MATE-EC2: A Middleware for Processing Data with AWS

Tekin Bicer David Chiu[†] Gagan Agrawal
Department of Computer Science and Engineering, Ohio State University
[†] School of Engineering and Computer Science, Washington State University

***Abstract*—Recently, there has been growing interest in using Cloud resources for a variety of high performance and data-intensive applications. While there is currently a number of commercial Cloud service providers, Amazon Web Services (AWS) appears to be the most widely used. One of the main services that AWS offers is the Simple Storage Service (S3) for unbounded reliable storage of data, which is particularly amenable to data-intensive processes. Certainly, for these types of applications, we need support for effective retrieval and processing of data stored in S3 environments.**

In this paper, we focus on parallel and scalable processing of data stored in S3, using compute instances in AWS. We describe a middleware, MATE-EC2, that allows specification of data processing using a high-level API, which is a variant of the Map-Reduce paradigm. We show various optimizations, including data organization, job assignment, and data retrieval strategies, that can be leveraged based on the performance characteristics of S3. Our middleware is also capable of effectively using a heterogeneous collection of EC2 instances for data processing. Our detailed experimental study further evaluates what factors impact efficiency of retrieving and processing S3 data. We compare our middleware with Amazon Elastic Map-Reduce and show how we determine the best configuration for data processing on AWS.

I. INTRODUCTION

The outset of Cloud computing has been apropos in facilitating today's increasingly data-intensive projects. In response to the data deluge, processing times can be expedited by harnessing the Cloud's *resource elasticity*, i.e., the ability for on-demand allocation of compute instances and ostensibly infinite storage units. While Cloud providers continue to grow in number, Amazon Web Services (AWS) has gained marked popularity among various stakeholders. AWS offers several services that are highly amenable for supporting data-intensive applications. For instance, the Simple Storage Service (S3) allows for highly accessible, reliable, and "infinite" data store. Certainly for data-intensive applications, one important consideration would involve the effective retrieval and processing of data based in S3. Another AWS service is the Elastic Compute Cloud (EC2), where virtual machine instances of varying capabilities, with varying pricing costs, can be leased for unbounded amounts of time — a welcome departure from traditional advanced resource reservation schemes.

This notion of on-demand resources has already prompted many users to adopt the Cloud for large-scale projects, including medical imaging [19], astronomy [5], BOINC applications [13], and remote sensing [15], among many others. Another distinct set of data-intensive applications fall under the Map-Reduce framework [3]. Spurred by its popularity, support for Map-Reduce was quickly embraced by prominent Cloud providers. The Hadoop-oriented [8] AWS version (Elastic Map-Reduce) allows users to upload their *map* and *reduce* code, as well as their data sets onto S3. Elastic Map-Reduce then launches a user-specified number of machine instances, and proceeds to invoke the code, thereby abstracting processing nuances from users.

Cloud environments and their use for high-performance and data-intensive applications are still quite new. Much effort is needed in understanding the performance characteristics of these environments. Similarly, we feel that additional services and tools should be developed for enabling wider use of these environments for end applications.

This paper focuses on the above two goals in the context of developing data-intensive applications on AWS. Particularly, the goal is to be able to use a (possibly heterogeneous) collection of EC2 instances to scalably and efficiently process data stored in S3. We have developed a Cloud middleware, *MATE-EC2 (Map-reduce with AlternaTE api over EC2)*, which deploys a variation of the Map-Reduce API over Amazon's EC2 and S3 services. We show that various optimizations based on S3's characteristics can be applied to improve performance. Particularly, we have used *threaded data retrieval* to improve the effective bandwidth of data retrieval. Additionally, our *selective job assignment* strategy ensures that different threads read data from different data objects in S3, thus avoiding bottleneck. We also show that heterogeneous clusters of various EC2 instance types can be leveraged effectively in our middleware.

Our experimental study reveals several issues which affect the retrieval and processing of data stored in S3. We show that with increasing chunk sizes, the data processing times increase, but the data retrieval times decrease. We also evaluate the optimal number of data retrieval threads for each data processing thread leads for our applications. Overall, for three popular data-intensive applications, we report excellent scalability with increasing number of compute instances. Furthermore, we are able to exploit the aggregated computing power of a heterogeneous cluster very effectively. Finally, we achieve a performance improvement ranging between 3 and 28 against Amazon Elastic MapReduce for our data-intensive applications.

The rest of this paper is organized as follows. In Section II, we discuss the background on AWS and Map-Reduce. We focus on considerations for system design in Section III. A nuanced experimental evaluation is described in Section IV, which is followed by a discussion of related research efforts in Section V. Finally, we summarize our conclusions and discuss future directions in Section VI.

II. Background

In this section, we present the background on data storage and computing on the Amazon Web Services (AWS) public Cloud, as well as on Map-Reduce and Amazon Elastic Map-Reduce.

A. Computing and Data Storage with AWS

AWS offers many options for on-demand computing as part of their Elastic Compute Cloud (EC2) service. EC2 nodes (*instances*) are virtual machines that can launch snapshot *images* of operating systems. These images can be deployed onto various *instance types* (the underlying virtualized architecture) with varying costs depending on the instance type's capabilities. For example, a Small EC2 Instance (`m1.small`), according to AWS[1] at the time of writing, contains 1.7 GB memory, 1 virtual core (equivalent to a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor), and 160 GB disk storage. Many other such instance types exist, also with varying costs and capabilities.

Amazon's persistent storage framework, Simple Storage Service (S3), provides a key-value store. Typically, the unique keys are represented by a filename and the path to the *bucket* which contains it, and the values are themselves the data objects. While the objects are limited to 5 TB, the number of objects that can be stored in S3 is unrestricted. Aside from the simple API, the S3 architecture has been designed to be highly reliable and available. It is furthermore very inexpensive in terms of price to store data on S3.

B. Map-Reduce and Amazon Elastic Map-Reduce

Map-Reduce [4] was proposed by Google for application development on data-centers with thousands of computing nodes. It can be viewed as a middleware system that enables easy development of applications that process vast amounts of data on large clusters. Through a simple interface of two functions, *map* and *reduce*, this model facilitates parallel implementations of many real-world tasks, ranging from data processing for search engine support to machine learning [2], [6].

The main benefits of this model are in its simplicity and robustness. Map-Reduce allows programmers to write functional style code that is easily parallelized and scheduled in a cluster environment. One can view Map-Reduce as offering two important components [18]: a *practical programming model* that allows users to develop applications at a high level and an *efficient runtime system* that deals with the low-level details. Parallelization, concurrency control, resource management, fault tolerance, and other related issues are handled by the Map-Reduce runtime.

Map-Reduce implementations, particularly the open-source *Hadoop*, have been used on various clusters and data centers [8]. In fact, AWS also offers Hadoop as a service called Amazon Elastic Map-Reduce, which allows processing of data hosted on S3 while using EC2 compute instances. For users of AWS, Elastic Map-Reduce serves as a high-level framework for data analysis, without having to perform substantial software installation. Furthermore, for users of Hadoop or Map-Reduce, Amazon Elastic MapReduce also alleviates the need for owning and/or managing their own cluster.

[1] AWS Instance Types, http://aws.amazon.com/ec2/instance-types

III. System Design

This section describes the design of the MATE-EC2 middleware. We will first describe the API for high-level specification of the data processing task, followed by a nuanced discussion of the middleware design.

A. Processing API

In this subsection, we describe the details of the Map-Reduce programming model and our alternate API implemented in the MATE-EC2 system, which we refer to as the *generalized reduction* API.

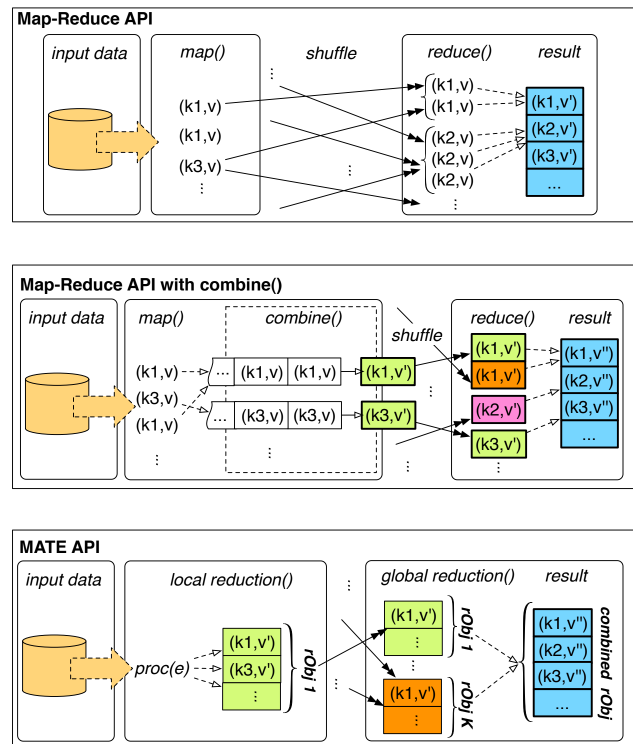


Fig. 1. Processing Structures

We show the processing structures for the MATE and the Map-Reduce programming model with and without the *Combine* function in Figure 1. The Map-Reduce programming model [4] can be summarized as follows. The *map* function takes a set of input points and generates a set of corresponding output $(key, value)$ pairs. The Map-Reduce library sorts and groups the intermediate $(key, value)$ pairs, then passes them to the *reduce* function in such a way that the same keys are always placed on the same *reduce* node. This stage is often referred to as *shuffle*. The *reduce* function, which is also written by the user, accepts a key and a set of values associated with that key. It merges together these values to form a possibly smaller set of values. Typically, just zero or one output value is produced per *reduce* invocation.

The Map-Reduce framework also offers programmers an optional *Combine* function, which can be used to improve the performance of many of the applications. Before the $(key, value)$ pairs are emitted from the mapping phase, they are grouped according to their key values and stored in a buffer on the

map nodes. When this buffer is flushed periodically, all grouped pairs are immediately reduced using the *Combine* function. These intermediate reduced pairs are then emitted to the *reduce* function. The use of *Combine* can decrease the intermediate data size significantly, and therefore reducing the amount of $(key, value)$ pairs that must be communicated from the *map* and *reduce* nodes.

We now explain the MATE-EC2 API, which also has 2-phases: The *local reduction* phase aggregates the processing, combination, and reduction operations into a single step, shown simply as `proc(e)` in our figure. Each data element e is processed and reduced locally before next data element is processed. After all data elements have been processed, a *global reduction* phase commences. All reduction objects from various local reduction nodes are merged with an *all-to-all* collective operation or a user defined function to obtain the final result.

The advantage of this design is to avoid the overheads brought on by intermediate memory requirements, sorting, grouping, and shuffling, which can degrade performance in Map-Reduce implementations [9]. At first glance, it may appear that our API is very similar to Map-Reduce with the *Combine* function. However, there are important differences. Using the *Combine* function can only reduce communication, that is, the $(key, value)$ pairs are still generated on each *map* node and can result in high memory requirements, causing application slowdowns. Our generalized reduction API integrates *map*, *combine*, and *reduce* together while processing each element. Because the updates to the reduction object are performed directly after processing, we avoid intermediate memory overheads.

The following are the components in MATE-EC2 that should be prepared by the application developer:

- `Reduction Object`: This data structure is designed by the application developer. However, memory allocation and access operations to this object are managed by the middleware for efficiency.
- `Local Reduction`: The local reduction function specifies how, after processing one data element, a *reduction object* (initially declared by the programmer) is updated. The result of this processing must be independent of the order in which data elements are processed on each processor. The order in which data elements are processed is determined by the runtime system.
- `Global Reduction`: In this function, the final results from multiple copies of a *reduction object* are combined into a single reduction object. A user can choose from one of the several common combination functions already implemented in the MATE-EC2 system library (such as aggregation, concatenation, etc.), or they can provide one of their own.

Our MATE-EC2's generalized reduction API is motivated by our earlier work on a system called FREERIDE (FRamework for Rapid Implementation of Datamining Engines) [10], [11], [9].

B. Design for Processing S3 Data

In this subsection we explain how data should be organized and accessed by MATE-EC2 in order to maximize processing throughput.

Data Organization

MATE-EC2 is developed in order to process large amounts of data. However, there are generally space limitations while storing data into a file system. Also, it can be more efficient to split data into many files. Therefore, the first stage of organizing data in MATE-EC2 is to split the application's data set into a required number of data objects.

By itself, this is a fairly trivial process, however, data organization must also involve the processing units' requirements in order to maximize the throughput. More specifically, the available memory at the processing node and caching mechanisms in the system should also be considered. Therefore, MATE-EC2 *logically* divides each data object into memory-friendly *chunks*, typically only some tens of MBs. In S3, it is not necessary to physically split data objects into smaller chunk units because it allows for partial data accesses.

Therefore, the overall organization of data sets in MATE-EC2 is as follows. The application's data set is first split into several data objects. Though the physical limit on the size of each obect is 5 TB, the number of data objects that the entire dataset is split into is a multiple of the number of processes. Therefore, processes can be evenly mapped among the data objects, which results in a lesser number of connections to each object and increase network utilization. We further store offsets for accessing each *chunk* in a metadata file associated with each S3 object. This metadata includes: (1) the full-path to the data object, (2) offset address of the chunk, (3) size of the chunk, and (4) the number of data points within the chunk, P. Each data point defines the smallest unit of data that can be analyzed by the application process.

Consider the following example. Assume we have a set of points in 3D space. If the data set size is 24 GB and the number of processes is 16, then the data can be split into 16 data objects. Therefore, each process can work on one data object and exploit the bandwidth. The size of chunks can vary according to the available resources in the processing unit and the desired workload per job. Assuming that a 128MB chunk size provides the optimal resource utilization, then each data object can be divided into 12 logical data chunks $(24\text{GB}/(16 \times 128\text{MB})$. Since the real processing deals with points in 3D space, the smallest size of the data unit can be expressed with 3 double-float numbers, and therefore, $P = 128\text{MB}/(3 \times sizeof(double))$. The metadata of the chunk offsets is stored in an *index file* and used for the purpose of chunk assignment to the processing nodes.

With the discussion of data organization, we now describe our data processing scheme in detail.

Data Retrieval Subsystem

Using the metadata pertaining to data chunks, jobs can now be assigned to the processing units. Although S3 provides high availability, scalability, and low latency for stored data, the low-level design details of the system are not made publicly available. Therefore, we have empirically investigated S3's performance. As a result from our experiments, we observed the following features that affect data access performance on S3: (1) *Threaded Data Retrieval* and (2) *Selective Job Assignment*.

Threaded Data Retrieval uses a predefined number of threads in order to efficiently retrieve data chunks from S3. The process first allocates the required memory for the data chunk. Shown in

Figure 2, each thread in the compute unit reads different parts of the data chunk from an S3 object via the offsets and writes them into the pre-specified memory location. When all threads have retrieved the desired data parts of the chunk, the memory address and the metadata information are passed to the computing layer in which data is processed.

Consider a situation where a process must retrieve a data chunk that is 128 MB in size, using 16 retrieving threads. Each thread receives a $128\text{MB}/16 = 8\text{MB}$ chunklet, i.e., a chunk within a data chunk, and writes it into their corresponding portions of the buffer. Because S3 allows for parallel reads per data object, this approach maximizes bandwidth usage, and reduces overall execution times of the processes, which we will show in the ensuing section.

Selective Job Assignment exploits the available bandwidth of the data objects in S3. As we stated above, the threaded retrieval approach potentially opens many connections to a single data object in order to efficiently retrieve the data chunks. Naturally, the number of active connections made on each data object in turn impacts the data retrieval speed. The jobs, therefore, are distributed among the processes so that the number of active connections is minimized on data objects. This is satisfied by checking the connections on each of the data object and selecting the job from the data object that has minimum number of active connections.

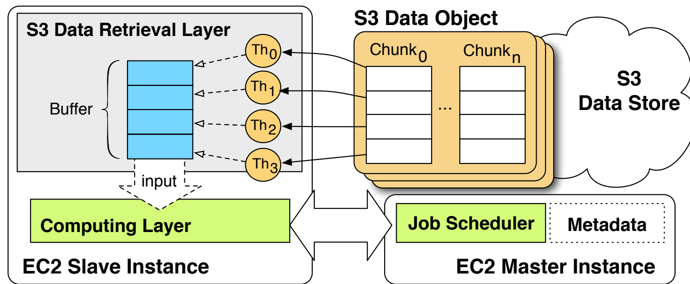


Fig. 2. MATE-EC2 Processing System

Figure 2 demonstrates the interaction between MATE-EC2 components. This figure depicts an EC2 master instance, an EC2 slave instance, and a data object with n data chunks. In practice, there is more than just one slave instance and many data objects. However, the execution flow follows the same pattern for any configuration.

The EC2 master instance distributes the unprocessed data chunks among the requesting processes in EC2 slave instances. Because the chunk information is in the metadata file, the master instance can compile a job and assign it to the process which is requesting it. When the process in a slave instance retrieves the job's metadata, it extracts the chunk information and begins requesting the chunk from the S3 object. The parts of the chunk are then written to the process' buffer by the retrieving threads. When the chunk is ready in the buffer, it is passed to the computing layer, where processing is handled at a coarser scale.

As we mentioned, the metadata pertaining to a chunk includes the number of data points, P, for that particular chunk. Therefore, the computing layer can calculate the data unit size and iteratively process the number of data units that can fit into the CPU's cache. Whereas the number of data points per chunk helps our system to effectively utilize the cache, the chunk size helps take advantage of memory usage. When all the data units in a chunk have been consumed, the process requests another job from the master instance. Once all the data chunks have been consumed, the processes synchronously enter the global reduction phase and combine all the reduction objects into the final result.

C. Load Balancing and Handling Heterogeneity

As discussed in previous sections, MATE-EC2 was also designed to manage computation on a heterogeneous set of compute instances. The motivation for this arises from how AWS makes instances available as *on-demand*, *reserved*, and *spot* instances, which are available due to the amount of money a user is willing to bid. Let us suppose an organization has several `large` instances reserved. During high workloads, it may choose to obtain additional spot instances to improve the processing rate. Depending on the organization's budget, if these spot instances happen to be `small` instances, the processing will have to be performed on heterogeneous set of instances. Currently, Amazon Elastic MapReduce is not capable of using a heterogeneous set of instances. In contrast, MATE-EC2 provides dynamic load balancing to allow the effective use of the aggregate processing power.

Algorithm 1: Assigning jobs to the EC2 Instances

```
Input: chunkInfoList, metadata of the chunks
Result: job, which is assigned to EC2 compute instance
s3ObjList ← createS3ObjList(chunkInfoList);
/* EC2 instance request handler loop        */
while true do
    ec2CompIns ← ReceiveReq();
    if CheckAssigned(ec2CompIns) then
        SetProcessed(ec2CompIns, s3ObjList);
    s3Obj ← AvailS3Obj(s3ObjList);
    job ← CreateJob(s3Obj);
    Transfer(job, ec2CompIns);
    if IsNotEmpty(job) then
        Assign(ec2CompIns,s3Obj);
```

Our approach for supporting dynamic load balancing is based on a *pooling mechanism*. The metadata information that is extracted from the data objects is used for this purpose. First, the EC2 *master instance* retrieves and reads the metadata file from S3 and creates the job pool. Next, the processes in *slave instances* commence requesting jobs from the master instance. The appropriate job is selected from the pool and assigned to the requesting process. Therefore, the system throughput and resource utilization are maximized.

Algorithm 1 shows how the master instance manages the compute instances, which is also illustrated in Figure 2. Initially, the master instance waits for the job requests from slave instances. When a job request is received, the scheduler checks if any of the chunks in the system was previously assigned to the requesting compute instance. If so, the scheduler sets their state as *processed*.

Algorithm 2: Processing Chunks on EC2 Compute Instance

```
Input: scheduler, Job Scheduler
     : rObj, User specified initial reduction object
Result: Updated final reduction object, rObj

/* Execute outer sequential loop          */
while true do
    /* Execute job request loop           */
    while true do
        job ← RequestJob(scheduler);
        if IsEmpty(job) then
            break;
        s3Obj ← GetS3Obj(job);
        chunksInfo ← GetChunksInfo(job);
        foreach chunk info cinfo in chunksInfo do
            {* Retrieve data chunk chk with cinfo from
            s3Obj *};
            (i, val) ← Process(chk);
            rObj(i) ← Reduce(rObj(i), val);
    {* Perform Global Reduction *};
```

Then, it creates another job with a new set of chunk information from the most suitable S3 objects in the system.

The main consideration for the scheduler in choosing an S3 object is effectively using the available bandwidth from each. Therefore, if the number of connections between all pairs of compute instances and S3 data objects is the same, the job mapping will be done in a *round robin* fashion. Conversely, if the number of connection on S3 objects varies, then more compute instances' requests will be mapped to the S3 objects with least number of connections, as long as they still have chunks that need to be processed. After creating the job from an S3 object, it is transferred to the requesting compute instance.

When a compute instance receives a job from the scheduler, it extracts the chunk's metadata information and begins requesting the data from the corresponding S3 data object. The data chunk is retrieved and written to the chunk buffer using the aforementioned threaded retrieval approach. The data retrieval layer then passes the chunk to the computing layer and begins the local reduction phase. After the data processing stage is finished, the compute instance will request another job, until all chunks are consumed. Finally, all compute instances reduce their execution with the global *all-to-all* reduction. This process is shown in Algorithm 2.

IV. Applications and Experimental Results

In this section, we extensively evaluate our data-intensive computing middleware with various configurations using Amazon EC2 instances and the S3 data store. The goals of our experiments are as follows: (1) Comparing our middleware with a similar and popular system, Amazon Elastic MapReduce, (2) investigating Amazon's environments and services with various settings to detect the most suitable configurations for data intensive computing, and (3) determining the performance of our middleware on homogeneous and heterogeneous resources.

We used large EC2 instances (`m1.large`) for our experiments. According to Amazon at the time of writing, these are 64-bit instances with 7.5 GB of memory. Large instances provide two virtual cores, and each core further contains two elastic compute units. Large instances are also rated as having *high* I/O performance by AWS.

Throughout our experiments, we used varying numbers of large instances as *slaves* for computing purposes and one separate large instance as the *master* node for managing execution flow. The original data file was divided into 16 data objects and stored in S3. Each of these data objects is logically separated into different number of chunks and metadata information were extracted during this operation.

Three representative data-intensive applications were used to evaluate our system: This includes two popular data mining algorithms, KMeans Clustering and Principal Component Analysis (PCA), and PageRank [16]. The data set size for both KMeans and PCA is 8.2 GB (thus, each data object stored in S3 is 8.2/16 GB or nearly 500 MB). The data set used for PageRank is 1 GB, contains 9.6 million nodes and 131 million edges.

During the experiments, we set the computation parameters constant and observed the execution time with processing, synchronization, and data retrieval times. The initialization times of the instances are omitted. For each configuration, we repeated the experiment at least five times and display the average of the results.

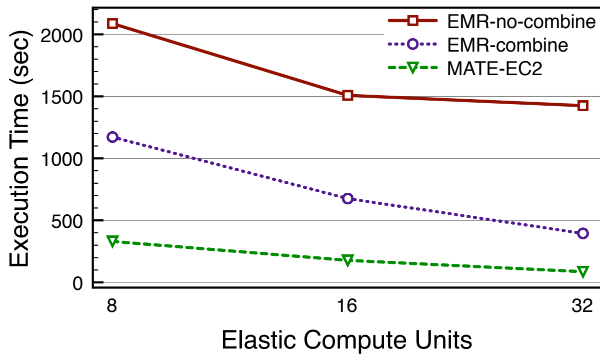
A. Comparison of MATE-EC2 and Amazon Elastic Map-Reduce

In this set of our experiments, we evaluate the performance and scalability of our middleware and compare it to Amazon Elastic MapReduce. Of the three applications we used, we implemented two different version of KMeans and PageRank applications on Elastic Map-Reduce (EMR): *with* and *without* using the *Combiner* function, which are referred to as `EMR-combine` and `EMR-no-combine`, respectively. The nature of computation in PCA does not allow us to use the combiner function, as we will explain later.

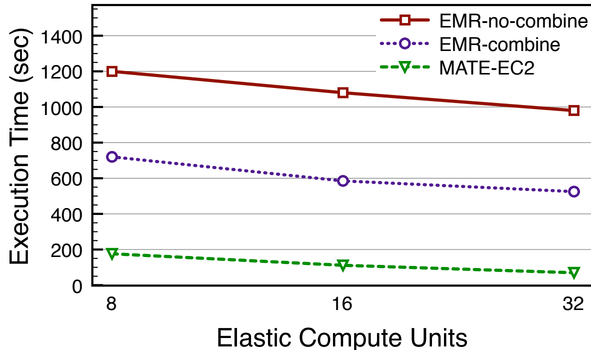
In Figure 3(a) we show the execution times of `MATE-EC2`, `EMR-combine` and `EMR-no-combine` with increasing number of elastic compute units for KMeans. If we consider the 8 compute units configuration as the baseline, the scalability ratios for `EMR-no-combine` are 1.38 and 1.46 for 16 and 32 compute units, respectively. In `EMR-combine`, these ratios are 1.73 for 16, and 2.97 for 32 compute units. Moreover, the speedups of using *Combiner* function are 1.78, 2.23 and 3.61 for 8, 16 and 32 compute units. Thus, Combiner's intermediate reduction, which reduces data transfers between the *map* and *reduce* nodes, is resulting in significant benefits.

In terms of scalability of MATE-EC2, if we consider the 8 compute units configuration as the baseline, the speedups of the 16 and 32 compute units are 1.86 and 3.82, respectively. That is, the relative parallel efficiency is 90% or higher. Considering the performance, MATE-EC2 runs 3.54, 3.81, and 4.58 times faster for 8, 16, and 32 compute units respectively than `EMR-combine`.

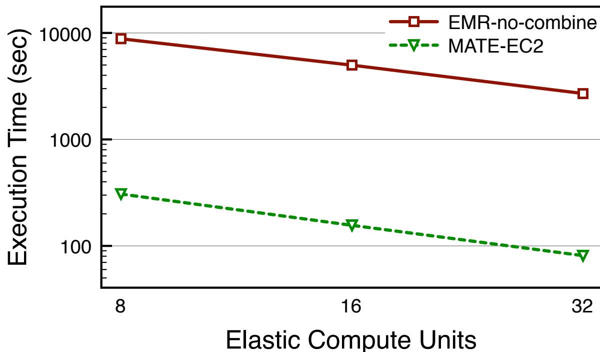
We believe there are several likely reasons for the higher performance with MATE-EC2. First is the difference between the processing APIs: Although *Combiner* function reduces the



(a) KMeans – 128MB Chunk Size, 16 Data Retrieval Threads



(b) PageRank – 128MB Chunk Size, 16 Data Retrieval Threads



(c) PCA – 128MB Chunk Size, 16 Data Retrieval Threads

Fig. 3. MATE-EC2 vs. Elastic MapReduce

pairs before they are emitted from the local process, they are still required to be sorted and grouped before they are passed to the *Combiner* function from the *map* function [2]. Furthermore, the pairs need to be shuffled after they are emitted. In MATE-EC2's processing structure, on the other hand, the pairs are accumulated into the reduction object according to their key values right after their generation, which eliminates these overheads.

Typically, Map-Reduce's task tracker maps the tasks where the data resides, thus the locality is maximized. However, in our experiments we examine the situations in which the data is stored in S3 resources. Therefore, data needs to be retrieved and then processed[3]. According to Amazon's documents, a good practice is to initiate EC2 resources in the same region where the data is located. Thus, the *map* and *reduce* tasks can exploit the bandwidth, though they cannot be mapped to the nodes in which the data resides. Considering MATE-EC2, our data organization, job assignment and data retrieval strategies optimize the interaction between processes and S3 data objects.

We repeated the same experiment for PageRank and present the results in Figure 3(b). The speedups of `MATE-EC2` are 1.58 for 16 and 2.53 for 32 compute units, whereas `EMR-combine` configuration's are 1.23 and 1.37. Furthermore, we observe that the execution times for `MATE-EC2` are 4.08, 5.25, and 7.54 times faster than `EMR-combine` for 8, 16 and, 32 compute units, respectively. The data read and write operations are the dominating factors for `EMR-combine` execution times.

In Figure 3(c), we repeated and displayed the execution times of PCA application with `EMR-no-combine` and `MATE-EC2`. The speedups of `MATE-EC2` are 1.96 and 3.77 for 16 and 32 compute units. On the other hand `EMR-no-combine` achieved speedups of 1.77 and 3.27. When we compare the execution times of MATE-EC2 and Elastic MapReduce, our middleware is 28 times faster than Map-Reduce execution for all three configurations.

The performance differences are much higher for PCA, and the reasons arise from its computing pattern. PCA involves two iterations (job flows) during its execution. The first iteration calculates the mean value of the matrix, which can be implemented efficiently with Map-Reduce. In the second iteration, however, the computation requires two rows from the matrix simultaneously in order to calculate the covariances of a target row. While one of the rows (target row) can be handled directly with an emitted $(key, value)$ pair, the other row must be traversed and identified in the data objects. This operation is done for *all* row pairs, which contributes to significant overhead. This computational nature is the reason why the `EMR-combine` version is not implemented for PCA.

In contrast, the offset information and the data unit size can be used in order to efficiently traverse over the data chunks and data objects with MATE-EC2. Therefore, the data organization of MATE-EC2 effectively improves the performance for fetching the matrix rows from S3 data objects.

Overall, we believe that this set of experiments offer strong evidence that MATE-EC2 outperforms Elastic MapReduce in terms of both scalability and performance.

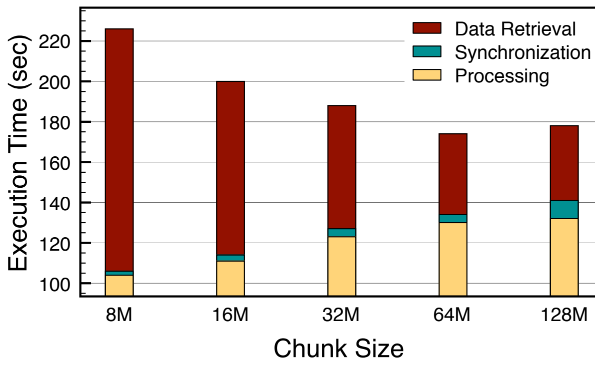
In the next subsection, we show how we evaluated the MATE-EC2's performance with different parameters and determined the best configuration. As the trends are similar with all three applications, we report detailed results only from KMeans and PCA.

B. Detailed Performance Studies on AWS

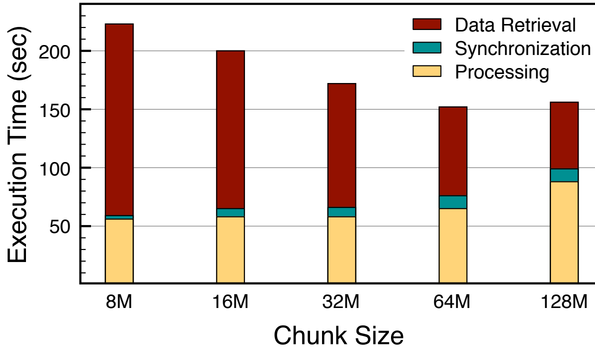
We focus first on the performance of MATE-EC2 with varying chunk sizes and retrieving threads. We employed 4 large instances (16 compute units) as slaves for processing and 1 large instance as the master for job scheduling.

Figure 5 shows the performance of MATE-EC2 with only one retrieving thread per processing thread, and with varying chunk sizes. For instance, the configuration with 128KB size splits each data object close to 4200 *logical chunks* within the S3 data object

[2]http://wiki.apache.org/hadoop/HadoopMapReduce
[3]http://aws.amazon.com/elasticmapreduce/faqs/#cluster-1

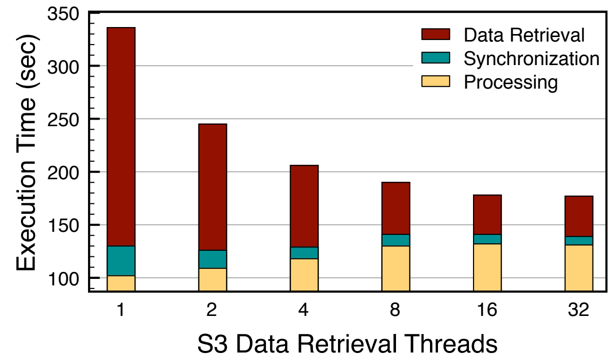


(a) KMeans - 16 Data Retrieval Threads for Each Processing Thread

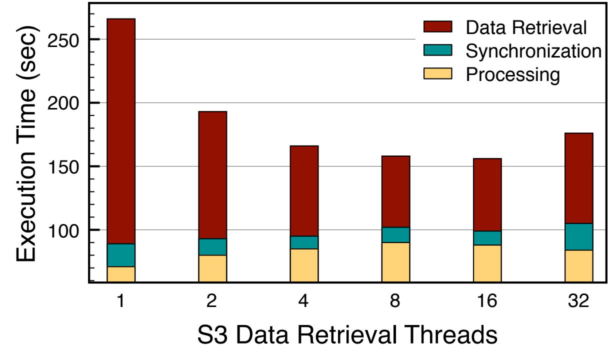


(b) PCA - 16 Data Retrieval Threads for Each Processing Thread

Fig. 4. Performance with Varying Chunk Sizes



(a) KMeans – 128MB Chunk Size



(b) PCA – 128MB Chunk Size

Fig. 6. Execution Times with Varying Number of Data Retrieval Threads

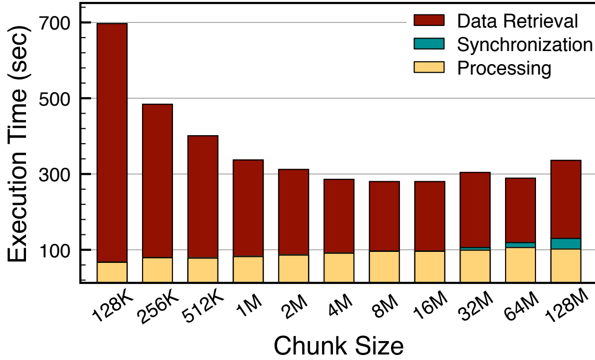


Fig. 5. KMeans – 1 Data Retrieval Thread for each Processing Thread

itself, whereas the 128MB size divides each data object into only 4 logical chunks.

This implies that the number of separate requests to S3 with a 128MB chunk size is significantly smaller than the configuration with 128KB chunk size. Considering the worst case, i.e., 128 KB, and the best performing cases, i.e., 16MB, 32MB, 64MB, and 128MB, the speedups of using larger data chunks range from 2.07 to 2.49. The execution times can further be analyzed according to their data retrieval, synchronization and processing times. Considering the 128KB chunk size as the base configuration, the speedups of data retrieval times with increasing chunk sizes range from 1.55 to 3.71 where 64MB chunk size configuration exhibits the best performance. The synchronization time does not introduce significant overhead to the system, however it increases while the chunk sizes increase. The slowdowns of processing times change from 17% to 59% with increasing chunk sizes, and the highest slowdown ratio was with 64MB configuration. Since, most of the execution time is spent during the data retrieval phase, the speedups of using larger chunk sizes dominate the slowdowns of processing times. Thus, chunk sizes have a very substantial impact on the performance while retrieving and processing data from S3. The minimum *data processing time* is with the 128 MB configuration, although the average time of the experiments with this configuration shows higher *overall execution time* than the 16MB, 64MB and 32MB configurations.

The number of retrieving threads used for each processing thread (or core) turns out to be a very important factor for overall performance. Thus, before detailed analysis of the trends observed in Figure 5, we also consider results for the case where threaded retrieval is used. Thus, as shown in Figure 4, the execution times of varying chunk sizes with 16 retrieval threads were observed.

Let us first consider the KMean application in Figure4(a). If the 8MB configuration is considered as the base setting, the speedups of 16MB, 32MB, 64MB and 128MB are 1.13, 1.20, 1.30, and 1.27, respectively. Similar to the previous experiment, the minimum execution times are observed with the 128MB setting, even though the average execution time is slightly higher than the 64MB. If we only consider the data retrieval times, the speedups range from 1.38 to 3.25, where the best performance is observed with 128MB chunk size configuration. Although

increasing chunk sizes introduce slowdowns, which are between 6% and 27%, in data processing times, this is again amortized with reductions in data retrieval times.

If we juxtapose Figures 5 and 4(a), we can clearly observe the effect of using multithreaded data retrieval, as the speedup of the 8MB setting with 16 retrieving threads over 1 retrieving thread is 1.24. Similarly, we can apply the same comparison to the other configurations: the speedups of using 16 retrieving threads are 1.40 for 16MB, 1.56 for 32MB, 1.65 for 64MB and 1.81 for 128MB. Clearly, using multithreaded data retrieval approach improves the performance of the execution of KMeans.

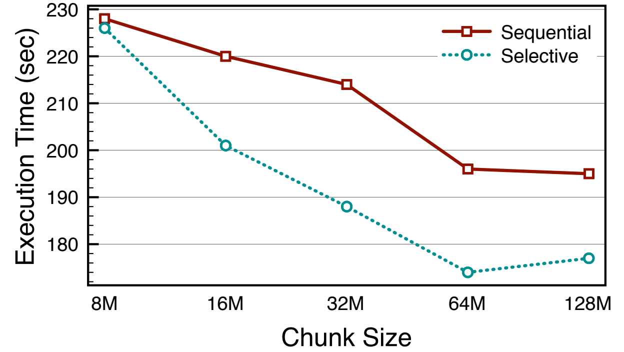
Figure 4(b) presents the same configuration with PCA application. The execution times follow the same pattern with KMeans. For this configuration, if we consider the 8MB setting as the base line, the speedups are 1.12, 1.30, 1.46, and 1.43 for 16MB, 32MB, 64MB, and 128MB chunk size configurations, respectively. The comparison of 1 threaded versus 16 threaded configurations follows a similar pattern that we had in KMeans application for PCA[4]. If we focus on the data retrieval times, the speedups change from 1.23 to 2.91, and the 128MB chunk size configuration provides the best performance.

In analyzing all experiments where chunk size is varied, we can see that, as data chunks increase in size, the processing times likewise increase in all cases. We believe this is because of the cache behavior on virtual cores and non-dedicated machines. With smaller chunk sizes, data can be cached effectively between retrieval and processing. At the same time, as chunk size increases, data retrieval times decrease, and this is likely due to the high latency of data accesses on S3. Thus, when the total number of distinct chunks to be read is smaller, we have lower data retrieval times.

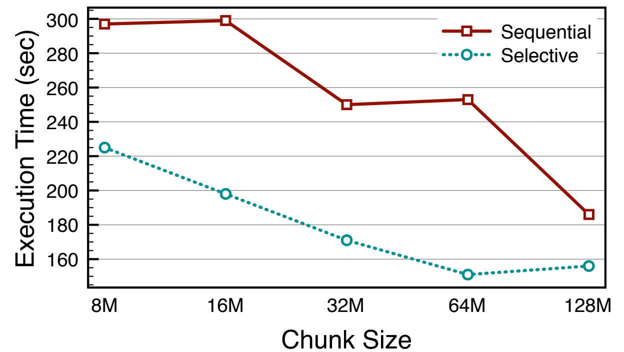
Another interesting observation pertains to synchronization times, i.e., the time taken to synchronize threads for transitioning from the data processing phase to the combination/reduction phase. These tend to increase while the chunk sizes become larger. We believe this is due to job granularity. That is, when all data chunks in the system have been consumed, the processing threads must aggregate all reduction objects and calculate the final result. At this point, all processes are required to interact and wait for each other. If the job size is large, all the threads must wait on the thread that was scheduled the final job. Therefore, synchronization time is impacted by the chunk size, as well as the throughput of the final process. In combining the three factors, the best performance is achieved with 64MB and 128MB chunk sizes. In general, however, this can be application dependent. In the future, we will add a performance model and/or an auto-tuning framework in our system to automatically select the optimal chunk size.

Continuing our study of different performance parameters, we next varied the number of data retrieval threads. The results are shown in Figures 6(a) and 6(b). We used 128MB chunk size as our default configuration. For both of applications, as the number of retrieving threads increase, the data transfer speed expectedly also increases, causing overall execution times to decrease. However, after 16 retrieving threads we begin experiencing diminishing returns due to scheduling overheads when there are too many threads. Our results show that the speedups of using many retrieving threads against 1 retrieving thread range from 1.38 to 1.71 for PCA, and from 1.37 to 1.90 for KMeans. Moreover, the speedups in data retrieval times are between 1.74 and 5.36 for KMeans, and 1.71 to 3.15 for PCA. For both applications, the best performance was observed with 16 retrieval threads.

[4]Results for the single thread version of PCA not shown due to space constraints



(a) KMeans – 16 Data Retrieval Threads

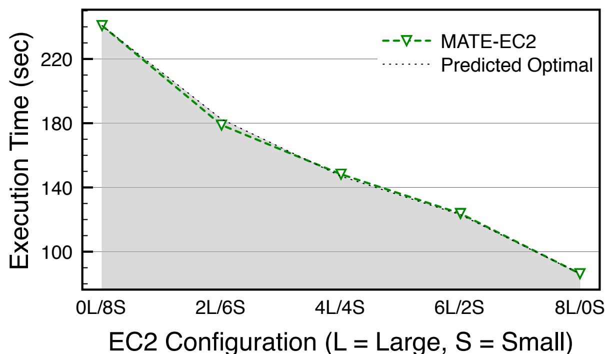


(b) PCA – 16 Data Retrieval Threads

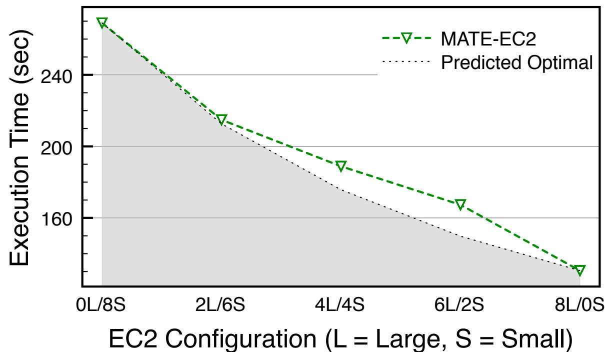
Fig. 7. Chunk Assignment Strategies, Varying Chunk Sizes

The next set of experiments addresses the effects of data assignment. Our threaded data retrieval approach opens many connections to the targeted S3 object. If several processes are assigned to the same data object, then the number of connections becomes a bottleneck. Without a selective job assignment scheme, the scheduler assigns jobs *sequentially*. Because logical chunks are consecutively placed among the data objects, consecutive job requests from various processes are mapped to the same data object. This results in a flood of connections on the same object, which slows down data retrieval. Conversely, a *selective* job assignment strategy, where the scheduler can choose to assign jobs to minimally accessed data objects, may offer performance gains.

In Figure 7(a) and 7(b) we show the execution times of KMeans and PCA with `selective` and `sequential` job assignment strategies. The speedups of using `selective` assignment for KMeans are 1.01 for 8MB and range from 1.10 to 1.14 for the other chunk size configurations. The reason why our approach does not show a significant speedup for 8MB configuration is due to the small chunk size that results in short data chunk retrieval times. This implies that there is little oppor-



(a) KMeans – 128MB Chunk Size, 16 Data Retrieval Threads



(b) PCA – 128MB Chunk Size, 16 Data Retrieval Threads

Fig. 8. MATE-EC2 in Heterogeneous Environments

tunity where a single object would be simultaneously accessed by multiple threads. The speedups of using `selective` assignment for PCA are 1.32, 1.51, 1.47, 1.68, and 1.19 for 8 MB, 16 MB, 32 MB, 64 MB, and 128 MB settings, respectively. The difference between the execution times are more clear in PCA because the application consists of two iterations, which can lead to the requirement of retrieving every data element twice. These results also indicate that splitting data into several data objects and concurrently processing them can lead to better performance than working on one large data object.

C. Performance of MATE-EC2 on Heterogeneous Environments

In the previous subsection, we empirically determined that the optimal configuration for MATE-EC2 involves 16 retrieving threads, a 64MB or 128MB data chunk size, and use of the selective job assignment strategy.

In this set of experiments, we evaluate our middleware on heterogeneous environments using this optimal configuration. In order to provide a heterogeneous environment, we ran our middleware on varying number of Amazon EC2 small and large instances. Small instances provide 1.7 GB memory and one virtual core with one elastic compute unit on a 32-bit architecture. Moreover, the I/O performance of small instances has been rated as being *moderate*. This is in contrast to large instances (7.5 GB memory, 2 virtual cores where each virtual core has 2 elastic compute units, and high I/O ratings). Therefore, we emphasize that heterogeneity does not only exist in terms of processing rate, but also I/O performance.

In our experiments, we used a total of 8 instances, but varied the mix of small and large instances. The *L* and *S* labels in Figures 8(a) and 8(b) refer to the large and small instance types respectively. For example, 6L/2S denotes that 6 large instances and 2 small instances were used.

Results in Figures 8(a) and 8(b) show a comparison between the actual execution times and *Predicted Optimal* times. The *Predicted Optimal* times reflect the perfect use of the aggregate processing power of a heterogeneous configuration. To calculate these, we first derive the ratio between the performance of the given application on the two homogeneous configurations (0L/8S and 8L/0S). This ratio indicates the throughput capacity of a large instance over a small instance, considering both the I/O and computation performance of the instances. Using this ratio, we can further compute the predicted execution times of each heterogeneous configurations.

We can now analyze the overheads of our middleware on heterogeneous environments. In Figure 8(a), we display the execution times of KMeans, whose overheads above the predicted optimal values are near 1% on all configurations. Similarly, Figure 8(b) shows the execution times of the same environment with PCA application. The overheads for PCA are 1.1% for 6S/2L, 7.4% for 4S/4L and 11.7% for 2S/6L configurations.

Even though we show that the execution times on heterogeneous environments are close to the *Predicted Optimal* times, two observations can be made: (1) The overheads increase while the number of large instances grow in the system and (2) PCA's overheads are higher than those of KMeans. We believe the reason for observation (1) is due to the synchronization issues that rise because of the difference between large and small instance types' I/O and throughput performance. Our approach for calculating the overheads assumes perfect timing for synchronization among the instances due to the ratio calculation which is being derived from homogeneous environments. The reason for observation (2) is because PCA involves two iterations, and therefore, the processes must synchronize twice. Because the granularity of jobs ultimately determines the synchronization intervals during execution, chunk sizes have a direct effect on such heterogeneity-related overheads. Therefore, the overheads due to the heterogeneity can further be controlled with smaller chunk sizes, i.e., jobs. Conversely, however, this may also cause additional overhead due to the inefficient bandwidth usage that we showed in the previous subsection.

V. Related Work

An increasing number of data-intensive projects have been migrated onto the Cloud since its emergence, and in particular, the celebrated class of Map-Reduce [3] applications. Driven by its popularity, Cloud providers including Google App Engine [7], Amazon Web Services, among others, began offering Map-Reduce as a service.

Several efforts have addressed issues in deploying Map-Reduce over the Cloud. For instance, the authors propose a preliminary heuristic for cost-effectively selecting a set of Cloud resources [12]. Related to performance, Zaharia, *et al.* analyzed *speculative execution* in Hadoop Map-Reduce and revealed that its assumption on machine homogeneity reduces performance [22]. They proposed the *Longest Approximate Time to End* scheduling

heuristic for Hadoop, which improved performance in heterogeneous environments. While our findings indeed echoed the authors' observation that heterogeneous nodes can be executed more effectively, our work nonetheless differs where processing patterns on data in S3 can lead to optimizations. In another related effort, Lin *et al.* have developed MOON (MapReduce On Opportunistic eNvironments) [14], which further considers scenarios where cycles available on each node can continuously vary. The dynamic load balancing aspect of our middleware has a similar goal, but our work is specific to AWS.

Many efforts have conducted cost and performance studies of using Cloud environments for scientific or data-intensive applications. For instance, Deelman *et al.* reported the cost of utilizing Cloud resources to support a representative workflow application, Montage [5]. They reported that CPU allocation costs will typically account for most of the cost while storage costs are amortized. Because of this, they found that it would be extremely cost effective to cache intermediate results in Cloud storage. Palankar *et al.* conducted an in-depth study on using S3 for supporting large-scale computing [17]. In another work, Kondo *et al.* compared cost-effectiveness of AWS against volunteer grids [13]. Yuan *et al.* proposed a strategy [21] for caching for large-scale scientific workflows on clouds. A tangential study by Adams *et al.* discussed the potentials of trading storage for computation [1]. Weissman and Ramakrishnan discussed deploying Cloud proxies [20] for accelerating web services. The goal of our work, in contrast, has been to study retrieval and processing of data stored in S3 and developing a middleware for data-intensive computing.

VI. Conclusions and Future Work

Recently, users are becoming increasingly attracted by the benefits in using Cloud computing resources for data-intensive projects. The Cloud's on-demand resource provisioning and infinite storage are highly suitable for various large-scale applications. This paper addressed two main issues in supporting data-intensive applications on Amazon Web Services, which have emerged as the most popular provider among the available Cloud services. The first issue is understanding the performance aspects of retrieving and processing data from Amazon S3. The second is developing an efficient and scalable middleware with a high-level API for data-intensive computing.

Our experiments with S3 have shown that high efficiency is achieved by having about 16 data retrieval threads for each processing thread, using a chunk size 64MB or 128 MB, and by minimizing the number of connections on each data object in S3. Using these observations, we have developed MATE-EC2, a middleware for facilitating the development of data processing services on AWS. We have shown that this middleware outperforms Amazon Elastic MapReduce by a factor of 3 to 28. Also, we are able to support data processing using a heterogeneous collection of instances in AWS, thus offering more flexibility and cost-effectiveness to users.

In the future, we would like to expand our work in several directions. First, we will experiment with additional applications. Second, a performance model and/or an autotuning framework is needed to automatically select chunk sizes and number of retrieving threads. Finally, we will like to extend MATE-EC2 to enable *cloud bursting*, i.e., use Cloud resources as a complement or alternative to local resources. Specifically, we wish to process data stored on a combination of local resources and S3, while using processing power from both EC2 and local resources.

References

[1] I. F. Adams, D. D. Long, E. L. Miller, S. Pasupathy, and M. W. Storer. Maximizing efficiency by trading storage for computation. In *Proc. of the Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2009.

[2] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. R. Bradski, A. Y. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In *Proceedings of the Twentieth Annual Conference on Neural Information Processing Systems (NIPS)*, pages 281–288. MIT Press, 2006.

[3] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[4] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of OSDI*, pages 137–150, 2004.

[5] E. Deelman, G. Singh, M. Livny, B. Berriman, and J. Good. The cost of doing science on the cloud: the montage example. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[6] D. Gillick, A. Faria, and J. Denero. Mapreduce: Distributed computing for machine learning. 2008.

[7] Google app engine, http://code.google.com/appengine.

[8] Hadoop, http://hadoop.apache.org/.

[9] W. Jiang, V. Ravi, and G. Agrawal. A Map-Reduce System with an Alternate API for Multi-Core Environments. In *Proceedings of Conference on Cluster Computing and Grid (CCGRID)*, 2010.

[10] R. Jin and G. Agrawal. A middleware for developing parallel data mining implementations. In *Proceedings of the first SIAM conference on Data Mining*, Apr. 2001.

[11] R. Jin and G. Agrawal. Shared Memory Parallelization of Data Mining Algorithms: Techniques, Programming Interface, and Performance. In *Proceedings of the second SIAM conference on Data Mining*, Apr. 2002.

[12] K. Kambatla, A. Pathak, and H. Pucha. Towards optimizing hadoop provisioning in the cloud. In *HotCloud'09: Proceedings of the 2009 conference on Hot topics in cloud computing*, Berkeley, CA, USA, 2009. USENIX Association.

[13] D. Kondo, B. Javadi, P. Malecot, F. Cappello, and D. P. Anderson. Cost-benefit analysis of cloud computing versus desktop grids. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[14] H. Lin, X. Ma, J. S. Archuleta, W. chun Feng, M. K. Gardner, and Z. Zhang. Moon: Mapreduce on opportunistic environments. In S. Hariri and K. Keahey, editors, *HPDC*, pages 95–106. ACM, 2010.

[15] J. Li, *et al.* escience in the cloud: A modis satellite data reprojection and reduction pipeline in the windows azure platform. In *IPDPS '10: Proceedings of the 2010 IEEE International Symposium on Parallel&Distributed Processing*, Washington, DC, USA, 2010. IEEE Computer Society.

[16] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[17] M. R. Palankar, A. Iamnitchi, M. Ripeanu, and S. Garfinkel. Amazon s3 for science grids: a viable solution? In *DADC '08: Proceedings of the 2008 international workshop on Data-aware distributed computing*, pages 55–64, New York, NY, USA, 2008. ACM.

[18] C. Ranger, R. Raghuraman, A. Penmetsa, G. R. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of 13th International Conference on High-Performance Computer Architecture (HPCA)*, pages 13–24. IEEE Computer Society, 2007.

[19] C. Vecchiola, S. Pandey, and R. Buyya. High-performance cloud computing: A view of scientific applications. *Parallel Architectures, Algorithms, and Networks, International Symposium on*, 0:4–16, 2009.

[20] J. Weissman and S. Ramakrishnan. Using proxies to accelerate cloud applications. In *Proc. of the Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2009.

[21] D. Yuan, Y. Yang, X. Liu, and J. Chen. A cost-effective strategy for intermediate data storage in scientific cloud workflow systems. In *IPDPS '10: Proceedings of the 2010 IEEE International Symposium on Parallel&Distributed Processing*, Washington, DC, USA, 2010. IEEE Computer Society.

[22] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *OSDI*, pages 29–42, 2008.