K Means of Cloud Computing: MapReduce, DVM, and Windows Azure

Lin Gu Zhonghua Sheng Zhiqiang Ma
Xiang Gao Charles Zhang
Department of Computer Science and Engineering
Hong Kong University of Science and Technology
Kowloon, Hong Kong SAR
Email: {lingu,szh,zma,xgaoaa,charlesz}@cse.ust.hk

Yaohui Jin
State Key Lab of Advanced Optical Communication
Systems and Networks, Shanghai Jiaotong University
800 Dongchuan Road, Minghang District
Shanghai, China
Email: jinyh@sjtu.edu.cn

***Abstract*—Cloud-based systems and the datacenter computing environment present a series of challenges to system designers for supporting massively concurrent computation on clusters with commodity hardware. The platform software should abstract the unreliable but highly provisioned hardware to provide a high-performance platform for a diversity of concurrent programs processing potentially very large data sets. Toward this goal, a number of solutions are designed or proposed. Among these products and systems, we elect three technologies, MapReduce/Hadoop, DVM, and Windows Azure, as representatives of three different approaches to constructing the infrastructure and instructing the programming in the cloud. We empirically study these technologies using a well-known and widely used application, k-means, and analyze their performance data in relation with the abstraction layers they establish. The implementations of k-means on the three platforms are presented with sufficient details to show the design patterns with these technologies. We analyze the evaluation results in the context of the design goals and constraints of the technologies, and show that the instruction-level abstraction can provide flexible programming capability as well as high performance.**

***Keywords*—Cloud computing; k-means; parallel programming; MapReduce; DISA; big data processing**

I. INTRODUCTION

We entered the cloud computing era without a consensus on how large-scale distributed computing systems should be constructed. As many problems remain unsolved for systems with hundreds of loosely-coupled nodes, leading Internet firms have constructed datacenters orders of magnitude larger than typical "large-scale" systems around 2000's. To system designers, datacenter systems present new technical challenges for the following reasons.

- First, the scale of a datacenter can reach hundreds of thousands of compute servers, which is out of the scope of many distributed algorithms.
- Second, constructing a loosely coupled system at such a scale with commodity hardware inevitably introduces faults in the system to the extent that failures of components are "norm" [1]. This design context departs significantly from traditional high-performance computing systems.
- Third, the applications in datacenters typically require extremely high availability and process very large data with high throughput [2]. Moreover, a number of computing tasks require deterministic output to ensure correctness, which is well accepted practice in computations of smaller scale but turns out to be very difficult in datacenter systems without noticeably affecting performance.
- Finally, a datacenter is a shared environment where a number of applications run concurrently and may interact with each other. In contrast, a typical high-performance computing (HPC) environment can run in a dedicated or isolated manner. In fact, many HPC users desire to have their application run in relatively isolated resource compartments.

In this context, the cloud computing infrastructure should abstract the unreliable but highly provisioned hardware to provide a high-performance platform for a diversity of concurrent programs processing potentially very large data sets. The programs should be easy to write, worry-free to deploy, and fast to execute.

Several technologies are developing towards this goal– besides earlier solutions developed by Google, Yahoo!, and other industry firms, integrated solutions start to emerge and combine existing software development practices.In addition, a few academic research systems exhibit excellent performance and potentially indicate future directions of innovation in this area.

Among these products and systems, we elect three technologies, MapReduce/Hadoop [3], DVM [4], and Windows Azure [5], as representatives of three different approaches to constructing the infrastructure and instructing the programming in the cloud. We empirically study these technologies using a well-known and widely used application, k-means, and analyze their performance data in relation with the abstraction layers they establish. The implementations of k-means on the three platforms are presented with sufficient details to show the design patterns with these technologies. Our study reveals some characteristics of the design space of cloud computing, and sheds light onto how to construct and program cloud-based systems and applications.

The rest of the paper is organized as follows. Section II introduces the background of the technologies discussed in this paper. Section III presents the k-means programming on

MapReduce/Hadoop, DVM and Windows Azure. Section IV evaluates the performance of k-means computation on the three platforms, and analyzes the experimental results. The related work is discussed in Section V, and we provide concluding remarks in Section VI.

II. Background

When Internet datacenters were first multiplexed to conduct serious data-intensive processing, it became obvious that there lacked a method to orchestrate the numerous compute nodes in such systems to conduct effective computation. In spite of immense work on distributed computing and parallel processing, traditional approaches are ill-suited for the new computing platform. Consequently, several technologies have been developed to enable large-scale distributed processing in datacenters, pioneered by Google's MapReduce [3]. Recently, Microsoft's Windows Azure integrated a full set of cloud-related technologies, including not only distributed execution but also programmable resource provisioning and data-layer abstractions, in the existing development frameworks [5], [6]. Another important trend is to conduct in-memory computation on commodity-hardware-based clusters. As one of the earliest approaches in this category, the DVM technology constructs an instruction-level abstraction to enable programs distribute computation in a large shared memory space [4].

A. MapReduce-style computation

MapReduce is perhaps the most widely recognized cloud computing technology. It simplifies the data dependence and regulates the semantics of the tasks (e.g., tasks should be idempotent) so that it is easy to implement "embarrassingly parallel" programs and utilize the large number of processor cores in a cluster [3]. Although multiple implementations extend the MapReduce framework to multicore and GPGPU processing [7], [8], MapReduce is mainly design for massively parallel data-intensive processing on a cluster of compute nodes.

While MapReduce is a computational framework, its design is highly dependent on the underlying filesystem abstraction, GFS [1]. First, the replicated data chunks in the filesystem effectively enhance the scheduling efficacy and the I/O bandwidth. Second, the filesystem provides a means of maintaining very large program state and providing a "global" namespace. Finally, atomic operations (e.g., rename) in the filesystem ensures the correctness of the MapReduce computation. The performance of MapReduce computation also relies on a datacenter-wide "meta-scheduler". The open source variant of MapReduce, Hadoop, has implemented a filesystem, HDFS, with similar semantics to those provide by GFS and a application-level task scheduler.

B. Languages, virtual machines, and DVM

Virtualization is considered part of the technical foundation of cloud computing. In fact, virtualization can take place at several different system layers, and the level of abstraction makes significant difference in generality, expressiveness, and performance. X10 represents an approach of abstracting computation at the language level [9]. Similar approaches include

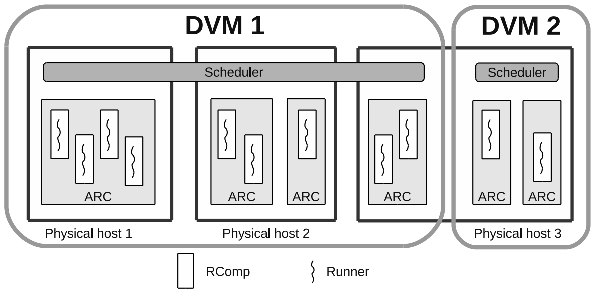

Fig. 1. Organization of two DVM virtual machines on three computers. Each DVM virtual machine utilizes computing resources provided by one single computer or many computers.

Fortress, Google App Engine, and Chapel. The language-level approach gives the programmers more precise control of the semantics of parallelization and synchronization, and X10's PGAS approach (Partitioned Global Address Space) can potentially support very large data for sophisticated processing. Amazon EC2, on the other hand, abstracts the platform at the instruction level, and builds on existing VMM (Virtual Machine Monitor) technology.

The DVM technology represents a new virtualization function, which provides a low-level abstraction but enables it to support large-scale clusters and sophisticated parallelizable processing [4]. It introduces a new ISA, Datacenter Instruction Set Architecture (DISA), which can be easily emulated on existing hardware. Above such a general architecture, a large virtual machine, DVM, can coordinate resources from a large number of physical hosts and support various programming languages.

Fig. 1 shows the organization of a DVM system composed of two virtual machines spanning three physical hosts. The instruction-level abstraction provided by DISA is very close to typical machine instructions directly supported by processor hardware. However, the semantics of the instructions and the memory model enable multiple tasks, each called a runner, to reside in a large shared memory space, conduct computation and orchestrate massively parallel processing in the abstraction of a "single computer". The runners resolve their dependence upon each other through a watcher mechanism provided by the DVM. In particular, a latter task depending on an earlier task's output can be implemented as a "watcher" that monitors the memory area where the former task writes the output data. Once the watched area is modified, the latter task is activated and allowed to proceed with its computation.

C. Windows Azure

The Windows Azure Platform, developed by Microsoft, provides a full set of abstractions and programming tools for developing cloud-based applications. It also uses several related technologies, e.g., VMM, scalable key value store, and datacenter-oriented programming languages, to construct a fairly complete solution.

An application materializes as services hosted in Windows Azure, consisting of one or multiple web roles together with a set of optional worker roles. The program running in the roles may invoke distributed key value store or database services via well-defined APIs to meet the requirements of a spectrum of applications, including commercial applications requiring strict transactional semantics. Similar to DVM, Azure allows programmers to use programming languages of their choice, given that the language is supported by the program development environment (e.g., Microsoft Visual Studio) and the Windows Azure SDK.

III. K-MEANS PROGRAMMING AND SYSTEM SUPPORT

k-means is a well-known data clustering application used in many areas such as data mining, computing vision and information retrieval.It partitions a data set into `k` clusters iteratively, and has been implemented in various software systems and applications.Moreover, k-means is widely used for evaluating cloud-based technologies, and, with its clear algorithmic design and adjustable problem size, presents a manageable workload with which various cloud-related technologies can be studied [10], [11].

The k-means process starts with `k` initial cluster centroids and iteratively refines the clusters by reassigning points to the closest centroids and updating the clusters' centroids. We implement the k-means algorithm with a similar iterative workflow to the one used in Mahout and X10, and optimize the algorithm to achieve better performance in the distributed environment. Similar optimizations are also used in some prior work [10], [12].

A. MapReduce and Hadoop

The iterative computation of k-means does not directly fit into the MapReduce framework, which mandates a reduce stage following a map stage. However, the computation in each iteration is similar with different cluster centroids and the two phases (assigning points to clusters and calculating the new centroids) in each iteration can be expressed as one MapReduce job—we use the map tasks to perform the distance computation and point assignment to clusters as the distance computation between one point and the centroids is irrelevant to the computation for other points in one iteration, and the distance computation can be executed in parallel. The calculation of the new centroids can be performed by the reduce tasks. Hence, we can iteratively run MapReduce jobs and each MapReduce job performs the computation in each iteration of the k-means algorithm. As the distance computation is the most intensive calculation in k-means algorithm, the computation is effectively parallelized using the MapReduce programming model.

Alg. 1 shows the k-means clustering algorithm on Hadoop. The input data are initially stored in files of roughly equal sizes. The input files contain data points' coordinates as a sequence of <key, value>pairs where the coordinates are stored in the value field. To share the centroids which are read and updated by each MapReduce job, we store the centroids in files in HDFS so that they are read by the map tasks for distance computation and are updated by reduce tasks with the new centroids. Hence, the final output of the k-means cluster program is the centroid files after the last iteration.

The combine function minimizes the communication among map and reduce tasks. Using multiple MapReduce jobs, we are able to implement the iterative computation required by

Algorithm 1 k-means clustering using Hadoop

```
 1: create current_centroids and new_centroids in the file system
 2: write new_centroids with the first $k$ points in the input files
 3: repeat
 4:   delete current_centroids, rename new_centroids to current_centroids, and create empty new_centroids
 5:   for all map tasks do
 6:     read the data points from the input files
 7:     read current_centroids
 8:     for all data points do
 9:       calculate the distances between the data point and each centroid
10:       n= the identity of the cluster with the closest centroid
11:       v=coordinates of the data point
12:       output the <n, v>(assign data point v to cluster n)
13:     end for
14:   end for
14:   Run the combine function to sum the values of data points assigned to the same cluster and output <n, V>for each distinct n where V is a composite value of the coordinates of the centroid of the data points being combined and the number of data points associated with n
15:   for all reduce tasks do
16:     sum all the intermediate values generated by the combine functions and compute the new cluster centroids.
17:     write the new centroids to file new_centroids
18:   end for
19: until the difference between the centroids in current_centroids and new_centroids is less than a threshold or the number of iterations reaches the maximum value
```

k-means. However, the transition between successive MapReduce jobs cannot be expressed inside the MapReduce framework itself, and external "glue" language must be employed to make such transition happen. It is also noteworthy that the external logic forces the program to use the distributed file system as the media for recording program state. These issues, although tolerable in "embarrassingly parallel" programs, results in non-trivial burden in programming and performance for this slightly sophisticated application.

B. DISA and DVM

DISA presents a generic programming platform, and DVM is constructed above this generic abstraction layer. Hence, it is not difficult to implement the k-means algorithm on a DVM. The program flows are instantiated to runners in DVM and the dependence between the iterations and phases inside each iteration is expressed with watchers. The k-means program on DVM reads its input from disks through one of its I/O channels. Alg. 2 shows the k-kmeans clustering algorithm on DVM.

It may appear to be an unnecessary overhead that the program creates M `dist_cal_runners` in each iteration. In fact, this design results from the snapshotted memory semantics in DISA–the runners see data in its snapshot created upon the runner's instantiation, and the new centroids created at the end of one iteration are visible to runners created at the beginning of the next iteration. It is very efficient to spawn new runners on a DVM, and this makes the overhead of creating runners practically negligible. In comparison, the MapReduce-style programming also requires the program to

Algorithm 2 k-means clustering on DVM

```
 1: read data points from the I/O channel and store them in designated memory areas
 2: new_centroids = the first $k$ data points
 3: repeat
 4:   current_centroids = new_centroids;
 5:   create $M$ (a program parameter) dist_cal_runners, each responsible for one partition of data points, and one dist_cal_runner_watcher.
 6:   for all dist_cal_runner and its associated partition P do
 7:     for all data point p in P do
 8:       calculate the distances between p and each centroid in current_centroids
 9:       assign p to the closest centroid n
10:       add p to the sum for centroid n in P – store n, the sum with p included and the number of data points, including p, associated with n in P
11:     end for
12:     dist_cal_runner_watcher is activated each time a dist_cal_runner exits and commits.
13:     dist_cal_runner_watcher checks whether all dist_cal_runner runners have completed
14:     if all dist_cal_runner runners have completed then
15:       the watcher creates the centroid_cal_runner
16:       centroid_cal_runner sums all the intermediate values generated by dist_cal_runners for each centroid, computes the new cluster centroids and assigns them to new_centroids
17:     else
18:       exit the watcher
19:     end if
20:   end for
21: until the difference between the centroids in current_centroids and new_centroids is less than a threshold or the number of iteration reaches the maximum value
```

create numerous map and reduce tasks in each iteration, but the tasking overhead is very heavy in the current implementations.

C. Windows Azure

The web and worker roles in Azure are general enough to implement almost any computational jobs with Windows Azure-enabled languages, with web roles incorporated with built-in web servers. However, it is still a technical challenge to use the distributed data services to construct a reliable mechanism for recording program state and enabling web and worker roles to exchange intermediate data. Alg. 3 shows the design of k-means on Windows Azure.

To implement k-means on Windows Azure, we use the Windows Azure blob storage to store the input dataset and the output results. The communication between different roles relies on the Windows Azure queue service. We build two types of worker roles – a *master* role and a *slave* role. There is only one master worker role (henceforth called *master*) instance which is responsible of partitioning the dataset, assigning tasks, and collecting results. There are one or multiple slave worker role (henceforth called *slave*) instances. They consume the tasks in the task queue, generate the intermediate results in its data partition, and write back to the result queue.

Algorithm 3 k-means clustering on Windows Azure

```
1: new_centroids = the first $k$ data points in the input
2: repeat
3:   current_centroids = new_centroids
4:   master partitions the dataset
5:   master writes centroids together with the task control information (e.g., the number of concurrent tasks) into the task queue
6:   for all slaves do
7:     retrieve the tasks from the queue and compute the intermediate results consisting of the centroid assignment, the sum of the coordinates of the data points assigned to a cluster in its partition and the number of data points in the corresponding cluster and partition
8:   end for
8:   master collects the intermediate results, computes the new centroids, and assigns them to new_centroids
9: until the difference between the centroids in current_centroid and new_centroids is less than a threshold or the number of iteration reaches the maximum value
```

IV. Performance, programmability, and empirical experience

With k-means implemented on Hadoop, DVM, and Azure, we conduct an empirical study on these implementations to study the performance of these solutions, and link the observed performance data to the design choices in cloud computing technologies. To ensure the applicability of our observations, we run the experiments on both research testbeds and industrial platforms such as industrial computing clusters and the Windows Azure platform.

Fig. 2 presents the execution time for k-means on DVM and Hadoop on 16 working nodes. From the results, we can see that DVM is at least 13 times faster than Hadoop. We believe this indicates that instruction-level abstractions can lead to more efficient computation and less tasking overhead. While an optimized language-layer construct, such as a MapReduce implementation using memory as the main data storage, can significantly increase the performance, such optimization is unlikely to close the gap between the language and instruction-layer abstractions.

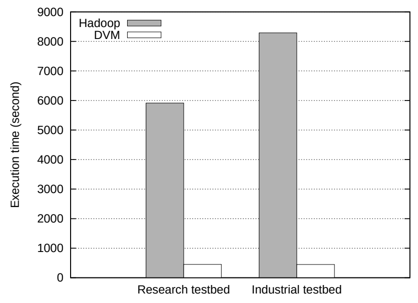

Fig. 2. Execution time for k-means on 16 nodes

Illustrating the speedup, Fig. 3 shows the relative performance for k-means on DVM and Hadoop on the research testbed ("/R" in the figure) and industrial testbed ("/I" in the figure) as we scale the number of compute nodes. The relative performance is calculated with respect to execution time on Hadoop with one node. Fig. 4 presents the execution time and

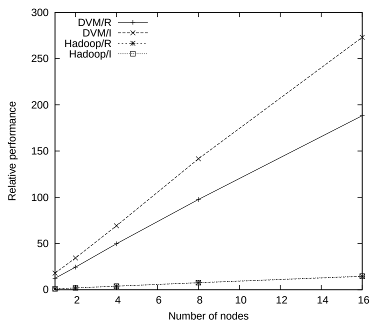

Fig. 3. Relative performance of k-means

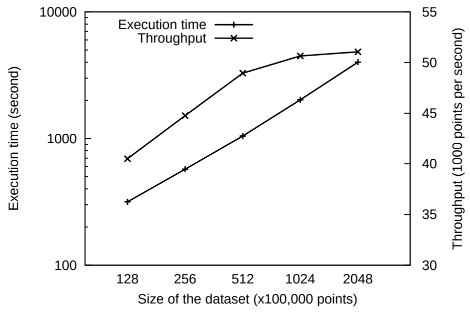

Fig. 4. Execution time and throughput of k-means

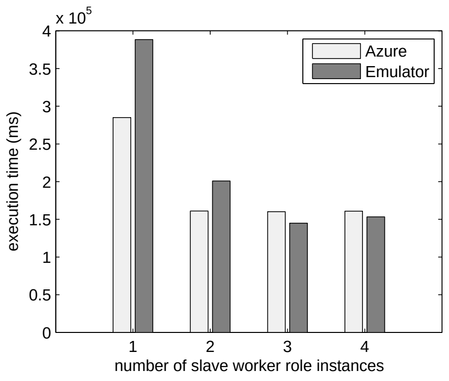

Fig. 5. Execution time of k-means computation on Windows Azure

throughput of k-means on DVM with 50 compute nodes as we scale the dataset from 12,800,000 to 204,800,000 points. The throughput is calculated through dividing the number of points by the execution time. The result shows that the throughput increases with the data size, which reflects that DVM scales well with the data size.

Since DVM shows excellent scalability and efficiency, it may appear that the instruction-level abstraction represents the best choice for constructing the cloud technology. However, similar to the situation with traditional ISAs, the instruction layer is mainly defining the interface between hardware and software, and may not provide a complete solution to programming. In fact, our experience of developing programs in the DISA assembly language verifies the challenge of developing programs at a level close to the instruction set, and has prompted us to start developing a compiler for DISA. The goal of the DVM is to provide a powerful foundation, rather than the completion, of the cloud computing technology, and new software tools and supportive routines shall be added to the platform to fully utilize its capability and enhance productivity.

To provide a complete programming environment, Windows Azure integrates the distributed data and processing services with the familiar Visual Studio based development environment. Fig. 5 shows the performance of k-means on Windows Azure platform with 1 *master* and 1 to 4 *slave*s. The worker roles reside on small instances in Windows Azure, and both the roles and the Azure storage service are located in the "South Central US" region. We also uses the local emulator to evaluate and compare the execution of k-means. The Windows Azure emulator runs on a server with 4 CPU cores and 6GB of memory.

We observe that, when the number of *slave*s increases from 3 to 4, the speedup becomes low or even negative. Based on our study, this is likely due to the sharing of CPU resources— 3 *slave*s and 1 *master* can each use 1 CPU core almost exclusively. However, when we have 1 *master* and 4 *slave*s, totally 5 worker role instances share the 4 CPU cores on the physical host, and this serializes a significant part of the computation. Similarly, on the Windows Azure platform, we also observed the same phenomenon. Such hardware-coupling overhead can be mitigated by better scheduling or more resources. Looking at the performance data, we can also conclude that, without the hardware-coupling overhead, k-means exhibits obvious speedup on Windows Azure. This verifies that the Windows Azure platform, although designed to provide an easy-to-program methodology in a familiar development environment, can potentially support parallelized scientific computing with the worker roles.

The evaluation clearly shows that the instruction-level abstraction, DISA and DVM, exhibits superior performance in the computation. More importantly, the Turing-complete instruction set of DISA presents a model that can express a wide range of applications. We believe that such generality is a key advantage in the future design of cloud computing systems. Meanwhile, MapReduce has been proved an effective solution to data-intensive computing when the processing logic and data dependence relation fit its specific computation model. Windows Azure, although optimized for Web-based applications, also exhibits a significant amount of flexibility in supporting scientific computation.

V. RELATED WORK

Many programming frameworks and languages are proposed and designed to exploit the computing power of the large number of compute servers inside today's gigantic datacenters. Dean et al. have created the MapReduce programming model for Google's datacenter environment [3]. Dryad takes a more general approach, using a "communication DAG (directed acyclic graph)" to depict the dependency among multiple task instances [13]. While these frameworks are successful in large data processing, the restricted programming model makes it difficult to design sophisticated and time-sensitive applications [11], [14], [15], [16].

DVM, on the other hand, allows programmers to easily design general-purpose applications running on a large number of compute nodes by providing a more flexible programming model [4]. DVM and DISA, the instruction set of DVM, represent a virtualization technology different from widely used virtualization systems, such as Xen and VMware on the x86 ISA [17], [18]. As comparison, VMware pioneered the virtualization of the x86 ISA, and vNUMA extends IA-64 to multiple hosts connected through an Ethernet network that provides “sender-oblivious total-order broadcast” [19]. The new DISA instruction set allows programs to scale up to much larger clusters. Currently, optimization of programs running on a DVM is programmer-driven. Although it has been shown that the performance of DISA programs are very high, optimizing compilers will make it much easier to harness such optimization techniques.

In the meantime, the instruction-level abstraction must combine with high-order languages and compilers to fully release its capability. High-level languages, such as X10, Sawzall and DryadLINQ, which are implemented on top of programming frameworks (MapReduce and Dryad), make the data-processing programs easier to design [9], [20], [21]. To implement their linguistic features efficiently, the language-level approach gives also calls for an underlying computing infrastructure that is capable of supporting general-purpose programs, follows a storage-computing coupled architecture, and provides measures for system-wide optimization. Towards these requirements, DVM provides a foundation upon which language-level instruments can be built.

VI. Conclusion

Our study shows that DVM has the best performance for computation in a datacenter, but higher-order languages and compilers must be used to make instruction-level abstraction easy to program. Meanwhile, Windows Azure provides a relatively full set of data services, language support, and development and deployment tools. The role constructs together with the queue-based communication can support a variety of applications, including scientific workloads, on Windows Azure.

Acknowledgment

This work was supported in part by the Huawei Technologies research grant HUAW17-15G00510/11PN, HKUST research grants REC09/10.EG06, DAG11EG04G and SJTU research grant 2011GZKF030902. We thank Microsoft for providing computing resources and support for the research on the Windows Azure platform.

References

[1] 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.

[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] J. Dean and S. Ghemawat, “MapReduce: simplified data processing on large clusters,” In Proc. of the 6th Symp. on Opearting Systems Design & Implementation (OSDI’04), Berkeley, CA, USA, 2004, pp. 137–149.

[4] Z. Ma, Z. Sheng, L. Gu, L. Wen, and G. Zhang, “DVM: Towards a datacenter-scale virtual machine,” in Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments, 2012, pp. 39–50.

[5] “Windows Azure,” http://www.windowsazure.com/, [last access: 3/31/2013].

[6] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum, “Fast crash recovery in ramcloud,” in Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, ser. SOSP ’11, 2011, pp. 29–41.

[7] B. He, W. Fang, Q. Luo, N. Govindaraju, and T. Wang, “Mars: a MapReduce framework on graphics processors,” in Proceedings of the 17th international conference on parallel architectures and compilation techniques, 2008, pp. 260–269.

[8] R. Yoo, A. Romano, and C. Kozyrakis, “Phoenix rebirth: Scalable mapreduce on a large-scale shared-memory system,” in Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on. IEEE, 2009, pp. 198–207.

[9] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar, “X10: an object-oriented approach to non-uniform cluster computing,” in ACM SIGPLAN Notices, vol. 40, no. 10, 2005, pp. 519–538.

[10] 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.

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

[12] W. Zhao, H. Ma, and Q. He, “Parallel k-means clustering based on mapreduce,” in roceedings of the First International Conference on Cloud Computiong (CloudCom), 2009, pp. 674–679.

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

[14] Z. Ma and L. Gu, “The limitation of MapReduce: A probing case and a lightweight solution,” in Proc. of the 1st Intl. Conf. on Cloud Computing, GRIDs, and Virtualization, 2010, pp. 68–73.

[15] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, “Evaluating MapReduce for multi-core and multiprocessor systems,” in Proc. of the 2007 IEEE 13th Intl. Symposium on High Performance Computer Architecture, 2007, pp. 13–24.

[16] H.-C. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker, “Map-Reduce-Merge: simplified relational data processing on large clusters,” in SIGMOD ’07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data, 2007, pp. 1029–1040.

[17] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, “Xen and the art of virtualization,” in Proceedings of the 19th ACM symposium on Operating Systems Principles, 2003, pp. 164–177.

[18] C. A. Waldspurger, “Memory resource management in VMware ESX server,” SIGOPS Oper. Syst. Rev., vol. 36, no. SI, pp. 181–194, 2002.

[19] M. Chapman and G. Heiser, “vnuma: A virtual shared-memory multiprocessor,” in Proceedings of the 2009 USENIX Annual Technical Conference, June 2009, pp. 15–28.

[20] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan, “Interpreting the data: Parallel analysis with Sawzall,” Sci. Program., vol. 13, no. 4, pp. 277–298, 2005.

[21] 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 the 8th Conference on Symposium on Operating Systems Design & Implementation, 2008, pp. 1–14.