THE SURVEY ON MAPREDUCE

V.VIJAYALAKSHMI
Assistant professor, Department of Computer Science and Engineering,
Christ College of Engineering and Technology, Puducherry, India,
E-mail: vivenan09@gmail.com.

A.AKILA
Final Yr M.Tech, Department of Computer Science and Engineering,
Christ College of Engineering and Technology, Puducherry, India,
E-mail: ramakila18@gmail.com.

S.NAGADIVYA
Final Yr M.Tech, Department of Computer Science and Engineering,
Christ College of Engineering and Technology, Puducherry, India,
E-mail: divyacse2008@gmail.com.

Abstract:

MapReduce is a software framework that allows developers to write programs that process massive amounts of unstructured data in parallel across a distributed cluster of processors or stand-alone computers. It was developed at Google in 2004. In the programming model, a user specifies the computation by two functions, Map and Reduce. The MapReduce as well as its open-source Hadoop, is aimed for parallelizing computing in large clusters of commodity machines. Other implementations for different environments have been introduced as well, such as Mars, which implements MapReduce for graphics processors, and Phoenix, the MapReduce implementation for shared-memory systems.

This paper provides an overview of MapReduce programming model, its various applications and different implementations of MapReduce. GridGain is another open source java implementation of mapreduce. We also discuss comparisons of Hadoop and GridGain.

KEYWORDS: MapReduce, Hadoop, cloud computing, clusters, distributed computing.

1. Introduction:

MapReduce [2] is a programming model created by Google. It has been designed for simplifying parallel data processing on large clusters. The programming model is stirred by the map and reduces primitives found in Lisp and other functional languages. Before developing the MapReduce framework, Google used hundreds of different implementations to process and compute large datasets. Most of the input data was very large but the computations were relatively simple. Hence the computations needed to be scattered across hundreds of computers in order to finish calculations in a reasonable time.

MapReduce is highly efficient and scalable, and thus can be used to process huge datasets. When the MapReduce framework was introduced, Google completely rewrote its web search indexing system to use the new programming model. The indexing system produces the data structures used by Google web search. There is more than 20 Terabytes of input data for this operation. At first the indexing system ran as a sequence of eight MapReduce operations, but several new phases have been added since then. Overall, an average of hundred thousand MapReduce jobs is run daily on Google's clusters, processing more than twenty Petabytes of data every day [2].

The idea of MapReduce is to hide the difficulty of data Parallelization, fault tolerance, data distribution and load balancing in a simple library [2]. In addition to the computational problem, the programmer only needs to define parameters for controlling data distribution and parallelism [10]. Google implemented to design for large clusters of machines connected in a network. Other implementations have been introduced since the original MapReduce. For example, Hadoop [1] is an open-source implementation of MapReduce, written in Java. Like Google's MapReduce, Hadoop uses many machines in a cluster to distribute data processing.

The parallelization doesn't necessarily have to be performed over many machines in a network. There are dissimilar implementations of MapReduce for parallelizing computing in dissimilar environments. Phoenix [12] is an implementation of MapReduce, which is intended at shared-memory, multi-core and multiprocessor systems, i.e. single computers with many processor cores. Mars [7], on the other hand, is a MapReduce framework for graphic processors (GPUs). GPUs are massively parallel processors with much higher computation power and memory bandwidth than CPUs, but they are harder to program since their architecture and interfaces are designed specifically for graphics applications. MapReduce framework hides this complexity, so programmers can easily tie together the computation power of the GPU for data processing tasks.

The rest of the paper is organized as follows: Section 2 insight into mapreduce. Section 3 presents different implementations of MapReduce, and Section 4 evaluates the performance of MapReduce implementations. Section 5 contains comparison of two implementations of mapreduce. Finally, Section 6 concludes the paper.

2. Mapreduce:

2.1 *Programming Model:*

MapReduce is a programming model introduced by Google. It is stirred by the map and reduce primitive found in many functional programming languages. MapReduce framework consists of user supplied Map and Reduce functions, and an execution of MapReduce library, which automatically handles data distribution, parallelization, load balancing, fault tolerance and other common issues.

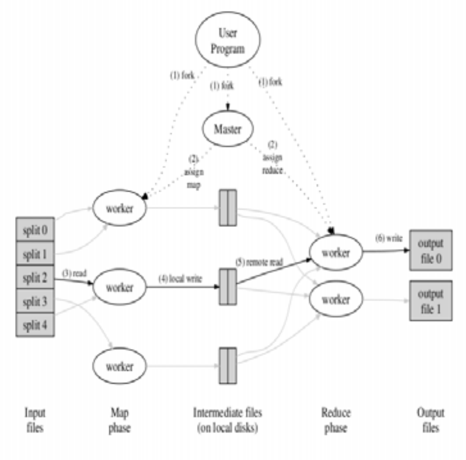

Figure 1: Execution overview

In addition, a user needs to write some configurations, like names of the input and output files, or some other, optional tuning parameters. The configurations also define how the input data is split into key/value pairs. In MapReduce programming model, users specify their calculations as two functions, Map and Reduce. The Map function takes a key/value pair as an input, and produces the output as the set of intermediate key/value pairs. Reduce takes as an input a key and a list of values assigned for it. Input values for Reduce are automatically grouped from intermediate results by the MapReduce library. After the necessary Map tasks have been completed, the library takes an intermediate key and groups it together with all the values related with it. The Reduce function takes an intermediate key and the value list assigned for it as an input. It merges the values in such the way that the user has specified in the implementation of the Reduce function, and produces a smaller set of values. Typically only zero or one output is produced per Reduce task. The programming model is intentionally limited, as it only provides map and reduce functions to be implemented by user. Because of the limitations, MapReduce can offer a simple interface for users to parallelize and distribute computations [2]. Limited programming model is good, because developers can focus on formulating the real problem with two simple functions. However, limitations make it hard to utter certain problems with the programming model. Still most data processing tasks can be efficiently implemented with MapReduce. It is easy to add new MapReduce phases to existing MapReduce operations. By adding MapReduce phases, more difficult problems can be uttered with the programming model.

Fig. 1 represents the overview of MapReduce execution. When running the user program, the MapReduce library first splits the input data into M pieces, which are typically 16- 64MB per piece. Next the library runs many copies of the program on the machines in a cluster. One of the copies is the master node, which assigns the Map and Reduce tasks to the worker nodes. There are M Map tasks to run, one for each input data split. When a worker node is assigned to a Map task, it reads the corresponding input split and passes the key/value pairs to the user-defined Map function. The intermediate key/value pairs are stored in the memory, and periodically written to local disk, partitioned into R pieces. User-defined partitioning function (e.g. hash (key) mod R) is used to produce R partitions. Locations of the intermediate key/value pairs are passed back to the master, which forwards the information to the Reduce workers when needed [2].

There are R reduce tasks. When a reduce worker receives the location of intermediate results from the master, it reads all the intermediate data for its partition from the local disk of the Map worker. Then it iterates over the intermediate pairs, and produces the output, which is appended to the final output file for the corresponding reduces partition. When all Map and Reduce are finished, master wakes up the user program, and the code execution returns from the MapReduce call back to the user code. After successful completion of the MapReduce, the output is stored in R output files, one for each Reduce task. File names of the output files are specified by the user, and they can be used for example as an input for another MapReduce operation.

2.2 Example:

The most popular example of using MapReduce is a problem of counting a number of distinct words in a large collection of documents. This example is from the original
MapReduce paper [2].

```
/* key: document name
* value: document contents*/

map(String key, String value){
for each word w in value:
emitIntermediate(w, "1");}

/* key: a word
* values: list of counts for the word*/

reduce (String key, Iterator values){
int result = 0;
for each v in values:
result += ParseInt(v);
emit(result);}
```

The Map function iterates through the document it receives as parameter, and simply emits the string”1” for every word in the document. Intermediate results are a set of key/values pairs, where keys are now different words found in the input documents, and values is a list of emitted values for each word. Before the intermediate results are passed to the Reduce tasks, values for different keys are grouped from all the Map tasks by the MapReduce library. The Reduce function takes a key and the list of values for it. Key is a word, and values is a list of”1”’s, one for each occurrence of the word in the input documents. Reduce just add these values together, and the resulting count is emitted as an output.

2.3 Applications:

Since the development of MapReduce framework, there has been quite lot of research in using MapReduce in different kinds of problem domains [8, 11, 3, 4, 6, 13]. Many computations can be done just by using the MapReduce programming model, but there are some that can’t be uttered with Map and Reduce functions.

For example, the iteration method of Genetic Algorithms cannot directly be uttered with Map and Reduce functions [8]. Genetic Algorithms are a class of development algorithms used in fields such as chemistry and biology. Parallel Genetic Algorithms have been adopted to get better effectiveness, since processing Genetic Algorithms generally takes very long time for large problems. MapReduce needs to be extensive to support such algorithms, which is achieved by adding a second reduce phase after the iterations, and a client for coordinating the implementation of iterations.

MapReduce can be used in SMS message mining [13]. SMS messages are well-liked and broadly used for simple communication between people. Number of SMS messages sent in a month in any country is very large, and so is the original dataset used in mining. Finding the most popular SMS messages can be precious information, but since the dataset is so large, parallelization is needed to complete this task in realistic time.

Hadoop, the open-source implementation of MapReduce, is used as a framework in SMS mining. Processing of the messages is done in three steps. First the original dataset is pre-processed and grouped by senders' mobile numbers. This is done by first MapReduce process. Second MapReduce process does an alteration to regroup the dataset by short content keys, and finally the third MapReduce phase is needed to mine the popular messages.

Error-correcting codes are useful in many situations. If data file needs to be saved on some defective medium, the file can be encoded with an error-correcting code. If the file is corrupted while storing, there is a chance it can be restored when decoding the error-correcting code. Encoding very large files is a dispute. Standard encoding and decoding algorithms can't handle a very large block length, that doesn't allow haphazard access to the data. Also encoding should be done without flouting the file into smaller pieces, since error-correcting achieves better performance on large files. Feldman [4] uses Google's computing infrastructure, along with Google's MapReduce implementation, to encode and decode a very large twister code. Twister codes are error-correcting codes with linear-time encoding and decoding algorithms. Twister code can be applied to huge files using parallelization offered by MapReduce framework.

Particle Swarm Optimization algorithms can be naturally expressed with MapReduce [11]. When parallelized, Particle Swarm Optimization algorithms can be used to optimize functions that have to evaluate large amounts of data. Generalized Stochastic Petri nets, on the other hand, are a popular graphical modeling formalism, that can be used in the performance analysis of computer and communications systems [6]. Calculation of reply times in such models can be done in parallel using MapReduce.

Most scientific data analysis appraises huge amounts of data. High Energy Physics experiments produce vast amounts data, which needs to be analyzed. For example, The Large Hadrons Collider is expected to produce tens of Petabytes of already filtered data in a year [3]. Another example is from the field of astronomy, where the Large Synoptic Survey Telescope produces about 20 Terabytes of data every night. Spatial queries can be decayed and processed using MapReduce to optimize the performance, such as reply time [16]. Spatial queries include spatial selection query, spatial join query, nearest neighbor query, etc. Most of spatial queries are computing intensive and individual query evaluation may take minutes or even hours. Parallelization seems to be a good solution for such problems.

3. Implementations:

3.1 Google's MapReduce:

The original MapReduce implementation by Google is targeted for large clusters of networked machines [18]. The MapReduce library mechanically handles parallelization and data distribution. Since the developers don't need to worry about things like parallel and network programming, they can focus on the real problem, i.e. presenting the computational problem with Map and Reduce functions.

Data is distributed and saved on local disks of networked machines. Google File System (GFS) [5] is a distributed file system used to supervise the data stored across the cluster. GFS makes duplicate of data blocks on multiple nodes for enhanced reliability and fault tolerance. GFS and MapReduce are intended to view machine failures as a default rather than an irregularity. MapReduce is highly scalable, and therefore it can be run on clusters comprising of thousands of low-cost machines, built on untrustworthy hardware. MapReduce library can assume that at any point, certain percentage of worker nodes will be engaged.

3.2 Hadoop:

Hadoop [1] is a MapReduce implementation by Apache. The architecture of Hadoop is essentially the same as in Google's implementation, and the main dissimilarity is that Hadoop is an open-source implementation.

Data is distributed across the machines in network using the Hadoop Distributed File System (HDFS). HDFS distributes data on computers approximately to the cluster, and creates numerous replicas of data blocks for enhanced reliability. Local drives of networked machines are used to store data, which makes the data accessible to other machines in network.

HDFS consists of two main processes, the Namenode and a number of Datanodes [1]. The elective Secondary Namenode can also be used as a back-up process for the Namenode. The Namenode runs on a single master machine. It has information about all the machines in the cluster, and details of the data blocks stored on the machines in the cluster. Datanode processes run on all the other machines in the cluster, and they communicate with the Namenode to know when to obtain data on their local hard drive. The MapReduce framework of

Hadoop consists of single JobTracker and a number of TaskTracker processes [1]. The JobTracker usually runs on the same master machine as the Namenode. Users drive their MapReduce jobs to the Job- Tracker, which splits the work between the machines in the cluster. Each other machine in cluster runs a TaskTracker process. TaskTracker communicates with the JobTracker, which assigns it a Map or Reduce task when possible. Hadoop can be organized to run multiple simultaneous Map tasks on single nodes [6]. In multi-core systems this is a great benefit, as it allows making full use of all cores.

3.3 GridGain:

Similar as Hadoop, the GridGain [17] is an open-source MapReduce implementation. From the technological point of view the biggest difference is in the preliminary process of *Map* tasks assignment to the nodes. In the MapReduce algorithm the task is split into subtasks and workers drag the split parts as soon as they have free processor time. In GridGain the subtasks are pushed to the nodes. Authors claim that this proves to be an advantage since it gives more load balancing capabilities. In practice it should be noted that this benefit is rather situational and depends on users needs. Apart of extra functionality it introduces some additional complexity – the developer has to plan in advance so that no worker does stay without reason idle. Although GridGain seems to be far less popular then Hadoop, it shows to be better documented and is more welcoming for beginners.

3.4 Phoenix:

Phoenix [12] is a MapReduce implementation intended for shared-memory systems. It consists of MapReduce programming model and related runtime library that handles resource management, fault tolerance and other issues mechanically. It uses threads to create parallel Map and Reduce tasks. Phoenix can be used to parallelize data intensive computations on multi-core and multiprocessor computers.

The principles in Phoenix implementations are essentially the same as in original MapReduce, except instead of large clusters, it is intended for shared-memory systems. Overheads caused by task spawning and data communications can be minimized when working in a shared-memory environment. The runtime uses P-threads to generate parallel Map or Reduce tasks, and schedules tasks dynamically to available processors [12].

In Phoenix, addition to Map and Reduce functions, the user provides a function that partitions the data before each step, and a function that implements key comparison. The programmer calls phoenix scheduler () to start the MapReduce process. The function takes arrangement structure as an input, in which the user specifies the user-provided functions, pointers to input/output buffers and other options. The scheduler controls the runtime, and manages the threads that run all the Map and Reduce tasks. Phoenix spawns threads on all available cores, trying to take full advantage of the system [12].

3.5 Mars:

Mars [7] implements the MapReduce framework for graphics processors (GPU). GPUs are massively parallel processors with 10x higher computation power and memory bandwidth than CPUs. Since GPUs are special purpose processors designed for gaming applications, their programming languages lack support for some basic programming structures, like variable-length data types or recursion. Additionally, different GPU vendors have different architectural details in their processors, which make programming even more difficult. Several GPGPU (General-Purpose computing on GPUs) languages have been introduced, that can be used to write GPU programs without the knowledge of the graphics rendering pipeline. An example of such language is NVIDIA CUDA, which was also used to implement Mars.

The purpose of the Mars framework is to hide all the complex details of the GPU. Threads are handled by the runtime library. Characteristics of the user defined Map and Reduce functions, and the number of multiprocessors and other computation resources are taken into account when deciding the number of threads. GPUs don’t support dynamic thread scheduling, so it is important to allocate threads correctly before executing the MapReduce process [7]. Since GPUs don’t support dynamic memory allocations, arrays are used as the main data structure in Mars. Space for all the input, intermediate and result data must be allocated on the device memory before executing the program on GPU. Three kinds of arrays are used to save the input data and results. Key and value arrays contain all the keys and values, and a directory index array consists of entries for each key/value pair. Directory index entries are in format <key offset, key size, value offset, value size>, and they are used to fetch keys or values from the corresponding arrays.

Mars workflow starts with preprocessing the raw input data into key/value pairs. CPU is exploited for this task, since GPUs don’t allow direct access to the disk [7]. The key/value pairs are then copied to the device memory of the GPU, and divided into chunks, such that the number of chunks is equal to the number of threads. Dividing the input data evenly on the threads makes this implementation load-balanced. After the Map stage is done, the intermediate key/value pairs are sorted. In Reduce stage, the split operation divides the sorted intermediate key/value pairs into multiple chunks, such that pairs with the same key belong to same chunk. Again, one thread is responsible of one chunk, so the number of chunks is same as the number of threads.

3.6 Map-Reduce-Merge:

Map-Reduce-Merge [14] can be considered as an extension to the MapReduce programming model, rather than an implementation of MapReduce. Original MapReduce programming model does not directly support processing multiple related heterogeneous datasets. For example, relational operations, like joining multiple heterogeneous datasets, can be done with MapReduce by adding extra MapReduce steps. Map-Reduce-Merge is an improved model that can be used to express relational algebra operators and join algorithms. This improved framework introduces a new Merge phase, that can join reduced outputs, and a naming and configuring scheme, that extends MapReduce to process heterogeneous datasets simultaneously [14]. The Merge function is much like Map or Reduce. It is supplied by the user, and it takes two pairs of key/values as parameters. Unlike Map, that reads a key/value pair, or Reduce that processes a value list for a key, Merge reads data (key/values pairs) from two distinguishable sources.

Workflow in MapReduce programs is restricted to two phases, i.e. mapping and reducing. Users have very few options to configure this workflow. Adding a new Merge phase creates many new workflow combinations that can handle more advanced data-processing tasks. Furthermore, Map-Reduce-Merge provides a configuration API for users to build custom workflows. Map-Reduce-Merge can be used recursively, because the workflow allows outputs to be used as an input for next Map-Reduce-Merge process.

4. Evaluation:

4.1 Fault tolerance

MapReduce handles failures by re-executing the failed job on some other machine in a network. The master process, JobTracker, periodically pings the worker nodes, TaskTrackers. JobTracker and TaskTracker are the main processes in Hadoop, but the original MapReduce has similar processes. If the master receives no response from a worker, that worker is marked as failed, and its job is assigned to another node [6]. Even completed Map tasks have to be re-executed on failure, since the intermediate results of the Map phase are on the local disk of the failed machine, and are therefore inaccessible. Completed Reduce tasks, on the other hand, do not need to be re-executed, as their output is stored in a global, distributed file system [2].

In a large cluster, chances of a worker node failing are quite high. Inexpensive IDE disks are often used as local storage space for the machines in a cluster, and therefore disk failures are common in large clusters. On the other hand, chances for the master node failing are low. That is why in Hadoop, there is no fault tolerance for JobTracker failures [6]. If the machine running JobTracker fails, the entire MapReduce job has to be re-executed. To minimize the chance of the master node failing, JobTracker should be run on a machine with better quality components and more reliable hardware than the worker nodes.

Common cause for a MapReduce operation to take much more time than expected is a straggler [2]. Straggler is a machine that takes an unusually long time to complete one of the lastMap or Reduce tasks. This can be caused by errors in machine, or simply by the machine being busy doing something else. MapReduce library uses backup tasks to deal with stragglers. When a MapReduce operation is close to finish, the master schedules a backup process for remaining tasks in progress. The task is marked as completed whenever the primary or the backup task completes. According to the authors of the original MapReduce paper [2], backup tasks significantly reduce the time to complete large MapReduce operations. The paper reports that an example sort program, presented in the same paper, takes 44% longer to complete when the backup task mechanism is disabled.

4.2 Performance

Network bandwidth is a valuable resource in a cluster. To reduce the amount of data needed to transfer across network, a Combiner function is run on the same machine that ran a Map task. The Combiner merges the intermediate results on the local disk, before it is transferred to the corresponding Reduce task. Map tasks often produce many key/value pairs with the same key. This way those key/value pairs with the same key are merged, instead of transferring them all individually. Another way to reduce network bandwidth in Hadoop is taking advantage of the data replication in HDFS.

When a node asks for some data from the Namenode, the master node in HDFS, it returns the location of data, which is closest to the worker node on the network path [6]. To further reduce the required bandwidth, the MapReduce framework always tries to run Map tasks on machines that already has copies of corresponding data blocks on their local disks. MRBench [9] is a benchmark for evaluating the performance of MapReduce systems. MRBench is based on TPCH, which is a decision support benchmark, containing industry related data with 8 tables and 22 queries. MRBench was implemented in Java, to be supported by the open-source MapReduce implementation, Hadoop. MRBench supports three configuration options: database size, the number of Map tasks and the number of Reduce tasks. However, the number of Map/Reduce operations is just a suggestion for the MapReduce, and final number of tasks depends on input file splits.

MRBench scalability is shown on the experiment that compares processing of two databases of different sizes. Default numbers of Map/Reduce tasks are used, and runtime of the system is measured with databases of size 1GB and 3GB. Experiment shows that runtime for 3GB dataset is almost three times longer than that of 1GB dataset on every query. When experimenting with various numbers of nodes for the computation, it is noted that the speedup gain is not linear. Computation time is calculated with 8 and 16 nodes. Only processing time, i.e. the time spent processing data on Map/Reduce phase, is reduced when increasing the number of nodes. Data management time, the time spent on managing intermediate data and transferring and writing files, might even be increased due to the doubled number of nodes. Experiments with different numbers of Map tasks showed that the optimal number of Map tasks increases as the input data grows [9].

The authors of Mars [7] compared their implementation with Phoenix, the MapReduce implementation for multicore CPUs. For large datasets, Mars is around 1.5-16x faster than Phoenix. The speedup varies for different kinds of computational problems. They also implemented Mars to use CPU instead of GPU, and noted that their CPU-based implementation achieves at least as good performance as Phoenix. Mars was also compared to its CPU-based implementation, which showed that the GPU-based Mars is up to 3.9x faster than its CPU implementation. The tests were run on a machine that had a CPU with 4 cores running at 2.4GHz, and a GPU consisting of 16 multiprocessors, each of which had 8 cores running at 1.35GHz.

The paper describing Phoenix [12] compared the performance of their implementation to parallel code written directly with P-threads. Algorithms that fit well to the key based structure of MapReduce, gain the most significant speedups. On the other hand, fitting an unsuitable algorithm to the model may lead to significant overhead, caused by key management, memory allocations, and copying and sorting data. For algorithms that don't directly fit into MapReduce model, P-threads implementations outperform the Phoenix. They conclude that despite of the implementation, MapReduce leads to good parallel efficiency whenever the problem is easily expressed with the model, but the model is not general enough to support all problem domains. When using MapReduce in a cluster of computers, it is easy to improve computation power by adding new machines to the network. More nodes mean more parallel Map/Reduce tasks.

5. Comparison

5.1 Hadoop

Hadoop is designed to work with large data sets (100's of TB in a single job). In Hadoop, each map and reduce may generate zero or more key/value pairs. Hadoop provides HDFS that provides reliable scalable store to 1000's of nodes and PB of data. It provides an automatic distributed sort of the data between the maps and reduces. Hadoop uses a much more flexible serialization, which can use either java.io or user-defined serialization. It provides a web interface to track the job's progress. Hadoop supports both C++ and text-based applications. Hadoop has More Supporters.

5.2 GridGain:

GridGain is not suitable for large data sets. In GridGain's system each map and reduce returns a single value. GridGain provides only distributed computation support and doesn't have a distributed file system. GridGain's framework does not sort the data between the maps and the reduce. GridGain uses java.io serialization. It does not support combiners, or counters. It does not provide a web interface to track the job's progress. GridGain doesn't have any support for non-Java applications. GidGain has only one supporter that is GridGain.

6. Conclusion:

MapReduce is a software framework that allows developers to write programs that process massive amounts of unstructured data in parallel across a distributed cluster of processors or stand-alone computers. This paper contains various applications, implementations and performance evaluations of mapreduce. Some of the characteristics, functionalities are compared with Hadoop and GridGain.

References

[1] Apache. Hadoop documentation, http://hadoop.apache.org/core. 2008.

[2] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. Commun. ACM, 51(1):107–113, 2008.

[3] J. Ekanayake, S. Pallickara, and G. Fox. Mapreduce for data intensive scientific analyses. eScience, 2008. eScience '08. IEEE Fourth International Conference on, pages 277–284, Dec. 2008.

[4] J. Feldman. Using many machines to handle an enormous error-correcting code. Information Theory Workshop, 2006. ITW '06 Punta del Este. IEEE, pages 180– 182, March 2006.

[5] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles, pages 29–43, New York, NY, USA, 2003. ACM.

[6] O. J. Haggarty, W. J. Knottenbelt, and J. T. Bradley. Distributed response time analysis of gspn models with mapreduce. Performance Evaluation of Computer and Telecommunication Systems, 2008. SPECTS 2008. International Symposium on, pages 82–90, June 2008.

[7] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: a mapreduce framework on graphics processors. In PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques, pages 260–269, New York, NY, USA, 2008. ACM.

[8] C. Jin, C. Vecchiola, and R. Buyya. Mrpga: An extension of mapreduce for parallelizing genetic algorithms. eScience, 2008. eScience '08. IEEE Fourth International Conference on, pages 214–221, Dec. 2008.

[9] K. Kim, K. Jeon, H. Han, S.-g. Kim, H. Jung, and H. Y. Yeom. Mrbench: A benchmark for mapreduce framework. Parallel and Distributed Systems, 2008. ICPADS '08. 14th IEEE International Conference on, pages 11– 18, Dec. 2008.

[10] R. Lammel. Google's mapreduce programming model – revisited. Science of Computer Programming, 70(1):1 – 30, 2008.

[11] A.McNabb, C. Monson, and K. Seppi. Parallel pso using mapreduce. Evolutionary Computation, 2007. CEC 2007. IEEE Congress on, pages 7–14, Sept. 2007.

[12] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluatingmapreduce formulti-core and multiprocessor systems. High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on, pages 13–24, Feb. 2007.

[13] T. Xia. Large-scale sms messages mining based on map-reduce. Computational Intelligence and Design, 2008. ISCID '08. International Symposium on, 1:7–12, Oct. 2008.

[14] H.-c. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data, pages 1029–1040, New York, NY, USA, 2007. ACM.

[15] J. H. Yeung, C. Tsang, K. Tsoi, B. S. Kwan, C. C. Cheung, A. P. Chan, and P. H. Leong. Map-reduce as a programming model for custom computing machines. Field-Programmable Custom Computing Machines, 2008. FCCM '08. 16th International Symposium on, pages 149–159, April 2008.

[16] Shubin Zhang, Jizhong Han, Zhiyong Liu, Kai Wang, Shengzhong Feng. Spatial Queries Evaluation with MapReduce. 2009 Eighth International Conference on Grid and Cooperative Computing

[17] Gridgain: http://www.gridgain.com/

[18] Tomi Aarnio. Parallel data processing with MapReduce. TKK T-110.5190 Seminar on Internetworking

[19] Mapreduce: http://searchcloudcomputing.techtarget.com/sDefinition/0,,sid201_gci1365297,00.html.