Parallelizing bioinformatics applications with MapReduce

Massimo Gaggero, Simone Leo, Simone Manca, Federico Santoni, Omar Schiaratura, Gianluigi Zanetti
CRS4, Edificio 1, Sardegna Ricerche, Pula, Italy

Abstract—**Current bioinformatics applications require both management 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 technology that appears to be particularly well adapted to this task. Here we report on its application, using its open source implementation Hadoop, to two relevant algorithms: BLAST and GSEA. 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 analysis of these algorithms is relevant to a wide class of complex applications, e.g., structural genomics and genome-wide association studies, since they typically contain a mixture of these two computational flavors. Our results are very promising 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

Over the past few years, advances in the field of molecular biology and genomic technologies have led to an explosive growth of digital biological information. The analysis of this large amount of data is commonly based on the extensive and repeated use of conceptually parallel algorithms, most notably in the context of sequence alignment and gene expression analysis. Here, we will show how a general-purpose parallelization technology (MapReduce[1], in its open source implementation, Hadoop[2]) could be easily and successfully tailored to tackle this class of problems with good performance and scalability, and, more importantly, how this technology could be the basis of a computational parallel platform for several problems in the context of computational biology.

MapReduce[1] is a computational paradigm where the application is divided into many small fragments, each of which may be executed on any node in a cluster of COTS machines. The framework takes care of data partitioning, scheduling, load balancing, machine failure handling and inter-machine communication, while developers need only write their application in terms of a *Map* function, which produces a set of intermediate key/value pairs for each input line, and a *Reduce* function, which merges together these values to form the final output.

Several bioinformatics applications seem compatible with this paradigm. Here we will consider two qualitatively different algorithms, BLAST[3] and GSEA[4]. 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 analysis of these algorithms is relevant to a wide class of complex applications – e.g., structural genomics and genome-wide association studies like EIGENSTRAT[5], PLINK[6], fastPHASE[7] and MACH[8] – since they typically contain a mixture of these two computational flavors.

II. Implementation

A. Hadoop

Hadoop[2] offers an open source implementation of the MapReduce framework and a distributed file system (HDFS, Hadoop Distributed File System) on which the framework can be run. Each node in a Hadoop implementation can act as a master or slave with respect to both HDFS and MapReduce. Hadoop Node types are:

namenode: it is the file system master. It splits files into blocks and distributes them across the cluster with replication for fault tolerance. The namenode holds all metadata information about stored files and acts as an interface that allows groups of data blocks to be seen as ordinary files.

job tracker: it is the MapReduce master. It queries the namenode about data block locations and assigns each task to the task tracker which is closest to where the data to be processed is physically stored.

datanodes: they are the HDFS slaves. They physically store data blocks, serve read/write requests from clients and propagate replication tasks as directed by the namenode. The namenode also acts as a datanode and holds its share of blocks.

task trackers: they are the MapReduce slaves. They execute map and reduce jobs on their assigned data partitions as directed by the job tracker. The job tracker also acts as a task tracker.

One of the key features of this framework is its WORM (Write Once, Read Many) data storage model, which allows for high-level parallelization in data access without the need for complex synchronization mechanisms. This model suits well bioinformatics applications which are typically characterized by large amounts of data that are not modified during a job run.

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

B. BLAST

BLAST[3] (Basic Local Alignment and Search Tool) is the most widely used sequence alignment tool. For a detailed description, see [9].

We used our own Python wrapper[10] for the NCBI C++ Toolkit[11] and Hadoop Streaming to build an executable mapper for BLAST (the reducer is trivial in this case). Although the natural way of developing applications for Hadoop is to write Java classes, the Hadoop Streaming utility (included in the Hadoop distribution) allows the user to create and run jobs with any executable as the mapper and/or the reducer. Sequence datasets have been converted to a one-sequence-per-line format to eliminate the need for an input formatter. The mapper simply reads one sequence at a time from standard input (which, at runtime, is fed by the framework with data read from the input directory), computes its alignment with the given query sequence and outputs a tab-separated results line.

C. GSEA

GSEA (Gene Set Enrichment Analysis) is a DNA microarray data analysis tool which tests whole gene sets for correlation with phenotype conditions. The statistic used to evaluate this correlation is called Enrichment Score (ES), 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 computed empirically by comparing the observed statistics with the ones obtained by N random reshufflings (permutations) of the phenotype class labels. False Discovery Rate (FDR) q-values are then computed to correct for multiple hypothesis tests. For further details, see[4].

Our implementation refers to the core GSEA algorithm, which computes a statistical significance value for all gene sets, without individual gene set reports and plots. We also used default values for the various statistical options, as in [4].

GSEA functions have also been rewritten[12] in Python and used with Hadoop Streaming for the MapReduce version. Before the actual MapReduce application is run, two pre-processing phases must be performed: generate N random permutations of the class labels vector and dump them to a file along with an integer index. The 0-index permutation is the original class label vector, with the actual ("observed") data; "compile" gene sets with respect to the dataset's genes. By this we mean that text gene labels are replaced with indices (integers) into the dataset file'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 meet the limits are left in the compiled gene set file.

These pre-processing phases are very low-weight and can be run locally on the machine that launches the MapReduce application.

The MapReduce version has been implemented in three phases, with the output of each phase acting as the input for the next one.

In phase 1, the mapper reads class label vector permutations from stdin and a (GCT, GMT) file pair (respectively the dataset and the gene set list file), computes enrichment scores for each gene set and outputs a (`gene_set_name`, `permutation_id`, `enrichment_score`) tab-separated stream. The GMT file must be pre-compiled as described above. The reducer reads observed and permuted statistics for each gene set to compute p-values. If N_{gs} is the total number of (compiled) gene sets and N_p is the number of permutations, the reducer outputs, for each gene set GS_i, $i = 1 \ldots N_{gs}$ a set of tab-separated tuples:

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

The tuple elements are respectively the gene set name, the observed enrichment score, the observed normalized enrichment score (NES), the p-value and the NES for permutation j. Enrichment scores are normalized to account for differences in gene set size and in correlations between gene sets and the expression dataset. The normalization factor is the mean of all enrichment scores for a specific gene set across all permutations, computed separately for positive and negative observed enrichment scores:

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

with

$$NES_{ij}^+ = \frac{ES_{ij}}{\frac{1}{N_p}\sum_{k=1}^{N_p} ES_{ik}[ES_{ik} \geq 0]}, \qquad (1)$$

$$NES_{ij}^- = \frac{ES_{ij}}{\frac{1}{N_p}\sum_{k=1}^{N_p} ES_{ik}[ES_{ik} < 0]}. \qquad (2)$$

Where $[S]$ denotes the characteristic function of S:

$$[S] \equiv \begin{cases} 0 & \text{if } S \text{ is false} \\ 1 & \text{if } S \text{ is true} \end{cases}.$$

Phase 2 and 3 use observed and permuted NES to compute the q-values. Two distinct phases are needed because the FDR algorithm used by GSEA requires iterations in both the direction of permutations and that of gene sets, while MapReduce works on a single reduce dimension. For each gene set GS_i, the mean fraction of gene sets with NES greater than NES_{i0} is computed over all permutations. This mean value is computed for both permuted and observed statistics; the FDR q-value is then set to the ratio between the former and the latter.

To implement this algorithm according to the MapReduce paradigm, we had the phase 2 mapper regenerate a permutation ID stream (note that there's no need to have a 0-index permutation because observed statistics have already been computed) to be used as key, so that the framework automatically groups records by permutation. The mapper outputs the following set of tuples for each gene set GS_i:

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

The phase 2 reducer will access this data stream sorted by ID and compute, for each permutation:

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

with

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

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

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

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

Using an identity mapper in phase 3, GSN_i becomes the key, therefore the phase 3 reducer receives the data stream sorted by gene set name and is able to compute:

$$\mu_i = \frac{\sum_{j=1}^{N_p} r_{ij}}{N_p} \qquad (5)$$

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

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

Following [4], we estimate μ_{i0} as $\mu_{i0} = r_{i0}$, that 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 switchings in phase 2 and 3 can be seen as transpositions of the NES matrix, which turn data in the two directions of gene sets and permutations as needed by each phases's reducer.

III. Results

Our test cluster consists 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. All nodes are connected through Gb Ethernet.

A. BLAST

We downloaded the "nt" database in FASTA format[13] from the NCBI ftp site[14]. As of Aug. 27, 2008, the nt database consists of 7348665 sequences for a total size of 24 GB. We set HDFS block size to 100 MB, obtaining 245 blocks: this is less than the number of cores, but we scheduled 10 mappers per node (following directions from [15]) and set replication factor to 5 to get a good load balance. Since BLAST results come directly out of the mapper, and their overall size is typically in the order of kilobytes (allowing for easy single-machine post-processing), we set up a mapper-only job (number of reducers set to 0). With this setup, we ran tblastx (the most computationally intensive algorithm in the BLAST family) for a query file consisting of 10 sequences with an average length of 814 bases, a subset of an input file for an application we are currently developing for automatic annotation of gene expression probes. Typical datasets for this application contain hundreds of sequences, but we limited our sample to 10 in order to be able to run multiple tests within reasonable time even with a small number of nodes. We measured scalability with cluster sizes in the 5-69 nodes range, with steps of 8 nodes.
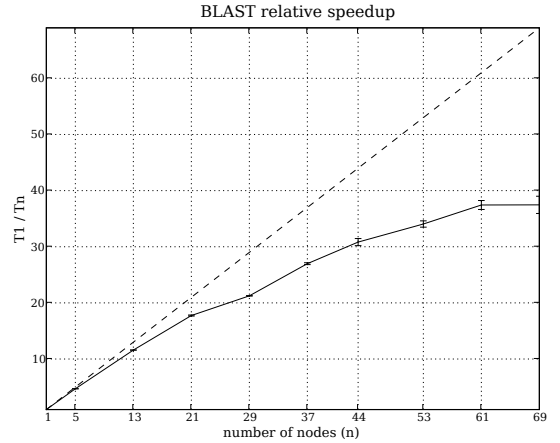

Fig. 1. BLAST relative speedup, average performance on 3 runs. Every node has four CPU cores, for a total of 276 cores with 69 nodes. Here, we ran a 10 sequences, 814 bases average length query against the nt database, using the tblastx program with an e-value threshold of 10^{-50}. We set the HDFS block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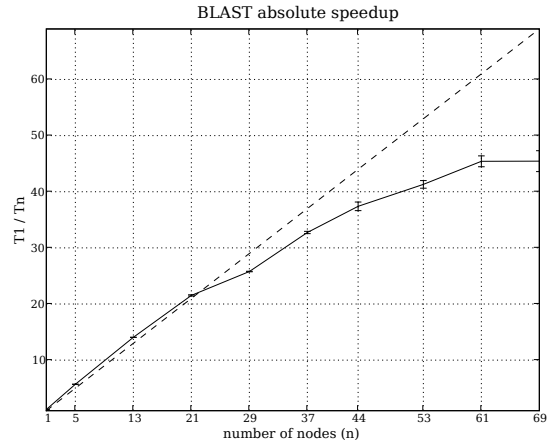

Fig. 2. BLAST absolute speedup, average performance on 3 runs. Every node has four CPU cores, for a total of 276 cores with 69 nodes. Parallel performance data is the same as in relative speedup. To get the reference implementation timing, we scheduled four blastall threads (one per CPU core) on a single machine. Two of the four threads were assigned a 3 query file, while the remaining two got a 2 query file.

As shown in fig. 1 and 2, scalability is good up to a cluster size of 21 nodes (84 cores), then it starts to get worse: a thorough analysis of the causes underlying this behavior allowed us to understand in greater depth the main factors affecting Hadoop performance. Machine load variation analysis shows that bad scalability is correlated to cluster underutilization. Cluster sizes in the bad scalability range (more than 21 nodes) are characterized by a load pattern like the one shown in fig. 3, which refers to one of the three iterations run on the full cluster (69 nodes): the cluster runs at near-full capacity (almost 4 threads per host) for a fraction of the job's running time, while a lot of CPU cycles are wasted in the beginning and, more consistently, the ending transient.

A comparison with the load pattern for a 13 nodes run (see
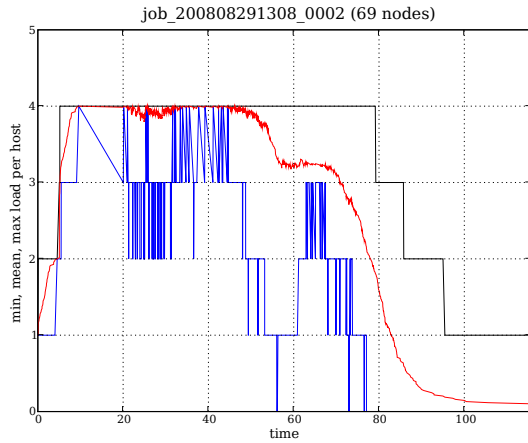
Fig. 3. Machine load pattern for a 69 nodes run. The blue, red and black lines indicate respectively the minimum, mean and maximum load per node.
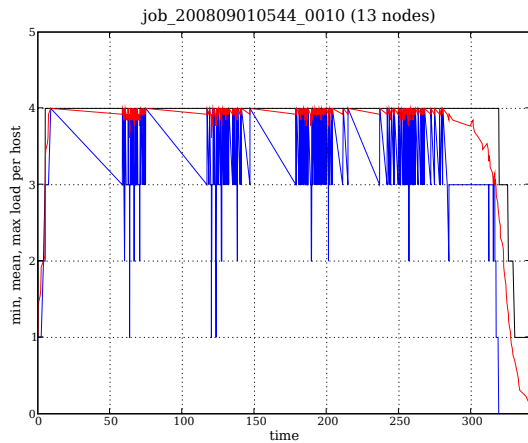

Fig. 4. Machine load pattern for a 13 nodes run. The blue, red and black lines indicate respectively the minimum, mean and maximum load per node.

fig. 4) shows that in the latter case much less CPU cycles are wasted. Since these are map-only jobs, and the maximum number of parallel map tasks was set to be equal to the number of CPU cores, it's easy to measure cluster utilization as the ratio between the area under the red curve and the whole rectangle corresponding to the ideal situation in which all hosts have 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 fixed block size, the most important variable governing load balance seems to be the total number of mappers, which can be set with the *mapred.map.tasks* job configuration parameter. It's worth noting that Hadoop does not allow this quantity to go below the total number of HDFS input blocks, therefore the block size used to upload input data imposes a lower limit on the total number of mappers. Since every MapReduce thread adds some 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. To test this behavior, we extracted an 832 MB chunk of nt 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.
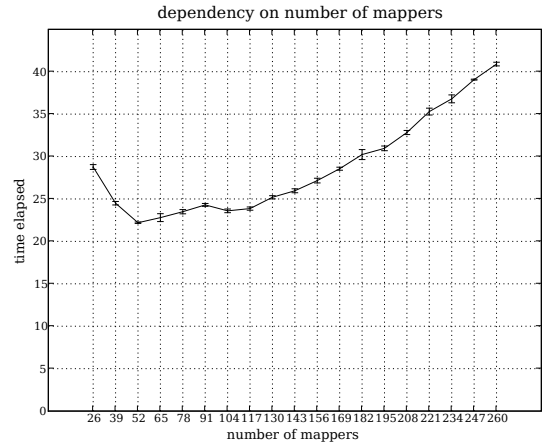

Fig. 5. Dependency of job completion time on the total number of mappers. The model dataset has been uploaded with a fixed number of blocks (26), equal to half the number of CPU cores.

As shown in fig. 5, job execution time reached its minimum when the number of mappers was equal to the number of CPU cores. As a rule of thumb, therefore, 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", a decrease in average load towards the end of the job due to the fact that different tasks are completed with different speeds and, since 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 are submitted at the same time, in fact, Hadoop places them in a queue and CPU cores start receiving tasks from the next job as soon as they are left idle.

Table I shows timings for a queue of five instances of the same job run for the scalability test, with 13 nodes. Since average completion time for this job when running alone was 342.9 seconds, running job queues has a different effect for different user perspectives: each job submitter after the first one sees a modest increase in elapsed time and a consistent increase in turnaround, proportional to the job's place in the queue; cluster administrators, on the other hand, see a net decrease in completion time per job and an overall increase in effective throughput.

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)**		

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

B. GSEA

We tested GSEA on data available from the authors[16]: the Lung_Boston dataset (5217 genes by 62 samples) vs 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] (minimum 15, maximum 500), but setting the number of permutations to 10^5 instead of 1000. 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), which has 21464 genes and 28 samples, obtaining a completion time of 405.6 seconds.

Of course, one could object that 10^5 permutations of the class labels are seldom needed, but even with 1000 permutations there is a consistent gain in parallelizing the algorithm if the dataset and/or the gene set database is sufficiently large. This is confirmed by our results on a dataset derived from a study on metastatic neuroblastoma[17] published on GEO, which measures 13238 genes by 117 samples, 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. Therefore, for computationally intensive jobs, even with a small number of nodes results are better by an order of magnitude.

IV. RELATED WORK

As discussed above, the MapReduce framework seems to be particularly well adapted to bioinformatics, since it provides a consistent and robust environment that can cope very well with large quantities of data and parallel problems where data can be parceled in 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 this paper, 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, BLAST has been the target of many parallel implementations, e.g., Soap-HT-BLAST[19], mpiBLAST[20], GridBLAST[21], W.ND BLAST[22], Squid[23], ScalaBlast[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 and 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 widespread parallelization strategy like Hadoop allows for piggybacking on its development, reaping the benefits of each new release with no effort.

The reference implementation of GSEA is in R/Java. As far as we know, this is the first parallel implementation and the first one that makes it feasible to run hundred of thousands of permutations on datasets of more than 10000 genes by 100 patients and gene sets databases with thousands of elements.

V. CONCLUSIONS

Our preliminary results suggest that MapReduce is a versatile framework that can easily handle both "canonical" problems (relatively light computations on very large datasets) and non-standard ones (heavy processing of small datasets). Due to the framework-driven nature of MapReduce applications, where every task is characterized by a competition between startup overhead and actual computing time, scalability range tends to be problem size-dependent. Elapsed time will eventually reach 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 with a few preliminary tests. Since many other bioinformatics application can be expressed in terms of the MapReduce paradigm, Hadoop constitutes a simple, easily deployable, open source solution that can easily harness the power of COTS computational clusters.

We are currently working on applying this computational approach to a broader set of problems, e.g., structural genomics and genome-wide association studies, 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., kosmos[26]).

REFERENCES

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified DataProcessing on Large Clusters," in *OSDI 2004: Sixth Symposium on Operating System Design and Implementation*, 2004. [Online]. Available: http://labs.google.com/papers/mapreduce-osdi04.pdf

[2] "http://hadoop.apache.org/core."

[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. [Online]. Available: http://www.pnas.org/cgi/content/abstract/102/43/15545

[5] A. L. Price, N. J. Patterson, R. M. Plenge, M. E. Weinblatt, N. A. Shadick, and D. Reich, "Principal components analysis corrects for stratification in genome-wide association studies," *Nat Genet*, vol. 38, no. 8, 2006.

[6] S. Purcell, B. Neale, K. Todd-Brown, L. Thomas, M. A. R. Ferreira, D. Bender, J. Maller, P. I. Sklar, P. I. W. de Bakker, M. J. Daly, and P. C. Sham, "Plink: A tool set for whole-genome association and population-based linkage analyses," *Am J Hum Genet*, vol. 81, no. 3, 2007.

[7] P. Scheet and M. Stephens, "A fast and flexible statistical model for large-scale population genotype data: Applications to inferring missing genotypes and haplotypic phase," *Am J Hum Genet*, vol. 78, no. 4, 2006.

[8] Y. Li and G. R. Abecasis, "Mach 1.0: rapid haplotype reconstruction and missing genotype inference," *Am J Hum Genet*, vol. 79, p. 2290, 2006.

[9] I. Korf, M. Yandell, and J. Bedell, *BLAST*. O'Reilly Media, Inc., 2003.

[10] S. Leo and G. Zanetti, "Blastpython: Python bindings for the ncbi c++ toolkit," unpublished.

[11] "http://www.ncbi.nlm.nih.gov/books/bv.fcgi?rid=toolkit.toc&depth=2."

[12] S. Leo, "pygsa: Python libraries for gene set analysis," unpublished.

[13] "http://www.ncbi.nlm.nih.gov/blast/fasta.shtml."

[14] "ftp://ftp.ncbi.nih.gov/blast/db/fasta."

[15] "http://wiki.apache.org/hadoop/howmanymapsandreduces."

[16] "http://www.broad.mit.edu/gsea/downloads.jsp."

[17] "http://www.ncbi.nlm.nih.gov/projects/geo/query/acc.cgi?acc=gse3446."

[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. [Online]. Available: http://bioinformatics.oxfordjournals.org/cgi/content/abstract/23/9/1175

[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. [Online]. Available: http://bioinformatics.oxfordjournals.org/cgi/content/abstract/19/14/1863

[20] F. Schmuck and R. Haskin, "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: Research articles," *Concurr. Comput. : Pract. Exper.*, vol. 17, no. 13, pp. 1607–1623, 2005.

[22] S. E. Dowd, J. Zaragoza, J. R. Rodriguez, M. J. Oliver, and P. R. Payton, "Windows .net network distributed basic local alignment search toolkit (w.nd-blast)," *BMC Bioinformatics*, vol. 6, no. 93, 2005.

[23] P. C. Carvalho, R. V. Gloria, A. B. de Miranda, and W. M. Degrave, "Squid - a simple bioinformatics grid," *BMC Bioinformatics*, vol. 6, no. 197, 2005.

[24] C. Oehmen and J. Nieplocha, "Scalablast: A scalable implementation of blast for high-performance data-intensive bioinformatics analysis," *IEEE Transactions on Parallel and Distributed Systems*, 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," in *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM, 2003, pp. 164–177.

[26] "http://highscalability.com/kosmos-file-system-kfs-new-high-end-google-file-system-option."