High Performance Parallel Computing with Clouds and Cloud Technologies

Jaliya Ekanayake and Geoffrey Fox

School of Informatics and Computing,
Indiana University, Bloomington, IN 47405, USA
{jekanaya,gcf}@indiana.edu

Abstract. Infrastructure services (Infrastructure-as-a-service), provided by cloud vendors, allow any user to provision a large number of compute instances fairly easily. Whether leased from public clouds or allocated from private clouds, utilizing these virtual resources to perform data/compute intensive analyses requires employing different parallel runtimes to implement such applications. Among many parallelizable problems, most "pleasingly parallel" applications can be performed using MapReduce technologies such as Hadoop, CGL-MapReduce, and Dryad, in a fairly easy manner. However, many scientific applications, which have complex communication patterns, still require low latency communication mechanisms and rich set of communication constructs offered by runtimes such as MPI. In this paper, we first discuss large scale data analysis using different MapReduce implementations and then, we present a performance analysis of high performance parallel applications on virtualized resources.

Keywords: Cloud, Virtualization, MapReduce, Dryad, Parallel Computing.

1 Introduction

The introduction of commercial cloud infrastructure services such as Amazon EC2/S3 [1-2] and GoGrid[3] allow users to provision compute clusters fairly easily and quickly by paying a monetary value only for the duration of the usage of resources. The provisioning of resources happens in minutes as opposed to the hours and days required in the case of traditional queue-based job scheduling systems. In addition, the use of such virtualized resources allows the user to completely customize the Virtual Machine (VM) images and use them with root/administrative privileges, which is another feature that is hard to achieve with traditional infrastructures.

The availability of open source cloud infrastructure software such as Nimbus [4] and Eucalyptus [5], and the open source virtualization software stacks such as Xen Hypervisor[6], allows organizations to build private clouds to improve the resource utilization of the available computation facilities. The possibility of dynamically provisioning additional resources by leasing from commercial cloud infrastructures makes the use of private clouds more promising.

With all the above promising features of cloud, we can assume that the accessibility to computation power is no longer a barrier for the users who need to perform large

D.R. Avresky et al. (Eds.): Cloudcomp 2009, LNICST 34, pp. 20–38, 2010.

scale data/compute intensive applications. However, to perform such computations, two major pre-conditions need to be satisfied: (i) the application should be parallelizable to utilize the available resources; and (ii) there should be an appropriate parallel runtime support to implement it.

We have applied several cloud technologies such as Hadoop[7], Dryad and DryadLINQ[8,9], and CGL-MapReduce[10], to various scientific applications wiz: (i) Cap3[11] data analysis; (ii) High Energy Physics(HEP) data analysis; (iv) Kmeans clustering[12]; and, (v) Matrix Multiplication. The streaming based MapReduce [13] runtime - CGL-MapReduce- developed by us extends the MapReduce model to iterative MapReduce domain as well. Our experience suggests that although most "pleasingly parallel" applications can be performed using cloud technologies such as Hadoop, CGL-MapReduce, and Dryad, in a fairly easy manner, scientific applications, which require complex communication patterns, still require more efficient runtime support such as MPI[14].

In order to understand the performance implications of virtualized resources on MPI applications, we performed an extensive analysis using Eucalyptus based private cloud infrastructure. The use of a private cloud gives us complete control over both VMs and bare-metal nodes, a feature that is impossible to achieve in commercial cloud infrastructures. It also assures a fixed network topology and bandwidth with the nodes deployed in the same geographical location, improving the reliability of our results. For this analysis, we used several MPI applications with different communication/computation characteristics, namely Matrix Multiplication, Kmeans Clustering, and Concurrent Wave Equation Solver and performed them on several VM configurations. Instead of measuring individual characteristics such as bandwidth and latency using micro benchmarks we used real applications to understand the effect of virtualized resources for such applications, which makes our results unique.

In the sections that follow, we first present the work related to our research followed by a brief introduction to the data analysis applications we used. Section 4 presents the results of our evaluations on cloud technologies and a discussion. In section 5, we discuss an approach with which to evaluate the performance implications of using virtualized resources for high performance parallel computing. Section 6 presents the results of this evaluation along with a discussion of the results. In the final section we give our conclusions and we discuss implications for future work.

2 Related Work

Traditionally, most parallel applications achieve fine grained parallelism using message passing infrastructures such as PVM [15] and MPI. Applications achieve coarse-grained parallelism using workflow frameworks such as Kepler [16] and Taverna [17], where the individual tasks could themselves be parallel applications written in MPI. Software systems such as Falkon [18], SWARM [19], and DAGMan [20] can be used to schedule applications which comprise of a collection of a large number of individual sub tasks.

Once these applications are developed, in the traditional approach, they are executed on compute clusters, super computers, or Grid infrastructures [21] where the focus on allocating resources is heavily biased by the availability of computational

power. The application and the data both need to be moved to the available computational power in order for them to be executed. Although these infrastructures are highly efficient in performing compute intensive parallel applications, when the volumes of data accessed by an application increases, the overall efficiency decreases due to the inevitable data movement.

Cloud technologies such as Google MapReduce, Google File System (GFS) [22], Hadoop and Hadoop Distributed File System (HDFS) [7], Microsoft Dryad, and CGL-MapReduce adopt a more data-centered approach to parallel runtimes. In these frameworks, the data is staged in data/compute nodes of clusters or large-scale data centers, such as in the case of Google. The computations move to the data in order to perform data processing. Distributed file systems such as GFS and HDFS allow Google MapReduce and Hadoop to access data via distributed storage systems built on heterogeneous compute nodes, while Dryad and CGL-MapReduce support reading data from local disks. The simplicity in the programming model enables better support for quality of services such as fault tolerance and monitoring. Table 1 highlights the features of three cloud technologies that we used.

Table 1. Comparison of features supported by different cloud technologies

Feature	Hadoop	Dryad & DryadLINQ	CGL-MapReduce
Programming Model	MapReduce	DAG based execution flows	MapReduce with *Combine* phase
Data Handling	HDFS	Shared directories/ Local disks	Shared file system / Local disks
Intermediate Data Communication	HDFS/ Point-to-point via HTTP	Files/TCP pipes/ Shared memory FIFO	Content Distribution Network (NaradaBrokering[23])
Scheduling	Data locality/ Rack aware	Data locality/ Network topology based run time graph optimizations	Data locality
Failure Handling	Persistence via HDFS Re-execution of map and reduce tasks	Re-execution of vertices	Currently not implemented (Re-executing map tasks, redundant reduce tasks)
Monitoring	Monitoring support of HDFS, Monitoring MapReduce computations	Monitoring support for execution graphs	Programming interface to monitor the progress of jobs
Language Support	Implemented using Java Other languages are supported via Hadoop Streaming	Programmable via C# DryadLINQ provides LINQ programming API for Dryad	Implemented using Java Other languages are supported via Java wrappers

Y. Gu, et al., present Sphere [24] architecture, a framework which can be used to execute user-defined functions on data stored in a storage framework named Sector, in parallel. Sphere can also perform MapReduce style programs and the authors compare the performance with Hadoop for tera-sort application. Sphere stores intermediate data on files, and hence is susceptible to higher overheads for iterative applications.

All-Paris [25] is an abstraction that can be used to solve a common problem of comparing all the elements in a data set with all the elements in another data set by applying a given function. This problem can be implemented using typical MapReduce frameworks such as Hadoop, however for large data sets, the implementation will not be efficient, because all map tasks need to access all the elements of one of the data sets. We can develop an efficient iterative MapReduce implementation using CGL-MapReduce to solve this problem. The algorithm is similar to the matrix multiplication algorithm we will explain in section 3.

Lamia Youseff, et al., presents an evaluation on the performance impact of Xen on MPI [26]. According to their evaluations, the Xen does not impose considerable overheads for HPC applications. However, our results indicate that the applications that are more sensitive to latencies (smaller messages, lower communication to computation ratios) experience higher overheads under virtualized resources, and this overhead increases as more and more VMs are deployed per hardware node. From their evaluations it is not clear how many VMs they deployed on the hardware nodes, or how many MPI processes were used in each VM. According to our results, these factors cause significant changes in results. Running 1-VM per hardware node produces a VM instance with a similar number of CPU cores as in a bare-metal node. However, our results indicate that, even in this approach, if the parallel processes inside the node communicate via the network, the virtualization may produce higher overheads under the current VM architectures.

C. Evangelinos and C. Hill discuss [27] the details of their analysis on the performance of HPC benchmarks on EC2 cloud infrastructure. One of the key observations noted in their paper is that both the OpenMPI and the MPICH2-nemsis show extremely large latencies, while the LAM MPI, the GridMPI, and the MPICH2-scok show smaller smoother latencies. However, they did not explain the reason for this behavior in the paper. We also observed similar characteristics and a detailed explanation of this behavior and related issues are given in section 5.

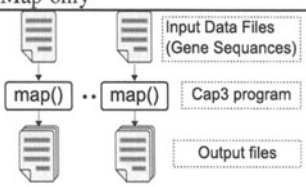
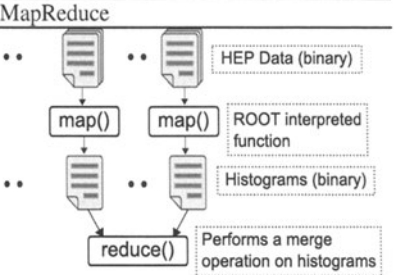
Edward Walker presents benchmark results of performing HPC applications using "high CPU extra large" instances provided by EC2 and on a similar set of local hardware nodes [28]. The local nodes are connected using infiniband switches while Amazon EC2 network technology is unknown. The results indicate about 40%-1000% performance degradation on EC2 resources compared to the local cluster. Since the differences in operating systems and the compiler versions between VMs and bare-metal nodes may cause variations in results, for our analysis we used a cloud infrastructure that we have complete control. In addition we used exactly similar software environments in both VMs and bare-metal nodes. In our results, we noticed that applications that are more susceptible to latencies experience higher performance degradation (around 40%) under virtualized resources. The bandwidth does not seem to be a consideration in private cloud infrastructures.

Ada Gavrilvska, et al., discuss several improvements over the current virtualization architectures to support HPC applications such as HPC hypervisors (sidecore) and self-virtualized I/O devices [29]. We notice the importance of such improvements and research. In our experimental results, we used hardware nodes with 8 cores and we deployed and tested up to 8VMs per node in these systems. Our results show that the virtualization overhead increases with the number of VMs deployed on a hardware node. These characteristics will have a larger impact on systems having more CPU cores per node. A node with 32 cores running 32 VM instances may produce very large overheads under the current VM architectures.

3 Data Analysis Applications

The applications we implemented using cloud technologies can be categorized into three classes, depending on the communication topologies wiz: (i) Map-only; (ii) MapReduce; and (iii) Iterative/Complex. In our previous papers [10,30], we have presented details of MapReduce style applications and a Kmeans clustering application that we developed using cloud technologies, and the challenges we faced in developing these applications. Therefore, in this paper, we simply highlight the characteristics of these applications in table 2 and present the results. The two new applications that we developed, Cap3 and matrix multiplication, are explained in more detail in this section.

Table 2. Map-Only and MapReduce style applications

Feature	Map-only	MapReduce
Program/data flow		
	Cap3 Analysis application implemented as a map-only operation. Each map task processed a single input data file and produces a set of output data files.	HEP data analysis application implemented using MapReduce programming model (ROOT is an object-oriented data analysis framework).
More Examples	Converting a collection of documents to different formats, processing a collection of medical images, and Brute force searches in cryptography	*Histogramming* operations, distributed search, and distributed sorting.

3.1 Cap3

Cap3 is a sequence assembly program that operates on a collection of gene sequence files which produce several output files. In parallel implementations, the input files are processed concurrently and the outputs are saved in a predefined location. For our analysis, we have implemented this application using Hadoop, CGL-MapReduce and DryadLINQ.

3.2 Iterative/Complex Style Applications

Parallel applications implemented using message passing runtimes can utilize various communication constructs to build diverse communication topologies. For example, a matrix multiplication application that implements Cannon's Algorithm [31] assumes parallel processes to be in a rectangular grid. Each parallel process in the gird communicates with its left and top neighbors as shown in Fig. 1(left). The current cloud runtimes, which are based on data flow models such as MapReduce and Dryad, do not support this behavior, where the peer nodes communicate with each other. Therefore, implementing the above type of parallel applications using MapReduce or Dryad models requires adopting different algorithms.

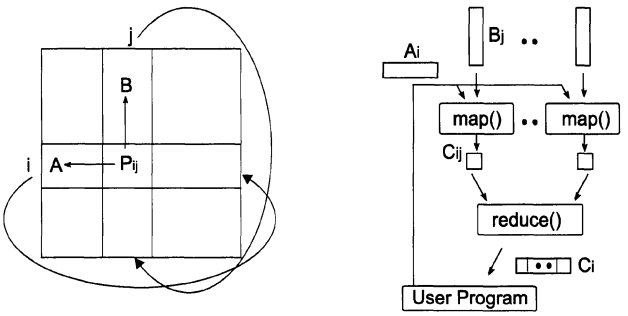


Fig. 1. Communication topology of matrix multiplication applications implemented using Cannon's algorithm (left) and MapReduce programming model (right)

We have implemented matrix multiplication applications using Hadoop and CGL-MapReduce by adopting a row/column decomposition approach to split the matrices. To clarify our algorithm, let's consider an example where two input matrices A and B produce matrix C, as the result of the multiplication process. We split the matrix B into a set of column blocks and the matrix A into a set of row blocks. In each iteration, all the map tasks consume two inputs: (i) a column block of matrix B, and (ii) a row block of matrix A; collectively, they produce a row block of the resultant matrix C. The column block associated with a particular map task is fixed throughout the computation while the row blocks are changed in each iteration. However, in Hadoop's programming model (typical MapReduce model), there is no way to specify this behavior and hence, it loads both the column block and the row block in each iteration of the computation. CGL-MapReduce supports the notion of long running map/reduce tasks where these tasks are allowed to retain static data in memory across

invocations, yielding better performance for iterative MapReduce computations. The communication pattern of this application is shown in Fig. 1(right).

4 Evaluations and Analysis

For our evaluations, we used two different compute clusters (details are shown in Table 3). DryadLINQ applications are run on the cluster Ref A while Hadoop, CGL-MapReduce, and MPI applications are run on the cluster Ref B. We measured the performance (average running time with varying input sizes) of these applications and then we calculated the overhead introduced by different parallel runtimes using the following formula, in which P denotes the number of parallel processes (map tasks) used and T denotes time as a function of the number of parallel processes used. T(1) is the time it takes when the task is executed using a single process. T(P) denotes the time when an application is executed using P number of parallel processes (For the results in Fig. 2 to Fig. 5, we used 64 CPU cores and hence the P=64). The results of these analyses are shown in Fig. 2 –5. Most applications have running times in minutes range and we noticed that the fluctuations in running time are less than 5% for most cloud runtimes. The average times shown in figures are calculated using the results of 5 repeated runs of the applications. We used Hadoop release 0.20, the academic release of DryadLINQ (Note: The academic release of Dryad only exposes the DryadLINQ API for programmers. Therefore, all our implementations are written using DryadLINQ although it uses Dryad as the underlying runtime).

$$\text{Overhead} = [P * T(P) - T(1)]/T(1). \quad (1)$$

Table 3. Different computation clusters used for the analyses

Cluster Ref	# Nodes used /Total CPU cores	CPU	Memory	Operating System
Ref A	8/64	2x Intel(R) Xeon(R) CPU L5420 2.50GHz	16GB	Windows Server 2008 – 64 bit HPC Edition (Service Pack 1)
Ref B	8/64	2 x Intel(R) Xeon(R) CPU L5420 2.50GHz	32GB	Red Hat Enterprise Linux Server release 5.3 - 64 bit

All three cloud runtimes work competitively well for the CAP3 application. In the Hadoop implementation of HEP data analysis, we kept the input data in a high performance parallel file system rather than in the HDFS because the analysis scripts written in ROOT could not access data from HDFS. This causes Hadoop's *map* tasks to access data remotely resulting lower performance compared to DryadLINQ and CGL-MapReduce implementations, which access input files from local disks. Both DryadLINQ and Hadoop show higher overheads for Kmeans clustering application,

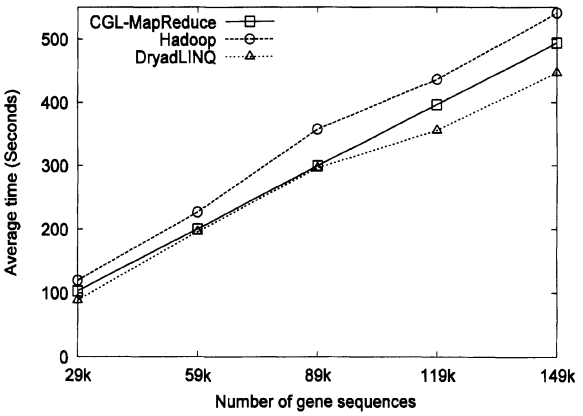


Fig. 2. Performance of the Cap3 application

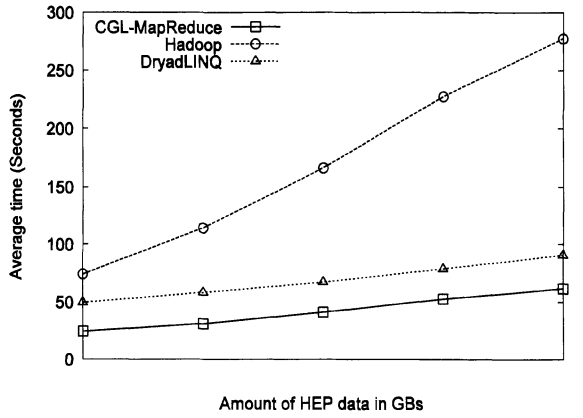


Fig. 3. Performance of HEP data analysis applications

and Hadoop shows higher overheads for the Matrix multiplication application. CGL-MapReduce shows a close performance to the MPI for large data sets in the case of Kmeans clustering and matrix multiplication applications, highlighting the benefits of supporting iterative computations and the faster data communication mechanism in the CGL-MapReduce.

From these results, it is clearly evident that the cloud runtimes perform competitively well for both the *Map-only* and the *MapReduce style* applications. However, for iterative and complex classes of applications, cloud runtimes show considerably high overheads compared to the MPI versions of the same applications, implying that, for these types of applications, we still need to use high performance parallel runtimes or use alternative approaches. (Note: The negative overheads observed in the matrix multiplication application are due to the better utilization of a cache by the parallel application than the single process version). These observations lead us to the next phase of our research.

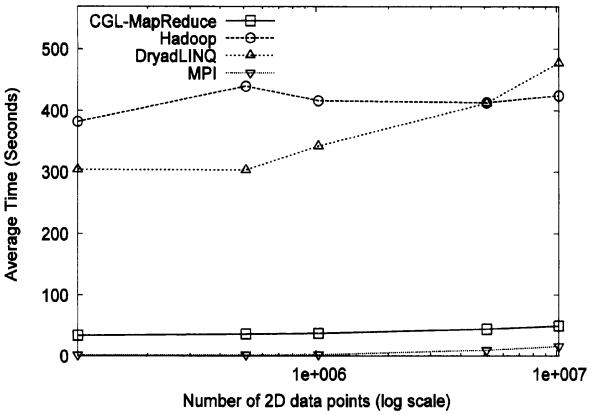


Fig. 4. Performance of different implementations of Kmeans Clustering application (Note: X axis is in log scale)

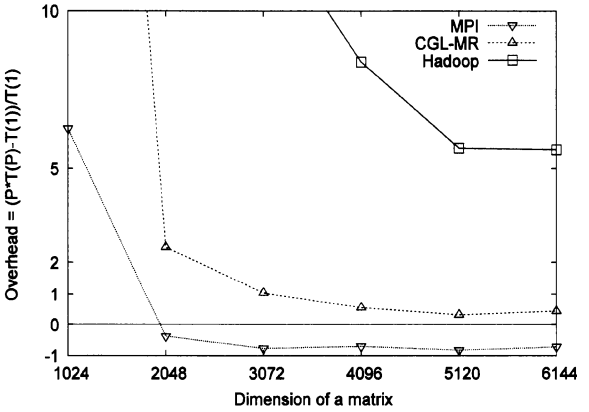


Fig. 5. Overhead induced by different parallel programming runtimes for the matrix multiplication application (8 nodes are used)

5 Performance of MPI on Clouds

After the previous observations, we analyzed the performance implications of cloud for parallel applications implemented using MPI. Specifically, we were trying to find the overhead of virtualized resources, and understand how applications with different communication-to-computation (C/C) ratios perform on cloud resources. We also evaluated different CPU core assignment strategies for VMs in order to understand the performance of VMs on multi-core nodes.

Commercial cloud infrastructures do not allow users to access the bare hardware nodes, in which the VMs are deployed, a must-have requirement for our analysis. Therefore, we used a Eucalyptus-based cloud infrastructure deployed at our university for this analysis. With this cloud infrastructure, we have complete access to both virtual machine instances and the underlying bare-metal nodes, as well as the help of

the administrators; as a result, we could deploy different VM configurations allocating different CPU cores to each VM. Therefore, we selected the above cloud infrastructure as our main test bed.

For our evaluations, we selected three MPI applications with different communication and computation requirements, namely, (i) the Matrix multiplication, (ii) Kmeans clustering, and (iii) the Concurrent Wave Equation solver. Table 4 highlights the key characteristics of the programs that we used for benchmarking.

Table 4. Computation and communication complexities of the different MPI applications used

Application	Matrix multiplication	Kmeans Clustering	Concurrent Wave Equation
Description	Implements Cannon's Algorithm Assume a rectangular process grid (Fig.1- left)	Implements Kmeans Clustering Algorithm Fixed number of iterations are performed in each test	A vibrating string is decomposed (split) into points, and each MPI process is responsible for updating the amplitude of a number of points over time.
Grain size (n)	Number of points in a matrix block handled by each MPI process	Number of data points handled by a single MPI process	Number of points handled by each MPI process
Communication Pattern	Each MPI process communicates with its neighbors in both row wise and column wise	All MPI processes send partial clusters to one MPI process (rank 0). Rank 0 distribute the new cluster centers to all the nodes	In each iteration, each MPI process exchanges boundary points with its nearest neighbors
Computation per MPI process	$O((\sqrt{n})^3)$	$O((\sqrt{n})^3)$	$O(n)$
Communication per MPI process	$O((\sqrt{n})^2)$	$O(1)$	$O(1)$
C/C	$O\left(\frac{1}{\sqrt{n}}\right)$	$O\left(\frac{1}{n}\right)$	$O\left(\frac{1}{n}\right)$
Message Size	$(\sqrt{n})^2 = n$	D – Where D is the number of cluster centers.	Each message contains a double value
Communication routines used	*MPI_Sendrecv_replace()*	*MPI_Reduce()* *MPI_Bcast()*	*MPI_Sendrecv()*

6 Benchmarks and Results

The Eucalyptus (version 1.4) infrastructure we used is deployed on 16 nodes of an iDataplex cluster, each of which has 2 Quad Core Intel Xeon processors (for a total of 8 CPU cores) and 32 GB of memory. In the bare-metal version, each node runs a Red Hat Enterprise Linux Server release 5.2 (Tikanga) operating system. We used OpenMPI version 1.3.2 with gcc version 4.1.2. We then created a VM image from

this hardware configuration, so that we have a similar software environment on the VMs once they are deployed. The virtualization is based on Xen hypervisor (version 3.0.3). Both bare-metal and virtualized resources utilize giga-bit Ethernet connections.

When VMs are deployed using Eucalyptus, it allows configuring the number of CPU cores assigned to each VM image. For example, with 8 core systems, the CPU core allocation per VM can range from 8 cores to 1 core per VM, resulting in several different CPU core assignment strategies. In Amazon EC2 infrastructure, the standard instance type has ½ a CPU per VM instance [28]. In the current version of Eucalyptus, the minimum number of cores that we can assign for a particular VM instance is 1; hence, we selected five CPU core assignment strategies (including the bare-metal test) listed in Table 5.

Table 5. Different hardware/virtual machine configurations used for performance evaluations

Ref	Description	Number of CPU cores accessible to the virtual or bare-metal node	Amount of memory (GB) accessible to the virtual or bare-metal node	Number of virtual or bare-metal nodes deployed
BM	Bare-metal node	8	32	16
1-VM-8-core	1 VM instance per bare-metal node	8	30 (2GB is reserved for Dom0)	16
2-VM-4-core	2 VM instances per bare-metal node	4	15	32
4-VM-2-core	4 VM instances per bare-metal node	2	7.5	64
8-VM-1-core	8 VM instances per bare-metal node	1	3.75	128

We ran all the MPI tests, on all 5 hardware/VM configurations, and measured the performance and calculated speed-ups and overheads. We calculated two types of overheads for each application using formula (1). The total overhead induced by the virtualization and the parallel processing is calculated using the bare-metal single process time as T(1) in the formula (1). The parallel overhead is calculated using the single process time from a corresponding VM as T(1) in formula (1). The average times shown in figures are obtained using 60 repeated runs for each and every measurement.

In all the MPI tests we performed, we used the following invariant to select the number of parallel processes (MPI processes) for a given application.

$$\textit{Number of MPI processes} = \textit{Number of CPU cores used.} \quad (2)$$

For example, for the matrix multiplication application, we used only half the number of nodes (bare-metal or VMs) available to us, so that we have 64 MPI processes = 64 CPU cores. (This is mainly because the matrix multiplication application expects the MPI processes to be in a square grid, in contrast to a rectangular grid). For Kmeans clustering, we used all the nodes, resulting in a total of 128 MPI processes utilizing all 128 CPU cores. Some of the results of our analysis highlighting different characterizes we observe are shown in Fig. 6 through 13.

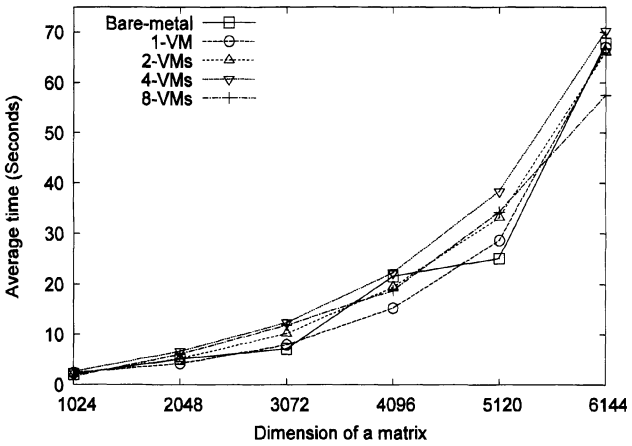


Fig. 6. Performance of the matrix multiplication application (Number of MPI processes = 64)

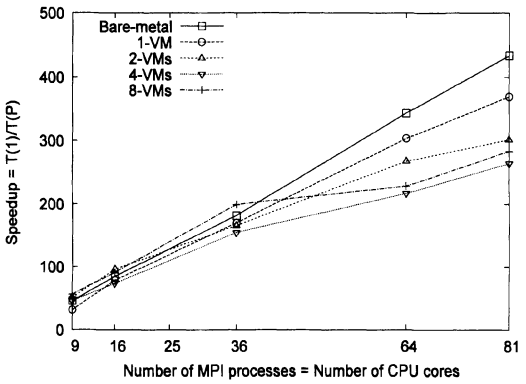


Fig. 7. Speed-up of the matrix multiplication application (Matrix size = 5184x5184)

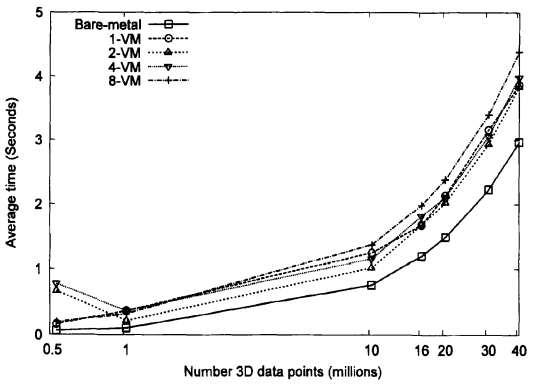


Fig. 8. Performance of Kmeans clustering (Number of MPI Processes = 128)

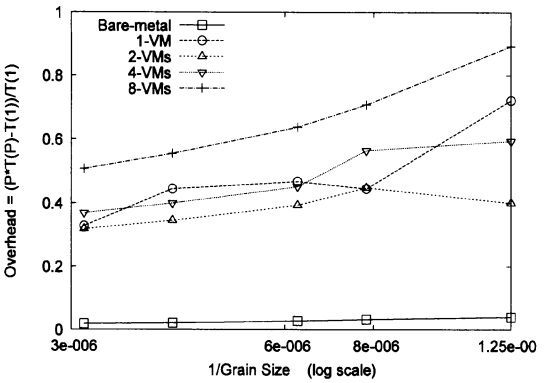


Fig. 9. Total overhead of the Kmeans clustering (Number of MPI Processes = 128)

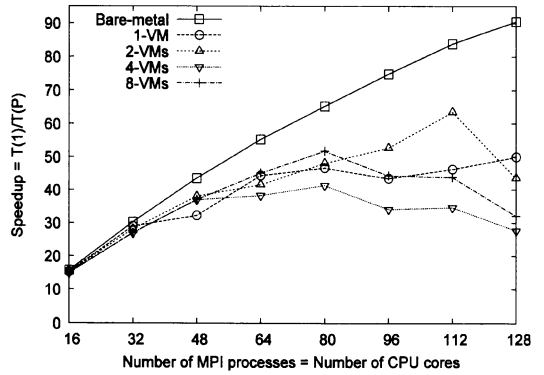


Fig. 10. Speed-up of the Kmeans clustering (Number of data points = 860160)

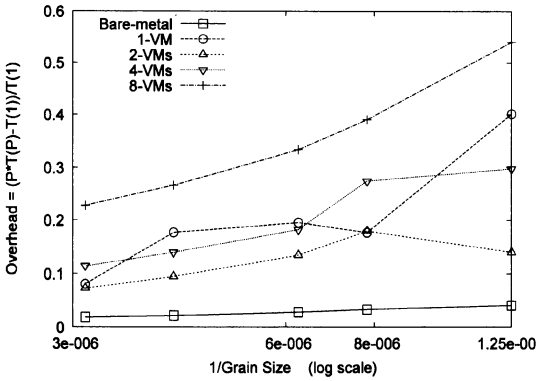


Fig. 11. Parallel overhead of the Kmeans clustering (Number of MPI Processes = 128)

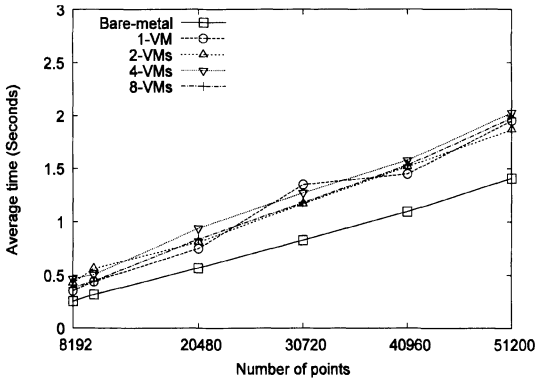


Fig. 12. Performance of the Concurrent Wave Solver (Number of MPI Processes = 128)

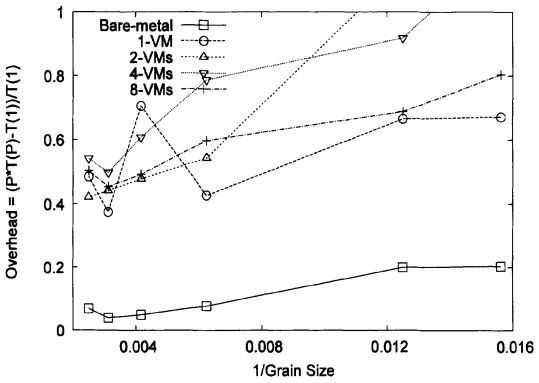


Fig. 13. Total overhead of the Concurrent Wave Solver (Number of MPI Processes = 128)

For the matrix multiplication, the graphs show very close performance characteristics in all the different hardware/VM configurations. As we expected, the bare-metal has the best performance and the speedup values, compared to the VM configurations (apart from the region close to the matrix size of 4096x4096 where the VM perform better than the bare-metal. We have performed multiple tests at this point, and found that it is a due to cache performances of the bare-metal node). After the bare-metal, the next best performance and speed-ups are recorded in the case of 1-VM per bare-metal node configuration, in which the performance difference is mainly due to the overhead induced by the virtualization. However, as we increase the number of VMs per bare-metal node, the overhead increases. At the 81 processes, 8-VMs per node configuration shows about a 34% decrease in speed-up compared to the bare-metal results.

In Kmeans clustering, the effect of virtualized resources is much clearer than in the case of the matrix multiplication. All VM configurations show a lower performance compared to the bare-metal configuration. In this application, the amount of data transferred between MPI processes is extremely low compared to the amount of data processed by each MPI process, and also, in relation to the amount of computations performed. Fig. 9 and Fig. 11 show the total overhead and the parallel overhead for

Kmeans clustering under different VM configurations. From these two calculations, we found that, for VM configurations, the overheads are extremely large for data set sizes of less than 10 million points, for which the bare-metal overhead remains less than 1 (<1 for all the cases). For larger data sets such as 40 million points, all overheads reached less than 0.5. The slower speed-up of the VM configurations (shown in Fig. 10) is due to the use of a smaller data set (~800K points) to calculate the speed-ups. The overheads are extremely large for this region of the data sizes, and hence, it resulted in lower speed-ups for the VMs.

Concurrent wave equation splits a number of points into a set of parallel processes, and each parallel process updates its portion of the points in some number of steps. An increase in the number of points increases the amount of the computations performed. Since we fixed the number of steps in which the points are updated, we obtained a constant amount of communication in all the test cases, resulting in a C/C ratio of O(1/n). In this application also, the difference in performance between the VMs and the bare-metal version is clearer, and at the highest grain size the total overhead of 8-VMs per node is about 7 times higher than the overhead of the bare-metal configuration. The performance differences between the different VM configurations become smaller with the increase in grain size.

From the above experimental results, we can see that the applications with lower C/C ratios experience a slower performance in virtualized resources. When the amount of data transferred between MPI processes is large, as in the case of the matrix multiplication, the application is more susceptible to the bandwidth than the latency. From the performance results of the matrix multiplication, we can see that the virtualization has not affected the bandwidth considerably. However, all the other results show that the virtualization has caused considerable latencies for parallel applications, especially with smaller data transfer requirements. The effect on latency increases as we use more VMs in a bare-metal node.

According to the Xen para-virtualization architecture [6], *domU*s (VMs that run on top of Xen para-virtualization) are not capable of performing I/O operations by themselves. Instead, they communicate with *dom0* (privileged OS) via an event channel (interrupts) and the shared memory, and then the *dom0* performs the I/O operations on behalf of the *domU*s. Although the data is not copied between *domU*s and *dom0,* the *dom0* needs to schedule the I/O operations on behalf of the *domU*s. Fig. 14(top) and Fig. 14 (bottom) shows this behavior in 1-VM per node and 8-VMs per node configurations we used.

In all the above parallel applications we tested, the timing figures measured correspond to the time for computation and communication inside the applications. Therefore, all the I/O operations performed by the applications are network-dependent. From Fig. 14 (bottom), it is clear that *Dom0* needs to handle 8 event channels when there are 8-VM instances deployed on a single bare-metal node. Although the 8 MPI processes run on a single bare-metal node, since they are in different virtualized resources, each of them can only communicate via *Dom0*. This explains the higher overhead in our results for 8-VMs per node configuration. The architecture reveals another important feature as well - that is, in the case of 1-VM per node configuration, when multiple processes (MPI or other) that run in the same VM communicate with

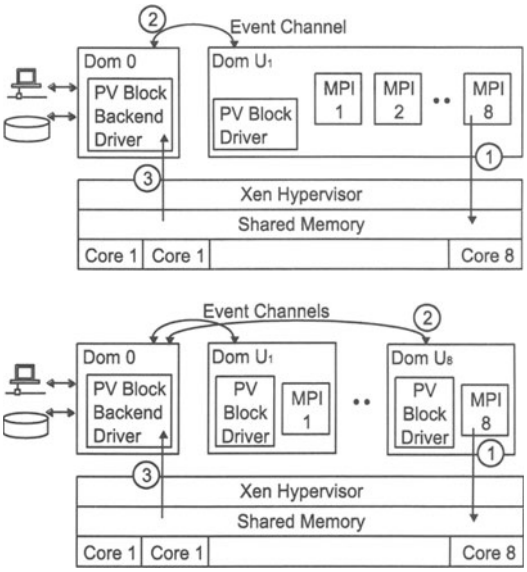


Fig. 14. Communication between dom0 and domU when 1-VM per node is deployed (top). Communication between dom0 and domUs when 8-VMs per node are deployed (bottom).

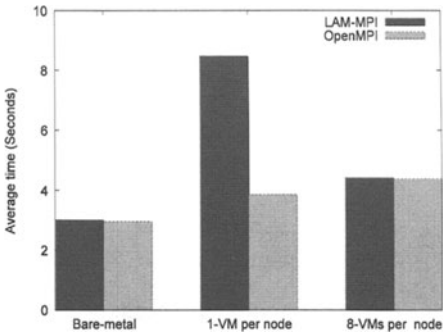


Fig. 15. LAM vs. OpenMPI (OMPI) under different VM configurations

each other via the network, all the communications must be scheduled by the *dom0*. This results higher latencies. We could verify this by running the above tests with LAM MPI (a predecessor of OpenMPI, which does not have improved support for in-node communications for multi-core nodes). Our results indicate that, with LAM MPI, the worst performance for all the test occurred when 1-VM per node is used. For example, Fig. 15 shows the performance of Kmeans clustering under bare-metal, 1-VM, and 8-VMs per node configurations. This observation suggests that, when using VMs with multiple CPUs allocated to each of them for parallel processing, it is better to utilize parallel runtimes, which have better support for in-node communication.

7 Conclusions and Future Work

From all the experiments we have conducted and the results obtained, we can come to the following conclusions on performing parallel computing using cloud and cloud technologies.

Cloud technologies work well for most pleasingly-parallel problems. Their support for handling large data sets, the concept of moving computation to data, and the better quality of services provided such as fault tolerance and monitoring, simplify the implementation details of such problems over the traditional systems.

Although cloud technologies provide better quality of services such fault tolerance and monitoring, their overheads are extremely high for parallel applications that require complex communication patterns and even with large data sets, and these overheads limit the usage of cloud technologies for such applications. It may be possible to find more "cloud friendly" parallel algorithms for some of these applications by adopting more coarse grained task/data decomposition strategies and different parallel algorithms. However, for other applications, the sheer performance of MPI style parallel runtimes is still desirable.

Enhanced MapReduce runtimes such as CGL-MapReduce allows iterative style applications to utilize the MapReduce programming model, while incurring minimal overheads compared to the other runtimes such as Hadoop and Dryad.

Handling large data sets using cloud technologies on cloud resources is an area that needs more research. Most cloud technologies support the concept of moving computation to data where the parallel tasks access data stored in local disks. Currently, it is not clear to us how this approach would work well with the VM instances that are leased only for the duration of use. A possible approach is to stage the original data in high performance parallel file systems or Amazon S3 type storage services, and then move to the VMs each time they are leased to perform computations.

MPI applications that are sensitive to latencies experience moderate-to-higher overheads when performed on cloud resources, and these overheads increase as the number of VMs per bare-hardware node increases. For example, in Kmeans clustering, 1-VM per node shows a minimum of 8% total overhead, while 8-VMs per node shows at least 22% overhead. In the case of the Concurrent Wave Equation Solver, both these overheads are around 50%. Therefore, we expect the CPU core assignment strategies such as ½ of a core per VM to produce very high overheads for applications that are sensitive to latencies.

Improved virtualization architectures that support better I/O capabilities, and the use of more latency insensitive algorithms would ameliorate the higher overheads in some of the applications. The former is more important as it is natural to run many VMs on future many core CPU architectures.

Applications those are not susceptible to latencies, such as applications that perform large data transfers and/or higher Communication/Computation ratios, show minimal total overheads in both bare-metal and VM configurations. Therefore, we expect that the applications developed using cloud technologies will work fine with cloud resources, because the milliseconds-to-seconds latencies that they already have under the MapReduce model will not be affected by the additional overheads introduced by the virtualization. This is also an area we are currently investigating. We are also building applications (biological DNA sequencing) whose end to end

implementation from data processing to filtering (data-mining) involves an integration of MapReduce and MPI.

Acknowledgements

We would like to thank Joe Rinkovsky and Jenett Tillotson from IU UITS for their dedicated support in setting up a private cloud infrastructure and helping us with various configurations associated with our evaluations.

References

1. Amazon Elastic Compute Cloud (EC2), http://aws.amazon.com/ec2/
2. Amazon Simple Storage Service (S3), http://aws.amazon.com/s3/
3. GoGrid Cloud Hosting, http://www.gogrid.com/
4. Keahey, K., Foster, I., Freeman, T., Zhang, X.: Virtual Workspaces: Achieving Quality of Service and Quality of Life in the Grid. Scientific Programming Journal 13(4), 265–276 (2005); Special Issue: Dynamic Grids and Worldwide Computing
5. Nurmi, D., Wolski, R., Grzegorczyk, C., Obertelli, G., Soman, S., Youseff, L., Zagorodnov, D.: The Eucalyptus Open-source Cloud-computing System. In: CCGrid 2009: the 9th IEEE International Symposium on Cluster Computing and the Grid, Shanghai, China (2009)
6. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. In: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, SOSP 2003, pp. 164–177. ACM, New York (2003), http://doi.acm.org/10.1145/945445.945462
7. Apache Hadoop, http://hadoop.apache.org/core/
8. Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: Distributed data-parallel programs from sequential building blocks. In: European Conference on Computer Systems (2007)
9. Yu, Y., Isard, M., Fetterly, D., Budiu, M., Erlingsson, U., Gunda, P., Currey, J.: DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In: Symposium on Operating System Design and Implementation (OSDI), San Diego, CA (2008)
10. Ekanayake, J., Pallickara, S., Fox, G.: MapReduce for Data Intensive Scientific Analysis. In: Fourth IEEE International Conference on eScience, Indianapolis, pp. 277–284 (2008)
11. Huang, X., Madan, A.: CAP3: A DNA Sequence Assembly Program. Genome Research 9(9), 868–877 (1999)
12. Hartigan, J.: Clustering Algorithms. Wiley, Chichester (1975)
13. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. ACM Commun. 51, 107–113 (2008)
14. MPI (Message Passing Interface), http://www-unix.mcs.anl.gov/mpi/
15. Dongarra, J., Geist, A., Manchek, R., Sunderam, V.: Integrated PVM framework supports heterogeneous network computing. Computers in Physics 7(2), 166–175 (1993)
16. Ludäscher, B., Altintas, I., Berkley, C., Higgins, D., Jaeger-Frank, E., Jones, M., Lee, E., Tao, J., Zhao, Y.: Scientific Workflow Management and the Kepler System. Concurrency and Computation: Practice & Experience (2005)

17. Hull, D., Wolstencroft, K., Stevens, R., Goble, C., Pocock, M., Li, P., Oinn, T.: Taverna: a tool for building and running workflows of services. Nucleic Acids Research (Web Server issue), W729 (2006)
18. Raicu, I., Zhao, Y., Dumitrescu, C., Foster, I., Wilde, M.: Falkon: a Fast and Light-weight tasK executiON framework. In: Proceedings of the ACM/IEEE Conference on Supercomputing, SC 2007, Nevada, ACM, New York (2007), http://doi.acm.org/10.1145/1362622.1362680
19. Pallickara, S., Pierce, M.: SWARM: Scheduling Large-Scale Jobs over the Loosely-Coupled HPC Clusters. In: Fourth IEEE International Conference on eScience, pp. 285–292 (2008)
20. Frey, J.: Condor DAGMan: Handling Inter-Job Dependencies, http://www.bo.infn.it/calcolo/condor/dagman/
21. Foster, I.: The Anatomy of the Grid: Enabling Scalable Virtual Organizations. In: Proceedings of the 7th international Euro-Par Conference Manchester on Parallel Processing (2001)
22. Ghemawat, S., Gobioff, H., Leung, S.: The Google file system. SIGOPS Oper. Syst. Rev. 37(5), 29–43 (2003), http://doi.acm.org/10.1145/1165389.945450
23. Pallickara, S., Fox, G.: NaradaBrokering: A Distributed Middleware Framework and Architecture for Enabling Durable Peer-to-Peer Grids. In: Endler, M., Schmidt, D.C. (eds.) Middleware 2003. LNCS, vol. 2672, pp. 41–61. Springer, Heidelberg (2003)
24. Gu, Y., Grossman, R.: Sector and Sphere: The Design and Implementation of a High Performance Data Cloud. Philosophical Transactions A Special Issue associated with the UK e-Science All Hands Meeting (2008)
25. Moretti, C., Bui, H., Hollingsworth, K., Rich, B., Flynn, P., Thain, D.: All-Pairs: An Abstraction for Data Intensive Computing on Campus Grids. IEEE Transactions on Parallel and Distributed Systems (2009)
26. Youseff, L., Wolski, R., Gorda, B., Krintz, C.: Evaluating the Performance Impact of Xen on MPI and Process Execution For HPC Systems. In: Proceedings of the 2nd international Workshop on Virtualization Technology in Distributed Computing. IEEE Computer Society, Washington (2006), http://dx.doi.org/10.1109/VTDC.2006.4
27. Constantinos, E., Hill, N.: Cloud Computing for parallel Scientific HPC Applications: Feasibility of Running Coupled Atmosphere-Ocean Climate Models on Amazon's EC2. In: Cloud Computing and Its Applications, Chicago, IL (2008)
28. Walker, E.: benchmarking Amazon EC2 for high-performance scientific computing, http://www.usenix.org/publications/login/2008-10/openpdfs/walker.pdf
29. Gavrilovska, A., Kumar, S., Raj, K., Gupta, V., Nathuji, R., Niranjan, A., Saraiya, P.: High-Performance Hypervisor Architectures: Virtualization in HPC Systems. In: 1st Workshop on System-level Virtualization for High Performance Computing (2007)
30. Fox, G., Bae, S., Ekanayake, J., Qiu, X., Yuan, H.: Parallel Data Mining from Multicore to Cloudy Grids. In: High Performance Computing and Grids workshop (2008)
31. Johnsson, S., Harris, T., Mathur, K.: Matrix multiplication on the connection machine. In: Proceedings of the 1989 ACM/IEEE Conference on Supercomputing, Supercomputing 1989, pp. 326–332. ACM, New York (1989), http://doi.acm.org/10.1145/76263.76298