CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2017; 29: e3834
Published online 6 May 2016 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3834

Balancing push and pull in Confuga, an active storage cluster file system for scientific workflows

Patrick Donnelly*,† and Douglas Thain

Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN 46556, USA

SUMMARY

Most big-data analysis systems require users to adopt restricted abstractions to achieve scaling and system stability. While highly effective at establishing data locality and eliminating interdependencies, this approach is not easily incorporated into scientific workflows that are often complex and irregular graphs of sequential programs with multiple dependencies. To address this, we have developed an active storage cluster file system named Confuga which harnesses the file information already available in the workflow to enable efficient and controlled distribution of dependencies across active storage nodes. Confuga is built upon the idea of leveraging a job's namespace to eliminate unknown transfers and to plan the replication of all job dependencies. Replication is carried out through two opposing transfer methodologies: centrally managed push transfers and distributed pulls. We evaluate the effectiveness of the two transfer mechanisms using workflows that stress the ability of the cluster to replicate dependencies. Ultimately, we show that a balance of the two approaches achieves optimal file distribution. This is shown in two bioinformatics workflows where a careful balance of the two mechanisms leads to 48% and 77% improvements over only push or pull.

Received 4 December 2015; Revised 7 March 2016; Accepted 8 March 2016

KEY WORDS: transfers; cluster; file system; workflow; Confuga; Makeflow

1. INTRODUCTION

Today's big-data analysis systems require users to adopt certain structural constraints for their workflow to achieve scaling and system stability. These constraints are typically expressed in the form of workflow abstractions. For example, both MapReduce [1] and Spark [2] encourage the user to perform simple transformations on a monolithic dataset. This approach is highly effective when the objective is to compute relatively simple functions on colossal amounts of data. The small expense of writing or porting a small, widely known algorithm (such as k-means clustering) to these new platforms is well worth the payoff of running at colossal scale.

Unfortunately, these limited programming models are not so easily incorporated into scientific workflows with data requirements that cannot be efficiently processed by these abstractions. For example, bioinformatics workflows often depend on a large genome database which must be wholly present at each computation node. While many users may turn to MapReduce [3], they will find that the *map* abstraction is not designed to assist with moving common large data dependencies for parallel execution [4]. Furthermore, the underlying file system (e.g., HDFS [5]) is designed to only support parallel computation on chunks of a file. It offers no facilities for whole-file data parallelism. In the end, scientists are left to puzzle out how to access their dataset in a scalable way without destabilizing the shared file system supporting their workflow.

*Correspondence to: Patrick Donnelly, Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN 46556, USA.
†E-mail: pdonnel3@nd.edu

There have been efforts to make suitable data-locality-aware abstractions [6, 7] available to scientific workflows [8], but adoption is limited as abstractions impose certain structural constraints on the workflow. Instead, the scientific community has adopted a flexible workflow model composed of standard sequential applications chained together by data dependencies and represented as a directed acyclic graph (DAG) of jobs. This work is based on the observation that abstractions are tools to express data dependencies to the compute engine but scientific workflows already express sufficient structural information for scalable dependency management without the use of abstractions. Because each job includes its list of dependencies, traditional workflow management systems like DAGMan [9], Makeflow [10], Pegasus [11], or Swift [12] are able to order and parallelize the execution of jobs and to transport dependencies with jobs. Unfortunately, the underlying compute platform does not use this information to control data access for scalability (e.g., Condor [13]) nor does it communicate dependencies to any coupled distributed storage system (e.g., SGE [14] on Panasas [15] or Condor with Hadoop [16, 17]).

We have developed an active storage cluster file system named **Confuga** [18] which harnesses the file dependency information already available in the workflow to allow for the efficient and *controlled distribution* of files across active storage nodes. Confuga combines the *workflow model* of scientific computing with the *storage architecture* of distributed cluster file systems. End users place their datasets in Confuga using standard file manipulation tools and then direct their workflow manager to submit jobs to Confuga. In this way, Confuga acts as a replacement for existing batch execution systems. The user does not need to redesign their workflow or provide additional consideration to the management of data dependencies used in their workflow.

Confuga can be seen as an extension of the active storage concept to cluster file systems where storage nodes execute full POSIX application jobs as defined by the workflow. Originally, active storage began as smart disks [19] and grew within the HPC community to smart object stores which can harness unused CPU to perform simple functions on data [20] with the goal of increasing I/O throughput and reducing data movement [21]. More recently, projects like Hadoop were developed for clusters built on commodity hardware that are dedicated to performing structured computation on large datasets. Confuga is a natural evolution of this approach whereby users can execute whole applications with multiple dependencies and full data locality. The result is the merging of the batch execution system and the file system, allowing the file system to respond to the changing data requirements of workflow jobs.

Confuga is built around the idea of leveraging the **job namespace** to achieve a stable system. While typical distributed file systems must be designed to support runtime access to any file at any time, Confuga is able to scope job visibility of the global namespace to the job's own defined subset. This idea is fundamental to the design of Confuga. Requiring the declaration of the job namespace allows Confuga to unobtrusively eliminate dynamic transfers by jobs and plan the replication of all job dependencies. This management of transfers empowers Confuga to control load on the cluster.

Confuga manages and tracks the transfer load on the cluster using two opposing methodologies: **push** and **pull** transfers. A push transfer is used to direct a storage node to replicate a file to another storage node. Each push is centrally scheduled and tracked to control file distribution which may prevent performance killing hot-spots. A pull transfer resembles a more traditional file distribution technique where the job fetches missing files itself prior to running the application. Confuga uses pulls to selectively off-load transfer scheduling and management to storage nodes when a controlled distribution has fewer benefits.

This paper examines the benefits of using push transfers in the cluster to control network and disk load on storage nodes. Pulls are used as a basis of comparison for existing uncontrolled dependency distribution techniques used by today's batch execution platforms. We have also found that there is a balance to strike between pushes and pulls: the careful use of pull transfers can avoid inefficiencies introduced by centralized management of transfers. We will show the following:

- Pushes enable full disk and network utilization of storage nodes. Load control of transfers can allow for an efficient spanning tree distribution that optimally distributes files in parallel. Using push transfers, Confuga achieves a 77% speedup over unmanaged replication via pulls. (Section 4.1)

DOI: 10.1002/cpe

- Replicating several large dependencies reduces the mutual interference pulls suffer, but performance is still unpredictable. On the other hand, push transfers still eliminate all load instability caused by concurrent transfers in a predictable way. (Section 4.2)
- When jobs have several dependencies that must be pulled, it is essential that these transfers occur in a random order to avoid hot-spots. The effects of hot-spots are usually only visible for larger files as pull transfers are more likely to interfere. (Section 4.3)
- While push transfers allow for fast distribution of large files through structured and high bandwidth transfers, there is room to tolerate some interference and hot-spots from pull transfers for small files. Push transfers for smaller files do not give a justifiable improvement to distribution time when there is pressure to transfer other larger files. Instead using pull transfers introduces small amounts of interference, but individual pull and push transfer bandwidth is mostly unaffected. (Section 4.4)

We conclude in Section 5 with two representative bioinformatics workflows evaluated using the push and pull transfer mechanisms. Ultimately, we show that a balance of the two mechanisms achieves optimal file distribution leading to 48% and 77% improvements over only push or pull.

This paper is an expansion on previously published work. A workshop paper [22] introduced the design and motivation of Confuga. A conference paper [18] studied the metadata benefits Confuga brings, its use in bioinformatics workflows, and tuning push transfers. Here, we study Confuga's ability to manage transfer load on the cluster and how to address the dilemma of push and pull transfers.

2. THE PROBLEM SETTING

Confuga targets typical researchers in science and engineering looking to execute a large workflow composed of regular POSIX executables on a large dataset. Normally, these workflows would be executed on clusters and grids to achieve scalability, but the introduction of large persistent data dependencies requires a more structured system that can support active storage and scalable replication.

To answer this need, Confuga offers a distributed active storage model for achieving data locality and parallelism. Users upload their workflow datasets to the Confuga distributed file system using existing file management utilities. Data processing is performed using existing workflow tools with Confuga as a drop-in replacement for their existing batch systems. In this way, Confuga combines the file system and the batch interface into a single system.

Confuga is a cluster file system composed of a head node and multiple storage nodes. The head node is the entry point to the system by external clients. Both file I/O and job submission are carried out through the head node which is responsible for all manipulation of storage nodes. The architecture is visualized in Figure 1.

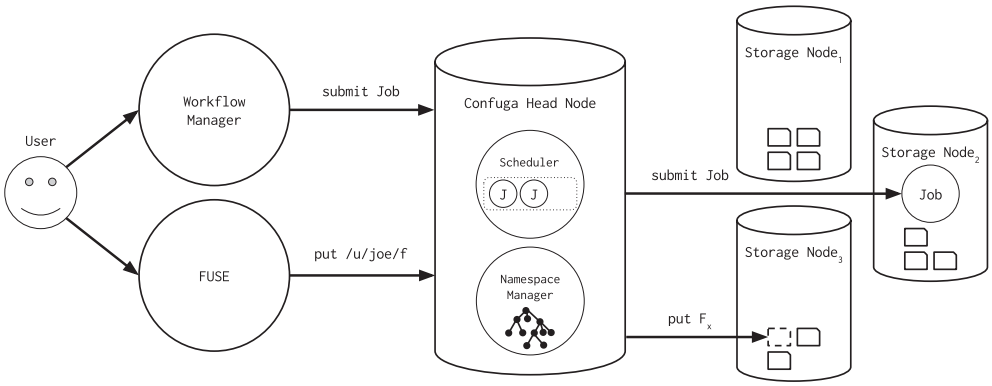


Figure 1. Confuga architecture.

DOI: 10.1002/cpe

2.1. Storage model

Just like GFS [23] and HDFS [5], the Confuga head node manages the cluster namespace and other file system metadata. Metadata and directory hierarchy operations like `stat`, `mkdir`, and `unlink` only require changes to the state on the head node.

An `open` of a file by external clients is also mediated by the head node. The creation of new files will result in the head node creating a new open replica on an available and randomly selected storage node. When opening an existing file, the head node looks up storage nodes hosting the replica, connects to an available storage node, and performs *read* operations on the client's behalf. We have chosen this model of Confuga mediating replica creation and reading to ease development and to simplify the security domains between clients and storage nodes. In particular, it allows external clients to browse the file system and load data into or pull data out of the cluster.

Storage nodes are used as dumb storage, unaware of their role within the cluster file system. File replicas are stored in whole on one or more storage nodes, not as chunks as in GFS/HDFS. Each replica is managed and tracked by the head node. Replication is performed by transfers created by the head node. This allows for redundancy and increased data parallelism for jobs. Storage nodes and the jobs they run do not independently interact with the head node. Instead, the head node dictates which replicas on a storage node the job may access. This is discussed in depth in Section 3.1.

The head node tracks replicas within a flat namespace on storage nodes. Replicas are named according to their replica identifier (RepId) that is either a universally unique identifier or the SHA1 hash of the replica content. Hashes are used for basic deduplication of files. In most circumstances, a SHA1 hash is used for the RepId except when a large output file is created by a job. Because jobs operate within a sandbox on a local POSIX file system, storage nodes cannot compute the hash of output files until after the job executable exits. To avoid delaying job completion in order to hash large output files (currently >16 MB), the storage node will assign a universally unique identifier instead. The head node learns the RepId of new output files after reaping jobs. Overall, the use of RepIds allow storage nodes to safely assign a content identifier for new files created by jobs without head node involvement.

Access to storage within Confuga is protected through three authentication realms: client to head node, head node to storage nodes, and storage node to storage node[‡]. Clients authenticate with the head node using several interoperable enterprise technologies. The head node is configured at startup to use a specific authentication credential to access all storage nodes. The head node's credential enables complete access to state located on storage nodes. In our campus cluster, we use a long duration ticket credential [24] that provides the strict subset of access the head node should have on storage nodes. Storage nodes access other storage nodes using a separate ticket which is setup and periodically renewed by the head node. The head node protects access to the ticket through access controls on the storage node file system and allows only jobs executing within the Confuga context access to the ticket. The ticket provides an even stricter subset of access, following the principle of least privilege, that only allows reads of replicas and the creation of new replicas in a separate directory. The latter restriction allows Confuga to check for successful replica creation (with consideration to myriad failures) before making the replica usable.

The Confuga cluster uses the Chirp [25] distributed file system for storage nodes. Clients also use the Chirp protocol to interact with the Confuga head node, which is based on previous work exporting HDFS for use on a campus grid [17]. Users may use a familiar POSIX-style I/O interface to interact with the file system through convenient command line tools, Parrot [26], FUSE [27], or the Confuga API. File operations on Confuga largely follow POSIX consistency semantics except new files are visible only after close, and files may only be written to once (just like HDFS). We have found these semantics are sufficient for clients to upload, manipulate, and download their datasets in Confuga. The user is free to organize files in a regular directory hierarchy with per-directory access controls that enable fine-grained sharing with colleagues.

[‡]Each storage node operates a multi-user remotely accessible file system which, depending on the cluster, may be used by other clients. For this reason, it is necessary for Confuga to protect its state by establishing secure authentication mechanisms.

DOI: 10.1002/cpe

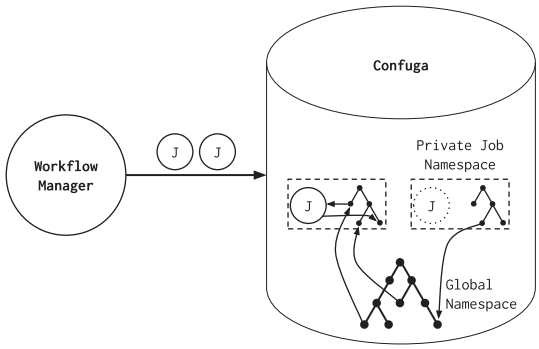


Figure 2. Confuga execution model.

2.2. *The execution model*

After users place large datasets on Confuga, they may begin running workflows. As shown in Figure 2, Confuga presents itself as a single system image [28] which executes multiple jobs that read from and write to the cluster file system. Each job executes an opaque executable within a private job namespace (i.e., a sandbox) constructed from a specification of the job's input and output files. **During execution, jobs cannot see the global file system, only their own sandbox.** All input files are read atomically prior to a job starting. On job completion, the global namespace is atomically updated with the new output files by the head node.

Jobs are submitted to Confuga using a traditional *submit* and *wait* RPC interface with two-phase commit for reliability. Each job specification is encoded in JSON [30], with attributes like the executable name, arguments, and environment. Confuga also requires the inclusion of the *job namespace* which lists the mapping of read-only input files from the global namespace to the sandbox and of output files from the sandbox to the global namespace. This namespace mapping is static and cannot be changed during job execution. Jobs may also access system files outside their sandbox on the storage node including executables (such as the shell), libraries, and so on. Listing 1 shows an example job specification.

```
{"executable": "./script", "arguments": ["./script", "-a"], "files": [
 {"confuga_path":"/u/joe/bin/sim" ,"sandbox_path":"script"  ,"type":"INPUT" },
 {"confuga_path":"/u/joe/data/db" ,"sandbox_path":"data/db" ,"type":"INPUT" },
 {"confuga_path":"/u/joe/wf/out1" ,"sandbox_path":"out"     ,"type":"OUTPUT"}
]}
```

Listing 1. Simple job description in JSON.

Normally, users do not concern themselves with writing these job specifications. Instead, Confuga expects to be invoked by a workflow manager, the user agent which submits and manages jobs on behalf of the user. Confuga does not order job execution by any specified dependency, and this is the workflow manager's responsibility. Jobs are tied together through a DAG which orders jobs by file dependencies: one job's output file becomes the input of the next. Figure 3 shows one of these DAG-structured workflows that we have run.

Our collaborators use the makeflow [10] workflow manager that builds on the venerable make syntax for expressing job dependencies, which creates an implicit job execution order. Given a makeflow specification file, makeflow creates a DAG of the entire workflow, submits jobs as dependencies become available, and handles certain workflow fault tolerance policies. It is designed to

DOI: 10.1002/cpe

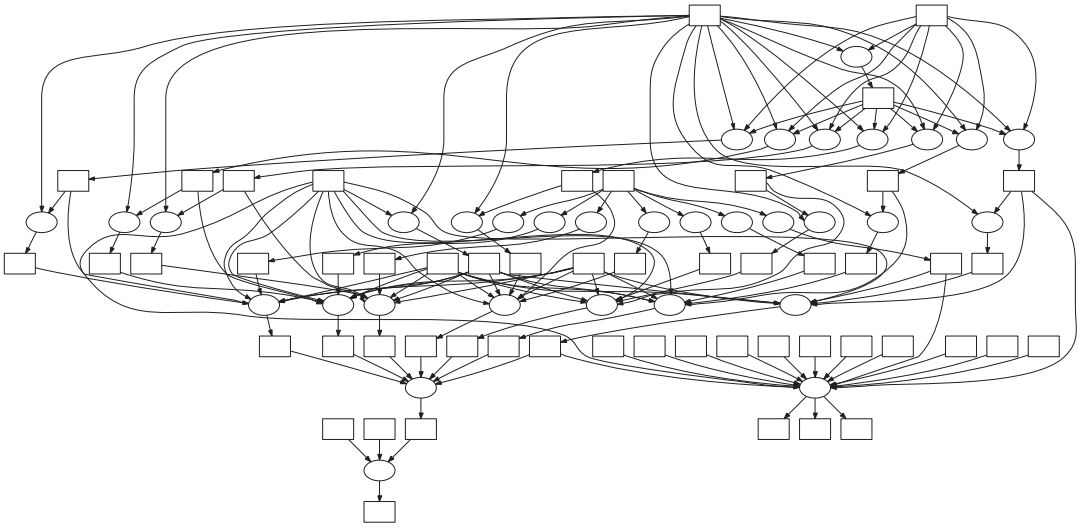


Figure 3. A typical directed acyclic graph-structured workflow. This comes from a bioinformatics workflow using SHRiMP [29]. Here, files are boxes, circles are tasks, and lines indicate dependencies.

easily switch between execution platforms and currently supports Condor [13, 31], SGE [14], Work Queue [32], and other systems. To programmatically create large workflows and the workflows used in our experimental results, we use the Weaver [8] workflow compiler.

Concurrent job execution in Confuga comes from dispatching jobs to multiple storage nodes. A scheduler on the Confuga head node handles the details of assigning jobs to storage nodes for execution, replicating necessary dependencies, and global namespace manipulation. The head node monitors the health of storage nodes via heartbeat messages from storage nodes sent to a catalog service. Using the catalog, the head node learns of unavailable storage nodes, newly available storage nodes, and other file system statistics. When the scheduler learns of a failure because of a lost storage node or a failed job, it will reschedule the job if the failure is transient (e.g., a failed transfer) or pass the failure to the workflow manager if it cannot be handled.

Each job submitted to Confuga goes through several states. First, the scheduler performs namespace remapping which allows the job to be executed on storage nodes. This is a static translation that is independent of the storage node the job will execute on. Once the new namespace mapping is constructed, it is scheduled or assigned to an available storage node with preference towards a node with the most input file bytes (as some files are larger than others). Next, the head node decides how to replicate missing input files because jobs must execute with all inputs files in their sandbox. Finally, the job is submitted to the storage node for execution. After several periodic waits, the scheduler will reap the finished job and set the job's outputs in the global namespace. We expand on the details of namespace remapping and replication of missing dependencies in Section 3.

In Confuga, data parallelism is achieved in two ways. Firstly, the user constructs their workflow in a way that jobs use whole dependencies. (This model is often structurally incompatible with other big data abstractions like MapReduce where data parallelism is established by mapping jobs to chunks of a monolithic file while expecting each job to largely read only the mapped chunk.) Because Confuga must work to obtain all data dependencies at the site a job executes, the effort is only justified if the job actually needs the whole dependencies. Secondly, Confuga adjusts the replication of data to respond to needs of jobs. When a job dependency is missing from the storage node that the job is to be executed on, Confuga will plan the replication of the file to that storage node. This dynamically responds to demand for hot files and allows increased replication to benefit future jobs relying on that dependency.

DOI: 10.1002/cpe

3. MANAGING TRANSFERS IN THE CLUSTER

The design of Confuga relies on the inclusion of the job's namespace. This enables the head node to make decisions knowing the dependencies for all jobs and avoids many common situations that cause instability within typical cluster file systems. In particular, unknown dynamic transfers are not permitted, and jobs have limited visibility of changes to the global namespace. This allows Confuga to plan transfers for unavailable job dependencies in a way that avoids load instability in the cluster.

In this section, we first describe how the scheduler manipulates the job namespace so jobs may execute with limited global namespace visibility. Next, we will discuss how the head node uses push transfers to manage transfer load within the cluster. Finally, we will introduce pull transfers which allow Confuga to delegate some of the transfer management to storage nodes.

3.1. Namespace remapping

The foundation for the design and optimizations of Confuga is the full description of the job namespace: each scientific workflow includes the full input and output file list for each job. This description is provided in the form of a **namespace mapping** of the job's namespace or sandbox to the workflow namespace. In Confuga, this workflow namespace would be a sub-tree of the global cluster file system namespace.

Prior to scheduling a job, Confuga performs access control checks for each input and output. These access controls are maintained per-directory. Once access control checks are complete, Confuga will **bind** each input file by looking up its replica identifier. Input files that are directories expand recursively to an equivalent input file list. Replicas in Confuga are complete and immutable, so this operation effectively causes each job to atomically read its inputs from the global namespace prior to execution. Binding each input file to a replica is carried out through namespace remapping, as shown in Figure 4. Each input file in the sandbox of the job is remapped to its corresponding replica in the flat replica namespace. Similarly, when a job completes, Confuga learns the replica identifiers for each of the job's output files and atomically updates the global Confuga file system namespace. Directories as output files are not permitted.

Besides constructing the sandbox and fetching necessary dependencies, Confuga relies on the job namespace to perform several optimizations: (a) Lookup and place replica locations for all input files with the job description which saves future lookups by the job (namespace remapping). (b) Prohibit access to the global namespace which enforces workflow consistency semantics and prevents uncontrolled dynamic file access. (c) Batch metadata and access control checks before job dispatch which improves performance and reduces scope of security checks. These optimizations

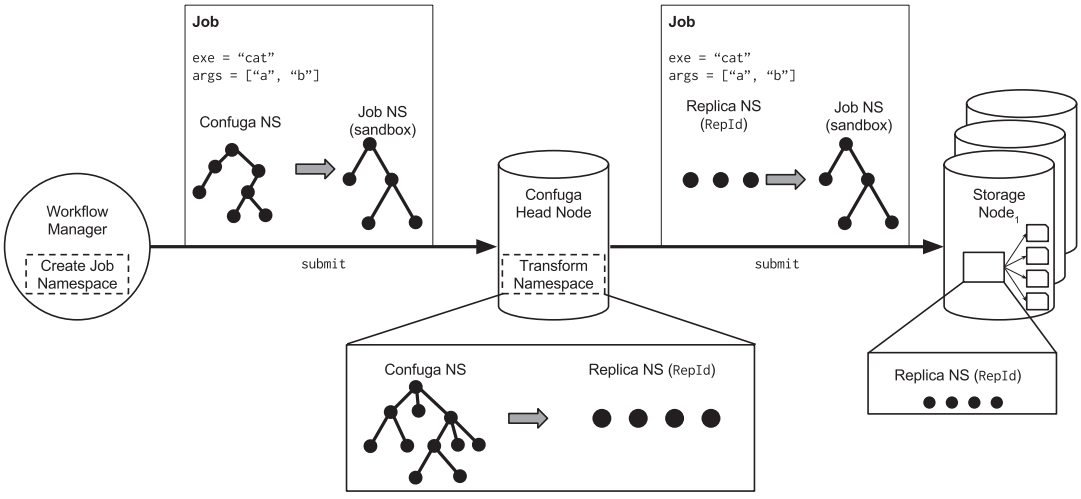


Figure 4. Confuga job namespace remapping.

DOI: 10.1002/cpe

and limitations affect the consistency semantics of the file system at the level of the workflow (jobs still execute within a POSIX sandbox). We refer to these consistency semantics as **read-on-exec** and **write-on-exit**.

Ultimately, exploiting the job's namespace mapping allows us to massively reduce metadata and file lookup load on the Confuga head node, but even more importantly, it allows Confuga to plan file transfers.

3.2. Push transfers

A push transfer is a head-node initiated replica transfer from a source storage node to a target storage node. The function of push transfers is to protect the cluster from load instability caused by uncontrolled transfers among storage nodes. Additionally, the head node may use push transfers to track transfer progress, limit transfer concurrency, and respond to failures.

Confuga implements push transfers using transfer jobs. These are special jobs executed by storage nodes which execute file system operations within the storage node file system instead of a user application. The head node benefits from this logical extension of jobs through asynchronous execution, code reusability, and reliable creation, tracking, and reaping of transfers within a distributed context. A failed transfer job is handled by the scheduler similarly to regular jobs (i.e., not transfer jobs); the head node will reschedule the transfer if it is a transient failure otherwise pass the fault up to the job which scheduled the transfer.

Transfer jobs execute the `putfile` RPC [25] which copies a given source replica to a temporary file on the target storage node. The head node monitors the job's progress through periodic `job_wait` RPCs on the source storage node and `stat` RPCs of the temporary file on the target storage node. When the transfer completes, the head node will atomically move the temporary file to the replica namespace on the target storage node. This prevents an incomplete replica from being used by other jobs. Authentication for transfer jobs is managed through authentication tickets which are setup and periodically renewed by the head node. This allows restricted storage node to storage node authentication and access control. Confuga's use of tickets was discussed in Section 2.1. The entire transfer job process is visualized in Figure 5.

The Confuga head node organizes push transfer load management using transfer slots. Each storage node has a set number of transfer slots which are occupied by pushes. An active push transfer uses a transfer slot at both the sender and the receiver. This mechanism enforces load control by limiting the number of push transfers a storage node may participate in. If the head node wishes to use an occupied transfer slot, it must wait for the active transfer to complete. This work limits the number of transfer slots to one. Previous work [18] explored the benefits of push concurrency by varying the number of transfer slots.

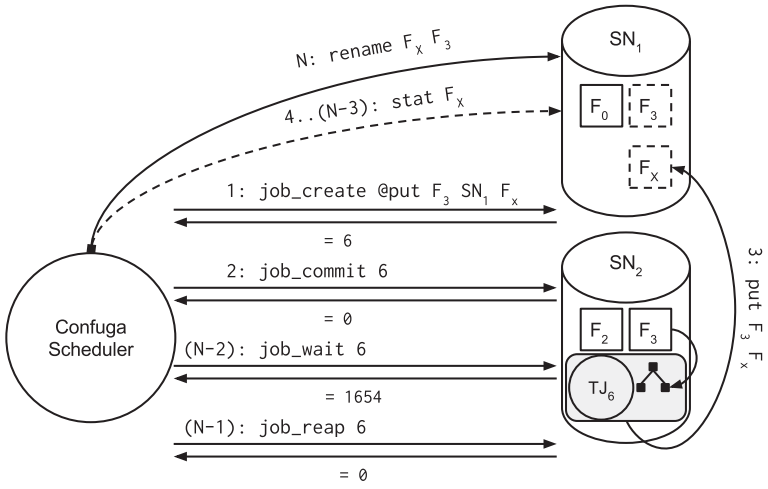


Figure 5. Push transfers are a series of short blocking operations that create a transfer job on the storage node. The transfer job executes asynchronously with the head node. Operations are numbered in order.

DOI: 10.1002/cpe

A job is not dispatched to a storage node until the head node is finished executing any required push transfers. However, other storage nodes involved in a push transfer may be executing regular jobs. So, while the number of concurrent push transfers is limited by the number of transfer slots, contention for storage node resources may still occur. In particular, some pushes may use up I/O otherwise used by a regular job.

3.3. Pull transfers

Pull transfers are replica transfers which are executed by the storage node executing the job and prior to the execution of the job application. Pulls allow the Confuga scheduler to delegate transfer management to storage nodes. This frees the scheduler to devote resources elsewhere, but pulls may introduce load instability on storage nodes. For example, several nodes may pull from the same replica simultaneously, with deteriorated performance. Additionally, as with push transfers, a job may pull a file from another storage node executing a job.

Pull transfers resemble normal whole-file sequential reads in a typical distributed file system. For example, a job *opens* the file, *looks up* an available replica, *reads* parts of the replica, and then *closes* the file. In Confuga, pull transfers behave similarly but differ in several important ways. First and foremost, a storage node does not decide *which* files are pulled. The Confuga scheduler is free to perform push transfers for some files and leave remaining dependencies to be pulled by the storage node. Second, jobs do not and cannot initiate a transfer during execution. Pulls are performed prior to application execution. Third, because the entire namespace is known, the Confuga scheduler is free to perform replica lookups in batch prior to job dispatch.

Pulls are executed as part of setting up the job sandbox by storage nodes. Because Confuga knows the replicas on each storage node, it is able to write the job description so that each input binds to either a replica on the storage node or to one or more replicas on other storage nodes. For each input file without a local replica, Confuga will include a set of randomly chosen remote replicas specified as a list URLs [33]. The storage node will attempt to fetch each URL for the file until success. The result is placed within the job's sandbox. If a pull ultimately fails to obtain the file from any of the possible replicas, then the job will abort. This failure is responded to by the head node when the job is reaped by either creating a new job or passing the failure up to the workflow manager.

4. EVALUATION

Now, we will move on to evaluating Confuga's use of push and pull transfers in several workflows. These experiments will explore the benefits gained by controlling cluster load through pushes or

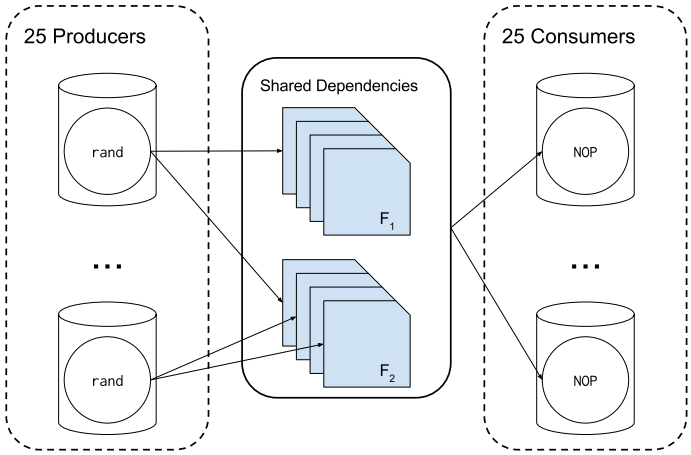


Figure 6. Producer/consumer workflow to stress the Confuga cluster through transfers of several dependencies. Each of the 25 storage nodes produce 1/25 subset of shared dependencies. The scheduler then must completely distribute these files across the cluster for 25 consumers (one consumer per storage node), which execute no operation. **Workflow A:** Shared dep.: 1 · 32 GB file. **Workflow B:** Shared dep.: 25 · 32 GB files. **Workflow C:** Shared dep.: 1 · 64 GB, 2 · 32 GB, 4 · 16 GB, 8 · 8 GB, 16 · 4 GB, 32 · 2 GB, 64 · 1 GB files.

DOI: 10.1002/cpe

relaxing control through pulls. To do this, the experiments are structured to stress the head node's ability to replicate dependencies across the cluster.

This work does not scrutinize the scheduling of jobs (i.e., assigning jobs to storage nodes for execution), which achieves certain well-studied goals like fairness. We acknowledge that scheduling (especially *rescheduling*) can minimize or eliminate data transfers but that analysis is beyond the scope of this paper. Our concern in these experiments is how to efficiently manage transfers once jobs are placed to minimize distribution time and storage node load.

We use a two-stage producer/consumer workflow shown in Figure 6 to evaluate Confuga's ability to distribute dependencies across storage nodes for various scheduler configurations. The function of the producers is to quickly generate the pool of dependencies randomly across all storage nodes. Each consumer is assigned by the scheduler to one of the available storage nodes, and dependencies are replicated.

We note here that we do not include a variation on this workflow where consumers also access files which are unique to their execution (i.e., not shared with other consumers). This variation is generally uninteresting when analyzing push and pull configurations because unique files are only replicated at most once, so therefore it is unlikely for this to introduce significant contention or load.

Cluster Hardware: We use a rack of 26 Dell PowerEdge R510 servers running RedHat Enterprise Linux 6.6, Linux kernel 2.6.32. Each server has dual Intel(R) Xeon(R) CPU E5620 @ 2.40GHz, for eight cores total, 32GB DDR3 1333MHz memory, a 1Gb link to a Summit X460 switch delivering 220Gbps aggregate bandwidth. Our tests use one Seagate ST32000644NS 2TB disk on each server, with advertised 140MB/s sustained I/O bandwidth, 8.5 ms seek time. Confuga's storage nodes each use a single disk formatted with the Linux *ext4* file system. For evaluation, we use one node as the head node and the 25 other nodes as storage nodes.

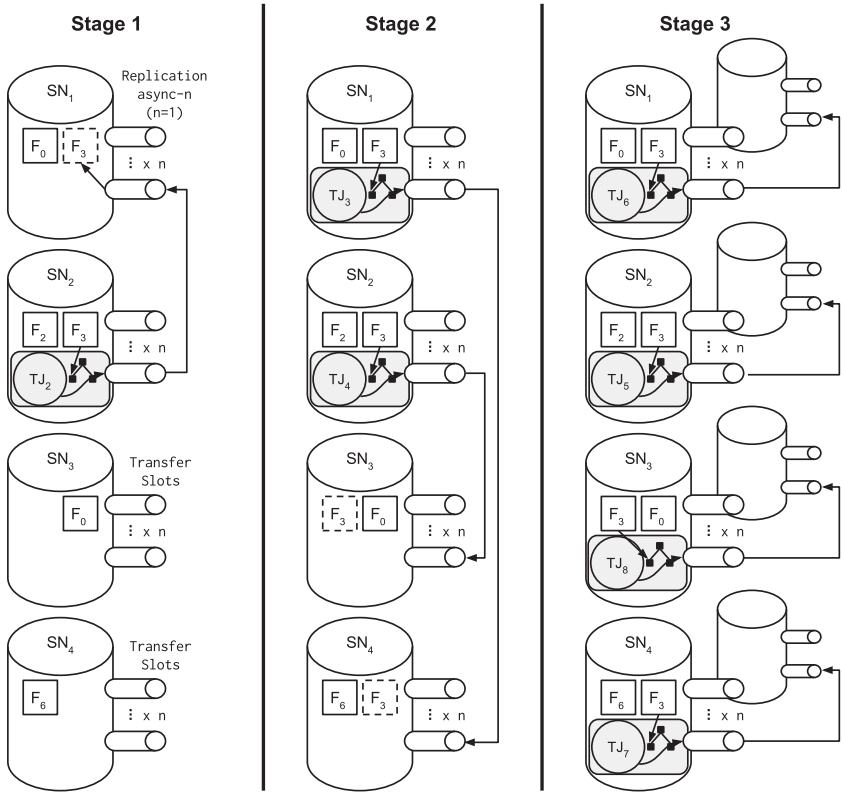


Figure 7. Spanning tree of push transfers. Each push is a transfer job (TJ). One transfer slot per SN ($n = 1$).

DOI: 10.1002/cpe

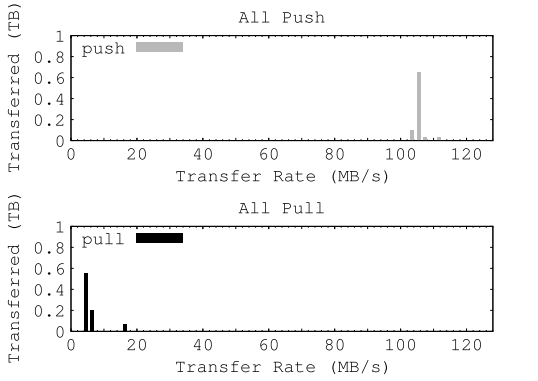
4.1. Spanning tree distribution

In this section, we will test our hypothesis that using push transfers can help control load on the cluster and improve file distribution time. Confuga performs load management by limiting concurrent push transfers for a storage node via transfer slots. This load management will result in a tree distribution of the file. The form of the tree is determined by the transfer load on the cluster. Under conditions where the file may be continually replicated using new replicas as they become available, a spanning tree file distribution [25] develops.

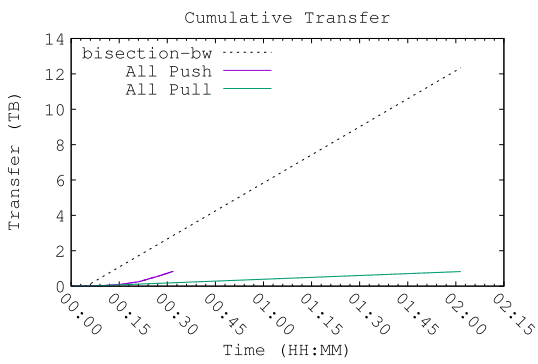
This process is visualized in Figure 7. The Confuga scheduler allocates to each storage node a single transfer slot which limits the storage node to one incoming or outgoing transfer. A transfer job is dispatched by the scheduler which occupies the transfer slot of the storage node it executes on and the transfer slot of its target. The use of transfer slots allows the scheduler to control the number of pushes executing in the cluster and to functionally create a spanning tree distribution for files.

We examine a workflow which requires the distribution of a single large file across all storage nodes. This is carried out using **Workflow A** described in Figure 6, where all consumers require a single shared 32 GB file. We look at the two cases where the file is distributed using push transfers or pull transfers.

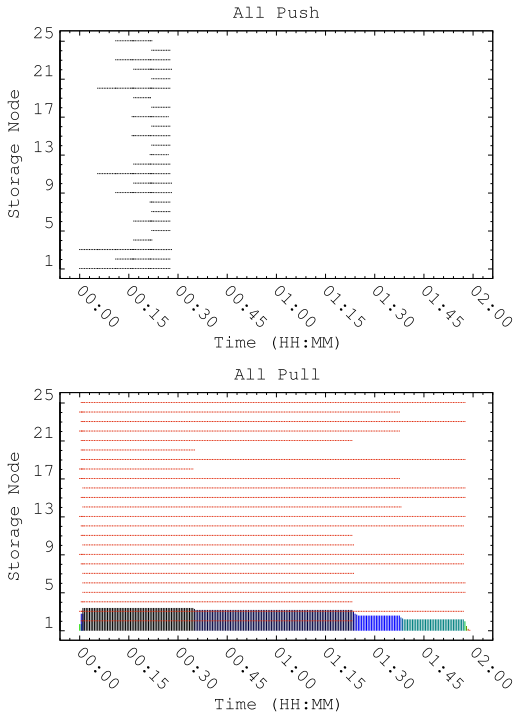
Figure 8 shows the results. For 'All Push' in (c), Confuga is visibly able to achieve a spanning tree distribution of the 32 GB file using push transfers. At the start, node 1 pushes the file to node 3. At



(a) Average Transfer Speed Histogram



(b) Cluster Cumulative Transfer



(c) Transfer Density (expl: see below caption)

Figure 8. **Workflow A:** single file spanning tree distribution; (a) Average transfer speed histogram groups transfer by average transfer rate and show the total bytes transferred for each group; (b) Cluster Cumulative Transfer visualizes the cumulative bytes transferred across the cluster. The `bisection-bw` line indicates the maximum bisection transfer bandwidth of the cluster, without saturating the cluster switch and limited by the 140 MB/s disk bandwidth; (c) Transfer density visualizes the ongoing transfers for each storage node for the duration of the workflow. Each row of the *y*-axis is a storage node in the cluster. The height of each tic in each row indicates the number of ongoing transfers for the storage node. The height of a storage node row is set to 10 concurrent transfers, so some tics exceed the height of a storage node row during heavy activity.

DOI: 10.1002/cpe

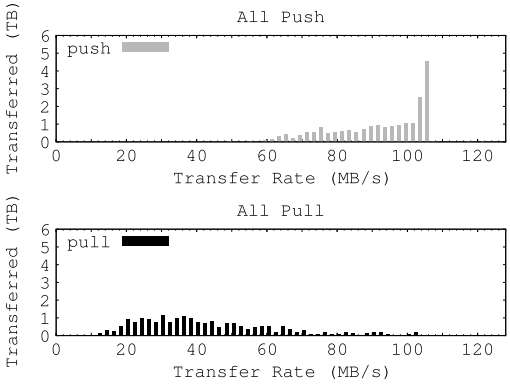
approximately 00:07, the replication finishes, and nodes 1 and 3 begin pushing to nodes 11 and 20, and so on. This distribution methodology minimizes storage node load and maximizes individual transfer bandwidth (‘All Push’ in (a)). On the other hand, using pull transfers cause all of the storage nodes to naively herd the single source storage node hosting the 32 GB file (‘All Pull’ in (c)) and thus suffer from low individual transfer bandwidth (‘All Pull’ in (a)).

Conclusion: Executing push transfers allow the scheduler to efficiently distribute large files while controlling load on the cluster. Storage nodes are able to transfer files using the full disk and network bandwidth. Ultimately, centralized management of transfers allows for an efficient tree distribution of files that maximizes transfer parallelism and minimizes contention. In this case, the file distribution using push transfers benefited from a 77% speedup.

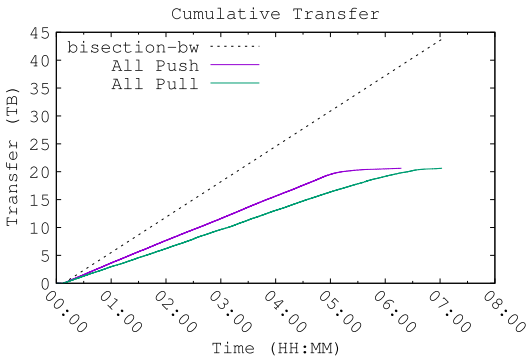
4.2. Concurrent distribution of multiple dependencies

Next, we test that load control from push transfers improves file distribution time even when there are more opportunities for transfer parallelism. We look at a workload where the cluster must fully distribute several large file dependencies across the entire cluster. Pull transfers in the previous workflow suffered because all of the jobs were pulling from the same replica simultaneously, allowing for no transfer parallelism. Here, we look at file distribution in the cluster when jobs pull from several large dependencies. We expect this to be significantly different from the previous Workflow A for two reasons: (1) the first push transfer for each of the several dependencies can be performed concurrently; and (2) the load on individual storage nodes by pull transfers is reduced because not all storage nodes are attempting to pull the same dependency simultaneously. (That is to say, the storage nodes are pulling dependencies in a random order. We look at pull ordering in Section 4.3.)

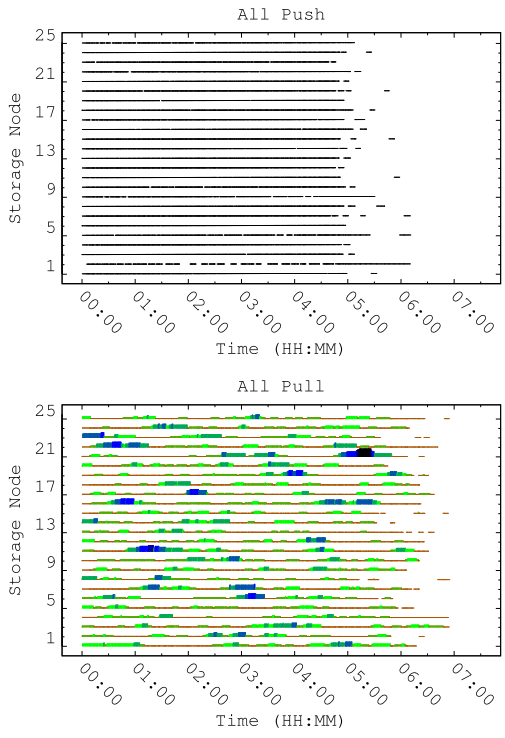
Workflow B expands on Workflow A by increasing to $25 \cdot 32$ GB shared files. This also requires all of the 32 GB files to be replicated across the entire cluster for each consumer job. Each file is



(a) Average Transfer Speed Histogram



(b) Cluster Cumulative Transfer



(c) Transfer Density (expl: see Figure 8)

Figure 9. **Workflow B:** multiple file concurrent spanning tree distribution.

DOI: 10.1002/cpe

replicated 24 times, so the cluster needs to transfer approximately 20 TB of data during the course of the workflow. Again, we run the workflow with two configurations: all push and all pull transfers.

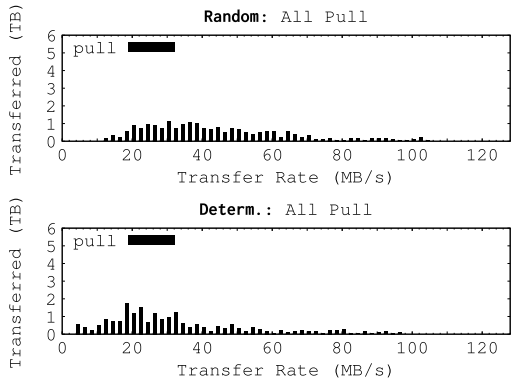
Figure 9 shows the results for Workflow B. The distribution time for push and pull is virtually the same. The spanning tree distribution used by push transfers has negligible impact because pulls also benefit from transfer parallelism via multiple dependencies. The aggregate cluster bandwidth for both configurations is constant except for a long tail (b). Pulls do marginally worse due to periods of storage node contention resulting in lower transfer bandwidth (a).

On the other hand, push transfers deliver consistently higher bandwidth compared with pulls by eliminating contention (a), but this does not lead to a significant improvement in distribution time. Because of the odd number of consumer jobs (25) and each storage node having a single transfer slot, only 12 transfer jobs can be scheduled at a time. This limit on push transfers leads to the long tail at the end of the distribution and several transfer gaps visible in 'All Push' in (c). Additionally, the spanning tree distributions for all of the dependencies do not progress in lock step. Because of random factors and opportunistic scheduling, some files will finish distribution much earlier in the workflow.

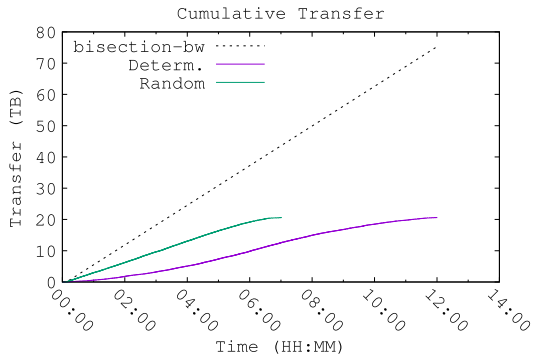
Conclusion: This workflow shows that random pulling of large dependencies across the cluster can achieve comparable performance with structured push transfers. Even so, the pull transfer bandwidth suffers in unpredictable ways. This makes it more difficult for the head node to predict transfer load on storage nodes. Additionally, the hot-spots on the cluster introduce more opportunities for transfer and job failures. Altogether, this makes pull transfers less attractive for distribution of large files.

4.3. Execution order of pull transfers

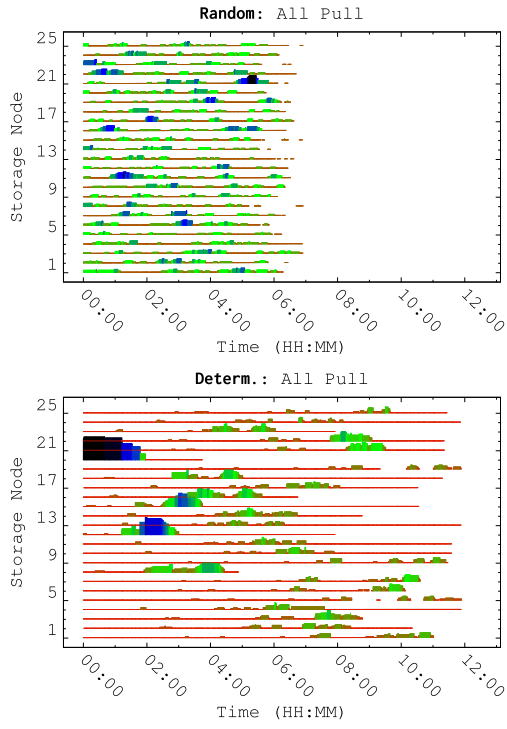
We will now examine how the execution order of pull transfers order can lead to unanticipated load, causing extreme transfer slow downs. This arises from a common situation in workflows where a group of jobs have one or more shared input file dependencies. These dependencies must be



(a) Average Transfer Speed Histogram



(b) Cluster Cumulative Transfer



(c) Transfer Density (expl: see Figure 8)

Figure 10. **Workflow B:** random versus deterministic pull transfer ordering.

DOI: 10.1002/cpe

distributed across the cluster as new replicas to support parallel execution of jobs. An unexpected problem is that the order of the pull transfers has a significant impact on reducing contention. If storage nodes perform pull transfers for common input files in the same order – frequently the case when executing a workflow on the cluster – then the storage node hosting the first dependency will suffer uncontrolled load.

To evaluate this, we use the same Workflow B used in Section 4.2 but with a deterministic pull ordering (the order used by the workflow manager specifying the jobs). So each consumer job will pull its dependencies in the same order. We are only interested in the behavior of the workflow and cluster for pull transfers. For comparison, we include the results of previous experiment which used randomly ordered pulls, with axis ranges adjusted if appropriate.

Figure 10 shows the results of the experiment. The transfer density of the cluster in (c) is the most telling figure of this experiment. It shows each storage node's transfer activity within the cluster across the duration of the workflow. For a deterministic pull ordering, some storage nodes (beginning with node 20) suffer debilitating transfer load because they host the replica first pulled by other jobs. In contrast, the random ordering has more uniform transfer load across the duration of the workflow with only a few relatively small hot-spots.

The deterministic ordering has the largest impact at the beginning because the first pulls are finished incrementally, not together. Once finished pulling the first dependency from node 20, jobs move on to pulling the next dependency from node 12. As some jobs get ahead of others in progress, there are fewer instances of extreme load on storage nodes. This is also indicated in the aggregate cluster bandwidth (derivative of cumulative) in (b) where there is a slower ramp up in the first 5 hours of the workflow.

Conclusion: When jobs have several dependencies that must be pulled, it is essential that these transfers occur in a random order to avoid hot-spots. Usually, the negative effects of hot-spots are only visible for larger files as pull transfers are more likely to interfere. We would expect push transfers to be preferred to avoid this problem entirely, but pull transfers may still be useful for larger files in certain circumstances.

4.4. Scaling pull threshold

Our next experiment will test whether a balance of push and pull transfers can be achieved, benefiting from load control of pushes and the increased parallelism of pulls. We use the pull threshold – the maximum file size for pull transfers – to achieve this balance. While push transfers allow Confuga to prevent debilitating load on storage nodes as shown in previous experiments, they come with costs. Setting up push transfers require several round-trip communications with the head node and incur the cost of setting up a transfer job. Push transfers also force a tree distribution when the effort may not be justified (e.g., for smaller files). On the other hand, pull transfers require minimal head node involvement because the head node includes with the job a list of potential storage nodes to pull from for each file. In short, pushes control load on the cluster at the cost of increased overhead and work for the scheduler, while pulls decrease work on the scheduler at the cost of potential hot-spots.

Workflow C evaluates varying the pull threshold such that the work moved from push transfers to pull transfers linearly increases as the pull threshold doubles. This workflow defines shared dependencies $64 \cdot 1$ GB, $32 \cdot 2$ GB, ..., $2 \cdot 32$ GB, $1 \cdot 64$ GB. So each job requires 448 GB of data. Each file is replicated 24 times (25 replicas for 25 jobs), resulting in 10.5 TB transferred over the course of the workflow. The pull threshold is scaled from 0 (all push transfers) to 64 GB (all pull transfers). As we double the pull threshold, the amount of data moved by pull transfers increases by a constant 64 GB per job (1.6 TB for all jobs), but the number of push transfers is halved.

Figures 11 and 12 show the results of this experiment. In general, increasing the pull threshold causes the cluster to be more loaded with concurrent transfers. As more pulls replace pushes, the time span of managed load arising from push transfers (1 transfer per node) overlaps with the uncontrolled load of pulls (greater than 1 transfer per node). Despite the increased load, using pulls for the smaller files in the workflow lead to significant distribution time improvements despite reduced transfer bandwidth.

DOI: 10.1002/cpe

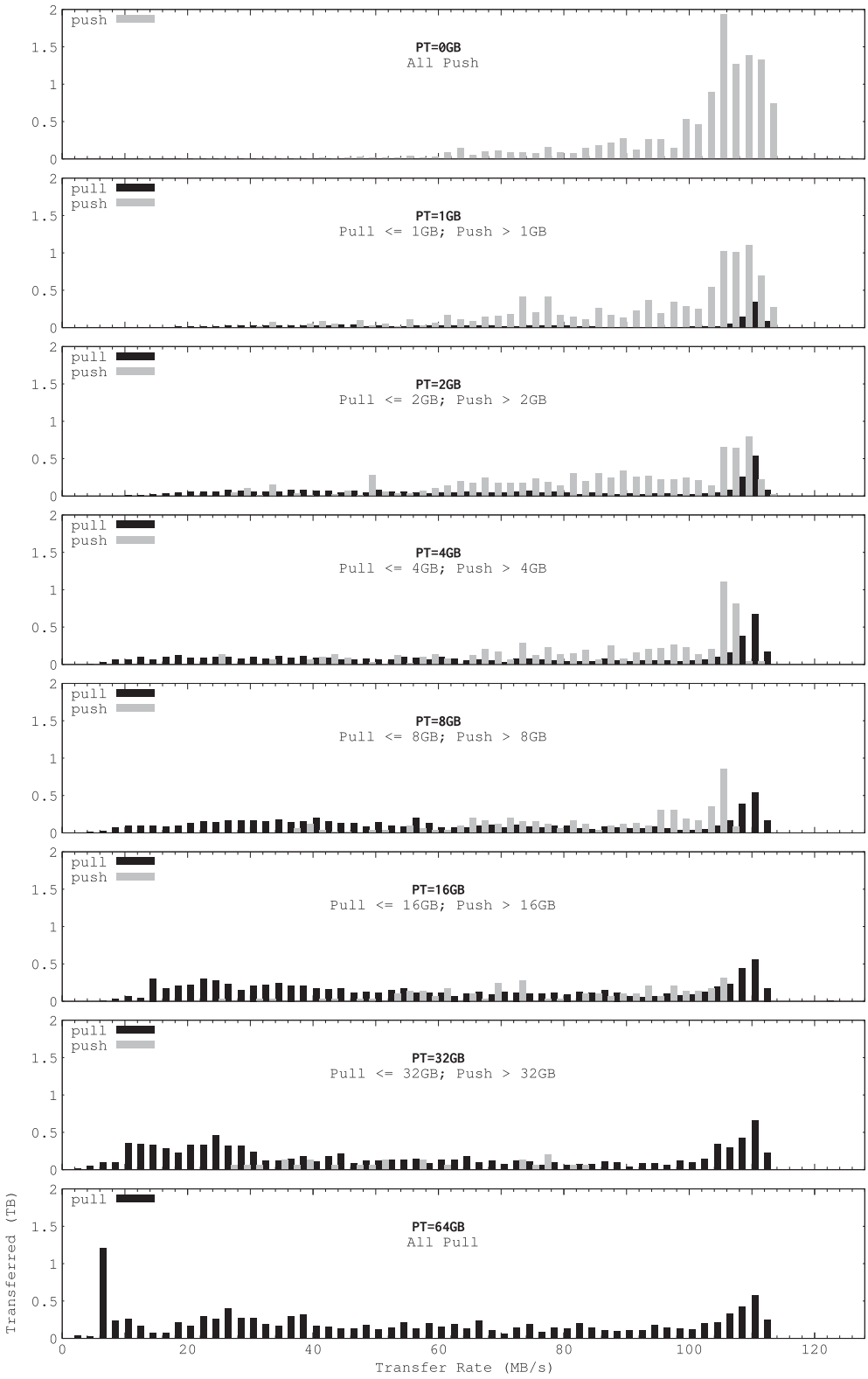


Figure 11. **Workflow C:** average transfer speed histogram.

DOI: 10.1002/cpe

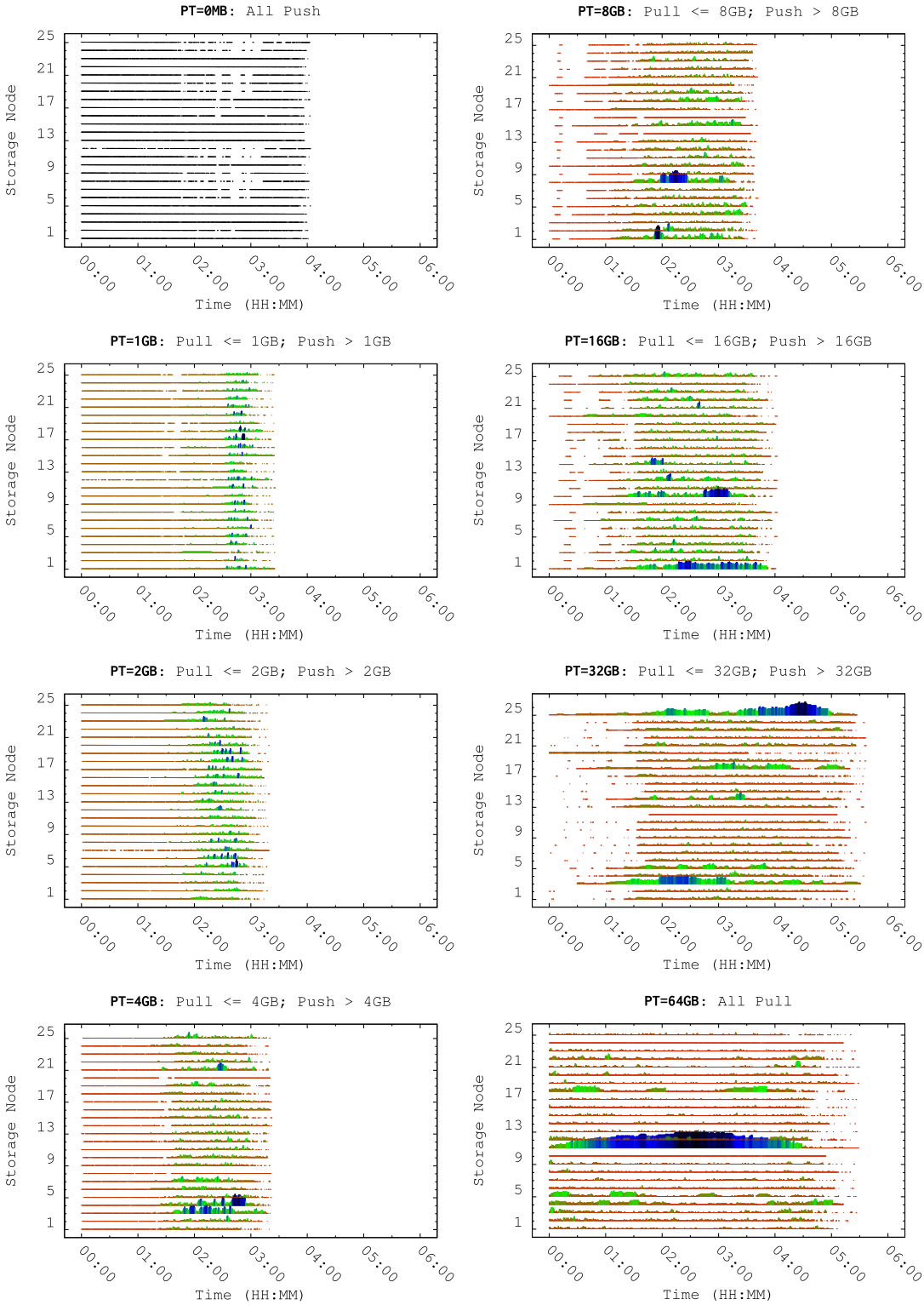


Figure 12. **Workflow C:** transfer density (expl: see Figure 8).

DOI: 10.1002/cpe

Figure 11 shows that the introduction of pull transfers can reduce the effectiveness of push transfers. As the pull threshold increases, the push transfers begin to have similar performance to pulls. This is caused by contention for the storage nodes' network and disk bandwidth. Even so, with a heavily loaded cluster, the push transfers can have a positive effect on the distribution time. The controlled distribution of the larger files benefits from the parallelism of the tree distribution as well as fewer disk buffer cache misses because there is only ever a single transfer reading the file sequentially.

Additionally, the time taken by the head node to manage smaller transfers may take longer than the time to pull the files from a single source. For `PT=1GB`, in Figure 11, the bandwidth for several individual pull transfers suffers because of contention and inefficient distribution of the files. Even so, using pulls for the 1 GB files avoid stalls caused by slow pushes and allow parallel distribution of the smaller files alongside of the pushes. Consequently, the distribution time for `PT=1GB` gives a 12% improvement over only push transfers.

Figure 12 shows the transfers executing in the cluster, visualizing hot-spots. These graphs also help show the two transfer modes (push and pull) used to replicate job dependencies. Generally, the beginning of the pull transfers is indicated by storage nodes having increased concurrent transfer activity as jobs begin simultaneously pulling the replicas from the same storage node. **Note that jobs do not begin pulls in lockstep.** While most jobs begin pulls at roughly the same time, some jobs may start pulls much earlier once the scheduler has finished push transfers for the dependencies larger than the pull threshold. This is especially noticeable for `PT=32GB` because the node hosting the 64 GB dependency – the only dependency which will be pushed – can start its job without delay. The first consumer job is immediately scheduled on and dispatched to storage node 20 hosting the 64 GB file. That job then begins by pulling all of its missing dependencies which causes the short transfer activity visible on the other storage nodes (the small tics from 00:00 to 00:30). After the second consumer job is scheduled to node 25, a push transfer is scheduled on storage node 20 which replicates the 64 GB dependency to storage node 25. In effect, this transfer job is replicating the 64 GB file to node 25, while the first consumer job is pulling its dependencies. The first 64 GB push transfer finishes at 00:30 and the last at 02:21. The last 3 hours of the workflow is occupied by pulls.

As the pull threshold is increased, the head node has fewer files it needs to push. This will cause the cluster to be underutilized because a few large pushes delay all other transfers. This is indicated in the transfer gaps visible at the beginning of the workflows for `PT=8GB`, `PT=16GB`, and `PT=32GB`. At these thresholds, there is an increasingly smaller pool of files to push. For `PT=8GB`, there are seven files which must be pushed: $4 \cdot 16$, $2 \cdot 32$, and $1 \cdot 64$ GB.

Supporting multiple large concurrent transfers also puts pressure on storage node virtual memory. The disk buffer cache is unable to support the simultaneous access forcing the kernel to drop pages

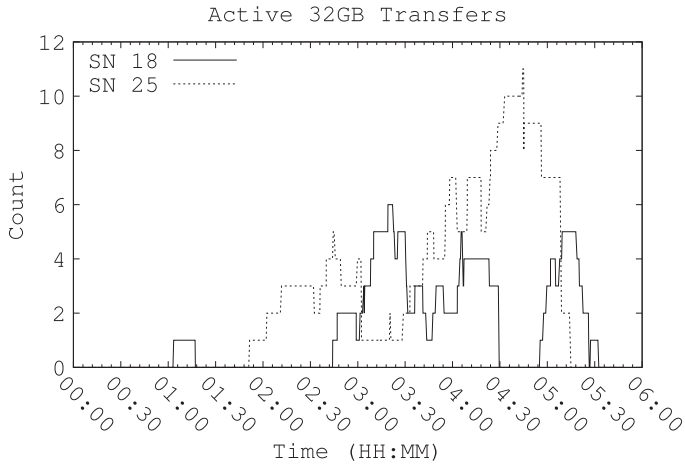


Figure 13. **Workflow C:** `PT=32 GB` active 32GB pull transfers. This graph shows the groups of pull transfers that are delayed together by disk seeks, resulting in periodic sharp drops in active transfers.

DOI: 10.1002/cpe

and perform disk seeks. For our systems with 32GB of RAM, this is especially noticeable when the pull threshold is increased from `PT=16GB` to `PT=32GB`. The distribution time increases by 40%. During our analysis, we have observed that pull transfers form groups riding the buffer cache as pages load from disk. This is indicated in Figure 13 which shows the two nodes (18 and 25) hosting the 32 GB files which are pulled for `PT=32GB`. There are several groups of pull transfers for both storage nodes which finish together as the last pages of the 32 GB file are read. For example, storage node 25 had five pull transfers complete in the span of a second at approximately 05:15.

When there are several transfers with reads satisfied by the kernel's buffer cache, we would expect a proportional sharing of network bandwidth. This would allow transfers to progress independently but slowly. However, when the buffer cache cannot satisfy all of the files being transferred, we observe significant slowdowns. This suggests the buffer cache should be taken into account by the head node when planning transfers. Generally, the head node should manage this by using one-at-a-time push transfers which benefit from sequential reads or by limiting the net file sizes of concurrent transfers.

Conclusion: This workflow shows that while push transfers allow for fast distribution of large files through structured and high bandwidth transfers, there is room to tolerate some interference and hot-spots from pull transfers for small files. Push transfers for smaller files do not give a justifiable improvement to distribution time when there is pressure to transfer other larger files. Instead, using pull transfers introduces small amounts of interference, but individual pull and push transfer bandwidth is mostly unaffected.

5. CASE STUDY: BIOINFORMATICS

We have evaluated the performance benefits of balanced push and pull transfers in the context of two bioinformatics workflows: Burrows–Wheeler Aligner alignment (BWA) [34] and iterative alignments of long reads [35], which we shorten to IALR in this section.

The BWA workflow aligns a number of smaller fragments, or reads, to a reference genome. It is composed of 826 jobs, beginning with a 274 way split of two query read databases each 3.1 GB in size. A 265 MB reference database is also shared by all jobs. The biological purpose of this specific alignment workflow was to uncover differences in sequenced individual mosquitoes such as single nucleotide polymorphisms relative to reference genomes. Full details are in [36]. The workflow is visualized in Figure 14.

The IALR workflow is a simulation of a new method that iteratively compares PacBio reads to improve a target genome using locality sensitive hashing for fast updates. The workflow is composed of 26 jobs and, like BWA, begins by splitting a genome database of 255 MB size into 25 parts. This workflow also has a shared 38 GB database of PacBio reads which is required by all 25 jobs performing comparisons because reads that do not align at the start might at completion once two reference sequences are joined and/or updated during execution. The workflow is visualized in Figure 15.

Table I shows the results of running both experiments while varying the pull threshold. The third column 'Time' is the workflow execution time including any transfers. The fourth column 'Total Transfer Time' is the cumulative time in seconds for all transfers (push or pull) not including setup overhead or latency.

Table I. Bioinformatics workflows.

Experiment	Transfer method	Time (hh:mm:ss)	Total transfer time (s)	Pushes	Pulls
IALR	Push All	01:52:00	8,891.75	97	0
	Auto; $PT = 256$ MB	01:52:13	8,976.17	24	74
	Pull All	03:38:16	163,802.00	0	96
BWA	Push All	03:33:05	364.34	1,946	0
	Auto; $PT = 256$ MB	00:49:04	44,713.10	25	5,991
	Pull All	01:11:19	77,975.30	0	6,244

BWA, Burrows–Wheeler Aligner alignment.

DOI: 10.1002/cpe

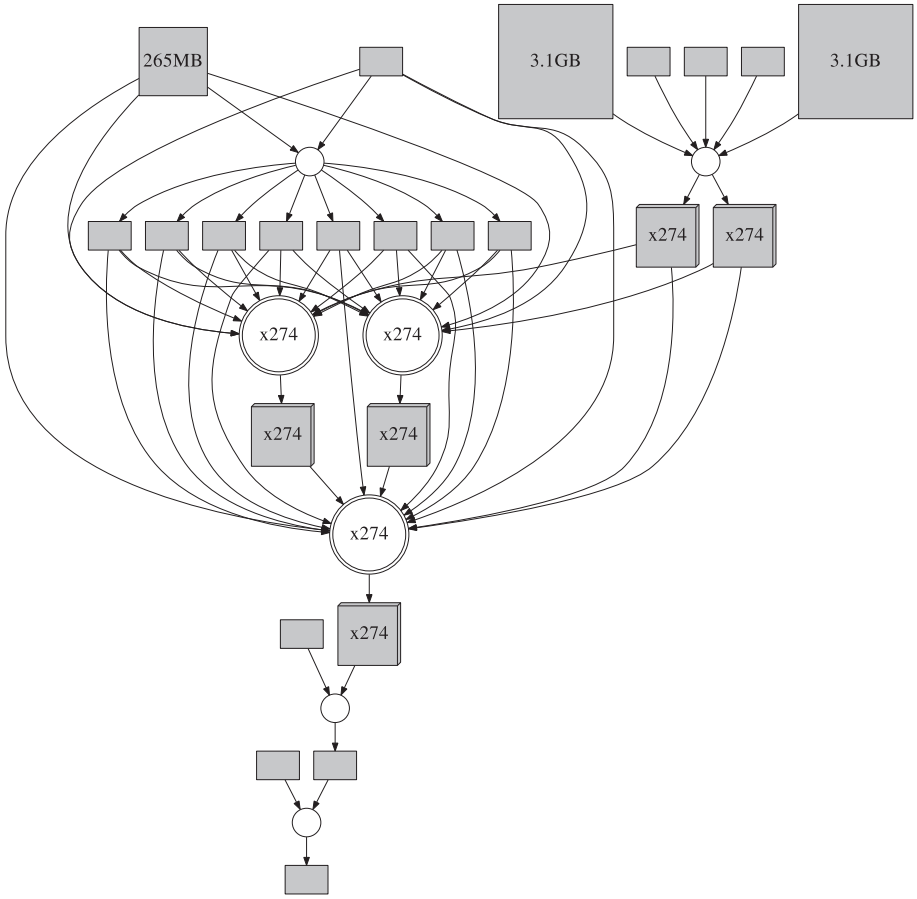


Figure 14. **BWA Workflow** Jobs are circles, and files are boxes. See text for workflow description.

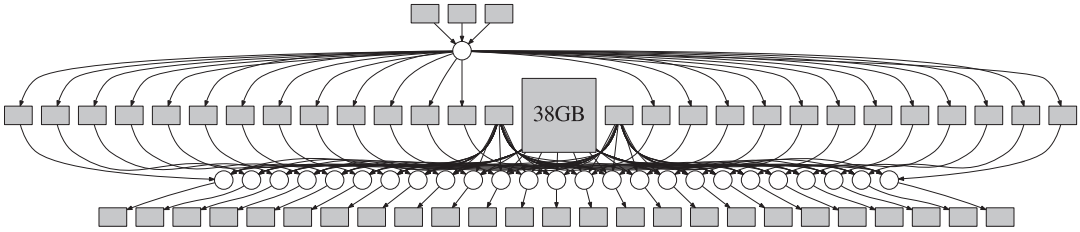


Figure 15. **IALR Workflow** Jobs are circles, and files are boxes. See text for workflow description.

As expected, the IALR workflow benefits greatly from centrally managed push transfers for the 32 GB dependency. This enabled a spanning tree distribution across all the nodes and eliminates node instability. The net result is a reduction in the workflow time-to-complete by 48% from all pulls and an order of magnitude reduction in time spent doing transfers. Because the other transfers have such a small impact compared with the 32 GB dependency, there is no significant difference between 256 MB pull threshold and all pushes.

The BWA workflow processes several hundred multi-megabyte files that must be transferred. The majority of these files are produced from splits of the two 3.1 GB queries by the first job. In this situation, using only push transfers lead to significant slowdowns as push transfers are executed serially on the storage node which performed the split (due to its single transfer slot). So while the all push configuration minimized the number of transfers (1964 pushes) and the total time executing transfers (364.34 s), the serial execution of push transfers and overheads introduced by centralized management severely impacts performance. On the other hand, the use of pulls resulted in a significant

DOI: 10.1002/cpe

improvement in workflow time, despite pulls being less efficient (77,975 s spent doing transfers vs. 364 s). Limited use of push transfers on larger shared inputs results in a performance improvement of 31% over only pulls and 77% over only pushes. The 256 MB pull threshold allows the efficient distribution of the 265 MB shared reference genome while freeing up resources up on the storage node which split the queries.

Conclusion: These two bioinformatics workflows show that a balanced use of push and pull transfers reduces time to distribute dependencies for real workloads. Indeed, the two workflows experienced different worst-case behavior depending on the transfer method. However, a hybrid approach to managing transfers in the cluster achieves as-good or better performance than only using a single methodology while eliminating common load instability.

6. RELATED WORK

Originally, active storage began as smart disks [19] and grew within the HPC community to smart object stores which can harness unused CPU to perform simple functions on data [20] with the goal of increasing I/O throughput and reducing data movement [21]. More recently, projects like Hadoop were developed for clusters built on commodity hardware that are dedicated to performing structured computation [1, 2, 37] on large datasets. Confuga is a natural evolution of this approach whereby users can execute whole applications with multiple dependencies and full data locality but do not need to modify their workflow to fit fixed computation frameworks like MapReduce [1]. Clients need only specify the complete list of dependencies for each job and execute jobs conforming to workflow consistency semantics (i.e., no run-time data dependencies).

Object-based storage devices (OSDs) [38] provide some support for using the processing power of disks. An OSD provides an abstracted storage container which produces objects tagged with metadata. A set of operations for manipulating and searching/selecting the objects on the devices is part of the standard [39]. OSD has become an integral component of numerous file systems including Lustre [40] and Panasas [15]. Computation on Lustre storage servers has also been supported [41] to allow client programs to have direct access to data, and in [20] as a user-space solution. Confuga uses the concepts of direct client access to OSDs in the design of storage nodes, where each job operates within a defined sandbox with access to each replica. The head node does not need to be contacted for access to files within this sandbox.

The Cplant [42] project is notable for managing distribution of certain shared dependencies (like an executable) across many nodes for parallel job launch. This is carried out through a spanning tree managed by the client's workflow manager. Confuga's use of push transfers is an evolution of this idea but differs in significant ways: the file system is able to place jobs on nodes which already have many or all of the job's dependencies which avoids redundant effort; replication is globally controlled across the cluster with awareness of all running workflows; replication can be to a subset of nodes/jobs requiring a dependency rather than all nodes; and, Confuga can push multiple dependencies in parallel without potential long tails caused by slow nodes.

Data diffusion [43] presents a technique for improving data locality by allowing data to be cached across all compute resources. In this model, all resources act as a cooperative cache [44] (a technique where clients use others' caches to distribute load). The task dispatch framework used, Falkon, is aware of the cache state on resources and is capable of scheduling tasks near data. Shark [45] is another distributed file system which allows clients to cooperatively cache data to improve scalability.

Workflow managers that operate within grids adopt a limited role for data locality. The usual problem is harnessing execution nodes and delivering data to geographically distinct sites. Pegasus [11] and GridBLAST [46] addressed this by operating in tandem with Globus [47] to deploy off-site dependencies to the local staging site or shared file system. Pegasus also schedules jobs at compute nodes hosting data otherwise schedules randomly. The Stork [48] data scheduler is designed to cooperate with the DAGMan [9] workflow manager to manage data placement. Stork acts as a transfer job manager between distinct sites on the grid and handles fault-tolerance and reliability of transfers. Job data may be stored on numerous data nodes [31], which are discovered and accessed via Parrot [26], a user-level virtual file I/O agent. While data placement between sites on the grid

DOI: 10.1002/cpe

has been well studied, Confuga manages transfers between active storage nodes within a cluster. Confuga also does not require the user to include data placement requests within the workflow description. Like Stork, Confuga uses the concept of transfer jobs to achieve reliable and fault-tolerant transfers between nodes.

Confuga focuses its attention on controlling transfer load once scheduling decisions have already been made but it uses concepts which appear in distributed and centralized schedulers. Partitioning storage nodes into *map* and *reduce* slots in Hadoop [49] is used to increase utilization of system resources (memory and network resources especially). This allows multiple map tasks to run concurrently and multiple long-running reduce tasks to accept input as map output is produced. Similarly, Hawk [50] uses cluster partitioning with dedicated short job servers. In Confuga, we use transfer slots to express the demand placed on the networking and disk. In previous work [18], we observed optimal push transfer performance when limiting storage nodes to one transfer slot under large transfer load.

Hawk also implements a hybrid scheduler design with a centralized scheduler for scheduling long-running *large* jobs and distributed schedulers for scheduling *short* jobs. The centralized scheduler improves performance by knowing the distribution of large jobs across the cluster and the associated wait time. The centralized scheduler is able to function under load because there are fewer large job than short jobs. Confuga also uses a hybrid design for transfers: *push* transfers are directed by the central scheduler, while *pull* transfers are initiated in a distributed fashion by the individual storage nodes.

Abstractions which manage data distribution to compute nodes have been used with some success in scientific workflows. Instead of just placing tasks near its inputs and coordinating the data transformations, the abstraction also conducts transfers as needed. The All-Pairs [7] abstraction is notable for using this approach: split a large dataset and compare every pair of splits. All-Pairs will manage an efficient spanning tree distribution of the dataset to support parallel comparisons. This approach has been used in biometrics research [51] to perform comparison functions across every pair of subjects.

Significant efforts have been made to explore namespaces as a solution to harnessing multiple machines as a single system. This first began with LOCUS [28] which presented a common system namespace for distributed processes. Plan9 took this a step further by abstracting many resources normally part of the process and making them part of the *namespace* of the process [52]. This allowed Plan9 to abstract several details such as the CPU architecture a binary was compiled for. Declaratively, expressing the resources needed to the system provide opportunities for optimization. In the same way, Confuga uses the namespace mapping provided by jobs to optimize and manages transfers within the cluster.

Early distributed file systems such as NFS [53] and AFS [54] followed POSIX consistency semantics for client access. AFS is notable for resolving certain performance issues by relaxing consistency semantics using `write-on-close`. More recently, the Ceph [55] cluster file system was designed to address metadata scalability by decoupling file system metadata from the file content or replicas. It does this using a metadata server cluster with dynamic subtree partitioning to balance load. Ceph also alleviates metadata server load using their CRUSH [56] algorithm which allows storage nodes to autonomously lookup replicas. Still, cluster file systems continue to suffer from a metadata bottleneck [57, 58]. Confuga avoids most metadata issues by designing the system for workflow semantics, where file access is known at dispatch and visibility of changes are only committed on task completion. This allows Confuga to batch many operations (`open`, `close`, `stat`, and `readdir`) at task dispatch and completion and opportunistically prohibits dynamic file system access.

7. CONCLUSIONS AND FUTURE WORK

This paper explored the performance impacts of two data distribution mechanisms, push and pull, used for replicating data dependencies in scientific workflows. We have shown that controlled distribution through push transfers can eliminate typical load instability caused by large common

DOI: 10.1002/cpe

dependencies of workflow jobs. Despite the success of push transfers, delegating transfer management to the storage node through pull transfers still has application. While push transfers allow for fast distribution of large files through structured and high bandwidth transfers, there is room to tolerate some interference and hot-spots from pull transfers for small files. Push transfers for smaller files do not give a justifiable improvement to distribution time when there is pressure to transfer other larger files. Instead, using pull transfers introduce small amounts of interference, but individual pull and push transfer bandwidth is mostly unaffected. Ultimately, a balance of the two approaches achieves optimal file distribution. This is exhibited in two bioinformatics workflows where a careful balance of the two mechanisms leads to 48% and 77% reductions over only push or pull.

The design of Confuga indicates that there is much to be gained by leveraging knowledge of data needs in scientific workflows. The cluster is able to avoid common load instability arising from 'hot' data by replicating files as needed. Confuga shares the perspective of other big data systems like Hadoop that effective data-locality is achieved by joining the batch system and the file system: the file system is able to respond to data demands through replication and place jobs according to data-locality.

Future work might explore how to parallelize the push and pull transfers for a job. Currently, a job performs pull transfer prior to `exec` of the job's executable. It may be advantageous to schedule a separate transfer job which performs the pulls so that it may execute in parallel with pushes. Additionally, executing the pull transfers in a separate job allow the new replicas to become visible earlier and usable for new transfers. We believe this mechanism would work best for smaller files which can cheaply fly under the radar of larger transfers.

ACKNOWLEDGEMENTS

This work was supported in part by National Science Foundation grant OCI-1148330. We are grateful to Dr Scott Emrich for his help designing the IALR workflow for our experimentation with Confuga and his suggestions for improvement.

Confuga's source code is distributed in the Cooperative Computing Toolset (CCTools) under the GNU General Public License. See the following URL for source code and workflows used in this paper: http://ccl.cse.nd.edu/ and https://github.com/cooperative-computing-lab/cctools/tree/papers/confuga-ccpe

REFERENCES

1. Jeffrey D, Sanjay G. MapReduce: simplified data processing on large clusters. *Communications of the ACM* January 2008; **51**(1):107–113.
2. Matei Z, Mosharaf C, Franklin MJ, Scott S, Ion S. Spark: cluster computing with working sets. *Proceedings of the 2nd Usenix Conference on Hot Topics in Cloud Computing*, 2010; 10–10.
3. Matsunaga A, Tsugawa M, Fortes J. CloudBLAST: combining MapReduce and virtualization on distributed resources for bioinformatics applications. *ESCIENCE, 2008. ESCIENCE '08. IEEE Fourth International Conference on*, 2008; 222–229.
4. Heshan L, Xiaosong M, Feng W, Samatova NF. Coordinating computation and I/O in massively parallel sequence search. *IEEE Transactions on Parallel and Distributed Systems* 2011; **22**(4):529–543.
5. Shvachko K, Kuang H, Radia S, Chansler R. The Hadoop distributed file system. *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010; 1–10.
6. Yu L, Christopher M, Scott E, Kenneth J, Thain D. Harnessing parallelism in multicore clusters with the All-Pairs sand wavefront abstractions. *Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing*, HPDC '09, ACM: New York, NY, USA, 2009; 1–10. DOI: 10.1145/1551609.1551613.
7. Moretti C, Bui H, Hollingsworth K, Rich B, Flynn P, Thain D. All-Pairs: an abstraction for data-intensive computing on campus grids. *IEEE Transactions on Parallel and Distributed Systems* 2010; **21**(1):33–46.
8. Bui P, Yu L, Thrasher A, Carmichael R, Lanc I, Donnelly P, Thain D. Scripting distributed scientific workflows using Weaver. *Concurrency and Computation: Practice and Experience* 2012; **24**(15):1685–1707.
9. Couvares P, Kosar T, Roy A, Weber J, Wenger K. Workflow management in Condor. In *Workflows for E-Science*, Taylor IanJ, Deelman Ewa, Gannon DennisB, Shields Matthew (eds). Springer: London, 2007; 357–375. DOI: 10.1007/978-1-84628-757-2_22.
10. Albrecht M, Donnelly P, Bui P, Thain D. Makeflow: a portable abstraction for data intensive computing on clusters, clouds, and grids. *1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, SWEET '12, ACM: New York, NY, USA, 2012; 1:1–1:13. DOI: 10.1145/2443416.2443417.

11. Deelman E, Singh G, Su MH, Blythe J, Gil Y, Kesselman C, Mehta G, Vahi K, Berriman GB, Good J, Laity A, Jacob JC, Katz DS. Pegasus: a framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming* 2005; **13**(3):219–237.
12. Wilde M, Hategan M, Wozniak JM, Clifford B, Daniel SK, Foster I. Swift: a language for distributed parallel scripting. *Parallel Computing* 2011; **37**(9):633–652.
13. Litzkow MJ, Livny M, Mutka MW. Condor – a hunter of idle workstations. *8th International Conference on Distributed Computing Systems*, Piscataway, NJ, 1988; 104–111.
14. Gentzsch W. Sun grid engine: towards creating a compute power grid. *Proceedings of the 1st IEEE/ACM International Symposium on Cluster Computing and the Grid*, Piscataway, NJ, 2001; 35–36.
15. Nagle D, Serenyi D, Matthews A. The Panasas ActiveScale storage cluster: delivering scalable high bandwidth storage. *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, SC '04, IEEE Computer Society, Washington, DC, USA, 2004; 53–53. DOI: 10.1109/SC.2004.57.
16. Woodard A, Wolf M, Mueller C, Valls N, Tovar B, Donnelly P, Ivie P, Anampa KH, Brenner P, Thain D, Lannon K, Hildreth M. Scaling data intensive physics applications to 10k cores on non-dedicated clusters with Lobster. *IEEE International Conference on Cluster Computing (Cluster)*, Piscataway, NJ, 2015; 322–331.
17. Donnelly P, Bui P, Thain D. Attaching cloud storage to a campus grid using Parrot, Chirp, and Hadoop. *IEEE 2nd International Conference on Cloud Computing Technology and Science (Cloudcom)*, Piscataway, NJ, 2010; 488–495.
18. Donnelly P, Hazekamp N, Thain D. Confuga: scalable data intensive computing for POSIX workflows. *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, Piscataway, NJ, 2015; 392–401.
19. Riedel E, Gibson GA, Faloutsos C. Active storage for large-scale data mining and multimedia. *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998; 62–73. (Available from: http://dl.acm.org/citation.cfm?id=645924.671345).
20. Piernas J, Nieplocha J, Felix EJ. Evaluation of active storage strategies for the Lustre parallel file system. *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, ACM, New York, NY, USA, 2007; 28:1–28:10. DOI: 10.1145/1362622.1362660.
21. Seung WS, Lang S, Carns P, Ross R, Thakur R, Ozisikyilmaz B, Kumar P, Liao WK, Choudhary A. Enabling active storage on parallel I/O software stacks. *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, Piscataway, NJ, 2010; 1–12.
22. Donnelly P, Thain D. Design of an active storage cluster file system for DAG workflows. *Proceedings of the 2013 International Workshop on Data-intensive Scalable Computing Systems*, DISCS-2013, ACM, New York, NY, USA, 2013; 37–42. DOI: 10.1145/2534645.2534656.
23. Ghemawat S, Gobioff H, Leung ST. The Google file system. *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, ACM, New York, NY, USA, 2003; 29–43. DOI: 10.1145/945445.945450.
24. Donnelly P, Thain D. Fine-grained access control in the Chirp distributed file system. *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, Piscataway, NJ, 2012; 33–40.
25. Thain D, Moretti C, Hemmes J. Chirp: a practical global filesystem for cluster and grid computing. *Journal of Grid Computing* 2009; **7**(1):51–72. DOI: 10.1007/s10723-008-9100-5.
26. Thain D, Livny M. Parrot: transparent user-level middleware for data-intensive computing. *Scalable Computing: Practice and Experience* 2005; **6**(3):9–18.
27. Szeredi M. FUSE: filesystem in userspace, 2005.
28. Walker B, Popek G, English R, Kline C, Thiel G. The LOCUS distributed operating system. *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, SOSP '83, ACM, New York, NY, USA, 1983; 49–70. DOI: 10.1145/800217.806615.
29. Rumble SM, Lacroute P, Dalca AV, Fiume M, Sidow A, Brudno M. SHRiMP: accurate mapping of short color-space reads. *PLoS Computational Biology* 2009; **5**(5):e1000386.
30. Crockford D. The application/JSON Media Type for JavaScript object notation (JSON), 2006.
31. Thain D, Tannenbaum T, Livny M. Distributed computing in practice: the Condor experience. *Concurrency and Computation: Practice and Experience* 2005; **17**(2-4):323–356. DOI: 10.1002/cpe.938.
32. Bui P, Rajan D, Abdul-Wahid B, Izaguirre J, Thain D. Work Queue + Python: a framework for scalable scientific ensemble Applications. *Workshop on Python for High Performance and Scientific Computing at SC11*, Seattle, WA, 2011.
33. Berners-Lee T, Masinter L, McCahill M. Uniform resource locators (URL), 1994.
34. Li H, Durbin R. Fast and accurate short read alignment with Burrows–Wheeler transform. *Bioinformatics* 2009; **25**(14):1754–1760.
35. Steele A, Emrich SJ. pbSandwich: scaffolding draft genomes with long reads. *7th International Conference on Bioinformatics and Computational Biology (BiCob)*, International Society for Computers and their Applications: Cary, NC, 2015; 155–160.
36. Fontaine MC, Pease JB, Steele A, Waterhouse RM, Neafsey DE, Sharakhov IV, Jiang X, Hall AB, Catteruccia F, Kakani E, Mitchell SN, Wu YC, Smith HA, Love RR, Lawniczak MK, Slotman MA, Emrich SJ, Hahn MW, Besansky NJ. Extensive introgression in a malaria vector species complex revealed by phylogenomics. *Science* 2015; **347**(6217). DOI: 10.1126/science.1258524.
37. Malewicz G, Austern MH, Bik AJC, Dehnert JC, Horn I, Leiser N, Czajkowski G. Pregel: a system for large-scale graph processing. *Proceedings of the 2010 ACM Sigmod International Conference on Management of Data*, SIGMOD '10, ACM: New York, NY, USA, 2010; 135–146. DOI: 10.1145/1807167.1807184.

38. Gibson GA, Nagle DF, Amiri K, Chang FW, Feinberg EM, Gobioff H, Lee C, Ozceri B, Riedel E, Rochberg D, Zelenka J. File server scaling with network-attached secure disks. *Proceedings of the 1997 ACMSIGMETRICS International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '97*, ACM: New York, NY, USA, 1997; 272–284.
39. Weber RO. Information technology-SCSI object-based storage device commands (OSD). *Technical Council Proposal Document T* 2004; **10**:92.
40. Braam PJ, Zahir R. Lustre: a scalable, high performance file system. *Cluster File Systems, Inc* 2002.
41. Felix EJ, Fox K, Regimbal K, Nieplocha J. Active storage processing in a parallel file system. *Proceedings of the 6th LCI International Conference on Linux Clusters: The HPC Revolution*, Chapel Hill, NC, 2006.
42. Brightwell R, Fisk LA. Scalable parallel application launch on Cplant. *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, SC '01, ACM, New York, NY, USA, 2001; 40–40. DOI: 10.1145/582034.582074.
43. Raicu I, Zhao Y, Foster IT, Szalay A. Accelerating large-scale data exploration through data diffusion. *Proceedings of the 2008 International Workshop on Data-aware Distributed Computing*, DADC '08, ACM New York, NY, USA, 2008; 9–18. DOI: 10.1145/1383519.1383521.
44. Dahlin MD, Wang RY, Anderson TE, Patterson DA. Cooperative caching: using remote client memory to improve file system performance. *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*: USENIX Association, 1994; 19.
45. Annapureddy S, Freedman MJ, Mazières D. Shark: scaling file servers via cooperative caching. *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation – Volume 2*, NSDI'05, USENIX Association, Berkeley, CA, USA, 2005; 129–142.
46. Krishnan A. GridBLAST: a Globus-based high-throughput implementation of BLAST in a Grid computing framework. *Concurrency and Computation: Practice and Experience* 2005; **17**(13):1607–1623. DOI: 10.1002/cpe.906.
47. Bill A, Joe B, John B, Chervenak AL, Foster I, Kesselman C, Meder S, Nefedova V, Quesnel D, Tuecke S. Data management and transfer in high-performance computational grid environments. *Parallel Computing* 2002; **28**(5): 749–771.
48. Kosar T, Livny M. Stork: making data placement a first class citizen in the grid. *Proceedings 24th International Conference on Distributed Computing Systems*, Piscataway, NJ, 2004; 342–349.
49. White T. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc.: Sebastopol, CA, 2012.
50. Delgado P, Dinu F, Kermarrec AM, Zwaenepoel W. Hawk: hybrid datacenter scheduling. *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, USENIX Association, Santa Clara, CA, 2015; 499–510.
51. Bui H, Wright D, Helm C, Witty R, Flynn P, Thain D. Towards long term data quality in a large scale biometrics experiment. *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, ACM New York, NY, USA, 2010; 565–572. DOI: 10.1145/1851476.1851559.
52. Pike R, Presotto D, Thompson K, Trickey H, Winterbottom P. The use of name spaces in Plan 9. *Proceedings of the 5th Workshop on ACM Sigops European Workshop: Models and Paradigms for Distributed Systems Structuring*, EW 5, ACM: New York, NY, USA, 1992; 1–5. DOI: 10.1145/506378.506413.
53. Sandberg R, Goldberg D, Kleiman S, Walsh D, Lyon B. Design and implementation of the Sun network filesystem. *Proceedings of the Summer USENIX Conference*, Berkeley, CA, 1985; 119–130.
54. Howard JH, Kazar ML, Menees SG, Nichols DA, Satyanarayanan M, Sidebotham RN, West MJ. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems* February 1988; **6**(1):51–81.
55. Weil SA, Brandt SA, Miller EL, Long DDE, Maltzahn Carlos. Ceph: a scalable, high-performance distributed file system. *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, USENIX Association: Berkeley, CA, USA, 2006; 307–320.
56. Weil SA, Brandt SA, Miller EL, Maltzahn C. CRUSH: controlled, scalable, decentralized placement of replicated data. *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, ACM: New York, NY, USA, 2006. DOI: 10.1145/1188455.1188582.
57. Alam SR, El-Harake HN, Howard K, Stringfellow N, Verzelloni F. Parallel I/O and the metadata wall. *Proceedings of the Sixth Workshop on Parallel Data Storage*: ACM: 2011; 13–18.
58. Carns P, Lang S, Ross R, Vilayannur M, Kunkel J, Ludwig T. Small-file access in parallel file systems. *IEEE International Symposium on Parallel Distributed Processing*, IPDPS 2009, Piscataway, NJ, 2009; 1–11. DOI: 10.1109/IPDPS.2009.5161029.