GERBIL: MPI+YARN

Luna Xu
Virginia Tech
xuluna@cs.vt.edu

Min Li
IBM T.J. Watson Research
minli@us.ibm.com

Ali R. Butt
Virginia Tech
butta@cs.vt.edu

***Abstract*—Emerging big data applications comprise rich multi-faceted workflows with both compute-intensive and data-intensive tasks, and intricate communication patterns. While MapReduce is an effective model for data-intensive tasks, the MPI programming model may be better suited for extracting high-performance for compute-intensive tasks. Researchers have recognized this need to employ specialized models for different phases of a workflow, e.g., performing computations using MPI followed by visualizations using MapReduce. However, extant multi-cluster approaches are inefficient as they entail data movement across clusters and porting across data formats. Consequently, there is a crucial need for disparate programming models to co-exist on the same set of resources.**

In this paper, we address the above issue by designing GERBIL, a framework for *transparently* co-hosting unmodified MPI applications alongside MapReduce applications on the same cluster. GERBIL exploits YARN as the model agnostic resource negotiator, and provides an easy-to-use interface to the users. GERBIL bridges the fundamental mismatch between YARN and MPI by designing an MPI-aware resource allocation mechanism. We also support five different optimizations: minimizing job wait time, achieving inter-process locality, achieving desired cluster utilization, minimizing network traffic, and minimizing job execution time, all in a multi-tenant environment. Our evaluation shows that GERBIL enables MPI executions with performance comparable to a native MPI setup, and improve compute-intensive applications performance by up to 133% when compared to the corresponding MapReduce-based versions.

I. INTRODUCTION

MapReduce [7] has emerged as the *de facto* distributed processing model for big data analytics due to its ease-of-use and ability to scale. Although a wide-range of applications, such as web crawling and text processing [23] and log querying [4], benefit from the MapReduce model, there exists numerous rich data analytics workflows that can not be fully or efficiently captured using MapReduce. Consider Metagenomics [40], a life science workflow that consists of clustering and multi-dimensional scaling operations that involve parallel linear algebra computations [45]. Such compute- and communication-intensive jobs entail intricate communication patterns that cannot be captured by only the shuffle supported in MapReduce. For such applications, the well established Message Passing Interface (MPI) [13] is more suitable due to its ability to support any communication pattern. Moreover, MPI has proved its mettle in large-scale high-performance scientific computing (HPC). However, simply porting the I/O-intensive data analytics workflow to MPI is impractical, given the demonstrated ability of MapReduce to deliver high-throughput for data-intensive workloads. Moreover, a MapReduce application has the added advantage of being able to run on public cloud services, e.g., Amazon AWS [1], which removes the need for dedicated in-house clusters and lower the barrier-to-entry for users. Consequently, a new two-cluster paradigm is emerging, wherein the compute-intensive tasks of a workflow are run using MPI on traditional HPC clusters, and then the data is moved to a MapReduce cluster for data-intensive processing [42]. This is undesirable due to unnecessary dual maintenance of two kinds of clusters. Moreover, such settings entail performance degrading manual copying of data from one model to another, e.g., from the cluster file systems of HPC to HDFS [42] for Hadoop/MapReduce.

An alternative approach is to port MPI applications to MapReduce, either via auxiliary libraries [3], [6], [29], [38], [52] or by re-designing parallel algorithms using MapReduce APIs [39], [44], [50], [51]. Though promising, this approach is not always possible without completely redesigning well-established applications [50], which undermines the effort done in validating and developing the original applications. A promising development here is the evolution of the standard monolithic Hadoop design [2] into YARN [48] via decoupling of resource management from task management. YARN lays the foundation for supporting diverse programming paradigms and not just MapReduce on the managed resources. However, while YARN is programming model agnostic in concept, it is based on refactored code from Hadoop 1.0 [48], and inherits many of the design decisions and implementation aimed at supporting only the MapReduce model.

In this paper, we address the above issues by designing GERBIL, a YARN-based framework for *transparently* co-hosting unmodified MPI applications alongside MapReduce applications on the same set of resources. Enhancing YARN to host MPI applications allows realization of rich data analytics workflows as well as efficient data sharing between the MPI and MapReduce models within a single cluster. GERBIL enables fine-grained application-to-model matching and has the promise to significantly improve performance and ease-of-use. Users can leverage say an MPI matrix multiplier with a MapReduce log analysis, both within a single workflow running on an integrated cluster without manual hacking. GERBIL hides the low-level details such as resource negotiations and management of dual models from the users, while supporting user-specified task allocation strategies, e.g., optimizing for minimum job wait time, inter-process locality, or desired cluster utilization, in a multi-tenant environment. Moreover, there is a great interest from the open source Hadoop community to embrace MPI as a first class citizen on YARN [43]. GERBIL aims to deliver on this promise.

We faced several challenges when designing GERBIL. YARN provides for customized application management component (application master, AM) that can be exploited to support multiple programming models. First, we need to design an AM that manages the life cycle of allocated resources and the assignment of MPI processes to the allocated resources in an efficient way, requiring in-depth understanding of YARN and its interactions. Second, we found that there is a significant difference in the design and resource management principles of MapReduce and MPI, precluding a straightforward extension of YARN to support MPI. For example, YARN allocates resources based on heart beats from node managers, which results in long resource allocation time and can lead to significantly long MPI application launching compared to a native MPI cluster. Third, we have to address cases where available resources are not enough to assign all tasks of an MPI application. A naive solution of waiting for more resources to become available, and not using the currently available resources, can lower cluster utilization and increase application turn-around time. We propose resource over-subscription so as to allow an MPI application to start albeit in a degraded mode due to sharing/over-subscription of the underlying resources.

Specifically, this paper makes the following contributions:

- We identify a fundamental mismatch between YARN resource negotiation mechanism and the MPI programming model (Section II). We found that the MapReduce-tailored, container-based resource negotiation protocol in YARN does not work well for MPI applications that in essence require gang resource allocation.
- We present the design and implementation of GERBIL (Section III and Section IV), which supports traditional MPI job submission semantics as well as allocation of cluster resources to MPI jobs via negotiating with YARN and transparently launching the applications.
- We present five resource provisioning strategies for GERBIL for multi-tenant environments (Section III-C). Such flexible resource provisioning helps in executing the MPI applications based on various cluster utilization goals such as minimizing application launching time, reducing network traffic and increasing cluster resource utilization.
- We conduct an in-depth performance study of GERBIL on an 19-node cluster [21] using representative applications (Section V). Our experiments show that GERBIL can run MPI applications with performance comparable to the native MPI setup. The runtime overhead of GERBIL, mostly incurred by the resource negotiation with YARN, is observed to be constant and would become negligible when amortized across long-running applications.

II. ENABLING TECHNOLOGIES

In this section, we describe MPI and YARN, which serve as the enabling technologies for GERBIL.

A. The MPI Framework

MPI (Message-Passing Interface) [13] uses explicit messages to coordinate between distributed processes. In our work, we use the publicly available MPICH [16]. MPICH provides several internal process managers [27] such as Gforker, Remshell, SMPD, Hydra [11], and MultiPurpose Daemon (MPD) [30] that we use in our prototype. A typical MPI job is submitted through the `mpiexec` command, which also accepts arguments such as a machine file or number of processes to use. The process manager then spawns connected daemon processes on participating nodes, which prepare the communication environment, handle process binding, and spawn the application MPI processes. The process manager also handles tasks such as signal forwarding and I/O forwarding, and cleanup upon task completion. The MPI model does not provide resource management mechanisms, and relies on systems such as Portable Batch System (PBS), SLURM [37], Torque [47], Moab [14] and Cobalt [32].

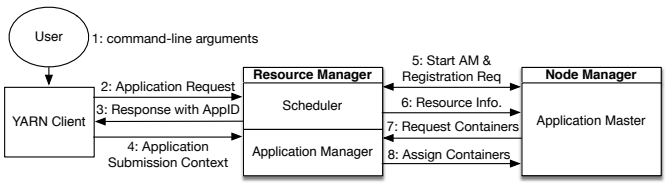

Fig. 1. Steps of launching an application in YARN.

B. YARN

YARN [48], the second generation Hadoop release, has two key components: a global *ResourceManager* (RM) that provides resource management across all available resources and allocates resources for applications, and a per-application *ApplicationMaster* (AM) that manages the allocated resources and handles job scheduling and monitoring. The RM leaves the control over the applications to their associated AMs. YARN uses a *container* abstraction to allocate and manage resources. The containers provide some degree of performance isolation and guarantees for application execution. Figure 1 shows the steps involved in running an application on YARN. A user submits jobs through a client to the RM by providing: (a) application submission context containing ApplicationID, user, queue, and other information needed to start the AM; and (b) container launching context (CLC) that specifies resource requirements (Memory/CPU), job files, security tokens, and other information needed by the AM to run. The RM then allocates the requested container, starts the AM in the container, and passes control to the AM that can then request more containers from the RM as needed.

Although YARN is designed as a flexible and scalable resource manager, it is not well-suited for MPI for two reasons. First, YARN does not provide support for gang scheduling needed by MPI jobs, which can lead to long waiting time before an MPI job can start. Second, fine-grained resource container allocation in YARN can lead to multiple containers allocated to a single physical node. This creates challenges for the MPI process manager design, as traditional MPI setups support only one manager process per node.

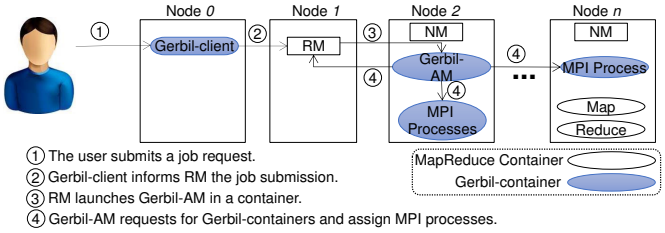

Fig. 2. GERBIL architecture for running MPI on YARN.

III. SYSTEM DESIGN

In this section, we first discuss the objectives of GERBIL design. Then, we give an overview of our architecture and describe the major components of GERBIL, followed by a discussion on how I/O and fault tolerance can be supported.

A. Design Objectives

The key design objectives of GERBIL are as follows.

Transparent System Interface: We want to ensure that GERBIL allows unmodified MPI and MapReduce applications. This is crucial for increasing the efficacy of our approach and easier adoption by the users, as any solution requiring modifications either to YARN or to MPI will induce additional maintenance difficulties or burden the application developers.

Ease of Use: MPI is the extant parallel programming framework for scientific computing, and users are already familiar with its process model and job submission semantics. We aim to preserve this ease-of-use and familiar user experience by enabling MPI-like interactions in GERBIL.

MPI-Aware Container Allocation: As explained earlier (Section II), the *fine-grained* container allocation in YARN does not match well with the *gang allocation* requirement of MPI applications. It is crucial for GERBIL to provide MPI-aware resource allocation atop the fine-grained resource allocation of YARN to mitigate this problem.

Tunable Allocation Strategy: GERBIL is aimed at multi-tenant environments, which entails adapting the resource allocation strategy based on dynamic cluser resource utilization and user demands. For example, the user may opt for minimizing job wait time, increasing overall resource utilization, or achieving better inter-process locality to reduce data movement across nodes. To facilitate this, GERBIL must support a flexible and extensible interface that would allow users to specify the allocation strategy best suited to their applications.

B. Architecture Overview

Figure 2 shows the architecture of GERBIL. We leverage YARN to manage the cluster, and allocate and launch containers on which MPI jobs would be run. We use the allocation to first provision a temporary MPI "cluster" (tMc) for each submitted MPI job. A tMc is composed of a set of containers that RM and GERBIL allocate, including a special container for the associated AM, and exists only for the duration of the associated job. GERBIL then instantiates tMc with an execution environment similar to that of a traditional MPI cluster. This ensures that no modifications or re-compilation is needed for running MPI jobs on the allocated resources comprising tMc. Moreover, multiple tMc's can co-exist with each other as they host different applications.

Options	Arguments
`-a, --mpi`	MPI executable file
`-l, --list`	MPI application list file
`-n, --processes`	number of processes
`-p, --priority`	proirity of the MPI application
`-f, --hostfile`	nodes/processes description file
`-r, --relax`	allocation algorithm to be used
`-o, --oversubscribe`	oversubscribe resources
`-i, --arguments`	arguments for the MPI application
`-k, --kill`	YARN application ID to terminate

TABLE I: Command-line arguments supported by GERBIL-Client.

GERBIL consists of three main components. GERBIL-Client is responsible for accepting user command-line arguments and providing the parameters and environments needed by YARN to launch GERBIL-AM. GERBIL-Client provides an MPICH compatible application launching interface shown in Table I. This allows MPI applications to transparently utilize YARN managed resources. After a user submits an MPI job, GERBIL-Client sets up the necessary environment for executing the application. The client then constructs the application submission context, including command line information, for the RM that schedules an available container and launches GERBIL-AM. At this point, control is handed over to GERBIL-AM.

GERBIL-AM is the customized application manager that handles execution of MPI jobs on YARN. This component requests needed containers from the YARN RM, manages the life cycles of allocated containers, assigns processes to containers based on user-specified allocation policies, and launches a GERBIL-Container on each allocated container. GERBIL-Containers launch associated MPI process manager, set up the MPI execution environment, and monitor the resource usage of MPI processes running within the containers.

C. User-Specified Container Allocation Modes

Table II shows the different modes supported by GERBIL, which allows the users to request cluster resources with specific characteristics. In the **`alloc_relax_none`** mode, a user submits an MPI host file describing the list of preferred hosts and the number of processes to run on each host. In this case, GERBIL strictly respects the specified requirements, requests one container per listed host and assigns exactly the number of processes per node as specified in the host file. This mode gives users fine-grained control over process allocation. The flip side is that when any one of the specified hosts does not have sufficient capacity to support the requested containers, GERBIL-Client has to wait until the resources become available for starting the GERBIL-AM. Here, users opt for control at the cost of potentially increased allocation latency. Alternatively, users can only specify the number of processes needed for their MPI jobs in the **`alloc_relax_all`** mode, where GERBIL freely allocates the nodes based on resource availability throughout the cluster. However, such flexibility comes at the cost of potential loss in data locality as containers are now assigned to arbitrary nodes.

Relaxation Mode: (`alloc_relax_`)	Location	Distribution	Node Count
`none`	·	·	·
`loc`	×	·	·
`dist`	·	×	·
`loc+dist`	×	×	·
`all`	×	×	×

TABLE II: Container request modes in GERBIL. For each mode, '×' and '·' denote whether a factor is relaxed or not, respectively. Here, location, distribution, and node count means the location (i.e., nodes on which processes are launched), the distribution (number) of processes per node, and the total number of nodes used to service a request, respectively.

The aforementioned allocation modes represent the two extreme cases of container allocation in GERBIL. To support a wider range of application needs as well as to mitigate the allocation latency due to the lack of gang scheduling support in YARN under multi-tenancy, GERBIL also supports three more container allocation modes with varying degree of flexibility as shown in Table II. These modes allow relaxing the requirement for the location of processes (**`alloc_relax_loc`**), distribution of processes per node (**`alloc_relax_dist`**), or both the location and distribution but not the number of requested nodes (**`alloc_relax_loc+dist`**). The different allocation modes provide users with flexibility to run their MPI applications based on different application characteristics by providing the appropriate host file and allocation mode arguments. For example, for communication-intensive MPI applications, it is preferred to allocate the containers and nodes close to each other so that the communication between processes on different machines does not involve multiple network hops. This can be achieved by using the **`alloc_relax_dist`** mode. Similarly, for CPU-intensive MPI applications where the location of the processes is not crucial, the **`alloc_relax_all`** mode provides for more flexibility and faster allocation. However, if the user wants performance guarantee, she can ensure a minimum number of compute nodes is allocated to the job by using the **`alloc_relax_loc+dist`** mode. Thus, the main purpose of supporting multiple allocation modes is to reduce application launch time without compromising their needs.

D. Container and MPI Process Management

GERBIL-AM is responsible for deciding the number and size of containers to request, as well as employing strategies for assigning processes to containers based on user-specified allocation mode and currently available cluster resources. The actual resource negotiation with the RM is done by GERBIL-AM using container requests and leases, communication for which are piggybacked on AM-RM heartbeat messages. To minimize the container launching time, we pack all the needed information, e.g., the number, size and location of the containers, etc., within a single heartbeat message. This information is used to assign processes to hosts depending on the allocated resource capabilities and the user-specified container allocation mode. GERBIL-AM then manages the application life cycle and affects policies to automatically recover from any failures.

1) Container Allocation and Process Assignment: MPI process management mechanisms force GERBIL to have only one container per node per MPI job, as only one MPI process manager per node is allowed per job. If multiple MPI processes are to be spawned on a node, they are all run inside the same container running on the node. Thus, while a node can have multiple containers each for a different MPI application, each application can have only one container per node. Moreover, keeping all the processes of an application in one container per node makes it easy for the GERBIL-Container to monitor and track the whole process tree, and determine the resource utilization of all the processes within the container. Thus, GERBIL-AM only requests one container per node for each MPI application, and assigns one or more MPI processes within each container. This is also beneficial in limiting the impact of YARN container allocation overhead on GERBIL as we show in our evaluation (Section V-F). Since GERBIL may host multiple MPI processes in a single container, the optimal size, e.g., the amount of memory and the number of vCPUs, of different containers can vary depending on the number of processes assigned to the container. For example, GERBIL attempts to allocate twice as much capacity, e.g., memory and vCPUs, to a container with two processes compared to a container that hosts just one process.

2) Resource Oversubscription: GERBIL-AM compares container allocation requests against available resources to detect oversubscription, which occurs if the number of requested resources exceeds the available resources. Such oversubscription may occur under any of the resource allocation modes described earlier in Section III-C. GERBIL handles such scenarios depending on the user command line option $oversubscription$. If the option is set to $false$,—and the requested resources do not exceed the max capacity of the setup, which results in an error—GERBIL simply waits until sufficient resources become available before allocating the container. If the option is set to $true$ (default), GERBIL-AM runs the MPI runtime in an oversubscription mode, i.e., it allows different containers/processes to compete for the same resources. The goal is to avoid the unpredictable wait times for resources to become available.

Different implementations of MPI handle such oversubscription related performance degradation differently. In OpenMPI [19], the degraded mode can be controlled by Modular Component Architecture (MCA) parameter `mpi_yield_when_idle`. In MPICH $v1.1$ or later, users can configure the system to control whether to perform in a degraded mode or to keep the performance but sacrifice the intra-node communication. Oversubscription can sometimes be beneficial, especially in competitive environments, e.g., modest levels of oversubscription has been shown to improve system throughput by 27% [36].

3) Container Allocation Algorithms: We design five container allocation algorithms in GERBIL-AM based on the five allocation modes described in Section III-C, namely **`ALGO_relax_none`**, **`ALGO_relax_loc`**, **`ALGO_relax_dist`**, **`ALGO_relax_loc+dist`**, and **`ALGO_relax_all`**.

Algorithm 1: Container allocation – common functions

```
Input: hostfile, relax, total_nproc, cluster_list
begin
    {m, v} ← minimum MEM, vCPU from YARN config;
    sort_list_descend(cluster_list);
    if hostfile exists then
        np ← total_nproc;
        local_list ← new localList;
        for each (node, nproc) in hostfile do
            node_size ← node.get_size();
            size ← {m × nproc, v × nproc};
            switch relax do
                case "none"
                    ALGO_relax_none();
                case "loc"
                    ALGO_relax_loc();
                case "dist"
                    ALGO_relax_dist();
                case "loc + dist"
                    ALGO_relax_loc+dist();
        if relax == "dist"||relax == "loc + dist" then
            round_robin(local_list, np);
        combine_containers(local_list);
    else
        /* relaxed allocation */
        ALGO_relax_all();
        combine_containers(cluster_list);
Function combine_containers(list)
    for each node in list do
        if node.get_num_containers() > 1 then
            combine containers into one;
Function round_robin(list, np)
    while np > 0 do
        node, node_size ← get first node from list;
        if node_size < {m, v} then
            degraded ← true ;
        request_container(node, {m, v});
        sort_list_descend(list);
        np − −;
```

Algorithm 2: Container allocation – mode implementation

```
Function ALGO_relax_none()
    if node_size ≥ size then
        request_container(node, size);
    else if node_size > {m, v} then
        request_container(node, node_size);
        degraded ← true ;
    else
        request_container(node, {m, v});
        degraded ← true ;
    local_list.add(node);
Function ALGO_relax_loc()
    top_node ← get the top node from cluster_list;
    top_size ← cluster_list.get_size(top_node);
    if top_size ≥ size then
        request_container(top_node, size);
    else if top_size > {m, v} then
        request_container(top_node, top_size);
        degraded ← true ;
    else
        request_container(top_node, {m, v});
        degraded ← true ;
    local_list.add(top_node);
    cluster_list.remove(top_node);
Function ALGO_relax_dist()
    local_list.add(node);
Function ALGO_relax_loc+dist()
    top_node ← get the top node from cluster_list;
    local_list.add(top_node);
    cluster_list.remove(top_node);
Function ALGO_relax_all()
    round_robin(cluster_list, total_nproc);
```

Algorithm 1 and Algorithm 2 show the steps taken during the container allocation process when supporting the five allocation modes. We set the $oversubscription$ option to a default of $true$ in our algorithms.

The input variables of Algorithm 1 are: (i) the host file path, $hostfile$, that contains a list of $\{node, nproc\}$ pairs where $node$ is a host and $nproc$ is the number of processes to run on the node; (ii) the relaxation mode $relax$; (iii) the total number of processes to use $total\_nproc$; and (iv) the cluster list $cluster\_list$ that specifies the available nodes and is retrieved by GERBIL-AM from the YARM RM. This algorithm retrieves the value of the minimum allocation of memory m and virtual cores v from the YARN configuration file and uses this information as the basic resource allocation unit. However, users can change the value of $\{m, v\}$. By default, a single process is assigned a unit of $\{m, v\}$ resources, while $nproc$ processes are assigned with $nproc \times \{m, v\}$ resources. We also support a user option that, if set to 'true,' implies that the system should sort the nodes from the cluster list in a descending order based on their available resources. We give higher priority to memory than CPU, as over-committing memory would cause extra disk I/Os and worse performance degradation than over-committing CPU resources. Thus, a node is ranked higher if it has more amount of free memory. If two nodes have the same amount of memory, then the node with more free virtual cores is ranked higher.

ALGO_relax_none: When the user specifies a host file with hosts and number of processes per host with **`alloc_relax_none`** mode, containers are requested strictly based on the user's request. For each node $node$ in the host file, GERBIL-AM adds it to a list of nodes to use, $local\_list$, and calculates the needed resources $size$ as $\{m, v\} \times nproc$. If the available resources $node\_size$ of $node$ is greater than $size$, GERBIL-AM requests a container of size $size$. If $node\_size$ is smaller than $size$ but greater than $\{m, v\}$, GERBIL requests a container of size $node\_size$. Otherwise, it requests a container of size $\{m, v\}$. Note that the last two cases result in over-committing of the container resources.

***A*LGO_relax_loc:** If the user selects mode `alloc_relax_loc`, the process distribution per node is respected but the specified hosts may be different depending on availability. GERBIL-AM retrieves the needed nodes from the top of the $cluster\_list$ to ensure a higher probability of getting the desired resources. The remaining process is the same as before. Finally, every time a container is requested on a node, the node is removed from the $cluster\_list$ and added to $local\_list$.

***A*LGO_relax_dist** and ***A*LGO_relax_loc+dist:** Under the modes `alloc_relax_dist` and `alloc_relax_loc+dist`, GERBIL-AM first builds the $local\_list$ from the $hostfile$ or the specified number of nodes from the top of the $cluster\_list$ for *A*LGO_relax_dist and *A*LGO_relax_loc+dist, respectively. Next, processes are assigned to the set of nodes in a round-robin fashion, where a process is assigned one $\{m, v\}$ resource at a time, and the process repeats until all of the $total\_nproc$ processes have been allocated.

***A*LGO_relax_all:** If the user only specifies the number of processes, GERBIL-AM employs round-robin process assignment using all the nodes in the $cluster\_list$.

Under all algorithms, if any of the nodes have resources less than the minimum $\{m, v\}$, the algorithm is blocked until more resources become available. Whenever over-subscription is detected, $degraded$ is marked as $true$. Finally, GERBIL-AM checks the $local\_list$ or $cluster\_list$ to combine multiple containers on the same node into one. This is done by adjusting the container size to make sure that only one aggregated container is requested per node per MPI job.

Since the algorithms are based on a snapshot of the available cluster resources, it is possible that the availability changes after a request has been issued but not serviced. The request can stall due to unavailability of actual resources. In order to reduce such wait time, we do not change the containers that are already allocated. Rather, we issue new requests for the remaining containers and wait until all new containers are allocated before launching the MPI job.

E. Discussion

a) Supporting HDFS I/O: GERBIL supports any distributed file system that is compatible with MPI. For clusters that are not equipped with a distributed file system, GERBIL supports reading and writing of data from direct attached storage by transparently transferring data between local storage and the default YARN distributed file system, HDFS. Moreover, alternative shared file systems such as fuse-dfs [15], HDFS NFS Gateway [10], GPFS [46], and GFS [33] can be also employed. Users can also leverage the HDFS native library to handle I/O in their programs directly. Nevertheless, it has been shown [31] that maintaining a conventional distributed shared file system on a commodity PC cluster, i.e., resources that typically support YARN, is operationally cheap compared with the performance cost of HDFS.

b) Handling Failures: Failures are the norm and not an exception in large-scale clusters. Thus, it is crucial to understand the failure behavior of our approach. In case of GERBIL-AM failure, YARN automatically relaunches the application master. Upon relaunch, GERBIL-AM requests the needed containers again and re-starts the MPI application automatically. This would be a heavyweight process and akin to cluster failure, e.g., due to power or networking loss, in traditional MPI setup. However, a more common failure is that of a container.

Container failure can lead to the failure of the hosted MPI job. GERBIL leverages built-in MPI fault tolerance to handle such job failures. There are a number of efforts in the MPI community to provide fault tolerance in case of process failure or communication failure, and help MPI programmers achieve resilience both at the application level and at the MPI framework level. To this end, techniques such as group communication can be used to write a robust MPI application [35]. MPI/FT [28] considers task redundancy to provide fault tolerance, while FT-MPI [34] adopts a dynamic process management approach. Moreover, different MPI implementations deliver different fault tolerance mechanisms. Users can register various built-in error handlers and customized handlers in different MPI implementations. For communication failures, techniques such as message re-transmission, message logging and automatic message rerouting have been developed [25], and can be used alongside GERBIL as needed.

Finally, checkpointing is widely supported for MPI job abort/recover in different MPI implementations [16], [19]. GERBIL supports checkpointing with different MPI implementations. For example in MPICH, GERBIL configures MPICH to checkpoint with BLCR [5]. Both manual and automated checkpointing with an interval can be configured in GERBIL. Although failures are handled by MPI, GERBIL-AM provides automated job restart without user intervention. GERBIL-AM first communicates with RM and checks whether sufficient number of containers are available based on the container allocation mode. Unless under heavy cluster load, the AM requests containers with the same size to replace the failed containers, after which the jobs are restarted.

IV. IMPLEMENTATION

We have implemented GERBIL with YARN Hadoop-2.2.0 and MPICH-$v1.1rc1$, though GERBIL can be easily extended to use other MPI implementations. For job submission, we extended the YARN standard client object, YarnClient, to GERBIL-Client that passes user specified arguments to YARN RM for launching GERBIL-AM.

1) GERBIL-*Container:* The GERBIL-Container encapsulates our MPI-specific handling mechanism and is responsible for starting the MPI MPD. The MPD in turn retrieves the appropriate CLC, and starts the MPI processes. We monitor the MPD through YARN NM and inform GERBIL-AM when the job is complete and the results are available.

2) GERBIL-*AM:* GERBIL-AM is responsible for requesting, allocating, and managing the containers, initiating the MPI runtime environment and launching MPI jobs. Figure 3 shows the different components of GERBIL-AM and their interactions. GERBIL-AM communicates with all of the associated

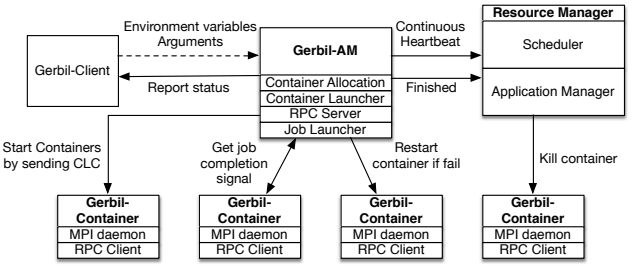

Fig. 3. GERBIL components and their interactions.

containers via RPC. To this end, an **RPC Server** is run within the GERBIL-AM and each of the launched container runs an RPC client. Communication such as when containers are ready for execution, when MPD has been launched/completed, and when a job is completed, is exchanged via the RPC Server. The **Container Allocator** implements the container allocation algorithms described in Section III-D3 to request needed containers from the YARN RM. The **Container Launcher** initiates the containers with the requisite execution environment, process/executable dependencies such as classpath, the application launching command, and arguments included in the CLC. The container launcher then sends the CLC to an appropriate NM and starts the associated GERBIL-Container. The **Job Launcher** starts the MPI job after GERBIL-AM has received the allocated containers and process assignments, using the following command:

mpiexec -hosts <number of host nodes> <host n ><number of processes> ... <mpi application> <-other options>

Upon job completion, GERBIL-AM collects the results (and writes them to HDFS), and returns the control back to GERBIL-Client, thus completing the process.

V. EVALUATION

A. Experimental Setup

We evaluate GERBIL using a 19 node (1 Master, 18 slaves) test cluster [21], where each node contains two Intel Quad-core Xeon E5462 2.8 GHz processors, 12 MB L2 cache, 8 GB memory and one 320 GB Seagate ST3320820AS_P SATA disk. Nodes are connected using 1 $Gbps$ Ethernet. The GERBIL-AM component is configured to use 3072 MB of memory and 2 cores. Each of the slave nodes has 6144 MB memory available for containers, with a minimum allocation of 1536 MB and 1 core. Thus, each node can have 4 containers, for a total of 70 containers available in our setup. We use MPICH version $1.1rc1$ with NFS for native MPI setup, and Hadoop-2.2.0 for GERBIL with NFS to support MPI-IO where needed in addition to HDFS. In the following, we conduct each experiment four times and report the average values.

B. Performance Comparison of GERBIL and Native MPI

In our first experiment, we compare the performance of GERBIL against native MPI. Here, we run only one job at a time to isolate the performance differences. For each application, we launch 18 containers using `alloc_relax_none` mode in GERBIL with one container per node and one process per container. Correspondingly for native MPI, we run 18 processes with one process per node.

First, we use MPI example programs CPI, MatrixMultiply, 2D-FFT and PrimeCount, which take small input and have relatively small execution time. For MatrixMultiply and 2D-FFT we use randomly generated input matrices, while CPI and PrimeCount need no input. Figure 4 shows the comparison of running these *small* applications using GERBIL versus native MPI cluster. We observe that GERBIL imposes a significant overhead (about 9 seconds) for all of the applications. To further understand this, Figure 5 shows the detailed execution time breakdown of the observed execution time overhead.

The ***AM Start*** phase involves job admission control, scheduling and container allocation for GERBIL-AM. Since we have only one job running at a time, this metric measures the RM overhead of YARN. The ***Container Allocation*** phase involves container negotiations between GERBIL-AM and the RM (dictated by 1 Hz heartbeat messages of YARN) as well as the container allocation time. The ***Container Ready*** phase involves container lease verification with the NM, configuring and instantiation of GERBIL-Container, loading the appropriate CLC, and starting the MPI daemon process. ***MPI Execution*** is the actual MPI job execution time. Finally, ***Finishing up*** shows the time used for cleaning up the containers and completion of GERBIL-AM and the job. We observe that while the MPI execution time is similar under GERBIL and native MPI, it is the YARN functions that incur the overhead.

Next, we repeat the experiment using six *large* MPI applications from the SPEC MPI 2007 benchmark [20]. These applications conduct large simulations from different domains, and have small input size (less than 50 KB) but employ complicated simulation algorithms. Figure 6 shows the results, which for this case shows only 0.3% average overhead across the applications under GERBIL compared to the native MPI. We also studied the breakdown for these applications as before, and as seen in Figure 7, compared to the MPI execution time the overhead is negligible.

These experiments show that the overhead of GERBIL is mainly due to YARN and would depend on the number of containers requested and not on the duration of the job. As discussed in Section III, GERBIL only allocates one container per node for hosting the MPI tasks, and thus the overhead is fixed (about $9s$ for our setup). Consequently, for the typical long-running MPI jobs, the overhead of GERBIL versus native MPI is amortized across job duration and becomes negligible.

C. Performance Comparison of GERBIL and MapReduce

In our next test, we compare GERBIL performance to MapReduce by running two compute-intensive applications: Pi calculation using Monte-Carlo (Pi-MC) and BaileyBorwein-Plouffe (Pi-BBP) algorithms with the same input parameters. we implemented the algorithms using an MPI version for GERBIL and a MapReduce version. We use 18 processes or mappers for each job in out test. Figure 8 shows that compared to MapReduce, GERBIL improves the average job execution time by 133% and 121% for Pi-MC and Pi-BBP, respectively.

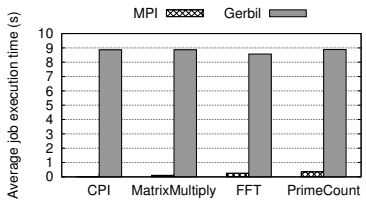

Fig. 4. Job execution time for small MPI applications.

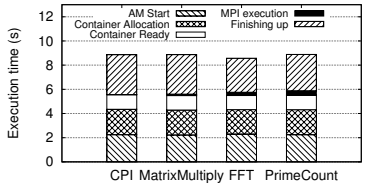

Fig. 5. Execution time breakdown for small MPI applications.

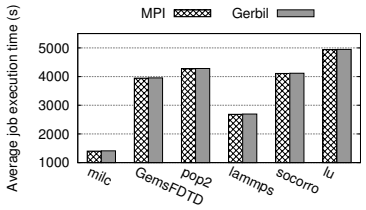

Fig. 6. Job execution time for large MPI applications.

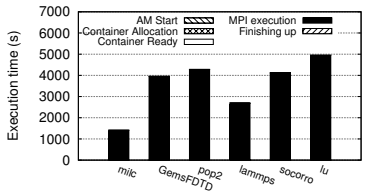

Fig. 7. Execution time breakdown for large MPI applications.

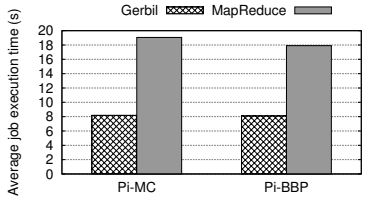

Fig. 8. Performance of GERBIL MPI vs. MapReduce application versions.

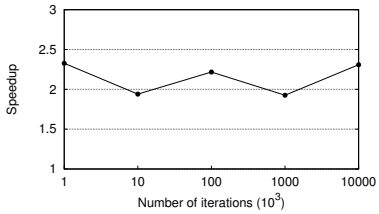

Fig. 9. Speedup of GERBIL MPI over MapReduce Pi-MC versions with increasing number of iterations.

Next, we selected the Pi-MC application and repeated the experiment with increased computational load by increasing the number of iterations. Figure 9 shows the job execution time under the MapReduce version normalized to that of MPI on GERBIL. We observe that GERBIL achieves a speedup of up to $2.3\times$. These tests illustrate that, as expected, GERBIL offers a better alternative than MapReduce for compute-intensive jobs.

D. Performance Impact of GERBIL *Container Allocations*

In our next set of experiments, we study the container allocation time and application execution time under the five algorithms of Section III-D3 in a multi-tenant environment. We carefully devise this test to investigate the numerous factors that impact these times, e.g., current cluster utilization, input data size, job arrival, etc. We first partition the cluster nodes into three groups Group A, B and C with 8, 6, and 4 nodes, respectively. We then launch a background job, BG1, with an approximate running time of 6 minutes on Group A, and BG2 (2 minute) on Group B. The background jobs can be MapReduce jobs, MPI jobs, or other jobs that run alongside the target MPI application. The background jobs consume all the containers in Groups A and B. Next, we wait for 5 seconds after the submission to ensure that BG1 and BG2 have started running within the containers. Then we submit one MPI application, Pi-MC (Section V-C) to the cluster, using one of the five different allocation modes. Each instance requests 16 total containers and two nodes per Group, with 3, 3 and 2 containers per node from Groups A, B and C, respectively. Table III shows the allocated container distribution on 6 of the 18 nodes for Pi-MC. We repeat the experiment for five times running Pi-MC with different allocation modes.

Figure 10 shows the container allocation and job execution times. Pi-MC has to wait for BG1 to finish before running under both **`alloc_relax_none`** and **`alloc_relax_dist`** modes, as no resources are available in Group A. Similarly, even though 4 nodes can be allocated in Group C under

	Group A		**Group** B		**Group** C	
Nodes	**`n1`**	**`n2`**	**`n3`**	**`n4`**	**`n5`**	**`n6`**
BG1	4	4	0	0	0	0
BG2	0	0	4	4	0	0
Pi-MC	3	3	3	3	2	2

TABLE III: Container allocation distribution with backgroud jobs BG1, BG2 and Pi-MC MPI job.

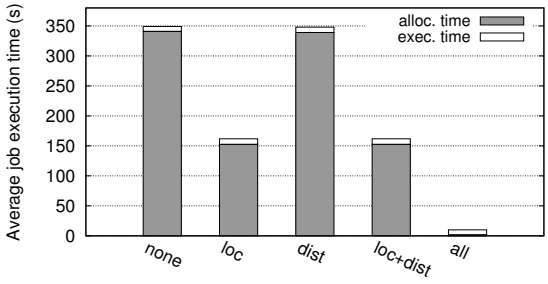

Fig. 10. Total job execution time under studied allocation algorithms.

`alloc_relax_loc`, the job has to wait until the remaining 2 become available in Groups A or B. Thus, the waiting time is non-deterministic depending on other jobs in the system. A similar behavior is observed under **`alloc_relax_loc+dist`** as well. Finally, as expected, the job is able to use all of the available resources in Group C under **`alloc_relax_all`** and exhibit the fastest performance. Note that the execution time of Pi-MC is the same under all cases. This is because this application is not sensitive to locality and there is no oversubscription. For applications that are sensitive to such factors, the five algorithms provide useful trade-offs to minimize the end-to-end application execution time, e.g., increasing allocation time to reduce execution time via better locality.

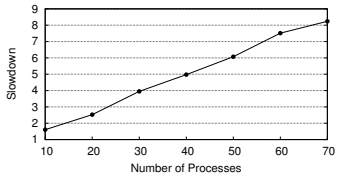

Fig. 11. Slowdown due to oversubscription with increasing number of processes, normalized to **alloc_relax_all**.

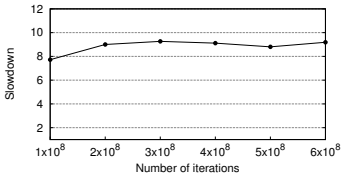

Fig. 12. Slowdown due to oversubscription with increasing number of iterations, normalized to **alloc_relax_all**.

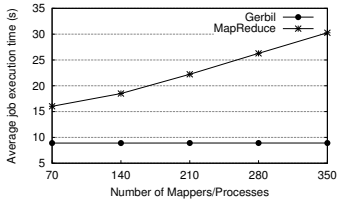

Fig. 13. GERBIL vs. MapReduce execution time under oversubsciption.

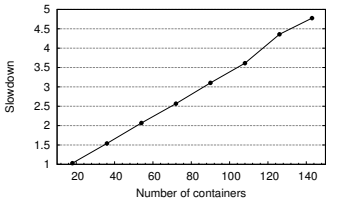

Fig. 14. Container allocation slowdown of YARN RM normalized to **alloc_relax_all**.

E. Impact of Oversubscription

In the next set of experiments, we investigate the performance penalty of oversubscription. For this purpose, we request an increasing number of processes under **alloc_relax_none** and force oversubscription by asking for all of the processes on a single node. We compare the results with the same requests under **alloc_relax_all**, where the requests are distributed to different nodes and no oversubscription occurs. Figure 11 shows the impact on Pi-MC execution time as the degree of oversubscription is increased, and shows a linear relation as more and more processes content for the same resources under **alloc_relax_none**.

Next, we repeated the experiment with the max of 70 processes as the computation is increased via increased number of iterations. In Figure 12, we observe that the slowdown of oversubscription remains unaffected (approximately $8.8\times$) by the computation load, and depends only on the number of competing processes.

YARN does not implement oversubscription for MapReduce. In case a MapReduce job has more tasks than the available capacity, it is queued until all the needed resources become available, even though such gang scheduling is not required for MapReduce. GERBIL mitigates this by employing oversubscription. In our next test, we study the impact of oversubscription on reducing allocation waiting times by comparing GERBIL with MapReduce. We run both the MPI GERBIL and MapReduce versions of Pi-MC with $10\ K$ iterations with 70 containers, i.e., the max in our setup. Figure 13 shows the results with increasing number of requested containers/processes. We see that while GERBIL immediately launches the required processes by oversubscribing them, MapReduce is unable to run all the mappers, instead running them in waves. Consequently, the execution time increases linearly under MapReduce but is effectively constant under GERBIL.

F. Impact on Application Launch Time

In our next experiment, we investigate the container launch overhead of the default YARN RM container allocator compared to GERBIL using **alloc_relax_all**. For this purpose, we request an increasing number of containers each with $100\ MB$ memory and 1 CPU core to a max request of $143(8 \times 18 - 1)$ containers possible on our setup. For each request, we measure the time it takes for the allocation to complete under the two cases. Figure 14 shows the slowdown in the YARN RM allocation normalized to GERBIL under **alloc_relax_all** as the number of requested containers is increased. Since the default container allocator works at the rate of heartbeat messages and can only allocate 18 containers at one time,—i.e., one container request per heartbeat to each of the 18 nodes in our setup—the allocation time increases linearly as more containers are requested. In contrast, GERBIL needs only one container per node to run multiple MPI processes; it can always allocate the needed containers in one heartbeat regardless of the number of processes. Thus, our design provides for mitigating long launching times as long as the user does not request additional constraints, i.e., under **alloc_relax_all**.

VI. RELATED WORKS

A number of recent works focus on bringing MPI and MapReduce models together. Ye et al. [51] proposed a modified OpenMPI that allows MPI jobs to run on Hadoop clusters via the Hadoop streaming [8] interface. Bai [26] implemented a hybrid framework of iterative MapReduce and MPI. These approaches have used Hadoop 1.0, and are limited in scope or are specialized. In contrast, GERBIL leverages the state-of-the-art YARN and provides general-purpose support for unmodified MPI applications on YARN. Conversely, projects such as MapReduce-MPI [12], DataMPI [41], and MR+ [18] supports MapReduce or Hadoop-friendly data-intensive jobs on MPI resources. However, these approaches do not port the well-established ecosystem around Hadoop, and thus are limited in utility. In contrast, GERBIL brings the entire MPI support to YARN and thus is expected to have easier transition and faster adaptation. The closest concept to GERBIL is Hamster [9], which was originally left unfinished but has recently been worked on by Yang et al. [17]. While Hamster also aims to support MPI applications on YARN, GERBIL is different and unique in its focus on efficiently provisioning and allocating resources. GERBIL also provides different resource allocation strategies that allow users to better match the resource allocation to the needs of their applications.

A number of works [22], [24], [49] also provide better application workflow processing by generating multi-language and multi-environment sub-workflows that can then be executed on clusters supporting the language/environment, and are complementary to GERBIL. However, GERBIL supports execution of two different models on the same set of resources and offers an integrated solution that avoids issues of supporting multiple types of resources/clusters and unnecessary data movement.

VII. Conclusion and Future Work

Co-hosting multiple programming models on the same resources helps avoid expensive data movement between clusters and greatly reduces workflow processing time. In this paper, we present GERBIL, a framework that supports the MPI programming model on YARN and enables execution of unmodified MPI applications on YARN managed resources alongside MapReduce applications. Our evaluation shows that the overhead of GERBIL is mainly due to the YARN framework and is observed to be constant regardless of the size of the applications. This is promising, as the overhead will become negligible when amortized over long-running applications. In our future work, we plan to extend GERBIL's static container management to consider dynamic cluster usage and runtime application profiling. We are also exploring techniques to reduce container allocation times of YARN to help mitigate the observed MPI instantiation latency. In our future work, we aim to enhance GERBIL to automatically select the best allocation strategy for applications based on their characteristics.

VIII. Acknowledgment

This work is sponsored in part by the NSF under the grants: CNS-1405697 and CNS-1422788.

References

[1] Amazon Web Services (AWS) - Clound Computing Services. http://aws.amazon.com/.
[2] Apache. Hadoop. http://hadoop.apache.org/.
[3] Apache Mahout: Scalable machine learning and data mining. http://mahout.apache.org/.
[4] AWS Case Study: Yelp. http://aws.amazon.com/solutions/case-studies/yelp/.
[5] Berkeley Lab Checkpoint/Restart (BLCR) for LINUX. http://crd.lbl.gov/groups-depts/ftg/projects/current-projects/BLCR.
[6] Crossbow: Genotyping from short reads using cloud computing. http://bowtie-bio.sourceforge.net/crossbow/index.shtml.
[7] Hadoop MapReduce. http://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html.
[8] Hadoop Streaming. http://hadoop.apache.org/docs/r1.2.1/streaming.html#Hadoop+Streaming.
[9] Hamster: Hadoop And Mpi on the same cluSTER. https://issues.apache.org/jira/browse/MAPREDUCE-2911.
[10] HDFS NFS Gateway. http://hadoop.apache.org/docs/r2.3.0/hadoop-project-dist/hadoop-hdfs/HdfsNfsGateway.html.
[11] Hydra process management framework. http://wiki.mcs.anl.gov/mpich2/index.php/Hydra.
[12] MapReduce-MPI Library. http://mapreduce.sandia.gov/.
[13] Message Passing Interface Forum. http://www.mpi-forum.org.
[14] MOAB Workload Manager. http://www.adaptivecomputing.com/products/hpc-products/moab-hpc-basic-edition/.
[15] MountableHDFS. https://wiki.apache.org/hadoop/MountableHDFS.
[16] MPICH. www.mpich.org.
[17] MPICH-yarn. https://github.com/alibaba/mpich2-yarn.
[18] MR+. http://www.open-mpi.org/video/?category=mrplus#Greenplum_RalphCastain.
[19] Open MPI: Open Source High Performance Computing. http://www.open-mpi.org/.
[20] Standard Performance Evaluation Corporation. http://www.spec.org/mpi2007/.
[21] System G. http://www.cs.vt.edu/facilities/systemg.
[22] The Kepler Project. https://kepler-project.org/.
[23] Wiki: Apache Hadoop. http://en.wikipedia.org/wiki/Apache_Hadoop.
[24] M. Albrecht, P. Donnelly, P. Bui, and D. Thain. Makeflow: a portable abstraction for data intensive computing on clusters, clouds, and grids. In *Proc. 1st ACM SIGMOD SWEET Workshop*, 2012.
[25] R. Aulwes, D. Daniel, N. Desai, R. Graham, L. Risinger, M. A. Taylor, T. Woodall, and M. Sukalski. Architecture of la-mpi, a network-fault-tolerant mpi. In *Proc. IPDPS*, 2004.
[26] S. Bai. *A Hybrid Framework of Iterative MapReduce and MPI for Molecular Dynamics Applications*. PhD thesis, Southern University, 2013.
[27] P. Balaji, W. Bland, W. Gropp, R. Latham, H. Lu, A. J. Pena, K. Raffenetti, R. Thakur, and J. Zhang. Mpich users guide. 2014.
[28] R. Batchu, Y. S. Dandass, A. Skjellum, and M. Beddhu. Mpi/ft: a model-based approach to low-overhead fault tolerant message-passing middleware. *Cluster Computing*, 7(4):303–315, 2004.
[29] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. Haloop: Efficient iterative data processing on large clusters. *VLDB Endow.*, 3(1-2):285–296, Sept. 2010.
[30] R. M. Butler, W. D. Gropp, and E. L. Lusk. A scalable process-management environment for parallel programs. In *7th EuroPVM/MPI Users Group Meeting*, 2000.
[31] G. G. Chuck Cranor, Milo Polte. Hpc computation on hadoop storage with plfs. Technical Report CMU-PDL-12-115, Parallel Data Laboratory, Carnegie Mellon University, Pittsburgh, PA, November 2012.
[32] N. Desai. Cobalt: An open source platform for hpc system software research. *Edinburgh BG/L System Software Workshop*, 2005.
[33] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proc. of the 19th ACM SOSP*, 2003.
[34] J. D. Graham E. Fagg. Ft-mpi: Fault tolerant mpi, supporting dynamic applications in a dynamic world. In *Proc. 7th European Users' Group Meeting on Recent Advances in PVM and MPI*, 2000.
[35] W. Gropp and E. Lusk. Fault tolerance in message passing interface programs. *Int. J. High Perform. Comput. Appl.*, 18(3):363–372, Aug. 2004.
[36] C. Iancu, S. Hofmeyr, F. Blagojevic, and Y. Zheng. Oversubscription on multicore processors. In *Proc. IPDPS*, 2010.
[37] M. Jette and M. Grondona. Slurm: Simple linux utility for resource management. In *Proc. of ClusterWorld Conference and Expo*, 2003.
[38] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *Proc. 9th IEEE ICDM*, 2009.
[39] Q. Ke, V. Prabhakaran, Y. Xie, Y. Yu, J. Wu, and J. Yang. Optimizing data partitioning for data-parallel computing. In *Proc. 13th USENIX HotOS*, 2011.
[40] V. Kunin, A. Copeland, A. Lapidus, K. Mavromatis, and P. Hugenholtz. A bioinformatician's guide to metagenomics. *Microbiology and Molecular Biology Reviews*, 72(4):557–578, 2008.
[41] X. Lu, F. Liang, B. Wang, L. Zha, and Z. Xu. Datampi: Extending mpi to hadoop-like big data computing. In *Proc. IEEE 28th IPDPS*, 2014.
[42] H. M. Monti, A. R. Butt, and S. S. Vazhkudai. Catch: A cloud-based adaptive data transfer service for hpc. In *Proc. IPDPS*, 2011.
[43] A. Murthy, V. K. Vavilapalli, D. Eadline, J. Markham, and J. Niemiec. *Apache Hadoop YARN: Moving Beyond MapReduce and Batch Processing with Apache Hadoop 2*. Pearson Education, 2013.
[44] I. Palit and C. K. Reddy. Scalable and parallel boosting with mapreduce. *Knowledge and Data Engineering, IEEE Transactions on*, 24(10):1904–1916, Oct 2012.
[45] J. Qiu, J. Ekanayake, T. Gunarathne, J. Choi, S.-H. Bae, H. Li, B. Zhang, T.-L. Wu, Y. Ruan, S. Ekanayake, A. Hughes, and G. Fox. Hybrid cloud and cluster computing paradigms for life science applications. *BMC Bioinformatics*, 11(Suppl 12):S3, 2010.
[46] F. Schmuck and R. Haskin. Gpfs: A shared-disk file system for large computing clusters. In *Proc. 1st USENIX FAST*, 2002.
[47] G. Staples. Torque resource manager. In *Proc. ACM/IEEE SC*, 2006.
[48] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache Hadoop YARN: Yet another resource negotiator. In *Proc. 4th SOCC*, 2013.
[49] J. Wang, D. Crawl, and I. Altintas. A framework for distributed data-parallel execution in the kepler scientific workflow system. *Computer Science*, 9(0):1620 – 1629, 2012.
[50] J. Xiang, H. Meng, and A. Aboulnaga. Scalable matrix inversion using mapreduce. In *Proc. 23rd HPDC*. ACM, 2014.
[51] J. Ye, J.-H. Chow, J. Chen, and Z. Zheng. Stochastic gradient boosted distributed decision trees. In *Proc. 18th ACM CIKM*, 2009.
[52] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proc. 2nd USENIX HotCloud*, 2010.