

This is the **accepted version** of the journal article:

Espinosa, Antonio; Hernández Budé, Porfidio; Moure, Juan C.; [et al.]. «Analysis and improvement of map-reduce data distribution in read mapping applications». The Journal of Supercomputing, Vol. 62 (2012), p. 1305–1317. DOI 10.1007/s11227-012-0792-8

This version is available at <https://ddd.uab.cat/record/288057>

under the terms of the  **CC BY-NC-ND** license

Analysis and optimization of map reduce data distribution in read mapping applications

A. Espinosa, P. Hernandez, J.C. Moure, J. Protasio, A. Ripoll

Abstract: Map-reduce paradigm has shown to be a simple and feasible way of filtering and analyzing large datasets in cloud and cluster systems. Algorithms designed for the paradigm must implement regular data distribution patterns so that appropriate use of resources is assured. Good scalability and performance on Map-Reduce applications greatly depends on the design of regular intermediate data generation-consumption patterns at the map and reduce phases. We describe the data distribution patterns found in current Map-Reduce read mapping bioinformatics applications and show some data decomposition principles to greatly improve their scalability and performance

Keywords: *Bioinformatics, Read Mapping, Map Reduce, Scalability*

1. Introduction

Map-reduce [1] distributed computing platforms [2] can provide a scalable way of storing large amounts of data and routinely launch data processing distributed jobs. Map-reduce platforms have provided a successful paradigm for data oriented cloud computing [3] with two important contributions:

- Simplified data oriented programming model where programmers just have to concentrate on two types of operations: map and reduce.
- Data distribution and process communication is automatically managed by the framework, making all distributed tasks work on their local partition of the input data.

Efforts to simplify programming of distributed systems usually hide a complex optimization process to make the best possible use of the system. Frameworks provide a long list of parameters to regulate computing system resources dedicated to application and framework requirements for its operation during runtime.

The task of application optimization needs a comprehensive view of resources used by both user and system tasks that cooperate during the execution of the application. The main sources of performance optimization of a map-reduce application are the following:

- Map system resource management
 - o Intermediate key-value storage

A. Espinosa, P. Hernandez, J. C. Moure, J. Protasio, A. Ripoll

Computer Architecture and Operating Systems Department. Universitat Autònoma de Barcelona. 08193, Bellaterra, Spain.

e-mail: AntonioMiguel.espinosa@uab.es

P. Hernandez

e-mail: Porfidio.Hernandez@uab.es

J. C. Moure:

e-mail: JuanCarlos.Moure@uab.es

J. Protasio:

e-mail: jprotasio@caos.uab.es

A. Ripoll

e-mail: Ana.Ripoll@uab.es

- Sort and merge of output values
 - Communication with reducers
- Data load balance
 - Input data distribution and access
- Reduce filtering of map output data

Some of these sources of optimization depend on the application design and the decisions made by programmers when implementing map and reduce. But others depend on the tuning of the system to make available the actual resources provided by the computers where it runs on. For example, input data access depends on data load balance of the distributed file system where data is stored. At the same time, the number of input data partitions to process in parallel depends on the number of map tasks that the user defines for the application.

Bioinformatics read mapping applications when ported to map reduce frameworks must define a data decomposition policy to specify what computation is going to be done in parallel by each task. Straightforward designs are found in recent map-reduce bioinformatics applications such as Cloudburst [4], Crossbow [5] and MrsRF [6]. In those cases, decomposition is done “sequence centric” where all map tasks receive chunks of input sequences and just generate some key-value pairs for each of them. All these pairs will be joined at the reduce phase automatically. Alternatively, a “genome centric” strategy can be applied, where each map task is responsible of a genome region where all provided reads must match. The application must compare in any case every read with all the reference genome.

Both application designs provide correct results but show very different performance. Sequence read centric designs show a scatter behavior, where each read is sent to be compared with the whole set of the sequences in the problem. This is not desirable in terms of memory and communications use because all possible sequence combinations must be stored temporary to allow all-to-all comparisons. When data to analyze becomes very large, application becomes slow and resource dependent.

On the other hand, genome-centric arrangements show gather access behavior, where each map task collects all input reads and processes their mapping into a genome region. That is, it gathers all input data to process the input sequence set locally. In this way, only useful or high quality mappings are to be reduced, saving system memory and bandwidth.

We have developed an implementation of bioinformatics MAQ read mapping application to show the effect of scatter-gather problem decomposition in the application performance. First, we introduce a simple map-reduce design to implement a scatter all-to-all design to compare two genomic sequence databases. Then, we show some limitations of the design and propose a gather transformation of the original design to improve the performance of the application. The use of Hadoop Map-Reduce platform allows the pattern conversion without modifying the programming view of the original design. Finally, we provide the execution times of the application with sequence datasets

of different sizes to provide a scalability measure of the application after the optimizations are applied.

2. Map reduce application structure

Map-Reduce was initially defined as a framework of distributed computing by Google [1]. Based on the principles presented, many implementations of it have followed. Most popular is Hadoop[2] which is an Apache project version programmed in Java. Other interesting implementations are Phoenix [7] and Metis [8] for shared memory systems.

Map-reduce programming is becoming an alternative to traditional High Performance Computing and Cloud platforms when considering large amounts of data to process. It is a data oriented SPMD (Simple Program Multiple Data) paradigm where all processes perform the same data transformations on a partition of a usually large input data set.

Application operation, shown in fig. 1, is divided in two stages: the map and the reduce phase. The map phase is usually simple as it receives a split of the input data that must be converted into domain meaningful key-value pairs. Keys store information to be used as base for comparisons and values usually keep auxiliary information. The reduce phase receives as input a set of sorted keys with all values generated for them in the map phase. Then, proceeds to aggregate all data for each key received.

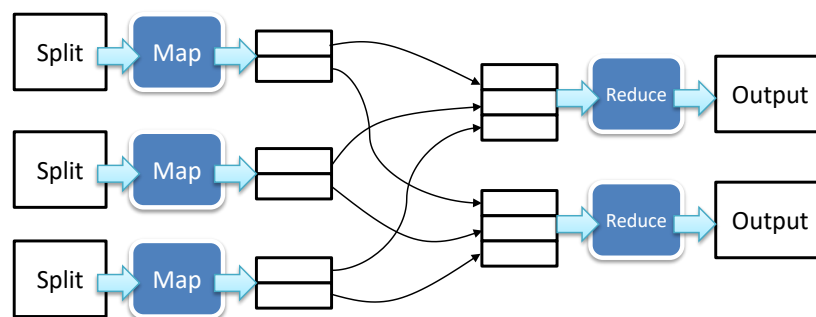


Fig 1: Basic structure of a map-reduce simple application. Map outputs are sorted by key, merged and then sent to reducers to be processed.

For example, when using map-reduce to count the number of times a word has appeared in a text, map tasks take each text word as a key while values store a simple "1" value expressing that the word has appeared once in the text.

Data distribution of map outputs is managed automatically in an intermediate phase where key-value pairs are sorted by key. All key-value pairs that fall into a determinate key interval are sent to a specific reducer. In this way, each reducer receives its own sorted list of key-values and, for each key, all the values generated at any map process.

Reducers then start receiving a list of sorted keys where, for each key, we have all values sent by all maps in the previous phase. In our example of counting words, reducers receive a sorted list of words where, for each word, we have a list of "1" values. Those

values can simply be added at the reduce phase to generate the number of times each word has been found in the text.

Regarding performance of Map-Reduce applications, we are going to evaluate the scatter-gather data partitioning design alternatives in terms of their usage of Hadoop computation and memory resources in specific stages of map reduce execution:

- Input data partitioning: all maps should receive an equivalent data split amount of input data. When we find maps that need to receive data blocks from other nodes, map execution tasks times will be affected.
- Intermediate-key management: when maps generate a large amount of key-values, eventually they will be spilled to disk. Spilling is a system process that can seriously affect the execution time of the map tasks.
- Sort and merge of map outputs: after maps have finished, all keys and values must be partitioned into a number of subsets. A system data management task sorts each subset by key and sends it to a specific reducer. When map output data is large and has been spilled to many files, local disk sort and merge operations could be costly.
- Communication with reducers: reduce operations must wait for all map output data to arrive. A large set of map outputs slow global computation by overloading communications and by letting reducers wait for the data.

3. MAQ short read alignment

Original MAQ algorithm [9] is designed to align short read DNA sequences to a reference genome. To do that, it builds a hash table to store and index the short read input set. MAQ alignment algorithm is based on the observation that a full length alignment of an n bp long read with at most k differences must contain at least one exact alignment of $n/(k+1)$ consecutive bases [10]. That is, for a 30 bp read to be mapped to a reference sequence with only one mismatch, there must be at least 15 consecutive bases that match exactly independently from the actual position of the mismatch.

MAQ builds six different hash tables to index the first 28 bp of the reads to ensure that alignments with up to two mismatches are hit. These six tables correspond to six non-contiguous templates like 11110000, 00001111, 11000011, 00111100, 11001100 and 00110011 if reads were 8 characters long.

After the read hashes are built, the reference is scanned on forward and reverse strands; each reference sequence scanned is looked up at the hashes. If a hit is found to be a read, MAQ calculates a quality score for the match as the probability that the alignment is wrong. For this, it uses a quality score (Qs) formula:

$$Qs = \min \left\{ \begin{array}{l} q_2 - q_1 - 4.343 \log n_2, \\ 4 + (3 - k') (q - 14) - 4.343 \log p_1(3 - k', 28) \end{array} \right\}$$

Where q_1 is the sum of quality values of mismatches of the best hit, q_2 is the corresponding sum for the second best hit, n_2 is the number of hits with the same number of mismatches as the best hit, k' is the minimum number of mismatches in the 28bp seed, and q is the average value of base quality in the 28bp seed. Other examples of similar spaced seed indexing algorithms like MAQ are SOAP [11] and RMAP [12].

4. Map-reduce MAQ implementation

MrMAQ is a map-reduce application programmed in Java using Hadoop open source map-reduce implementation. The basic principles of the application are the same as MAQ. That is, to find matches of a list of short reads with a provided genome reference by using a seed and extend algorithm.

Bioinformatics applications mentioned share a similar way of partitioning the problem in non overlapping tasks. As a simple step from the serial algorithm, both genome reference and read files are split to be processed in parallel. Then, map tasks must generate sequence reads with a pre-defined length from both inputs:

- 1) Map tasks must process short read files and genome reference files. Each task receives a piece of the input sequence sets and then scatters the result of applying original MAQ templates to each read. In short, for each data sequence provided, it must generate N key-value pairs. These pairs must be later compared all-to-all so that reads that match references are used to find alignments.

In fig. 2, we represent the data pattern shown by maps in this scatter implementation.

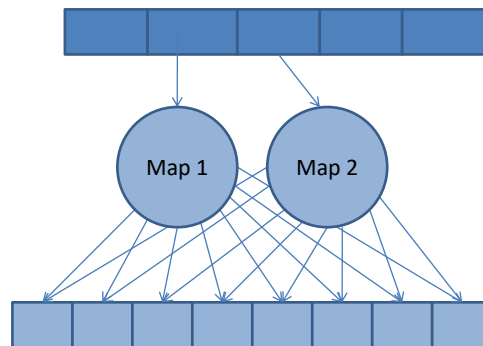


Fig. 2: Scatter map task design. Reads inputs are processed by applying templates and generating several seed keys that have to be matched later on.

- 2) Reduce phase: the alignment extension. For every key that a reduce task receives, it also receives a set of all reads and references that have the same seed. For each of these matches, the reduce task will extend the alignment to the rest of the read and calculate the sum of qualities of mismatched bases q over the whole length of the read, extending out from the seed without gaps. Fig. 3 shows the process of extending the alignment of the hit found. Then it outputs the best quality alignments found for every reference position

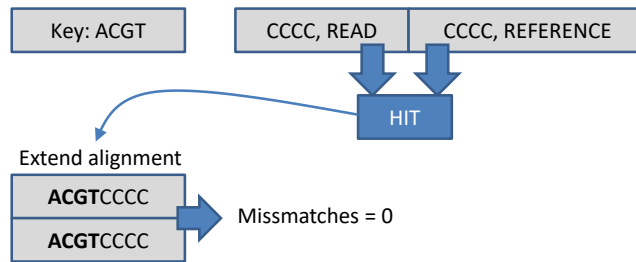


Fig 3: reduce task operation. For each hit, whole sequences are aligned to find mismatch number and quality values of alignment

5. Performance implications of scatter-distributed applications in map reduce

In the case of MAQ and many other sequence alignment map-reduce programs, application follows a simple map design where keys are sequence seeds from two input sets that have to be compared all-to-all. Implications of the scatter application pattern are not always visible to programmers, as main impact of the design affects the performance of system management of data.

This is a simple map to design, but requires that all messages to be generated, sorted, merged and sent to all reducers before doing any operation on them. The main problem of scatter application pattern is that potentially all tasks need to address the same sequence. Therefore, it may reproduce an execution bottleneck when having to reduce all potentially matching sequences.

In summary, threats to scatter map reduce application designs fall into the following basic problems:

- Intensive use of local disk write operations for storing map intermediate data output: sequence seeds generated at map tasks
- Overload of communications when sending all single seed sequences generated to reducers
- Costly overall computation when large number of sequences have to be processed as all seeds have to be stored and sent before being compared and, eventually, discarded.

Our objective is to improve the design of the application to convert it to a more scalable design. In this case, we want to transform map tasks to perform gather operations instead of a distribution of the data they receive.

6. Optimization of Map-Reduce applications: building a gather pattern from a scatter application

With the large amount of single sequences to be processed in read mapping applications, it is desirable to look for a scalable design that filters all sequences that are not useful to

find quality alignments. In this sense, we want to redesign our map tasks as gatherers of single sequence seeds from the input. As shown in fig. 4, instead of generating a large amount of seeds to be matched, we want map tasks to compare their inputs with the genome, so that only those sequence seeds that actually hit the genome with a minimum quality will be processed to the reduce phase. In summary, we will transform the behavior of map tasks from generators to filters of sequences.

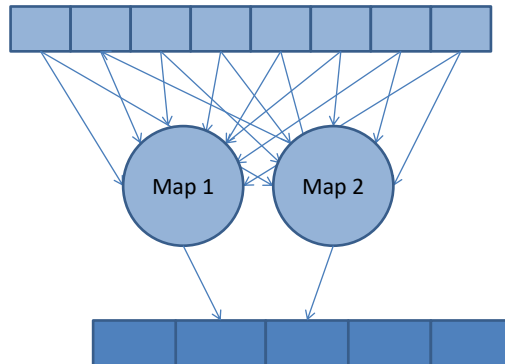


Fig. 4: gather map task design for read mapping.

The most important objective of this design in terms of performance is the exploitation of data locality at map tasks. That is, make use of data stored at map tasks to filter data sequence seeds. We should decrease the number of messages and data sent to reducers. Then, improve the communication and data reduce requirements of the application.

The objective of the map tasks design will be send only relevant data to remote reducers. That is, for all input sequence reads, map tasks will only send high quality hits found with the genome locally available for the map task. Then, all map tasks gather large amount of input sequences only to send a smaller amount of high quality matches to reducers.

Map tasks can filter input data by careful design of combiner functions. Combiner functions work as a “map side reducers” that usually aggregate locally repeated key-value pairs before sending them to reducers. For example, in the word count example, a simple combiner function would send only a key-value pair for each different word found, so that no repeated key-value pairs travel out of a map task. The efficient use of combiners is relevant for reducing data traffic and execution time of large dataset applications

Our combiner implementation of gather pattern is designed following these principles:

- All map tasks have an identical local copy of the input reference file (human genome) to process with the use of Hadoop distributed cache.
- Each map task receives its own piece of the input sequence database.
- Map task generates all seeds from read and reference files as in the straightforward map designs.

- A gather combiner function is called at the end of map tasks to only keep local seed hits: those reference seeds that have matched a read.

This way, each map task is responsible of finding read hits into its own piece of genome reference and sending them to reducers. In fig. 4 we show the process of each map task when combiner is applied. Templates have been applied to input sequence reads, and the genome reference has been scanned. Combiner sends to reducers only those key-value pairs where it has found a reference and a read with the same key. The rest are discarded. Once all hits are received, reducers proceed to calculate potential alignment quality scores.

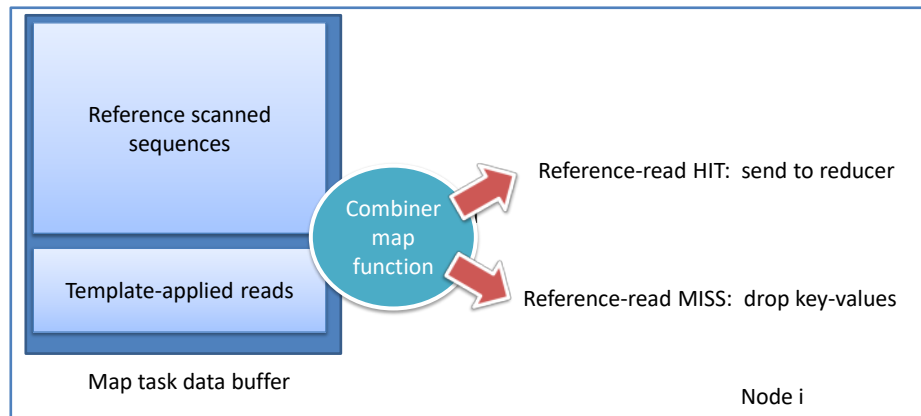


Fig. 4: map-side filtering of reads, reference seeds that do not match any read are not sent.

Benefits of this new design are the reduced memory needs of the application as less data is needed at each stage of the execution. Specially sort, merge and data transmission of map data to reduce tasks.

These map gather design principles can be applied to any map-reduce implementation of sequence alignment problems, as a general technique to reduce input/output volume and improve both execution times and scalability.

Notice that original map and reduce designs must not be changed. All optimizations are provided to the application in the combiner function and a specific way of placing the data at each map. This way, similar techniques could be applied to any map-reduce algorithm without modifying existing map-reduce definitions.

7. Hadoop optimizations to increment the gathering capacity of map tasks

Besides redesigning the map phase of the application to implement a gather pattern, it is important to measure the actual volume of data generated in the map phase and how it fits into the resources of the system that executes the program. There is a group of Hadoop parameters that can be tuned to configure the resource usage of the execution. Once we know the application needs we can modify these parameters to maximize resource usage and set them as new defaults.

In our case, we want our mrMAQ application to run the map-side filtering as fast as possible. We have studied some relevant Hadoop configuration parameters that control the resources available for map tasks looking for optimized values that we consider relevant for minimizing execution time [13, 14].

Our optimization principles are to avoid the access to disk when possible and to allow the local usage of application data a the map tasks.

7.1. Storing reference genome locally

To allow map combiner functions to search genome reference sequences, we use Hadoop distributed-cache feature to automatically distribute reference genome files to all map tasks at the beginning of the execution. That is essentially a distributed file copy to local map task disk space that takes a few seconds.

We also have to increment the amount of Java Virtual Machine Heap memory so that both Hadoop framework buffers and the Map-Reduce application structures can handle up to four Gigabytes of local memory.

7.2. Allocating memory for sequence hits

Our optimization strategy modifies the default behavior of the system management of map tasks output data. While maps run, Hadoop must allocate all intermediate key-values into map tasks local memory buffer. When this buffer is filled up to a threshold limit all generated pairs are spilled to disk. In this way, when map output data is large, we will have several spilled files that need to be sorted, merged and sent. Default size of this buffer is 100MB. Considering the amount of memory available at each node of our cluster, we made buffer size big enough so as not to spill any data during all map execution. Needless to say, allocated buffer sizes must fit into Java Heap allocation defined previously.

Hadoop structure for a map task intermediate value buffer is defined by *io.sort.mb* parameter. Our objective is to let *io.sort.mb* to be large enough to store all local intermediate values generated by map. Parameters of buffer configuration are shown in fig. 5.

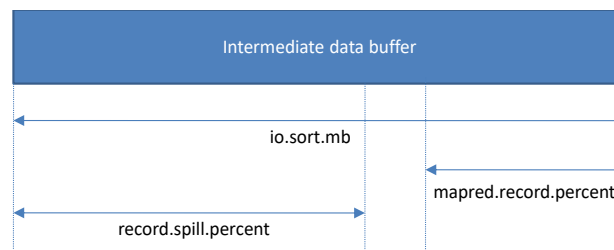


Fig. 5: buffer for map output intermediate values and its configuration parameters

Together with the size definition of intermediate map buffers, we must also configure a few related parameters

- `io.sort.record.percent`: percentage of buffer dedicated to store record boundaries. Set to 30% to allow a larger number of records
- `io.sort.spill.percent`: limit of buffer occupation to start spill process. Set to 99% to avoid spilling to disk

With this configuration, map tasks store all read and reference values in memory while buffer limit is not reached. Then, Hadoop automatically calls the combiner function to filter all redundant map output registers. After local combiner is applied, many key-value pairs are eliminated, keeping only actual read-reference hits found. This process is done in local memory so map-side filtering is done in the fastest way. Hits found then are sorted, merged, and sent to reducers to find alignment qualities. These system operations make use of the map task buffer, so their operation is much faster than having to merge spilled-to-disk map output key-values. Finally, messages sent and reduce operations work with a percentage of the original map output data.

Automatic split of input reference genome is managed by defining a large enough number of maps. Hadoop does the job of assigning each piece of reference genome to compare with map local set of reads. As every map task is sure to have generated all hits found at its own piece of the reference, local filtering is successful to avoid the sending of seeds that do not match.

7. Results obtained

Data used for performance analysis of the application were subsets of publicly available Illumina/Solexa sequencing reads from the 1000 Genomes project (accession numbers: SRR001113) to defined segments of the human genome (NCBI build 39) allowing up to 2 mismatches. All reads were exactly 36 base-pairs long. Data sets used were of 1, 3, 5, and 7 million reads against full human genome (2.87GB), chromosome 1 (247 MB), and chromosome 22 (49,7 MB).

Test cluster had 25 IBM JS20 blade nodes with dual PowerPC 970MP processor, 4 GB of main memory and a global aggregated disk space capacity of 7.3 TB running GPFS file system version 3.1. Interconnection network was Gigabit Ethernet. All computer nodes were running Linux version 2.6.16.60 and Hadoop/HDFS version 0.20.1. Hadoop jobs were launched using Slurm batch queue system version 2.2.4.

Original serial MAQ application was run on a node of the cluster to have a reference of original application times. To evaluate its performance in comparison with new map-reduce implementation, we are showing in table 1 the execution times of mapping a growing number of reads to human chromosome 22. For a small number of reads both implementations take a similar amount of time to process the mapping. Execution times start to grow when mapping more than 3 millions of reads. Original MAQ execution needs hours to find alignments of a 5 or 6 million reads set to a small chromosome.

Number of reads	MAQ execution time	MrMAQ execution time
1 Million	300	180
3 M	900	210
5 M	25800	410
7 M	not finished	500

Table 1: Execution time (in seconds) of original serial MAQ and MrMAQ when comparing different sequence read datasets to chromosome 22.

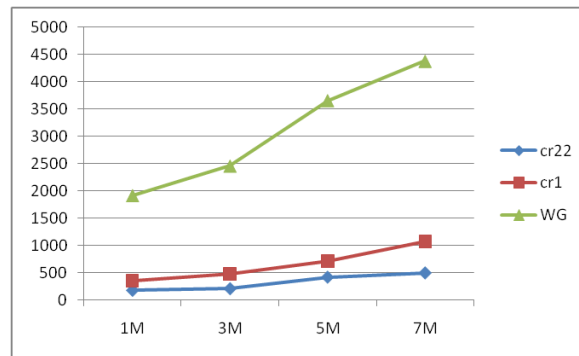


Fig. 6: Execution time (in seconds) for mrMAQ read mapping with increasing data set sizes.

Fig. 6 shows mrMAQ execution time when mapping growing set of reads to chromosome 22, chromosome 1, and the whole human genome. Map tasks memory allocation of reads to process leaves the rest of locally available memory for storing reference sequences. Then, for increasingly bigger set of reads we will have less space for reference and will need more map tasks for partitioning the reference in smaller portions. Then, the increments of execution time shown in fig. 7 are mainly devoted to the execution of more map tasks to process all reference sequences.

Millions of reads	Chromosome 22	Chromosome 1	Whole Genome
1M	3.27%	3.44%	3,33%
3M	6.76%	9.33%	9,26%
5M	9.26%	12.40%	12,27%
7M	2.76%	6.39%	9.63%

Table 2: Percentage of data filtered sent from map phase after applying gather combiner function.

In table 2 we show a general overview of the efficiency of the map-side filtering applied to mrMAQ map-reduce application. This table shows that we are reducing an average of 93% of number of messages merged and sent to reducers across the computer network when considering all executions.

This reduction of messages sent also affects the reduce phase of the application. Reducers receive less volume of input data, so data management operations like shuffling incoming data from all map tasks become less relevant in the overall application execution. To provide a measure of the execution time reduction, we show in fig. 7 a comparison of total execution times when mapping read data sets to chromosome 22. In this case, the difference between both executions is the application of the combiner optimizations.

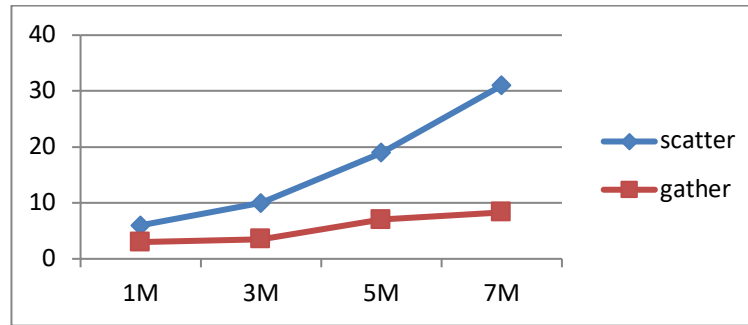


Fig. 7: Comparison of scatter vs. gather mrMAQ execution times in minutes.

Filtering of map intermediate output values provide a much more scalable use of Hadoop resources. Whole genome can be scanned for read matches without running out of machine disk buffers to store intermediate values generated by map tasks. As Hadoop parameters are configurable, resources conditions like buffer sizes and distributed cache contents can be defined before launching the execution so that a small size cluster can process all reads and references in minutes.

8. Conclusions

We describe scatter as the most common pattern of sequence alignment map reduce applications. Some of the performance implications of scatter designs are shown. Then, we suggest a method to redesign scatter to gather map reduce applications. Also we propose a list of Hadoop performance optimizations specific for sequence alignment map reduce applications describing the parameters used, their impact on the execution and some criteria to define values to regulate them.

We have used a new map-reduce implementation of MAQ algorithm to create the Hadoop equivalent of original hashed tables of reads with predefined templates to allow up to 2 mismatches. The application finds all hits for every specific reference position then returning alignment qualities for those hits with highest score.

Application gather redesign shows how locality optimization policies at map phase can improve overall execution performance by saving the processing of redundant data messages. Work describes how any simple map-reduce application can be redesigned to

follow a gathering pattern by the usage of auxiliary combiner functions. Well designed combiners help to reduce unnecessary data messages without altering the simplicity of the application design.

We also propose a particular optimization of the usage of intermediate in memory buffers and input data partitions so that Hadoop automatically manages input data assignment of map tasks and call of combiner functions. The input data split and the map filtering is done without adding new particular functions to the general Hadoop application structure and can be applied to any existing application.

In the future, Hadoop and other map reduce platforms will provide methods to expose well known diverse parallel programming patterns such as scatter to gather conversion, tiling or binning. This will allow designers to adapt their algorithm designs to be more scalable taking notice of the resource requirements of their applications.

9. Acknowledgements

We want to thank Eduard Ayguade, David Carrera and the staff at Barcelona Supercomputing Center (BSC) for their help and support to the usage of the IBM Blade computer cluster.

This paper was supported by Consolider Project CSD2007-00050 of the Spanish Ministerio de Ciencia y Tecnologia.

10. References

1. Dean J. et al. (2008): "MapReduce: simplified data processing on large clusters". Communications of the ACM, 51, pp. 107-113.
2. A. Bialecki, M. Cafarella, D. Cutting, and O. O'Malley (2005) "Hadoop: a framework for running applications on large clusters built of commodity hardware", 2005. Wiki at <http://hadoop.apache.org/>
3. S. Ibrahim, H. Jin, L. Lu, L. Qi, S. Wu: (2009) "Evaluating MapReduce on virtual machines: The Hadoop Case ". CloudCom, 2009. LNCS 5931, pp 519-528. Springer-Verlag, 2009.
4. M. Schatz (2009) "CloudBurst: highly sensitive read mapping with MapReduce". Bioinformatics, vol. 25, n. 11, pp. 1363-1369.
5. B. Langmead, M. C. Schatz, J. Lin, M. Pop, S. L. Salzberg (2009) "Searching for SNPs with cloud computing". Genome biology, 10:R134
6. S. J. Matthews, T.L. Williams (2010) "MrsRF: an efficient MapReduce algorithm for analyzing large collections of evolutionary trees". BMC Bioinformatics, 11: S15
7. C. Ranger, R. Raghurama, A. Penmetsa, G. Bradski, C. Kozykari (2007) "Evaluating MapReduce for Multi-core and Multiprocessor Systems". Proceedings of the 13th Intl. Symposium on High-Performance Computer Architecture (HPCA), Phoenix, AZ.
8. Metis
9. H. Li, J. Ruan, R. Durbin (2008) "Mapping short DNA sequencing reads and calling variants using mapping quality scores". Genome Research. 18. Pp 1851-1858
10. R. A. Baeza-Yates, et al. (1992) "Fast and practical approximate string matching". Proceedings of the combinatorial pattern matching. Third annual Symposium. Tucson, pp. 185-192.
11. R. Li, Y. Li, K. Kristiansen, J. Wang (2008) "SOAP: Short oligonucleotide alignment program", Bioinformatics, 24(5), pp. 713-714.

12. A.D. Smith et. Al. (2008) "Using quality scores and longer reads improves accuracy of Solexa read mapping". BMC Bioinformatics, 9, 128.
13. S. Babu (2010): "Towards Automatic Optimization of MapReduce Programs". Proceedings of the 1st ACM Symposium on Cloud computing. ACM, New York, 2010.
14. K. Palla (2009): "A Comparative Analysis of Join Algorithms using the Hadoop Map/Reduce Framework". Master of Science Thesis. School of Informatics. University of Edinburgh.