# HadoopJitter: The Ghost in the Cloud and How to Tame It

Vivek Kale*, Jayanta Mukherjee†, Indranil Gupta‡, William Gropp§
Department of Computer Science, University of Illinois at Urbana-Champaign
201 North Goodwin Avenue, Urbana, IL 61801-2302, USA
Email: *vivek@illinois.edu, †mukherj4@illinois.edu, ‡indy@illinois.edu, §wgropp@illinois.edu

***Abstract*—The small performance variation within each node of a cloud computing infrastructure (i.e. cloud) can be a fundamental impediment to scalability of a high-performance application. This performance variation (referred to as jitter) particularly impacts overall performance of scientific workloads running on a cloud. Studies show that the primary source of performance variations comes from disk I/O and the underlying communication network [1]. In this paper, we explore the opportunities to improve performance of high performance applications running on emerging cloud platforms. Our contributions are 1. the quantification and assessment of performance variation of data-intensive scientific workloads on a small set of homogeneous nodes running Hadoop and 2. the development of an improved Hadoop scheduler that can improve performance (and potentially scalability) of these application by leveraging the intrinsic performance variation of the system. In using our enhanced scheduler for data-intensive scientific workloads, we are able to obtain more than a 21% performance gain over the default Hadoop scheduler.**

## I. Introduction

Certain high-performance applications such as weather prediction or algorithmic trading require the analysis and aggregation of large amounts of data geo-spatially distributed across the world, in a very short amount of time (i.e. on-demand). A traditional supercomputer may be neither a practical nor an economical solution because it is not suitable for handling data that is distributed across the world. For such application domains, the ease and inexpensiveness of getting access to a cloud has shown to be an advantage over high performance clusters.

The strength of cloud computing infrastructures has been the reliabilty and fault-tolerance of an application at a very large scale. Google's MapReduce [2] programming model and Yahoo's subsequent implementation of Hadoop [3] have allowed one to harness the power of such cloud infrastructures. However, for certain applications (particularly data-intensive scientific workloads) the small performance variations in a distributed system are a fundamental impediment to application scalability on a large-scale distributed system. It has been observed that the primary sources of performance variation come from disk I/O and interactions with the memory hierarchy, the communication network delay, network congestion, and bandwidth related issues. Since the vision in cloud computing is to use clusters of commodity machines to be able to scale up quickly, we believe the problem of the performance variation on a cloud is particularly relevant if clouds are to be used for such applications.

Indeed, a distributed system's jitter due to performance variation of the hardware can be reduced by using enhanced networking hardware or better devices for data storage such as solid-state drives. However, using high-quality devices is typically very expensive and not widely available as a commodity. The expense of such devices goes against the philosophy within a cloud computing paradigm of using cheap commodity machines in the cloud. In this case, we believe the classic end-to-end argument [4] is valid. We believe the problem seems to lie at a higher layer than hardware, and so our approach to solving this problem must be to directly modify the scheduling algorithm so that it provides performance predictability while still maintaining good load balance.

Thus, to mitigate the overhead caused by performance variation on a Hadoop Cluster, we first categorize the sources of jitter in a cloud. We then modify the default Hadooop Task scheduler by introducing new parameters that incorporate jitter characteristics. Our specific strategy involves scheduling more work to those nodes with less jitter, while still maintaining overall load balance. By doing this, we minimize the chance that a job will complete late because of end-to-end system interference or noise on one node. By scheduling tasks to nodes in this way, we show how one can improve performance while reducing the growth of performance variation with an increasing number of nodes in the cluster.

By using our enhanced scheduling algorithm for three representative scientific workloads, we obtain more than a 21% performance improvement over the default Hadoop scheduler.

The remainder of this paper is organized as follows. In section II, we define the basic terms and explain our concepts of jitter in the cloud. In section IV, we formulate this problem as a Constrained Optimization Problem. In section V, we describe an enhanced JitterAware Task scheduling algorithm. In section VI, we describe our experimental design and deployment. In section VII, we present our results for the default Hadoop scheduler and the enhanced JitterAware Task scheduler. In section IX, we present relevant related work. Finally, in section X, we conclude and discuss our future work.

## II. Concepts of Jitter in a Cloud

Jitter is the general performance variation in execution times over multiple runs of the same job under the same

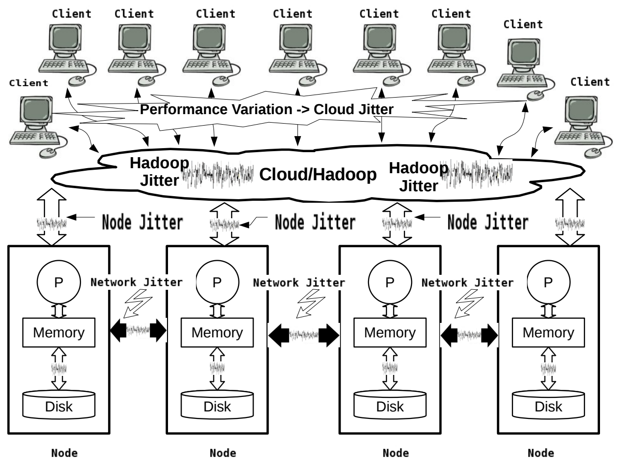


Fig. 1: Breakdown of the sources of Jitter in a Distributed System running Hadoop

conditions. Jitter can be mathematically expressed as the standard-deviation from the expected (average) performance.

$$Jitter = \frac{1}{n}\sum_{i=1}^{n} T_i^2 - \bar{T_a}^2 \quad (1)$$

Here, $T_i$ = Execution time for the i-th run and $\bar{T_a}$ is the mean execution time over multiple runs of the same applications.

With this basic concept of jitter established, we define jitter in the context of a distributed system running Hadoop, and decompose this jitter into smaller jitter sources.

## A. CloudJitter

*CloudJitter* is the performance variation (fluctuation in the execution time) that a user experiences over multiple runs of the same job on the cloud. This $CloudJitter$ is typically beyond direct control of the application programmer, as there are many intricate system services that are intended to maintain end-to-end [4] system performance. For short jobs, this jitter is not noticed, but for longer jobs (particularly data-intensive scientific computations), CloudJitter is particularly relevant. But where does this performance variation come from? To understand this, we decompose this CloudJitter into three separate independent components: NodeJitter, NetworkJitter, and HadoopJitter. The figure below shows how these different jitters within a distributed system contribute to the overall CloudJitter observed by the user of a system.

## B. NodeJitter

*NodeJitter* is the localized jitter (performance variation) caused by systems services (such as disk-I/O, cache-coherence, OS-daemons) within a node. NodeJitter is impacted primarily by disk I/O (**DiskJitter**) and local operating system noise (**OSJitter**). DiskJitter may occur due to variations in seek time or rotation delays. As hard disks may wear out at different rates, each node has its own characteristic DiskJitter. The OS jitter comes primarily from a nodes' local operating system services. Examples of os jitter are process migration, os timers and interrupts, page faults, TLB misses. For data-intensive applications running on clouds, the DiskJitter is dominant (as will be shown in section VII). This deviation in performance is local to a node, but impacts the overall performance.

## C. NetworkJitter

We define *NetworkJitter* as the jitter (performance variation) caused by network services due to congestion, latency and the deployment of the TCP/IP protocols. On a large distributed system (e.g. a network of data-centers, where all data centers are geo-spatially distributed) the node to node latency and thereby, the $NetworkJitter$ can dominate over other jitter values mentioned in equation 1. Indeed, in such large distributed systems, certain protocols can be optimized and tuned for the vendor-specific architectures. Yet, performance predictability is sacrificed for the sake of optimal performance and reliability of packet arrival. If the variations in packet delay are very high, data-intensive scientific workloads degrade in performance very easily as we scale to a large number of nodes.

## D. HadoopJitter

We define *HadoopJitter* as the performance variation that a job suffers when using a particular Hadoop scheduling algorithm. The HadoopJitter of a distributed system can vary depending on the specific scheduler used (e.g. FairScheduler, FIFO Scheduler, CapacityScheduler), as well as any other daemon's associated with the scheduler. HadoopJitter is caused by anything that the software developers of Hadoop could have done (but have not done) to reduce overall CloudJitter. HadoopJitter depends on how Hadoop has been configured on a particular system. The scheduler and its associated daemons aim to maintain efficiency and load balance across nodes of the distributed system. HadoopJitter is impacted by the scheduler's effectiveness in handling multiple jobs. The overhead is caused by suboptimal parameter values (e.g. task granularity), and the inefficiencies of the implementation of the application in MapReduce.

We decompose the CloudJitter into NodeJitter, NetworkJitter, and HadoopJitter to analyze the impact of each component on the overall performance, with the simplified assumption that each of the components influence the performance independently. The rationale for our assumption is that NetworkJitter, HadoopJitter, and each of the NodeJitters are pair-wise independent. The total NodeJitter can be expressed as the linear combination of all the computing nodes available in the cloud. With this, we formulate the *CloudJitter* as follows:

$$\begin{aligned} \textit{CloudJitter} &= \textit{HadoopJitter} + \textit{NetworkJitter} \\ &\quad + \textstyle\sum_{i=1}^{p} C_i \times NodeJitter_i \end{aligned}$$

Here, p is the number of computing nodes on the cloud, $C_i$ is a constant.

Note that this constant is the fraction of load currently on node i. In our actual algorithm we give full weight of 1 to the

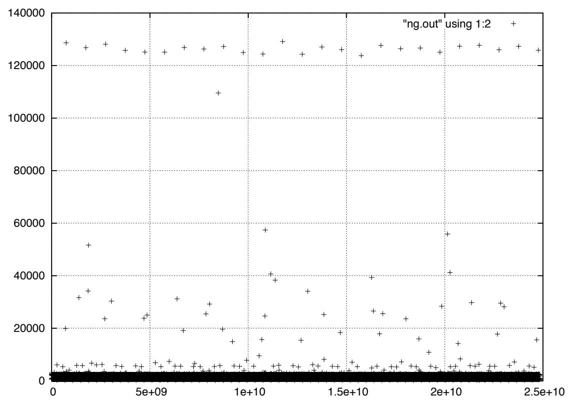


Fig. 2: Noise for 1 node of our cluster.

node with largest NodeJitter and set all other nodes' weights to zero, effectively using the maximum rather than weighted average of NodeJitters.

In the context of this work, our focus is to reduce the HadoopJitter by tuning the default Hadoop Task Scheduler so as to leverage the *measured* NodeJitter and NetworkJitter.

## III. Measuring the Sources of Jitter

To realize the inherent jitter in our system, we show below a sample reading of a *Diagnostic test* to measure and capture the noise of the system. The netgauge reading from one of the nodes (shown in figure 2) shows the measured load over the execution of the noise benchmark running on just 1 node.

As we can see from figure 2, the impact of jitter is seen in the top band of data points in the graph. The band is considerably dense and has a high amplitude. The spikes in load indicate a relatively high frequency of jitter events throughout the run. For the nodes that we used were commodity Intel Xeon processors, modern high-performance clusters are designed to induce much less jitter than we have observed here. The noise may not seem large for most applications running on the cloud(even on the order of 10,000 nodes, but for high-performance applications this noise can be significant impedence to performance due to noise amplification (as has been shown in [1]).

For data-intensive workloads, the disk I/O is critical to the performance of the applications. After a more detailed analysis of our test run, the performance variation due to Disk I/O (DiskJitter) accounted for more than 23% of the total Jitter for these runs, while the network jitter was 10%. The range of DiskJitter was 8.2% across all nodes. From this, we noticed that some disks were inherently very stable while other disks seemed to be unstable (particularly the Master node).

## IV. Problem Statement

Having defined jitter and the types of jitter within a cloud computing environment, we now define the problem as follows: given the intrinsic *NodeJitter* and *NetworkJitter* in the distributed system, our objective is to minimize the *CloudJitter*, while maximizing the throughput (overall performance). We formulate the problem as a Constraint Optimization Problem as shown below:

$$\textit{minimize (CloudJitter) and maximize (Throughput)}$$
$$\textit{given NodeJitter} + \textit{NetworkJitter} \geq 0$$

In other words, the load distribution is optimal if we maximize the probability that a job will be completed(i.e. we minimize CloudJitter), while maximizing throughput and resource utilization of all nodes during the job (i.e. we maximize resource utilization). To obtain optimal performance, the scheduler must minimize the HadoopJitter by putting less load on a node with high jitter, and putting more load on a node with low jitter. However, if the load is offset by "too much", we will get very good performance predictability, but the benefits of Hadoop's basic load balancing strategies will be lost. As a large jitter event is more likely to affect a long-running job, our method will be effective in reducing the chance that jitter will offset the execution of a MapReduce Job consisting of this long-running job (perhaps with multiple map and reduce phases).

While NodeJitter within the node may seem small and insignificant for an application involving just one iteration, it can be detrimental for scientific workloads involving repeated iterative steps. For such workloads, the small NodeJittter from any of the nodes is accumulated over iterations. A jitter event on any of the nodes can result in a serious performance loss for the entire job. We make note that DiskJitter and OSJitter are based on a measured value. At the beginning of a job, a jitter reading is taken to extract these Jitter levels for each node. These readings give us the ability to assess which nodes have high jitter, and which nodes have low jitter. Indeed, for very long running jobs (perhaps taking many hours), jitter values may change over a period of time. In this case, we optimize over the entire duration of job execution, updating the jitter values over a pre-specified period of time.

## V. JitterAware Scheduling Algorithm

Hadoop's default scheduler uses First-In-First-Out (FIFO) approach to determine how to assign a new task in the queue to a node [5]. This scheduler takes into account hardware specification of a node such as virtual memory, but does not consider performance variations caused by disk I/O. A major component of hadoop performance depends on the TaskScheduler. This method assigns tasks to TaskTrackers with the least amountof load.

However, the default scheduler makes an assumption that all nodes on a homogenous cluster have equal capacity. While this may be true for a high-performance cluster, this does not necessarily hold for commodity clusters. Simply put, some nodes are "noisy"(have more jitter) while other nodes are not. For nodes of a cloud, it turns out that this noise is primarily due to performance variations of disk I/O. If a node is unstable because of its inherent local jitter, then it should not be a favorable candidate to run the task of a job that is latency sensitive. In other words, the node which is the least likely to be perturbed during its execution should be the one that is preferred to run more tasks. Our modified scheduler is more conservative because we add padding for the jitter. Taking this into account may not have much impact for short jobs, but for long running jobs with multiple map and reduce phases, we believe this provides a large performance benefit.

To mitigate the impact of performance variation (jitter), we modify the default Hadoop Task scheduler to be jitter aware as shown in Algorithm 1. Our algorithm takes as input the currently available TaskTracker along with its associated NodeJitter values (calculated using NetGauge [6]). The scheduler obtains the number of TaskTrackers `numTaskTrackers` using `clusterStatus.getTaskTrackers()`. We need the total number of TaskTrackers because we are considering NetworkJitter to be a function of the number of TaskTrackers. We set TotalJitter to be the sum of NodeJitter and NetworkJitter.

For a particular job in the JobQueue, our algorithm calculates the remaining map-load (`remainMapLoad`). MapLoadFactor is the specific load factor for the TaskTracker running on that node. The load factor for the mapper (`mapLoadFactor`) is adjusted by 1-TotalJitter. The total map capacity cannot be exploited due to presence of NetworkJitter and the NodeJitter of that TaskTracker (note that TotalJitter is calculated as the sum of NetworkJitter and NodeJitter). The effective map capacity of a cluster is dependent on the underlying communication network properties and the properties of the hardware, devices, and operating system support of a node. With these adjustments to the load factor, our scheduling algorithm proceeds, taking into account these adjustments to the capacity of the TaskTracker and assigning tasks to TaskTrackers with the highest available capacity.

At this point, we adjust the capacity and maximum load of a TaskTracker node based on the measured jitter readings. (Recall that we obtain the jitter values through periodic readings of netgauge). Our algorithm calculates the available capacity of the TaskTracker to be (1 - NodeJitter) multiplied by the product of the `mapLoadFactor` and `trackerMapCapacity`. In this way, we take into consideration the fact that performance is highly dependent on the inherent jitter of that node, and that the map capacity for a particular node is principally dependent on its own NodeJitter value. Thus, the current Map capacity of a TaskTracker (`trackerCurrentMapCapacity`) is calculated by taking the minimum of `ceil(mapLoadFactor×trackerMapCapacity×(1 - NodeJitter))` and `ceil(trackerMapCapacity×(1- NodeJitter))`.

In calculating the current map capacity of the TaskTracker, we have effectively multiplied it by the fraction $\frac{(1-TotalJitter)}{(1-NodeJitter)}$, which is always greater than or equal to 1, as TotalJitter $\geq$ NodeJitter. Thus, we are able enhance the accuracy of the estimate for the current Map Capacity of the TaskTracker (`trackerCurrentMapCapacity`). In theory, given an infinite bandwidth network with zero latency, the map capacity would be only influenced by NodeJitter as 1 - TotalJitter converges to 1 - NodeJitter. Yet, NetworkJitter is significantly high for clouds running on geo-spatially distributed servers, or servers connected by very low-bandwidth (high-latency) Ethernet cables. Therefore, even if the NetworkJitter is much higher than the NodeJitter, this algorithm will still enhance the TaskTracker capacity so as to improve performance.

With this, the number of available map slots on a TaskTracker (`availableMapSlots`) is calculated by taking the difference of the TaskTracker's current map capacity (`trackerCurrMapCapacity`) and `trackerRunMaps`. As commented in the algorithm, note that we do not modify the Reduce parameters as there is one reducer being used per TaskTracker.

In summary, our strategy is to schedule more work to those nodes with less NodeJitter, and we believe our strategy has the potential to be useful for any cloud running Hadoop. Our algorithm aims to minimize HadoopJitter in such a way that the overall impact of the CloudJitter is minimized. The intuition is that if we schedule more work to nodes with less performance variation, there is less chance that a system service unexpectedly delays a task running on a node(due to a jitter event) that in turn delays an entire job. By doing this, we minimize the chance that a job will complete late because of system noise or interference on one node.

**Algorithm 1** Jitter Aware Task Scheduler

**Filename:** `JobQueueTaskScheduler.java`
**Function:** assignTasks
**Input:** taskTracker
**Output:** Synchronized List of Tasks

```
taskTracker name ⟵ Parse(taskTracker.getTrackerName())
NodeJitter ⟵ getNodeJitter (taskTracker)
numTaskTrackers ⟵ clusterStatus.getTaskTrackers()
NetworkJitter ⟵ (numTaskTrackers-1) × NetworkJitter
TotalJitter ⟵ NodeJitter + NetworkJitter

for ((JobInProgress job : jobQueue) do
  remainMapLoad+=(job.desiredMaps() - job.finishedMaps())
end for

if (clusterMapCapacity > 0) then
  mapLoadFactor ← $\frac{remainMapLoad}{clusterMapCapacity \times (1-TotalJitter)}$
end if

if (clusterReduceCapacity > 0) then
  reduceLoadFactor ← $\frac{remainReduceLoad}{clusterReduceCapacity}$
end if

trackerCurrentMapCapacity ← min(ceil(mapLoadFactor×
trackerMapCapacity × (1 - NodeJitter)),
ceil(trackerMapCapacity × (1- NodeJitter)))
availableMapSlots ← trackerCurrMapCapacity - trackerRunMaps

/* Keeping the Reducer calculation same as it is */
```

## VI. Experimental Design and Deployment

Throughout our experimentation with the basic Hadoop FIFO scheduler and our modified Hadoop scheduler, we strive to answer 3 specific research questions:

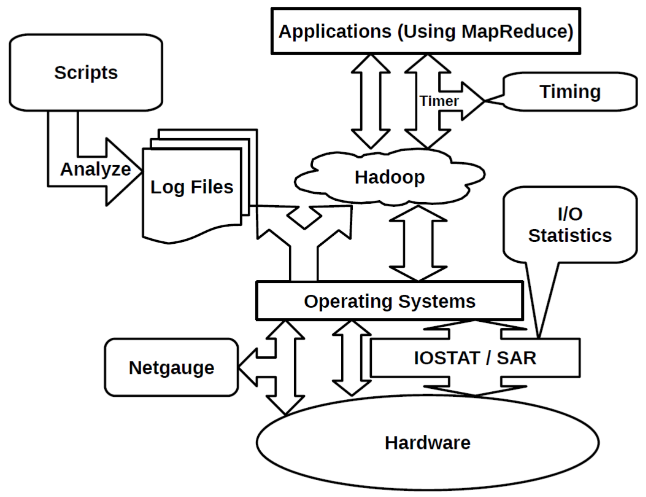


Fig. 3: HadoopJitter Collection Framework

1) Does the NodeJitter or NetworkJitter for a scientific application amplify the performance variation (CloudJitter) as we scale to multiple nodes?
2) What are the specific sources jitter that affect performance variation the most?
3) How do we tune the Hadoop scheduler to mitigate jitter so as to improve performance?

Through our experimentation and analysis in Section VII, we answer each of these questions based on our observations.

## A. System Configuration and Setup

For our experimental setup, we use a homogenous system of 4 dedicated nodes of the CCT at UIUC [7]. Note that in spite of this being a homogeneous cluster, we still witness jitter in our setup. We use Hadoop version 0.20.0 throughout our experiments. We assume one TaskTracker per node. Also, we assume a fully subscribed node; we use a maximum of 16 mappers per TaskTracker (each node has 8 cores with 2-way SMT). In addition, we ensure that no other jobs are running in the background. The layout of our experimental setup is shown in figure 3. In this figure, we illustrate the basic layers of abstraction and show where timers and profilers are plugged in. To obtain the time for the run, we use the UNIX system`time` function, and use the real time as our measurement. The NetworkJitter was approximated using a netgauge ping-pong benchmark provided in the netgauge package [6]. NodeJitter was calculated using a netgauge noise benchmark [6]. For collecting fine-grained metrics of each of the CCT nodes, we use the `iostat` and `sar` command-line tools [8]. The `sar` tool uses sampling strategies that induce minimal overhead, collecting important metrics of memory utilization and CPU utilization. For estimating disk I/O, we use the command-line tool `iostat` during the execution of the job, and we collect disk I/O measurements over a period of 2 second intervals. We profile transactions per second (tps) of the disk during the execution of the job. This tool allows us accurately measure the disk I/O throughout the execution of the job, as well as the total disk I/O of that job, without interfering with the main application. To understand performance variations across separate runs of a job as we scale, we observe performance trends on our 4 dedicated nodes, using the basic scheduler, and an enhanced JitterAware scheduler. This requires specifying the scheduler java class in the mapred-site.xml file, as has been demonstrated in [9].

## B. Applications Considered

In our experiments, we consider 3 different scientific applications: PiEstimator, Mean-shift clustering, and Dirichlet clustering. Below, we describe each of these applications in terms their application characteristics and their relevance to this work.

1) **PiEstimator** is representative of Monte Carlo simulations often used for financial trading algorithms. The PiEstimator application randomly generates a large number of data points. It approximates the value of Pi by obtaining the ratio of samples within a circle to that outside the circle. PiEstimator is important to our study because it requires almost no coordination among tasktrackers (it is embarassingly parallel); this allows us to measure performance variation caused by NetworkJitter. Also, it requires very little disk I/O; this allows us to identify sources of jitter other than the DiskJitter. Throughout our experiments, our PiEstimator workload is set to generate 1 billion samples.
2) **Mean-shift** is an example of a data-intensive scientific workload from Mahout [9]. Means-shift consists of a relatively small number of iterations (on the order of 100), each iteration having a map and reduce phase. The small number of iterations allows the clustering algorithm to execute in a relatively short amount of time(about 2 minutes), but results in a relatively unrefined clustering of a large data set. For this application, we use a problem size that is representative of real-world settings: the Mean-shift clustering algorithm operates on a large Synaptic [9] data set.
3) **Dirichlet** is another example of a data-intensive scientific workload also from Mahout [9]. Unlike Mean-shift, Dirichlet involves a relatively large number of iterations(on the order of 100,000), each iteration having a map and reduce phase. The large number of iterations and the coordination among TaskTrackers at each Reduce phase causes this algorithm to take much more time (about 15 minutes) than the Mean-shift clustering algorithm. However, the result is a highly refined clustering of this large data set. Again, we use a problem size that is representative of real-world settings: the Dirichlet clustering algorithm operates on a large Synaptic [9] data set.

|  | Time |  | Jitter |  | % Jitter |  |
|---|---|---|---|---|---|---|
| Application | 1 Node | 4 Nodes | 1 Node | 4 Nodes | 1 Node | 4 Nodes |
| PiEstimator | 278.161 | 177.739 | 0.28 | 3.612 | 0.101 | 2.036 |
| MeanShift | 178.587 | 151.909 | 1.124 | 9.884 | 0.629 | 6.507 |
| Dirichlet | 698.351 | 587.176 | 10.377 | 44.263 | 1.486 | 7.538 |

TABLE I: Performance and Jitter Characteristics using the Default Scheduler

|  | Time |  | Jitter |  | % Jitter |  |
|---|---|---|---|---|---|---|
| Application | 1 Node | 4 Nodes | 1 Node | 4 Nodes | 1 Node | 4 Nodes |
| PiEstimator | 276.935 | 178.645 | 0.857 | 1.505 | 0.3095 | 0.8424 |
| MeanShift | 179.056 | 137.642 | 1.228 | 2.156 | 0.686 | 1.566 |
| Dirichlet | 692.2357 | 482.4538 | 9.222 | 38.967 | 1.3322 | 8.077 |

TABLE II: Performance and Jitter characteristics using a Jitter Aware Task Scheduler on Hadoop

## VII. Results

In this section, we answer each of our questions laid out in section VI by describing our experimentation and the results, and providing an explanation and analysis for the performance characteristics of each test.

### A. *Basic FIFO scheduler*

We first evaluate the performance using the baseline FIFO scheduler, and illustrate some of the issues associated with it. Table I shows the average execution time and the performance variation for the three different applications we tested with 1 node and 4 nodes.

For the short PiEstimator job, we noticed that when we used 4 nodes, the TotalJitter was about the same as for that of 1 node. For the medium-size MeanShift clustering algorithm the TotalJitter increases by a factor of 2. For the long Dirichlet job , the TotalJitter increases by a factor of 6. In general, the longer the job, the greater the increase in jitter when we scaled from 1 to 4 nodes.

Not only is the performance predictability worse with an increasing number of nodes for a long job, but we see that the the pure performance also degrades for the long job. When moving from 1 to 4 nodes, there is little performance degradation of the PiEstimator workload. But for the medium size Mean-shift clustering job, the performance degradation is about 4%. For the large Dirichlet clustering job, the performance overhead is more than 30%. Ideally, the performance degradation should be the same for jobs of equal problem size when scaling from 1 to 4 nodes, whether short or long. Why is increase in jitter so much when scaling from 1 to 4 nodes for long jobs? More importantly, why does the performance degrade so much for the longer running jobs?

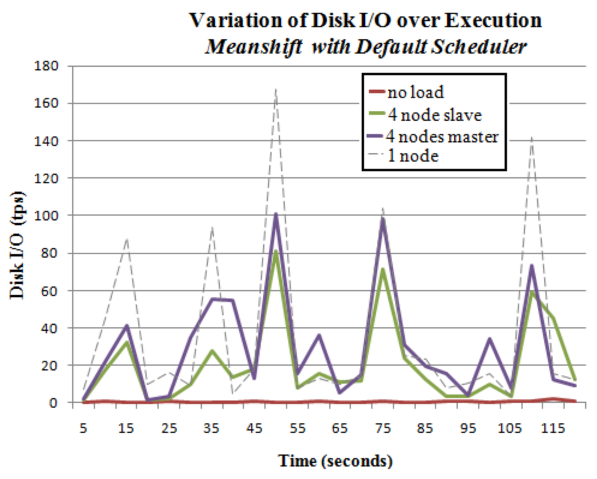
We noticed DiskJitter was different across each node, for a multiple node run. Figure 4a shows the DiskJitter for a medium size (means-shift) clustering job. For each type of job, we show the comparison between a 1 node run and a 4 node run (with the master's and slave's disk I/O).

While running the jobs on a single-node, the disk I/O during the job execution has a few spikes during the runs, but the rest of the time the disk I/O is steady. The variations seem to be due to variations in seek time and disk performance. At 4 nodes, we notice a similar amount fluctuation for both the master node and the slave node as we did for 1 node, for each of the three applications. But what would account for the overall jitter? Looking more closely and comparing the times that the spikes occur, we notice that they actually do not occur at the same time. We noticed that when no job is running the disk I/O stays steady between 0 and 2 tps and we see a relatively low fluctuation here. This skew in the spikes is the primary cause of the amplification of jitter when scaling to multiple nodes. For a short job, these "skewed spikes" do not impact the overall performance. However, for a longer job involving iterations of map and reduce phases, each iteration suffers these skewed spikes from the previous iteration, as well as the skewed spikes in its own iteration. This is why the performance variation is much higher with multiple nodes as compared to one single node. For the long running job, there is a larger "window of opportunity" for a large DiskJitter event to seriously impact the performance on all nodes. The long running job may contain a few large DiskJitter events which can be enough to impact overall performance. Furthermore, even if this large DiskJitter event occurred on only one node, all nodes must suffer this DiskJitter at the reduce phase. Thus, the difference in performance between PiEstimator and the longer clustering algorithms demonstrates the impact of small amounts of local DiskJitter. This DiskJitter can be accumulated over many iterations, and amplified across many nodes. For the clustering algorithms, all 4 nodes must wait at each iteration for the slowest node to complete its work before all nodes can continue to the next iteration. This jitter builds up at each subsequent phase, and the overall jitter is amplified over the entire execution of the job.

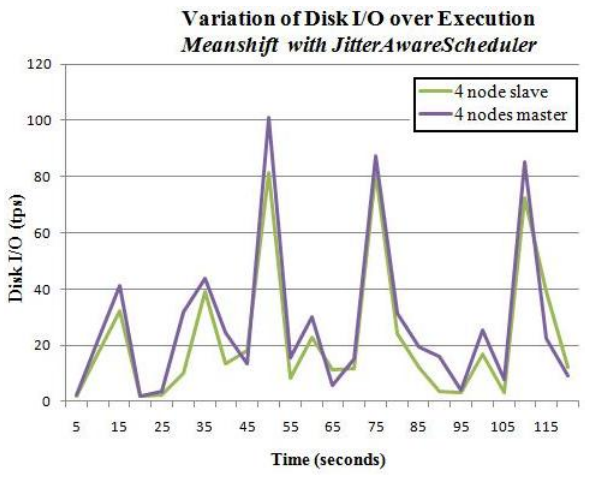
### B. *The Improved JitterAware Scheduler*

To mitigate the jitter observed in the above experiments, we use the algorithm presented in section V. The adjustment for jitter is done per node (task tracker). The results with our improved JitterAwareScheduler are shown in Table II.

Table II shows the performance results of our applications with 1 node and 4 nodes. With our jitter aware scheduler, a single node run took almost the same time as with the base scheduler for all applications. In using our jitter aware scheduler for 4 nodes, PiEstimator seemed to perform the same as the standard scheduler, and we saw very little differences here in performance predictability. However, we noticed a general improvement in performance and performance predictability of our longer clustering algorithms. For four nodes we attained 10% performance improvement over a standard FIFO scheduler when we ran the MeanShift Clustering algorithm. Also, the total jitter at 4 nodes reduced by a factor of 2: from 17.1% to 7.2%. In our tests with the FIFO scheduler, the large performance variation was impacted greatly by the DiskJitter, but our JitterAware scheduler was able to absorb this DiskJitter so as to ensure that all nodes in the system would not be affected by one node with high jitter. We were able to do

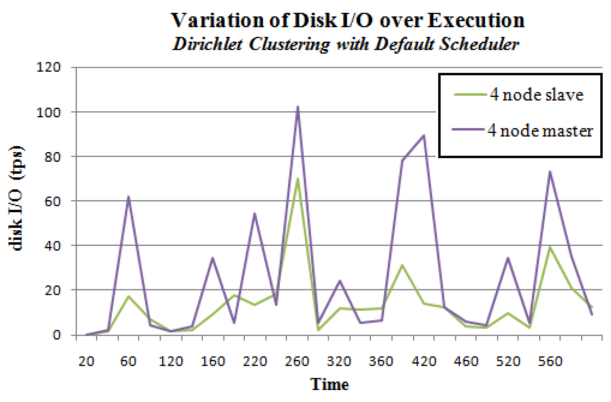


(a) Using Basic FIFO Scheduler

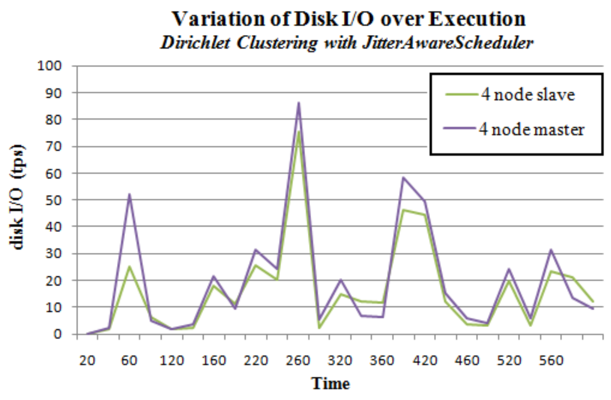


(b) Using Jitter-Aware Scheduler

Fig. 4: Comparison of the Jitter for MeanShift using Default FIFO and Jitter-Aware Scheduler: (a) The Disk I/O for 1 node and 4 nodes (master and slave) as a medium size mean shift clustering job progresses; (b) The Disk I/O for 1 node and 4 nodes (master and slave) is shown for MeanShift clustering;



(a) Using Basic FIFO Scheduler



(b) Using Jitter-Aware Scheduler

Fig. 5: Comparison of the Jitter for Dirichlet Clustering using Default FIFO and Jitter-Aware Scheduler: (a) The Disk I/O for 1 node and 4 nodes (master and slave) as a very long Dirichlet clustering job progresses.; (b) The Disk I/O for 1 node and 4 nodes (master and slave) is shown for Dirichlet clustering for the JitterAware scheduler.

this by incorporating the inherent local jitter within each node and adjusting load to ensure that more "stable" nodes get to do slightly more work than unstable nodes, without offsetting overall load balance that Hadoop provides. The next section delves deeper into why we were able to reduce the jitter for the longer jobs, and how we improved performance for these jobs.

To understand the reduction in performance variation, we show in figure 4b the disk I/O over one of our runs for the medium size job. For a short running job, we saw that the disk I/O fluctuates on the Master periodically (causing DiskJitter to be higher), but because we put slightly less load on the noisy node, these fluctuations had less impact on actual disk I/O required for the job. Our performance was slightly worse here because of the load imbalance caused by our algorithm. For a longer running job, we saw that the disk jitter spikes still exist for the master node and the slave node. Even so, since our enhanced scheduler uses more mappers and allocates more capacity for those nodes with less noise, we saw that the overall jitter over multiple trials decreases for our data-intensive workloads.

As we saw with the FIFO scheduler, across many iterations of the algorithm the local DiskJitter on one node was amplified across all nodes. Yet, since our algorithm is aware of the jitter on each node (particularly the DiskJitter), the scheduler assigned slightly fewer map tasks to nodes with less local DiskJitter and NodeJitter. This helped reduce the amplification of Jitter that we noticed in the basic FIFO scheduler. Because the amplification of jitter is lower, we got better performance overall for a long-running job.

## C. *Performance Comparison of Schedulers*

In this section, we compare our JitterAware Scheduler to the basic FIFO scheduler, shown in Figure 6 and Figure 7.

In Figure 6 and Figure 7, every odd entry represents the performance for **MeanShift** (MeanShift 1N : for 1 node and

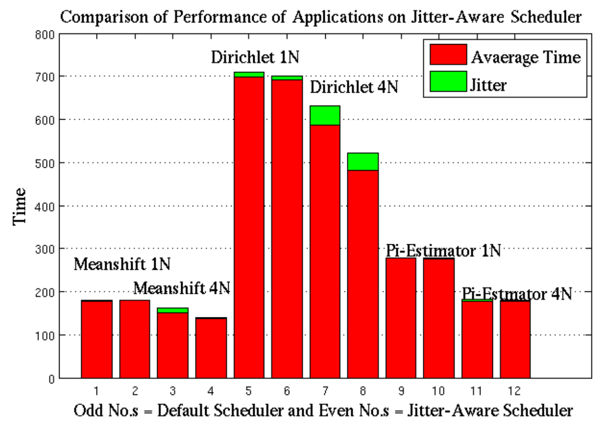


Fig. 6: Performance Comparison of Basic Scheduler to Jitter-AwareScheduler

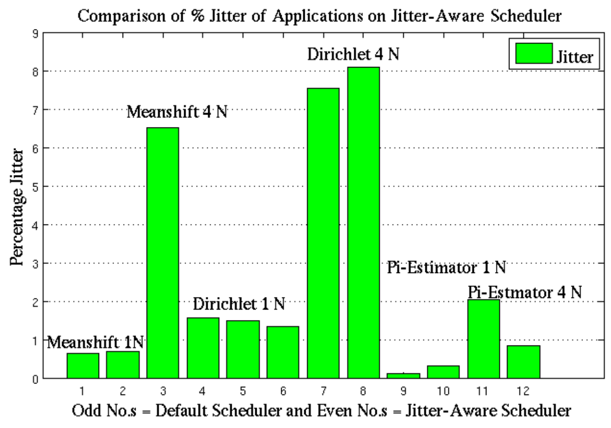


Fig. 7: Percent Jitter Comparison of Basic Scheduler to JitterAwareScheduler

MeanShift 4N for running on 4 nodes) and **Dirichlet** data clustering algorithm (Dirichlet 1N : for 1 node and Dirichlet 4N for running on 4 nodes) provided in Mahout [9]. Also, we have shown **Pi-Estimator** (Pi-Estimator 1N : for 1 node and Pi-Estimator 4N for running on 4 nodes) which is a computation-intensive workload.

The figures show that the JitterAware scheduler reduced the amplification of jitter significantly when moving from 1 to 4 nodes for all cases. For the PiEstimator, the performance improved by 9%, with an jitter reduction of 12%. For the clustering algorithms the amplification reduction was more substantial. For the MeanShift clustering algorithm, the overall performance improved by roughly 20% algorithm with a jitter reduction of 30%. For the Dirichlet algorithm, we saw an 18% performance improvement for the Dirichlet clustering algorithm, with a jitter reduction of 28%.

## VIII. Discussion

Our strategy is to schedule more work to those nodes with less NodeJitter. By doing this, we minimize the chance that a job will complete late because of system noise or interference on one node. The intuition is that if we schedule more work to nodes with less performance variation, there is less chance that a system service holds up a task running on a node that in turn holds up an entire job. In this way, we minimize the idle time for all nodes, while maximizing the probability that the job will complete within a given time limit. Our approach is to add this jitter level as a weight within the load balancing step in the basic FIFO scheduler.

The Hadoop Scheduler can be improved through very sophisticated scheduling algorithms, but our approach is to modify the scheduler algorithm by introducing a new parameter that incorporates jitter characteristics. We believe that our approach will improve performance while still maintaining fault-tolerance. From an application programmer's point of view, the fluctuation in performance due to jitter is a result of the underlying hardware, network infrastructure, the system software. Certain applications such as streaming media this fluctuation can be detrimental to entertainment value of these applications. Thus, the application programmer can only improve the performance to enhance the user experience by utilizing the available resources effectively. An application programmer utilize such resources effectively by taming the observed jitter(i.e. the ghost) into the load balancing and resource allocation strategy.

Indeed, Hadoop's primary goal is to be fault-tolerant and reliable at a large-scale. One of the key features of Hadoop is speculative execution [10]. However, our improved scheduler is intended to work with the speculative execution feature in Hadoop that is so crucial to Hadoop's ability to maintain fault-tolerance.

## IX. Related Work

In the 1990s, much work had been done to use grid computers for running scientific simulations [11]. More recently though, the design elements of reliability within cloud computing in industry seem to have addressed many of the issues in using scientific applications running on grids, and so many have turned to clouds for running such scientific workloads. Hill et al. [5] examine a case study of one specific scientific application (ocean simulation) running on the cloud. An important aspect of this work is that they discuss the use of MPI on EC2, and evaluate the NAS Parallel Benchmarks [12] on a cloud. More recently, barrier-less techniques [13], proposed by Verma. et al., have had the potential to improve the performance of scientific applications running on the cloud. Scientific Computing applications for large scale problems (like Oil-reservoir flow simulation [14]) have been solved on Grid-enabled infrastructure by M Parashar et. al. It shows the imporatance of the distributed system for solving large-computational problems effectively. Franck Cappello et al [15] synthesized the motivations, observations and research issues considered as determinant of several complimentary experts of HPC in applications, programming models, distributed systems and system management.

The effects and comprehensive categorization of OS jitter is done on a Power6 32-processor architecture, and has been examined through a command-line tool called osjitter. They discuss methods of observing jitter as related to a specific application. However, it does not account for TLB misses, cache misses, or other hardware jitter (at least as of now) [16].

The notion that a network of heterogenous nodes could be used to provide an inexpensive alternative to large-scale high-performance clusters for scientific computation has been demonstrated through the Berkeley NOW project in the mid 1990s [17]. They demonstrated the use of a network of commodity workstations that are geospatially distributed. The concept of building inexpensive, yet powerful parallel computing systems was appealing primarily because of availability of these commodity clusters.

Many different improvements to the Hadoop scheduler have been considered. Among these is the LATE scheduling algorithm [10], and it addresses many specific design decisions for speculative execution and fault-tolerance in Hadoop. Their position is that tasks that seem to be taking the longest time to end should be scheduled as early as possible. By scheduling them later, they are at a risk of being run speculatively, which can waste many resources. In our work, we adjust the scheduler to take into account performance variations even on a homogeneous cluster.

Network Jitter impacts the performance more severely on a Cloud-Computing environment where the physical servers are geographically far apart. A systematic study of the in an IP Network has been done by Rizo-Dominguez et al [18] using a Cacuchy approach. In that paper, they showed that an Alpha-Stable jitter model is adequate, and that in some cases the Cauchy distribution provides a satisfactory approximation. Also, they demonstrated how the jitter dispersion increases with the number of hops in the path, following a power law with scaling exponent dependent on the index of stability $\alpha$. Boppana et. al. [19] studied impact of noise on a mobile ad hoc network. They presented two analytical models to describe the noise levels in a real network: a generalized extreme value (GEV) random process model and a Markov chain model. The system noise impacts the performance of a single node.

As we are moving towards the Multicore era, the noise on a Multicore which impedes the scalability has been studied in [20] by Seelam et. al. The work by Sahni et. al [21], describes a a novel framework for supporting e-Science applications that require streaming of information between sites. They demonstrated an algorithmic solution to determine how much bandwidth is allocated to each edge while satisfying temporal constraints on collaborative tasks.

Sriram et. al. proposed an approach [22] to combine the benefits of probing and backtracking based algorithms (better adaptiveness and wider search) with the advantages of distance-vector type algorithms (lower setup time). Their scheme is flexible in that a variety of heuristics can be employed to order the neighbouring links of any given node. Sajal K Das, et. al. proposed [23] a latency-tolerant partitioning scheme that dynamically balances processor workloads on the IPG, minimizing data movement and runtime communication. By simulating an unsteady adaptive mesh application on a wide area network, they studied the performance of their load balancer under the Globus environment, which is essentially a Grid Computing Infrastructure.

## X. Conclusion and Future Work

All in all, our study has illuminated a key tradeoff between load balance and performance predictability. We proposed a methodology for assessing CloudJitter, the general variation in performance of an application one can expect due to the Hadoop scheduler, and showed how performance of data-intensive scientific applications can be improved in the cloud. Our results have shown how the Hadoop scheduler can be improved to account for the jitter within each node.

We quantified CloudJitter for data-intensive workloads (Data-clustering using Mahout) as well as for a computation-intensive workload (a Monte Carlo type method for estimating Pi). Through the use of fine-grained profiling, we were able to realize that the primary source of this jitter is from Disk I/O.

While a set of homogeneous nodes should ideally have similar jitter characteristics, our experimentation showed that even homogeneous nodes suffer from different levels of inherent noise. We tuned the Task Scheduler of Hadoop to take into account this different jitter on each node, and improved the performance by utilizing the mappers of a particular TaskTracker effectively. With this, we performed the same experiments and observed more than a 21% performance improvement for our data-intensive scientific workloads.

For future work, we will tune the scheduler to improve performance in a heterogeneous environment. We also will test our clustering algorithms with a much larger Wikipedia data set. Finally, we hope to deploy and run our jitter evaluation and experimentation with our jitter aware scheduler on a larger distributed cluster (PlanetLab [24] or Emulab [25]). We believe that by testing on a larger number of nodes, we will be able to better assess the jitter impact on scalability.

## Acknowledgments

We would like to thank Milind Bhandarkar from Yahoo! for his suggestions and feedback on features of Hadoop that were of interest to our project goals. We would also like to thank Franck Cappello for his insights about Jitter and the current methods to measure it. Finally we would like to thank TSG for their support and maintanence of the cluster we tested on.

## References

[1] A. Nataraj, A. Morris, A. D. Malony, M. Sottile, and P. Beckman, "The ghost in the machine: observing the effects of kernel operation on parallel application performance," in *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 2007, pp. 1–12.

[2] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[3] D. Borthakur, *The Hadoop Distributed File System: Architecture and Design*, The Apache Software Foundation, 2007.

[4] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end arguments in system design," *ACM Trans. Comput. Syst.*, vol. 2, no. 4, 1984.

[5] C. Evangelinos and C. N. Hill, “Cloud computing for parallel scientific hpc applications: Feasibility of running coupled atmosphere-ocean climate models on amazon’s ec2,” in *Cloud Computing for parallel Scientific HPC Applications: Feasibility of Running Coupled Atmosphere-Ocean Climate Models on Amazon’s EC2*. Cloud Computing and Its Applications, October 2008.

[6] T. Hoefler, T. Mehlan, A. Lumsdaine, and W. Rehm, “Netgauge: A network performance measurement framework,” p. 659.671, 2007.

[7] “Illinois cloud computing testbed.” [Online]. Available: http://cloud.cs.illinois.edu

[8] Sysstat, “Sysstat.” [Online]. Available: http://pagesperso-orange.fr/sebastien.godard/

[9] S. Owen, R. Anil, T. Dunning, and E. Friedman, *Mahout in Action*, The Apache Software Foundation, 2009.

[10] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, “Improving mapreduce performance in heterogeneous environments,” EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2008-99, Aug 2008. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-99.html

[11] J. H. Abawajy and S. P. Dandamudi, “Fault-tolerant grid resource management infrastructure,” *Neural, Parallel Scientific Computing*, vol. 12, no. 3, pp. 289–306, 2004.

[12] M. Frumkin and R. F. V. der Wijngaart, “NAS grid benchmarks version 1.0,” pp. 315–, July 2002.

[13] A. Verma, N. Zea, B. Cho, I. Gupta, and R. H. Campbell, “Breaking the mapreduce stage barrier,” *SIGIR 09*, 2009. [Online]. Available: http://www.ideals.illinois.edu/handle/2142/14819

[14] M. Parashar, R. Muralidhar, W. Lee, D. Arnold, J. Dongarra, and M. Wheeler, “Enabling interactive and collaborative oil reservoir simulations on the grid: Research articles,” *Concurrency and Computation: Practice and Experience*, vol. 17, no. 11, pp. 1387–1414, 2005.

[15] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir, “Toward exascale resilience,” *International Journal of High Performance Computing Applications*, vol. 23, no. 4, pp. 374–388, 2009.

[16] D. Tsafrir, Y. Etsion, D. G. Feitelson, and S. Kirkpatrick, “System noise, os clock ticks, and fine-grained parallel applications,” in *ICS ’05: Proceedings of the 19th annual international conference on Supercomputing*. New York, NY, USA: ACM, 2005, pp. 303–312.

[17] D. E. Culler, “Parallel computing on the berkeley now,” in *In Proceedings of the 9th Joint Symposium on Parallel Processing*, 1997.

[18] L. Rizo-Dominguez, D. Torres-Roman, D. Munoz-Rodriguez, and C. Vargas-Rosales, “Jitter in ip networks: a cauchy approach,” *Comm. Letters.*, vol. 14, no. 2, pp. 190–192, 2010.

[19] X. Su and R. V. Boppana, “On the impact of noise on mobile ad hoc networks,” in *IWCMC ’07: Proceedings of the 2007 international conference on Wireless communications and mobile computing*. New York, NY, USA: ACM, 2007, pp. 208–213.

[20] S. R. Seelam, L. Fong, A. Tantawi, J. Lewars, J. Divirgilio, and K. Gildea, “Extreme scale computing: Modeling the impact of system noise in multicore clustered systems,” in *IPDPS*, 2010.

[21] E. Jung, S. Ranka, and S. Sahni, “Bandwidth allocation for iterative dependent e-science applications,” in *Cloud and Grid Computing (CCGrid), IEEE/ACM International Symposium on Cluster*, 2010.

[22] R. Sriram, G. Manimaran, and C. S. R. Murthy, “Preferred link based delay-constrained least-cost routing in wide area networks,” *Computer Communications*, vol. 21, no. 18, pp. 1655 – 1669, 1998.

[23] S. K. Das, D. J. Harvey, and R. Biswas, “Latency hiding in dynamic partitioning and load balancing of grid computing applications,” *Cluster Computing and the Grid, IEEE International Symposium on*, vol. 0, p. 347, 2001.

[24] J. S. Turner, P. Crowley, J. DeHart, A. Freestone, B. Heller, F. Kuhns, S. Kumar, J. Lockwood, J. Lu, M. Wilson, C. Wiseman, and D. Zar, “Supercharging planetlab: a high performance, multi-application, overlay network platform,” *SIGCOMM Comput. Commun. Rev.*, 2007.

[25] K. Webb, M. Hibler, R. Ricci, A. Clements, and J. Lepreau, “Implementing the emulab-planetlab portal: Experience and lessons learned,” in *In Workshop on Real, Large Distributed Systems WORLDS*, 2004.