BlastReduce: High Performance Short Read Mapping with MapReduce

Michael C. Schatz

University of Maryland
Center for Bioinformatics and Computational Biology
mschatz@umiacs.umd.edu

Abstract

Next-generation DNA sequencing machines generate sequence data at an unprecedented rate, but traditional single-processor sequence alignment algorithms are struggling to keep pace with them. *BlastReduce* is a new parallel read mapping algorithm optimized for aligning sequence data from those machines to reference genomes, for use in a variety of biological analyses, including SNP discovery, genotyping, and personal genomics. It is modeled after the widely used *BLAST* sequence alignment algorithm, but uses the open-source *Hadoop* implementation of *MapReduce* to parallelize execution to multiple compute nodes. To evaluate its performance, BlastReduce was used to map next generation sequence data to a reference bacterial genome in a variety of configurations. The results show *BlastReduce* scales linearly for the number of sequences processed, and with high speedup as the number of processors increases. In a modest 24 processor configuration, *BlastReduce* is up to 250x faster than *BLAST* executing on a single processor, and reduced the execution time from several days to a few minutes at the same level of sensitivity. Furthermore, *BlastReduce* is fully compatible with cloud computing, and can be easily executed on massively parallel remote resources to meet peak demand. *BlastReduce* is available open-source at: http://www.cbcb.umd.edu/software/blastreduce/.

1. Introduction

Next-generation high-throughput DNA sequencing technologies from 454, Illumina/Solexa, and Applied Biosystems are changing the scale and scope of genomics. These next generation sequencing machines can sequence more DNA in a few days than a traditional Sanger sequencing machine could in an entire year, and at a significantly lower cost [1]. James Watson's genome was recently sequenced [2] using technology from 454 Life Sciences in just four months, whereas previous efforts to sequence the human genome required several years and hundreds of machines [3]. If this trend continues, an individual will be able to have his or her DNA sequenced in only a few days and perhaps for as little as $1000.

The data from the new machines consists of millions of short sequences (25-250bp) of DNA called reads, collected randomly from the DNA sample. After sequencing, researchers will often align, or map, the reads to a reference genome, to find the locations where each read occurs in the reference sequence, allowing for a small number of differences. This can be used, for example, to catalog differences in one person's genome relative to the reference human genome, or compare the genomes of different species. These comparisons are used for a wide variety of biological analyses including genotyping, gene expression, metagenomics, comparative genomics, SNP discovery and personal genomics. These results are of great interest since even a single base pair difference between two genomes can have huge impact on the health of those organisms. As such, researchers are beginning to generate sequence data at an incredible rate, and desperately need inexpensive and highly scalable algorithms to analyze their data.

One of the most widely used programs for sequence alignment is *BLAST* [4]. It uses an algorithmic technique called *seed-and-extend* to quickly find highly similar alignments between sequences. During its 20 years of development, *BLAST* has been optimized for fast operation, but as a single processor algorithm, is struggling to keep pace with the volume of data generated by next generation sequencing machines. *BlastReduce* is a new parallel seed-and-extend alignment algorithm modeled on the serial *BLAST* algorithm. It is optimized for efficiently mapping reads from next generation sequence data to reference genomes, while allowing a small number of differences. *BlastReduce* uses the open-source implementation of *MapReduce* called *Hadoop* [5] to schedule, monitor, and manage the parallel execution. Our results show that even in a modest configuration of 24 processors *BlastReduce* is up to 250x faster than a single processor execution of *BLAST* for aligning large sets of Illumina/Solexa reads to a bacterial genome. Furthermore, *BlastReduce* can easily scale to execute on hundreds or thousands of remote processors using cloud computing, and thus can cut the execution time of even the largest read mapping task from days to minutes.

1.1 MapReduce

MapReduce [6] is the software framework invented by and used by Google to support parallel execution of their data intensive applications. Google uses this framework internally to execute thousands of *MapReduce* applications per day, each processing many terabytes of data all on commodity hardware. The framework automatically provides common services for parallel computing, such as the partitioning the input data, scheduling, monitoring, and inter-machine communication necessary for remote execution. As such, application developers for *MapReduce* need only write a small number of functions relevant to their problem domain, and the framework automatically executes those functions for parallel processing. Computation in *MapReduce* is divided into two major phases called *map* and *reduce*, separated by an internal grouping of the intermediate results. This two-phase computation was developed after recognizing that many different computations could be solved by first computing a partial result (map phase), and then combining those partial results into the final result (reduce phase). The power of *MapReduce* is the map and reduce functions are executed in parallel over potentially hundreds or thousands of processors with minimal effort by the application developer.

The first phase of computation, the map phase, computes a set of intermediate key-value pairs from the input data, as determined by a user-defined function. Application developers can implement any relationship applicable to their problem from the input including outputting multiple pairs for a given input. For example, if the overall computation was to count the number of occurrences of all of the English words in a large input text, the map function could output the key value pair *(w,1)* for each word *w* in the input text. If the input text is very large, many instances of the map function could execute in parallel on different portions of the input text.

After all of the map functions are complete, the *MapReduce* framework then internally sorts the pairs so all values with the same key are grouped together into a single list. It also partitions the data, and creates one file of key-value lists for each processor to be used in the reduce phase. The application can use standard object comparison and hash functions, or custom comparison and partition functions based on arbitrary computations of the key. Sorting the keys in this way is similar to constructing a large distributed hash table indexed by the key, with a list of values for each key. In the word frequency computation example, the framework creates a list of *1*s for each word in the input set corresponding to each instance of that word.

The reduce phase computes a new set of key-value pairs from the key-value lists generated by the map phase. These key-value pairs can serve as input for another *MapReduce* cycle, or output to the user. In the word frequency computation, the reduce function adds together the value lists of 1s to compute the number of occurrences of each word, and outputs the key-value pair (w, # occurrences). Each instance of the reduce function executes independently, so there can be as many reduce functions executing in parallel as there are distinct words in the input text. Since the words and the computations are independent, the total execution time should scale linearly with the number of processors available, i.e. a 10 processor execution should take $1/10^{th}$ the time of a 1 processor execution. In practice, it is very difficult to achieve perfect linear speedup, because Amdahl's law [7] limits the maximum speedup possible given some amount of overhead or serial processing. For example, if an application has just 10% non-parallelizable fixed overhead, than the maximum possible speedup is only 10x regardless of the number of processors used.

MapReduce uses data files for the inter-machine communication between map and reduce functions. This could become a severe bottleneck if a traditional file system was used and thousands of processors accessed to the same file at the same time. Instead, Google developed the robust distributed Google File System (GFS) [8] to support efficient *MapReduce* execution, even with extremely large data sets. The file system is designed to provide very high bandwidth, but may potentially have high latency for individual random reads and writes. This choice was deliberately made to match the access pattern of *MapReduce*, which almost exclusively requires bulk read and write access. Files in the GFS are automatically partitioned into large chunks, which are distributed and replicated to several physical disks attached to the compute nodes. Metadata and directory services are managed through a master IO node, which stores the location of each chunk of each file and grants access to those chunks. Furthermore, *MapReduce* is data-aware and attempts to schedule computation at the compute nodes that store the required data whenever possible. Therefore, aggregate IO performance can greatly exceed the performance of an individual drive, and chunk redundancy ensures reliability even when used with inexpensive commodity drives with relatively high failure rates.

Hadoop and the *Hadoop Distributed File System* (*HDFS*) [5] are an open source version of *MapReduce* and the *Google File System* sponsored by Yahoo, Google, IBM, Amazon, and other major vendors. Like Google's proprietary *MapReduce* framework, applications developers need only write custom *map* and *reduce* functions, and the *Hadoop* framework

automatically executes those functions in parallel. *Hadoop* is implemented in Java, and is bundled with a large number of support classes to simplify code development. *Hadoop* and *HDFS* is used to manage clusters with 10,000+ nodes and operating on petabytes of data, and supports, in part, every Yahoo search result [9].

In addition to in-house *Hadoop* usage, *Hadoop* is quickly becoming a de-facto standard for cloud computing. Cloud computing is a model of parallel computation where compute resources are accessed generically, without regard for physical location or specific configuration. The generic nature of cloud computing allows resources to be purchased on-demand, such as to augment local resources for specific large or time critical tasks. Several companies now sell cloud compute cycles that can be accessed via *Hadoop*. For example, Amazon's Elastic Compute Cloud [10] contains tens of thousands of processors priced at $.10 per hour per processor, and supports *Hadoop* with minimal effort.

1.2 Read Mapping

After sequencing DNA the newly created reads are often aligned or mapped to a reference genome to find the locations where each read approximately occurs. A read mapping algorithm reports all alignments that are within a scoring threshold, commonly expressed as the maximum acceptable number of differences between the read and the reference genome (generally at most 1%-10% of the read length). The alignment algorithm can allow just mismatches as differences, the k-mismatch problem, or it can also consider gapped alignments with insertions or deletions of characters, the k-difference problem). The classical Smith-Waterman sequence alignment algorithm [11] computes gapped alignments using dynamic programming. It considers all possible alignments of a pair of sequences in time proportional to the product of their lengths. A variant of the Smith-Waterman algorithm, called a banded alignment, uses essentially the same dynamic programming but restrict the search to those alignments that have a (small) fixed number of differences. For a single pair of sequences computing a Smith-Waterman alignment is usually a fast operation, but becomes computational infeasible as the number of sequences increases.

Instead researchers use a technique called *seed-and-extend* to accelerate the search for highly similar alignments. The key insight is the observation that within highly similar alignments there must be significant exact alignments. By the pigeon-hole principle, for a 20bp read to align with only 1 difference, there must be at least a 10bp exact alignment someplace in the alignment. In general, a full-length alignment of a m bp read with at most e mismatches must contain at least 1 $m/(e+1)$ bp exact alignment. Several sequence alignment algorithms, including the very popular *BLAST* [4] and *MUMmer* [12] tools use this technique to accelerate alignment. In the seed phase, the tools find substrings that are shared between the sequences. For example, *BLAST* constructs a hash table of fixed length overlapping substrings called *k-mers* of the reference sequences to find seeds, and *MUMmer* constructs a suffix tree of the reference sequences to find variable length maximal exact substrings as seeds. Then in the extension phase, the tools compute more expensive in-exact banded Smith-Waterman alignments restricted to the relatively short substrings near the shared seeds. This technique can greatly decrease the time required to align sequences at a given level of sensitivity. However, as the sensitivity increases by allowing more differences, the length of the seed will decrease, and thus the number of randomly matching seeds will increase as will the total execution time.

The Landau-Vishkin k-difference alignment algorithm [13] is an alternative dynamic programming algorithm for determining if two strings align with at most k-differences. Unlike the Smith-Waterman dynamic programming algorithm, which considers all possible alignments, the Landau-Vishkin algorithm considers only the most similar alignments up to a fixed number of differences by computing how many characters of each string can be aligned with i=0 to k differences. The number of characters that can be aliged using i differences is computed from the (i-1) result by computing the exact extensions possible after introducing 1 mismatch, 1 insertion or 1 deletion from the end of the (i-1) alignment. The algorithm ends when $i=k+1$, indicating no k-difference alignment exists for those sequences, or the end of the sequence is reached. This algorithm is much faster than the full Smith-Waterman algorithm for small values of k, since only a small number of potential alignments are considered. See Gusfield's classical text on sequence alignment for more information [14].

2. BlastReduce Algorithm

BlastReduce is a parallel read mapping algorithm implemented in Java with *Hadoop*. It is modeled on the *BLAST* algorithm, and is optimized for mapping short reads from next generation sequencing machines to a reference genome. Like *BLAST*, it is a seed-and-extend alignment algorithm, and uses fixed length mers as seeds. Unlike *BLAST*, *BlastReduce* uses the Landau-Vishkin algorithm to extend the exact seeds and quickly find alignments with at most k-differences. This

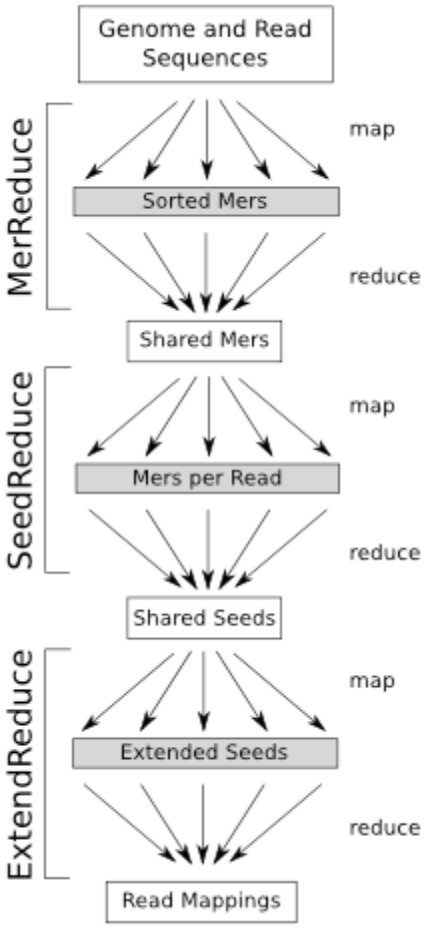

Figure 1. Overview of the BlastReduce algorithm using 3 MapReduce cycles. Intermediate files used internally by MapReduce are shaded.

extension algorithm is more appropriate for short reads with a small number of differences (typically k=1 or k=2 for 25-50bp reads). The seed size (s) is automatically computed based on the length of the reads and the maximum number of differences (k) as specified by the user.

The input to the application is a multi-fasta file containing the reads and a multi-fasta file containing one or more reference sequences. These files are first converted to a compressed *Hadoop SequenceFile* suitable for processing with *Hadoop*. *SequenceFile*'s do not natively support sequences with more than 65,565 characters so long sequences are partitioned into chunks. The sequences are stored as key-value pairs in the *SequenceFile* as *(id, SeqInfo)* where *SeqInfo* is the tuple *(sequence, start_offset, tag))* where *start_offset* is the offset of the chunk within the full sequence. The chunks overlap by *s-1* bp so that all seeds are present only once, and reference sequences are indicated with *tag=1*. After conversion, the *SequenceFile* is copied into the HDFS so the main read mapping algorithm can execute.

The read mapping algorithm requires 3 *MapReduce* cycles, as described below (Figure 1). The first 2 cycles, MerReduce and SeedReduce, find all maximal exact matches at least *s* bp long, and the last cycle, ExtendReduce, extends those seeds with the Landau-Vishkin algorithm into all alignments with at most k differences.

1. MerReduce: Compute Shared Mers

This *MapReduce* cycle finds mers of length *s* that are shared between the reads and the reference sequences. The map function executes on every sequence chunk independently, and mers that only occur in the reads or only in the reference are automatically discarded. ExtendReduce needs the sequence flanking the exact seeds for the alignment, but HDFS is inefficient for random access. Therefore, the flanking sequences (up to read_length – s + k bp) are included with the read and reference mers so they will be available when needed.

Map: For each mer in the input sequence, the map function outputs *(mer, MerPos),* where *MerPos* is the tuple *(id, position, tag, left_flank, right_flank))*. If the sequence is a read (tag=0) also output the *MerPos* records for the reverse complement sequences. The map function outputs s(M+N) mers total, where M is the total length of the reads, and N is the total length of the reference sequences. After all of the map function have completed, *Hadoop* will internally sort the key-value pairs, and group together all of the pairs with the same mer sequence into a single list of MerPos records.

Reduce: The reduce function outputs position information about mers that are shared by at least 1 reference sequence and 1 read. It requires 2 passes through each list of MerPos records for each mer. It first scans the list to find MerPos records from a reference. Then it scans the list a second time and outputs a *(read_id, SharedMer)* key-value pair for each mer that occurs in a read and a reference sequence. A *SharedMer* is the tuple *(read_position, ref_id, ref_position, read_left_flank, read_right_flank, ref_left_flank, ref_right_flank)*.

2. SeedReduce: Coalesce Consistent Mers

This *MapReduce* cycle reduces the number of seeds by merging consistent shared mers into larger seeds. Two shared mers are consistent if they are 1 bp offset in the read and the reference. Those 2 consistent mers can be

safely merged since they must refer to the same alignment.

Map: The map function outputs the same *(read_id, SharedMer)* pairs as are input. After the map function is complete, all *SharedMer* records from a given read are internally grouped together for the reduce phase.

Reduce: Each list of *SharedMer* records is first sorted by read position, and consistent mers are collasced into seeds. The final seeds are all maximal exact matches at least *s* bp long. They are output as the pairs *(read_id, SharedSeed)* where *SharedSeed* is the tuple *(read_position, seed_length, ref_id, target_position, read_left_flank, read_right_flank, ref_left_flank, ref_right_flank)*.

3. ExtendReduce: Extend Seeds

This MapReduce cycle extends the exact alignment seeds into a longer inexact alignment using the Landau-Vishkin k-difference algorithm.

Map: For each *SharedSeed*, the code attempts to extend the shared seed into and end-to-end alignment with at most k-differences. If such an alignment exists, output the pair *(read_id, AlignmentInfo)*, where *AlignmentInfo* is the tuple *(ref_id, ref_align_start, ref_align_end, num_differences)*. After all of the map functions are complete, *Hadoop* groups all *AlignmentInfo* records from the same read for the reduce function.

Reduce: The reduce function filters duplicate alignments, since there may be multiple seeds in the same alignment. For each read, it first sorts the *AlignmentInfo* records by *ref_align_start*, and then in a second pass, outputs unique *(read_id, AlignmentInfo)* pairs that have difference *ref_align_start* fields.

The output of the ExtendReduce is a file containing every alignment of every read with at most k-differences. This file can be copied from HDFS to a regular file system, or the HDFS file can be post-processed with the bundled reporting tools.

3. Results

We evaluated how well *BlastReduce* performs in a variety of configurations for the task of mapping many short reads to a reference genome while allowing a small amount of differences. For this evaluation, *BlastReduce* mapped random subsets of 2,726,374 publically available Illumina/Solexa sequencing reads to the corresponding 2.0 Mbp *S. suis* P1/7 genome [15] allowing for either 1 or 2 differences. The reads are all 36 bp long, and represent a single lane of data from a

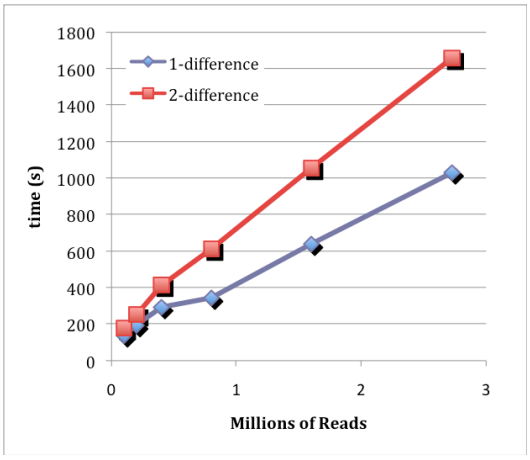

Figure 2. Running Time vs Number of Reads. BlastReduce scales linearly as the number of reads increases.

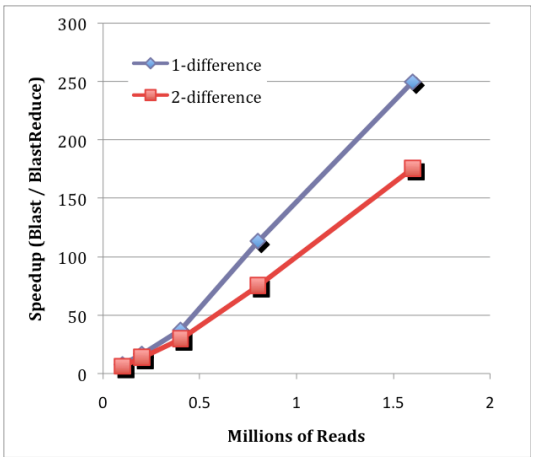

Figure 3. Blast Speedup vs Number of Reads. BlastReduce dramatically outperforms BLAST for large read sets.

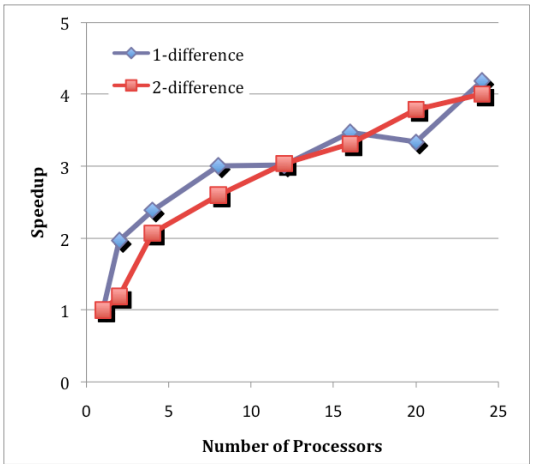

Figure 4. BlastReduce Speedup vs. Number of Processors. BlastReduce continues to have higher speedup through 24 processors configurations.

full Illumina/Solexa run. The test cluster has 12 compute nodes, each with a dual core 3.2 GHz Intel Xeon (24 processors total) and 250 GB of local disk space available. The compute nodes were running RedHat AS Release 3 Update 4, and *Hadoop* 0.15.3 set to execute 2 tasks per node. In the results below, the time to convert and load the data into the HDFS is excluded, since this time was the same for all tasks. In addition, once the data is loaded into HDFS, it can be reused for multiple analyses, similar to the pre-processing needed for searching with *BLAST*.

The first test explored how well *BlastReduce* scales as the number of reads increases. In this test, all 24 processors were used to align subsets of the reads allowing for 1 difference (seed length=18) or 2 differences (seed length=12). Figure 2 shows the runtime of these tasks. The results show *BlastReduce* scales linearly in execution time as the number of reads increases, as expected since the reads are independent. The 2-difference mapping take substantially longer than the 1-difference mapping (30-70% more), since there are many more seeds to consider. Extrapolating the curves to the y-axis shows there is approximately 90 seconds in overhead for *Hadoop* to schedule to all of the compute nodes, process the genome sequence, and map the first read. Thus, *BlastReduce* performs best only when there is a large number of reads to processes. However, this should not be a problem for *BlastReduce*, since large workloads are expected when used in conjunction with new sequencing technologies.

The second test compared the performance of the parallel execution of *BlastReduce* on 24 processors to a serial execution of NCBI *BLAST* (version 2.2.12) on the same hardware as the *BlastReduce*. *BLAST* was configured to use the same seed lengths as *BlastReduce*, 18 bp (-W 18) for the 1-difference alignment, and 12 bp (-W 12) for the 2-difference alignment. Both configurations disabled the low complexity filtering (-F F) since researchers often want to find slight differences between otherwise high copy repeats. BLAST was configured to use a compact tabular output format (-w 8) very similar to the *BlastReduce* output format.

Figure 2 shows the results of the test, and plots the speedup of *BlastReduce* over *BLAST* as the number of reads increase. BLAST requires more than 2 days to map all 2.7M reads, so that comparison was excluded. Nevertheless with just 1.6M reads, *BlastReduce* is 175x-250x times faster than *BLAST* for 2- and 1-difference read mappings, and is likely to be significantly faster for even larger data sets. Since *BlastReduce* was running on 24 processors, the expected speedup was only 24x. The extra order of magnitude super-linear speedup is most likely because of both algorithm and computational resources changes. Specifically, *BlastReduce* uses the Landau-Vishkin k-difference algorithm for extending the seeds, while *BLAST* uses a banded Smith-Waterman algorithm that stops extending when the alignment score drops below a threshold rather than a specific after a fixed number of differences. As such, *BlastReduce* can filter extensions with more than the desired number of differences immediately, but noisy read mappings with more than this number of differences must be manually filtered after *BLAST* is complete. Failure to filter those high noise mappings is necessary to ensure all mappings at the same level of similarity are found. Finally, by running on 24 processors, *BlastReduce* has 24x as much IO and CPU caching, both of which can boost performance by allowing a larger fraction of the data to fit in cache, and thus faster access to that data.

The final test explored how well *BlastReduce* scales as the number of processors increases for a fixed problem size. Figure 3 shows the speedup of *BlastReduce* mapping 100,000 reads on between 1 and 24 processors, and for 1- and 2- difference alignments. *Hadoop* allows an application to specify the number of processors for map or reduce functions, but there is no mechanism to control where the map or reduce functions run. Consequently, these operations may run on physically different machines, so each data point indicates the number of processors used for the map or reduce functions, but not necessarily the total number of processors used. The full *BlastReduce* execution may be on a larger number of processors, up to the full 24 processor capacity of the cluster. However, since the reduce functions can only operate after the map functions have completed potentially using different processors for the map and reduce functions should have minimal impact on the overall performance.

The speedup curves in Figure 3 show the results of the test, and shows the speedup of the two alignment sensitivity levels are nearly the same. In both cases *BlastReduce* achieves approximately 1/6 the performance of ideal speedup for 24 processors (24x speedup). This speedup is explained by large amount of overhead (~90s) on 24 processors relative to the total runtime (137s). For larger problem sizes with more reads and higher execution time 24 processors would likely show improved speedup. Furthermore, even for 100,000 reads the speedup curves do not appear to have reached the plateau described by Amdahl's law, suggesting additional processors would further improve the performance of the application for this sized problem. Larger problems would delay reaching this plateau until even later.

4. Discussion

BlastReduce is a new parallel read mapping algorithm optimized for next generation short read data. It uses a seed-and-extend alignment algorithm similar to *BLAST* to efficiently find alignments with a small number of differences. The alignment algorithm is implemented as 3 consecutive *MapReduce* computations, each of which can be executed in parallel to many compute nodes. The performance experiments show *BlastReduce* is highly scalable to large sets of reads and has high speedup in a variety of processor configurations. In a modest configuration with 24 processors, *BlastReduce* is up to 250x faster than *BLAST* running on a single processor, but at the same level of sensitivity.

BlastReduce's high performance is made possible by the *Hadoop* implementation of the *MapReduce* framework. This framework makes it straightforward to create massively scalable applications, since the common aspects of parallel computing are automatically provided, and only the application specific code must be written. *Hadoop*'s ability to deliver high-throughput, even in the face of extremely large data sets, is a perfect match for many algorithms in computational biology. Consequently, *Hadoop*-based implementations of those algorithms are expected in the near future with similar order of magnitude speedups. Massively parallel applications, running on cloud clusters with tens of thousands of nodes will drastically change the scale and scope of DNA sequence analysis, and allow researchers to perform analyses that are otherwise impossible.

5. Acknowledgements

I would like to thank Jimmy Lin for introducing me to *Hadoop* in his Cloud Computing Class, Arthur Delcher for his helpful discussions, and Steven Salzberg for reviewing the manuscript. I would also like to thank the generous hardware support of IBM and Google via the Academic Cloud Computing Initiative used in the development of *BlastReduce*.

6. References

1. Shaffer, C., *Next-generation sequencing outpaces expectations.* Nat Biotechnol, 2007. **25**(2): p. 149.
2. Wheeler, D.A., et al., *The complete genome of an individual by massively parallel DNA sequencing.* Nature, 2008. **452**(7189): p. 872-6.
3. Venter, J.C., et al., *The sequence of the human genome.* Science, 2001. **291**(5507): p. 1304-51.
4. Altschul, S.F., et al., *Basic local alignment search tool.* J Mol Biol, 1990. **215**(3): p. 403-10.
5. *Hadoop*. [cited; Available from: http://hadoop.apache.org/.
6. Jeffrey, D. and G. Sanjay, *MapReduce: simplified data processing on large clusters.* Commun. ACM, 2008. **51**(1): p. 107-113.
7. Krishnaprasad, S., *Uses and abuses of Amdahl's law.* J. Comput. Small Coll., 2001. **17**(2): p. 288-293.
8. Sanjay, G., G. Howard, and L. Shun-Tak, *The Google file system*, in *Proceedings of the nineteenth ACM symposium on Operating systems principles*. 2003, ACM: Bolton Landing, NY, USA.
9. *Yahoo Launches Worlds Largest Hadoop Production Application*. [cited; Available from: http://developer.yahoo.com/blogs/hadoop/2008/02/yahoo-worlds-largest-production-hadoop.html.
10. *Running Hadoop MapReduce on Amazon EC2 and Amazon S3*. [cited; Available from: http://developer.amazonwebservices.com/connect/entry.jspa?externalID=873.
11. Smith, T.F. and M.S. Waterman, *Identification of common molecular subsequences.* J Mol Biol, 1981. **147**(1): p. 195-7.
12. Kurtz, S., et al., *Versatile and open software for comparing large genomes.* Genome Biol, 2004. **5**(2): p. R12.
13. Landau, G.M. and U. Vishkin, *Introducing efficient parallelism into approximate string matching and a new serial algorithm*, in *Proceedings of the eighteenth annual ACM symposium on Theory of computing*. 1986, ACM: Berkeley, California, United States.
14. Gusfield, D., *Algorithms on strings, trees, and sequences: computer science and computational biology*. 1997: Cambridge University Press. 534.
15. *Streptococcus Suis Sequencing Webpage*. [cited; Available from: http://www.sanger.ac.uk/Projects/S_suis/.