BIOINFORMATICS ORIGINAL PAPER Vol. 25 no. 11 2009, pages 1363–1369
doi:10.1093/bioinformatics/btp236

Sequence analysis

CloudBurst: highly sensitive read mapping with MapReduce

Michael C. Schatz*

Center for Bioinformatics and Computational Biology, University of Maryland, College Park MD 20742, USA

Received on November 25, 2008; revised on March 12, 2009; accepted on April 03, 2009

Advance Access publication April 8, 2009

Associate Editor: Alfenso Valencia

ABSTRACT

Motivation: Next-generation DNA sequencing machines are generating an enormous amount of sequence data, placing unprecedented demands on traditional single-processor read-mapping algorithms. CloudBurst is a new parallel read-mapping algorithm optimized for mapping next-generation sequence data to the human genome and other reference genomes, for use in a variety of biological analyses including SNP discovery, genotyping and personal genomics. It is modeled after the short read-mapping program RMAP, and reports either all alignments or the unambiguous best alignment for each read with any number of mismatches or differences. This level of sensitivity could be prohibitively time consuming, but CloudBurst uses the open-source Hadoop implementation of MapReduce to parallelize execution using multiple compute nodes.

Results: CloudBurst's running time scales linearly with the number of reads mapped, and with near linear speedup as the number of processors increases. In a 24-processor core configuration, CloudBurst is up to 30 times faster than RMAP executing on a single core, while computing an identical set of alignments. Using a larger remote compute cloud with 96 cores, CloudBurst improved performance by >100-fold, reducing the running time from hours to mere minutes for typical jobs involving mapping of millions of short reads to the human genome.

Availability: CloudBurst is available open-source as a model for parallelizing algorithms with MapReduce at http://cloudburst-bio.sourceforge.net/.

Contact: mschatz@umiacs.umd.edu

1 INTRODUCTION

Next-generation high-throughput DNA sequencing technologies from 454 Life Sciences, Illumina, Applied Biosystems and others are changing the scale and scope of genomics. These machines sequence more DNA in a few days than a traditional Sanger sequencing machine could in an entire year, and at a significantly lower cost (Shaffer, 2007). James Watson's genome was recently sequenced (Wheeler *et al*., 2008) using technology from 454 Life Sciences in just 2 months, whereas previous efforts to sequence the human genome required several years and hundreds of machines (Venter *et al*., 2001). If this trend continues, an individual will be able to have their 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 of DNA (25–250 bp) called reads, collected randomly from the target genome. After sequencing, researchers often map the reads to a reference genome to find the locations where each read occurs, allowing for a small number of differences. This information can be used to catalog differences in one person's genome relative to a reference human genome, or compare the genomes of closely related species. For example, this approach was recently used to analyze the genomes of an African (Bentley *et al*., 2008) and an Asian (Wang *et al*., 2008) individual by mapping 4.0 and 3.3 billion 35 bp reads, respectively, to the reference human genome. These comparisons are used for a wide variety of biological analyses including SNP discovery, genotyping, gene expression, comparative genomics and personal genomics. Even a single base pair difference can have a significant biological impact, so researchers require highly sensitive mapping algorithms to analyze the reads. As such, researchers are generating sequence data at an incredible rate and need highly scalable algorithms to analyze their data.

Many of the currently used read-mapping programs, including BLAST (Altschul *et al*., 1990), SOAP (Li *et al*., 2008b), MAQ (Li, *et al*., 2008a), RMAP (Smith *et al*., 2008) and ZOOM (Lin *et al*., 2008), use an algorithmic technique called *seed-and-extend* to accelerate the mapping process. These programs first find sub-strings called seeds that exactly match in both the reads and the reference sequences, and then extend the shared seeds into longer, inexact alignments using a more sensitive algorithm that allows for mismatches or gaps. These programs use a variety of methods for finding and extending the seeds, and have different features and performance. However, each of these programs is designed for execution on a single computing node, and as such requires a long running time or limits the sensitivity of the alignments they find.

CloudBurst is a new highly sensitive parallel seed-and-extend read-mapping algorithm optimized for mapping single-end next generation sequence data to reference genomes. It reports all alignments for each read with up to a user-specified number of differences including both mismatches and indels. *CloudBurst* can optionally filter the alignments to report the single best non-ambiguous alignment for each read, and produce output identical to RMAPM (RMAP using mismatch scores). As such *CloudBurst* can replace RMAP in a data analysis pipeline without changing the results, but provides much greater performance by using the open-source implementation of the distributed programming framework *MapReduce* called *Hadoop* (http://hadoop.apache.org). The results presented below show that *CloudBurst* is highly scalable: the running times scale linearly as the number of reads increases, and with near linear speed improvements

*To whom correspondence should be addressed.

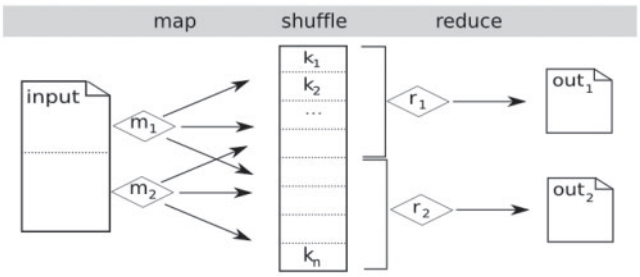

Fig. 1. Schematic overview of MapReduce. The input file(s) are automatically partitioned into chunks depending on their size and the desired number of mappers. Each mapper (shown here as m_1 and m_2) executes a user-defined function on a chunk of the input and emits key–value pairs. The shuffle phase creates a list of values associated with each key (shown here as k_1, k_2 and k_n). The reducers (shown here as r_1 and r_2) evaluate a user-defined function for their subset of the keys and associated list of values, to create the set of output files.

over a serial execution of RMAP for sensitive searches. Furthermore, *CloudBurst* can scale to run on large remote compute clouds, and thus map virtually any number of reads with high sensitivity in relatively little time.

1.1 MapReduce and Hadoop

MapReduce (Dean *et al*., 2008) is the software framework developed and used by Google™ to support parallel distributed execution of their data intensive applications. Google uses this framework internally to execute thousands of *MapReduce* applications per day, processing petabytes of data, all on commodity hardware. Unlike other parallel computing frameworks, which require application developers explicitly manage inter-process communication, computation in *MapReduce* is divided into two major phases called *map* and *reduce*, separated by an internal *shuffle* phase of the intermediate results (Fig. 1), and the framework automatically executes those functions in parallel over any number of processors.

The *map* function computes key–value pairs from the input data, based on any relationship applicable to the problem, including computing multiple pairs from a single input. For example, the map function of a program that counts the number of occurrences of all length k substrings (k-mers) in a set of DNA sequences could emit the key–value pair (*k-mer*, 1) for each k-mer. If the input is large, many instances of the map function can execute in parallel on different portions of the input and divide the running time by the number of processors available. Once the mappers are complete, *MapReduce* shuffles the pairs so all values with the same key are grouped together into a single list. The grouping of key–value pairs effectively creates a large distributed hash table indexed by the key, with a list of values for each key. In the k-mer counter example, the framework creates a list of 1s for each k-mer in the input, corresponding to each instance of that k-mer. The reduce function evaluates a user-defined function on each key–value list. The reduce function can be arbitrarily complex, but must be commutative, since the order of elements in the key–value list is unstable. In the k-mer counting example, the reduce function is called once for each k-mer with its associated list of 1s, and simply adds the 1s together to compute the total number of occurrences for that k-mer. Each instance of the reduce function executes independently, so there can be as many reduce functions executing in parallel as there are distinct keys, i.e. k-mers in the input.

As an optimization, *MapReduce* allows reduce-like functions called *combiners* to execute in-memory immediately after the *map* function. *Combiners* are not possible in every application because they evaluate on a subset of the values for a given key, but when possible, reduce the amount of data processed in the shuffle and reduce phases. In the k-mer counting example, the *combiner* emits a partial sum from the subset of 1s it evaluates, and the *reduce* function sums over the list of partial sums.

Computations in *MapReduce* are independent, so the wall clock running time should scale linearly with the number of processor cores available, i.e. a 10-core execution should take 1/10th the time of a 1-core execution creating a 10× speedup with complete parallel efficiency. In practice, perfect linear speedup is difficult to achieve because serial overhead limits the maximum speedup possible as described by Amdahl's law (Krishnaprasad, 2001). For example, if an application has just 10% non-parallelizable overhead, then the maximum possible end-to-end speedup is only 10× regardless of the number of cores used. High speedup also requires the computation is evenly divided over all processors to maximize the benefit of parallel computation. Otherwise the wall clock running time will be limited to the time for the longest running task, and reduce overall efficiency. *MapReduce* tries to balance the workload by assigning each *reducer* $\sim 1/N$ of the total key space, where N is the number of cores. If certain keys require substantially more time than others, however, it may be necessary to rebalance the workload using a custom partition function or adjusting how keys are emitted.

MapReduce is designed for computations with extremely large datasets, far beyond what can be stored in RAM. Instead it uses files for storing and transferring intermediate results, including the inter-machine communication between *map* and *reduce* functions. This could become a severe bottleneck, so Google developed the robust distributed Google File System (GFS) (Ghemawat *et al*., 2003) to efficiently support *MapReduce.* GFS is designed to provide very high-bandwidth for *MapReduce* by replicating and partitioning files across many physical disks. Files in the GFS are automatically partitioned into large chunks (64 MB by default), which are replicated to several physical disks (three by default) attached to the compute nodes. Therefore, aggregate I/O performance can greatly exceed the performance of an individual memory storage device (e.g. a disk drive), and chunk redundancy ensures reliability even when used with commodity drives with relatively high-failure rates. *MapReduce* is also 'data aware': it attempts to schedule computation at a compute node that has the required data instead of moving the data across the network.

Hadoop and the *Hadoop Distributed File System* (*HDFS*) are open source versions of *MapReduce* and the GFS implemented in Java and sponsored by Amazon™, Yahoo™, Google, IBM™ 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* and *HDFS* are used to manage production clusters with 10 000 + nodes and petabytes of data, including computation supporting every Yahoo search result. A Hadoop cluster of 910 commodity machines recently set a performance record by sorting 1 TB of data (10 billion 100 bytes records) in 209 s (http://www.hpl.hp.com/hosted/sortbenchmark/).

In addition to in-house *Hadoop* usage, *Hadoop* is becoming a *de facto* standard for cloud computing where compute resources are accessed generically as a service, without regard for physical location or specific configuration. The generic nature of cloud computing allows resources to be purchased on-demand, especially to augment local resources for specific large or time-critical tasks. Several organizations offer cloud compute cycles that can be accessed via *Hadoop*. Amazon's Elastic Compute Cloud (EC2) (http://aws.amazon.com) contains tens of thousands of virtual machines, and supports *Hadoop* with minimal effort. In EC2, there are five different classes of virtual machines available providing different levels of CPU, RAM and disk resources with price ranging from $0.10 to $0.80 per hour per virtual machine. Amazon offers preconfigured disk images and launches scripts for initializing a *Hadoop* cluster, and once initialized, users copy data into the newly created HDFS and execute their jobs as if the cluster was dedicated for their use. For very large datasets, the time required for the initial data transfer can be substantial, and will depend on the bandwidth of the cloud provider. Once transferred into the cloud, though, the cloud nodes generally have very high-internode bandwidth. Furthermore, Amazon has begun mirroring portions of Ensembl and GenBank for use within EC2 without additional storage costs, thereby minimizing the time and cost to run a large-scale analysis of these data.

1.2 Read mapping

After sequencing DNA, researchers often map the reads to a reference genome to find the locations where each read occurs. The read-mapping algorithm reports one or more alignments for each read within a scoring threshold, commonly expressed as the minimal acceptable significance of the alignment, or the maximum acceptable number of differences between the read and the reference genome. The algorithms generally allow 1–10% of the read length to differ from the reference, although higher levels may be necessary when aligning to more distantly related genomes, or when aligning longer reads with higher error rates. Read-mapping algorithms can allow mismatch (mutation) errors only, or they can allow insertion or deletion (indel) errors, for both true genetic variations and artificial sequencing errors. The number of mismatches between a pair of sequences can be computed with a simple scan of the sequences, whereas computing the edit distance (allowing for indels) requires a more sophisticated algorithm such as the Smith–Waterman sequence alignment algorithm (Smith *et al*., 1981), whose runtime is proportional to the product of the sequence lengths. In either case, the computation for a single pair of short sequences is fast, but becomes costly as the number or size of sequences increases.

When aligning millions of reads generated from a next-generation sequencing machine, read-mapping algorithms often use a technique called *seed-and-extend* to accelerate the search for highly similar alignments. This technique is based on the observation that there must be a significant exact match for an alignment to be within the scoring threshold. For example, for a 30 bp read to map to a reference with only one difference, there must be at least 15 consecutive bases, called a seed, that match exactly regardless of where the difference occurs. In general, a full-length end-to-end alignment of an m bp read with at most k differences must contain at least one exact alignment of $m/(k+1)$ consecutive bases (Baeza-yates *et al*., 1992). Similar arguments can be made when designing spaced seeds of non-consecutive bases to guarantee finding all alignments with up to a certain numbers of errors (Lin *et al*., 2008). Spaced seeds have the advantage of allowing longer seeds at the same level of sensitivity, although multiple spaced seeds may be needed to reach full sensitivity.

In all *seed-and-extend* algorithms, regions that do not contain any matching seeds are filtered without further examination, since those regions are guaranteed to not contain any high-quality alignments. For example, BLAST uses a hash table of all fixed length k-mers in the reference to find seeds, and a banded version of the Smith–Waterman algorithm to compute high-scoring gapped alignments. RMAP uses a hash table of non-overlapping k-mers of length $m/(k+1)$ in the reads to find seeds, while SOAP, MAQ and ZOOM use spaced seeds. In the extension phase, RMAP, MAQ, SOAP and ZOOM align the reads to allow up to a fixed number of mismatches, and SOAP can alternatively allow for one continuous gap. Other approaches to mapping include using suffix trees (Kurtz *et al*., 2004; Schatz *et al*., 2007) to quickly find short exact alignments to seed longer inexact alignments, and Bowtie (Langmead *et al*., 2009) uses the Burrows–Wheeler transform (BWT), to find exact matches coupled with a backtracking algorithm to allow for mismatches. Some BWT-based aligners are reporting extremely fast runtimes, especially in configurations that restrict the sensitivity of the alignments or limit the number of alignments reported per read. For example, in their default high-speed configuration, SOAP2 (http://soap.genomics.org.cn/), BWA (http://maq.sourceforge.net) and Bowtie allow at most two differences in the beginning of the read, and report a single alignment per read selected randomly from the set of acceptable alignments. In more sensitive or verbose configurations, the programs can be considerably slower (http://bowtie-bio.sourceforge.net/manual.shtml).

After computing end-to-end alignments, some of these programs use the edit distance or read quality values to score the mappings. In a systematic study allowing up to 10 mismatches, Smith *et al*. (2008) determined allowing more than two mismatches is necessary for accurately mapping longer reads, and incorporating quality values also improves accuracy. Several of these programs, including RMAPQ (RMAP with quality), MAQ, ZOOM and Bowtie, use quality values in their scoring algorithm, and all are more lenient of errors in the low-quality 3' ends of the reads by trimming the reads or discounting low-quality errors.

Consecutive or spaced seeds dramatically accelerate the computation by focusing computation to regions with potential to have a high-quality alignment. However, to increase sensitivity the length of the seeds must decrease (consecutive seeds) or the number of seeds used must increase (spaced seeds). In either case, increasing sensitivity increases the number of randomly matching seeds and increases the total execution time. Decreasing the seed length can be especially problematic because a seed of length s is expected to occur $\sim L/4^s$ times in a reference of length L, and each occurrence must be evaluated using the slower inexact alignment algorithm. Therefore, many of the new short read mappers restrict the maximum number of differences allowed, or limit the number of alignments reported for each read.

2 ALGORITHM

CloudBurst is a *MapReduce*-based read-mapping algorithm modeled after RMAP, but runs in parallel on multiple machines with *Hadoop*. It is optimized for mapping many short reads

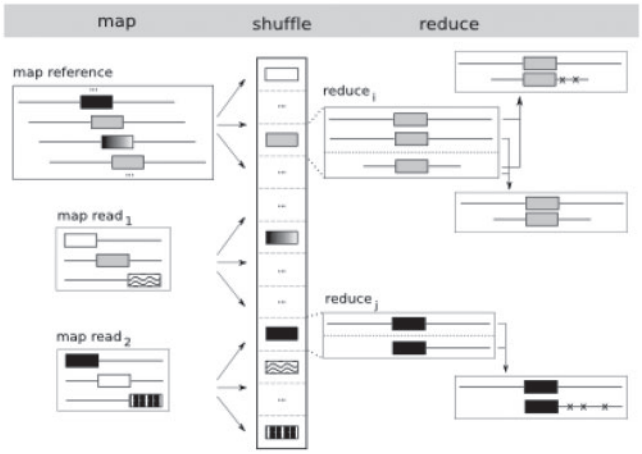

Fig. 2. Overview of the CloudBurst algorithm. The map phase emits k-mers as keys for every k-mer in the reference, and for all non-overlapping k-mers in the reads. The shuffle phase groups together the k-mers shared between the reads and the reference. The reduce phase extends the seeds into end-to-end alignments allowing for a fixed number of mismatches or indels. Here, two grey reference seeds are compared with a single read creating one alignment with two errors and one alignment with zero errors, while the black shared seed is extended to an alignment with three errors.

from next-generation sequencing machines to a reference genome allowing for a user specified number of mismatches or differences. Like RMAP, it is a *seed-and-extend* algorithm that indexes the non-overlapping k-mers in the reads as seeds. The seed size $s = m/(k+1)$ is computed from the minimum length of the reads (m) and the maximum number of differences or mismatches (k). Like RMAP, it attempts to extend the exact seeds to count the number of mismatches in an end-to-end alignment using that seed, and reports alignments with at most k mismatches. Alternatively, like BLAST, it can extend the exact seed matches into end-to-end gapped alignments using a dynamic programming algorithm. For this step, *CloudBurst* uses a variation of the Landau–Vishkin k-difference alignment algorithm (Landau *et al*., 1986), a dynamic programming algorithm for aligning two strings with at most k differences in $O(km)$ time where m is the minimum length of the two strings. See Gusfield's (1997) classical text on sequence alignment for more details.

As a *MapReduce* algorithm, *CloudBurst* is split into *map*, *shuffle* and *reduce* phases (Fig. 2). The *map* function emits k-mers of length s as seeds from the reads and reference sequences. The *shufffle* phase groups together k-mers shared between the read and reference sequences. Finally, the *reduce* function extends the shared seeds into end-to-end alignments allowing both mismatches and indels. 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 binary *Hadoop* SequenceFiles and copied into the HDFS. The DNA sequences are stored as the key–value pairs (*id*, *SeqInfo*), where *SeqInfo* is the tuple (*sequence*, *start_offset*) and *sequence* is the sequence of bases starting at the specified offset. By default, the reference sequences are partitioned into chunks of 65 kb overlapping by 1 kb, but the overlap can be increased to support reads longer than 1 kb.

2.1 Map: extract K-mers

The map function scans the input sequences and emits key–value pairs (*seed*, *MerInfo*) where *seed* is a sequence of length s, and *MerInfo* is the tuple (*id*, *position*, *isRef*, *isRC*, *left_flank*, *right_flank*). If the input sequence is a reference sequence, then a pair is emitted for every k-mer in the sequence, with *isRef* = 1, *isRC* = 0, and position set as the offset of the k-mer in the original sequence. If the given input sequence is a read, then i*sRef* = 0, and a pair is emitted for the non-overlapping k-mers with appropriate *position*. Seeds are also emitted for the non-overlapping k-mers of the reverse complement sequence with *isRC* = 1. The flanking sequences [up to $(m - s + k)$ bp) are included in the fields *left_flank* and *right_flank*. The seeds are represented with a 2 bit/bp encoding to represent the four DNA characters (ACGT), while the flanking sequences are represented with a 4 bit/bp encoding, which also allows for representing an unknown base (N), and a separator character (.).

CloudBurst parallelizes execution by seed, so each reducer evaluates all potential alignments for approximately $1/N$ of the 4^s seeds, where N is the number of reducers. Overall this balances the workload well, and each reducer is assigned approximately the same number of alignments and runs for approximately the same duration. However, low-complexity seeds (defined as seeds composed of a single DNA character) occur a disproportionate number of times in the read and reference datasets, and the reducers assigned these high-frequency seeds require substantially more execution time than the others. Therefore, *CloudBurst* can rebalance low-complexity seeds by emitting redundant copies of each occurrence in the reference and randomly assigning occurrences in the reads to one of the redundant copies. For example, if the redundancy is set to 4, each instance of the seed *AAAA* in the reference will be redundantly emitted as seeds *AAAA*-0, *AAAA*-1, *AAAA*-2 and *AAAA*-3, and each instance of AAAA from the reads will be randomly assigned to seed *AAAA*-R with $0 \leq R \leq 3$. The total number of alignments considered will be the same as if there were no redundant copies, but different subsets of the alignments can be evaluated in parallel in different reducers, and thus improve the overall load balance.

2.2 Shuffle: collect shared seeds

Once all mappers have completed, *Hadoop* shuffles the key–value pairs, and groups all values with the same key into a single list. Since the key is a k-mer from either the read or reference sequences, this has the effect of cataloging seeds that are shared between the reads and the reference.

2.3 Reduce: extend seeds

The reduce function extends the exact alignment seeds into longer inexact alignments. For a given *seed* and *MerInfo* list, it first partitions the *MerInfo* tuples into the set R from the reference and set Q from the reads. Then it attempts to extend each pair of tuples from the Cartesian product $R \times Q$ using either a scan of the flanking bases to count mismatches, or the Landau–Vishkin k-difference algorithm for gapped alignments. The evaluation proceeds block-wise across subsets of R and Q to maximize cache reuse, and using the bases flanking the shared seeds stored in the *MerInfo* tuples. If an end-to-end alignment with at most k mismatches or k differences is found, it is then checked to determine if it is a duplicate alignment. This is necessary because multiple exact seeds may be present within the same alignment. For example, a perfectly matching end-to-end

alignment has $k+1$ exact seeds, and is computed $k+1$ times. If another exact seed with smaller offset exists in the read the alignment is filtered as a duplicate, otherwise the alignment is recorded. The value for k is small, so only a small number of alignments are discarded.

The output from *CloudBurst* is a set of binary files containing every alignment of every read with at most k mismatches or differences. These files can be converted into a standard tab-delimited text file of the alignments using the same format as RMAP or post-processed with the bundled tools.

2.4 Alignment filtration

In some circumstances, only the unambiguous best alignment for each read is required, rather than the full catalog of all alignments. If so, the alignments can be filtered to report the best alignment for each read, meaning the one with the fewest mismatches or differences. If a read has multiple best alignments, then no alignments are reported exactly as implemented in RMAPM. The filtering is implemented as a second *MapReduce* algorithm run immediately after the alignments are complete. The map function reemits the end-to-end alignments as key–value pairs with the read identifier as the key and the alignment information as the value. During the shuffle phase, all alignments for a given read are grouped together. The reduce function scans the list of alignments for each read and records the best alignment if an unambiguous best alignment exists. As an optimization, the reducers in the main alignment algorithm report the top two best alignments for each read. Also, the filtration algorithm uses a *combiner* to filter alignments in memory and reports just the top two best alignments from its subset of alignments for a given read. These optimizations improve performance without changing the results.

3 RESULTS

CloudBurst was evaluated in a variety of configurations for the task of mapping random subsets of 7.06 million publicly available Illumina/Solexa sequencing reads from the 1000 Genomes Project (accession SRR001113) to portions of the human genome (NCBI Build 36) allowing up to four mismatches. All reads were exactly 36 bp long. The test cluster has 12 compute nodes, each with a 32 bit dual core 3.2 GHz Intel Xeon (24 cores total) and 250 GB of local disk space. The compute nodes were running RedHat Linux AS Release 3 Update 4, and *Hadoop* 0.15.3 set to execute two tasks per node (24 simultaneous tasks total). 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, and once loaded the data was reused for multiple analyses.

The first test explored how *CloudBurst* scales as the number of reads increases and as the sensitivity of the alignment increases. In this test, sub-sets of the reads were mapped to the full human genome (2.87 Gbp), chromosome 1 (247.2 Mbp) or chromosome 22 (49.7 Mbp). To improve load balance across the cores, the number of mappers was set to 240, the number of reducers was set to 48, and the redundancy for low-complexity seeds was set to 16. The redundancy setting was used because the low-complexity seeds required substantially more running time than the other seeds (>1 h compared with <1 min), and the redundancy allows their alignments to be processed in parallel in different reducers. Figure 3 shows the running time of these tasks averaged over three runs, and shows

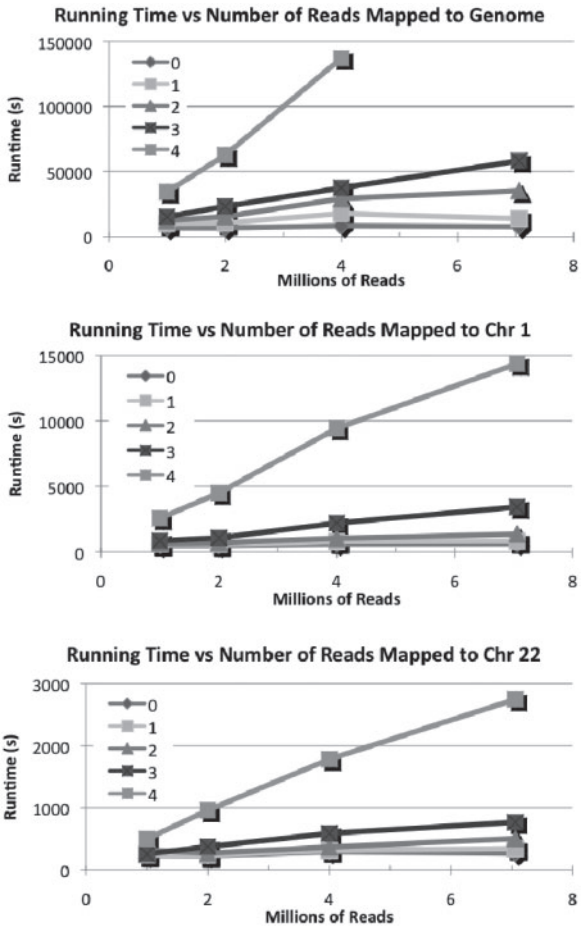

Fig. 3. Evaluation of CloudBurst running time while scaling the number of reads and sensitive for mapping to the (**A**) full human genome; (**B**) chromosomes 1; and (**C**) 22 on the local cluster with 24 cores. Tinted lines indicate timings allowing 0 (fastest) through four (slowest) mismatches between a read and the reference. As the number of reads increases, the running time increases linearly. As the number of allowed mismatches increases, the running time increases superlinearly from the exponential increase in seed instances. The four mismatch computation against the full human genome failed to complete due to lack of available disk space after reporting ~25 billion end-to-end alignments.

that *CloudBurst* scales linearly in execution time as the number of reads increases, as expected. Aligning all 7M reads to the full genome with four mismatches failed to complete after reporting ~25 billion mappings due to lack of available disk space. Even allowing zero mismatches created 771M end-to-end perfect matches from the full 7M read set, but most other tools would report just one match per read. Allowing more mismatches increases the runtime superlinearly, because higher sensitivity requires shorter seeds with more chance occurrences. The expected number of occurrences of a seed length s in a sequence of length L is $(L-s+1)/4^s$, so a random 18 bp sequence ($k=1$) is expected to occur ~0.04, ~0.003 and ~0.001 times in the full genome and chromosomes 1 and 22, respectively, while a 7 bp sequence ($k=4$) is expected to occur >17 500, >15 000 and >3000 times, respectively. Consequently, short seeds have drastically more chance occurrences

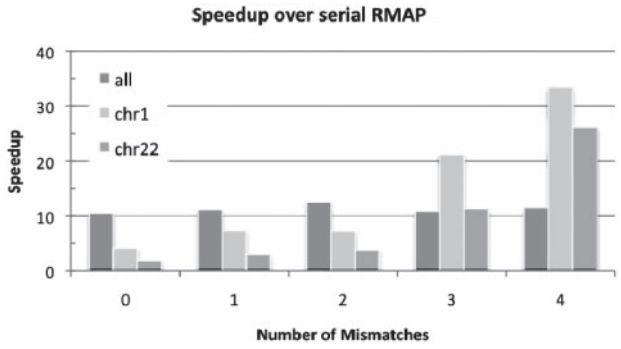

Fig. 4. CloudBurst running time compared with *RMAP* for 7M reads, showing the speedup of CloudBurst running on 24 cores compared with *RMAP* running on 1 core. As the number of allowed mismatches increases, the relative overhead decreases allowing CloudBurst to meet and exceed 24× linear speedup.

and correspondingly more running time even though most chance occurrences will fail to extend into end-to-end matches.

The second test compared the performance *CloudBurst* on 24 processor cores with a serial execution of *RMAPM* (version 0.41) on 1 core with the full read set to chromosomes 1 and 22. *RMAP* requires a 64 bit operating system, so it was run on 1 core of a 64 bit dual core 2.4 GHz AMD Opteron 250 with 8 GB of RAM running RedHat Enterprise Linux AS Release 3 Update 9. *CloudBurst* was configured as before, except with the alignment filtration option enabled so only a single alignment was reported for each read identical to those reported by *RMAPM*. Figure 4 shows the results of the test, and plots the speedup of *CloudBurst* over *RMAP* for the different levels of sensitivity. The expected speedup is 24, since *CloudBurst* runs in parallel on 24 cores, but *CloudBurst*'s speedup over *RMAP* varies between 2× and 33× depending on the level of sensitivity and reference sequence. At low sensitivity (especially $k=0$), the overhead of shuffling and distributing the data over the network overwhelms the parallel computation compared with the in-memory lookup and evaluation in *RMAP*. As the sensitivity increases, the overhead becomes proportionally less until the time spent evaluating alignments in the reduce phase dominates the running time. The speedup beyond 24× for high-sensitivity mapping is due to implementation differences between *RMAP* and *CloudBurst*, and the additional compute resources available in the parallel environment (cache, disk IO, RAM, etc.). The speedup when mapping to the full genome did not improve as the level of sensitivity increased because of the increased overhead from the increased data size. This effect can be minimized by aligning more reads to the genome in a single batch, and thus better amortize the time spent emitting and shuffling all of the k-mers in the genome.

The next experiment compared CloudBurst with an *ad hoc* parallelization scheme for RMAP, in which the reads are split into multiple files, and then RMAP is executed on each file. In the experiment, the full read set was split into 24 files, each containing 294k reads, and each file was separately mapped to chromosome 22. The runtimes were just for executing RMAP, and do not consider any overhead of partitioning the files, remotely launching the program, or monitoring the progress, and thus the expected speedup should be a perfect 24×. However, the runtimes of the different files varied considerably depending on which reads were present, and the corresponding speedup is computed based on the runtime for the longest running file: between 18 and 41 s with a 12× speedup for zero mismatches, 26–67 s with a 14× speedup for one mismatch, 34–98 s with a 16× speedup for two mismatches, 132–290 s with a 21× speedup for three mismatches and 1379–1770 s with a 29× speedup for four mismatches. The superlinear speedup for four mismatches was because the total computation time after splitting the read set was less than the time for the full batch at once, presumably because of better cache performance for RMAP with fewer reads. This experiment shows the *ad hoc* scheme works well with speedups similar to CloudBurst, but fails to reach perfect linear speedup in most cases because it makes no special considerations for load balance. In addition, an *ad hoc* parallelization scheme is more fragile as it would not benefit from the inherent advantages of Hadoop: data-aware scheduling, monitoring and restart and the high-performance file system.

4 AMAZON CLOUD RESULTS

CloudBurst was next evaluated on the Amazon EC2. This environment provides unique opportunities for evaluating *CloudBurst*, because the performance and size of the cluster are configurable. The first test compared two different EC2 virtual machine classes with the local dedicated 24-core *Hadoop* cluster described above. In all three cases, the number of cores available was held constant at 24, and the task was mapping all 7M reads to human chromosome 22 with up to four mismatches, with runtimes averaged over three runs. The first configuration had 24 'Small Instance' slaves running *Hadoop* 0.17.0, priced at $0.10 per hour per instance and provides one virtual core with approximately the performance of a 1.0–1.2 GHz 2007 Xeon processor. The second configuration had 12 'High-CPU Medium Instance' slaves, also running *Hadoop* 0.17.0 and priced at $0.20 per hour per instance, but offers two virtual cores per machine and have been benchmarked to have a total performance approximately five times the small instance type. The running time for the 'High-CPU Medium Instance' class was 1667 s, and was substantially better per dollar than the 'Small Instance' class at 3805 s, and even exceeds the performance of the local dedicated cluster at 1921 s.

The final experiment evaluated *CloudBurst* as the size of the cluster increases for a fixed problem. In this experiment, the number of 'High-CPU Medium Instance' cores varied between 24, 48, 72 and 96 virtual cores for the task of mapping all 7M reads to human chromosome 22. Figure 5 shows the running time with these clusters averaged over three runs. The results show *CloudBurst* scales very well as the number of cores increases: the 96-core cluster was 3.5 times faster than the 24-core cluster and reduced the running time of the serial RMAP execution from >14 h to ~8 min (>100× speedup). The main limiting factor towards reaching perfect speedups in the large clusters was that the load imbalance caused a minority of the reducers running longer than the others. This effect was partially solved by reconfiguring the parallelization settings: the number of reducers was increased to 60 and the redundancy of the low-complexity seeds was increased to 24 for the 48-core evaluation, 144 and 72 for the 72-core evaluation and 196 and 72 for the 96-core evaluation. With these settings, the computation had better balance across the virtual machines and decreased the wall clock time of the execution.

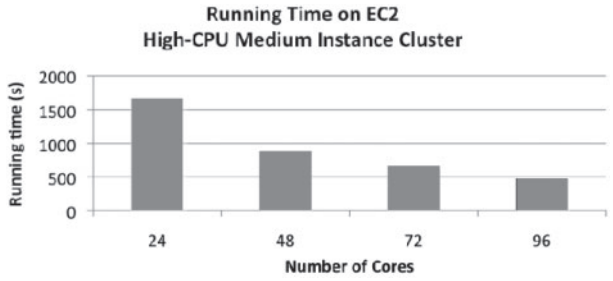

Fig. 5. Comparison of CloudBurst running time (in seconds) while scaling size of the cluster for mapping 7M reads to human chromosome 22 with at most four mismatches on the EC2 Cluster. The 96-core cluster is 3.5× faster than the 24-core cluster.

5 DISCUSSION

CloudBurst is a new parallel read-mapping algorithm optimized for next-generation sequence data. It uses *seed-and-extend* alignment techniques modeled after RMAP to efficiently map reads with any number of mismatches or differences. It uses the *Hadoop* implementation of *MapReduce* to efficiently execute in parallel on multiple compute nodes, thus making it feasible to perform highly sensitive alignments on large read sets. The results described here show *CloudBurst* scales linearly as the number of reads increases, and with near linear parallel speedup as the size of the cluster increases. This high level of performance enables computation of extremely large numbers of highly sensitive alignments in dramatically reduced time, and is complementary to new BWT-based aligners that excel at quickly reporting a small number of alignments per read.

CloudBurst's superior performance is made possible by the efficiency and power of *Hadoop*. This framework makes it straightforward to create highly scalable applications with many aspects of parallel computing automatically provided. *Hadoop*'s ability to deliver high performance, even in the face of extremely large datasets, is a perfect match for many problems in computational biology. *Seed-and-extend* style algorithms, in particular, are a natural fit for *MapReduce*, and any of the hash-table based seed-and-extend alignment algorithms including BLAST, SOAP, MAQ or ZOOM could be implemented with *MapReduce*. Future work for CloudBurst is to incorporate quality values in the mapping and scoring algorithms and to enhance support for paired reads. We are also exploring the possibility of integrating *CloudBurst* into RNA-seq analysis pipeline, which can also model gene splice sites. Algorithms that do not use a hash table, such as the BWT based short-read aligners, can also use *Hadoop* to parallelize execution and the HDFS.

Implementing algorithms to run in parallel with *Hadoop* has many advantages, including scalability, redundancy, automatic monitoring and restart and high-performance distributed file access. In addition, no single machine needs to have the entire index in memory, and the computation requires only a single scan of the reference and query files. Consequently, *Hadoop* based implementations of other algorithms in computational biology might offer similar high levels of performance. These massively parallel applications, running on large compute clouds with thousands of nodes, will drastically change the scale and scope of computational biology, and allow researchers to cheaply perform analyses that are otherwise impossible.

ACKNOWLEDGEMENTS

I would like to thank Jimmy Lin for introducing me to *Hadoop*; Steven Salzberg for reviewing the manuscript; and Arthur Delcher, Cole Trapnell and Ben Langmead for their helpful discussions. 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 *CloudBurst*, and the Amazon Web Services *Hadoop* Testing Program for providing access to the EC2.

Funding: National Institutes of Health (grant R01 LM006845); Department of Homeland Security award NBCH207002.

Conflicts of Interest: none declared.

REFERENCES

Altschul,S.F. *et al*. (1990) Basic local alignment search tool. *J. Mol. Biol.*, **215**, 403–410.

Baeza-yates,R.A. *et al*. (1992) Fast and practical approximate string matching. In *Proceedings of the Combinatorial Pattern Matching, Third Annual Symposium.* Tucson, AZ, pp. 185–192.

Bentley,D.R. *et al*. (2008) Accurate whole human genome sequencing using reversible terminator chemistry. *Nature*, **456**, 53–59.

Ghemawat,S. *et al*. (2003) The Google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. ACM, Bolton Landing, NY, USA, pp. 29–43.

Gusfield,D. (1997) *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, England.

Dean,J. *et al*. (2008) MapReduce: simplified data processing on large clusters. *Commun. ACM*, **51**, 107–113.

Krishnaprasad,S. (2001) Uses and abuses of Amdahl's law. *J. Comput. Small Coll.*, **17**, 288–293.

Kurtz,S. *et al*. (2004) Versatile and open software for comparing large genomes. *Genome Biol.*, **5**, R12.

Landau,G.M. *et al*. (1986) Introducing efficient parallelism into approximate string matching and a new serial algorithm. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*. ACM, Berkeley, CA, USA, pp. 220–230.

Langmead,B. *et al*. (2009) Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biol.*, **10**, R25.

Li,H. *et al*. (2008a) Mapping short DNA sequencing reads and calling variants using mapping quality scores. *Genome Res.*, **18**, 1851–1858.

Li,R. *et al*. (2008b) SOAP: short oligonucleotide alignment program. *Bioinformatics*, **24**, 713–714.

Lin,H. *et al*. (2008) ZOOM! zillions of oligos mapped. *Bioinformatics*, **24**, 2431–2437.

Schatz,M.C. *et al*. (2007) High-throughput sequence alignment using Graphics Processing Units. *BMC Bioinformatics*, **8**, 474.

Shaffer,C. (2007) Next-generation sequencing outpaces expectations. *Nat. Biotechnol.*, **25**, 149.

Smith,T.F. *et al*. (1981) Identification of common molecular subsequences. *J. Mol. Biol.*, **147**, 195–197.

Smith,A.D. *et al*. (2008) Using quality scores and longer reads improves accuracy of Solexa read mapping. *BMC Bioinformatics*, **9**, 128.

Venter,J.C. *et al*. (2001) The sequence of the human genome. *Science*, **291**, 1304–1351.

Wang,J. *et al*. (2008) The diploid genome sequence of an Asian individual. *Nature*, **456**, 60–65.

Wheeler,D.A. *et al*. (2008) The complete genome of an individual by massively parallel DNA sequencing. *Nature*, **452**, 872–876.