Building a Java MapReduce Framework for Multi-core Architectures

George Kovoor, Jeremy Singer and Mikel Luján
Advanced Processor Technologies Group
The University of Manchester, UK

Abstract. MapReduce is a programming pattern that has been proved to be a simple abstraction on top of which can be built an efficient platform for large-scale data processing in distributed environments, such as Google or Hadoop. With this pattern, application logic is expressed using sequential *map* and *reduce* functions. Thus, a runtime system can exploit the lack of side effects (*pure functions*) in these functions to execute concurrently. The runtime framework also takes care of the low-level parallelisation and scheduling details. The success of the MapReduce pattern has led to several implementations for various scenarios. This paper introduces MR-J, a MapReduce Java framework for multi-core architectures, and reports the scalability results from the first experiments.

Keywords: MapReduce, parallel software framework.

1 Introduction

The MapReduce programming pattern was, by no means, invented by Google. Its roots can be traced back to functional programming [1]. Nonetheless, it has attracted a fair amount of attention from industry, academics and open-source projects (Hadoop [3]), since Google made public [2] that in their experience this pattern was easy to use and provided a highly effective means of attaining massive parallelism in large data-centers. The pattern is not a silver bullet that can be applied to any general-purpose application (some consider it a step backwards [8]), but it covers an important part of the application spectrum. Our objective is to investigate the MapReduce pattern in the context of multi-core architectures and not within data-centers as commonly used by Amazon, Facebook, Google and Yahoo, to name a few [5].

We have selected Java as the programming language because the main open-source implementation of MapReduce, *i.e.* Hadoop, is also developed in this language. Thus, our efforts and findings can contribute directly to the evolution of Hadoop. These efforts have produced MR-J, a MapReduce Java framework for multi-core architectures, and this paper reports the results from the first experiments.

The Phoenix framework [6, 7] is the only other MapReduce framework that targets multi-core architectures. Since Phoenix is implemented in C, the implementation can take advantage of pointers (*e.g.*, to avoid copying data) in ways that are not feasible within Hadoop or MR-J. Thus, the contribution of this paper is to report for the first

time the scalability of 4 benchmarks implemented with MapReduce in Java on a small multi-core architecture.

The paper is organised as follows. Section 2 presents the background on MapReduce and a brief description of its implementation on clusters or data-centers. Section 3 takes a closer look at a MapReduce implementation focused on a single chip; specially Phoenix. Section 4 describes the implementation of MR-J based around a divide-and-conquer approach. Section 5 shows the results from our first experiments on a small multi-core (Intel core i7) system using 4 different benchmarks. Finally Section 6 summarises MR-J and its first performance evaluation.

2 Background on MapReduce

The primary advantage of using the MapReduce pattern is that the application logic is expressed using sequential *map* and *reduce* functions. Grounded on functional programming, these functions must not have side effects; *i.e.* must be *pure functions*. Thus, a runtime system can exploit this property and execute the functions concurrently. MapReduce facilitates the automatic parallelisation of applications through the guarantees provided by the functional programming construct.

The `map` function takes <key, value> pair as input and emits an intermediate <key, value> pair as output. The inputs to the `map` and `reduce` functions are processed independently without any dependency on other elements of data, thereby avoiding the need for synchronisation and contention for shared data. In general, the majority of the frameworks implementing the MapReduce pattern has at least map, merge and reduce phases. In the map phase, the `map` function is executed in parallel by different worker threads as map subtasks. The output from each map subtask is written to a local data structure; such as arrays, lists, or queues. The execution of the map phase is followed by the merge phase. In this phase the runtime system combines the intermediate <key, value> pair output from each map subtask so that values from the same keys are grouped together to form a unique <key, value> pair. The framework partitions the unique <key, value> pairs among the reduce task workers. As with the map task, the reduce tasks are executed in parallel without any dependencies on other elements. The reduce task usually performs some kind of reduction operation such as summation, sorting and merging operations. The output from each reduce task worker is written to either a distributed file system in the case of cluster-based implementations, or it is written to a local data structure, which is then merged to produce a single output, in the case of multi-core architectures.

Cluster-based implementations of MapReduce (such as Google's and Hadoop) normally target large data-centers using commodity systems interconnected using high-speed Ethernet networks. These implementations rely on specialised distributed file systems such as Google's GFS [11] and Hadoop's HDFS to manage data across the distributed network. The implemented runtime system spawns worker threads on each node in the cluster. Each of these worker threads is assigned either a map or a reduce task. Only one of the worker threads is elected to be the master. Each job consists of a set of map and reduce tasks along with the input data. The runtime system automatically partitions the input data based on a *splitting* function into

smaller partitions or *chunks*. The selection of the chunk size is based on the block size of the distributed file system used by the framework. In the case of Google a default chunk size of 16MB to 64MB is used, whereas in the case of Hadoop a default chunk size of 64MB is used.

The master worker assigns these partitioned data to map task workers, which are distributed among the cluster nodes. Each map task worker processes the input data in parallel without any dependency. The output from the map task worker is stored locally on the node on which the task is executing. Once all the map tasks are completed, the runtime system automatically sorts the output from each map task worker so that the values from the same intermediate keys are grouped together before it is assigned to the reduce task worker. The reduce tasks are assigned to the reduce task workers by partitioning the intermediate sorted keys using the partitioning function. The default partitioning function used in both Google's model and Hadoop is based on key hashing: `hash(key) mod R`, where R is the number of reduce task workers.

In a distributed environment the master is responsible for the following tasks. It maintains the status information and identity for map and reduce tasks. It is responsible for transferring the location of the file from the map task worker to the reduce task worker. It delegates map or reduce tasks to each worker, maintains locality by assigning tasks locally to the workers and manages the termination of each worker. An essential feature for any cluster-based implementation is its ability to detect failures and slow tasks during execution. This is important because machine failures can be frequent due to the large cluster size. Both Google's framework and Hadoop implement effective fault-tolerance mechanisms that can detect slow nodes and node failures.

3 MapReduce on multi-cores

Implementations of MapReduce that are not targetting clusters have started appearing since 2007. For example, He *et al.* [9] and Kruijf *et al*. [10] have developed implementations for GPGPUs and Cell processors, respectively. The Phoenix project [6, 7] is the only previous implementation that focuses on shared memory multi-core architectures. Accordingly we discuss this implementation at greater length. The underlying principle of Phoenix is based on Google's MapReduce framework; hence they share several features. Phoenix is implemented in C and P-threads. The major difference when comparing Phoenix with Google's framework is that it uses threads to do the processing instead of worker nodes and relies on inter thread communication instead of remote procedure calls [1]. The fundamental structure of the Phoenix API resembles Google's MapReduce API, as both maintain a narrow and simplified interface. The Phoenix runtime system creates and manages threads across multiple cores, dynamically schedules map and reduce tasks to the worker threads, handles communication among worker threads and maintains state for each worker thread.

Figure 1 summarises the execution workflow within Phoenix. The user program initialises the scheduler by invoking an initialisation function. The scheduler spawns several worker threads according to the number of cores supported by the multi-core

chip. Each worker thread is assigned map or reduce tasks by the scheduler. The scheduler partitions the input data using the splitter function. The default splitter function uses the cache sizes of the chip to determine the chunk size. Partitioned data is forwarded to the map task workers assigned dynamically by the scheduler. Once the map task worker completes processing the input data, it stores the output in a buffer. It also merges the values from the same key to form a unique key for each set of values. Reduce tasks start only when the entire map task completes.

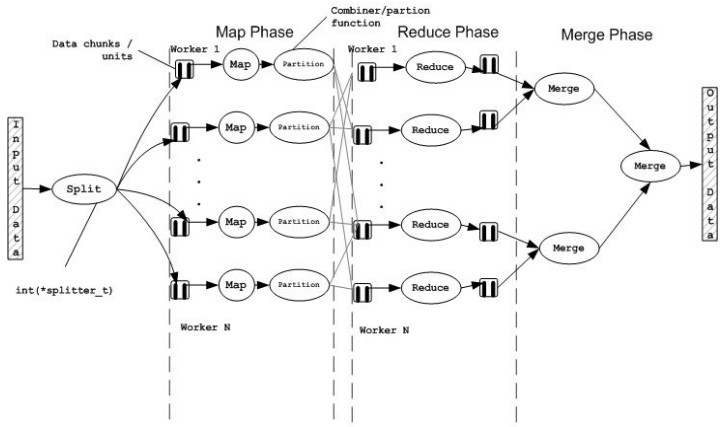

Figure 1 Phoenix execution overview (image adapted from [6]).

In the reduce phase the reduce tasks are assigned dynamically by the scheduler based on the partitioning function. The reduce task worker computes the data from the intermediate map queue. The output from the reduce task worker is stored in the local reduce queue to minimise the contention.

The merge phase is considered optional. In this phase the output from the reduce queue is merged together to produce a single output. In Google's implementation, this phase is achieved by making a recursive call to MapReduce. Ranger *et al.* [6] have reported that a merge phase is not required for most of their benchmarks and the overhead associated with it is comparatively lower than making recursive calls to MapReduce.

The default splitting function to partition the input data uses the L1 data cache size to determine a good chunk size of the partition. Phoenix also provides an option for developers to specify a custom partitioning function. This custom function can take advantage of application knowledge and avoid problems with load imbalance; keys can be associated with different values, the computation required by various keys may differ resulting in load imbalance.

Phoenix detects faults through a timeout mechanism. It uses execution time of a similar task as a measure to determine the maximum duration required to execute a task. Limitations of the fault-tolerance mechanism implemented in Phoenix system are identified as follows; lack of support to determine if a task is completed successfully by a worker thread and single point of failure for the scheduler.

To sum up, Phoenix is a mature and sophisticated software that is in its second public release. The implementation of MR-J is similar in many aspects, but we make no attempt, for the time being, to provide fault-tolerance mechanisms. We are focusing on evaluating whether a Java implementation of MapReduce can scale. In other words, we want to understand whether not having the same low level of control as in C will affect scalability.

4 Overview of MR-J

The design of MR-J shares many features with Hadoop at the application interface level (and also, *e.g.*, for job submission, setting the configuration parameters, and initialising the framework) as both frameworks are implemented in Java. In contrast to Hadoop, the current implementation of MR-J is designed specifically for multi-core architectures; as a result, the execution flow is closer to Phoenix.

Comparing the application interface with Phoenix, the WordCount application (see description in Table 1) requires the programmer to provide implementation for four user-defined functions (`map`, `reduce`, `splitter`, and `keycmp`). On the other hand, MR-J requires only the first two functions.

The distinguishing feature of MR-J is that it exploits a recursive divide-and-conquer approach. The implementation takes advantage of this by relying on work stealing and the Java fork-join framework (part of pre-release version of `java.util.concurrent` package for JDK1.7).

The sequence diagrams illustrated in Figure 2 and 3 provide an execution overview of the map and reduce phases. UML2 notations for parallel combined fragments (highlighted box in the diagrams) are used to indicate fork/join parallel executions. Partitioning of the input data creates subtasks with equal size of partitioned data units. Worker threads in the fork-join pool execute the generated subtasks. The number of worker threads initialised in the pool is a parameter of the framework and, normally, corresponds to the number of cores available. Thus, MR-J generates tasks dynamically and has a flexible mechanism to control the granularity of subtasks.

In order to improve performance for recursive calls to the `map` and `reduce` functions, a job-chaining functionality, similar to Hadoop's, has been implemented in MR-J. Job chaining enables multiple calls to map and reduce phases during a single MapReduce execution. This feature is required, for example, to implement the Kmeans benchmark and is not part of Phoenix where the runtime iterates through the map and reduce phases to compute the final cluster for a given set of coordinates. The benefit of using the job-chaining feature is that all the worker threads and data structures created during the first map and reduce phases are reused.

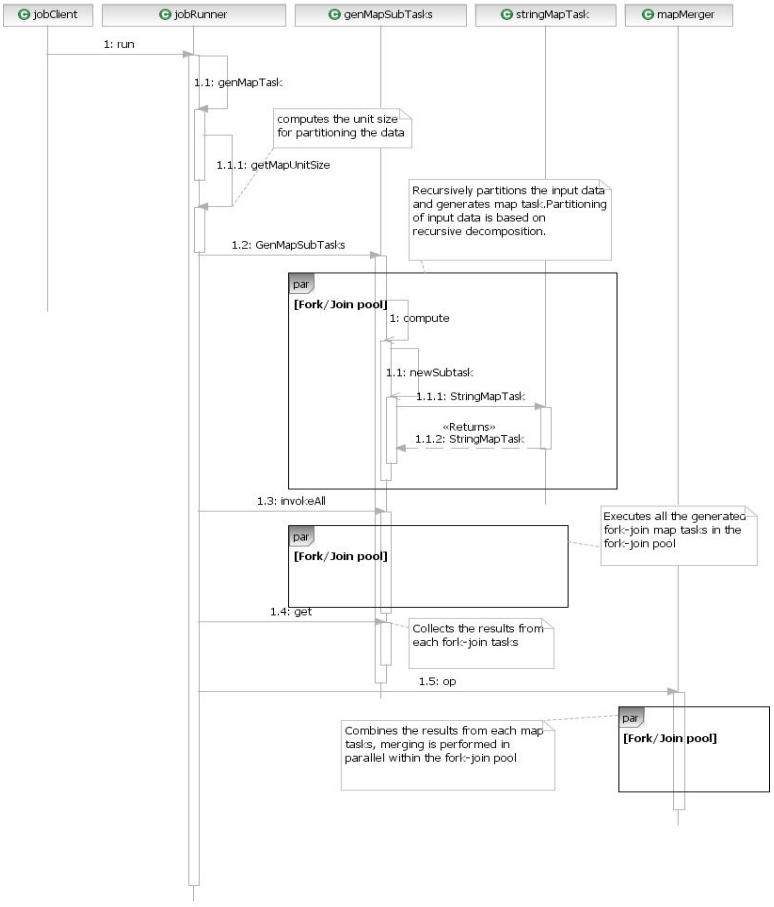

Figure 2 Sequence diagram for the map phase in MR-J.

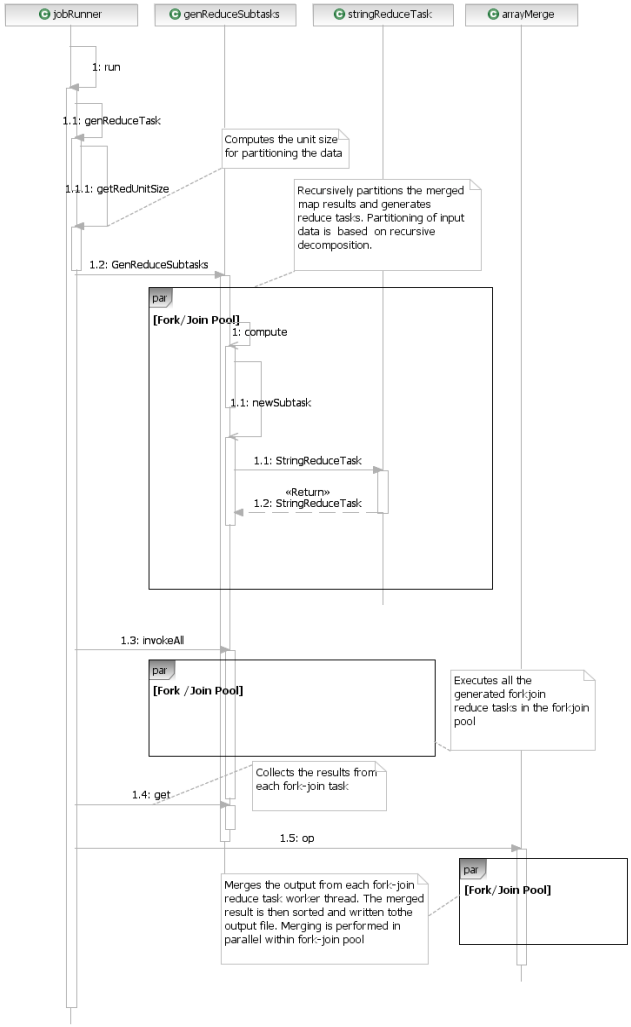

Figure 3 Sequence diagram for the reduce phase in MR-J.

5 Evaluation

The experiments consider the scalability of the MR-J framework and also examine the execution time breakdown of the different phases of execution of MapReduce.

5.1 Experimental Setup

Experiments are performed on a small multi-core system with one Intel Core i7 processor (*i.e.* four cores, 2 hyper-threads per core) running at 2.6GHz with 6GB of memory. The machine has an OpenSuse Linux 11.1 installation with Sun JVM version 1.6 (build 14.0-b08) using a fixed size heap of 4GB. Each experiment is executed five times and the average time is used in the results. We use the nanosecond resolution timer and the speedup is calculated according to T_1/T_p, where T_1 refers to the execution time using MR-J on a single thread and T_p refers to the parallel execution time on p threads. For all the experiments MR-J is configured so that it generates the same number of map tasks as the number of threads in the pool. The number of reduce tasks is adjusted dynamically based on the estimates of the reduce task cost and the number is always equal or less than the number of threads in the pool.

Table 2 presents a summary of the benchmarks we have implemented and the datasets used in the experiments. These benchmarks have been used for the evaluation of Phoenix as well as other MapReduce frameworks. It follows a brief description of how each benchmark is implemented within the MapReduce framework:

WordCount: Counts the number of times each word is repeated in the input document (text file). Each map task operates on different chunks of input data (text file). It reads each line from the input chunk and emits an intermediate <key, value> pair, where key is a word in the line and value is a number assigned to each word. Implementation of map function uses `StringTokenizer` to extract the word from each line based on whitespaces (*i.e.*, default delimiter). Reduce task sums the values associated with the word.

Grep: Extracts a given regular expression pattern from the input file and counts the occurrence of the pattern in each line. Grep is implemented in a similar way to WordCount with some minor changes to the implementation of the `map` function. The `map` function in Grep makes use of `java.util.regex.Matcher` to extract <key, value> pairs, where key represents a matching pattern and value is a number associated with it. The reduce task sums the values for each matching pattern together. As with the WordCount application, in Grep a `map` function is called for each line in the input document and `reduce` function is called for each unique key produced by the map tasks.

Kmeans: Groups a set of coordinates to the nearest clusters (K). The algorithm mainly consists of the following tasks: to determine the centre for each cluster, to calculate the distance from each coordinate to the clusters, and to assign the coordinates to the nearest cluster. Since the computation performed in Kmeans is iterative, the implementation uses job chaining, a feature discussed in Section 4. The coordinates are partitioned dynamically among the map tasks. Each map task computes the distance from each set of allocated coordinates to the clusters and identifies the nearest cluster for a given coordinate. The <key, value> pair output from the map tasks consists of a cluster index as key and an index of the coordinate as value. Reduce tasks are executed for each cluster updating the cluster and recalculating the centroid for each cluster. This process continues until all the coordinates converge to the nearest cluster.

Matrix-Multiply: For given matrices A, B, and C, the matrix B is partitioned column wise (along the second dimension) among the map tasks. Each map task computes the result of the corresponding column in C, using a block partition of matrix B and reference to all elements in matrix A. The reduce phase is not required.

Each of the benchmarks provides an optional argument to write the results to an output file. The output results from the benchmarks are verified by comparing it with the sequential version.

Name	**Description**		**Data-Set**	
		Small	Medium	Large
WordCount	Counts the frequency of words in a file.	10MB	50MB	100MB
Grep	Extracts a given regex pattern from the input file.	10MB	50MB	100MB
Kmeans	Clustering algorithm for 3D data points	10K Points	50K Points	100K Points
Matrix-Multiply	Performs integer matrix multiplication.	512x512	1024x1024	4096x4096

Table 1 Summary of the benchmarks used in the experiments.

5.2 Scalability

Figures 4, 5 and 6 show the speedup graphs for the three different datasets. Immediately we can observe that the benchmarks can be separated into two groups. The speedup curve for Grep and WordCount presents good scalability reaching their peak values with 6 threads. For the small dataset, the speedups are above 5.5, while for the other two datasets they are above 10 and 12. With more than 6 threads, the interference between the JVM threads, OS threads and the application itself prevent higher improvements. The hyper-threading mechanism shows improvements when executing 4 to 6 threads.

The speedup curves for Kmeans and Matrix-Multiply are more modest reaching only values between 2 and 4. For all the datasets, these two curves are flatter, specially with more than 6 threads.

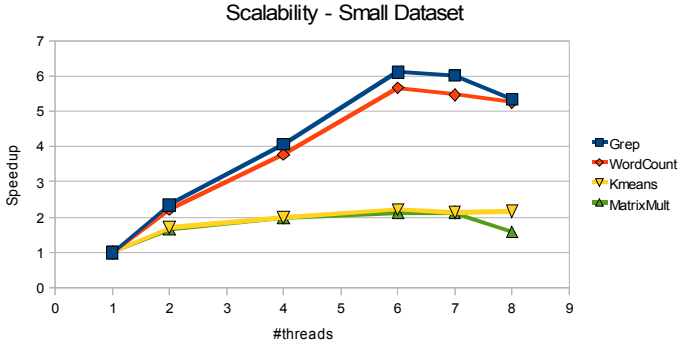

Figure 4 Scalability results with the small datasets.

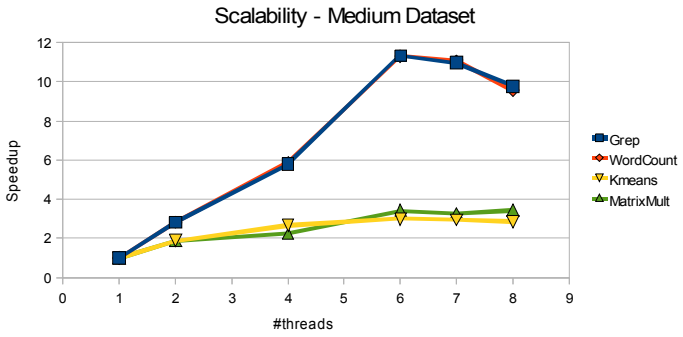

Figure 5 Scalability results with the medium datasets.

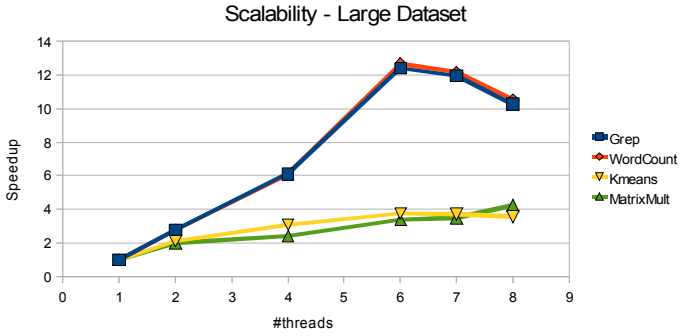

Figure 6 Scalability results with the large datasets.

5.3 Execution Time Breakdown

Figure 7 provides the execution time breakdown for the four benchmarks, but only for the large datasets. Note that the Matrix-Multiply benchmark only executes the map phase. The vertical axis (Y-axis) represents the normalised execution time relative to the map task execution time using only one thread. The horizontal axis denotes the number of threads; each benchmark application is grouped. The times for the map and reduce phases include the time required to generate and execute the Map and Reduce tasks. The time is clearly dominated by the merge phase. Kmeans is the only benchmark that exhibits noticeable reduce and merge phases.

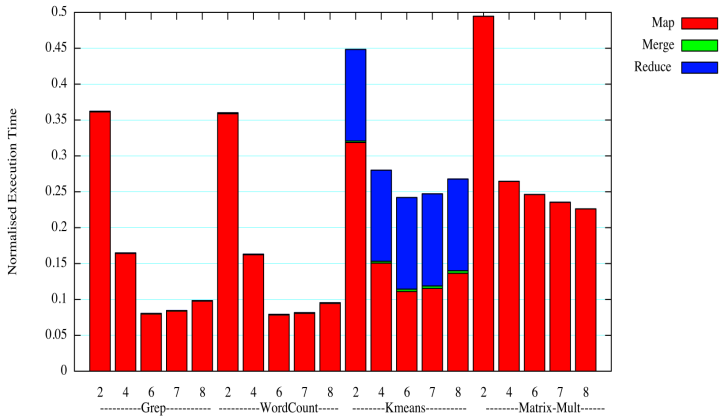

Figure 7 Execution time breakdowns for the large datasets.

6. Summary

MapReduce is a programming pattern that has been proved to be a simple and efficient platform for large-scale data processing in distributed environments, such as Google or Hadoop. Since 2007, implementations of MapReduce that are not targetting clusters or data-centers have started to appear. For example, He *et al.* [9] and Kruijf *et al*. [10] have developed implementations for GPGPUs and Cell processors, respectively. The Phoenix project [6, 7] is the only previous implementation that focuses on shared memory multi-core architectures.

MR-J is the only Java MapReduce framework that targets these multi-core architectures and shares similarities with Phoenix and Hadoop. However, a distinguishing feature is that MR-J is designed around a divide-and-conquer approach. The contribution of this paper is to report its first scalability analysis on an Intel core i7 system. We have shown that for two of the benchmarks MR-J is able to reach above 5 times speedup (maximum more than 12 times) over the T_1 using up to 8 threads. For the remaining two benchmarks scalability is more modest reaching speedup values between 2 and 4. Looking at the execution time breakdown, the most demanding phase is map, which dominates on all the benchmarks.

As future work, we want to investigate the impact of task granularity and the work-stealing behaviour on larger multi-core architectures. We also plan on increasing the number of benchmarks ported to MR-J.

Acknowledgments. Dr. Mikel Luján is supported by a Royal Society University Research Fellowship.

References

1. J. Dean, and S. Ghemawat, "MapReduce: simplified data processing on large clusters" Communications of the ACM, vol. 51, no. 1, pp. 107-113, 2008.
2. J. Dean, and S. Ghemawat, "MapReduce: simplified data processing on large clusters" In Proceedings of the 6th Symposium on Operating Systems Design & Implementation, pp. 137-150, 2004.
3. "Hadoop: Open source implementation of MapReduce", http://lucene.apache.org/hadoop/
4. T. White, "Hadoop: The Definitive Guide", O'Reilly, 2009.
5. Organizations using Hadoop. http://wiki.apache.org/hadoop/PoweredBy
6. C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating MapReduce for Multi-core and Multiprocessor Systems" In Proceedings of the 13th International Symposium on High Performance Computer Architecture (HPCA), pp. 13-24, 2007,.
7. R.M. Yoo, A. Romano, and C. Kozyrakis, "Phoenix Rebirth: Scalable MapReduce on a NUMA System" In Proceedings of the International Symposium on Workload Characterization (IISWC), pp. 198-207, 2009.
8. D.J. DeWitt, and M. Stonebraker, "MapReduce: A major step backwards" The Database Column, 2008.
9. B. He, W. Fang, Q. Luo, N.K. Govindaraju, and T. Wang, "Mars: a MapReduce framework on graphics processors" In Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT), pp. 260-269, 2008.
10. M. de Kruijf, and K. Sankaralingam, "MapReduce for the Cell B.E. Architecture" IBM Journal of Research and Development, 53(5), 2009.
11. S. Ghemawat, H. Gobioff, S.-T. Leung, "The Google file system" In Proceedings of the 19th ACM Symposium on Operating Systems Principles, pp. 29-43, 2003.