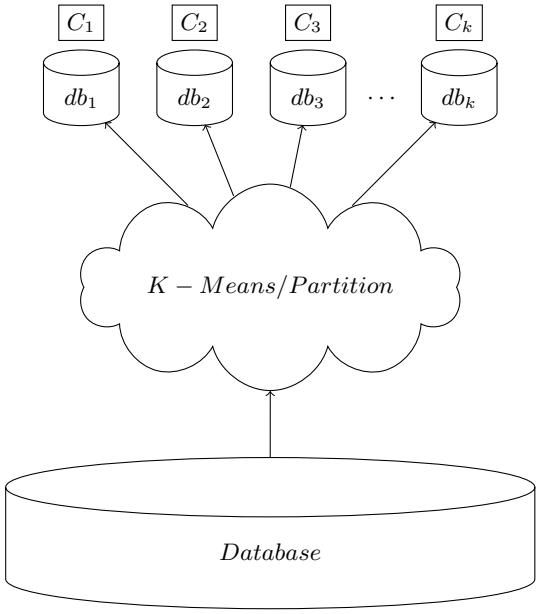

Figure 1: The database is partitioned into k smaller disjoint clusters using the k-means clustering algorithm. The center of each cluster is tracked as they are fundamental for subsequent parts of K-Mulus.

query sequences to all centers. This comparison is explained in detail in Section 3. The key idea is that by comparing the input query sequence to the cluster centers, we can determine whether a potential match is present in a given cluster. If this is the case, we run the NCBI BLAST algorithm on the query sequence and the database clusters that we determined as relevant for the query. Fig 2. shows a diagram of the post-processing step carried out by K-mulus.

2.1 Pre-processing

Since text input by default is split on newlines in MapReduce, the sequences are transformed from *FASTA* format to *simpleFasta* format, where each sequence shares the same line as its header, separated by a single white space. In the future, we plan on implementing a custom Hadoop InputFormat that can read native *FASTA* formatted files.

2.2 Database-specific

First the database sequences are transformed into *simpleFasta* format and uploaded to HDFS. A k-mer presence vector is created for each sequence. The details of the k-mer presence vector are described in Section 3. During the map phase, each sequence is emitted as a k-mer presence vector. This vector has 20^k entries, one for each possible k-mer. These vectors are then clustered using one of the clustering algorithms detailed below. For each cluster, a center vector is produced that is the union of all k-mer presence vectors of the cluster. In other words, the center vector represents all k-mers present in the cluster.

In the final stage of database preparation, the formatdb program is used to build BLAST databases for each cluster. The resulting collection of BLAST databases are zipped and uploaded back to HDFS. The databases have to be zipped because the HDFS namenode stores each file, regardless of size, as a series of 128MB chunks. Compressing the databases into a single archive not only dramatically saves HDFS space, but makes it simpler to load all databases to each machine during the BLAST MapReduce. In the future, the user will be able to download the clusters of the NCBI Non-Redundant database from our website.

2.3 Clustering

K-mulus attempts to minimize the amount of k-mer overlap between different clusters of sequences. If two clusters share a high percentage of k-mers in common, then a query that shares a k-mer with one cluster has a higher chance of overlapping with the other. The query will then be sent to both clusters during the reduce phase of K-mulus described below. Determining the correct number of clusters is very important. If the number of clusters is decreased, the k-mer overlap between clusters increases. The percentage of k-mer overlap of different numbers of clusters is discussed in the results.

K-mulus only clusters the database sequences. We considered clustering the query sequences as well, but decided not to for several reasons. The advantages of clustering the query sequences are that it requires slightly less network overhead in Hadoop to move a cluster of queries instead of single queries, and that it requires less computation to compare the k-mers of cluster centers against database centers, than to compare all query k-mers against database centers. However, the k-mer look up is quite fast already, as it can be done on the order of the length of the query. The trade off for this approach is that overlapping the k-mers of two clusters is inherently noisier than overlapping one query to a cluster. This means that each sequence will be mapped to more partitions of the database which results in unnecessary work by BLAST. Furthermore, this approach incurs the cost of performing the query clustering.

By default BLAST ignores regions of low complexity in the input by repeat masking. For proteins, BLAST uses the SEG program to mask both the query and database sequences [20]. K-mulus similarly masks low complexity sequences within the query and database to spurious matches cause by such regions. Since K-mulus uses an approach identical to that of BLAST in this regard, this optimization of K-mulus causes no loss of sensitivity when compared to a normal BLAST search.

2.4 BLAST

The actual BLAST step of K-mulus is a single MapReduce job. The query protein sequences are converted into *simpleFasta* format, and uploaded to HDFS. At the start of the MapReduce job, the compressed archive of all previously compiled BLAST databases are added to Hadoop's DistributedCache. Archived files added to the DistributedCache are extracted to the local working directory of each compute node that executes a map or reduce function. This way we can effectively transfer the databases and BLAST binaries provided by the user to use during the reduce step.

Prior to the map phase, each computing node loads the k-mer presence vectors of the cluster centers. The centers are only loaded once per node, regardless of the number

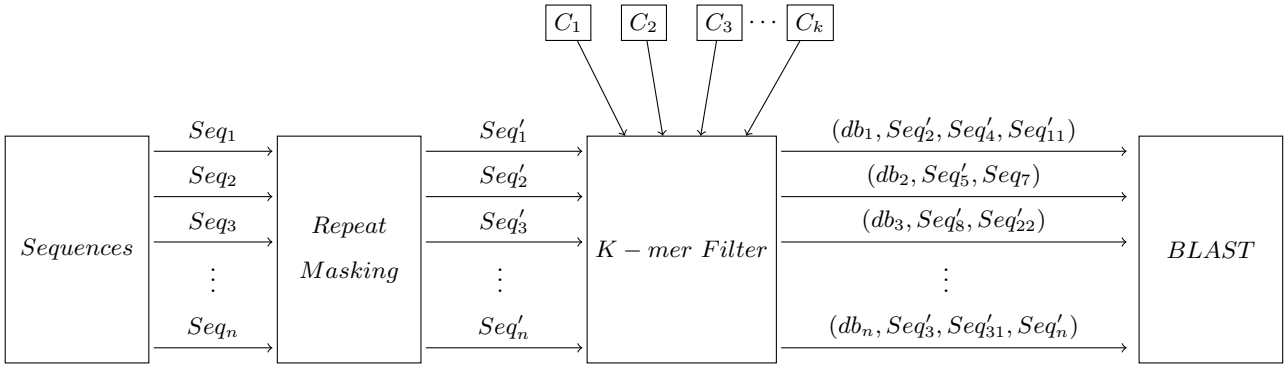

Figure 2: K-mulus takes a collection of protein sequences and BLASTs them against a collection of databases. First, the query's low complexity regions are masked, then each query is compared to the cluster center k-mer vectors. If the query shares a k-mer with a cluster center, the query is sent to the core responsible for running BLAST with that cluster's database.

of map functions the node is responsible for. During the map phase, each k-mer is checked against all cluster centers' presence vectors. If a cluster contains a given k-mer, then that sequence must be aligned to some sequence within that cluster. Therefore, if there exists a shared k-mer between the query sequence q and a cluster center c, then the mapper $emits(c, q)$.

The reducer aggregates all query sequences that share a k-mer with a given cluster. These query sequences are written to a temporary file on the local disk of the reduce node. The reduce node then forks a child process that executes the BLAST binary along with any arguments passed by the user. The BLAST binary determines which database to load based on the *key* received. The output of the forked process is captured and redirected to the reducer's output.

2.5 Post-processing

Since BLAST takes into account the size of the database when computing alignment statistics, the individual BLAST results must have their scores adjusted for the database segmentation. In order to determine the k and λ values, a test query of one sequence is run against the entire non-segmented database.

3. METHODS DETAILS

3.1 Presence Vector

In the context of this paper a *Presence Vector* is a vector of bits in which the value at each position indicates the presence of a specific sequence *K-mer*. The index of each *K-mer* in the vector is trivial to compute.

3.2 Clustering

We considered three different clustering algorithms for use in K-mulus: K-means, K-medoids, and a modified form of hierarchical clustering. Pseudo-code for these algorithms is given below in addition to details about our hierarchical clustering algorithm

We used a modified form of *agglomerative clustering* to perform hierarchical clustering on the database sequences. This general algorithm uses a simple bottom-up approach, in which all sequences begin in disjoint clusters of size one, and are iteratively merged together with the nearest cluster [10]. Implementations of *agglomerative clustering* are differentiated by the distance function used to compare clusters.

Our algorithm performs clustering with respect to the *presence vectors* of each input sequence. For each cluster, a center *presence vector* is computed as the union of all sequence *presence vectors* in the cluster. The distance between clusters is taken as the Hamming distance, or number of bitwise differences, between these cluster centers. This design choice creates a tighter correspondence between the clustering algorithm and the metrics for success of the results, which depend entirely on the cluster *presence vectors* as computed above.

Note that this algorithm requires the intensive computation of all-pairs distances at every clustering iteration. For this reason, a novel algorithm was used to efficiently make these computations in the cloud. This algorithm is described in detail in the next section.

Algorithm 1 Hierarchical Clustering Algorithm (Modified)

```
1: Define each input presence vector as a cluster c
2: repeat
3:    for all Clusters c do
4:       Assign c.k as the union of all presence vectors in c
5:    end for
6:    Compute all pairwise distances for all c.k
7:    repeat
8:       Merge the two clusters c1, c2 with the shortest distance
9:    until All previous clusters c have been merged
10: until One cluster remains
```

3.3 Efficient Distributed All-Pairs

We initially considered clustering the query sequences, in addition to clustering the database sequences. We decided not to explore this option, for reasons mentioned elsewhere in the paper. However, we recognize that for some applications, such an approach would be useful. Query clustering may require all pairwise distances to be computed in parallel,

Algorithm 2 K-Means Algorithm

```
Select k centroids randomly
repeat
    for all i in the database do
        Assign i to the cluster with the closest centroid
    end for
    for all Clusters c do
        c ← (1/|c|) Σ_{s∈c} s
    end for
until Centroids do not move
```

Algorithm 3 K-Medoid Algorithm

```
Select k centroids randomly
repeat
    for all i in the database do
        Assign i to the cluster with the closest centroid
    end for
    for all Clusters c do
        for all Elements e in cluster c do
            Find e such that it minimizes the total distance
            to all other e in c
            Make e the new centroid of cluster c
        end for
    end for
until Centroids do not move
```

and for this reason we include our efficient distributed all-pairs algorithm which we used for clustering of the database sequences.

It is trivial to use Hadoop to perform exactly $\frac{n(n-1)}{2}$ comparisons. However, naive use of memory and network overhead can lead to run times which can be worse than serial [12]. Our algorithm maps a sizable number of comparisons to each node, which reduces overhead. Furthermore, for the given workloads, our algorithm minimizes network overhead and guarantees that each comparison occurs only once. An implication of this is that for all data a node receives, it may freely compute all pairs within that set without any global duplication of work. Proof of this point is trivial when you consider that in any other configuration (where some available comparisons are not executed at the node), fewer comparisons per sequence are being done at the node. It follows that such a configuration will require a strictly larger amount of overall data movement.

The above is achieved through projective geometry. In projective geometry a *projective plane*, the simplest of which is shown in Fig 3, has the following properties:

- For all pairs of distinct points, there is exactly one line which contains that pair.
- For any pair of lines, there is exactly one point at which they intersect.
- There exist four points such that no line intersects more than two of them.

Allow me to equate 'lines' to our task input groups, and 'points' to the data items. Using this transformation with bullet one, it follows that for all pairs of items, there is exactly one group which contains that pair. A configuration of data distribution such as this meets the condition listed above, and is therefore an optimal distribution of data for clusters of that size.

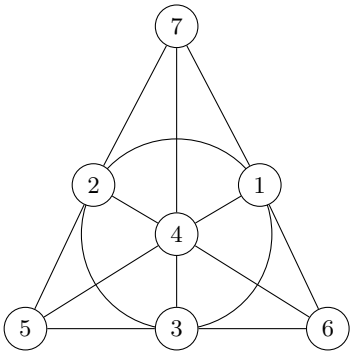

Figure 3: Fano Plane

Projective planes of order p can be calculate trivially, where p is a prime power and the plane has a capacity for $p^2 + p + 1$ points. There are some complications to this algorithm which arise as a result of this restriction on p. For input sizes which vary from this order, additional positions must be 'padded' out and some efficiency is lost. Overall, this algorithm is very efficient in the average case for performing intensive all pairs comparisons.

4. RESULTS

Common BLAST word sizes range from 2-5, thus we experimented with k-mers of size 2-5[15]. For all subsequent analyses we used 3-mers. While three different clustering algorithms were considered in this paper, all subsequent K-mulus results were generated with k-means clustering (explanation below). In BLAST, neighboring words are k-mers which are biologically similar to a given k-mer within a certain threshold. In our results, neighboring words are not considered during K-mulus database indexing. This had a negligible effect on results as nearly all queries already matched every cluster index.

4.1 Speedup Potential

First we will show that K-mulus achieves significant speedups on well clustered data. To demonstrate this we simulated an ideal data set of 1,000 sequences, where the sequences were comprised of one of two disjoint sets of 3-mers. The database sequences were clustered into two even-size clusters. The sample query was 10,000 sequences, also comprised of one of two disjoint sets of 3-mers. Table 1 shows the result of running BLAST on the query using Hadoop Streaming with query segmentation (the method CloudBLAST uses to run BLAST) and K-mulus. K-mulus running on 2 cores with 2 databases yields a 225% improvement over running BLAST using Hadoop Streaming on 2 cores. In practice this degree of separability is nearly impossible to replicate, but this model allows us to set an upper bound for the speedup contributed by clustering and search space reduction.

4.2 K-mulus in Practice

A more practical BLAST query would be using the *nr* database, containing 3,429,135 sequences. K-mulus is compared against the naive Hadoop streaming method using a realistic query of 30,000 sequences from the HMP project (Table 2). K-mulus performs poorly compared to the default Hadoop Streaming BLAST implementation because of the very high k-mer overlap between clusters. Due to the high k-mer overlap, each query sequence is being replicated and compared against nearly all clusters. Since the complete *nr* database can fit into memory, we achieve no benefit from segmenting the database. Running K-mulus with 100,000 clusters performs slightly worse compared to only 100 clusters because of the additional overhead of rebuilding the query index for each BLAST instance.

	Hadoop Streaming	K-mulus
1 core	15 mins, 38 sec	6 mins, 55 sec
2 cores	9 mins, 48 sec	4 mins, 25 sec

Table 1: Runtimes of BLAST using Hadoop Streaming and K-mulus on a query of 10,000 protein sequences. K-mulus is run with two clustered databases which contain no overlapping k-mers. K-mulus running on 2 cores with 2 databases yields a 225% improvement over running BLAST using Hadoop Streaming on 2 cores.

Method	Time
Hadoop Streaming	0 hr, 39 mins
K-mulus 100 clusters	1 hr, 50 mins
K-mulus 10,000 clusters	2 hr, 17 mins

Table 2: Runtimes of BLAST using Hadoop Streaming and K-mulus using the *nr* database on a query of 30,000 sequences on 100 cores. K-mulus is run using both 100 and 10,000 clusters to compare performance. Due to the low variance of k-mers between sequences in *nr*, K-mulus performs poorly compared to Hadoop Streaming. K-mulus with 10,000 clusters performs worse than K-mulus with 100 clusters because of the additional overhead of query indexing between BLAST instantiations.

5. DISCUSSION

In our *nr* query results, K-mulus shows poor performance due entirely to noisy, overlapping clusters. In the worst case, K-mulus will map every query to every cluster and devolve to a naive parallelized BLAST on database segments, while also including some overhead due to database indexing. This is close to the behaviour we observed when running K-mulus on the *nr* database. In order to describe the best possible clusters we could have generated from a database, we considered a lower limit on the exact k-mer overlap between single sequences in the *nr* database (Fig 4). We generated this plot by taking 50 random samples of 3000 *nr* sequences each, computing the pairwise k-mer intersection between them, and plotting a histogram of the magnitude of pairwise k-mer overlap. This shows that there are very few sequences in the *nr* database which have no k-mer overlap which makes the generation of disjoint clusters impossible. Furthermore, this plot is optimistic in that it does not include BLAST's neighboring words, nor does it illustrate comparisons against cluster centers which will have intersection greater than or equal to that of a single sequence.

In order to show the improvement offered by repeat masking, we ran SEG[20] on the sequences before computing the intersection. On average, SEG resulted in a 6% reduction in the number of exact k-mer overlap between two given sequences. Repeat masking caused a significant, favorable shift in k-mer intersection and would clearly improve clustering results. However, the *nr* database had so much existing k-mer overlap that using SEG preprocessing would have almost no effect on the speed of K-mulus.

While we considered three different clustering algorithms in this paper, we determined that the cluster overlap was so excessive that any difference between the clustering output was negligible. For this reason, we exclusively used K-means clustering for our analysis.

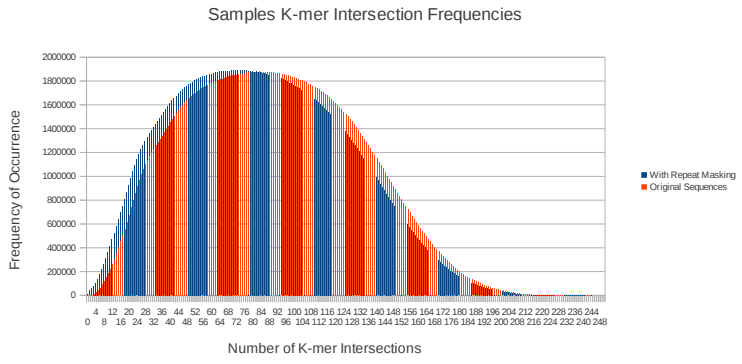

Figure 4: A histogram of the number of exact pairwise k-mer intersections within a set of *nr* database sequences. 50 samples of 3,000 random sequences were taken from *nr* and repeats were masked by SEG[20]. K-mer overlaps were plotted pairwise within each original sample and each masked sample. At a high level, this plot indicates k-mer variance in *nr*. Lower magnitudes of intersection imply better possible clusters and improved K-mulus performance.

6. FUTURE WORK

While K-mulus did not meet the goals set forth in this paper, it has great potential as a platform for improving distributed BLAST performance. The logical next step for K-mulus is nucleotide database indexing, which has historically had far more success than protein indexing in BLAST. With a four character alphabet and simplified substitution rules, nucleotides are much easier to work with than amino acids, and allow for much more efficient hashing by avoiding of the ambiguity inherent in amino acids. Furthermore

we expect a more random distribution of nucleotide k-mers than amino acids k-mers, which allows for better clustering. While we chose to pursue improvements to protein BLAST, in our analysis it has become clear that a K-mulus nucleotide BLAST not only has a promising outlook for a speedup, but could serve as a model for effectively executing the more complex task of protein clustering.

Our work did not include analysis of any clustering or indexing methods which would have resulted in a loss of BLAST search sensitivity. For example, K-mulus clustering might benefit from the positive results shown for protein k-mer indexing of large k-mers over compressed alphabets[17]. Another possible improvement would be to map queries according to percent k-mer identity to a cluster, or to raise the threshold for required k-mer overlap with a cluster index. While these approaches would reduce BLAST sensitivity, the trade off with search speed may be favorable. The motivation for this work comes from our evidence that if protein clustering in K-mulus can be improved, large speedups can be achieved.

7. REFERENCES

[1] *Amazon elastic compute cloud. http://aws.amazon.com/ec2/.*

[2] *Human microbiome project. http://commonfund.nih.gov/hmp/*, 2006.

[3] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene Myers, and David Lipman, *Basic local alignment search tool*, Journal of Molecular Biology **215** (1990), 403 – 410.

[4] Dhruba Borthakur, *The Hadoop Distributed File System: Architecture and Design*, The Apache Software Foundation, 2007.

[5] Aaron E. Darling, Lucas Carey, and Wu chun Feng, *The design, implementation, and evaluation of mpiblast*, In Proceedings of ClusterWorld 2003, 2003.

[6] Jeffrey Dean and Sanjay Ghemawat, *Mapreduce: simplified data processing on large clusters*, Commun. ACM **51** (2008), 107–113.

[7] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, *The google file system*, SIGOPS Oper. Syst. Rev. **37** (2003), 29–43.

[8] J. A. Hartigan and M. A. Wong, *A K-means clustering algorithm*, Applied Statistics **28** (1979), 100–108.

[9] Ying Li, Hong M. Luo, Chao Sun, Jing Y. Song, Yong Z. Sun, Qiong Wu, Ning Wang, Hui Yao, Andre Steinmetz, and Shi L. Chen, *Est analysis reveals putative genes involved in glycyrrhizin biosynthesis*, BMC Genomics **11** (2010), no. 1, 268+.

[10] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze, *Introduction to information retrieval*, Cambridge Univ. Press, New York, NY, 2008.

[11] Andréa Matsunaga, Maurício Tsugawa, and José Fortes, *Fourth ieee international conference on escience cloudblast: Combining mapreduce and virtualization on distributed resources for bioinformatics applications.*

[12] C. Moretti, J. Bulosan, D. Thain, and P.J. Flynn, *All-pairs: An abstraction for data-intensive cloud computing*, Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on, april 2008, pp. 1 –11.

[13] Aleksandr Morgulis, George Coulouris, Yan Raytselis, Thomas L. Madden, Richa Agarwala, and Alejandro A. Schäffer, *Database indexing for production megablast searches*, Bioinformatics **24** (2008), 1757–1764.

[14] J Murray, J Larsen, T E Michaels, A Schaafsma, C E Vallejos, and K P Pauls, *Identification of putative genes in bean (phaseolus vulgaris) genomic (bng) rflp clones and their conversion to stss.*, Genome **45** (2002), no. 6, 1013–24.

[15] NCBI, *Ncbi c++ toolkit book*, http://www.ncbi.nlm.nih.gov/books/NBK7160/pdf/TOC.pdf.

[16] J. Schloss, E. Mitchell, M. White, R. Kukatla, E. Bowers, H. Paterson, and S. Kresovich, *Characterization of rflp probe sequences for gene discovery and ssr development in sorghum bicolor (l.) moench.*, Theor Appl Genet **105** (2002), no. 6-7, 912–920.

[17] Sergey A. Shiryev, Jason S. Papadopoulos, Alejandro A. Schäffer, and Richa Agarwala, *Improved blast searches using longer words for protein seeding*, Bioinformatics **23** (2007), no. 21, 2949–2951.

[18] Mark Van Der Laan, Katherine Pollard, and Jennifer Bryan, *A new partitioning around medoids algorithm*, Journal of Statistical Computation and Simulation **73** (2003), no. 8, 575–584.

[19] Hugh E. Williams and Justin Zobel, *Indexing nucleotide databases for fast query evaluation*, EDBT, 1996, pp. 275–288.

[20] John C. Wootton and Scott Federhen, *Statistics of local complexity in amino acid sequences and sequence databases*, Computers and Chemistry **17** (1993), no. 2, 149 – 163.