Supporting Dynamic Load Balancing in a Parallel Data Mining Middleware

Tekin Bicer Gagan Agrawal
Department of Computer Science and Engineering
The Ohio State University
Columbus, OH 43210
{bicer,agrawal}@cse.ohio-state.edu

Abstract

As parallel data mining applications are being executed in grid and cloud settings, there is a need for considering virtualized, non-dedicated, and/or heterogeneous environments. Supporting dynamic load balancing becomes an important challenge in such environments. Particularly, two important problems that need to be addressed are: Optimal distribution of tasks among disparate processing units and minimizing the runtime overhead of the system. With these goals, this paper describes and evaluates an approach for enabling parallel data mining with dynamic load balancing. Our approach is based on an API which deals with independent data elements that can be processed by any processing resource in the system.

We have extensively evaluated our dynamic load balancing system using two parallel data mining applications. Our results show that the overheads of our scheme are extremely low. Furthermore, our system successfully distributes tasks among processing units even in highly heterogeneous configurations.

1 Introduction

Increasingly, data mining needs to be performed in grid and cloud environments, leading to the supporting execution in non-dedicated and/or heterogeneous clusters. As science has become increasingly data-driven, support for data-intensive computing is becoming a crucial component of the cyber-infrastructure or e-science. For example, *community-driven data grids* have received significant attention recently [2]. Grids inherently comprise heterogeneous resources, and often include non-dedicated use of resources.

More recently, the trend is towards data-intensive computing on the emerging cloud environments. Two common characteristics of cloud environments are also leading to non-dedicated use of resources, and/or execution in heterogeneous environments. The first is the use of virtualization technologies, which enable applications to set up and deploy a customized virtual environment suitable for their execution. The second is the *pay-as-you-go* model for resource allocation and pricing. Consistent with the utility vision of computing, recent research points to the progression of clouds towards supporting fine-grained sharing of CPU cycles (and memory) between instances [3, 4]. Current virtualization technologies (for example, Xen [5]) can already allow a change in CPU cycle percentage and/or memory allocation at any point during the execution. Thus, in a cloud environment, it is quite possible that an application may be executed on a set of machines that differ in CPU cycle percentage allocation, and furthermore, this allocation can even change over time for each node.

Overall, there is clearly a need for executing data-intensive applications with dynamic load balancing, and harnessing the net processing power available in the cluster. This paper presents an approach for addressing this problem. Our approach has been implemented in the context of a data-intensive computing middleware, FREERIDE-G [6, 7]. This middleware system uses a specialized API for developing scalable data-intensive applications. It supports *remote data analysis*, which implies that data is processed on a different set of nodes than the ones in which it is hosted. Our work on supporting dynamic load balancing exploits the processing structure supported by our API, particularly, the fact that independent data elements can be processed by any processing resources in the system. This enables us to dynamically assign tasks to the processing units without considering the order or dependencies of the tasks.

We have evaluated how effectively our dynamic load balancing system can perform using two data mining applications. Our results show that the overheads of our system are negligible. Furthermore, our load balancing approach can effectively distribute jobs among the processing units even in highly heterogeneous configurations.

2 Background

This section gives an overview of an API on which our work is based. We then describe the remote data analysis paradigm and the FREERIDE-G system, which uses this API and supports remote data analysis.

```
FREERIDE
{* Outer Sequential Loop *}
While() {
    {* Reduction Loop *}
    Foreach(element e) {
        (i, val)    =   Process(e) ;
        RObj(i)   =   Reduce(RObj(i),val) ;
    }
    Global Reduction to Combine RObj
}

Map-Reduce
{* Outer Sequential Loop *}
While() {
    {* Reduction Loop *}
    Foreach(element e) {
        (i, val)    =   Process(e) ;
    }
    Sort (i,val) pairs using i
    Reduce to compute each RObj(i)
}
```

Figure 1: Processing Structure: FREERIDE(top) and Map-Reduce(bottom)

2.1 API for Parallel Data-Intensive Computing: Before describing our alternative API, we initially review the map-reduce API which is now being widely used for data-intensive computing.

The map-reduce programming model can be summarized as follows [8]. The user of the map-reduce library expresses the computation as two functions: *Map* and *Reduce*. *Map*, written by the user, takes a set of input points and produces a set of intermediate $\{key, value\}$ pairs. The map-reduce library groups together all intermediate values associated with the same key and passes them to the *Reduce* function. The *Reduce* function, also written by the user, accepts a key and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically, only zero or one output value is produced per *Reduce* invocation.

Now, we describe the alternative API this work is based on. This API has been used in a data-intensive computing middleware, FREERIDE, developed at Ohio State [9, 10]. This middleware system for cluster-based data-intensive processing shares many similarities with the map-reduce framework. However, there are some subtle but important differences in the API offered by these two systems. First, FREERIDE allows developers to explicitly declare a reduction object and perform updates to its elements directly, while in Hadoop/map-reduce, the reduction object is implicit and not exposed to the application programmer. Another important distinction is that, in Hadoop/map-reduce, all data elements are processed in the map step and the intermediate results are then combined in the reduce step, whereas in FREERIDE, both map and reduce steps are combined into a single step in which each data element is processed and reduced before the next data element is processed. This choice of design avoids the overhead due to sorting, grouping, and shuffling, which can be significant costs in a map-reduce implementation.

Hadoop/map-reduce provides *Combiner* function which can partially decrease the sorting, grouping and data transfer overheads. More specifically, if a Combiner function is defined in the system, the pairs are grouped in different lists according to their key values on local machine. When the number of pairs exceeds a threshold, Combiner function reduces the pairs and emits the new ones. Typically, Combiner function is similar to Reduce function, however it processes the pairs that are already in local memory. Reduce function, on the otherhand, needs to collect the emitted pairs. FREERIDE-G processing structure naturally accumulates $\{key, value\}$ pairs right after their generation which avoids the mentioned overheads in map-reduce.

The following functions must be written by an application developer as part of the API:

Local Reductions: The data instances owned by a processor and belonging to the subset specified are read. A local reduction function specifies how, after processing one data instance, a *reduction object* (declared by the programmer), is updated. The result of this process must be independent of the order in which data instances are processed on each processor. The order in which data instances are read from the disks is determined by the runtime system.

Global Reductions: The reduction objects on all processors are combined using a global reduction function.

Iterator: A parallel data-intensive application comprises of one or more distinct pairs of local and global reduction functions, which may be invoked in an iterative fashion. An iterator function specifies a loop which is initiated after the initial processing and invokes local and global reduction functions.

Throughout the execution of the application, the reduction object is maintained in main memory. After every iteration of processing all data instances, the results from multiple threads in a single node are combined locally depending on the shared memory technique chosen by the application developer. After local combination, the results produced by all nodes in a cluster are combined again to form the final result, which is the global combination phase. The global combination phase can be achieved by a simple all-to-one reduce algorithm. If the size of the reduction object is large, both local and global combination phases perform a parallel merge to speed up the process. The local combination and the communication involved in the global combination phase are handled internally by the middleware and is transparent

to the application programmer.

Fig. 1 further illustrates the distinction in the processing structure enabled by FREERIDE and map-reduce. The function $Reduce$ is an associative and commutative function. Thus, the iterations of the for-each loop can be performed in any order. The data-structure *RObj* is referred to as the reduction object.

Our recent work has shown a substantial performance improvement with our API [11]. In addition, we believe that this API offers a significant advantage in supporting dynamic load balancing. Since, almost all of the execution time is spent in the local reduction stage, the processing can be distributed between the nodes in a non-uniform and dynamic fashion. In comparison, with a map-reduce API, for most applications, significant amount of time is spent on both map and reduce stages. Moreover, the reduce stage is dependent on a large intermediate data structure, which can make dynamic load balancing very difficult to support.

2.2 Remote Data Analysis and FREERIDE-G: Our support for dynamic load balancing is in the context of supporting *transparent remote data analysis*. In this model, the resources hosting the data, the resources processing the data, and the user may all be at distinct locations. Furthermore, the user may not even be aware of the specific locations of data hosting and data processing resources.

If we separate the concern for supporting dynamic load balancing, co-locating data and computation, if feasible, achieves the best performance. However, there are several scenarios co-locating data and computation may not be possible. For example, in using a networked set of clusters within an organizational grid for a data processing task, the processing of data may not always be possible where the data is resident. There could be several reasons for this. First, a data repository may be a shared resource, and cannot allow a large number of cycles to be used for processing of data. Second, certain types of processing may only be possible, or preferable, at a different cluster. Furthermore, grid technologies have enabled the development of *virtual organizations* [12], where data hosting and data processing resources may be geographically distributed.

The same can also apply in cloud or utility computing. A system like Amazon's Elastic Compute Cloud has a separate cost for the data that is hosted, and for the computing cycles that are used. A research group sharing a dataset may prefer to use their own resources for hosting the data. The research group which is processing this data may use a different set of resources, possibly from a utility provider, and may want to just pay for the data movement and processing it performs. In another scenario, a group sharing data may use a service provider, but is likely to be unwilling to pay for the processing that another group wants to perform on this data. As a specific example, the San Diego Supercomputing Center (SDSC) currently hosts more than 6 Petabytes of data, but most potential users of this data are only allowed to download, and not process this data at SDSC resources. The group using this data may have its own local resources, and may not be willing to pay for the processing at the same service provider, thus forcing the need for processing data away from where it is hosted.

When co-locating data and computation is not possible, remote data analysis offers many advantages over another feasible model, which could be referred to as *data staging*. Data staging implies that data is transferred, stored, and then analyzed. Remote data analysis requires fewer resources at the data analysis site, avoids caching of unnecessary or *process once* data, and may abstract away details of data movement from application developers and users.

We now give a brief overview of the design and implementation of the FREERIDE-G middleware. More details are available from our earlier publications [13, 7]. The FREERIDE-G middleware is modeled as a *client-server* system, where the *compute node* clients interact with both *data host* servers and a *code repository* server. The overall system architecture is presented in Figure 2.

A data host runs on every on-line data repository node in order to automate data retrieval and its delivery to the end-users' processing node(s). Because of its popularity, for this purpose we used Storage Resource Broker, a middleware that provides distributed clients with uniform access to diverse storage resources in a heterogeneous computing environment. The code repository is used to store the implementations of the FREERIDE-G-based applications, as specified through the API. A compute node client runs on every end-user processing node in order to initiate retrieval of data from a remote on-line repository, and perform application specific analysis of the data, as specified through the API implementation. The processing is based on the generic loop we described earlier, and uses application specific iterator and local and global reduction functions.

Figure 2 demonstrates the interaction of system components. Once data processing on the compute node has been initiated, data index information is retrieved by the client and a plan of data retrieval and analysis is created. In order to create this plan, a list of all data chunks is extracted from the index. From the work-list a schedule of remote read requests is generated to each data repository node. After the creation of the retrieval plan, the SRB-related information is used by the compute node to initiate a connection to the appropriate node of the data repository and to authenticate such connection. The connection is initiated through an SRB Master, which acts as a main connection daemon. To service each connection, an SRB Agent is forked to perform authentication and other services, with MCAT metadata catalog providing necessary information to the data server. Once the data repository connection has been authenticated, data re-

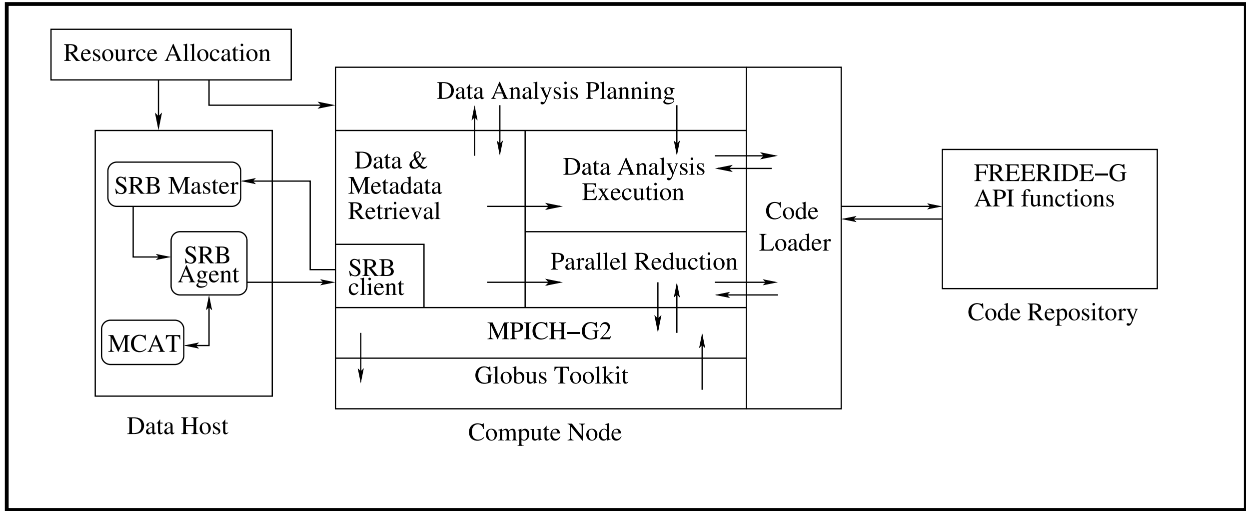

Figure 2: FREERIDE-G System Architecture

trieval through an appropriate SRB Agent can commence. To perform data analysis, the code loader is used to retrieve application specific API functions from the code repository and to apply them to the data.

3 Supporting Dynamic Load Balancing for Remote Data Analysis

In this section, we describe our dynamic load balancing approach and its implementation in the context of FREERIDE-G.

3.1 Our Approach: In the previous version of our middleware, the workflow of the application was set at the very beginning of the execution. Moreover, the jobs were evenly distributed among the compute nodes and each compute node was responsible for processing only its own jobs. If the compute nodes have different processing powers, the static job distribution may result in a large slowdown. Specifically, the compute nodes which have high throughput will have to wait until the slowest compute node finishes its execution.

Our approach for supporting dynamic load balancing exploits the properties of the processing structure of FREERIDE-G. Let us consider the processing structure supported by our middleware, shown earlier in Figure 1. Assume that the set of data elements to be processed is E. Furthermore, suppose a subset E_i of these elements is processed by the processor i, resulting in $RObj(E_i)$. Let G be the *global reduction function*, which combines the reduction objects from all nodes, and generates the final results.

The key observation in our approach is as follows. Consider any possible disjoint partition $E_1, E_2, \ldots, E_n$ of the processing elements between n nodes. The result of the

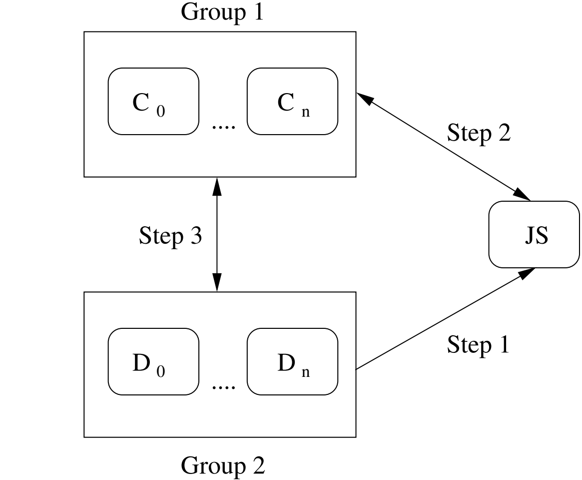

Figure 3: Load Balancing System's Workflow

global reduction function,

$$G(RObj(E_1), RObj(E_2), \ldots, RObj(E_n)) \quad (3.1)$$

will be same for any such disjoint partition of the element set E. In other words, if $E_1, E_2, \ldots, E_n$ and $E'_1, E'_2, \ldots, E'_n$ are two disjoint partitions of the element set E, then,

$$G(RObj(E_1), RObj(E_2), \ldots, RObj(E_n)) = G(RObj(E'_1), RObj(E'_2), \ldots, RObj(E'_n)) \quad (3.2)$$

If we examine the Equation (3.2), we can conclude that the processing structure that supports independent data elements can be exploited for dynamic load balancing system.

Specifically, any element, E_i, can be requested by any processing unit in the system during the execution. Therefore, the processing units which have high throughput can request and process more data elements than the others.

```
Input: dataNodes, List of data nodes that were
       registered to job scheduler
Result: job, which is assigned to compute node

/* Execute request handler loop   */
while true do
    compNode ← ReceiveReq();
    if CheckAssigned(compNode) then
        SetProcessed(compNode, dataNodes);
    end
    dataNode ← AvailDataNode(dataNodes);
    chunkNumb ← GetChunkNumb(compNode);
    job ← CreateJob(dataNode, chunkNumb);
    Transfer(job, compNode);
    if IsNotEmpty(job) then
        Assign(compNode,dataNode);
    end
end
```

Algorithm 1: Assigning jobs to Compute Nodes

```
Input: numbChunks, Number of chunks per job
 request
     : scheduler, Job Scheduler
Result: Final ReductionObject

/* Execute outer sequential loop  */
while true do
    /* Execute job request loop    */
    while true do
        job ← RequestJob(numbChunks, scheduler);
        if CheckJob(job) then
            break;
        end
        dataNode ← GetDataNode(job);
        chunksInfo ← GetChunksInfo(job);
        foreach chunk info cinfo in chunksInfo do
            {* Retrieve data chunk chk with cinfo
            from dataNode *};
            {* Process retrieved data chunk *};
            {* Update reduction object *};
        end
    end
    {* Perform Global Reduction *};
end
```

Algorithm 2: Processing Chunks on Compute Node

However, implementing such a dynamic scheme is also challenging. If all compute nodes request every chunk from a central scheduler, the overheads can be very high. Thus, while our approach is based on a central *job scheduler*, this scheduler works at a higher granularity. A compute node makes a request for a *set* of chunks to the job scheduler. The scheduler, then, returns a job, which includes the data node and the set of assigned chunk information. This chunk information consist of the exact offset addresses of the data elements in the data node. When the compute node receives the job, it starts retrieving the specified chunks from the data node.

A compute node again contacts the scheduler after the set of chunks have been retrieved from the data host and processed. This process is repeated until there is no more data to be processed. This scheduler is able to balance the workload between compute nodes with different processing power, while keeping the overheads very low.

3.2 Detailed Design and Implementation: We now discuss how our approach is implemented in FREERIDE-G. Figure 3 shows the interaction among the system elements. *Group 1* refers to the compute nodes, $C_{0 \cdots n}$, which are responsible for the processing of the data elements. Data nodes are represented with $D_{0 \cdots n}$ in *Group 2* where the data elements, i.e. chunks, are stored. It should also be noted that Group 1 and Group 2 are geographically separated. The *JS*, job scheduler, collects the necessary data information from Group 2 and then distributes the jobs to the compute nodes in Group 1.

Initially, the metadata information about the data needs to be prepared by data nodes in Group 2. The data in the system is stored in several files in which data is packed in data chunks (block). The metadata information about these chunks are stored into an index file. In this, each data chunk location is described with a data file name, offset address and the size of the data chunk. Each of these index information also corresponds to a metadata information of the smallest job in the system. Several of these index information can be combined and coarse-grained jobs can be generated.

After index information is generated, each data node in Group 2 prepares its specific node information which consists of the address information, available bandwidth of the data node and the chunk information of the data. It is then registered to the scheduler.

Job scheduler, on the other hand, waits for the data node registration requests. Whenever a registration request is received, the scheduler adds the data node information to the *dataNodes* list. This interaction is illustrated by *Step 1* in Figure 3.

After data nodes are registered and chunk information are specified in the scheduler, the job requests are handled. Algorithm 1 shows how the scheduler manages the compute nodes, which is also shown in Figure 3 with *Step 2*. At first,

the scheduler waits for the job requests from the compute nodes. When a job request is received, the scheduler checks if any of the chunks in the system was previously assigned to the requesting compute node. If so, the scheduler sets them as processed. Then, it creates another job with a new set of chunk information from the most suitable data node in the system. The main consideration for the scheduler in choosing a data node is effectively dividing the available bandwidth from each data node. Therefore, if the bandwidth between all pairs of compute nodes and data nodes is the same, the data node mapping will be done in a round robin fashion. If available bandwidths vary, more compute nodes' requests will be mapped to the data nodes with higher bandwidth, as long as they still have data that needs to be processed. Since the data and compute nodes are geographically separated, the bandwidth utilization and data node mapping are crucial for the overall execution time. After creating the job from a data node, it is transfered to the requesting compute node.

When a compute node receives a job from scheduler, it extracts the chunk information and starts requesting the chunks from the corresponding data node. With the retrieval of the data, the compute node starts executing the local reduction phase. When the data processing stage is finished, the compute node asks for another job. This continues until all the chunks are consumed. At last, all compute nodes finalize their execution with global combination. This process is shown in Algorithm 2.

4 Experimental Results

In this section, we report results from a number of experiments that evaluate our approach for supporting load balancing.

Two data-intensive applications we used are k-means clustering and Principal Component Analysis. Most of our experiments with k-means used a 25.6 GB dataset, whereas a 17 GB dataset was used for PCA. While our experiments with k-means involved only 1 iteration over the dataset, those for PCA had 3 iterations. As a result, the total amount of requested data is 51 GB for PCA application. All the datasets are divided into 4096 data blocks. Therefore, the sizes of each data block for k-means and PCA are 6.4 MB and 4 MB, respectively.

The configuration used for our experiments is as follows. Our compute nodes have dual processor Opteron 254 (single core) with 4GB of RAM and are connected through Mellanox Infiniband (1 Gb). We report experiments from the use of 4, 8, and 16 computing nodes. The number of data hosting nodes is always 4, and the data blocks are evenly distributed among these 4 nodes.

4.1 Effectiveness and Overheads: In this set of experiments, we evaluated the effectiveness and overheads of our

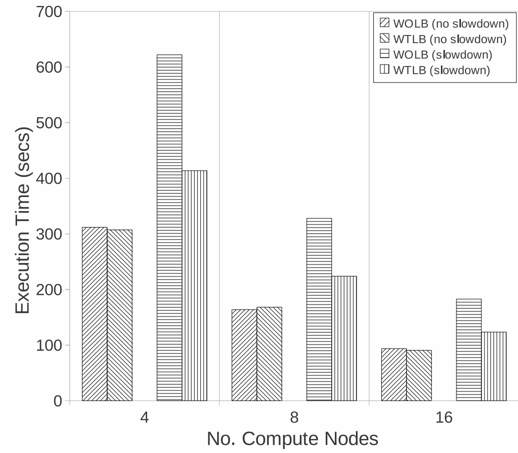

Figure 4: Evaluating Overheads using K-means clustering (25.6 GB dataset)

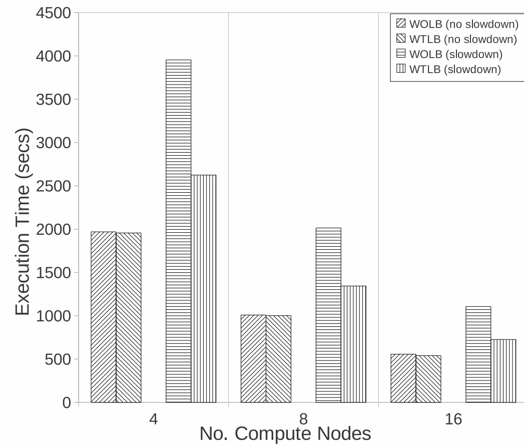

Figure 5: Evaluating Overheads using PCA (17 GB dataset)

dynamic load balancing system. For this experiment, we executed each of the two different versions of the middleware in two different environments. The two versions of the middleware are: Without load balancing system support, `WOLB`, and with load balancing system support, `WTLB`. `WOLB` is the first version of the FREERIDE-G, and is not able to balance the load among the compute nodes. Thus, the data elements are distributed evenly between the compute nodes and all the compute nodes have to wait until the slowest compute node finishes its processing. On the other hand, the enhanced version, `WTLB`, can dynamically balance the load between compute nodes.

The two environments in which we executed these two versions were the regular *homogeneous* cluster, and an environment with `slowdown`. Here, half of the compute nodes

are slowed down by 50% of their real processing power. The regular environment is also referred to as `no slowdown`.

In Figure 4, we present results from k-means application. The overheads of the load balancing system in `no slowdown` environments are 2.69% for 8 and close to 0% for 4 and 16 compute node cases. This shows that the implementation of our dynamic scheme is very efficient, and does not cause noticeable overheads.

In the `slowdown` environment, the speedups of the system range from 1.46 to 1.50 over the static partitioning system.

We can further analyze the costs of our load balancing implementation. The *absolute overhead* of the system can be calculated with the expected execution time of FREERIDE-G with `WOLB (slowdown)` configuration , say $time_{exp}$, which has the perfect data distribution among its compute nodes; and the execution time of `WTLB (slowdown)` configuration , say $time_{wtlb}$. The perfect data distribution for `WOLB (slowdown)`, in this case, means the data distribution among the compute nodes that satisfy the same execution time for every compute node. For instance, if half of the compute nodes in the system are limited to use 50% percentage of their CPU power and can process 450 data chunks during the execution, then the compute nodes which have 100% percentage CPU utilization are expected to process 900 data chunks in the same time period. With such configuration, the data chunks are perfectly distributed and the execution time of the system, $time_{exp}$, is optimum. Consequently, the absolute overhead can be found with:

$$overhead_{absolute} = \frac{time_{wtlb} - time_{exp}}{time_{exp}} \qquad (4.3)$$

If we apply (4.3) to Figure 4, then the absolute overheads of our system are again close to 0% for the three compute node cases. The retrieval and the processing time of the assigned data dominate the communication time between the scheduler and the compute nodes. Moreover, the scheduler can successfully select the appropriate data node in which compute node can benefit from the available bandwidth and maximize its data transfer speed.

In Figure 5, the same combination of versions and environments are repeated with PCA as the application. The overheads of the `no slowdown` version and the absolute overheads are close to 0%. The speedups of our system with `slowdown` version change from 1.49 to 1.52, considering without load balancing system with `slowdown` version.

4.2 Overheads With Different Slowdown Ratios: The experiments that we reported in previous section, half of the compute nodes were 50% slowed down. In this subsection, we evaluate our system's performance with two additional slowdown ratios: 25% and 75%. These slowdowns are applied to half of the compute nodes, again.

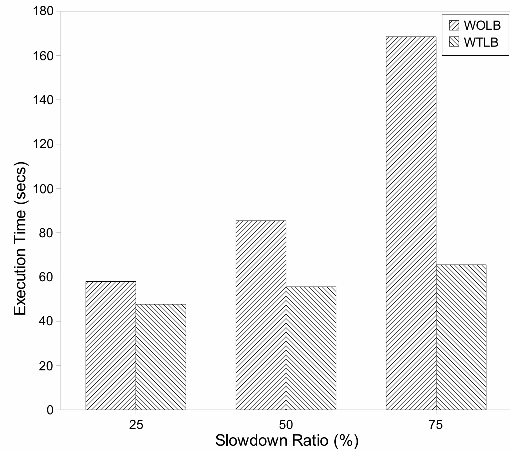

Figure 6: K-means clustering with Different Slowdown Ratios (6.4 GB dataset, 8 comp. nodes)

In Figure 6, we evaluated the k-means clustering application with 8 compute nodes and a 6.4 GB dataset. The absolute overheads, in each case, are close to 0%. The speedups of our system with respect to `WOLB` are 1.21, 1.53, and 2.57 for 25%, 50% and 75% slowdowns, respectively. The higher slowdown ratios indicate longer execution times for slow processing units. Furthermore, the system should wait for the slowest processing unit in case of static job assignment, i.e. `WOLB` configuration. On the other hand, the faster compute nodes can consume slow compute nodes' data elements with `WTLB` configuration which results in high speedups.

Same experiment was repeated with PCA and the results are shown in Figure 7. The absolute overheads are 3.50%, 1.63% and 4.07% for the three cases. Furthermore, the speedups over the static case are 1.23, 1.49 and 2.42 for 25%, 50% and 75% slowdown ratios, respectively. As we mentioned before, the PCA has three iterations, and requests more data elements (3 times) than k-means application. Moreover, the volume of retrieved data is significantly larger than the other configurations. Thus, the overheads become more visible.

4.3 Distribution of Data Elements with Varying Slowdown Ratios: In this section, we focused on how successfully our system distributes data elements among the compute nodes. In previous sections, the slowdown ratios were kept same for all the applied compute nodes. However, in this set of experiments, we varied the slowdown ratios between 8 compute nodes. More specifically, the slowdown ratios are increased by 12.5% for each of the compute node starting from 0%.

We showed the number of processed data elements for each of the compute node in Figure 8. The last bar in

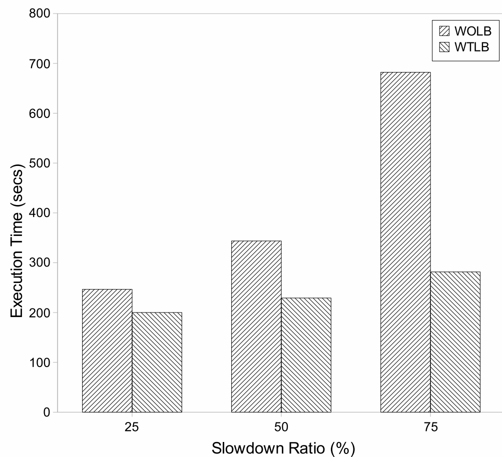

Figure 7: PCA with Different Slowdown Ratios (4 GB dataset, 8 comp. nodes)

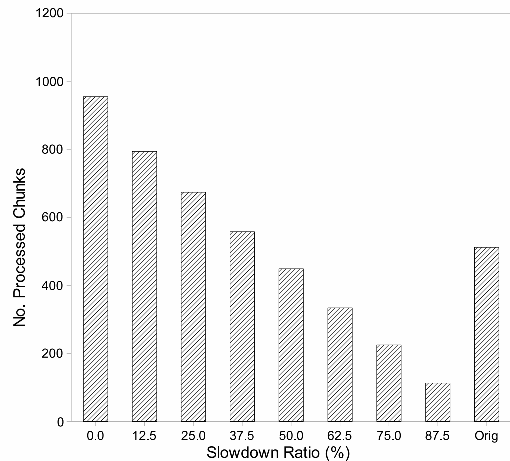

Figure 8: K-means clustering with Different Slowdown Distribution (6.4 GB dataset, 8 comp. nodes)

the figure shows the expected number of data elements that need to be processed in case of perfect CPU utilization for all compute nodes. The number of processed chunks ranges from 109 to 161 for consecutively slowed down processing units. The absolute overhead of our system is again very close to 0%. These results show that our system can successfully distribute the chunks even in a highly heterogeneous environment.

In Figure 9, we repeated our experiment with PCA application. Note that the total number of chunks is three times more than the k-means clustering application due to the iterations. The number of processed chunks ranges from 233 to 477 for consecutively slowed down compute nodes. The absolute overhead in this case is 8.5%. The basic reasons of this overhead are the high imbalance in the CPU utilization among the processing units, and the number of iterations which results in more job requests to the scheduler.

4.4 Overheads with Different Assignment Granularity:

In all experiments reported in previous subsections, the scheduler was set to assign 4 data chunks for every job request. In this subsection, the number of assigned data chunks per request is varied. K-means clustering application with 6.4 GB dataset was used for the evaluation; and the assigned data chunks per request were changed to 4, 16, 64 and 256, respectively.

The results are shown in Figure 10. The absolute overheads are 0.72%, 1.58%, 6.31% and 16.01% for 4, 16, 64, and 256 data chunks cases, respectively. The load balancing system's overhead increases with the increasing number of data chunks per request, and a fine-grained assignment results in the best performance for our system. This is because the overheads of dynamic load balancing are still very low with fine-grained assignments. At the same time, the performance with coarse-grained assignments is worse, because we do not achieve perfect load balance.

5 Related Work

The topics of data-intensive computing and map-reduce have received much attention within the last 2-3 years. Projects in both academia and industry are working towards improving map-reduce. CGL-MapReduce [14] uses streaming for all the communications, and thus improves the performance to some extent. Mars [15] is the first attempt to harness GPU's power for map-reduce.

Yahoo's map-reduce system, Hadoop, is one of the popular implementations. Even though, our system and Hadoop share important similarities, the differences are significant. Hadoop assigns tasks to the racks where data locality is maximized and high throughput is satisfied. Our system, on the other hand, works in the context of remote data analysis in which exploiting such locality is not possible. However, our system minimizes the data retrieval time through exploiting bandwidth usage of the data nodes which results in optimum computation throughput.

Lin et al. extended Hadoop with MOON [16] which provides better performance in unreliable volunteer computing systems using a small set of dedicated nodes. Our dynamic load balancing system extends our previous work which focuses on fault tolerance [17] in data-intensive computing environments. We believe our system can also perform well in such unreliable environments.

Farivar *et al.* introduced an architecture named MITHRA [18] and integrated the Hadoop map-reduce with the power of GPGPUs in the heterogeneous environments. Zaharia *et al.* [19] improved Hadoop response times by de-

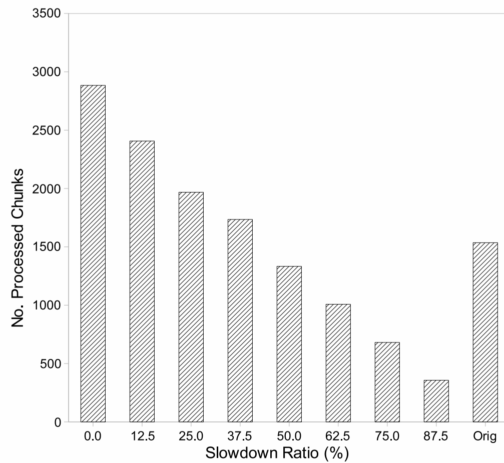

Figure 9: PCA with Different Slowdown Distribution (4 GB dataset, 8 comp. nodes)

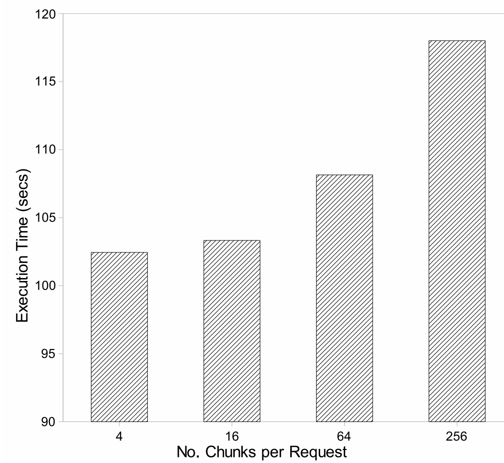

Figure 10: Performance with Different Number of Chunks per Request (k-means, 6.4 GB dataset, 4 comp. nodes)

signing a new scheduling algorithm in a virtualized data center. Seo *et al.* [20] proposed two optimization schemes, prefetching and pre-shuffling, to improve Hadoop's overall performance in a shared map-reduce environment. Ranger *et al.* [21] have implemented Phoenix, a map-reduce system for multi-cores.

Facebook uses *Hive* [22] as the warehousing solution to support data summarization and ad-hoc querying on top of Hadoop. Yahoo has developed *Pig Latin* [23] and Map-Reduce-Merge [24], both of which are extensions to Hadoop, with the goal being to support more high-level primitives and improve the performance. Google has developed *Sawzall* [25] on top of map-reduce to provide higher-level API. Microsoft has built *Dryad* [26], which is more flexible than map-reduce, since it allows execution of computations that can be expressed as *DAGs*.

OpenMP is an API which supports parallel shared memory processing on many architectures and supports different scheduling strategies such as static, dynamic and guided work sharing. However, the programmer should involve solving the data dependencies and synchronization issues. These are automatically handled by our framework.

6 Conclusions

In this work, we developed and evaluated a dynamic load balancing scheme for a data-intensive computing middleware. We focused on heterogeneous environments such as non-dedicated machines in grids and virtualized machines in clouds. We proposed an approach which effectively solves the problem of task distribution among processing units that show different processing performance.

Two data-intensive applications were used in order to evaluate the system. Our results show that the overheads of our system are very small. Moreover, our approach can successfully distribute tasks among processing units even in highly heterogeneous configurations.

References

[1] R. E. Bryant, "Data-intensive supercomputing: The case for disc," School of Computer Science, Carnegie Mellon University, Tech. Rep. Technical Report CMU-CS-07-128, 2007.

[2] T. Scholl, A. Reiser, and A. Kemper, "Collaborative query coordination in community-driven data grids," in *Proceedings of the Conference on High Performance Distributed Computing (HPDC)*, Jun. 2009.

[3] J. Heo, X. Zhu, P. Padala, and Z. Wang, "Memory overbooking and dynamic control of xen virtual machines in consolidated environments," in *Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management (IM09)*, June 2009, pp. 630–637.

[4] H. Lim, S. Babu, J. Chase, and S. Parekh, "Automated control in cloud computing: Challenges and opportunities," in *Proceedings of the 1st Workshop on Automated Control for Datacenters and Clouds (ACDC09)*, June 2009, pp. 13–18.

[5] P.Barham, B.Dragovic, K.Fraser, S.Hand, T.Harris, A.Ho, R.Neugebauer, I.Pratt, and A.Warfield, "Xen and the art of virtualization," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP03)*, 2003, pp. 64–177.

[6] L. Glimcher and G. Agrawal, "A Performance Prediction Framework for Grid-based Data Mining Applications," in *In proceedings of International Parallel and Distributed Processing Symposium (IPDPS)*, 2007.

[7] ——, "A Middleware for Developing and Deploying Scalable Remote Mining Services," in *In proceedings of Conference on Clustering Computing and Grids (CCGRID)*, 2008.

[8] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of OSDI*, 2004, pp. 137–150.

[9] 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.

[10] ——, "Shared Memory Parallelization of Data Mining Algorithms: Techniques, Programming Interface, and Performance," in *Proceedings of the second SIAM conference on Data Mining*, Apr. 2002.

[11] W. Jiang, V. T. Ravi, and G. Agrawal, "Comparing map-reduce and freeride for data-intensive applications," in *Proceedings of the 2009 IEEE Cluster*. IEEE, 2009.

[12] I. Foster, C. Kesselman, and S. Tuecke, "The Anatomy of Grid: Enabling Scalable Virtual Organizations," *International Journal of Supercomputing Applications*, 2001.

[13] L. Glimcher, R. Jin, and G. Agrawal, "FREERIDE-G: Supporting Applications that Mine Data Repositories," in *In proceedings of International Conference on Parallel Processing (ICPP)*, 2006.

[14] J. Ekanayake, S. Pallickara, and G. Fox, "Mapreduce for data intensive scientific analyses," in *IEEE Fourth International Conference on e-Science*, Dec 2008, pp. 277–284.

[15] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: a mapreduce framework on graphics processors," in *Proceedings of PACT 2008*. ACM, 2008, pp. 260–269.

[16] H. Lin, J. Archuleta, X. Ma, W. Feng, Z. Zhang, and M. Gardner, "Moon: Mapreduce on opportunistic environments," in *ACM International Symposium on High Performance Distributed Computing (HPDC)*, June 2010.

[17] T. Bicer, W. Jiang, and G. Agrawal, "Supporting fault tolerance in a data-intensince computing middleware," in *Proceedings of the 24th International Parallel and Distributed Processing Symposium (IPDPS)*, April 2010.

[18] R. Farivar, A. Verma, E. Chan, and R. Campbell, "Mithra: Multiple data independent tasks on a heterogeneous resource architecture," in *Proceedings of the 2009 IEEE Cluster*. IEEE, 2009.

[19] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *Proceedings of OSDI*. USENIX Association, 2008, pp. 29–42.

[20] S. Seo, I. Jang, K. Woo, I. Kim, J.-S. Kim, and S. Maeng, "Hpmr: Prefetching and pre-shuffling in shared mapreduce computation environment," in *Proceedings of the 2009 IEEE Cluster*. IEEE, 2009.

[21] C. Ranger, R. Raghuraman, A. Penmetsa, G. R. Bradski, and C. Kozyrakis, "Evaluating mapreduce for multi-core and multiprocessor systems," in *Proceedings of 13th HPCA*. IEEE Computer Society, 2007, pp. 13–24.

[22] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive - a warehousing solution over a map-reduce framework," *PVLDB*, vol. 2, no. 2, pp. 1626–1629, 2009.

[23] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: a not-so-foreign language for data processing," in *Proceedings of SIGMOD Conference*. ACM, 2008, pp. 1099–1110.

[24] H. chih Yang, A. Dasdan, R.-L. Hsiao, and D. S. P. Jr., "Map-reduce-merge: simplified relational data processing on large clusters," in *Proceedings of SIGMOD Conference*. ACM, 2007, pp. 1029–1040.

[25] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan, "Interpreting the data: Parallel analysis with sawzall," *Scientific Programming*, vol. 13, no. 4, pp. 277–298, 2005.

[26] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *Proceedings of the 2007 EuroSys Conference*. ACM, 2007, pp. 59–72.