Chapter 12

High-Performance Parallel Computing with Cloud and Cloud Technologies

Jaliya Ekanayake, Xiaohong Qiu, Thilina Gunarathne, Scott Beason, and Geoffrey Fox

Contents

12.1 Introduction 276
12.2 Cloud Technologies 278
12.2.1 Hadoop 278
12.2.2 Dryad and DryadLINQ 279
12.2.3 CGL-MapReduce 279
12.2.4 MPI 280
12.3 Programming Models 281
12.4 Data Analyses Applications 284
12.4.1 CAP3—Sequence Assembly Program 284
12.4.2 High-Energy Physics 285
12.4.3 Iterative MapReduce—Kmeans Clustering and Matrix Multiplication 285

K10175_C012.indd 275 3/2/2010 6:50:00 PM

12.4.4 ALU Sequencing Studies 287
12.4.4.1 ALU Clustering 287
12.4.4.2 Smith–Waterman Dissimilarities 287
12.4.4.3 The $O(N^2)$ Factor of 2 and Structure of Processing Algorithm 288
12.4.4.4 Dryad Implementation 288
12.4.4.5 MPI Implementation 289
12.5 Evaluations 290
12.5.1 Introduction 290
12.5.2 CAP3 and Particle Physics Case Studies 290
12.5.3 Kmeans and Matrix Multiplication Case Studies 293
12.5.4 ALU Sequence Analysis Case Study 294
12.5.4.1 Performance of Smith–Waterman–Gotoh Algorithm 294
12.5.4.2 Threaded Implementation 294
12.6 Performance of MPI on Clouds 295
12.6.1 Benchmarks and Results 296
12.7 Conclusions and Future Work 305
Acknowledgments 306
References 307

12.1 Introduction

Cloud and cloud technologies are two broad categories of technologies related to the general notion of Cloud Computing. By "cloud," we refer to a collection of infrastructure services, such as Infrastructure as a service (IaaS) and Platform as a service (PaaS), provided by various organizations where virtualization plays a key role. By "cloud technologies," we refer to various cloud runtimes, such as Hadoop (ASF, *core*, 2009a), Dryad (Isard et al. 2007), and other MapReduce (Dean and Ghemawat 2008) frameworks, and also the storage and communication frameworks, such as Hadoop Distributed File System (HDFS) and Amazon S3 (Amazon 2009).

The introduction of commercial cloud infrastructure services, such as Amazon EC2, GoGrid (ServePath 2009), and ElasticHosts (ElasticHosts 2009), has allowed users to provision compute clusters fairly easily and quickly, by paying a monetary value for the duration of their usages of the resources. The provisioning of resources happens in minutes, as opposed to 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, another feature that is hard to achieve with traditional infrastructures. The availability of open-source cloud infrastructure softwares, such as Nimbus (Keahey et al. 2005) and Eucalyptus (Nurmi et al. 2009), and open-source virtualization software stacks,

such as Xen Hypervisor (Barham et al. 2003), 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.

Among the many applications that benefit from cloud and cloud technologies, the data/compute-intensive applications are the most important. The deluge of data and the highly compute-intensive applications found in many domains, such as particle physics, biology, chemistry, finance, and information retrieval, mandate the use of large computing infrastructures and parallel processing to achieve considerable performance gains in analyzing data. The addition of cloud technologies creates new trends in performing parallel computing. An employee in a publishing company who needs to convert a document collection, terabytes in size, to a different format can do so by implementing a MapReduce computation using Hadoop, and running it on leased resources from Amazon EC2 in just a few hours. A scientist who needs to process a collection of gene sequences using the CAP3 (Huang and Madan 1999) software can use virtualized resources leased from the university's private cloud infrastructure and Hadoop. In these use cases, the amount of coding that the publishing agent and the scientist need to perform is minimal (as each user simply needs to implement a *map* function), and the MapReduce infrastructure handles many aspects of the parallelism.

Although the above examples are successful use cases for applying cloud and cloud technologies for parallel applications, through our research, we have found that there are limitations in using current cloud technologies for parallel applications that require complex communication patterns or require faster communication mechanisms. For example, Hadoop and Dryad implementations of Kmeans clustering applications, which perform an iteratively refining clustering operation, show higher overheads compared to implementations of MPI or CGL-MapReduce AQ2 (Ekanayake et al. 2008)—a streaming-based MapReduce runtime developed by us. These observations raise questions: What applications are best handled by cloud technologies? What overheads do they introduce? Are there any alternative approaches? Can we use traditional parallel runtimes such as MPI in cloud? If so, what overheads does it have? These are some of the questions we try to answer through our research.

In Section 12.1, we give a brief introduction of the cloud technologies, and in Section 12.2, we discuss with examples the basic functionality supported by these cloud runtimes. Section 12.3 discusses how these technologies map into programming models. We describe the applications used to evaluate and test technologies in Section 12.4. The performance results are discussed in Section 12.5. In Section 12.6, we present details of an analysis we have performed to understand the performance implications of virtualized resources for parallel MPI applications. Note that we use MPI running on non-VMs in Section 12.5 for comparison with cloud technologies. We present our conclusions in Section 12.7.

12.2 Cloud Technologies

Cloud technologies such as MapReduce and Dryad have created new trends in parallel programming. The support for handling large data sets, the concept of moving computation to data, and the better quality of services provided by the cloud technologies make them a favorable choice to solve large-scale data/compute-intensive problems.

The granularity of the parallel tasks in these programming models lies in between the fine-grained parallel tasks that are used in message-passing infrastructures such as PVM (Dongarra et al. 1993) and MPI (Forum n.d.), and coarse-grained jobs in workflow frameworks such as Kepler (Ludscher et al. 2006) and Taverna (Hull et al. 2006), in which the individual tasks could themselves be parallel applications written in MPI. Unlike the various communication constructs available in MPI, which can be used to create a wide variety of communication topologies for parallel programs, in MapReduce, the "map→reduce" is the only communication construct available. However, our experience shows that most *composable* applications can easily be implemented using the MapReduce programming model. Dryad supports parallel applications that resemble Directed Acyclic Graphs (DAGs), in which the vertices represent computation units, and the edges represent communication channels between different computation units.

AQ3

In traditional approaches, once parallel applications are developed, they are executed on compute clusters, supercomputers, or grid infrastructures (Foster 2001), 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 resource in order for them to be executed. These infrastructures are highly efficient in performing compute-intensive parallel applications. However, when the volume 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) (Ghemawat et al. 2003), Hadoop and HDFS, 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 the 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.

12.2.1 Hadoop

Apache Hadoop has a similar architecture to Google's MapReduce runtime, where it accesses data via HDFS, which maps all the local disks of the compute nodes to

a single file system hierarchy, allowing the data to be dispersed across all the data/computing nodes. HDFS also replicates the data on multiple nodes so that failures of any nodes containing a portion of the data will not affect the computations that use that data. Hadoop schedules the MapReduce computation tasks depending on the data locality, improving the overall I/O (input/output) bandwidth. The outputs of the *map* tasks are first stored in local disks until later, when the *reduce* tasks access them (pull) via HTTP connections. Although this approach simplifies the fault-handling mechanism in Hadoop, it adds a significant communication overhead to the intermediate data transfers, especially for applications that produce small intermediate results frequently.

12.2.2 Dryad and DryadLINQ

Dryad is a distributed execution engine for coarse-grained data parallel applications. It combines the MapReduce programming style with dataflow graphs to solve the computation tasks. Dryad considers computation tasks as DAGs, where the vertices represent computation tasks and the edges act as communication channels over which the data flows from one vertex to another. The data is stored in (or partitioned to) local disks via the Windows shared directories and metadata files, and Dryad schedules the execution of vertices depending on the data locality. (Note: The academic release of Dryad only exposes the DryadLINQ (Yu et al. 2008) API for programmers. Therefore, all our implementations are written using DryadLINQ, although it uses Dryad as the underlying runtime.) Dryad also stores the output of vertices in local disks, and the other vertices that depend on these results access them via the shared directories. This enables Dryad to re-execute failed vertices, a step that improves fault tolerance in the programming model.

12.2.3 CGL-MapReduce

CGL-MapReduce is a lightweight MapReduce runtime that incorporates several improvements to the MapReduce programming model, such as (1) faster intermediate data transfer via a pub/sub broker network, (2) support for long-running *map/reduce* tasks, and (3) efficient support for iterative MapReduce computations. The architecture of CGL-MapReduce is shown in Figure 12.1 (left). AQ4

The use of streaming enables CGL-MapReduce to send the intermediate results directly from its producers to its consumers, and eliminates the overhead of the file-based communication mechanisms adopted by both Hadoop and Dryad. The support for long-running *map/reduce* tasks enables configuring and reusing *map/reduce* tasks in the case of iterative MapReduce computations, and eliminates the need for reconfiguring or reloading static data in each iteration. This feature comes with the distinction of "static data" and "dynamic data" that we support in CGL-MapReduce. We refer to any data set that is static throughout the computation as "static data," and the data that is changing over the computation as

Figure 12.1 (Left) Components of CGL-MapReduce. (Right) Different synchronization and intercommunication mechanisms used by parallel runtimes.

"dynamic data." Although this distinction is irrelevant to the MapReduce computations that have only one *map* phase followed by a *reduce* phase, it is extremely important for iterative MapReduce computations, in which the map tasks need to access a static (fixed) data again and again. Figure 12.1 (right) highlights the synchronization and communication characteristics of Hadoop, Dryad, CGL-MapReduce, and MPI.

Additionally, CGL-MapReduce supports the distribution of smaller variable data sets to all the *map* tasks directly, a functionality similar to *MPI_Bcast*() that is often found to be useful in many data analysis applications. Hadoop provides a similar feature via its distributed cache, in which a file or data is copied to all the compute nodes. Dryad provides a similar feature by allowing applications to add resources (files) that will be accessible to all the vertices. With the above features in place, CGL-MapReduce can be used to implement iterative MapReduce computations efficiently. In CGL-MapReduce, data partitioning and distribution is left to the users to handle, and it reads data from shared file systems or local disks. Although the use of streaming makes CGL-MapReduce highly efficient, implementing fault tolerance with this approach is not as straightforward as it is in Hadoop or Dryad. We plan to implement fault tolerance in CGL-MapReduce by re-execution of failed *map* tasks and redundant execution of *reduce* tasks.

12.2.4 MPI

MPI, the de facto standard for parallel programming, is a language-independent communications protocol that uses a message-passing paradigm to share the data and state among a set of cooperative processes running on a distributed memory system. MPI specification (Forum, MPI) defines a set of routines to support various parallel programming models, such as point-to-point communication, collective communication, derived data types, and parallel I/O operations.

Most MPI runtimes are deployed in computation clusters where a set of compute nodes are connected via a high-speed network connection yielding very low communication latencies (typically in microseconds). MPI processes typically have a direct mapping to the available processors in a compute cluster or to the processor cores in the case of multi-core systems. We use MPI as the baseline performance measure for the various algorithms that are used to evaluate the different parallel programming runtimes. Table 12.1 summarizes the different characteristics of Hadoop, Dryad, CGL-MapReduce, and MPI.

12.3 Programming Models

When analyzing applications written in the MapReduce programming model, we can identify three basic execution units, namely, (1) map-only, (2) map-reduce, and (3) iterative map-reduce. Complex applications can be built by combining these three basic execution units under the MapReduce programming model. Table 12.2 shows the data/computation flow of these three basic execution units, along with examples.

In the MapReduce programming model, the tasks that are being executed at a given phase have similar executables and similar input and output operations. With zero *reduce* tasks, the MapReduce model reduces to a map-only model, which can be applied to many "embarrassingly parallel" applications. Software systems such as batch queues, Condor (Condor 2009), Falkon (Raicu et al. 2007), and SWARM (Pallickara and Pierce 2008) all provide similar functionality by scheduling large numbers of individual maps/jobs. Applications that can utilize a "reduction" or an "aggregation" operation can use both phases of the MapReduce model, and, depending on the "associativity" and "transitivity" nature of the reduction operation, multiple reduction phases can be applied to enhance the parallelism. For example, in a histogramming operation, the partial histograms can be combined in any order and in any number of steps to produce a final histogram.

The "side effect–free" nature of the MapReduce programming model does not promote iterative MapReduce computations. Each of the *map* and *reduce* tasks are considered as atomic execution units with no state shared in between executions. In parallel runtimes, such as those of the MPI, the parallel execution units live throughout the entire life of the program; hence, the state of a parallel execution unit can be shared across invocations. We propose an intermediate approach to develop MapReduce computations. In our approach, the *map/reduce* tasks are still considered free from side effects, but the runtime allows configuring and reusing the *map/reduce* tasks. Once configured, the runtime caches the *map/reduce* tasks. This way, both *map* and *reduce* tasks can keep the static data in memory, and can be called iteratively without loading the static data repeatedly.

Hadoop supports configuring the number of *reduce* tasks, which enables the user to create "map-only" applications by using zero *reduce* tasks. Hadoop can be

Table 12.1 Comparison of Features Supported by Different Parallel Programming Runtimes

Feature	*Hadoop*	*Dryad*	*CGL-MapReduce*	*MPI*
Programming model	MapReduce	DAG-based execution flows	MapReduce with a *Combine* phase	Variety of topologies constructed using the rich set of parallel constructs
Data handling	HDFS	Shared directories/local disks	Shared file system/local disks	Shared file systems
Intermediate data communication	HDFS/point to point via HTTP	Files/TCP pipes/ shared memory FIFO	Content distribution network (NaradaBrokering (Pallickara and Fox 2003))	Low-latency communication channels
Scheduling	Data locality/rack aware	Data locality/ network topology–based runtime graph optimizations	Data locality	Available processing capabilities
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)	Program-level check-pointing OMPI (Gabriel et al. 2004), FT MPI
Monitoring	Monitoring support of HDFS, monitoring MapReduce computations	Monitoring support for execution graphs	Programming interface to monitor the progress of jobs	Minimal support for task-level monitoring
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	C, C++, Fortran, Java, C#

Table 12.2 Three Basic Execution Units under the MapReduce Programming Model

Map-Only	*Map-Reduce*	*Iterative Map-Reduce*
Cap3 analysis (we will discuss more about this later)	HEP data analysis (we will discuss more about this later)	Expectation maximization algorithms
Converting a collection of documents to different formats, processing a collection of medical images, and brute-force searches in cryptography; parametric sweeps	*Histogramming* operations, distributed search, and distributed sorting; information retrieval	Kmeans clustering, matrix multiplication

used to implement iterative MapReduce computations, but the framework does not provide additional support to implement them efficiently. The CGL-MapReduce supports all the above three execution units, and the user can develop applications with multiple stages of MapReduce by combining them in any order. Dryad execution graphs resembling the above three basic units can be generated using DryadLINQ operations. DryadLINQ adds the LINQ programming features to Dryad where the user can implement various data analysis applications using LINQ queries, which will be translated to Dryad execution graphs by the compiler. However, unlike in the MapReduce model, Dryad allows the concurrent vertices to have different behaviors and different I/O characteristics, thus enabling a more workflow-style programming model. Dryad also allows multiple communication channels in between different vertices of the dataflow graph. Programming languages such as Swazall (Pike et al. 2005), introduced by Google for its MapReduce runtime, enable high-level language support for expressing MapReduce computations, and the Pig (ASF, *pig*, 2009b) available as a subproject of Hadoop allows query operations on large data sets.

Apart from these programming models, there are other software frameworks that one can use to perform data/compute-intensive analyses. Disco (Nokia 2009) is an open-source MapReduce runtime developed using a functional programming

language named Erlang (Ericsson 2009). The Disco architecture shares clear similarities with both Google and Hadoop MapReduce architectures. Sphere (Gu and Grossman 2009) is a framework that can be used to execute user-defined functions in parallel on data stored in a storage framework named Sector. Sphere can also perform MapReduce-style programs, and the authors compare its performance with Hadoop for tera-sort applications. All-Pairs (Moretti et al. 2009) is an abstraction that can be used to solve the 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 Hadoop and Dryad as well, and we discuss a similar problem in Section 12.4.4. We can also develop an efficient iterative MapReduce implementation using CGL-MapReduce to solve this problem. The algorithm is similar to the matrix multiplication algorithm that we explain in Section 12.4.3.

MPI and threads are two other programming models that can be used to implement parallel applications. MPI can be used to develop parallel applications in distributed memory architectures, whereas threads can be used in shared memory architectures, especially in multi-core nodes. The low-level communication constructs available in MPI allow users to develop parallel applications with various communication topologies involving fine-grained parallel tasks. The use of low-latency network connections between nodes enables applications to perform a large number of inter-task communications. In contrast, the next-generation parallel runtimes, such as MapReduce and Dryad, provide a small number of parallel constructs, such as "map-only," "map-reduce," "Select," "Apply," and "Join," and do not require high-speed communication channels. These constraints require adopting parallel algorithms that perform coarse-grained parallel tasks and less communication. The use of threads is a natural approach in shared memory architectures, where communication between parallel tasks reduces to the simple sharing of pointers via the shared memory. However, the operating system's support for user-level threads plays a major role in achieving better performances using multi-threaded applications. We will discuss the issues in using threads and MPI in more detail in Section 12.5.4.2.

12.4 Data Analyses Applications

12.4.1 CAP3—Sequence Assembly Program

CAP3 is a DNA sequence assembly program developed by Huang and Madan (1999) that performs several major assembly steps: These steps include computation of overlaps, construction of contigs, construction of multiple sequence alignments, and generation of consensus sequences to a given set of gene sequences. The program reads a collection of gene sequences from an input file (FASTA file format) and writes its output to several output files, as well as the standard output:

Input.fsa → CAP3 → Stdout + Other output files

The program structure of this application fits directly with the "map-only" basic execution unit, as shown in Table 12.2. We implemented a parallel version of CAP3 using Hadoop, CGL-MapReduce, and DryadLINQ. Each *map* task in Hadoop and in CGL-MapReduce calls the CAP3 executable as a separate process for a given input data file (the input "Value" for the *map* task), whereas in DryadLINQ, a "homomorphic Apply" operation calls the CAP3 executable on each data file in its data partition as a separate process. All the implementations move the output files to a predefined shared directory. This application resembles a common parallelization requirement, where an executable script, or a function in a special framework such as MATLAB® or R, needs to be executed on each input data item. The above approach can be used to implement all these types of applications using any of the above three runtimes.

12.4.2 High-Energy Physics

Next, we applied the MapReduce technique to parallelize a High-Energy Physics (HEP) data analysis application, and implemented it using Hadoop, CGL-MapReduce, and Dryad. The HEP data analysis application processes large volumes of data, and performs a histogramming operation on a collection of event files produced by HEP experiments. The details regarding the two MapReduce implementations and the challenges we faced in implementing them can be found in Ekanayake et al. (2008). In the DryadLINQ implementation, the input data files are first distributed among the nodes of the cluster manually. We developed a tool to perform the manual partitioning and distribution of the data. The names of the data files available in a given node were used as the data to the DryadLINQ program. Using a homomorphic "Apply" operation, we executed a ROOT-interpreted script on groups of input files in all the nodes. The output histograms of this operation were written to a predefined shared directory. Next, we used another "Apply" phase to combine these partial histograms into a single histogram using DryadLINQ.

12.4.3 Iterative MapReduce—Kmeans Clustering and Matrix Multiplication

Parallel applications that are implemented using message-passing runtimes can utilize various communication constructs to build diverse communication topologies. For example, a matrix multiplication application that implements Fox's Algorithm (Fox et al. 1987) and Cannon's Algorithm (Johnsson et al. 1989) assumes parallel processes to be in a rectangular grid. Each parallel process in the grid communicates with its left and top neighbors, as shown in Figure 12.2 (left). The current cloud runtimes, which are based on dataflow models such as MapReduce and AQ5

Figure 12.2 (Left) Communication topology of Cannon's Algorithm implemented using MPI. (Middle) Communication topology of matrix multiplication application based on MapReduce. (Right) Communication topology of Kmeans clustering implemented as a MapReduce application.

Dryad, do not support this behavior, in which the peer nodes communicate with each other. Therefore, implementing the above type of parallel applications using MapReduce or DryadLINQ requires adopting different algorithms.

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 us 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 process two inputs: (1) a column block of matrix B and (2) 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 (a typical MapReduce model), there is no way to specify this behavior. 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 the memory across invocations, yielding better performance for "iterative MapReduce" computations. The communication pattern of this application is shown in Figure 12.2 (middle).

Kmeans clustering (Macqueen 1967) is another application that performs iteratively refining computation. We also implemented Kmeans clustering applications using Hadoop, CGL-MapReduce, and DryadLINQ. In the two MapReduce implementations, each *map* task calculates the distances between all the data elements in its data partition and all the cluster centers produced during the previous run. It then assigns data points to these cluster centers, based on their Euclidian distances. The communication topology of this algorithm is shown in Figure 12.2 (right). Each *map* task produces partial cluster centers as the output; these are then

AQ6

combined at a *reduce* task to produce the current cluster centers. These current cluster centers are used in the next iteration, to find the next set of cluster centers. This process continues until the overall distance between the current cluster centers and the previous cluster centers reduces below a predefined threshold. The Hadoop implementation uses a new MapReduce computation for each iteration of the program, while CGL-MapReduce's long-running *map/reduce* tasks allow it to reuse *map/reduce* tasks. The DryadLINQ implementation uses various DryadLINQ operations, such as "Apply," "GroupBy," "Sum," "Max," and "Join," to perform the computation, and it also utilizes DryadLINQ's "loop unrolling" support to perform multiple iterations as a single-large query.

12.4.4 ALU Sequencing Studies

12.4.4.1 ALU Clustering

The ALU clustering problem (Batzer and Deininger 2002) is one of the most challenging problems for sequence clustering, because ALUs represent the largest repeat families in human genome. There are about 1 million copies of ALU sequences in human genome, in which most insertions can be found in other primates and only a small fraction (~7000) are human specific. This indicates that the classification of ALU repeats can be deduced solely from the 1 million human ALU elements. Notably, ALU clustering can be viewed as a classical case study for the capacity of computational infrastructures, because it is not only of great intrinsic biological interest, but also a problem of a scale that will remain as the upper limit of many other clustering problems in bioinformatics for the next few years, for example, the automated protein family classification for a few millions of proteins predicted from large metagenomics projects. AQ7

12.4.4.2 Smith–Waterman Dissimilarities AQ8

We identified samples of the human and chimpanzee ALU gene sequences using Repeatmasker (Smith et al. 2004) with Repbase Update (Jurka 2000). We have been gradually increasing the size of our projects with the current largest samples having 35,339 and 50,000 sequences, and these require a modest cluster, such as Tempest (768 cores), for processing in a reasonable time (a few hours, as shown in Section 12.5). Note from the discussion in Section 12.4.4.1 that we are aiming at supporting problems with a million sequences—quite practical today on TeraGrid, and equivalent facilities given basic analysis steps scale like $O(N^2)$. AQ9

We used an open-source version NAligner (Smith–Waterman software) of the Smith–Waterman–Gotoh (SW-G) algorithm (Smith and Waterman 1981, Gotoh 1982) modified to ensure low start-up effects by each thread processing large numbers (above a few hundreds) at a time. The memory bandwidth needed was reduced by storing data items in as few bytes as possible. AQ10

12.4.4.3 The $O(N^2)$ Factor of 2 and Structure of Processing Algorithm

The ALU sequencing problem shows a well-known factor-of-2 issue present in many $O(N^2)$ parallel algorithms, such as those in direct simulations of astrophysical stems. We initially calculate in parallel the distance, $D(i,j)$, between points (sequences) i and j. This is done in parallel over all processor nodes selecting criteria $i < j$ (or $j > i$ for the upper triangular case) to avoid calculating both $D(i,j)$ and the identical $D(j,i)$. This can require substantial file transfer, as it is unlikely that nodes requiring $D(i,j)$ in a later step will find that it was calculated on nodes where it is needed.

AQ11 For example, the MDS and PW (PairWise) clustering algorithms, described in Fox et al. (2008), require a parallel decomposition where each of N processes (MPI processes, threads) has $1/N$ of sequences, and for this subset $\{i\}$ of sequences stores in memory $D(\{i\},j)$ for all sequences j and the subset $\{i\}$ of sequences for which this node is responsible. This implies that we need $D(i,j)$ and $D(j,i)$ (which are equal) stored in different processors/disks. This is a well-known collective operation in MPI called either gather or scatter.

12.4.4.4 Dryad Implementation

We developed a DryadLINQ application to perform the calculation of pairwise SW-G distances for a given set of genes by adopting a coarse-grained task decomposition approach that requires minimum inter-process communication to ameliorate the higher communication and synchronization costs of the parallel runtime. To clarify our algorithm, let us consider an example where N gene AQ12 sequences produce a pairwise distance matrix of size $N \times N$. We decompose the computation task by considering that the resultant matrix groups the overall computation into a block matrix of size $D \times D$, where D is a multiple (>2) of the available computation nodes. Due to the symmetry of the distances $D(i,j)$ and $D(j,i)$, we only calculate the distances in the blocks of the upper triangle of the block matrix, as shown in Figure 12.3 (left). The blocks in the upper triangle are partitioned (assigned) to the available compute nodes, and an "Apply" operation is used to execute a function to calculate $(N/D) \times (N/D)$ distances in each block. After computing the distances in each block, the function calculates the transpose matrix of the resultant matrix, which corresponds to a block in the lower triangle, and writes both these matrices into two output files in the local file system. The names of these files and their block numbers are communicated back to the main program. The main program sorts the files based on their block numbers and performs another "Apply" operation to combine the files corresponding to a row of blocks in a single-large row block, as shown in Figure 12.3 (right).

Figure 12.3 Task decomposition (left) and DryadLINQ vertex hierarchy (right) of the DryadLINQ implementation of SW-G pairwise distance calculation application.

12.4.4.5 MPI Implementation

The MPI version of SW-G calculates pairwise distances using a set of either single- or multi-threaded processes. For N gene sequences, we need to compute half of the values (in the lower triangular matrix), which is a total of $M = N \times (N - 1)/2$ distances. At a high level, computation tasks are evenly divided among P processes and execute in parallel, namely, the computation workload per process is M/P. At a low level, each computation task can be further divided into subgroups and run in T concurrent threads. Our implementation is designed for flexible use of a shared memory multi-core system and distributed memory clusters (tight-coupled to medium-tight-coupled communication technologies, such threading and MPI). We provide options for any combinations of thread versus process versus node, as shown in Figure 12.4. The real computation workload per parallel unit is decided by $M/(T \times P \times \#$ nodes).

As illustrated in Figure 12.4, the data decomposition strategy runs a "space-filling curve through the lower triangular matrix" to produce equal numbers of pairs for each parallel unit such as process or thread. It is necessary to map indexes in each pairs group back to corresponding matrix coordinates (i,j) for constructing a full matrix later on. We implemented a special function, "PairEnumerator," as AQ13
the convertor. We tried to limit runtime memory usage for performance optimization. This is done by writing a triple of i,j and also writing the distance value of pairwise alignment to a stream writer, and the system flushes accumulated results to a local file periodically. As the final stage, individual files are merged to form a full distance matrix.

Figure 12.4 Task decomposition (left) and MPI (right) implementation of SW-G pairwise distance calculation application.

12.5 Evaluations

12.5.1 Introduction

For our evaluations, we used three compute clusters (details are given in Table 12.3) with two 32-node clusters having almost identical hardware configurations and one latest 32-node cluster of 24-core machines with Infiniband connections. DryadLINQ and the MPI application that performs SW-G computation were run on the Windows cluster (Ref B, Ref C), while Hadoop, CGL-MapReduce, and other MPI applications were run on the Linux cluster (Ref A). We measured the performance of these applications, and present the results in terms of parallel overhead defined for parallelism P by

$$f(P) = \frac{P \times T(P) - T(1)}{T(1)} \tag{12.1}$$

where

P denotes parallelism (e.g., processes, threads, and *map* tasks) used
T denotes time as a function of the number of parallel processes used

$T(1)$ is replaced in practice by $T(S)$, where S is the smallest number of processes that can run the job. We used Hadoop release 0.20, the academic release of DryadLINQ, Microsoft MPI, and OpenMPI (OMPI) version 1.3.2 for our evaluations.

12.5.2 CAP3 and Particle Physics Case Studies

The results of our performance measurements for CAP3 and particle physics are shown in Figures 12.5 through 12.8.

Table 12.3 Different Computation Clusters Used for the Analyses

Feature	*Linux Cluster (Ref A)*	*Windows Cluster (Ref B)*	*Windows Cluster (Ref C)*
# Node	32	32	32
CPU	Intel(R) Xeon(R) CPU L5420 2.50 GHz	Intel(R) Xeon(R) CPU L5420 2.50 GHz	Intel(R) Xeon(R) CPU E7450 2.40 GHz
# CPU/# cores	2/8	2/8	4/24
Total cores	256	256	768
Memory	32 GB	16 GB	48 GB
Disk	1 disk of Western Digital Caviar RE 160 GB SATA 7200	2 disks of 1000 GB (1 TB) Ultrastar A7K1000 7200	2 HP 146 GB 10K 2.5 SAS HP SP HDD
Network	Gigabit Ethernet	Gigabit Ethernet	20 Gbps Infiniband
Operating system	Red Hat Enterprise Linux Server release 5.3—64 bit	Windows Server Enterprise—64 bit	Windows Server 2008 HPC Edition (Service Pack 1)

Figure 12.5 Performance of the CAP3 application—average time (in s) against the number of gene reads processed.

Figure 12.6 Performance of the HEP data analysis application—average time (in s) against the amount of input data processed (in GB).

Figure 12.7 Overhead induced by different parallel programming runtimes for the Kmeans clustering application—overhead against the number of 2D data points clustered. (Note: Both axes are in log scale.)

From these results, it is clearly evident that the cloud runtimes perform competitively well for both "map-only-style" and "map-reduce-style" applications. In the HEP data analysis, both CGL-MapReduce and DryadLINQ access input data from local disks, where the data is partitioned and distributed beforehand. Currently, HDFS can be accessed using Java or C++ clients only, and the ROOT-interpretable scripts (ROOT—data analysis framework

Figure 12.8 Overhead induced by different parallel programming runtimes for the matrix multiplication application—overhead against the dimension of an input matrix.

developed at CERN) are not capable of accessing data from HDFS. Therefore, we placed the input data in the IU (Indiana University) Data Capacitor—a high-performance parallel file system based on the Lustre file system, which allows each *map* task in Hadoop to directly access data from this file system. AQ14 The performance results show that this dynamic data movement in the Hadoop implementation incurred considerable overhead to the computation, while the ability of reading input data from local disks gave significant performance improvement to both DryadLINQ and CGL-MapReduce, as compared to the Hadoop implementation.

12.5.3 Kmeans and Matrix Multiplication Case Studies

For an iterative class of applications, cloud runtimes show considerably high overheads, compared to the MPI and CGL-MapReduce versions of the same applications; the results shown in Figures 12.7 and 12.8 imply that, for these types of applications, we still need to use high-performance parallel runtimes or 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.) CGL-MapReduce shows a close performance closer to the MPI for large data sets in the case of Kmeans clustering and matrix multiplication applications, an outcome that highlights the benefits of supporting iterative computations and the faster data communication mechanism present in CGL-MapReduce.

12.5.4 ALU Sequence Analysis Case Study

12.5.4.1 Performance of Smith–Waterman–Gotoh Algorithm

We performed the Dryad and MPI implementations of ALU SW-G distance calculations on two large data sets and obtained the following results.

There is a short partitioning phase for DryadLINQ, and then both approaches calculate the distances and write these out to intermediate files, as discussed in Section 12.4. We note that the merge time is currently much longer for MPI than DryadLINQ, while the initial steps are significantly faster for MPI. However, the total times in Table 12.4 indicate that both MPI and DryadLINQ implementations perform well for this application, with MPI a few percent faster with current implementations. As expected, the times scale proportionally to the square of the AQ15 number of distances. On 744 cores, the average time of 0.0067 ms/pair that corresponds to roughly 5 ms/pair calculated per core is used. The coarse-grained Dryad application performs competitively with the tightly synchronized MPI application. It proves once more the applicability of the cloud technologies for the composable applications.

12.5.4.2 Threaded Implementation

In Section 12.5.4.1, we looked at using MPI with one process per core and compared this with a threaded implementation, with each process having several threads. Labeling the configuration as $t \times m \times n$ for t threads per process, m MPI processes per node, and n nodes, we compare choices of t, m, and n in Figure 12.9.

Table 12.4 Comparison of DryadLINQ and MPI Technologies on ALU Sequencing Application with SW-G Algorithm

Technology		*Total Time (s)*	*Time per Pair (ms)*	*Partition Data (s)*	*Calculated and Output Distance(s)*	*Merge Files (s)*
Dryad	50,000 sequences	17200.413	0.0069	2.118	17104.979	93.316
	35,339 sequences	8510.475	0.0068	2.716	8429.429	78.33
MPI	50,000 sequences	16588.741	0.0066	N/A	13997.681	2591.06
	35,339 sequences	8138.314	0.0065	N/A	6909.214	1229.10

Figure 12.9 Performance of ALU gene alignments for different parallel patterns.

The striking result for this step is that MPI easily outperforms the equivalent threaded version of this embarrassingly parallel step. In Figure 12.9, all the peaks in the overhead correspond to patterns with large values of t. Note that the MPI intra-node $1 \times 24 \times 32$ pattern completes the full 624 billion alignments in 2.33 h—4.9 times faster than the threaded implementation $24 \times 1 \times 32$. This 768-core MPI run has a parallel overhead of 1.43 corresponding to a speedup of 316.

The SW-G alignment performance is probably dominated by memory bandwidth issues, and we are pursuing several points that could affect this, though it is not at our highest priority as SW-G is not the dominant step. We have tried to identify the reason behind the comparative slowness of threading. Using Windows monitoring tools, we found that the threaded version has about a factor of 100 more context switches than in the one-thread-per-process MPI version. This could lead to a slowdown of the threaded approach and correspond to Windows handing of paging of threads with large memory footprints. AQ16

12.6 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 VM instances and to 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, (1) the matrix multiplication, (2) Kmeans clustering, and (3) the Concurrent Wave Equation Solver. Table 12.5 highlights the key characteristics of the programs that we used as benchmarks.

12.6.1 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 the OMPI version 1.3.2 with the gcc version 4.1.2. We then created a VM image from this hardware configuration, so that we would have a similar software environment on the VMs once they were deployed. The virtualization is based on the Xen hypervisor (version 3.0.3). Both bare-metal and virtualized resources utilized gigabit Ethernet connections.

When VMs are deployed using Eucalyptus, it allows us to configure 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 an Amazon EC2 infrastructure, the standard instance type has half a CPU per VM instance (Evangelinos and Hill 2008). 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 12.6.

We ran all the MPI tests, on all five hardware/VM configurations, and measured the performance and calculated speedups and overheads. We calculated two types of overheads for each application using formula (1). The total overhead induced by virtualization and parallel processing is calculated using the bare-metal single-process time as T(1) in formula (1). The parallel overhead is calculated using the single-process time from a corresponding VM as T(1) in formula (1).

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:

$$\text{Number of MPI processes} = \text{Number of CPU cores used}$$

Table 12.5 Computation and Communication Complexities of Different MPI Applications Used

Application	*Matrix Multiplication*	*Kmeans Clustering*	*Concurrent Wave Equation*
Description	Implements Cannon's Algorithm Assume a rectangular process grid (Figure 12.1, left)	Implements Kmeans Clustering algorithm A 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)	The number of points in a matrix block handled by each MPI process	The 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 both row-wise and column-wise	All MPI processes send partial clusters to one MPI process (rank 0); rank 0 distributes 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	$[O((\sqrt{n})]^3)$	$O(n)$	$O(n)$
Communication per MPI process	$[O((\sqrt{n})]^2) = O(n)$	$O(1)$	$O(1)$
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 $D \ll n$	Each message contains a double value
Communication routines used	*MPI_Sendrecv_replace*()	*MPI_Reduce*() *MPI_Bcast*()	*MPI_Sendrecv*()

Table 12.6 Different Hardware/VM Configurations Used for Our 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 (2 GB 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

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 had 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 the different characteristics we observed are shown in Figures 12.10 through 12.17.

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 speedup values, compared to the VM configurations (apart from the region close to the matrix size of 4096 × 4096, where the VM performed better than the bare-metal; we have performed multiple tests at this point, and found that it is due to the cache performances of the bare-metal node). After the bare-metal, the next-best performance and speedups were recorded in the case of 1 VM per bare-metal node configuration, in which the performance difference was mainly due to the overhead induced by the virtualization. However, as we increased the number of VMs per bare-metal node, the overhead increased

Figure 12.10 Performance of the matrix multiplication application—average time (in s) against the size of a matrix (number of MPI processes = 64).

Figure 12.11 Speedup of the matrix multiplication application—speedup against the number of MPI processes = number of CPU cores used (fixed matrix size = 5184 × 5184).

as well. At 81 processes, the 8 VMs per node configuration shows about a 34% decrease in speedup compared to the bare-metal results.

In Kmeans clustering, the effect of virtualized resources is much clearer than in the case of 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

Figure 12.12 Performance of Kmeans clustering—average time (in s) against the number of 2D data points clustered (number of MPI processes = 128).

AQ17

Figure 12.13 Speedup of Kmeans clustering—speedup against the number of MPI processes = number of CPU cores used (number of data points = 860,160).

amount of data processed by each MPI process, and also in relation to the amount of computations performed. Figures 12.14 and 12.15 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 (for all cases). For larger data sets, such as those of 40 million points, all overheads reached less than 0.5. The slower speedup

Figure 12.14 Total overhead of Kmeans clustering—overhead against 1/grain size, grain size = number of 2D data points per parallel task (number of MPI processes = 128).

Figure 12.15 Parallel overhead of Kmeans clustering—parallel overhead against 1/grain size (number of MPI processes = 128).

of the VM configurations (shown in Figure 12.13) is due to the use of a smaller data set (~800K points) to calculate the speedups. The overheads are extremely large for this region of the data sizes, and hence, this resulted in lower speedups for the VMs.

The 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

Figure 12.16 Performance of the Concurrent Wave Equation Solver—average time (in s) against the number of points computed (number of MPI processes = 128).

Figure 12.17 Total overhead of the Concurrent Wave Equation Solver—overhead against 1/grain size, grain size = number of points assigned per parallel task (number of MPI processes = 128).

number of steps. An increase in the number of points increases the amount of computations performed. Since we fixed the number of steps in which the points were 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 was clearer, and at the highest grain size, the total overhead of 8 VMs per node is about seven times

higher than the overhead of the bare-metal configuration. The performance differences between the different VM configurations became smaller with the increase in grain size.

From the above experimental results, we can see that the applications with lower C/C ratios experienced 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 (Barham et al. 2003), domUs (VMs that run on top of a 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 domUs. Although the data is not copied between domUs and dom0, dom0 needs to schedule the I/O operations on behalf of domUs. Figure 12.18 (left) and (right) shows this behavior in the 1 VM per node and 8 VMs per node configurations, respectively, that 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 Figure 12.19 (right), it is clear that dom0 needs to handle eight event channels when there are eight VM instances deployed on a single bare-metal node. Although the eight 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 the 1 VM per node configuration, when multiple processes

Figure 12.18 (Left) Communication between dom0 and domU when 1 VM per node is deployed. (Right) Communication between dom0 and domUs when 8 VMs per node are deployed.

Figure 12.19 LAM versus OMPI under different VM configurations.

(MPI or others) that run in the same VM communicate with each other via the network, all the communications must be scheduled by dom0. This results in
AQ18 higher latencies. We could verify this by running the above tests with LAM MPI (a predecessor of OMPI, 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 tests occurred when 1 VM per node was used. For example, Figure 12.19 shows the performance of Kmeans clustering under bare-metal, 1 VM per node, 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.

Several others have also performed relevant research on the performance implications of virtualized resources. Youseff et al. (2006) present an evaluation of the performance impact of Xen on MPI. According to their evaluations, the Xen does
AQ19 not impose considerable overheads for HPC (high-performance computing) applications. However, our results indicate that the applications that are more sensitive to latencies (smaller messages, lower C/C ratios) also 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 possible results. Running 1 VM per hardware node produces a VM instance with a similar number of CPU cores, such 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.

Evangelinos and Hill (2008) discuss the details of their analysis of the performance of HPC benchmarks on the EC2 cloud infrastructure. One of the key observations noted in their paper is that both the OMPI and the MPICH2-nemsis show extremely large latencies, while the LAM MPI, the GridMPI, and the MPICH2-scok show smaller, smoother latencies. This observation is similar to what we observed with the LAM MPI in our tests, and the same explanation holds valid for their observation as well.

Walker (2008) presents benchmark results of the performance of HPC applications using "high-CPU extra-large" instances provided by EC2, and on a similar set of local hardware nodes. The local nodes are connected using Infiniband switches, whereas the Amazon EC2 network technology is unknown. The results indicate about a 40%–1000% performance degradation on the EC2 resources, compared to the local cluster. Since the differences in operating systems and the compiler versions between the VMs and bare-metal nodes may cause variations in results, for our analysis, we used a cloud infrastructure over which we have complete control. In addition, we used similar software environments in both VMs and bare-metal nodes. In our results, we noticed that applications that are more susceptible to latencies experience a higher performance degradation (around 40%) under virtualized resources. Bandwidth does not seem to be a consideration in private cloud infrastructures.

Gavrilovska et al. (2007) discuss several improvements over the current virtualization architectures to support HPC applications, such as HPC hypervisors and self-virtualized I/O devices. We notice the importance of such improvements and research. In our experimental results, we used hardware nodes with 8 cores, and deployed and tested up to 8 VMs 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.

12.7 Conclusions and Future Work

We have described several different studies of clouds and cloud technologies on both real applications and standard benchmark. These address different aspects of parallel computing using either traditional (MPI) or the new cloud-inspired approaches. We find that cloud technologies work well for most pleasingly parallel problems ("map-only" and "map-reduce" classes of applications). In addition, 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, all serve to simplify the implementation details of such problems. Applications with complex communication patterns observe higher overheads when implemented using

cloud technologies, and even with large data sets, these overheads limit the usage of cloud technologies for such applications. Enhanced MapReduce runtimes, such as CGL-MapReduce, allow iterative-style applications to utilize the MapReduce programming model, while incurring minimal overheads, as compared to 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 whether 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 the data 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 an 8% total overhead, while 8 VMs per node show at least a 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 half a core per VM, to produce very high overheads for applications that are sensitive to latencies. Applications that are not susceptible to latencies, such as those that perform large data transfers and/or higher C/C 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 (Fox et al. 2008).

Acknowledgments

We would like to thank Joe Rinkovsky and Jenett Tillotson from University Information Technology Services, Indiana University (IU UITS), for their dedicated support in setting up a private cloud infrastructure and helping us with various configurations associated with our evaluations. We would also like to thank the ARTS team at Microsoft Research for their support on hardware and software infrastructures. We are grateful to Mina Rho and Haixu Tang from the Indiana University School of Informatics and Computing for their help in understanding ALU sequence clustering and providing human and chimpanzee gene sequence data.

AQ20

References

Amazon.com, Inc. 2009. Simple Storage Service (S3). http://aws.amazon.com/s3

ASF. 2009a. Apache Hadoop Core. http://hadoop.apache.org/core

ASF. 2009b. Apache Hadoop Pig. http://hadoop.apache.org/pig/

Barham, P., B. Dragovic et al. 2003. Xen and the art of virtualization. *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, NY.

Batzer, M.A. and P.L. Deininger. 2002. ALU repeats and human genomic diversity. *Nat. Rev. Genet*. **3**(5): 370–379.

Condor Team. 2009. Condor DAGMan. http://www.cs.wisc.edu/condor/dagman/

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

AQ21 Dongarra, J., C.A. Geist et al. 1993. Integrated PVM framework supports heterogeneous network computing. *Comput. Phys*. **7**(2): 166–175.

Ekanayake, J., S. Pallickara et al. 2008. MapReduce for data intensive scientific analyses. *IEEE Fourth International Conference on eScience '08*, Indianapolis, IN.

ElasticHosts Ltd. 2009. Cloud Hosting. http://www.elastichosts.com/

Ericsson 2009. Erlang programming language. http://www.erlang.org/

Evangelinos, C. and C. Hill. 2008. Cloud computing for parallel scientific HPC applications: Feasibility of running coupled atmosphere-ocean climate models on Amazon's EC2. *The First Workshop on Cloud Computing and its Applications* (*CCA'08*), Chicago, IL.

Forum, MPI. n.d. MPI (Message Passing Interface). http://www.mcs.anl.gov/research/projects/mpi/

Foster, I. 2001. The anatomy of the grid: Enabling scalable virtual organizations. *Int. J. Supercomput. Appl*. **15**: 200–222.

Fox, G.C., A. Hey, and S. Otto. 1987. Matrix algorithms on the hypercube I: Matrix multiplication. *Parallel Comput*. **4**: 17.

Fox, G., S. Bae et al. 2008. Parallel data mining from multicore to cloudy grids. *Proceedings of the International Advanced Research Workshop on High Performance Computing and Grids* (*HPC2008*), Cetraro, Italy.

Gabriel, E., G.E. Fagg et al. 2004. Open MPI: Goals, concept, and design of a next generation MPI implementation. *Proceedings of the 11th European PVM/MPI Users' Group Meeting*. Budapest, Hungary.

Gavrilovska, A., S. Kumar et al. March 2007. Abstract high-performance hypervisor architectures: Virtualization in HPC systems. *Proceedings of the HPCVirt 2007*, Lisbon, Portugal.

Ghemawat, S., H. Gobioff et al. 2003. The Google file system. *SIGOPS Oper. Syst. Rev*. **37**(5): 29–43.

Gotoh, O. 1982. An improved algorithm for matching biological sequences. *J. Mol. Biol*. **162**: 705–708.

Gu, Y. and R.L. Grossman. 2009. Sector and sphere: The design and implementation of a high-performance data cloud. *Phil. Trans. A: Math Phys. Eng. Sci*. **367**(1897): 2429–2445.

Huang, X. and A. Madan. 1999. CAP3: A DNA sequence assembly program. *Genome Res*. **9**(9): 868–877.

Hull, D., K. Wolstencroft et al. 2006. Taverna: A tool for building and running workflows of services. *Nucleic Acids Res*. 34(Web Server issue): W729–32.

Isard, M., M. Budiu et al. 2007. Dryad: Distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev*. **41**(3): 59–72.

Johnsson, S.L., T. Harris et al. 1989. Matrix multiplication on the connection machine. *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, Reno, NV. ACM, New York.

AQ22

Jurka, J. 2000. Repbase update: A database and an electronic journal of repetitive elements. *Trends Genet*. **9**:418–420.

Keahey, K., I. Foster et al. 2005. Virtual workspaces: Achieving quality of service and quality of life in the Grid. *Sci. Program*. **13**(4): 265–275.

Ludscher, B., I. Altintas et al. 2006. Scientific workflow management and the Kepler system [research articles]. *Concurr. Comput. Pract. Exper*. **18**(10): 1039–1065.

Macqueen, J. 1967. Some methods for classification and analysis of multivariate observations. *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, Vol. 1: *Statistics*, Berkeley, CA, pp. 281–297.

AQ23

AQ24

Moretti, C., H. Bui et al. 2009. All-pairs: An abstraction for data intensive computing on campus grids. *IEEE Trans. Parallel Distrib. Syst*. **99**(1).

Nokia. 2009. Disco project. http://discoproject.org/

Nurmi, D., R. Wolski et al. 2009. The eucalyptus open-source cloud-computing system. *Proceedings of the Ninth IEEE/ACM International Symposium on Cluster Computing and the Grid*, Shanghai, China.

Pallickara, S. and G. Fox. 2003. NaradaBrokering: A distributed middleware framework and architecture for enabling durable peer-to-peer grids. *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*, Rio de Janeiro, Brazil. Springer-Verlag, New York.

Pallickara, S. L. and M. Pierce. 2008. SWARM: Scheduling large-scale jobs over the loosely-coupled HPC clusters. *Proceedings of the IEEE Fourth International Conference on eScience '08* (*eScience*, *2008*), Indianapolis, IN.

Pike, R., S. Dorward et al. 2005. Interpreting the data: Parallel analysis with Sawzall. *Sci. Program*. **13**(4): 277–298.

Raicu, I., Y. Zhao et al. 2007. Falkon: A fast and light-weight tasK executiON framework. *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, Reno, NV. ACM, New York.

ServePath. 2009. GoGrid Cloud Hosting. http://www.gogrid.com/

Smith, T.F. and M.S. Waterman. 1981. Identification of common molecular subsequences. *J. Mol. Biol*. **147**:195–197.

Smith, A.F.A., R. Hubley, and P. Green. 2004. Repeatmasker. http://www.repeatmasker.org

Smith Waterman Software. http://jaligner.sourceforge.net/naligner/

Vermorel, J. 2005. NAligner (Smith Waterman software with Gotoh enhancement). http://jaligner.sourceforge.net/naligner/

Walker, E. 2008. Benchmarking Amazon EC2 for high-performance scientific computing. http://www.usenix.org/publications/login/2008–10/openpdfs/walker.pdf.

Youseff, L., R. Wolski et al. 2006. Evaluating the performance impact of Xen on MPI and process execution for HPC systems. *Proceedings of the First International Workshop on Virtualization Technology in Distributed Computing*, Tampa, FL.

Yu, Y., M. Isard et al. 2008. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. *Proceedings of the Symposium on Operating System Design and Implementation* (*OSDI*), San Diego, CA.

AUTHOR QUERIES

[AQ1] Please check if the fixed running head is ok.

[AQ2] Please expand "MPI," if appropriate.

[AQ3] Please expand "PVM," if appropriate.

[AQ4] Please check whether "italics" for emphasis be changed to bold.

[AQ5] Fox and Hey (1987) has been changed to Fox et al. (1987) to match with the reference list. Please check.

[AQ6] Please check if the edit of the sentence starting "In the two…" is correct.

[AQ7] Please expand "ALU," if appropriate.

[AQ8] En dashes have been inserted for the combinations "Smith Waterman" and "Smith Waterman Gotoh" in this chapter. Please check.

[AQ9] Smith and Hubley (2004) has been changed to Smith et al. (2004) to match with the reference list. Please check.

[AQ10] Smith et al. (1981) has been changed to Smith and Waterman (1981) to match with the reference list. Please check.

[AQ11] Please expand "MDS," if appropriate.

[AQ12] Please check if the edit of the sentence starting "We decompose…" is correct.

[AQ13] "PairEnumertator" has been changed to "PairEnumerator" in the sentence starting "We implemented…" Please check.

[AQ14] In the sentence starting "Therefore, we…," please check whether the phrase at the end of the sentence "this file system" refers to the "high-performance parallel file system" or the "Lustre file system," and modify the sentence accordingly for clarity.

[AQ15] Please check if the edit of the sentence starting "On 744 cores…" is correct.

[AQ16] Please check the last part of the sentence starting "This could lead…" for sense, and whether "Windows handing" should be changed to "Windows handling."

[AQ17] Please check the caption of Figure 12.12 against the respective artwork. Caption says "2D data points," whereas artwork says "3D data points."

[AQ18] Please expand "LAM," if appropriate.

[AQ19] Please check if the expansion inserted for "HPC" is correct.

[AQ20] Please expand "ARTS," if appropriate.

[AQ21] Please check the inserted volume no. in Dongarra et al. (1993) for correctness.

[AQ22] Please check the location of publisher in Johnsson et al. (1989) and Raicu et al. (2007) for correctness.

[AQ23] Please check the inserted volume no. and page range in Macqueen (1967) for correctness.

[AQ24] Please provide page range for Moretti et al. (2009).