DELMA: Dynamically ELastic MApReduce Framework for CPU-Intensive Applications

Zacharia Fadika [1], Madhusudhan Govindaraju [2]

Department of Computer Science, State University of New York (SUNY) at Binghamton
[1] `zfadika@cs.binghamton.edu`, [2] `mgovinda@binghamton.edu`

Abstract—Since its introduction, MapReduce implementations have been primarily focused towards static compute cluster sizes. In this paper, we introduce the concept of dynamic elasticity to MapReduce. We present the design decisions and implementation tradeoffs for DELMA, (Dynamically ELastic MApReduce), a framework that follows the MapReduce paradigm, just like Hadoop MapReduce, but that is capable of growing and shrinking its cluster size, as jobs are underway. In our study, we test DELMA in diverse performance scenarios, ranging from diverse node additions to node additions at various points in the application run-time with various dataset sizes. The applicability of the MapReduce paradigm extends far beyond its use with large-scale data intensive applications, and can also be brought to bear in processing long running distributed applications executing on small-sized clusters. In this work, we focus both on the performance of processing hierarchical data in distributed scientific applications, as well as the processing of smaller but demanding input sizes primarily used in small clusters. We run experiments for datasets that require CPU intensive processing, ranging in size from millions of input data elements to process, up to over half a billion elements, and observe the positive scalability patterns exhibited by the system. We show that for such sizes, performance increases with data and cluster size increases. We conclude on the benefits of providing MapReduce with the capability of dynamically growing and shrinking its cluster configuration by adding and removing nodes during jobs, and explain the possibilities presented by this model.

I. INTRODUCTION

Introduced at the USENIX Symposium on Operating Systems Design and Implementation (OSDI) in 2004 [1], MapReduce is a programming model that has steadily gained popularity in the scientific community. Before the application of MapReduce, Yahoo! took 26 days of processing to build automatic completion indexes for their search engine. By applying the MapReduce model, the same operation was reduced to 20 minutes with a cluster of computing nodes [2]. The model has been applied to various fields and disciplines, ranging from satellite observation [3], to earthquake simulation [4]. The appeal and performance displayed by this new paradigm spawn from its ability to apply uniform functions, *map* and *reduce* to data locally present on given processing nodes. All cluster nodes apply the same primitives provided by the programmer to data provided to them by the system as a result of input splitting. MapReduce was inspired by the functional programming construct *"map"*, which allows a user to define a function to be applied to an input set, then subsequently returns an output set of the same length containing elements resulting from the transformation operated by the given function. In

`Haskell` for example, *map* is a higher-order polymorphic function that applies a given function `f` to a list of elements, on an element to element basis. For instance, calling: `map square [1,2,4]` will yield `[1,4,16]`. This concept is used in MapReduce as the *map* function constitutes a procedure to be applied to a given input. In a different paradigm, perhaps procedural, this is akin to the application of a routine to a list of elements. The only constraint offered by the model is the requirement of taking an element from an input pool, submitting it to a procedure or process, and returning with a transformed or processed element.

The model, however efficacious, in its current form, can only be applied to static compute clusters. The ability to grow or shrink a MapReduce cluster, as a job is underway, is currently not available in widely used toolkits. Such a capability can offer significant advantages to the model and further its applicability, performance, and all around impact. While Cloud computing frameworks like Microsoft Azure [5] and Amazon EC2 [6] offer "elastic MapReduce", such elasticity as it pertains to MapReduce in Cloud computing thusfar can be applied in the form of instant node increase or decrease demands in between jobs, rather than during jobs. Similarly, while an iterative approach to MapReduce such as Twister's [7] permits for jobs to be broken down as to allow reconfiguration to occur in between sub-jobs, faced with a monolithic job as most traditional MapReduce jobs are, the possibility of reconfiguration is not available.

In this paper, we present DELMA for (Dynamically ELastic MapReduce), a MapReduce framework similar to Apache Hadoop, but with the additional capability for the cluster size to be reconfigured in an on-demand manner during job run-time. DELMA allows for node count to be increased or decreased, speeding up a job, permitting faulty nodes to be replaced, and transient nodes to be added for short periods of times, providing short performance boosts. Transient nodes, can later be seamlessly removed or replaced if needed. Importantly, DELMA allows these operations to take place without the need to terminate a given running task for which the additional nodes are destined. MapReduce has thusfar been used for large-scale data-centric applications, but can also be effective in dealing with small-sized data, requiring CPU intensive computing, and running in small-size clusters. In this paper, we present the design of DELMA, and evaluate the framework in a myriad of processing scenarios including progressive node additions at different stages in the job run-

time. We then conclude that performance can be improved by dynamically scaling a MapReduce cluster, up to 50% in the best cases, and up 10% in the worst cases, when nodes are added late into a job. The contributions of this paper are as follows:

- We introduce an elastic MapReduce framework, providing the ability for a cluster to be dynamically elastic, i.e. expand or reduce its size on-demand, without requiring for the job to be stopped and restarted but rather paused while reconfiguration occurs.
- We provide the application programmer the ability to start a given job early before all resources necessary for such a job are available, and then progressively add compute nodes as they become available to the user, thus saving valuable time.
- We provide the possibility for transient or voluntary nodes to be added to a MapReduce cluster whenever available.
- We provide the ability for slow jobs to be sped up on the fly by adding more resources once a given job is underway.

II. THE ARCHITECTURE OF A DYNAMICALLY ELASTIC MAPREDUCE PLATFORM

Apache Hadoop is an open-source MapReduce software framework [8]. Hadoop is in use in many scientific applications [4], and dwells as the classic, and so far best known implementation of the MapReduce paradigm. The framework relies on its own file system: the HDFS [9], a file system inspired from the Google Filesystem [1]. The HDFS offers its own tree structure, along with commands for providing input to applications. In the case of Hadoop, the HDFS also directly manages internal files and directory structures pertaining to application I/O. The file system in Hadoop operates an *InputSplitter* inherent to HDFS [1]. The role of the *InputSplitter* is primarily to dissect and distribute shards of input before starting the cluster, more precisely during input placement. The HDFS keeps track of file blocks by storing metadata information describing each block size and locations. The metadata is not only useful for block locality and processing, but it is also useful should node failure occur, as restarting a job requires redundant file blocks to be located and provided to available nodes for job re-runs. This is not only the case in HDFS, but a general standard in most high performance distributed file systems such as [10], Swift [11] and xFs [12], which rely on this system for file placements. The metadata information in Hadoop's case consists of a list of input files, file attributes such as creation time, replication factor, and a list of `DataNodes` hosting each block. The Hadoop cluster is also composed of a node designated as the `NameNode`. The `NameNode` is "de facto" the master node. Its role is not only to manage the file system but also to direct file access requests from worker nodes and `DataNodes`. The `NameNode` achieves this task by mapping a file block to a block identification number and a list of `DataNodes`. `DataNodes` themselves manage blocks and file storage located on the nodes controlling them. They map block-ids to physical disk locations. They also regulate I/O requests from and to other cluster nodes requiring it. Beyond file system considerations, the framework also consists of a `JobTracker`, usually the master node and a `TaskTracker`, often designated among worker nodes. The `JobTracker`'s responsibility is to schedule and monitor jobs across the cluster. This includes detecting and tagging failed and slow jobs, then duplicating slow executions, and restarting failed ones. The user is responsible for providing the *map* and *reduce* procedures, as well as input and output specifications. These procedures are provided to each node participating in the cluster. Hadoop gives the ability to users to specify the number of desired mappers and reducers. Having multiple mappers per node allows for concurrency to occur within a job itself, and procures an added benefit to architectures that thrive in concurrent executions. Mappers running on each node create a key/value pair for each unit of input processed. The key represents a reference to the input value, and is usually the input line itself. The value, depending on the nature of the application run by the user, is the dataset upon which the *map* operation takes place. Hadoop's *map*, much like in functional programming, applies the *map* function to every value in the input. This process produces an intermediary list of new key/value pairs, which is in turn passed to the reducer: `Map(`Key_1, v_1`)` $\rightarrow$ `list(`Key_2, v_2`)`. The reduction process then takes place with the intermediary list produced in the previous *map* step. Key/value pairs are supplied to the *reduce* function, which sorts and rearranges them by key. The function then iterates through each pair and applies the *reduce* logic based on similar keys. `Reduce(`Key_2`, list(`v_2`))` $\rightarrow$ `list(`v_2`)`. Before this task can take place, each intermediary list needs to be considered.

III. MOTIVATIONS FOR A DYNAMICALLY ELASTIC MAPREDUCE PLATFORM

The problem at hand deals with the fact that for all the operations described above to take place, Hadoop requires its cluster to be clearly defined by the time a given job is to be started. This is required for the HDFS to properly map blocks to the nodes holding them, and for providing a sense of locality in processing. Input chunks are already apportioned by application run-time, and so are replication blocks for fault-tolerance during the run. Once a job has started, Hadoop MapReduce does not allow for cluster membership changes, as its filesystem would not be able to maintain block mappings and integrity tallies. Nodes can neither be added nor replaced in the cluster while an application is underway. Similarly, node removal is not left to the user's choice, but rather to node failure.

With the widespread availability of small-size clusters in universities, national labs, and corporations, there exists an opportunity to apply the MapReduce paradigm for parallel applications that suit this computing model. However, the availability of such resources cannot always be guaranteed as being solely dedicated to a given scientific application.
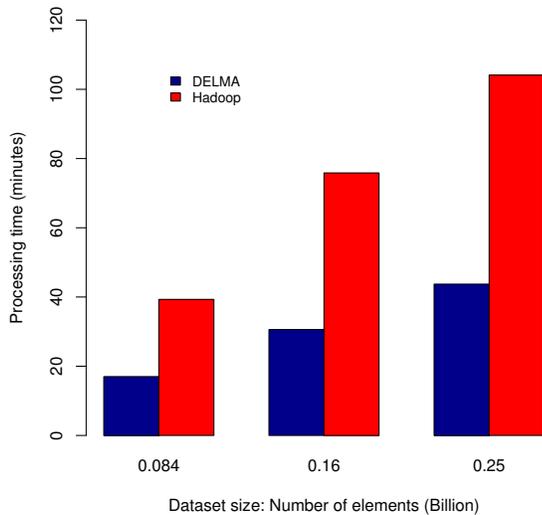
Fig. 1. Processing time comparison between a static Hadoop cluster running 16 nodes on various input sizes, then stopping, upgrading the cluster to 32 nodes, and running the job over. This is contrasted against a DELMA cluster starting with 16 nodes, and mid-way through the job's run-time, upgrading its clusters to 32 members, by dynamically adding 16 additional nodes. The performance difference stems from the fact that DELMA does not need to stop and restart a job in order to upgrade the node count.

The rigid nature of traditional MapReduce clusters means that applications cannot benefit from temporary computing power provided by temporarily available nodes, perhaps not available before the start of a given task, but rather during it. Such resources include volunteer computer systems and opportunistic frameworks, such as MOON [13]. MapReduce, as it currently stands, requires all its nodes to be accounted for before application run-time and does not allow node addition during run-time, lest the application be terminated, the cluster reconfigured and the job restarted.

The problems with the static nature of a MapReduce cluster size can be seen in **Figure 1**. For 84 Million data elements to process, the processing time taken by DELMA is shown as 43% better than that of Apache Hadoop. For 0.16 Billion and 0.25 Billion input elements, DELMA posts 53% and 60% improvement respectively compared to Hadoop.

We can conclude the following from Figure 1:

- The static nature of MapReduce's cluster size would result in significant time loss, especially for long running applications.
- Furthermore, for such applications, should node replacement become evident due to poor performing computers, the user is stuck with a static configuration of their cluster for the entirety of the job.
- Similarly, faulty or dead nodes cannot be replaced but can only pass their burden to peer nodes, which are already processing their own share of the work.

With Hadoop MapReduce, a user has the luxury of tracking

job progress. The linear nature of most MapReduce jobs, due to the uniformity of the *map* and *reduce* functions across all nodes, can help a user predict the remaining amount of time a job is likely to require. This is usually done by observing the progress report provided by the system. Such a report is however available after the job has started. In cases where the user needs to speed up a given job, they are left with no options. The ability to provide more nodes, and thus more power to applications while they are running, has thusfar not been explored in the MapReduce context. In a corollary condition, the inability to predict before making a job live, what the optimal cluster configuration should be for that particular job can create a condition where extraneous nodes hinder the performance produced by the optimal number of nodes required for a job [14]. A user in such straights would be unable to remove the extra nodes, and let the cluster run efficiently at its best, but would instead have to suffer the loss of performance, and wait to correct the condition in subsequent runs. One possible approach is to use a "Performance model" to predict cluster optimality. However, such an approach may not be able to address the need for node replacement in the face of faulty or dead machines. Even if one knows the adequate cluster size before a run, losing a node during that run would leave the cluster in a sub-optimal state. With DELMA, the dead node can be readily replaced. Similarly, the use of a "Performance model" does not solve the problem of resource unavailability, where a user might need to start a job with limited resources, then progressively supply the cluster with more nodes as those become available. All of these shortcomings to the traditional MapReduce model inspired us to devise an elastic version of the model. DELMA allows for:

- The ability for a cluster to grow or reduce its size on the fly, without requiring for the job to be stopped and restarted. This feature is common in Cloud computing frameworks such as Microsoft Azure, and Amazon EC2. Neither however allow for this possibility with their MapReduce installations, as they require a stateless nature to their framework.
- The ability for a job to be started early before all resources necessary for such job are available, and then progressively add compute nodes as those become available. This is a concept inspired from Cloud computing wherein nodes can be called in to participate in an already running task. This feature in Cloud computing does not however support cases where a node's task is dependent upon a partitioned input set, as it is the case in MapReduce.
- The ability for transient or voluntary nodes to be added to a MapReduce cluster whenever available, allowing for opportunistic systems to host MapReduce implementations.
- The ability for faulty or poor performing nodes to be replaced as a job is on-going, without the need to restart the already in-process job, and potentially losing valuable time.
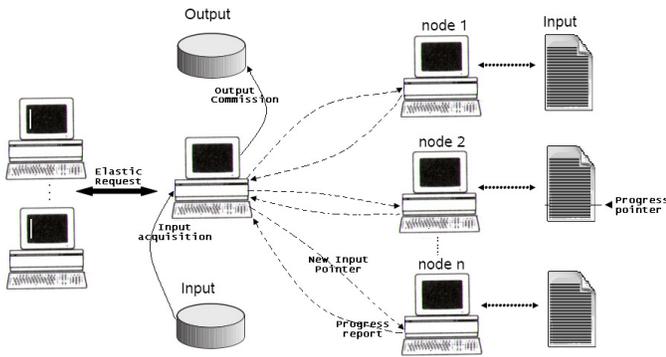- The ability for slow jobs to be sped up on the fly by

Fig. 2.   Upon an elastic (reconfiguration) request the master node pauses the job, accounts for progress reports from all participating nodes at the time of the request, collects the so far completed work and rebuilds an input unit based on the work left to be done. New nodes are either added in the case of expansion, or just removed or replaced in the case of reduction or replacement. New input is then communicated to the workers, and the job resumes anew.

adding more resources once a given job is underway, and the user gets an estimate of the completion time, given their current configuration.

A. Design considerations in DELMA

Like Hadoop MapReduce, DELMA allows for input to be managed through a "central manager". Application progress is also kept track of in both cases. DELMA however has the capability of receiving individual job progression tallies *on-demand* from all nodes present in the cluster. This allows the framework to request job suspension upon cluster upgrade or downgrade. During that time, the central manager accounts for both the processed and the "left to be processed" input from each nodes, through the completion progress reports. The completed output is saved, the unprocessed input is evaluated while new members to the cluster are accounted for. The unprocessed input is then redistributed among all the nodes, new nodes included. Existing nodes resume execution with their new input, while the new nodes start execution with their input as well. Cluster downgrade operates in a similar fashion, with the grid sustaining a pause, the remaining work accounted for and then redistributed to the smaller cluster size.

The key to DELMA's ability to reconfigure its clusters, is in its decoupling of input and nodes, and its job accounting capability. **Figure 2** shows the operation of the framework vis-a-vis cluster expansion and reduction. A chief concern while implementing this framework is the cost of reconfiguration which includes checkpointing and input redistribution. Early on, we established that for such an endeavor as DELMA to be productive, the cost of widely reconfiguring a running cluster should be trumped by the extra boost in performance provided by the nodes added to the system in the case of cluster expansion for performance. Throughout the rest of the paper, we focus more in depth on cluster expansion for performance. We assume that it is evident to the reader as to the performance gains harvested by replacing dead nodes with healthy ones, rather than letting the system run with
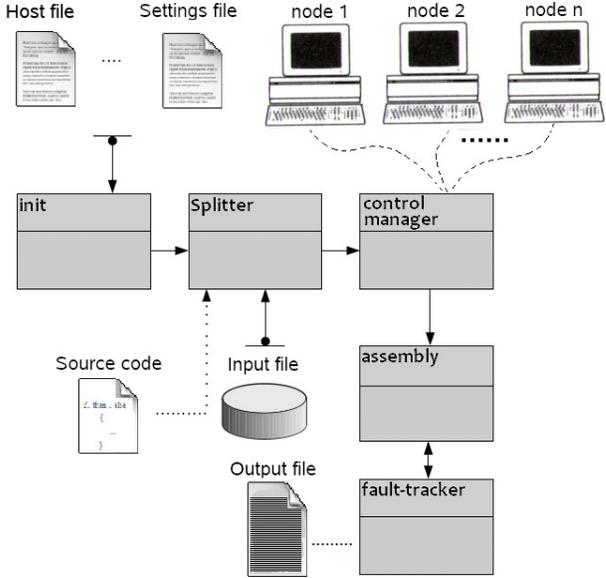

Fig. 3.   Architecture used for DELMA. The user provides a list of hosts to the host file, and a settings file. Upon launch, the *init* module parses both files, updates its data structure with relevant variables including input directory location, source code path, and number of hosts to use from the host file. The relevant information is then passed on to the *splitter* which splits the input and dispatches it to the chosen nodes. The *MapReduce Control Manager* then provides the worker nodes with running instructions. Once the *MapReduce Control Manager* (MCM) accounts for all the necessary completion signals, the *assembly* module consults the fault table through the *fault-module* and returns the flow to the MCM for re-runs if necessary. Alternatively, the *assembly* module assembles and provides the output to the user through the output directory specified in the settings file.

less resources than envisaged. We also assume as evident the negative impact of removing well performing nodes on application performance, and do not show the performance data here as it is inverse to node addition. Additionally, node removal is akin to node failure handled by fault tolerance in Hadoop and DELMA itself. The question of incertitude however dwells in the realm of node addition and performance especially when it comes to minimal node additions, and late node additions. In other words: would adding two nodes to a job be worth reconfiguring an entire cluster? Or would doubling the number of existing nodes 90% into a job completion time improve performance at all, given the fact that the task is nearing completion and could benefit from using the incurred reconfiguration time for job completion with the original sized cluster? These are the questions we strive to address and answer in section IV.

B. DELMA: Comparison with HADOOP

1) MapReduce Input Management: **Figure 3** shows the design used for our Dynamically ELastic MApreduce frame-work.

DELMA, like Hadoop MapReduce, features a *splitter* module which operates either as a dispatcher, or as a file management system. The *splitter*'s role is to divide the input into blocks to be provided to the worker nodes. In dispatcher mode,

input is sent each time the application is run. As part of file management functions, the central manager or master node keeps a mapping of nodes and blocks, and only reconfigures the block arrangements upon user request, new input supply, or run-time reconfiguration requests. Unlike Hadoop, this block mapping can be shifted from node to node, or redistributed to a bigger or smaller cluster. The *splitter* module, also unlike Hadoop MapReduce, operates directly with the underlying file system, thus eliminating potentially inhibiting layers of indirection between the application and the user provided input. The *splitter*'s role during reconfiguration is to account for the new size of the cluster and divide the remaining input to be processed accordingly. Hadoop's splitter does not allow for *on-the-fly* node expansion through input reconfiguration.

2) Master-Worker relationship: With the motivation for providing an efficient framework in a dynamically elastic setting, we aimed at mitigating potential performance inhibiting factors such as network constriction caused by extraneous Master to node communications. Glimcher et. al. [15] show the negative impact of such a condition in distributed environments. This led us to opt for an almost all on-demand approach to Master-Worker communications, ranging from job progress to an active fault-tolerance strategy. In DELMA, unlike Hadoop, the Master does not directly track node progress, but rather receives a completion signal, or sends an inquiry should it fail to receive one after a satisfactory number of nodes have done so. The Master also inquires about node progress when a user request for node increase, decrease or a replacement occurs. At the start of a job, the Master is tasked with starting the workers with their work. Upon a reconfiguration request, the master node pauses execution, collects progress reports, and accordingly fashions the input blocks left to be processed into a new input unit to be distributed to the new cluster configuration. The Master calls on the *splitter* module for input distribution, but not before providing each node with their *map* and *reduce* instructions in the form of the source code provided by the user. The master node, through the MapReduce Control Manager (MCM), is also tasked with receiving node health reports. Nodes advertise their condition by throwing an exception, which is caught and handled by the MCM. Dead nodes break a communication pipe between them and the MCM, throwing an exception handled by recording such nodes as unavailable and reassigning their work. These fault-tolerance operations are handled by the Fault-tracker. As jobs complete, the Master gradually collects completion stubs from each of its workers and upon reception of all completion signals, fetches back the intermediate output produced by each worker. Should the Master account for a satisfiable percentage of its workers, it can elect to entrust the completed nodes with the work still being performed by the active nodes, and then discard the late arrivals, whichever those are. This is similar to the mechanism used by Hadoop and other platforms to account for slow workers, also called stragglers [1].

3) Fault-tracker: The *Fault-Tracker* module keeps a tally of available and unavailable nodes. Failed nodes are marked as such and depending on the size of their input blocks, a block can either be entrusted to a completed node in the cluster or redistributed among all nodes. This is a system controlled setting, and depends on the chunk size of the input block processed by the failed node as well as the size of the cluster being dealt with. DELMA's fault-tolerance system itself is recursive, meaning failed rescues are themselves rescued until either all nodes die, causing the whole job to fail, or a user controlled timeout value is reached. Hadoop uses a heart-beat mechanism in order to keep track of its workers node. Our modular design is configurable and can incorporate a heart-beat based fault-tracking mechanism, similar to Hadoop, if needed.

IV. DISTRIBUTED LARGE-SCALE DATA PROCESSING

In this section, we set out to test the impact of cluster expansion turn-around time. We not only test the impact of early and late node additions, but also the impact of diverse cluster sizes on DELMA, as well as that of progressive node addition. Our prior work with MapReduce applications [14] has shown that for application turn-around time to be efficient, the overhead introduced by additional processing units must dwarf the time gained by the work produced by the participating nodes. In a traditional MapReduce context, this condition is encountered when input processing sizes are small, or when the processing per input element within the input itself is insignificant. In [14], we showed that a cluster of 5 Hadoop nodes could be outstaged by a single system for input sizes below a given threshold of data. In brief, a MapReduce job would yield poor performance if the time used to provide the nodes the input is significantly greater than the time spent by the collection of nodes processing that input. The idea of a dynamically elastic MapReduce concept, is not only subject to the same considerations, but must tackle the additional "burden" of node pause, input accounting, and input redistribution.

In **Figure 4** we isolate the time lost in reconfiguration for datasets containing 84 Million, 0.3, and 0.6 Billion data elements being streamed in, or processed from a file, while **Figure 5** shows the said overhead to be independent of cluster size, but rather dependent on workload.

The experiments that follow took place on both Hadoop MapReduce and DELMA clusters. Without dynamic node expansion, both clusters' performances are identical. Without node addition, DELMA would perform identically performance-wise to Hadoop with identical input in (form and size), identical node count, cluster configuration, and similar application code. In all the experiments showcased, we ran tests for DELMA along side Hadoop using identical nodes, identical data and similar user code.

We chose for these experiments the application of XML parsing by both clusters, because both scientific and business applications increasingly perform scalable processing of large
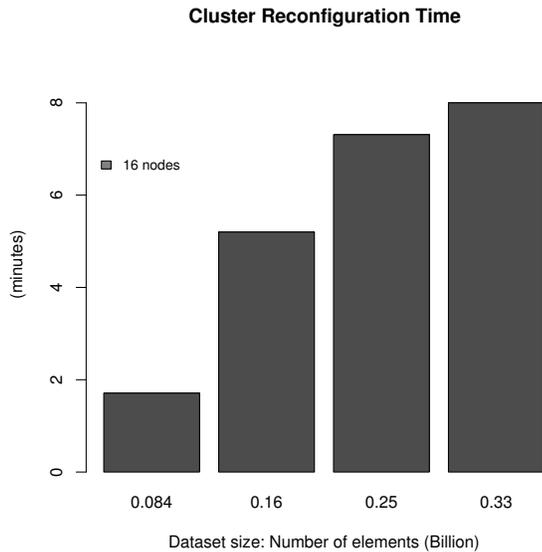
Cluster Reconfiguration Time

Fig. 4. Cluster reconfiguration time vis-a-vis input size. This graph shows that the setting up the data and nodes for MapReduce is heavily dependent on input size.
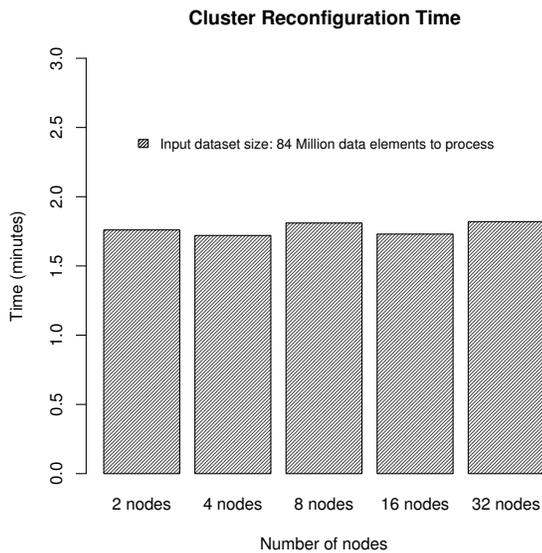
Cluster Reconfiguration Time

Fig. 5. Cluster resize time in the face of diverse cluster configurations ranging from 2 to 32 nodes. This figure displays a nearly similar reconfiguration time regardless of cluster size, and shows that cluster reconfiguration time is not affected by how big or small a cluster is.

datasets. The MetaData Catalog Service (MCS) [16] provides access via a Web service interface to store and retrieve descriptive XML information (metadata) on millions of data items. Popular toolkits such as Axis [17], Libxml [18], and Piccolo [19] are designed for small-sized data processing. In our experiments, we incorporated the use of AxisJava. AxisJava is web services-based toolkit boasting a processing intensive SOAP engine capable of creating SOAP processors allowing data content parsing. Resulting tokens from this parsing can then subsequently be streamed into application content and returned to the user. Our input data for testing purposes is composed of arrays of floating point numbers. In [20], [21], we have shown that AxisJava scales poorly for CPU intensive applications that require processing arrays of floating point numbers. The computation is intensive and demanding on the systems involved, regardless of overall data size. This led us in the decision of our application choice. XML is also a widely used data standard in many scientific, healthcare, and financial applications [22]. An advantage of the MapReduce model is that it has relaxed synchronization constraints, which works favorably for large-scale XML datasets, wherein typically namespaces that are once defined at the start of the document are not redefined in the inner-elements. MapReduce is a particularly good choice in such environments. The overall experimental data and analysis in this paper can be used to quantify the exact configuration to be used to gain performance enhancements for a given application setting. Even though we use XML to represent our input data, MapReduce applications, and by extension, DELMA applications can use any other scientific data formats and even hierarchical formats such as HDF5 [23] and NetCDF [24], whenever the MapReduce model is applicable. For the following experiments, we extracted the AxisJava parsing module and integrated it into our applications. Both Hadoop and DELMA run off similar source code powered by the AxisJava parsing engine. In all the experiments, we ran tests for DELMA along side Hadoop using identical node counts and input data.

V. PERFORMANCE RESULTS

We run our tests on a selection of two machine classes:

- $1\times$ dual core – One desktop-class machine, which has a single 2.4Ghz Intel Core2 6600 with 2 GB of ECC RAM, running Linux 2.6.24. The filesystem in use here is ext3fs.
- $2\times$ uniprocessor – 1U nodes in a cluster, each of which has two 3.2Ghz Intel Xeon CPUs, 4 gigabytes of RAM and run a 64 bit version of Linux 2.6.15. Results on this class of machines are taken by averaging the timings produced on these nodes. The filesystem in use in the test directory here is reiserfs.

In **Figure 6** We show the benefit of DELMA over static MapReduce clusters. While the Hadoop cluster runs with 16 nodes from job start to job end, DELMA receives 16 additional machines half-way (50%) through the application run-time. This request causes the system to pause its original execution, account for the work already done, register the new nodes,
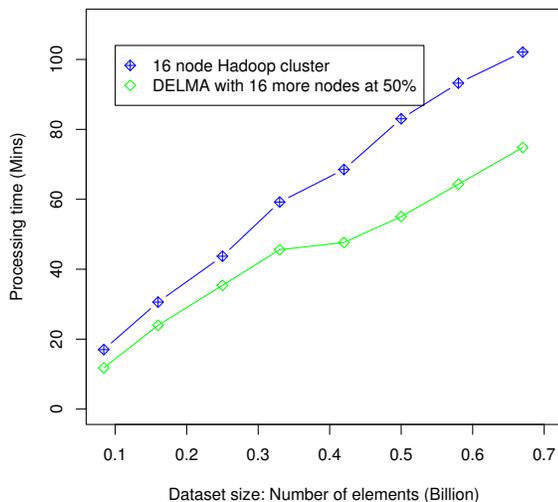
Fig. 6. 16 node Hadoop cluster versus a 16 node DELMA cluster expanded to 32 nodes 50% through the job. In this graph, DELMA receives a node expansion request half-way (50%) through the job with 16 additional nodes, and continues its running with the extra boost received for the diverse input sizes shown. The times displayed by DELMA are the total application times encompassing all the operations taking place in the framework, from the start of the job with 16 nodes, to the reconfiguration of the cluster to the resumption and completion of the run with 32 nodes.
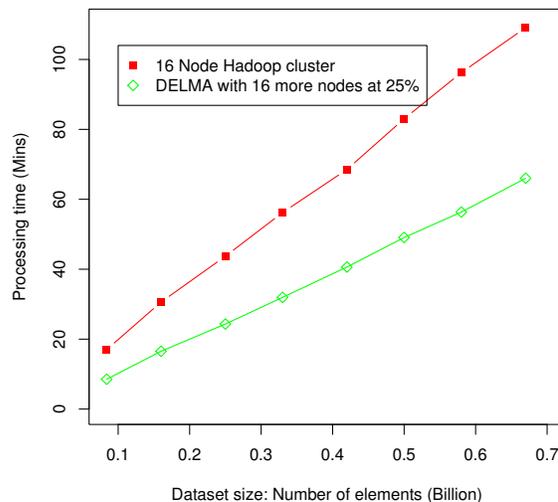

Fig. 7. 16 node Hadoop cluster versus a 16 node DELMA cluster expanded to 32 nodes 10% through the job. In this graph, DELMA receives a node expansion request 10% through the job with 16 additional nodes, and continues its running with the extra boost received for the diverse input sizes shown. The times displayed by DELMA are the total application times encompassing all the operations taking place in the framework, from the start of the job with 16 nodes, to the reconfiguration of the cluster to the resumption and completion of the run with 32 nodes.

redistribute the remaining input data and resume execution. Despite the overhead generated by reconfiguration, DELMA runs up to 30% faster than the static Hadoop cluster. For small input sizes, the impact of the dynamic cluster expansion is significantly lower in DELMA. However, as the input size grows, the added power shows an increasing performance gap in DELMA's favor. This can be explained by the fact that as input sizes grow, so does the need for more processing to efficiently deal with such input. **Figure 5** shows that as input grows so does reconfiguration delays. These delays are however overpowered by the work produced with the help of the additional nodes provided to the cluster.

Figure 7 shows a trend similar to that shown in **Figure 6**. In this case however, cluster expansion comes much earlier in the job's run-time. In **Figure 7** we decide to add 16 additional nodes to our DELMA cluster only 10% into the application's run-time. The first observation made confirms our hypothesis concerning node addition time and application performance. It reaffirms, the intuitive hypothesis that it is better to "add early." Later addition constitutes node addition, at 50% through a job as previously shown, or 75% through a job as shown next. As it can be observed, expanding our cluster early (10%) rather than 50% through a job increases the performance gap between the classic MapReduce framework and the elastic one. In this particular case, the job ran up to 52% faster on DELMA than on Hadoop. This is because the earlier the nodes are added, the bigger the cluster is for more data, and thus the processing takes place much faster. For very long jobs, it is reasonable

to assume that during the first 10% of the time, the user can analyze cluster progress and decide in favor of node addition, or faulty node replacement.

In **Figure 8** we show the impact of late cluster expansion in DELMA. Despite the relatively late nature of the cluster expansion, DELMA still posts faster execution time than its static counterpart. This graph however shows the mitigating effect of data size on late cluster resizing. With small data sizes, both framework post close times, but as the data sizes grow, the elastic MapReduce framework shows a clear demarcation from the static framework. This pattern follows much of the same trends observed in our two previous graphs.

In **Figure 9**, following **Figure 8**'s performance with late node addition, this figure shows node addition at the diverse completion stages of the user application. For this experiment, we chose the smallest input set to process. This is because as evidenced by prior graphs, big data sizes work in DELMA's favor by providing its additional nodes with more work, and thus more time gained. By performing significant node additions even up to 90% through the application completion time with our smallest input set, we can still clearly show the advantages of a dynamically elastic MapReduce approach over a classic MapReduce model. Even as the additional 16 nodes are added towards the end of our job, the time gains are still significant on the part of the elastic framework. These trends, as we have shown with bigger sizes in our previous graphs, increase with data size, and as we will subsequently show, also increase with cluster size. Even as the performance
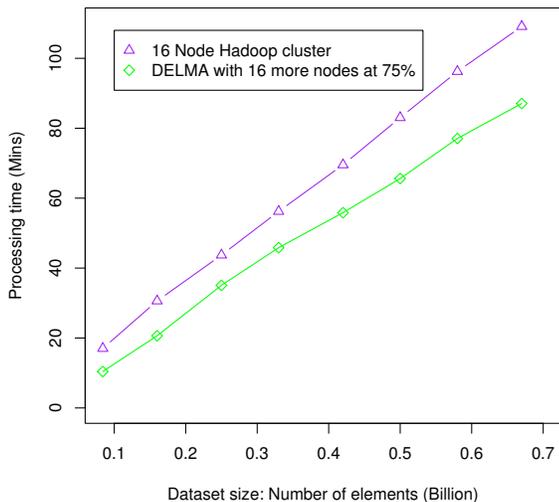
Fig. 8. 16 node Hadoop cluster versus a 16 node DELMA cluster expanded to 32 nodes 75% through the job. In this graph, DELMA receives a node expansion request late (75%) through the job with 16 additional nodes, and continues its running with the extra boost received for the diverse sizes shown. The times displayed by DELMA are the total application times encompassing all the operations taking place in the framework, from the start of the job with 16 nodes, to the reconfiguration of the cluster to the resumption and completion of the run with 32 nodes.

Fig. 10. This graph shows a progressive addition of nodes to the job for DELMA, while Hadoop runs with a static configuration of 16 machines. With, DELMA we add 2,3,4,5,6 up to 10 nodes progressively in two cases. The blue line shows such additions at 10% through the job in each case, and the red line shows the same additions at 50% through the job. In the blue line's case, the job starts like Hadoop with 16 nodes, then 10% through the job, 2 nodes are added, then the job is allowed to complete. Subsequent runs are taken with 3,4,5 up to 10 added nodes. At its biggest, the DELMA cluster starts with 16 nodes and finishes the job with 26. The input dataset is composed of over a quarter billion data elements. Both Hadoop and DELMA run off similar source code and identical input datasets.

difference shows a gap with the processing of 84 Million data elements with the cluster expanded at 90% through the application completion, **Figure 6**, **7**, and **8** show that the performance gap would further grow for much bigger input sets. The more work is required through bigger input, the more useful additional nodes begin to be, and the more performance gain they produce. This figure also shows what it would be like for the system to be constantly stressed with repetitive node additions. In our future work section, we lay out how we plan to intensify such stress tests in an evaluation of DELMA in highly volatile opportunistic grids.

As observed from **Figure 10**, as more nodes are progressively added, the faster a job will run. With the addition of a small number of nodes, it is interesting to note that node addition overhead is proportionally lower than the performance produced by such added nodes. In practice, with constant overhead, perhaps, it could be that a threshold would have been needed for cluster performance. Here, however, we are able to add as little as 2 nodes and still notice an even slight performance improvement over the static Hadoop framework. Further additions also fare better than both Hadoop and previous smaller DELMA clusters. As expected however, the DELMA cluster expanded ever so slightly at 10% through the application runs faster than its subsequent run with nodes added at 50% (late addition).
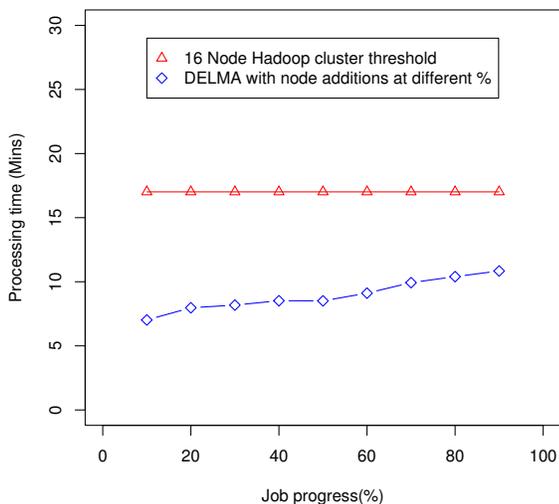

Fig. 9. This figure shows the parsing of 84 Million data elements by both Hadoop and DELMA. While Hadoop runs with a cluster of 16 nodes, DELMA's cluster size is expanded to 32 nodes at 10%, 20%, up to 90% through the job run-time.
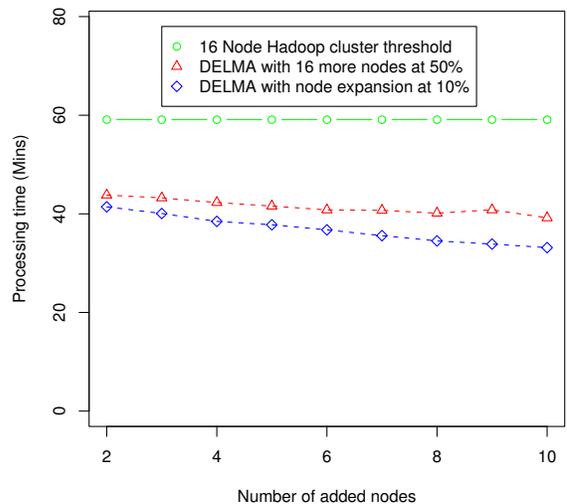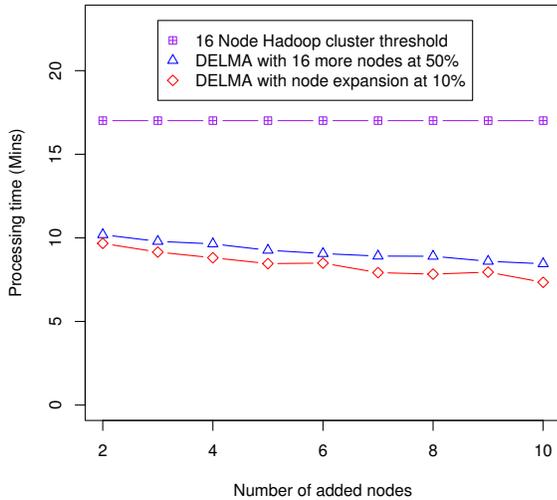
Fig. 11. This graph shows a progressive addition of nodes to the job for DELMA, much like the previous graph, but this time for an 84 Million element dataset. Hadoop here runs with a static configuration of 16 machines. With, DELMA we add 2,3,4,5,6 up to 10 nodes progressively in two cases. The blue line shows such additions at 10% through the job in each case, and the red line shows the same additions at 50% through the job. In the blue line's case, the job starts like Hadoop with 16 nodes, then 10% through job completion, 2 nodes are added and the job is allowed to complete. Subsequent runs are taken with 3,4, and up to 10 added nodes. At its biggest here, the DELMA cluster starts with 16 nodes and finishes the job with 26. At its smaller, the cluster starts with 16 nodes and finishes the job with 18.

Figure 11 shows an alternate case to **Figure 9** with a smaller input file. As **Figure 6, 7, 10** show, big input file sizes can help DELMA perform much faster by providing its added nodes more work. Faced with that observation, it was then imperative to showcase the previous graph with our smallest data set to confirm the trends previously observed. As shown, the performance advantage over Hadoop is still strongly maintained. The need for cluster expansion however when faced with small sizes and small node addition only yields a small advantage. In this graph, the user gains a 10% improvement of the application run-time by early node addition over late node addition. 10%, for Terabyte data sizes however is a massive time saving.

VI. Related Work

MOON [13] studies the applicability of MapReduce to opportunistic systems. Although MOON touches on cluster volatility and the ability for MapReduce to survive in opportunistic environments such as that offered by Condor [25], the proposed framework does not address load redistribution, but rather load passing from voluntary nodes to dedicated nodes in the cluster. Amazon has produced EC2 [6], a cloud computing framework allowing for MapReduce applications to be implemented. In a similar fashion, Microsoft has produced Azure [5]. While EC2 and Azure provide elasticity in

a grid environment, they do not tackle input redistribution as required in jobs targeted towards a shared input source. With Microsoft Azure and Amazon EC2, users can request additional nodes to jobs operating independently from each other, lest the user implement themselves a data redistribution algorithm. GridBus [26] in a different approach, focuses on data intensive applications much like Condor [25], [27] by employing a *job broker* to match an application to the best nodes capable of running it most efficiently. The analysis and the node choice in the case of GridBus is driven by available resources on the compute nodes. Jobs, also are not simultaneously run like in DELMA, but rather, sequentially pipelined. Twister [7] is a MapReduce framework espousing an iterative approach to solving MapReduce problems. Despite its iterative nature, Twister does not not allow for machines to be added or removed after a job has started and before it has finished. The iterative approach of Twister allows for tasks to be broken down as to permit reconfiguration to occur in between sub-jobs, but faced with a monolithic task which is the target application domain for DELMA, the possibility of reconfiguration is not available in Twister. None of the works highlighted above as of yet provide the full set of capabilities and implementations provided by DELMA, especially cluster expansion and shrinkage in MapReduce with a highlight of performance benefits against a static MapReduce approach.

VII. Conclusions

MapReduce has over the years steadily grown to be a successful and widely used data processing model. The model has been applied in almost all disciplines and domains, be it space monitoring or data mining to search engine technology. However, since its inception, MapReduce has only been implemented with rigid and static cluster configurations. The widely available implementations do not yet support cluster elasticity, and as such, the ability to dynamically add, remove, or replace nodes as a MapReduce job is underway has so far been nonexistent. Throughout this paper, we showed that the model can be expanded to include this essential feature. We presented DELMA (Dynamically ELastic MApReduce), a framework just like Hadoop MapReduce, but capable of supporting various node addition and removal requests during run-time. Throughout our experiments, we showed that DELMA posts significant performance compared to an already efficacious MapReduce framework: Apache Hadoop. Besides traditional benefits offered by MapReduce such as ease of programming, synchronization and parallelization details abstracted away from the programmer, DELMA also allows for:

- The ability for a cluster to be dynamically elastic, i.e. expand or reduce its size on-demand, without requiring for the job to be stopped and restarted but rather paused while reconfiguration occurs.
- The ability for the application programmer and user to start a given job early before all resources necessary for such job are available, and then progressively add compute nodes as they become available to the user.

- The ability for transient or voluntary nodes to be added to a MapReduce cluster whenever available.
- The ability for faulty or poor performing nodes to be replaced as a job is on-going, without the need to restart the already in-process job, and potentially losing valuable time.
- The ability for slow jobs to be sped up on the fly by adding more resources once a given job is underway, and the user gets an idea of what their completion time would be given their current configuration.

These goals are reached by the framework's ability to decouple input chunk ownership from its nodes, on-demand task progress report, and the framework's ability to pause, checkpoint, save, reconfigure, and resume jobs.

VIII. FUTURE WORK

In future work, we plan to stress DELMA with constant node additions and removals, so as to quantify the possible negative impact of a highly volatile opportunistic environment. While the paper showcased such a behavior at different application completion stages ranging from 10%, 20% and progressively all the way up to 90%, it would be interesting to showcase the same behavior for incessant node arrivals and departures. This will allow us to determine how frequently nodes can disappear and reappear on clusters for application runtime to be negatively impacted. We also plan to analyze the effect of disk I/O on our implementation. We plan to quantify the threshold points for our custom implementation for a wider range of grid application datasets. Apart from size of datasets, we will also identify the threshold points for different network, I/O, memory, and processor configurations that are available in widely used grid infrastructures.

REFERENCES

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] G. Orenstein, "Digging Deeper Into Data With Hadoop," *Available at http://gigaom.com/2009/06/07/digging-deeper-into-data-with-hadoop*, 2009.

[3] A. S. Szalay, P. Z. Kunszt, A. Thakar, J. Gray, D. Slutz, and R. J. Brunner, "Designing and mining multi-terabyte astronomy archives: the sloan digital sky survey," *SIGMOD*, 2000.

[4] S. W. S. Shimin Chen, "Map-reduce meets wider varieties of applications." May 2008.

[5] Microsoft Research. [Online]. Available: http://www.microsoft.com/windowsazure/

[6] AMAZON, "Amazon Elastic Computer Cluster." [Online]. Available: http://aws.amazon.com/ec2

[7] Pervasive Technology Institute. [Online]. Available: http://www.iterativemapreduce.org/

[8] Apache Hadoop. [Online]. Available: http://hadoop.apache.org

[9] D. Borthakur, "The hadoop distributed file system: Architecture and design," *Hadoop Project*, 2007.

[10] S. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A Scalable, High-Performance Distributed File System," in *OSDI '06: Proceedings of the 7th Conference on Operating Systems Design and Implementation*, November 2006.

[11] L.-F. Cabrera, D. D. E., and L. Swift, "Using distributed disk striping to provide high I/O data rates," *Computer Systems*, 1991.

[12] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang, "Serverless network file systems," in *15th ACM Symposium on Operating System Principles*, December 1995.

[13] L. Heshan, A. Ma, and M. Feng, "Moon: Mapreduce on opportunistic environments," in *HPDC '10: the ACM International Symposium on High Performance Distributed Computing*. ACM, 2010.

[14] Z. Fadika, M. R. Head, and M. Govindaraju, "Parallel and Distributed Approach for Processing Large-Scale XML Datasets," in *10th IEEE/ACM International Conference on Grid Computing*, October 2009, pp. 5–7.

[15] L. G. V. T. Ravi and G. Agrawal, "Supporting Load Balancing for Distributed Data-Intensive Applications," in *IEEE International Conference on High Performance Computing (HiPC'09)*, Kochi, India, December 2009.

[16] G. Singh, S. Bharathi, A. Chervenak, E. Deelman, C. Kesselman, M. Manohar, S. Patil, and L. Pearlman, "A Metadata Catalog Service for Data Intensive Applications," in *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2003, p. 33.

[17] AxisJava, "The Apache Project," 2002, http://ws.apache.org/axis/.

[18] D. Veillard, "The XML C parser and toolkit of Gnome," 2006, http://xmlsoft.org/.

[19] Y. Oren, "Piccolo is a small, extremely fast XML parser for Java," 2006, http://piccolo.sourceforge.net/.

[20] M. R. Head, M. Govindaraju, A. Slominski, P. Liu, N. Abu-Ghazaleh, R. van Engelen, K. Chiu, and M. J. Lewis, "A Benchmark Suite for SOAP-based Communication in Grid Web Services," in *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2005, p. 19.

[21] W. Zhang and R. A. van Engelen, "A Table-Driven Streaming XML Parsing Methodology for High-Performance Web Services," in *ICWS '06: Proceedings of the IEEE International Conference on Web Services*. Los Alamitos, CA, USA: IEEE Computer Society, 2006, pp. 197–204.

[22] World Wide Web Consortium, "w3c." [Online]. Available: http://www.w3.org

[23] *Hierarchical Data Format (HDF)*, http://hdf.ncsa.uiuc.edu/.

[24] *Network Common Data Form (netCDF)*, http://www.unidata.ucar.edu/packages/netcdf/.

[25] D. Thain and M. Livny, "Building reliable clients and servers," in *The Grid: Blueprint for a New Computing Infrastructure*, I. Foster and C. Kesselman, Eds. Morgan Kaufmann, 2003.

[26] J. Yu and R. Buyya, "Gridbus workflow enactment engine," in *Grid Computing: Infrastructure, Service, and Applications*, W. J. L. Wang and J. Chen, Eds., April 2009, pp. 119–146.

[27] D. Thain, T. Tannenbaum, and M. Livny, *Grid Computing: Making the Global Infrastructure a Reality*. Wiley, 2003, ch. 11, Condor and the Grid, pp 299-335.