2009 International Conference on Parallel Processing Workshops

Biodoop: Bioinformatics on Hadoop

Simone Leo, Federico Santoni, Gianluigi Zanetti
CRS4
Pula, CA, Italy
firstname.lastname@crs4.it

***Abstract*—Bioinformatics applications currently require both processing of huge amounts of data and heavy computation. Fulfilling these requirements calls for simple ways to implement parallel computing. MapReduce is a general-purpose parallelization model that seems particularly well-suited to this task and for which an open source implementation (Hadoop) is available. Here we report on its application to three relevant algorithms: BLAST, GSEA and GRAMMAR. The first is characterized by relatively low-weight computation on large data sets, while the second requires heavy processing of relatively small data sets. The third one can be considered as containing a mixture of these two computational flavors. Our results are encouraging and indicate that the framework could have a wide range of bioinformatics applications while maintaining good computational efficiency, scalability and ease of maintenance.**

I. Introduction

Recent advances in molecular biology and genomics have led to a huge growth of digital biological information. This large amount of data is typically analyzed by repeated use of conceptually parallel algorithms: example contexts include sequence alignment, gene expression analysis, QTL (Quantitative Trait Locus) analysis and haplotype reconstruction. We are currently developing a bioinformatics applications suite, Biodoop, that shows how a general-purpose parallelization technology (MapReduce [1], in its open source implementation, Hadoop [2]) can be easily and successfully tailored to tackle this class of problems with good performance and, more importantly, how this technology can be the basis of a distributed computing platform for several computational biology problems.

MapReduce is an easy-to-use general-purpose parallel programming model tailored for large dataset analysis on a commodity hardware cluster. Developers only need to write two functions: *Map*, which converts an input key/value pair to a set of intermediate key/value pairs, and *Reduce*, which merges together all intermediate values associated with a given intermediate key. The framework automatically handles all low level details such as data partitioning, scheduling, load balancing, machine failure handling and inter-machine communication.

Several bioinformatics applications seem compatible with this paradigm. We chose to consider three qualitatively different algorithms: BLAST [3], GSEA [4] and GRAMMAR [5], [6]. The first is characterized by streaming computation on large data sets, while the second requires a multi-pass computational strategy on relatively small data sets. The third one can be considered as containing a mixture of these two computational flavors.

Part of this work has been previously presented in poster form at *CCA08: Cloud Computing and its Applications*, Chicago, Oct. 22/23, 2008.

II. Implementation

A. Hadoop

Hadoop includes an open source implementation of MapReduce and a distributed file system (HDFS, Hadoop Distributed File System) on which applications can run. Each node in Hadoop can act as a master or slave with respect to both HDFS and MapReduce, leading to four node types: *namenode* (HDFS master), *job tracker* (MapReduce master), *datanode* (HDFS slave) and *task tracker* (MapReduce slave).

One of the key features of Hadoop is its WORM (Write Once, Read Many) storage model, which allows for a very high aggregate bandwidth in data access without the need for complex synchronization mechanisms. Bioinformatics applications, typically characterized by large amounts of data which are not modified during a job run, are well supported by this model.

Instead of the standard Java API, we used the Hadoop streaming library (included in the Hadoop distribution), which allows the user to create and run jobs with any executable as the mapper and/or the reducer. In Hadoop streaming input values (with the default formatter) are lines of text read from one or more files (input keys are discarded).

All results described in this paper, if not otherwise specified, were obtained using Hadoop version 0.17.

B. BLAST

BLAST [3] is the most widely used sequence alignment tool. We developed a Python module [7] for BLAST by wrapping the relevant parts of the NCBI C++ toolkit [8] with the Boost.Python library [9] to build an executable mapper script for Hadoop streaming (no reducer is needed in this case). Sequence datasets have been converted to a one-sequence-per-line format to eliminate the need for a custom input formatter. Mappers read one sequence at a time from standard input, compute its alignment with the given

1530-2016/09 $26.00 © 2009 IEEE
DOI 10.1109/ICPPW.2009.37

IEEE computer society

query sequence(s) and output a customizable tab-separated results line with fields such as query/subject start/end, score etc. Since we started developing our MapReduce version of BLAST as part of an application for automatic annotation of gene expression probes, which only requires relative statistical significance estimates, E-value adjustments for query/database length are currently not performed. However, we plan to include this functionality in the future.

C. GSEA

GSEA [4] is a statistical analysis method for DNA microarray datasets which tests whole gene sets for correlation with phenotype conditions. Correlation is evaluated by means of the Enrichment Score (ES) statistic, a measure of the degree to which a gene set is overrepresented at the top or bottom of a ranked list of genes. P-values are estimated empirically by comparing the statistics vector computed using the actual ("observed") phenotype class labels with the ones obtained with N_p random permutations of the same set of labels, where N_p depends on the desired precision level. Correction for multiple tests is finally performed with False Discovery Rate (FDR) techniques. In its original implementation, GSEA also includes a lot of extra features like individual gene set reports and plots. We focused on the core GSEA algorithm, which computes statistical significance for all gene sets in an input database, using default values for the various statistical options as in [4]. Options for size-based gene set filtering have been included.

We developed a Python library [10] for GSEA and used it to build executable scripts for Hadoop streaming. An initial local data pre-processing step generates N_p random permutations of the class labels vector and dumps them to a file, each indexed by a progressive integer. The observed labels vector is also written to the file as a fake 0-index permutation. Gene sets are then "compiled" with respect to the microarray dataset's genes (text gene labels are replaced with their indices in the dataset's gene list). Limits on maximum and minimum number of genes per set are also applied during this phase, so that only those sets that fall in the desired size range are left in the "compiled" gene set file. These pre-processing steps are very fast and can be executed on the fly on the job tracker hosting machine before launching the application. The algorithm has been implemented as a three-phase MapReduce cascade.

In phase 1, each mapper receives permutations of the observed labels vector as standard input lines. Microarray dataset and gene set database are both read from local file system (Hadoop streaming includes an option for local file distribution). Let N_{gs} be the number of gene sets (after size-based filtering). For each permutation p_j, $j = 0 \dots N_p$ (note that the fake 0-index permutation is included) and gene set GS_i, $i = 1 \dots N_{gs}$, each mapper outputs a (GSN_i, j, ES_{ij}) tuple, where GSN_i is the name of the i-th gene set and ES_{ij} its ES at permutation p_j. According to the Hadoop streaming protocol, all tuples are encoded as tab-separated text lines, where the first field (corresponding to the first tuple element) is by default considered the MapReduce intermediate key. Here and in the following, we will refer to ES_{i0} as the "observed ES", as a shorthand for "ES computed using the observed labels".

In GSEA, ESs are normalized to account for differences in gene set size and in correlations between gene sets and the expression dataset, yielding a Normalized ES (NES). The normalization factor is the mean of all ESs for a given gene set over all permutations, computed separately for positive and negative observed ES:

$$NES_{ij} \equiv \begin{cases} NES^+_{ij} & \text{if} \quad ES_{ij} \geq 0 \\ NES^-_{ij} & \text{if} \quad ES_{ij} < 0 \end{cases}$$

with

$$NES^+_{ij} = \frac{ES_{ij}}{\frac{1}{N_p} \sum_{k=1}^{N_p} ES_{ik}[ES_{ik} \geq 0]}, \tag{1}$$

$$NES^-_{ij} = \frac{ES_{ij}}{\frac{1}{N_p} \sum_{k=1}^{N_p} ES_{ik}[ES_{ik} < 0]}, \tag{2}$$

where [.] denotes the Iverson bracket (this notation will be used throughout the rest of this paper). Since each NES_{ij} depends on all values of ES_{ik}, $k = 1 \dots N_p$, NESs are computed in the reducer together with empirical p-values. The reducer outputs, for each gene set GS_i, $i = 1 \dots N_{gs}$, the following set of tuples:

$$\{(GSN_i, ES_{i0}, NES_{i0}, PV_i, NES_{ij})\}_{j=1 \dots N_p},$$

where NES_{i0} and PV_i are respectively the observed NES and the p-value of the i-th gene set; NES_{ij} is the NES of the i-th gene set at permutation p_j.

Phase 2 and 3 use observed and permuted NES to compute FDR q-values. Two separate MapReduce phases are needed because GSEA uses an FDR algorithm which requires iterations over both permutations and gene sets, while MapReduce works on a single input stream at a time. For each gene set GS_i, the mean fraction (over all permutations) of gene sets whose NES is greater than NES_{i0} is computed for both permuted and observed statistics; the FDR q-value is the ratio between the former and the latter.

To implement this in MapReduce, we had the phase 2 mapper generate a new set of permutation IDs to be used as MapReduce keys (note that this time there is no need of a 0-index permutation because observed statistics have already been computed), so that the framework automatically groups records by permutation. Each mapper, for each gene set GS_i, outputs the following set of tuples:

$$\{(j, GSN_i, ES_{i0}, NES_{i0}, PV_i, NES_{ij})\}_{j=1 \dots N_p}.$$

The framework sorts this stream by j, the permutation ID, and feeds it to the phase 2 reducer which, for each

permutation, computes:

$$r_{ij} \equiv \begin{cases} r_{ij}^{+} & \text{if} \quad NES_{i0} \geq 0 \\ r_{ij}^{-} & \text{if} \quad NES_{i0} < 0 \end{cases}$$

with

$$r_{ij}^{+} = \frac{\sum_{i=1}^{N_{gs}} [NES_{ij} \geq NES_{i0}]}{\sum_{i=1}^{N_{gs}} [NES_{ij} \geq 0]}, \tag{3}$$

$$r_{ij}^{-} = \frac{\sum_{i=1}^{N_{gs}} [NES_{ij} \leq NES_{i0}]}{\sum_{i=1}^{N_{gs}} [NES_{ij} < 0]} \tag{4}$$

and outputs the following set of tuples for each gene set GS_i:

$$\{(GSN_i, ES_{i0}, NES_{i0}, PV_i, r_{ij})\}_{j=1 \ldots N_p}.$$

In phase 3 we use an identity mapper to make the framework sort this record set by GSN_i. The phase 3 reducer reads the sorted stream and computes:

$$\mu_i = \frac{\sum_{j=1}^{N_p} r_{ij}}{N_p} \tag{5}$$

for each gene set. The general formula to get the q-value is:

$$q_i = \min\left(\frac{\mu_i}{\mu_{i0}}, 1\right). \tag{6}$$

Following [4], we estimate μ_{i0} as $\mu_{i0} = r_{i0}$. This value can be easily computed by the phase 2 reducer and passed on as input data to phase 3 (this value is not shown in the above equations for simplicity).

The record key switching in phase 2 and 3 can be seen as a specific instance of a general method for transposing matrices in MapReduce. In this case, data is unrolled along the two directions of gene sets and permutations as needed by each phase's reducer.

D. GRAMMAR

GRAMMAR [5] is a fast method for QTL association analysis. A QTL is a region of DNA associated with a particular phenotypic trait which is quantitative in nature (such as body mass index). Genome-Wide Association (GWA) analysis is a family of statistical methods for studying the complex relations between quantitative traits and the underlying genetic structure, described in terms of high variability DNA regions called *markers*. Currently, the most widely adopted marker type is the Single Nucleotide Polymorphism (SNP), a mutation of a single nucleotide. GRAMMAR's statistic measures the degree of correlation between each SNP in a genotyping dataset and one or more phenotypic traits. P-values are estimated empirically with a method similar to the one described in section II-C.

An extension to GRAMMAR, called GRAMMAR-GC [6], is also available. This new version, implemented as part of the GenABEL R package [11], has increased statistical power and does not require precise knowledge of pedigree structure. GRAMMAR-GC involves three main steps: estimation of trait heritability and environmental residuals from genealogy (PedGR-GC) or genomic data (GenGR-GC); association test; correction of test statistics with Genomic Control (GC) [12]. Despite being faster than previous methods, GRAMMAR-GC is still computationally heavy. Currently available genotyping datasets can easily reach dimensions of millions of SNPs by thousands of individuals. Since analyzing these datasets on a single CPU core with GRAMMAR-GC can take days, a parallel implementation is highly desirable.

We chose to parallelize GenGR-GC because of its higher statistical power [6]. In this version of the method, heritability estimation is done with the kinship matrix K, whose coefficients are in turn estimated from genomic data using the following formula:

$$k_{ij} = \frac{1}{M_{ij}^{*}} \sum_{m=1}^{M} \frac{(g_{im} - p_m)(g_{jm} - p_m)[g_{im} \wedge g_{jm}]}{p_m(1 - p_m)} \tag{7}$$

$$M_{ij}^{*} = \sum_{m=1}^{M} [g_{im} \wedge g_{jm}]$$

where g_{im} and g_{jm} ($i = 2 \ldots N$, $j = 1 \ldots i-1$) are the genotypes of the i-th and j-th individual at the m-th SNP, encoded as 1/2 for heterozygote and 0 or 1 respectively for rare or common allele homozygote, p_m is the frequency of the major allele, M and N are respectively the number of SNPs and the number of individuals. Note that sums extend to SNPs for which both genotypes are non-missing (or "measured").

The strictly lower triangular part of K is given by (7). The diagonal, instead, contains homozygosity indices of individuals (0.5 + inbreeding):

$$k_{ii} = 0.5 + \frac{O_i - E_i}{L_i - E_i}, \; i = 1 \ldots N, \tag{8}$$

where O_i is the observed homozygosity:

$$O_i = \sum_{m=1}^{M} \left[g_{im} \neq \frac{1}{2} \right], \tag{9}$$

E_i the expected homozygosity:

$$E_i = \sum_{m=1}^{M} \left(1 - \frac{2p_m(1 - p_m)T_{Am}}{T_{Am} - 1} \right), \tag{10}$$

L_i and T_{Am} the number of measured genotypes respectively for the i-th individual and for the m-th SNP.

In GenABEL, K is computed with the *ibs* function, implemented in ANSI C and exposed as an extension to the the R package via R's foreign function interface. It should also be noted that GenABEL uses a custom data structure for datasets that packs multiple genotypes in a single byte. Despite all this optimization, though, for current genotyping datasets the computation can take up to several hours.

Our MapReduce implementation of *ibs* allows for scalable parallel computation of the kinship matrix. Inspecting (7) and (8) immediately suggests parallelization on SNPs. Our choice of data formats was dictated by the fact that, as explained later, in the qtscore calculation phase we use GenABEL's functions. GenABEL understands an Illumina/Affymetrix-like genotyping dataset format, a space-separated csv-like text file with one line for each SNP and one column for each individual. Since Hadoop streaming feeds input one line at a time to mappers, this format is directly usable to compute K with parallelization on SNPs. We developed a conversion tool from the popular HapMap format [13] to the one described above.

The *ibs* routine has also been implemented with Hadoop streaming and Python scripts. Each map task starts by initializing buffers for the lower triangular part of K and the three vectors O_i, E_i, L_i used to compute the diagonal. Following GenABEL's implementation, we store the M^*_{ij} coefficients in the upper triangular part of the matrix:

$$k_{ji} = M^*_{ij}; \tag{11}$$

a fifth buffer is allocated to hold this data. After buffer initialization, each mapper starts reading dataset rows from standard input. When a new line is read, genotypes are encoded as described above and all buffers are updated by adding the m-th term of the sum they refer to. After the last input line has been read, the mapper serializes all buffers with Python's standard *pickle* module, encodes them in base64 (Hadoop streaming works on text data) and writes them out as a single tab-separated line. Note that the output stream contains only one line for each map task, allowing us to use a single reducer and thus consistently simplify the code. The reducer splits input lines on tabs, deserializes buffers and adds them up to compute the final sums. O_i, E_i and L_i are then used to compute the diagonal with (8) and lower triangular elements are divided by their symmetrics to finish evaluation of (7). Finally, K is reconstructed from its components and written to standard output.

In GenABEL, the remaining steps in the GenGR-GC analysis are performed by the *polygenic* and the *qtscore* functions. The permutation-based empirical p-values (similar to the ones calculated by GSEA) can be computed by passing an optional parameter to *qtscore* which specifies the desired number of iterations. Since the number of permutations required for this kind of analysis can easily reach the order of the millions, a full qtscore execution can take days on a single CPU core; a single iteration, on the other hand, executes in negligible time. We were therefore able to achieve high-level parallelization by externally reshuffling the environmental residues vector and repeatedly calling single qtscore iterations.

Since in this second step we use GenABEL directly, this is a hybrid Python-R implementation. *qtscore*, which takes the residues vector and the whole dataset as a parameters, must be recomputed for all N_p permutations of the residues vector: to parallelize on permutations, therefore, we could generate an input stream containing one reshuffling of the residues vector per line. Unfortunately, even though a single reshuffling executes in negligible time, running millions of permutations requires local generation of a consistent amount of data, with the result that disk I/O is by itself sufficient to severely hit scalability. We therefore opted for an "inputless" MapReduce implementation: a local preprocessing script computes observed residues and qtscore p-values, saves them to local files and generates a dummy input stream containing only N_p newline characters (in negligible time, for all practical N_p values). When a map task starts, it reads the observed residues and qtscore p-values files from the local file system (these files are distributed by Hadoop streaming) and initializes a buffer for counting the number of qtscore p-values which are smaller than the observed ones. For each (empty) line read, the mapper reshuffles the observed residues vector and recalculates *qtscore*; the resulting p-values vector is compared to the observed one and the buffer is updated accordingly. After reading the last input line, the mapper outputs a tuple containing (serializations of) the buffer and the number of lines read. The reducer (again, we can use a single reducer because it only needs to sum a number of vectors equal to the number of mappers) sums all line counts to compute N_p, sums all buffers and divides the resulting vector by N_p to get empirical p-values, which are finally written to standard output.

III. Results

BLAST and GSEA tests have been conducted on a cluster consisting of 69 nodes, each of whom equipped with two dual-core AMD Opteron 2218 (2600 MHz) CPUs and two Western Digital WD2500JS hard disks (250 GB, 3 Gb/s, 7200 RPM). 23 of these nodes have 16 GB of RAM, while the remaining 46 have 8. GRAMMAR kinship tests took place on a 32 node subset of the same cluster. Finally, GRAMMAR qtscore tests were conducted on a cluster of 24 nodes with two dual-core AMD Opteron 265 (1800 MHz), two 160 GB hard disks and 4 GB RAM each. All nodes are connected through Gb Ethernet.

A. BLAST

BLAST tests have been conducted on the Aug. 27, 2008 version of the "nt" database [14], which consists of 7348665 sequences for a total size of 24 GB. We set HDFS block size to 100 MB, obtaining 245 blocks. Since BLAST results come directly out of the mappers, and their overall size is typically in the order of the kilobytes (allowing for easy single-machine post-processing), we set up a mapper-only job (in terms of Hadoop configuration, this is done by setting the number of reducers to 0). With this setup, we ran the most computationally intensive algorithm in the BLAST family, tblastx, on a query file consisting of 10

sequences with an average length of 814 bases (we used a small query set in order to be able to run multiple tests within reasonable time even with a small number of nodes). Scalability measures have been conducted with cluster sizes in the 5-69 nodes range, with steps of 8 nodes.

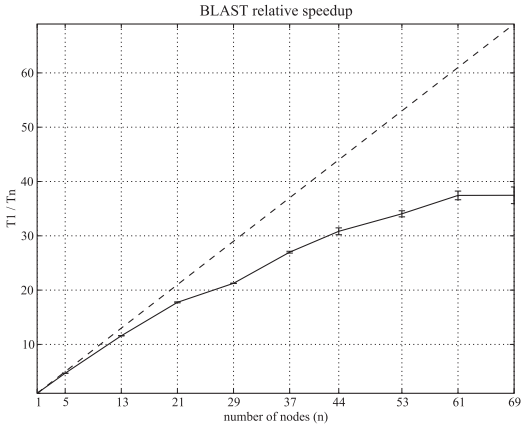


Figure 1. BLAST speedup, average performance on 3 runs. Every node has four CPU cores, for a total of 276 cores with 69 nodes. Here, we ran a 10 sequences, 814 bases average length query against the nt database, using the tblastx program with an e-value threshold of 10^{-50}. We set the HDFS block size to 100 MB.

As shown in fig. 1, scalability seems to enter a "saturation region" after a cluster size of 21 nodes: a thorough analysis of the causes underlying this behavior allowed us to understand in greater depth some of the main factors affecting Hadoop performance. Analysis of machine load variation shows that bad scalability is related to cluster underutilization. Cluster sizes in the "saturation region" are characterized by a load pattern like the one shown on the left side of fig. 2, which refers to one of the three iterations run with 69 nodes: the cluster runs at near-full capacity (almost 4 threads per host) only for a fraction of the job's running time. A lot of CPU cycles are wasted in the beginning and (more consistently) the ending transient. A comparison with the load pattern for one of the 13 nodes run (right side) shows that much less CPU cycles are wasted in the latter case. Since these are map-only jobs, and the maximum number of concurrent map tasks was set to be equal to the number of CPU cores, cluster utilization can be expressed as the ratio between the area under the red curve and the whole rectangle corresponding to the ideal situation in which every host has a running thread on each CPU core from start to end: this yields 69.23% and 92.73% (on average across three runs) respectively for 69 and 13 nodes.

For a given fixed block size, the most important variable governing load balance seems to be the total number of map tasks, configurable through the *mapred.map.tasks* job configuration parameter. It is worth noting that Hadoop does not allow this quantity to go below the total number of HDFS input blocks, with the result that the block size used to upload input data imposes a lower limit on the total number of tasks. Since every MapReduce thread adds a certain amount of startup overhead, it would be desirable to run as few maps as possible. On the other hand, having less map tasks than CPU cores will surely result in cluster underutilization due to idle processors, suggesting that the theoretical optimal number of tasks is somewhere close to the number of CPU cores. To test this behavior, we extracted an 832 MB chunk from the dataset and uploaded it to a 13 nodes cluster, obtaining 26 blocks of 32 MB each (half the number of cores). With this setup we ran the same job used to test blast scalability, with a number of map tasks varying between 26 and 260.

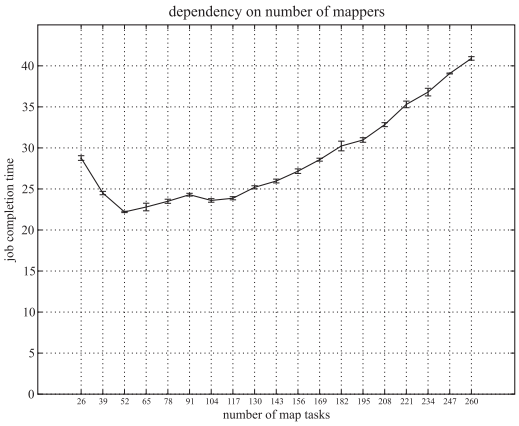


Figure 3. Dependency of job completion time on the total number of mappers. The model dataset has been uploaded with a fixed number of blocks (26), equal to half the number of CPU cores.

As shown in fig. 3, job execution time reached its minimum when the number of map tasks was equal to the number of CPU cores. Of course, larger clusters with random failures and high network traffic should not be expected to behave this regularly. As a rule of thumb, however, datasets should be uploaded to HDFS with a block size that results in a total number of blocks close to the number of available CPU cores. In our scalability test, on the other hand, we set the number of mappers to ten times the number of nodes, (i.e., 2.5 times the number of cores). For cluster sizes up to and including 21 nodes, though, this value falls below the 245 (number of data blocks) mappers threshold imposed by the framework, which explains why scalability stays good in this range.

Even with optimal balancing, though, all load patterns exhibit a more or less pronounced "tail effect", an easily observable decrease in average load towards the end of the job. This is likely due to the fact that different tasks are completed with different speeds and, when all tasks have already been scheduled, some cores are forced to sit idle waiting for the others to finish. This effect is greatly reduced when there are more jobs to run in succession: if all jobs

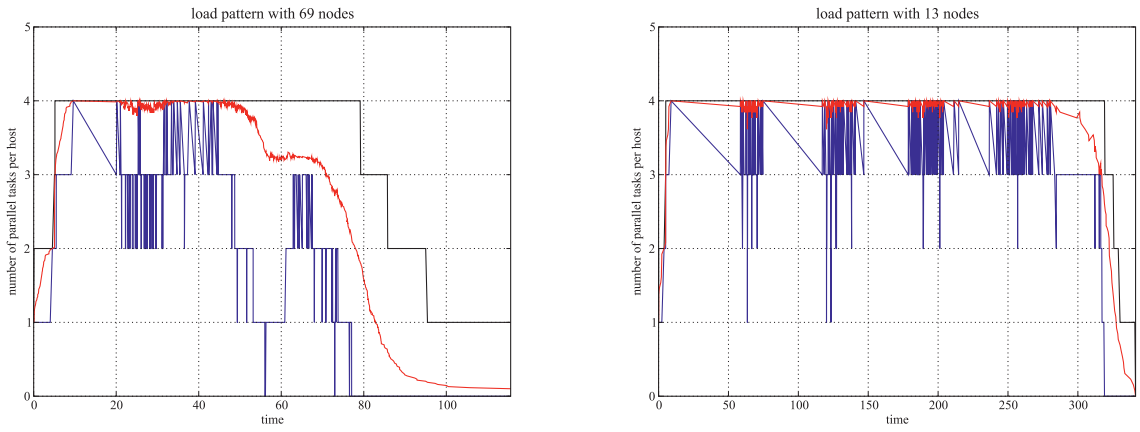


Figure 2. Machine load pattern for a 69 nodes (left) and a 13 nodes (right) run. The blue, red and black lines indicate respectively the minimum, mean and maximum load per node. Note the difference in resource utilization, especially in the ending transient.

Table I
HADOOP JOB QUEUE TIMINGS (IN SECONDS)

job	elapsed	wait	turnaround
1	342.9	1.3	344.2
2	350.9	316.1	667.0
3	351.5	637.7	989.2
4	353.6	959.4	1313.0
5	351.5	1281.2	1632.7
mean	350.1	639.1	989.2
queue	**1633.7 (326.7/job)**		

Table II
GSEA TIMINGS (IN SECONDS)

nodes	elapsed			
	phase 1	phase 2	phase 3	global
5	937.1	154.0	75.2	1166.3
13	382.6	74.8	47.6	505.0
21	246.5	60.1	39.5	346.1
29	189.6	57.9	41.2	288.7

are submitted at the same time, Hadoop places them in a queue and CPU cores start receiving tasks from the next job shortly after the old ones are completed.

Table I shows timings for a queue of five instances of one of the jobs run during the course of the scalability test (in the 13 nodes iteration step) which, when running alone, finished in 342.9 seconds. These timings show that running job queues has different effects for different users: from the job submitter's perspective there is a modest increase in elapsed time and an increase in turnaround proportional to the job's place in the queue; the cluster administrator (or a user which owns all of the queue's jobs), on the other hand, sees a net decrease in average completion time per job and an overall increase in effective throughput.

B. GSEA

We tested GSEA on data available from the authors [15]: the Lung_Boston dataset (5217 genes by 62 samples) and the C2 functional pathways database (522 gene sets with a mean size of 32.66 genes). We imposed the same limits on gene set size used in [4] (15–500), but setting the number of permutations to 10^5 instead of 10^3. It is worth noting that this job cannot even run on a single machine in its original R implementation (the script exits with a memory allocation error).

Table II shows GSEA timings for cluster sizes in the 5-29 range. There is little gain in using more nodes for this kind of job. We tested the full 69 nodes cluster with a much larger dataset from a medulloblastoma study (not yet published) measuring 21464 genes by 28 samples. In this case the job finished in 405.6 seconds.

Even though 10^5 permutations is a rather large value for this class of problems, it should be noted that, if the dataset and/or the gene set database is sufficiently large, even with 10^3 permutations (the standard) the algorithm is sufficiently heavy to justify a parallel implementation. This is confirmed by our results on a dataset derived from a study on metastatic neuroblastoma [16] of 13238 genes by 117 samples, tested using version 2 of the C2 database, which consists of 1687 gene sets with a mean size of 66.97 genes. In this case our implementation, using only four nodes, finished in 05'36 and the R one (modified to perform only basic p-value and FDR analysis) in 45'20.

C. GRAMMAR

We tested GRAMMAR on a genotyping dataset related to androgenetic alopecia [17] (AGA). We used the full dataset (500185 SNPs by 1251 individuals) to test kinship

matrix computation and a subsection (7047 SNPs by 443 individuals) to test qtscore/empirical p-value. The chosen subsection contains only SNPs from chromosome X (the one with the highest correlation with AGA) with a minimum allele frequency cutoff of 0.05 and a maximum ratio of missing genotypes of 0.1. To be able to execute the tests in a reasonable amount of time, we set the number of permutations for computing the empirical p-values to 10^5.

As shown in fig. 4, scalability for both phases is comparable to the one achieved by other applications. In each run, HDFS block size was set to a value that resulted in a number of map tasks equal to the number of CPU cores used in that run.

IV. Related Work

Hadoop seems to be particularly well adapted to bioinformatics, since it provides a consistent and robust environment specifically tailored to deal with large quantities of data and parallel problems where data can be split into loosely coupled blocks of sizes up to roughly the amount of RAM of a node. Within these constraints, the framework shows good computational efficiency, scalability and ease of maintenance. In [18] Y. Sun and others describe a framework that is in the same spirit of the one we developed, but implemented as an ad-hoc grid solution, with a backup task system inspired by MapReduce but without the central feature of moving computation close to where data reside.

Being one of the most widely used bioinformatics tools, and due to its naturally parallel structure (at the sequence database level), BLAST has been the target of many parallel implementations [19]–[24]. However, these implementations are based on a different computing infrastructure model from the one described here and are directly tailored to specific low-level details (message passing libraries such as MPI, or grid frameworks like Globus). They are also BLAST-specific rather than general-purpose. Moreover, their installation and maintenance tend to be complicated if compared to the deployment of a Hadoop cluster. Finally, using a popular, robust and frequently updated parallelization framework like Hadoop allows for piggybacking on its development, reaping the benefits of each new release with little effort on the application developer's side.

The reference implementations of GSEA and GRAMMAR have been written in R (for GSEA there is also a Java version). To the best of our knowledge, the ones described here are the first parallel implementations of these algorithms, and the first ones that make it feasible to run hundreds of thousands of permutations on large datasets.

V. Conclusions and future work

Our preliminary results suggest that Hadoop is a versatile framework that can easily handle both "canonical" problems (relatively light computation on very large datasets) and non-standard ones (heavy processing of small or even "empty" datasets). Due to the framework-driven nature of MapReduce applications, where every task is characterized by a competition between startup overhead and computing time, scalability range tends to be problem size-dependent. Elapsed time eventually reaches a saturation point, theoretically when all of a task's running time is occupied by startup overhead but, in practice, for the various reasons discussed in the above sections, much earlier. Most of these effects, though, are much less relevant in a production environment tailored to handle continuous job flows, where cluster underutilization is negligible. Even though performance is highly dependent on a limited number of parameters, these can be easily adjusted to the problem at hand after a few preliminary tests.

Of course, given the general-purpose nature of the framework, we should not expect Hadoop implementations to be able to compete with custom application-specific solutions, especially for those parts that more deeply deviate from the processing model for which MapReduce has been designed. In the context of bioinformatics, however, many problems seem largely adaptable to this model, allowing even non-specialists to develop parallel implementations with relatively low effort.

We are currently working on applying this computational approach to a broader set of problems and to explore the related configuration tuning issues, including the effects of data and computation distribution strategies on the overall job completion time. From preliminary tests run with Xen [25] we have indications that using virtual machines introduces a very small overhead (less than 5%). We are currently taking advantage of this to quickly set up, migrate and destroy large scale virtual clusters that make use of different MapReduce and/or distributed file system implementations (e.g., CloudStore [26]).

Possible improvements on current applications include better post-processing of BLAST results and lower level parallelization of GRAMMAR's qtscore/empirical p-value computation phase.

References

[1] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," in *OSDI '04: 6th Symposium on Operating Systems Design and Implementation*, 2004.

[2] Hadoop. [Online]. Available: http://hadoop.apache.org

[3] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *J. Mol. Biol.*, vol. 215, no. 3, pp. 403–410, 1990.

[4] A. Subramanian, P. Tamayo, V. K. Mootha, S. Mukherjee, B. L. Ebert, M. A. Gillette, A. Paulovich, S. L. Pomeroy, T. R. Golub, E. S. Lander, and J. P. Mesirov, "Gene set enrichment analysis: a knowledge-based approach for interpreting genome-wide expression profiles," *PNAS*, vol. 102, no. 43, pp. 15 545–15 550, 2005.

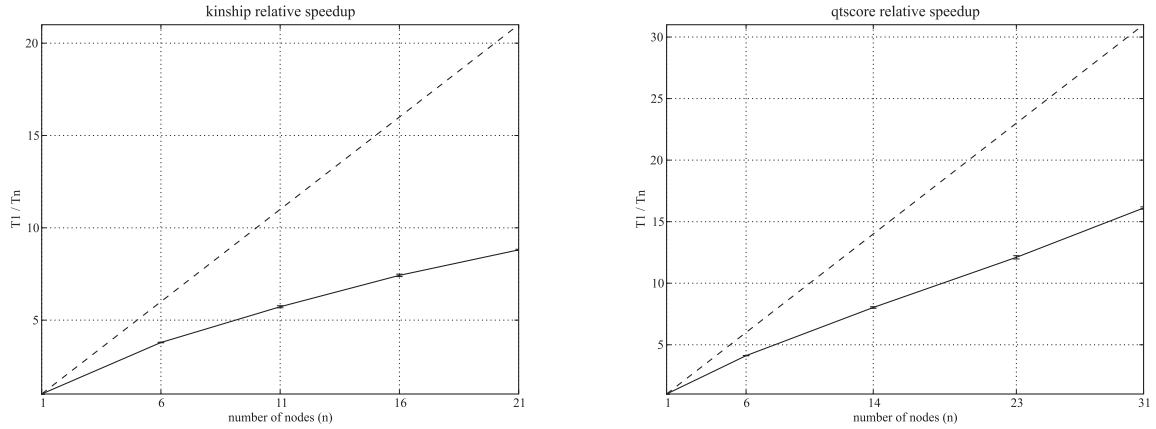


Figure 4. GRAMMAR speedup, average on 3 runs. Every node has four CPU cores. The leftmost image refers to the computation of the kinship matrix for a 500185 SNPs by 1251 individuals dataset, the rightmost one to a qtscore/empirical p-value test run with 10^5 permutations on a 7047 SNPs by 443 individuals dataset. In every run, HDFS block size was set to a value that resulted in a number of map tasks equal to the number of CPU cores.

[5] Y. Aulchenko, D. de Koning, and C. Haley, "Genomewide rapid association using mixed model and regression: a fast and simple method for genomewide pedigree-based quantitative trait loci association analysis," *Genetics*, vol. 177, no. 1, p. 577, 2007.

[6] N. Amin, C. van Duijn, and Y. Aulchenko, "A genomic background based method for association analysis in related individuals," *PLoS ONE*, vol. 2, no. 12, 2007.

[7] S. Leo and G. Zanetti, "BlastPython: python bindings for the NCBI C++ toolkit," unpublished.

[8] NCBI C++ toolkit.

[9] D. Abrahams and R. Grosse-Kunstleve, "Building hybrid systems with Boost.Python," *C/C++ Users Journal*, vol. 21, no. 7, pp. 29–36, 2003.

[10] S. Leo, "pygsa: ython libraries for gene set analysis," unpublished.

[11] Y. Aulchenko, S. Ripke, A. Isaacs, and C. van Duijn, "GenABEL: an R library for genome-wide association analysis," *Bioinformatics*, vol. 23, no. 10, p. 1294, 2007.

[12] B. Devlin and K. Roeder, "Genomic control for association studies," *Biometrics*, vol. 55, no. 4, pp. 997–1004, 1999.

[13] G. A. Thorisson, A. V. Smith, L. Krishnan, and L. D. Stein, "The international HapMap project web site," *Genome Research*, vol. 15, no. 11, pp. 1592–1593, 2005.

[14] BLAST databases. [Online]. Available: ftp://ftp.ncbi.nih.gov/blast/db/FASTA

[15] GSEA: example datasets. [Online]. Available: http://www.broad.mit.edu/gsea/datasets.jsp

[16] S. Asgharzadeh, R. Pique-Regi, R. Sposto, H. Wang, Y. Yang, H. Shimada, K. Matthay, J. Buckley, A. Ortega, and R. C. Seeger, "Prognostic significance of gene expression profiles of metastatic neuroblastomas lacking MYCN gene amplification," *J. Natl. Cancer Inst.*, vol. 98, no. 17, pp. 1193–1203, 2006.

[17] D. Prodi, N. Pirastu, G. Maninchedda, A. Sassu, A. Picciau, M. Palmas, A. Mossa, I. Persico, M. Adamo, A. Angius *et al.*, "EDA2R is associated with androgenetic alopecia," *Journal of Investigative Dermatology*, vol. 128, no. 9, pp. 2268–2270, 2008.

[18] Y. Sun, S. Zhao, H. Yu, G. Gao, and J. Luo, "ABCGrid: application for bioinformatics computing grid," *Bioinformatics*, vol. 23, no. 9, pp. 1175–1177, 2007.

[19] J. Wang and Q. Mu, "Soap-HT-BLAST: high throughput BLAST based on web services," *Bioinformatics*, vol. 19, no. 14, pp. 1863–1864, 2003.

[20] A. E. Darling, L. Carey, and W. chun Feng, "The design, implementation, and evaluation of mpiBLAST," in *Proc. ClusterWorld*, 2003.

[21] A. Krishnan, "GridBLAST: a globus-based high-throughput implementation of BLAST in a grid computing framework," *Concurr. Comp. Pract. E.*, vol. 17, no. 13, pp. 1607–1623, 2005.

[22] S. Dowd, J. Zaragoza, J. Rodriguez, M. Oliver, and P. Payton, "Windows .NET network distributed basic local alignment search toolkit (W.ND-BLAST)," *BMC Bioinformatics*, vol. 6, no. 1, p. 93, 2005.

[23] P. Carvalho, R. Gloria, A. de Miranda, and W. Degrave, "Squid – a simple bioinformatics grid," *BMC Bioinformatics*, vol. 6, no. 1, p. 197, 2005.

[24] C. Oehmen and J. Nieplocha, "ScalaBLAST: a scalable implementation of BLAST for high-performance data-intensive bioinformatics analysis," *IEEE Trans. Parallel Distrib. Syst.*, vol. 17, no. 8, pp. 740–749, 2006.

[25] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 164–177, 2003.

[26] CloudStore file system. [Online]. Available: http://kosmosfs.sourceforge.net