The Limitation of MapReduce: A Probing Case and a Lightweight Solution

Zhiqiang Ma Lin Gu
The Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
Kowloon, Hong Kong
Email: {zma,lingu}@cse.ust.hk

***Abstract*—MapReduce is arguably the most successful parallelization framework especially for processing large data sets in datacenters comprising commodity computers. However, difficulties are observed in porting sophisticated applications to MapReduce, albeit the existence of numerous parallelization opportunities. Intrinsically, the MapReduce design allows a program to scale up to handle extremely large data sets, but constrains a program's ability to process smaller data items and exploit variable-degrees of parallelization opportunities which are likely to be the common case in general application. In this paper, we analyze the limitations of MapReduce and present the design and implementation of a new lightweight parallelization framework, MRlite. MRlite can efficiently process moderate-size data with dependences among numerous computational steps. In the mean time, the parallelization on each step emulates the MapReduce model. Hence, the MRlite framework can also scale up for large data sets if massive parallelism with minimal dependence exists. MRlite can significantly improve the flexibility and parallel execution performance for a number of typical programs. Our evaluation shows that MRlite is one order of magnitude faster than Hadoop on problems that MapReduce has difficulty in handling.**

***Keywords*-Distributed computing; Parallel architectures**

I. Introduction

MapReduce [1] is arguably the most successful parallelization framework used in datacenters comprising commodity computers [2]. The open-source variant of MapReduce, Hadoop [3], has seen active development activities and increasing adoption. Many cloud computing services provide MapReduce functions [4], and the research community uses MapReduce and Hadoop to solve data-intensive problems in bioinformatics, computational finance, chemistry, and environmental science [5][6][7][8].

On the other hand, the MapReduce model has its discontents. DeWitt et al. argues that MapReduce is much less sophisticated or efficient than parallel database query systems [9]. It is pointed out that the MapReduce model imposes too strong assumptions on the dependence relation among data, and the correctness often depends on the commutativity, associativity, and other properties of the operations [10]. Others point out that the unreliable communication model and retry mechanisms are far from being satisfactory, and the master node can easily become a single point of failure. The performance study on MapReduce-based algorithms exhibits mixed results [5]. Finally, the recently granted MapReduce patent raises question on the long-term viability of using this parallelization mechanism in open environments [11].

We argue, however, that the facts and observations above do not reveal the real limitation of the MapReduce technology–they are either not significant enough to taint the technical merits of MapReduce, or not technical issues at all. In addition to the capability of exploring massive parallelism, the MapReduce framework has its generality to make it attractive to a wide class of analytics applications. Otherwise, it would not have been used for many years as a fundamental piece of software in the Google architecture, which is a complex system solving many challenging problems [2].

The intrinsic limitation of MapReduce is, in fact, the "one-way scalability" of its design. The design allows a program to scale up to process very large data sets, but constrains a program's ability to process smaller data items. The one-way scalable design reflects assumptions made in a design context where large data sets were the dominating challenges, and affects several important design choices in the MapReduce framework.

While the one-way scalability was a legitimate choice when MapReduce was initially designed, it introduces severe difficulty in extending this programming framework to more general computation. It has become imperative to design a new parallelization framework that is not only scalable but also flexible and generally applicable as cloud computing evolves to cover more dynamic, interactive, and semantic-rich applications, such as multiple-user collaborative applications [12], scientific computing, development tools, and commercial applications [13].

In this paper, we use a specific case to probe the limitation of MapReduce, and design a new lightweight parallelization framework to mitigate the one-way scalability problem and improve the system performance. The probing case is a distributed compilation tool, and the new parallelization framework is called "MRlite", which can efficiently scale down to process moderate-size data. Our evaluations on MRlite show that it is more than 12 times faster than Hadoop in the distributed compiling workload.

The rest of this paper is organized as follows. Section II discusses related work. Section III describes the probing case and our evaluation on it. Section IV presents the design of

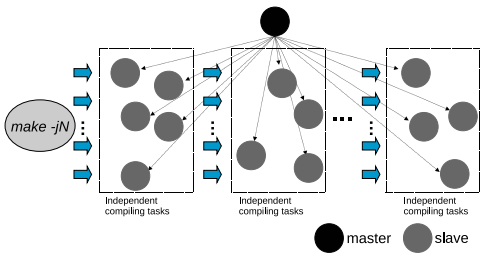

Figure 1. The architecture of mrcc

MRlite. Section V describes the prototype and the evaluation result of MRlite. We give a brieve conclusion in Section VI.

II. Related work

MapReduce is initially designed and implemented by Google for processing and generating large data sets [1]. Solutions to a wide class of real world problems can be expressed in this model. MRlite subsumes the parallel execution capability of MapReduce so that the problems handled by MapReduce can also be solved by MRlite. In addition, MRlite can handle workloads that MapReduce cnanot efficiently process.

The open-source variant of MapReduce, Hadoop [3], as well as its underlying data persistency layer, HDFS [14], which is loosely modelled after GFS [15], has seen active development and increasing adoption. Hadoop also has the "one-way scalability" limitation in its design. Different design choices are made in MRlite, which mitigate the "one-way scalability" problem.

Dryad is another distributed execution engine which allows an application to specify an arbitrary "communication DAG (directed acyclic graph)" [16]. Dryad also allows one vertex in the graph to consume multiple inputs and generate multiple outputs. DryadLINQ compiles the LINQ (a set of .NET constructs for queries) [17] programs into a distributed Dryad execution plan which can be executed directly on Dryad [10]. MRlite does not use the DAG based approach.

The limitation of MapReduce is also manifested in problems with large data sets. Chen et al. points out that it is tricky to achieve high performance for programs using Mapreduce, although implementing a MapReduce program is easy [18]. MRlite's programming interface and lightweight design help developers explore more potential parallelization opportunities in solving a wider range of problems.

III. A case study

To probe the limitation of the MapReduce framework, we design mrcc [19], a distributed compilation system, and examine its performance and overhead. MapReduce is not designed for the compilation workload which contains moderate-size data with complex dependency. We choose the compilation workload to probe the limitation of MapReduce. Meanwhile, a large class of applications share the features of compilation workload, such as variable-size data and dependency among them, and variable degree of parallelization at different algorithmic steps.

mrcc consists of one master node that controls the compilation job and many slave nodes that handle the compilation tasks as shown in Figure 1.

When one project is compiled on the master node, *make* builds the dependency tree for this project, and invokes multiple mrcc program instances to compile multiple source files in parallel. Hence, the parallelization are leveraged by *make* invoking multiple concurrent mrcc instances. Each mrcc instance runs one compilation task on a slave node. A slave node is one of the worker machines that receive map/reduce tasks from MapReduce master.

When conducting remote compilation on a slave node, mrcc preprocesses the source file, places a batch of preprocessed source files into a network file system used by the framework, then starts a compilation job on MapReduce. The map operation of this MapReduce job is done by a program called "mrcc-map". Running on the slave node, mrcc-map first retrieves the source file from the network file system, then calls the compiler locally to process the source file on the slave. After the compilation finishes, mrcc-map places the object file which is the result of the compilation back into the network file system. After the mrcc-map task is finished, mrcc on the master node retrieves the object file from the network file system and places it into the master node's local file system. After one batch of files are compiled, *make* continues to release more files to be compiled that depend on the completed ones.

A. Implementation

The mrcc compilation system consists of two core parts: the main program mrcc which runs on the master node and mrcc-map which runs on the slave nodes. mrcc is an open-source project under the GNU General Public License, version 2. The source files of mrcc can be downloaded from [19]. The work flow of the mrcc program is shown in Figure 2.

mrcc forks the preprocessor process after scanning the compiler arguments. The preprocessor inserts the header file(s) into the source file so that the remote nodes can assume a much simpler execution environment. mrcc then places the preprocessed file into a network file system and conducts remote compilation. When running mrcc on Hadoop, we use Hadoop Streaming [20] which can run MapReduce jobs with any executable or script to perform the map or reduce operation. mrcc submits the job to Hadoop and mrcc-map on the slave node is invoked by Hadoop to perform the map operation.

mrcc-map is implemented as a program residing in local directories of all the slave nodes because the mrcc-map program does not change during the process of compiling

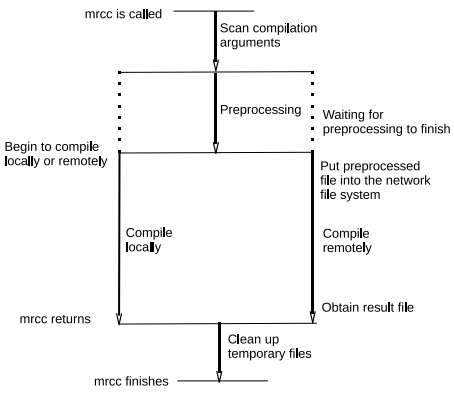

Figure 2. The work flow of mrcc

Table I
NODE CONFIGURATION

	CPU	Memory (GB)	Number
mrcc master	4	2	1
Slaves	2	2	10
Hadoop or MRlite master	2	2	1
NFS server	2	14	1

one project. This also makes it "easier" for the MapReduce framework to handle the compilation tasks, and, hence, the performance penalty we observe shall reflect more accurately the intrinsic limitations of the methodology.

mrcc-map first parses its arguments to obtain the source file name on the network file system and the compilation arguments. mrcc-map then retrieve the preprocessed file from the network file system. After that, mrcc-map calls the local *gcc* compiler and passes the compilation arguments to it. When *gcc* exits with a successful return value, mrcc-map places the object file into the network file system and returns immediately.

Upon the completion of all mrcc-map tasks, the Hadoop Streaming job returns. mrcc obtains the object files from the network file system and returns.

B. Performance of mrcc on Hadoop

We set up an experiment platform that consists of Xen virtual machines. We isolate these virtual machines by assigning each virtual machine except the mrcc master a physical CPU core which contains two CPUs. The configuration of the slave nodes are identical. Details of the node configuration are listed in Table I. The Hadoop master node and three slave nodes reside on one physical machine while the other seven slave nodes is on top of another physical machine. The mrcc master node is on top of the third physical machine. The physical machines are connected by a Netgear JG5516 1Gbps switch. The bandwidth between two virtual machines on top of one physical machine is also 1Gbps.

Table II
TIME FOR COMPILING PROJECTS USING GCC ON ONE NODE AND MRCC ON HADOOP

Project	Time for gcc	Time for mrcc/Hadoop
Linux	48m56.2s	150m44.4s
ImageMagick	5m12.1s	10m52.7s
Xen tools	2m7.6s	23m38.9s

We use mrcc to compile the Linux kernel 2.6.31.6, ImageMagick 6.6.3-8, and Xen tools 4.0.0 on Hadoop to examine the performance of Hadoop when it processes jobs in which complex dependencies exist between tasks while the input data files are also dynamically generated. The Hadoop version is 0.20.2 [3].

Table II shows the performance data. The compilation time using mrcc on 10 nodes is at least twice as long as that on one node (sequential compilation). Further investigation reveals that Hadoop takes more than thirty seconds to complete one compilation task. We also measures that Hadoop takes more than 20 seconds to finish one "null" job even though the job does not do any work. Storing or retrieving one file on the network file system takes at least 2 seconds while compiling one file on one node usually takes less than 2 seconds. While such overheads are acceptable in the special class of applications where relatively simple processing logic is applied to a large number of independent data, the prohibitive tasking and data transportation cost limits the applicability of MapReduce/Hadoop in more general workloads.

IV. DESIGN

The experimental results in Section III-B show that the current design and implementation of the MapReduce framework cannot provide the flexibility and efficiency required by programs with numerous parallelizable steps, instead of one massive parallelizable step. Hence, the current MapReduce framework does not work effectively for a large class of applications with not only sizable data but also non-trivial application logic. Representing the "common" case in scientific and business computing, such applications require the programming framework to efficiently handle variable-size data, support data dependence, and harness variable degrees of parallelization with controlled latency.

To overcome the limitation of MapReduce, we have designed and implemented MRlite, a lightweight parallelization framework that optimizes for not only massive parallelism, but also low latency to provide a more general and flexible parallel execution capability in cloud computing environments. The data in a complex computing system are often dynamically generated, thus introducing dependence among data. In fact, "data with dependence" shall be considered the common case in general applications. Such

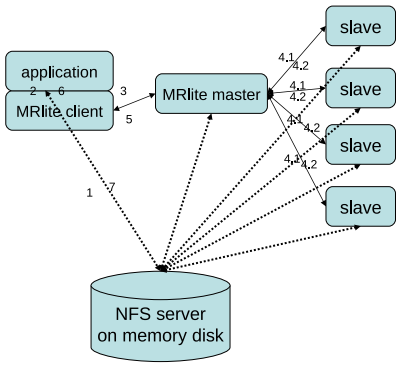

Figure 3. The architecture of MRlite

dependence naturally divides the processing into numerous steps to be taken in order, but each step may have sufficient parallelism to be exploited. The key is, consequently, to significantly reduce the overhead in data transportation and task management so that most of the parallelizable steps can invoke the parallelization mechanism, with the performance gain from parallel execution outweighing the latency induced by the overheads.

A. Architecture

The MRlite system consists of four parts as shown in Figure 3: the MRlite master, MRlite slaves, the NFS server in memory, and the MRlite client.

The MRlite master controls the parallel execution of tasks. It accepts jobs from the MRlite client, and distributes the tasks of the jobs to MRlite slaves. The MRlite slaves accept and execute the tasks from the MRlite master. The MRlite client is a library that can be linked to ther user application. The MRlite client accepts the application's parallelization requests, and submits the job to the MRlite master. MRlite includes an NFS server whose files are stored in one participating node's memory to profide a file system abstraction. The NFS file system is mounted on the MRlite master node, all the MRlite slave nodes, and the node on top of which the user application runs.

B. Latency optimization

The MRlite master cooperates with the MRlite client to minimize overhead and perform low-latency operations. In addition, the MRlite framework includes a timing control feature in its design as part of the low-latency execution mechanism. The application can estimate a timeout limit for the job and provide the timeout limit when it sends parallelization request for one job to the MRlite client. According to the timeout limit for the whole job, the MRlite master provides a suggested timeout value for the tasks to be executed on the slaves. In the current design, the timeout value is specified by the programmer. In the future work, we will extend this to a more flexible mechanism. After receiving the task command from the master, the slave tries its best to complete the task during the time slot specified by the timeout value. The timing enforcement also take care of reliability through retries. If one task times out, the master treats it as a failed one and may retry that task on another slave or just report the failure to the master. Similarly, a timed-out job may be treated as a failed one by the MRlite client, and the client may retry the job for a certain number of times or report the failure to the application according to that job's configuration. The application logic ultimately decides how to proceed when a job fails.

Besides the execution time enforcement, the MRlite master submits tasks to slaves without sophisticated queueing to maximize the possibility of finishing the job within the timeout limit. As there are dependences between jobs and among map and reduce tasks, the master submits the tasks as soon as the dependence is resolved.

The latency caused by the run-time overhead in each step may become critical when processing moderate data sets though it may not be a concern for processing a huge amount of data. Unlike the Hadoop design, MRlite uses a run-time daemon and thread pools to support the operations of the master and the slaves. This design reduces the cost of creating a process every time a job request or task request is issued. As the multi-thread and multi-core technology is widely available in modern computing platforms, we believe it is a pleasantly acceptable cost to dedicate one thread for each task on a slave and one thread for managing a job on the master.

Data transportation is an important aspect in distributed computing. In our lightweight parallelization framework, it is convenient to provide a distributed file system to store data, but we only store intermediate data files and the run-time data in the network file system. The design choice on the network file system implementation must balance the performance and usability. MRlite includes an NFS server, which runs on one participating node, to provide a file system abstraction, and mount the NFS file system on the MRlite master, all MRlite slaves, and the MRlite clients. The NFS server rides on a *tmpfs* file system [21], a virtual memory file system in Linux, so that the I/O speed of the NFS directory is as fast as operations in memory. This design choice reflects the observation that modern gigabit and 10 gigabit NICs provide comparable throughput bewteen networked computers to the I/O bandwidth between memory and hard drives. To maximize the I/O speed of the network file system, we do not duplicate the intermediate files as some distributed file systems do [15].

In the current design of MRlite, data persistency is supposed to be provided by a separate layer of data storage. The software based reliability through multiple-way replication is not included in MRlite since these are not the focus

of this work, and solutions that provide these features already exist [14][15]. Replication can potentially increase the serving bandwith of read operations, at the cost of first increasing the cost of writing operations. Both GFS and HDFS employs 3-way replication [14][15] to improve concurrency and reliability. In our experiments, it has not been observed that the lack of 3-way serving bandwidth limits the parallelization capability or the overall application-level performance when the network bandwidth is sufficient. Nevertheless, the MRlite architecture does not prohibit the addition of a replicated data storage, given that intermediate data files are still stored on and served from the low-latency network file system.

C. Programming interface

The MRlite client is designed as a library that can be compiled and linked to the user application. It provides a simple API so that the application developers can use the parallel computing capacity by simply calling a function. The developer can define the map program, the reduce program, how many map tasks and reduce tasks should be invoked, the input data directory/file, the output data directory/file and the time out value for the job. The MRlite client parses the application's parallelization requests, and submits the job to the MRlite master with the options specified in the API.

V. Prototype and evaluation

We have prototyped the MRlite parallelization service, and implemented mrcc on MRlite. In this section, we discuss the implementation details of MRlite, and report the evaluation results of mrcc on MRlite.

A. Implementation

The MRlite jobs are represented as sets of native Linux applications written in any programming language of choice. After each job is split into numerous tasks, each task is executed as a Linux program by one of the MRlite slaves.

At each parallelizable step, the MRlite framework dispatches a group of concurrent tasks, emulating the MapReduce model inspired by list primitives in Lisp. The tasks executed on MRlite slaves are defined as map and reduce tasks, with reduce tasks aggregating the intermediate results generated by the map tasks. It worths noting that we do not restrict map and reduce tasks to use key/value pairs. The MRlite slaves monitor the tasks' execution and report the execution's status and return values to the MRlite master.

There are 7 steps during the process of executing one job as shown in Figure 3:

1) The application places input data files to the input NFS directory.
2) The application submits the MapReduce job to the MRlite client.
3) MRlite client accepts the job, and submits it to the MRlite master.
4) The MRlite master submits tasks of the job to slaves (4.1), and waits for slaves to respond (4.2). Perform steps 4.1 and 4.2 for all tasks.
5) The MRlite master sends a success or fail message back to MRlite client.
6) MRlite client returns to the application with a return value that represents the result.
7) The application retrieves the output data files from the output NFS directory.

Table III
Comparison of speedups of mrcc on Hadoop and MRlite

	Speedup on Hadoop	Speedup on MRlite	MRlite vs. Hadoop
Linux	0.32	5.8	17.9
ImageMagick	0.48	6.2	13.0
Xen tools	0.09	2.0	22.0

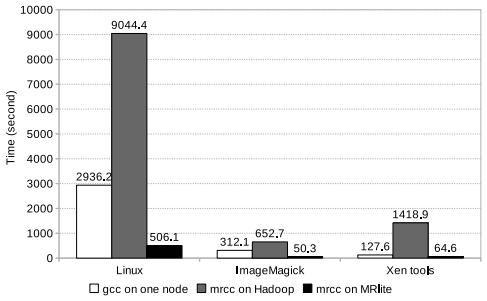

Figure 4. Execution time for compiling projects

B. Evaluation

Because MRlite is a general parallelization framework, we can port the mrcc program to MRlite, and compare the performance with mrcc on Hadoop using the workloads as described in Section III-B with complex dependencies existing among tasks and some input data files dynamically generated. This evaluation examine MRlite's ability to scale "down" to handle moderate-size data, mixed sequential and parallel work flow, and latency-aware tasks.

Most of the components in the mrcc/MRlite implementation are identical to those in the Hadoop based implementation except the parts invoking programming interface. The configurations of the nodes and the client nodes are listed in Table I. The MRlite platform has the same number and hardware configuration of slave nodes as Hadoop in Section III-B.

Figure 4 shows the evaluation results. The compilation of the three projects using mrcc on MRlite is much faster than compilation on one node, with a speedup of at least 2 and the best speedup reaches 6. Table III lists the speedup mrcc achieves on Hadoop and MRlite, and shows that the

average speedup of MRlite is more than 12 times better than that of Hadoop. This comparison shows that the MRlite is more effective than MapReduce for workloads with numerous computational steps where each step may have parallelization opportunities. From the result we also find that a project that is easier (a higher speedup) for mrcc on Hadoop to compile is easier for mrcc on MRlite. When compiling the "easier" project such as ImageMagick, mrcc on MRlite can achieve better speedup than mrcc on Hadoop.

The evaluation shows that MRlite is one order of magnitude faster than Hadoop on problems that MapReduce has difficulty in handling. The MRlite framework can still handle the massive parallelism with simple dependency as MapReduce does. MRlite shows that we can implement a flexible and efficient parallelization framework which programs can easily invoke to handle both large and small data sets.

VI. Conclusion

In this paper, we use the distributed compilation case to probe the limitation of MapReduce. The probing case shows that the overhead of Hadoop is too high for mrcc and the performance of mrcc on Hadoop is far from satisfactory.

We design MRlite, a new lightweight parallelization framework, to mitigate the one-way scalability problem and improve the system performance. The evaluation result shows that MRlite can efficiently process moderate-size data and handle data dependence. The MRlite framework can still handle the massive parallelism with simple dependency. The design significantly improves the parallel execution performance for general applications.

VII. Acknowledgment

This work is supported in part by HKUST research grants REC09/10.EG06 and SBI08/09.EG03.

References

[1] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters." in *the 6th Conference on Symposium on Operating Systems Design & Implementation*, vol. 6, San Francisco, CA, 2004, pp. 137–150.

[2] L. Barroso, J. Dean, and U. Hoelzle, "Web search for a planet: The Google cluster architecture," *IEEE Micro*, vol. 23, no. 2, pp. 22–28, 2003.

[3] "Apache Hadoop," *http://hadoop.apache.org/*, [last access: 9/2, 2010].

[4] "Amazon Elastic Compute Cloud – EC2," *http://aws.amazon.com/ec2/*, [last access: 9/2, 2010].

[5] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun, "Map-Reduce for machine learning on multicore," in *Proc. of NIPS'07*, 2007, pp. 281–288.

[6] M. C. Schatz, "CloudBurst: highly sensitive read mapping with MapReduce," *Bioinformatics*, vol. 25, pp. 1363–1369, 2009.

[7] J. Ekanayake, S. Pallickara, and G. Fox, "MapReduce for data intensive scientific analysis," in *Fourth IEEE International Conference on eScience*, 2008, pp. 277–284.

[8] Y. Chen, D. Pavlov, and J. F. Canny, "Large-scale behavioral targeting," in *KDD '09: Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2009, pp. 209–218.

[9] D. DeWitt and M. Stonebraker, "MapReduce: a major step backwards," *http://databasecolumn.vertica.com/database-innovation/mapreduce-a-major-step-backwards/*, [last access: 9/2, 2010].

[10] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey, "DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language," in *OSDI*, 2008, pp. 1–14.

[11] D. Jeffrey and G. Sanjay, "System and method for efficient large-scale data processing," U.S. Patent 7,650,331, 2010.

[12] "Google Docs," *http://docs.google.com*, [last access: 9/2, 2010].

[13] "Salesforce.com," *http://www.salesforce.com*, [last access: 9/2, 2010].

[14] "Hadoop Distributed File System," *http://hadoop.apache.org/hdfs/*, [last access: 9/2, 2010].

[15] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *Proc. of the 9th ACM Symposium on Operating Systems Principles (SOSP'03)*, 2003, pp. 29–43.

[16] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, 2007, pp. 59–72.

[17] D. Box and A. Hejlsberg, "LINQ: .NET language-integrated query," *http://msdn.microsoft.com/library/bb308959.aspx*, [last access: 9/2, 2010].

[18] S. Chen and S. W. Schlosser, "Map-Reduce meets wider varieties of applications," Intel Research Pittsburgh, Tech. Rep. IRP-TR-08-05, May 2008.

[19] Z. Ma, "mrcc," *http://www.cse.ust.hk/~zma/proj/mrcc.html*, [last access: 9/2, 2010].

[20] "Hadoop Streaming," *http://hadoop.apache.org/common/docs/r0.20.2/streaming.html*, [last access: 9/2, 2010].

[21] P. Snyder, "tmpfs: A virtual memory file system," in *In Proceedings of the Autumn 1990 European UNIX Users Group Conference*, 1990, pp. 241–248.