Int. J. Appl. Math. Comput. Sci., 2011, Vol. 21, No. 2, 275–284
DOI: 10.2478/v10006-011-0020-3

amcs

PERFORMANCE EVALUATION OF MAPREDUCE USING FULL VIRTUALISATION ON A DEPARTMENTAL CLOUD

HORACIO GONZÁLEZ-VÉLEZ [*,**], MARYAM KONTAGORA [*]

[*] School of Computing
Robert Gordon University, St Andrew Street, Aberdeen AB25 1HG, UK
e-mail: h.gonzalez-velez@rgu.ac.uk

[**] IDEAS Research Institute
Robert Gordon University, St Andrew Street, Aberdeen AB25 1HG, UK

This work analyses the performance of Hadoop, an implementation of the MapReduce programming model for distributed parallel computing, executing on a virtualisation environment comprised of $1 + 16$ nodes running the VMWare workstation software. A set of experiments using the standard Hadoop benchmarks has been designed in order to determine whether or not significant reductions in the execution time of computations are experienced when using Hadoop on this virtualisation platform on a departmental cloud. Our findings indicate that a significant decrease in computing times is observed under these conditions. They also highlight how overheads and virtualisation in a distributed environment hinder the possibility of achieving the maximum (peak) performance.

Keywords: MapReduce, server virtualization, cloud computing, algorithmic skeletons, structured parallelism, parallel computing.

1. Introduction

Widely considered a logical successor of the Internet, cloud computing entails the exchange of computer and data resources across global networks. It constitutes a new value-added paradigm for network computing, where higher efficiency, massive scalability, and speed rely on effective software development (Armbrust *et al.*, 2010). Major Internet powerhouses such as Amazon, Google, Microsoft, IBM, and Yahoo have embarked upon substantial business endeavours centred on this new paradigm.

A cloud typically comprises inter-connected, virtualised computers coupled with security and a programming model. From an application developer's perspective, the programming model and its performance are undoubtedly the main criteria to select a cloud environment.

Widely considered one of the most popular cloud programming models (Buyya *et al.*, 2009), MapReduce has been introduced by Google for efficient deployment of computationally intensive parallel algorithms. It is a distributed programming model and framework used to compute problems that can be parallelised by mapping a function over a given dataset and then combining the results (Dean and Ghemawat, 2004; 2008). As a framework, it is used to implement MapReduce jobs which encapsulate the features of the model while hiding the complexities inherent in cloud computing.

Virtualisation is a software technology that allows a machine to run different operating systems on the same physical computer. Because virtualisation involves running an increased number of processes on the host machine, there is always some degree of performance tradeoff when using this technique. Today, pure software-based virtualisation is achieved by using one of two methods:

- full virtualisation, or
- paravirtualisation.

In full virtualisation, a guest operating system runs as a process on a host system. It simulates a complete hardware environment and the guest operating system requires no changes. The virtual machine created runs in isolation, unaware of the underlying layer of system software. Full virtualisation relies on binary translation by the host machine in order to catch and execute operations initiated by the guest operating system. Although binary

translation can incur a large performance overhead, making the operations of the virtual machine slower (Buzen and Gagliardi, 1973; Whitaker *et al*., 2002), full virtualisation does bring practical advantages in terms of software availability in the guest operating system.

In paravirtualisation, a virtual machine runs a modified version of an operating system. It requires changes to the original operating system to create a special API through which it can run processes directly on the physical hardware of the host machine. The logic behind using a modified API is to reduce performance overheads, as the virtual machine can initiate the execution of tasks that will run on the physical hardware without any need for translation. For this reason, paravirtualised machines incur lower performance overheads than fully virtualised machines.

1.1. Contribution. The contribution of this article lies in the methodological approach to run benchmarks on an off-the-shelf hardware configuration with generic, widely used host and guest operating systems (Windows and Linux, respectively) employing a commercial full virtualisation environment in a departmental cloud.

For the purpose of this paper, a set of experiments based on Hadoop's RandomWriter and Sort algorithms has been designed to determine whether or not significant reductions in the execution time of computations have been observed.

Our evaluation has employed Hadoop's MapReduce implementation on Virtual Machines (VMs) running the VMWare Workstation software and Linux sitting on top of a common off-the-shelf Intel hardware and the Windows system software platform. The use of VMs is particularly relevant to clouds as several can be started and stopped on-demand on a single physical machine to flexibly satisfy distinct service requests. By providing isolated environments, every VM can feature a distinct environment isolated from the rest but within a single physical machine. To this end, our experimental setup can then be considered a fully virtualised departmental cloud.

This paper extends our initial work (Kontagora and González-Vélez, 2010) by incorporating detailed system information from the nodes in our departmental cloud, which has important implications for task scheduling in clouds. Our findings indicate a steep decrease in the scalability, ergo an impact on execution times, and we also highlight how overheads and virtualisation in a distributed environment may hinder the possibility of gaining the maximum achievable speedup. In particular, our results are consistent with overheads related to Linux-on-Windows virtualisation and input/output operations.

2. Related work

The MapReduce programming model provides scalable support to a distributed functional mapping and its associated reduction, which is continually employed in numerous applications with large sets of data on hundreds of nodes. This model is thence composed of two higher-order functions (map and reduce) and, arguably, can be considered a skeletal programming model.

Initially introduced by Cole (1989), algorithmic skeletons have been construed as specialised higher-order functions from which one must be selected as the outermost purpose in a parallel program. González-Vélez and Leyton (2010) present a recent survey of the field.

To this end, the *map* and *reduce* algorithmic skeletons have been defined and used extensively. On the one hand, *map* is probably the quintessential data parallelism skeleton and its origins are closely related to functional languages. The semantics behind *map* specify that a function or a sub-skeleton can be applied simultaneously to all elements of a list to achieve parallelism. The data parallelism occurs because a single data element can be split into multiple data, then the sub-skeleton is executed on each data element, and finally the results are united again into a single result. The *map* skeleton can be conceived as single instruction, multiple data parallelism. On the other hand, *reduce* is employed to compute prefix operations in a list by traversing the list from left to right and then applying a function to each pair of elements, typically summation. As opposed to *map*, *reduce* maintains aggregated partial results.

In addition to copious functional programming implementations, the *map* and *reduce* skeletons have been deployed as application programming interfaces in procedural languages. By using C procedure calls within a pre-initialised MPI environment, the Skeleton-based Integrated Environment (SkIE) (Bacci *et al*., 1999), and the Edinburgh Skeleton Library (eSkel) (Cole, 2004), deliver data- and task-parallel skeletal APIs. Moreover, the master/worker paradigm has been analysed from the skeletal perspective (Danelutto, 2004; González-Vélez, 2006; Kuchen and Striegnitz, 2005; González-Vélez and Cole, 2010b).

Of particular importance is Buono *et al*.'s (2010) research, where clear correspondence between the skeleton paradigm and the post-Google MapReduce model is established. The authors describe the MareMare environment, which includes not only the traditional map and reduce skeletons but also fault resiliency in worker nodes and reconfiguration capabilities based on dynamic performance requirements.

From the virtualisation perspective, different research groups have explored the application of highly-tuned para-virtualisation platforms, namely, Xen, for high-performance computing codes (Nagarajan *et al*., 2007; Youseff *et al*., 2006). In terms of the frameworks for MapReduce computation frameworks, Zaharia *et al*. (2008) analysed the impact of using alternative scheduling policies on major Hadoop configurations, while Sand-

holm and Lai (2009) experimented with priority-based scheduling mechanisms for different MapReduce-like applications.

Particularly relevant is the work by Ibrahim *et al*. (2009), which employs Xen with Linux in order to provide VMs for a Hadoop computing environment. Their experiments also feature the Sort benchmark for up to 2 GB, but they have failed to ascertain the individual load on a per node basis, a key contribution in this paper.

Our work is therefore aimed at providing a performance evaluation perspective of a simplistic off-the-shelf departmental cloud configuration, which is arguably more prevalent in many scientific and commercial organisations with MapReduce computational requirements.

3. MapReduce model

The MapReduce programming model systematically combines a series of *map* and *reduce* higher-order functions in order to abstract complex parallel computations.

In functional programming, a *map* functor applies a given function f element-wise to a list of n elements $\vec{x}$ and returns a list of results,

$$map\ f\ \langle x_1, \ldots, x_n \rangle = \langle f(x_1), \ldots, f(x_n) \rangle \qquad (1)$$

while a *reduce* combines the n elements of a list $\vec{x}$ using a certain associative operation $\circ$,

$$reduce\ \circ\ \langle x_1, \ldots, x_n \rangle = x_1 \circ x_2 \circ \cdots \circ x_n. \qquad (2)$$

Figure 1 demonstrates pragmatic implementation of the map-reduce pairing as a distributed computational MapReduce model (Dean and Ghemawat, 2004), which consists of

1. a series of *map* processes that output key/value pairs after a number of iterations over the partitioned input dataset, and
2. a series of *reduce* processes that merge pairs having the same key to produce the final output.

It is noted that the n calculations of f over $\vec{x}$ are independent and can therefore be applied to all elements in the list concurrently. The $\circ$ operation is fully associative (and typically commutative) and maintains aggregated partial results allowing disjunct out-of-order execution.

Thence, MapReduce is highly suitable for the solution of embarrassingly parallel problems, and its computation resembles a divisible load model (Robertazzi, 2003), where the input datasets are independently *split* between several computing nodes (*workers*) by a *master* node and then processed independently of the output and the sequence of node execution and completion. The scheduling of such divisible loads in heterogeneous systems remains an active area in computer science (Beaumont *et al*., 2005), where different innovative heuristics continue to be proposed (González-Vélez and Cole, 2010a; Mesghouni *et al*., 2004).

In summary, MapReduce is especially suitable for the implementation of parallel solutions over a wide area network due to negligible inter-node communication. MapReduce makes automatic parallelisation of data-intensive computations possible by allowing computations to be expressed in terms of their functionality alone while taking care of the complex features of the parallel infrastructure within its implementation.

MapReduce implementations. There are currently three popular implementations of the MapReduce programming model:

1. Google MapReduce (Dean and Ghemawat, 2008),
2. Apache Hadoop (The Apache Software Foundation, 2008), and
3. Stanford Phoenix (Ranger *et al*., 2007).

Table 1 shows a comparative analysis of their features. Additionally, there are a few emerging open-source implementations such as Skynet, a Ruby-based (Pisoni, 2007) version, and Disco, developed in Erlang by Nokia (Nokia Research Center, 2009).

As our evaluation intends to be as inclusive as possible and geared towards clusters, we have focused on two implementations: Google and Hadoop. Phoenix is not a viable option in this precise instance as it is intended to work on symmetric multiprocessing systems, where access to memory is shared.

We have selected Hadoop as we believe there are three advantages of Hadoop over Google, namely,

1. *Failure resiliency*: The ability to withstand failures in the master node is of particular importance in highly-demanding cloud computing environments, where changing conditions require the different system components to dynamically adapt to allow the calculation to continue in spite of various interconnection conditions.
2. *Open-source licensing*: Access to source code under an open-source licensing is appealing as this work has been produced in an academic environment and results are expected to be freely available to other individuals to advance knowledge.
3. *Java*: Hadoop and its file system are written in Java and our programming background includes proven expertise in the Java platform.

Hadoop. A scalable distributed computing platform, Hadoop includes a file system called the Hadoop Distributed File System (HDFS) to store large datasets as well

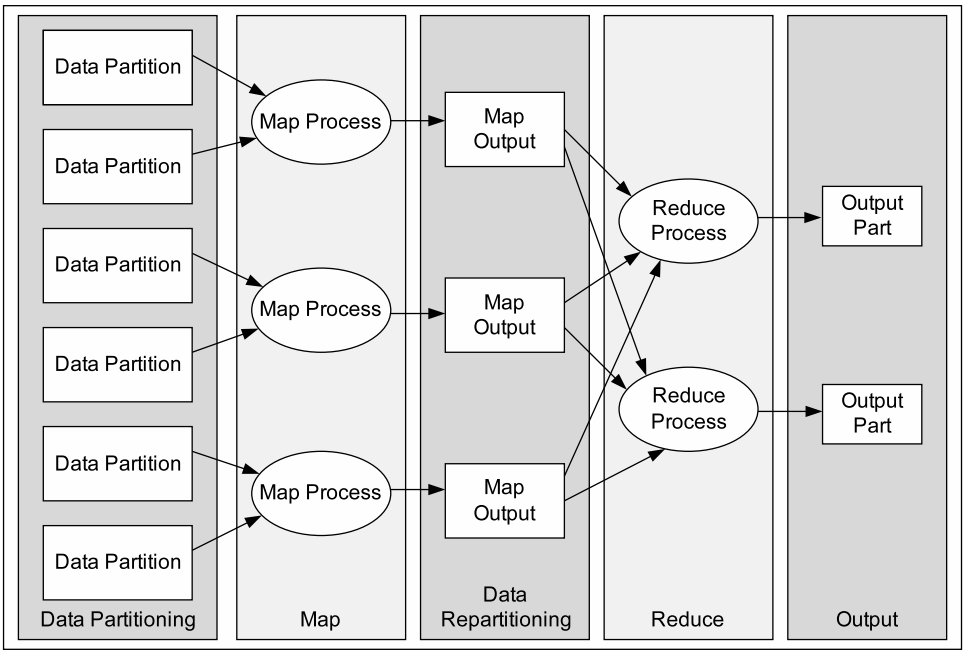


Fig. 1. Computing stages of the MapReduce model.

Table 1. Comparison of three MapReduce implementations: Google, Phoenix and Hadoop.

	Google	Phoenix	Hadoop
Hardware platform	Tailored towards cluster-based environments	Multi-core and multiprocessor systems	Tailored towards cluster-based environments
Worker failure handling	Proactive monitoring of computation time of nodes. Tasks are re-executed if there are delays or timeouts	Tasks are re-executed if there are delays or timeouts	Proactive monitoring of computation time of nodes. Tasks are re-executed if there are delays or timeouts
Master failure handling	Not implemented. Restart the job	Not implemented. Restart the job	Checkpointing of the master node
File system	Google file system	None. Uses pointers to shared memory locations	Hadoop distributed file system
Available APIs	Python and Java	C and C++. Can be extended to similar languages like Java and C#	Java, Python, Streaming API, C++ API—via sockets. Also extensible
Combinator functions	Yes	No	Yes
Licensing	Proprietary	Open source	Open source

as a Java MapReduce implementation to process the data. Hadoop executes map/reduce jobs by distributing the jobs across computing nodes in a cluster. The processing of jobs across a cluster of regular personal computers creates a high-performance computer that provides increased performance, while the HDFS stores large files across nodes in a cluster. Hadoop employs block-based file replication to underpin its fault tolerance mechanisms.

Based on master/worker architecture, the nodes in Hadoop are classified as

- *namenodes*, and
- *datanodes*.

A complete MapReduce job is divided into independent tasks that the namenode allocates to the datanodes, understood as block data requests from the master to the workers. By using the HDFS, Hadoop creates multiple copies of the data for each task, placing them on nodes in the cluster where they are processed. The namenode schedules and monitors jobs through the *JobTracker* pro-

Table 2. System configuration for the 17 nodes of our departmental cloud.

Processor	Intel(R) Core (TM) 2 Duo CPU E6750 2.66GHz
Memory	3.48GB
Host operating system	Microsoft Windows XP Professional Version 2002 SP3
Virtualisation software	VMWare Workstation 6.0
Guest operating system	Ubuntu Release 8.10 (Intrepid) Kernel Linux 2.6.2.27-7-generic
Number of virtual machines per node	1
MapReduce environment	Hadoop 0.20.0

cess, and the namenodes execute and track tasks through *Tasktracker*. To avoid a single point of failure, a namenode can be replicated.

Since many modelling tools require the manipulation of complex combinations of computation and data, Hadoop's forte is its scalability. According to the project site, Hadoop has been demonstrated on clusters with over 2000 nodes and can reliably process and store petabytes of data (The Apache Software Foundation, 2008). As complex computations are typically data intensive and hard to solve on a single computer, using the MapReduce computation model can speed up the computation because MapReduce computes large datasets by using all available resources concurrently.

Under ideal conditions, it is expected that MapReduce scales linearly, i.e., the completion time should decrease linearly with the increase of resources (nodes) used for computation. Gains in performance can permit the computation of more complex models and, consequently, the exploration of bigger solution spaces and the examination of more data in less time.

Yet, virtualisation promotes efficient utilisation of idle computing resources, providing the flexibility required to run different software setups on existing resources, but the conundrum is the adequate balance between the practicality of full virtualisation and the performance gains of para-virtualisation (VMware, 2007).

Nonetheless, scant research has been conducted on systematic evaluation of the MapReduce model under fully virtualised environments, whereby the impact of the staged application of idle resources and associated overheads is measured.

4. Implementation

Our departmental cloud has been deployed using 17 identical computers (nodes) running the VMWare Workstation with one virtual machine per node, executing Ubuntu and Windows XP as guest and host operating systems, respectively. One node was used as master while the remaining 16 as workers and, subsequently, arranged as five distinct configurations: $1+1,\ 1+2,\ 1+4,\ 1+8$, and $1+16$.

In the remainder of the discussion we will refer to these configurations omitting the master node, i.e., 1, 2, 4, 8, and 16 nodes. Table 2 shows the system configuration for each node.

Table 3. Characteristics of the datasets generated by RandomWriter using 16 nodes.

Dataset no.	Map size (MB)	Maps per node	Dataset size (GB)
D1	8	8	1
D2	16	8	2
D3	32	8	4

Our evaluation has employed the *RandomWriter* and *Sort* algorithms, freely available as part of the Hadoop distribution.

RandomWriter writes random data to the HDFS using maps, where each *map* operation writes random byte keys and values to the HDFS sequence file.

Sort reads data in maps, and then combines sorts and stores them through reduces. It is modelled after the Terabyte sort (Anon, 1998), a disc sort of 1-million records where the input is 1-million 100-byte records stored in a sequential disc file. While the source and target files are sequential, the partitioning function for this benchmark has built-in knowledge of the distribution of keys, adding intrinsic MapReduce parallelism (Dean and Ghemawat, 2008).

Since RandomWriter has only maps, we have used it as the dataset generator and Sort as our primary benchmark.

Table 3 gives details of how three datasets of the algorithm generated data for three sizes, 1, 2 and 4 Gigabytes, using 16 nodes. The replication factor—how many copies of a data block must be available in the cluster at a time—was set at 2 (its default value is 1) to facilitate the decommissioning of nodes. The overall empirical evaluation testbed is shown in Fig. 2.

Our performance evaluation has employed five different configurations corresponding to 1, 2, 4, 8, and 16 nodes. In all, 75 experiment runs have been carried out, i.e., 25 for each dataset size (1 GB, 2 GB and 4 GB). For each data size, the Sort map/reduce task has been executed five times for each of the five different cluster sizes.

Two identical master node virtual images were used to perform the experiments alternately. We have also alternated between machines to ensure that cached processes were not reducing memory access time and, thereby, speeding up the execution of tasks in consecutive runs.

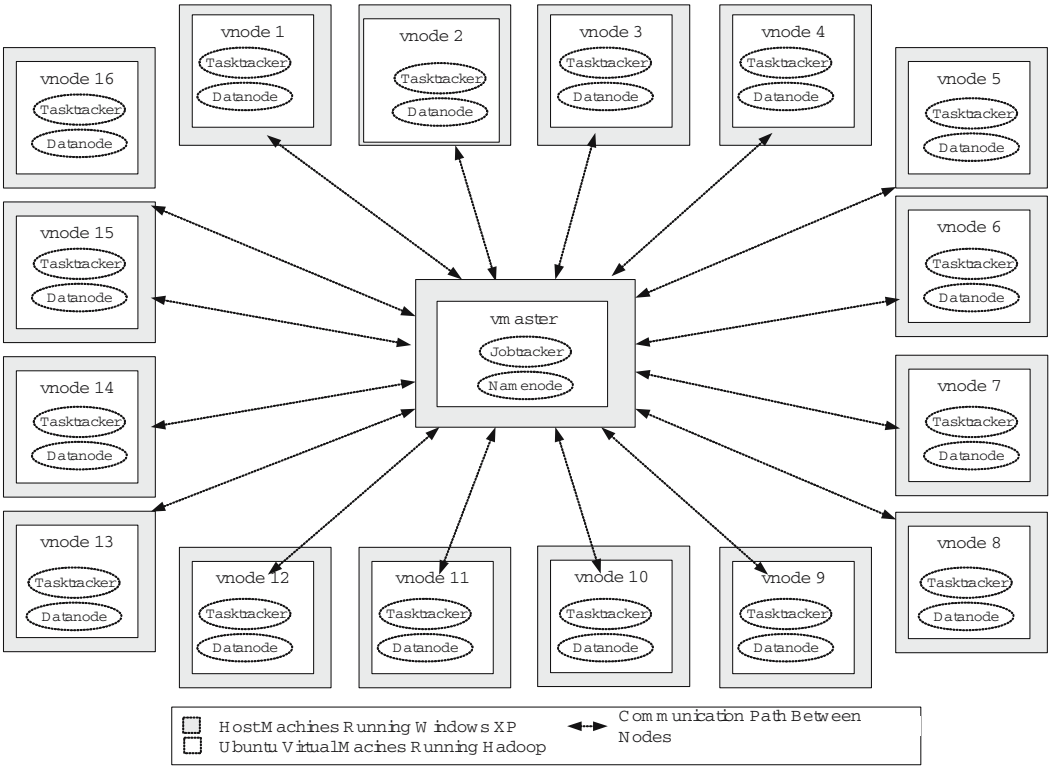


Fig. 2. Departmental cloud testbed employed in our empirical evaluation. This cloud is composed of 17 identical virtualised nodes.

Note that we have repeated each experiment five times and fixed the testbed conditions in order to ensure the low variability and reproducibility of our results.

After every five successful runs of the Sort algorithm for each data size and a number of nodes (starting at 16), half of those nodes decommissioned—by editing a configuration file in Hadoop that lists hosts to be excluded from the cluster, while preserving the data in the HDFS for use with the remaining nodes. This was done progressively until the tasks involving only one node were completed; then the data size was changed and the process repeated. After the completion of each task, the total execution time was logged by Hadoop and retrieved via its JobTracker web interface.

During each run of the primary benchmark, the CPU activity of each node was concurrently captured in 6-second intervals for each node, using the 1-minute readings from the Linux `top` command. This top load figure is particularly useful as it provides a consistent dynamic view, which is normalised through an exponential function in order to diminish the impact of transient, short-lived load peaks.

In order to increase the statistical significance of our results, we have randomly rotated the namenodes and datanodes between execution to avoid any cache memory effects. Additionally, we have calculated the coefficient of variation CV for the execution times t, based on the standard deviation σ and the mean μ,

$$CV = \frac{\sigma}{\mu}, \tag{3}$$

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(t_i - \mu)^2}, \tag{4}$$

$$\mu = \frac{1}{N}\sum_{i=1}^{N} t_i. \tag{5}$$

5. Results

Table 4 reports the execution times in seconds and their associated CV expressed as a percentage. A close examination of the results in the table yields the following two key observations:

- the time to complete each experimental run decreased for a given data size when the number of nodes increased;
- the load distribution across nodes for each experimental set is largely unfair, despite the fact that the independent tasks are of similar complexity.

Moreover, the overall efficiency also decreased with the node increments. We have discovered that datanodes

Table 4. Results of the primary benchmark (Hadoop's Sort) execution for 1, 2, 4, and 16 nodes with datasets of 1 GB, 2 GB, and 4 GB.

Dataset size	1		2		4		8		16	
	Time	CV	Time	CV	Time	CV	Time	CV	Time	CV
1GB	434.6s	2.1%	287.4s	3.1%	167.4s	3.0%	95.6s	4.2%	58.6s	6.8%
2GB	680.0s	0.8%	415.6s	4.4%	254.0s	5.1%	137.4s	3.1%	83.6s	4.8%
4GB	1232.0s	2.2%	752.4s	1.2%	412.0s	3.1%	314.8s	5.4%	181.6s	2.6%

were doing substantially more processing than the namenode, typically a difference of an order of magnitude. The comparison of the actual scale-up and the ideal scale-up is depicted in Fig. 3.

While the empirical observations above indicate that the system scalability has consistently increased (this is evidenced by the faster times recorded when nodes were added), the low efficiency figures indicate that there were three factors existent in the system that impeded the attainment of a linear scale-up of computation times.

Firstly, despite the fact that the (worker) nodes had been identically configured and placed, the master node seemed more likely to schedule tasks to certain nodes, especially in scenarios where tasks needed to be reassigned. This indicates that some nodes were more available or easier to reach by the master node. This explanation is also supported by the average number of tasks killed during experimentation, which depicts the number of tasks reassigned to other nodes as a result of delayed responses from the nodes to which they were assigned. One of the reasons why such a scenario would occur is due to slow network communication or failures. Copying and sending of data blocks from one node to another is also done via the network, so network issues could also have hampered this process and slowed down the whole system. As depicted in Fig. 4, it is also relevant to note that the master node is mostly idle, with a system load well beyond the average

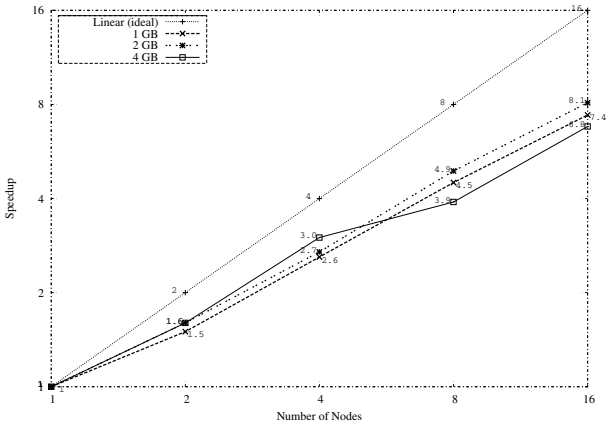


Fig. 3. Speedup curves for 1 GB, 2 GB, and 4 GB dataset sizes.

even for the 16 node configuration. This reinforces the need for improved scheduling algorithms in Hadoop.

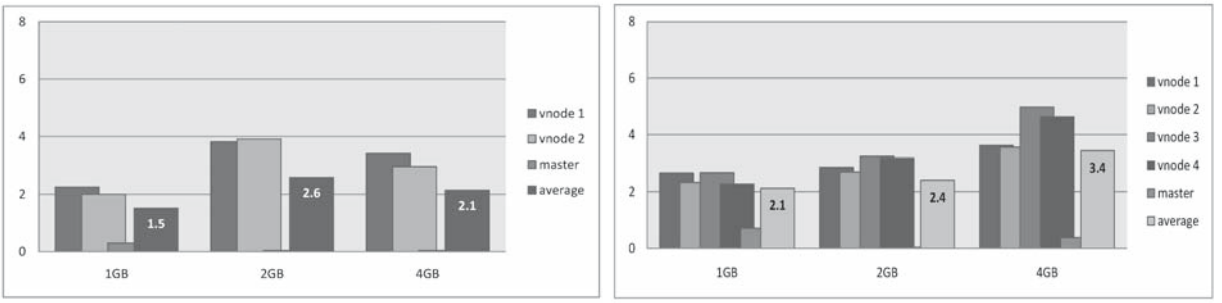
Secondly, during job executions, the data had to be migrated between nodes prior to the execution of a map or a reduce. This was almost always the case for higher configuration clusters used in these experiments as the replication factor was set at 2. The reading and writing of data to and from the HDFS and to and from memory most likely created overheads. This is evidenced by the fact that the node with the smallest CPU load is the master node, which does not run any data nodes.

Finally, in terms of virtualisation, the experiments were set up such that Hadoop was running on Ubuntu virtual machines hosted on machines with Windows XP installed with the VMWare Workstation software. Operations of full virtualisation software, such as VMWare, affect the performance of the system because they run as processes on the host machine. All the processes initiated in the Ubuntu environment by Hadoop needed to be translated onto the physical machine and executed before feedback could be sent back to the virtual machine. This included processes that control network communication, input/output, memory and disk space allocation. On a non-virtual system, these already constitute overheads so the overall effect in our testbed was two-fold when using a virtual machine.

6. Conclusions

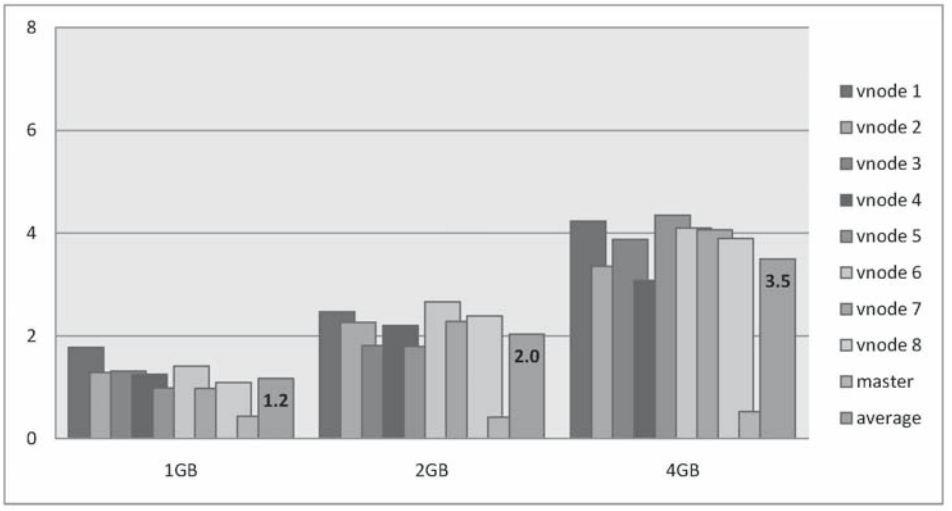
Hadoop has been deployed and tested with positive results that indicate that speedup of computations is possible under full virtualisation conditions. Even with the reduced efficiency recorded, there are increases in computation speed and this bodes well for the Hadoop implementation employed.

Nevertheless, an investigation into performance bottlenecks in Hadoop's implementation has been carried out and the results of the experiments provide a valuable insight into how Hadoop runs in a truly heterogeneous environment. Because Hadoop is written in Java, it can run on any machine that has the Java runtime environment installed, ergo Hadoop does not interact directly with machine hardware as this is handled by the Java interpreter. In essence, this means that if two machines are performing identically, then to Hadoop they will seem to be of the

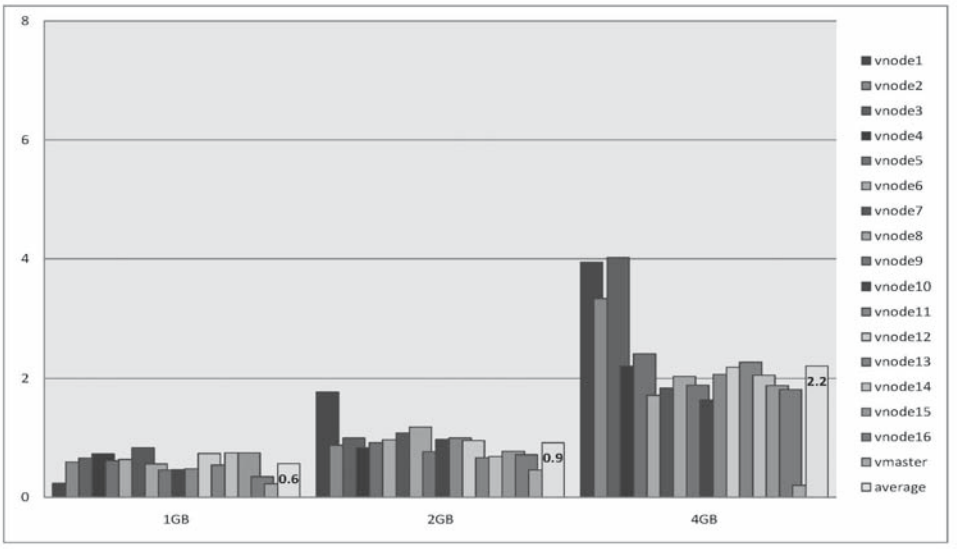


(a)

(b)



(c)



(d)

Fig. 4. System load, as measured by the Linux uptime command, for 2(a), 4(b), 8(c), and 16(d) nodes. Note that the master node load, indicated by the penultimate solid bar, in each configuration is well below the average, illustrated by the last hatched bar.

same configuration. This also means that factors external to the physical machine, e.g., network communication rates, affect how Hadoop perceives the performance of a machine.

The aforementioned observations suggest that when Hadoop is scheduling tasks, the nodes with the better performance—those that return results faster for whatever reason—seem to have better processing power and are therefore assigned new tasks earlier than others, regardless of their actual capabilities or configuration. This scenario also highlights the scalability and fault tolerance of Hadoop. A demonstration of a possible use of idle processing resources via virtualisation in order to run another completely different and separate application on the same physical machine has been one of the fringe benefits of our implementation. It also opens up the possibility of further experimentation with alternative schedulers.

We believe that the data sizes used for the experiments are somehow too small to completely mitigate the effect of overheads. From the results, we can conclude that the computational sections of the benchmarks have not been large enough to take care of the overhead (i.e., the communication to computation ratio is high) and therefore the evaluation runs cannot produce the linear speedup we had hoped to achieve. Future work may well include a comprehensive set of benchmarks deployed on a very large configuration (hundreds of nodes) using paravirtualisation and full virtualisation platforms, e.g., Xen versus VMWare.

Finally, it is also important to emphasise the impact of our results in terms of the Terabyte sort benchmark, which is widely considered in fact a standard for data sorting. A common scenario in the Internet, data sorting enables advertisements on social networks, custom recommendations on online shopping sites and manipulation of search engine results.

In conclusion, the majority of the aims of this work were achieved. An implementation of MapReduce was used to evidence increases in speedup and the possibility of heterogeneous configurations was put forward without any ad hoc hardware and software configuration or investment.

Acknowledgment

An earlier version of this paper was presented at the 2010 International Conference on *Complex, Intelligent and Software Intensive Systems*, sponsored by the IEEE, held in Cracow (Poland) in 2010 (and published in its proceedings).

Special thanks are owed to Iain Bell for his help in setting up our experimental cloud computing environment. This research has been supported through a *strategic funding grant* from the IDEAS Research Institute.

References

Anon, E.A. (1998). A measure of transaction processing power, *in* M. Stonebraker and J.M. Hellerstein (Eds.), *Readings in Database Systems*, 3rd Edn., Morgan Kaufmann, San Francisco, CA, pp. 609–621.

Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R.H., Konwinski, A., Lee, G., Patterson, D.A., Rabkin, A., Stoica, I. and Zaharia, M. (2010). A view of cloud computing, *Communications of the ACM* **53**(4): 50–58.

Bacci, B., Danelutto, M., Pelagatti, S. and Vanneschi, M. (1999). SkIE: A heterogeneous environment for HPC applications, *Parallel Computing* **25**(13): 1827–1852.

Beaumont, O., Casanova, H., Legrand, A., Robert, Y. and Yang, Y. (2005). Scheduling divisible loads on star and tree networks: Results and open problems, *IEEE Transactions on Parallel and Distributed Systems* **16**(3): 207–218.

Buono, D., Danelutto, M. and Lametti, S. (2010). Map, reduce and MapReduce, the skeleton way, *Procedia Computer Science* **1**(1): 2089–2097.

Buyya, R., Yeo, C.S., Venugopal, S., Broberg, J. and Brandic, I. (2009). Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility, *Future Generation Computer Systems—The International Journal of Grid Computing: Theory Methods and Applications* **25**(6): 599–616.

Buzen, J.P. and Gagliardi, U.O. (1973). The evolution of virtual machine architecture, *Proceedings of the National Computer Conference and Exposition, AFIPS '73* , ACM, New York, NY, pp. 291–299.

Cole, M. (1989). *Algorithmic Skeletons: Structured Management of Parallel Computation*, Pitman/MIT Press, London.

Cole, M. (2004). Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming, *Parallel Computing* **30**(3): 389–406.

Danelutto, M. (2004). Adaptive task farm implementation strategies, *12th Euromicro Workshop on Parallel, Distributed and Network-Based Processing, PDP 2004*, IEEE, La Coruña, pp. 416–423.

Dean, J. and Ghemawat, S. (2004). MapReduce: Simplified data processing on large clusters, *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation OSDI'04*, Vol. 6, USENIX, San Francisco, CA, pp. 137–150.

Dean, J. and Ghemawat, S. (2008). MapReduce: Simplified data processing on large clusters, *Communications of the ACM* **51**(1): 107–113.

González-Vélez, H. (2006). Self-adaptive skeletal task farm for computational grids, *Parallel Computing* **32**(7–8): 479–490.

González-Vélez, H. and Cole, M. (2010a). Adaptive statistical scheduling of divisible workloads in heterogeneous systems, *Journal of Scheduling* **13**(4): 427–441.

González-Vélez, H. and Cole, M. (2010b). Adaptive structured parallelism for distributed heterogeneous architectures: A methodological approach with pipelines and

farms, *Concurrency and Computation: Practice and Experience* **22**(15): 2073–2094.

González-Vélez, H. and Leyton, M. (2010). A survey of algorithmic skeleton frameworks: High-level structured parallel programming enablers, *Software: Practice and Experience* **40**(12): 1135–1160.

Ibrahim, S., Jin, H., Lu, L., Qi, L., Wu, S. and Shi, X. (2009). Evaluating MapReduce on virtual machines: The Hadoop case, *in* M. Jaatun, G. Zhao, and C. Rong (Eds.) *CloudCom 2009*, Lecture Notes in Computer Science, Vol. 5931, Springer-Verlag, Berlin/Heidelberg, pp. 519–528.

Kontagora, M. and González-Vélez, H. (2010). Benchmarking a MapReduce environment on a full virtualisation platform, *in* L. Barolli, F. Xhafa, S. Vitabile and H.-H. Hsu (Eds.), *CISIS 2010, The Fourth International Conference on Complex, Intelligent and Software Intensive Systems, Krakow, Poland, 15-18 February 2010*, IEEE Computer Society, Washington, DC, pp. 433–438.

Kuchen, H. and Striegnitz, J. (2005). Features from functional programming for a C++ skeleton library, *Concurrency and Computation: Practice and Experience* **17**(7–8): 739–756.

Mesghouni, K., Hammadi, S. and Borne, P. (2004). Evolutionary algorithms for job-shop scheduling, *International Journal of Applied Mathematics and Computer Science* **14**(1): 91–103.

Nagarajan, A.B., Mueller, F., Engelmann, C. and Scott, S.L. (2007). Proactive fault tolerance for HPC with Xen virtualization, *in* B. J. Smith (Ed.), *Proceedings of the 21th Annual International Conference on Supercomputing, ICS 2007, Seattle, Washington, USA, June 17–21, 2007*, ACM, New York, NY, pp. 23–32.

Nokia Research Center (2009). Disco, *Manual version 0.2.3*, Nokia Research Center, `discoproject.org`.

Pisoni, A. (2007). Skynet, *Manual version 0.9.3*, Geni.com, `skynet.rubyforge.org`.

Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G. and Kozyrakis, C. (2007). Evaluating MapReduce for multi-core and multiprocessor systems, *13th International Conference on High-Performance Computer Architecture (HPCA-13 2007), Phoenix, AZ, USA*, pp. 13–24.

Robertazzi, T.G. (2003). Ten reasons to use divisible load theory, *Computer* **36**(5): 63–68.

Sandholm, T. and Lai, K. (2009). MapReduce optimization using regulated dynamic prioritization, *in* J.R. Douceur, A.G. Greenberg, T. Bonald, J. Nieh (Eds.), *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems, SIGMETRICS/Performance 2009, Seattle, WA, USA, June 15–19, 2009*, ACM, New York, NY, pp. 299–310.

The Apache Software Foundation (2008). Hadoop MapReduce tutorial, *Manual version 0.15*, Hadoop Project, `hadoop.apache.org`.

VMware (2007). Understanding full virtualization, paravirtualization, and hardware assist, *White Paper Revision: 20070911*, VMware, Inc., Palo Alto, CA.

Whitaker, A., Shaw, M. and Gribble, S.D. (2002). Scale and performance in the Denali isolation kernel, *ACM SIGOPS Operating Systems Review* **36**(SI): 195–209.

Youseff, L., Wolski, R., Gorda, B. and Krintz, C. (2006). Paravirtualization for HPC systems, *in* G. Min, B. Di Martino, L.T. Yang, M. Guo and Gudula Rünger (Eds.), *Frontiers of High Performance Computing and Networking— ISPA 2006 International Workshops, Sorrento, Italy, December 4-7, 2006*, Lecture Notes in Computer Science, Vol. 4331, Springer-Verlag, Berlin/Heidelberg, pp. 474–486.

Zaharia, M., Konwinski, A., Joseph, A., Katz, R. and Stoica, I. (2008). Improving MapReduce performance in heterogeneous environments, *in* R. Draves and R. van Renesse (Eds.), *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8–10, 2008, San Diego, California, USA*, USENIX Association, Berkeley, CA.

Horacio González-Vélez, as a lecturer with the School of Computing of Robert Gordon University, conducts research in parallel and distributed computing including structured parallelism and grid/cloud computing. He holds a doctoral degree in informatics from the University of Edinburgh, where he was also a research fellow with the School of Informatics. Previously, he held different positions in marketing and systems engineering at Sun Microsystems and Silicon Graphics, where he was involved in different innovation-related projects at a number of companies and universities. He has been recognised with the NESTA Crucible Fellowship for his inter-disciplinary research on computational science in 2009 and with a European Commission award for his efforts on scientific dissemination in 2006. His research has been funded by the Engineering and Physical Sciences Research Council (UK), the European Commission, NESTA (UK), NVIDIA, and Sun Microsystems.

Maryam Kontagora is an information technology professional currently employed in the oil and gas industry. She spends most of her work time engaged in system architecture design and the development of business applications. Educated in Nigeria and the UK, Maryam holds a master's degree in computing (information engineering) from Robert Gordon University in Aberdeen. Her passion for learning has led her to explore different facets of computing. Research interests include the areas of parallel computing and virtualisation.

Received: 28 June 2010
Revised: 16 November 2010
Re-revised: 2 January 2011