Parallelizing the Execution of Sequential Scripts

Zhao Zhang
Department of Computer Science
University of Chicago
zhaozhang@uchicago.edu

Daniel S. Katz
Computation Institute
University of Chicago &
Argonne National Laboratory
d.katz@ieee.org

Timothy G. Armstrong
Department of Computer Science
University of Chicago
tga@uchicago.edu

Justin M. Wozniak
Mathematics and Computer Science Division
Argonne National Laboratory
wozniak@mcs.anl.gov

Ian Foster
Computation Institute
University of Chicago &
Argonne National Laboratory
foster@anl.gov

ABSTRACT

Scripting is often used in science to create applications via the composition of existing programs. Parallel scripting systems allow the creation of such applications, but each system introduces the need to adopt a somewhat specialized programming model. We present an alternative scripting approach, AMFS Shell, that lets programmers express parallel scripting applications via minor extensions to existing sequential scripting languages, such as Bash, and then execute them in-memory on large-scale computers. We define a small set of commands between the scripts and a parallel scripting runtime system, so that programmers can compose their scripts in a familiar scripting language. The underlying AMFS implements both collective (fast file movement) and functional (transformation based on content) file management. Tasks are handled by AMFS's built-in execution engine. AMFS Shell is expressive enough for a wide range of applications, and the framework can run such applications efficiently on large-scale computers.

Categories and Subject Descriptors

D.1.3 [**Software**]: Programming Techniques—*Parallel Programming*

General Terms

Design; Performance

Keywords

Many-task computing, Parallel scripting, Shared file system

1. INTRODUCTION

Many scientists seek to express applications in the MapReduce [11] model, because it provides a convenient mechanism for combining existing applications that offers simple parallelism and through Hadoop [4], is available on many platforms. However, many applications do not fit the MapReduce model well, for one or more of the following reasons:

- Applications include iterations, while the same data used in multiple iterations must be re-read from storage in each iteration with Hadoop.
- Applications have a trivial reduce stage.
- Applications want to run multiple distinct tasks in the mapper, where the parallelism does not only comes from data parallelism but also task parallelism.

The root problem is that many applications written in MapReduce [11] model are actually general purpose scripting applications: existing programs glued together with files as intermediate data. Since the MapReduce model only offers one dataflow pattern (reduce), this oversimplifies the dataflow patterns many applications require. Various research has been carried out to remedy these issues, such as Iterative MapReduce [15, 9] which provides iterative control flow. However, there is no single system that solves all issues.

We introduce here a fundamentally different approach: combining a simple scripting language with an underlying runtime system (an execution engine and in-memory file system) to enable parallel scripting applications. We ask: are a scripting language and the underlying system interface expressive enough to allow scientists to compose their applications? What runtime and file system support is required to enable efficient parallel execution of such scripts on large-scale computers?

More specifically, we integrate the Bash [26] scripting language with a redesigned AMFS (an Any-scale Many-task-computing File System) [35], a distributed in-memory file system with a built-in execution engine. The resulting *AMFS Shell* implements a scripting language that extends the Bash language with a small group of commands with which programmers can decorate an existing script to obtain a parallel program. Programmers prefix application statements with these commands to enable parallel execution; the implementation of these commands invoke AMFS operations to manage data movement, initiate parallel computations, and so forth. For example, the following fragment executes as an ordinary sequential script if the AMFS_queue prefix is omitted, and as a parallel computation if it is included.

```
mkdir temp/
for file in `ls link/`
do
  AMFS_queue PageRank_Distribution \
    link/${file} score.txt \
    temp/${file}.temp
done
```

The resulting system allows:

- **Simple programming:** Programmers can construct parallel scripts by modifying serial scripts.
- **Expressive programming:** Programmers can express the algorithms they desire.
- **Control flow:** Programmers can add script code to access the data between application stages and make dynamic decisions.
- **Scalable performance:** The scripts can run on thousands of compute nodes with modest overhead.
- **Seamless gluing:** Programmers do not need to change the applications that are called from their scripts.

As we will see when we introduce the AMFS Shell scripting language, the main abstraction that the language introduces is that of a parallel file system. AMFS implements a distributed in-memory file system spread across all compute nodes, with distributed metadata and data management. Programmers can explicitly load/store files and directories between AMFS and local storage sequentially or shared persistent storage in parallel. Programmers can execute application commands in parallel by calling AMFS commands, and have the flexibility to specify the output target: either to AMFS or persistent storage. AMFS provides collective and functional data management interfaces to programmers. With insight into the dataflow pattern of their scripts, programmers can expedite data movement or reorganize data based on file contents. AMFS currently supports Multicast, Gather, Scatter, Allgather, and Shuffle dataflow patterns.

The main challenge in designing this system is to define a set of commands that (a) can be inserted into scripts while preserving the semantics of the original application's POSIX interfaces, and (b) can be implemented in a manner that permits efficient execution and data movement. Figure 1 shows the design complexity of interfaces among the three entities of script, execution engine, and runtime file system. For efficiency, AMFS needs data locality information for data-aware scheduling and file system configuration to enable collective data movement. Existing interfaces for storage on large-scale computers, e.g., HDFS [6], only offer a customized file system interface to support locality exposure, so applications dependent on the POSIX interface can not be run. Traditional shared file systems, e.g., GPFS [27] and PVFS [10], do not expose data locality or system configuration though either POSIX or non-POSIX interfaces because they are treated as peripheral storage devices on large-scale computers. Neither approaches can offer both efficient execution and data movement *and* preserve the application interface, as AMFS Shell does.

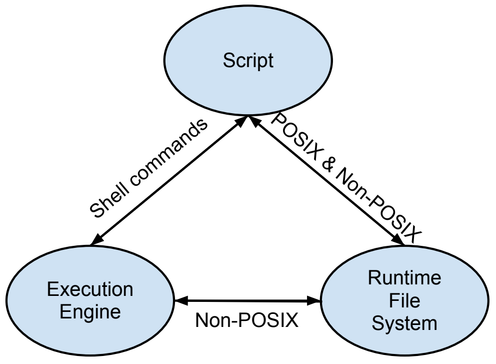

Figure 1: Interface complexity among AMFS Shell components

A second challenge is scalable metadata management, since we want to scale to thousands of compute nodes. Many existing distributed file system deployments, e.g., GPFS [27] and Lustre [14], preserve the consistency of concurrent metadata operations by placing all the metadata for any one directory on a single metadata server and applying a locking scheme for metadata in that directory. Because parallel scripting applications feature a large amount of metadata and data operations, this metadata server design has a side effect of limiting throughput, as we have documented in previous MTC-Envelope work [34].

In contrast to these systems, AMFS provides two different sets of interfaces to programmers and applications. AMFS provides programmers with a non-POSIX collective/functional data management interface in the form of shell commands, and it provides applications with a POSIX-compatible. The internal interface between the AMFS file system and its execution engine uses remote procedure calls. AMFS thus exposes data locality and system configuration through a non-POSIX interface to programmers, while preserving the POSIX interface to applications. AMFS also implements a distributed metadata server design, in which every node is both a metadata server and a storage server. Metadata are spread across nodes based on the hash value of the file name, while data is stored where they are produced.

To determine whether the AMFS Shell approach is a good fit for parallel applications, we studied a set of applications that have previously been expressed with MapReduce, namely: CloudBLAST [23], genome sequence alignment; CAP3 [20], sequence assembly; and high energy physics data analysis [16]. These applications build on a set of common primitives, namely PageRank, K-means, and Monte Carlo, which we show can be implemented effectively with AMFS Shell. we show scripts for the Montage, PageRank, K-means, and Monte Carlo applications as examples of AMFS Shell's expressiveness.

We use the MTC Envelope benchmark to evaluate performance; our results show that AMFS Shell runs 44 times faster than GPFS on 2,048 compute nodes. We also consider Montage [21] as an example of an application that has traditionally been expressed by a parallel script or workflow.

The time-to-solution for parallel stages of Montage are improved by 120% at various scales, compared to performance when using GPFS. PageRank performance on 1,024 compute nodes is improved by 40% compared to GPFS, ignoring the shuffle stage (GPFS does not support shuffle). Though AMFS Shell performs identical to GPFS on K-means and Monte Carlo, it allows simpler composition of parallel scripts and the applications that scientists currently run with MapReduce frameworks.

The contributions of this work are: a new parallel scripting programming model that extends a scripting language (Bash); a general runtime system interface design that works with existing scripting languages (that can execute shell commands); a novel file system access interface design that combines both POSIX and non-POSIX interfaces to ease programming without loss of efficiency; a scalable distributed file system design with no central point that achieves good scalability; and the implementation of the entire framework to enable various applications on large-scale computers.

2. PROGRAMMING MODEL

In the scripting programming paradigm, files and directories are often used as programmable elements much as in-memory variables are used in MPI [18] and key-value pairs in MapReduce [11]. A script glues programs together by specifying shared files or directories that are used to pass intermediate data. Thus, to produce or consume files and directories, applications must stick to the POSIX interface.

2.1 AMFS Shell Interface

AMFS Shell provides three groups of commands to programmers. Those in the first group are functional, meaning that the script can not correctly execute without them; they include:

- **AMFS_load**: loads a directory from persistent storage to AMFS; the transfer can be done in parallel if the persistent storage is globally accessible
- **AMFS_dump**: dumps a directory from AMFS to persistent storage; the transfer can be done in parallel if the persistent storage is globally accessible
- **AMFS_shuffle**: shuffles all files in a directory to another directory based on the first column of the file
- **AMFS_diff**: compares two directories to see if they contain an identical set of file contents; the comparison is on the file contents, regardless of the file names

The collective data movement commands in the second group can be thought of as performance hints, since a script runs correctly without calling them, but with lower performance; they include:

- **AMFS_multicast**: multicasts a file or a directory to all compute nodes
- **AMFS_gather**: gathers all files in a directory to one compute node
- **AMFS_allgather**: gathers all files in a directory to all compute nodes
- **AMFS_scatter**: scatters all files in a directory to all compute nodes

The third group implements an interface to the execution engine's task management system:

- **AMFS_queue**: push a task into the queue; user can specify the data-aware option though parameters
- **AMFS_execute**: execute all tasks in the queue, blocked until all tasks finish

AMFS_queue returns immediately, while AMFS_execute returns only when all queued tasks have finished. AMFS uses a work-stealing algorithm [35] to ameliorate inefficiencies that may result from trailing tasks [5] at the end of each AMFS_execute loop.

2.2 Examples

Listing 1 implements a data-parallel PageRank [8]. The script has three explicit stages. First, lines 6 to 11 run, for each file in link/, a PageRank_Distribution task to calculate the PageRank score distribution from one page to another, and then sum the scores for the same page. As score.txt is needed for every task, the programmer uses **AMFS_multicast** to broadcast score.txt to every node. Second, line 15 calls **AMFS_shuffle** to reorganize the contents of all files in temp/ to target/ according to the hash value of the first column of the files in temp/. Third, lines 17 to 24 invoke a PageRank_Sum task for each file in the target/ directory. PageRank_Sum sums the score for each page, applies a damping factor to the score, and writes the result to a file in result/. The programmer then calls **AMFS_gather** on result/ and redirects all files in result/ to new-score.txt. The script continues until a convergence condition (line 28) is satisfied.

Programmers require some insight into the dataflow and parallel execution patterns of their script if they are to determine which AMFS Shell commands are needed to achieve efficient parallel execution. For example, in parallel PageRank, if the **AMFS_gather** is missing, files are transferred sequentially, which may be acceptable on a small computer but slow on many processors. If **AMFS_allgather**, **AMFS_multicast**, and **AMFS_scatter** are missing, I/O traffic is congested due to the large number of concurrent connections to a single server; on large-scale computers, such congestion may be inefficient, or even compromise the stability of the parallel computer.

Listing 1: Parallel Script for PageRank

```
1  #!/bin/bash
2  while [ ${converge} -ne 0 ];do
3    AMFS_multicast score.txt
4
5    mkdir temp/
6    for file in ‘ls link/‘
7    do
8      AMFS_queue PageRank_Distribution \
9        link/${file} score.txt \
10       temp/${file}.temp
11   done
12   AMFS_execute
13
14   mkdir target/
15   AMFS_shuffle temp/ target/
16
17   mkdir result
18   for file in ‘ls target/‘
19   do
20     AMFS_queue PageRank_Sum \
      target/${file} \
      result/${file}.result
  done
  AMFS_execute

  AMFS_gather result/
  cat result/* | sort > new-score.txt
  converge = `diff score.txt \
    new-score.txt | echo $?`
  mv new-score.txt score.txt
done
```

Listing 2 is an AMFS Shell implementation of Montage [21]. Eight stages are expressed in 40 lines of code. The mProjectPP, mDiffFit, and mBackground stages can run in parallel; mImgtbl, mConcatFit, mAdd can benefit from parallel file transfer. mOverlaps and mBgModel can run on any AMFS server. In Line 20, the programmer produces the input file names for mDiffFit tasks by processing the content of diffs.tbl.

Listing 2: Parallel Script for Montage

```
1  #!/bin/bash
2
3  #mProjectPP
4  mkdir tempdir/
5  for file in `ls rawdir/`
6  do
7    AMFS_queue mProjectPP rawdir/${file} \
8      tempdir/hdu_${file} template.hdr
9  done
10 AMFS_execute
11
12 AMFS_gather  tempdir/
13 mImgtbl tempdir/ images.tbl
14
15 mOverlaps images.tbl diffs.tbl
16
17 mkdir diffdir/
18 #processing diffs.tbl requires
19 #programmer's interaction
20 for filepair in `process diffs.tbl`
21 do
22   AMFS_queue mDiffFit ${filepair} diffdir/
23 done
24 AMFS_execute
25
26 AMFS_gather diffdir/
27 mConcatFit diffdir/ fits.tbl
28
29 mBgModel images.tbl fits.tbl corr.tbl
30
31 mkdir corrdir/
32 for file in `process corr.tbl`
33 do
34   AMFS_queue mBackground \
35     ${file} corrdir/${file} corr.tbl
36 done
37 AMFS_execute
38
39 AMFS_gather corrdir/
40 mAdd corrdir/ final/m101.fits
```

Listing 3 shows an AMFS Shell script for K-means.

Listing 3: Parallel Script for K-means

```
#!/bin/bash
#File content format is: v(x, y)
while [ ${converge} -ne 0 ];do
  AMFS_multicast centeroid.txt

  #calulate the distance of every node
  #to the centeroids, output the closest
  #centeroid in the form of vertex
  #pair of (v c)
  mkdir temp/
  for file in `ls input/`
  do
    AMFS_queue KMeans-Group \
    input/${file} centeroid.txt \
    temp/${file}.temp
  done
  AMFS_execute

  #shuffle the intermediate file
  #according to centeroids
  mkdir temp/
  AMFS_shuffle temp/ cluster/

  #recalculate the centeroid of each
  #cluster
  mkdir centeroid/
  for file in `ls cluster/`
    AMFS_queue KMeans-centroid \
      cluster/${file} \
      centeroid/${file}.centeroid
  done
  AMFS_execute

  #aggregate new centeroid file
  AMFS_gather centeroid/
  cat centeroid/* | sort > \
    new-centeroid.txt
  converge = `diff centeroid.txt \
    new-centeroid.txt | echo $?
  mv new-centeroid.txt centeroid.txt
done
```

2.3 Further AMFS Script Examples

We next show how AMFS Scripts can handle file dependencies and data-aware scheduling.

2.3.1 *File Dependencies*

If an application involves two parallel stages, with each task in the second group consuming an output file from a task in the first stage (see Listing 4), it may appear that the tasks in the second stage cannot execute until all tasks in the first complete. However, as explained in §3.5, AMFS supports asynchronous file accesses, and thus this program can be rewritten, as shown in Listing 5, so that inter-task file dependencies are resolved in a distributed manner. In this alternative formulation, stage two tasks will fail if they are started before their predecessor stage two tasks have finished, but AMFS will put those tasks on hold until the files they are waiting for are produced, then restart them, leading to the same overall application functionality. Programmers can also express an asynchronous gather in this way, by queuing a task that consumes all output files of stage1 right before calling AMFS_exec.

Listing 4: Two Stages of Computation With File Dependencies

```
#!/bin/bash
for file in `ls input/`
do
   AMFS_queue stage1 input/${file} \
     temp/temp_${file}
done
AMFS_execute
```

```
for file in ‘ls temp/‘
do
   AMFS_queue stage2 temp/$temp_{file} \
     output/out_${file}
done
AMFS_execute
```

Listing 5: AMFS Resolving File Dependency

```
#!/bin/bash
for file in ‘ls input/‘
do
   AMFS_queue stage1 input/${file} \
     temp/temp_${file}
   AMFS_queue stage2 temp/temp_${file} \
     output/out_${file}
done
AMFS_execute
```

2.3.2 *Data-aware Scheduling*

AMFS supports data-aware scheduling as follows. The execution engine checks if the first parameter of an application task is a file. If so, and if the file exists, the task is forwarded to the node that holds the file.

2.4 Advantages & Limitations of AMFS Shell

AMFS Shell can express parallel scripting applications that feature simple data flows, iterative computation, and interactive analysis. It provides efficient execution because:

- AMFS caches files in memory;
- AMFS's scalable file access makes scripting performance scalable;
- the AMFS execution engine interface simplifies the parallelization of Bash scripts;
- the collective data management interface and the underlying system support moving data efficiently and scalably; and
- the AMFS interface to persistent storage enables checkpointing by copying or moving the files from AMFS to a persistent storage location.

AMFS Shell has limitations: it requires that all data fit in the compute node's memory. It can not launch tasks that need multiple compute nodes. The application shell commands are not declarative, so the AMFS execution engine doesn't know which files in the command line are inputs or outputs. Optimization based on declaring file usage does not work with AMFS Shell. AMFS Shell prefers longer tasks than shorter tasks. Finer-grained dataflow based applications can be better addressed by a dataflow-based parallel programming language, such as Swift/T [33] or DAGuE [7].

3. FILE SYSTEM REIMPLEMENTATION

AMFS was originally a distributed runtime file management system that ran in RAM disk. It did not handle OS I/O traffic directly, but managed the status and location of files stored in RAM disk. Application tasks queried and updated the status and location of files managed by AMFS via a customized command line interface. Collective file movement used FTP between nodes. In this work, we have redesigned and reimplemented AMFS as a distributed file system with data and metadata stored in compute node memory and added a POSIX-compatible interface based on FUSE [17].

The AMFS implementation places an AMFS service instance on every compute node involved in a computation. Each compute node is thus both an I/O server and a metadata server. Because parallel scripting applications have a multi-read single-write I/O pattern, we further define the AMFS I/O routines to be local-read-if-possible and local-write. That is, AMFS tries its best to place a task that reads a file on the node where that file is located, but cannot guarantee that all input files are local to the task. All writes are local to the task to maximize write performance. AMFS can be configured to support file striping, but in the results we present here, we constrain every file to be stored on a single server.

3.1 Metadata Management

AMFS uses different strategies to manage directory and file metadata. Directory metadata are synchronized across all AMFS servers, while file metadata are spread across AMFS servers based on the hashed value of the file path and name, as shown in Figure 2.

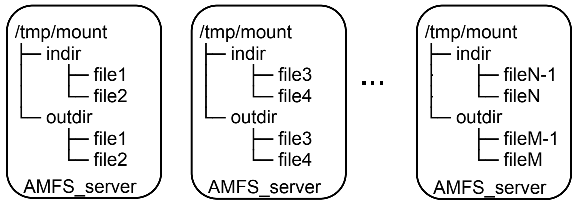

Figure 2: AMFS metadata placement

3.1.1 *Directory Metadata Management*

AMFS maintains directory metadata consistent across all nodes. Upon the creation of a directory, a create message is broadcast to all AMFS servers, and the initiating server blocks until all servers return. Every AMFS server creates an entry for this directory. The same logic is used to remove a directory. In a directory read operation, the initiating server broadcasts a read message to all AMFS servers, and along with the acknowledgement, each AMFS server sends all file metadata it stores locally. The initiating server then adds the subdirectory information to the file information, and returns. Directory metadata operations are optimized using a minimum spanning tree (mst) algorithm. The time complexity of such directory operations is O($logN$), where N is the number of AMFS servers. Since directory operations require synchronization among all AMFS servers, they do not work well when there are multiple concurrent directory metadata operations or directory-intensive operations.

3.1.2 *File Metadata Management*

File metadata are spread across AMFS servers based on the hash value of the file name. This metadata includes both regular information about a file, a list of the location(s) that store the actual file data, and a list of AMFS servers that have tried to access this file before it is produced. Once the file is produced, the AMFS server will broadcast the file's location to all AMFS servers that have tried to access it.

This feature supports distributed data dependency resolution. Distributing file metadata over AMFS servers is important to balance the many file metadata access operations that can occur in parallel scripting applications.

3.2 Data Management

A file's contents are always stored where the file is first produced. Upon a read, the file is replicated by the reading AMFS server. The appropriate metadata are also updated to record the new replica. AMFS also supports appending files, although this feature is not used by parallel scripting applications that feature a multi-read single write I/O pattern.

3.3 Collective Data Management

AMFS supports four types of collective data movement: Multicast, Gather, Allgather, and Scatter. Multicast, Gather, and Scatter are implemented with two algorithms: sequential and minimum-spanning tree. As many have discussed (e.g., [29, 28]), MST performs significantly faster for small files, as the data movement is latency bound and MST only invokes O(log$N$) file transfers on N servers. When the file size is large, the two algorithms deliver similar performance, as the data movement is bandwidth bound Programmers can select the sequential algorithm where the number of concurrent connections to a server is always one, if the network is congestion sensitive, compared to log$N$ connections in the MST algorithm. The AMFS collective data management interface allows programmers to specify which algorithm to use. We could also implement even faster algorithms based on network topology. Allgather is implemented as a gather followed by a multicast. The gather involves O(log$N$) inbound file transfers and the multicast another O(log$N$) outbound file transfers. §5.1 provides a model for collective data management.

3.4 Functional Data Management

The AMFS functional data management interface includes AMFS_load, AMFS_dump, AMFS_shuffle, and AMFS_diff. If persistent storage is not globally accessible by all AMFS servers, then the initiating AMFS server first computes a distribution plan to map files to AMFS servers and then executes that plan. If persistent storage is globally accessible by all AMFS servers, the initiating AMFS server computes a distribution plan and passes that plan to all AMFS servers; the AMFS servers concurrently load files from persistent storage. The same algorithm is used for AMFS_dump.

AMFS_shuffle is implemented as a four-step procedure. First, the initiating AMFS server notifies all servers via MST and returns once all servers have received the message. Second, all servers process the files in the specified directory locally and partition the files based on the hash value of the first column into hashmaps that are indexed according to which AMFS server is addressed. Third, every node starts exchanging data with its right-hand neighbor (using a virtual ring overlay), followed by its second nearest right-hand neighbor, etc., until it has communicated with all other AMFS servers. Fourth, the AMFS servers notify their parent server in the MST. The computational complexity of this operation is as follows. The control traffic (notification and acknowledgment) finishes in O(log(N)) time, where N is the number of servers. The local data processing is performed in parallel, i.e., in O(1) steps. The data transfer phase takes $O(N-1)$ rounds. The partition and exchange step of a simple shuffle on two AMFS servers is shown in Figure 3.

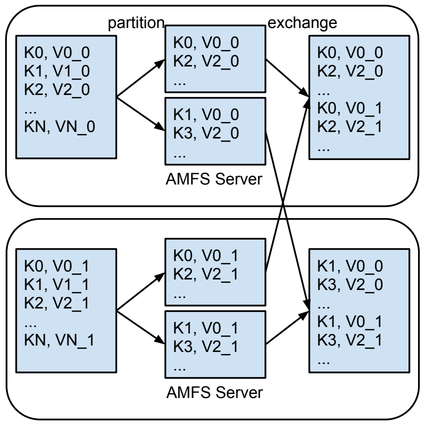

Figure 3: Shuffle example on two AMFS servers

AMFS_diff compares two directories. Unlike the standard diff, AMFS_diff fetches the metadata of all the files in two directories, and compares the file content checksum of all files in the directory, regardless of the file names.

3.5 Task Management

AMFS's task management scheme allows for two types of execution: straightforward and data-aware. Once the execution engine receives an execute command, it first evenly spreads all tasks over all AMFS servers. In data-aware execution, each AMFS server checks the location of the file that a program accesses, and forwards the task to the AMFS server that has that file. After the initiating server receives acknowledgements from the task dispatch phase, it sends another wait message to all AMFS servers. Those wait messages do not return until all tasks finish.

AMFS also allows for asynchronous file access: when a task execution fails because a file does not exist, the execution engine temporarily puts the failed task in a task hashmap, using the missing file as the key and the task as the value. At the same time, the AMFS server that will eventually hold the metadata of the file associates the requester's address with this file. When the file is later produced, the metadata entry is created on the AMFS server. The AMFS server then notifies the requester. The requester replicates the file from its producer and then retries the previously failed task. If a task has multiple input files, this process can repeat if other files are available.

4. INTERFACE DESIGN

As shown in Figure 1, AMFS Shell involves three interfaces, between scripts, execution engine, and file system. For each, we consider the tradeoffs between performance and programmability, then explain our decision.

4.1 Script - Execution Engine

In parallel scripting languages such as Swift [32], programmers declare inputs and outputs when defining applications tasks; Makeflow [2] uses a similar strategy by defining tasks with make file rules, so the interpreter knows how files are used. In addition to the commonly used input and output definitions, Pegasus [13] also uses 'intermediate' to mark files that do not need to be saved after they are read, to eliminate unnecessary data movement.

Intuitively, a programmer can express a task as a command line in the exact format in which the task is executed.

- File-usage blind: *bin/exec input.txt output.txt*

Here, the engine does not know which files are inputs and which files are outputs. Alternatively, the programmer can provide file usage information on the command line:

- File-usage aware: *bin/exec input.txt output.txt -if input.txt -of output.txt*

The file-usage aware format could enable the execution engine's logic to use techniques such as data-aware scheduling, automatic data replication, and smart data placement. On the other hand, the file-usage aware format is not intuitive. It requires the knowledge of file usage in addition to the command line that will execute the task. In enabling task dependency resolution on files, as we described in §3.5, the file-usage aware format can allow the execution engine to know on which files the task depends. When those files are available, the engine can execute the task.

In AMFS Shell, we compromise between the file-usage blind format and the file-usage aware format in the interface between script and execution engine. We require the file-usage blind format, as it is closer to real shell scripts that programmers would write. At the same time, AMFS Shell allows programmers who understand their file usage to write their script in a way that can permit the use of two performance-improving features:

1. Data-aware scheduling: In AMFS Shell, the runtime system assumes that the first parameter of an application task is file, whose location is the target node to run the task. If the first parameter is not a file or is a file that doesn't exist, AMFS Shell use its default task scheduling algorithm (an even distribution over all available AMFS servers) for this task.

2. Task dependency resolution: the AMFS Shell interface allows a task to run in a run-fail-queue-wake-run mode based on input file availability, as described in §3.5. However, the programmer needs to guarantee the correctness of the retried task, and make sure that the retried task does not have any side effects outside the task. (If the task has potential side effects, the programmer can call AMFS_execute on each stage separately, avoiding the use of this feature.)

4.2 Script - Runtime File System

The design options for the interface between the scripts and the file system are POSIX and non-POSIX. When a script needs to access a directory to find the file names in that directory (e.g., line 6 in Listing 1), the programmer can use `ls`. This is more convenient for programmers than a customized library remote procedure call. Collective file movement in this runtime file system design is not supported by the POSIX standard, so the only option is to implement the interface as customized command lines that are callable inside a script.

In AMFS Shell, the interface between the script and the runtime file system is a mix of POSIX compatible operations and non-POSIX commands. The POSIX interface allows a programmer to access files that are spread across multiple nodes, while the non-POSIX commands give performance improvements (AMFS_multicast, AMFS_gather, AMFS_scatter, AMFS_allgather), or build in support for particular file formats (AMFS_shuffle). This mixed design preserves the ease of programming without loss of potential performance improvements, though it requires the programmer to have knowledge of the dataflow patterns.

4.3 Execution Engine - Runtime File System

An execution engine communicates with the runtime file system for file location lookup and task execution. File locations can be exposed either by a customized remote procedure call or they can be embedded in the POSIX standard's additional attributes. In either case, extra execution engine logic is required to process the information. We chose a customized remote procedure call for AMFS Shell, as it is more flexible than enabling asynchronous file access.

When a task is executed, a POSIX-compatible file system can trivially provide file access. Since many programmers lack root access on the machines where they run, FUSE [17] is often used as a POSIX interface. However, the use of FUSE can cause I/O performance to drop dramatically because the FUSE write buffer is at most 128 KB. The FUSE based in-memory file system's write bandwidth is about 10% of that achieved by RAM disk, which is also in memory. A second solution is using a customized file system interface, which is more efficient but needs the task to be in file-usage-aware format and requires a translation layer between the file system and the execution engine. Our previous AMFS implementation used a Python wrapper to process input and output files to make the non-POSIX distributed file system transparent to execution engines. However, many applications can not benefit from the higher bandwidth of the non-POSIX interface, as the I/O is limited by the application tasks themselves. In such cases, a POSIX interface makes the communication between execution engine and runtime file system much easier. AMFS Shell implements the POSIX interface for task execution, since we previously decided that the task format would be file-usage blind.

5. EVALUATION

We evaluate AMFS Shell by performing three sets of experiments on an IBM Blue Gene/P supercomputer. The first set confirms the correctness of the collective data management implementation in AMFS and permits us to compare performance with that of our previous implementation based on files in RAM disk [35].

The second set profiles AMFS Shell's performance by measuring its MTC Envelope [34] at different scales. The MTC Envelope characterizes the capability of a parallel scripting applications on a given system in terms of eight performance metrics: file open operation throughput; file create operation throughput; 1-to-1 read data throughput; 1-to-1 read data bandwidth; N-to-1 read data throughput; N-to-1 read data bandwidth; write data throughput; and write data bandwidth.

The third set of experiments benchmarks application performance. We evaluate AMFS Shell implementations of three scientific applications for which we also have Hadoop implementations: PageRank,Monte Carlo, and K-means.We also show the performance of the Montage application to emphasize the benefits to interactive computation that are not apparent with today's MapReduce applications.

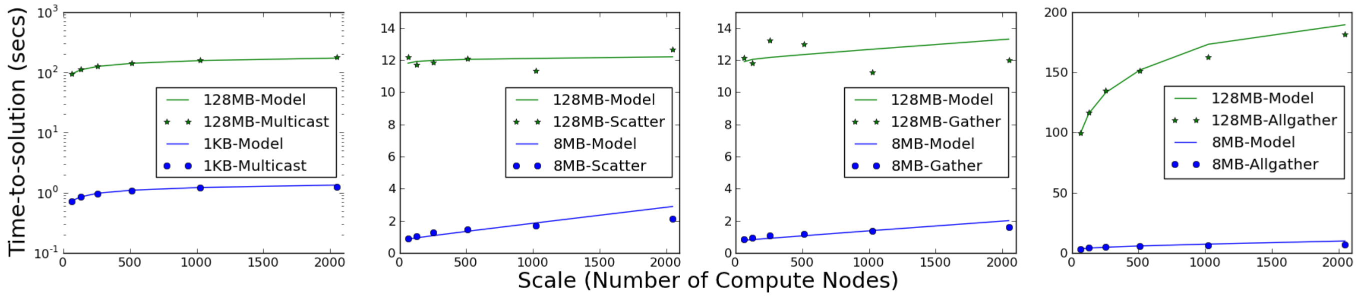

Figure 4: Collective file movement performance

5.1 Collective and Functional Data Management

The time consumption for the data functions are:

- Multicast: $T = (\log_2 N) * (a + b * S)$
- Gather: $T = (\log_2 N) * a + \frac{N-1}{N} * S * b + (M-1) * c$
- Scatter: $T = (\log_2 N) * a + \frac{N-1}{N} * S * b + (M-1) * c$
- Allgather: $T = 2 * (\log_2 N) * a + ((\log_2 N) + \frac{N-1}{N}) * S * b + (M-1) * c$
- Shuffle: $T = (N-1) * a + \frac{S}{N} * b + \frac{S}{N} * d$

where: T = time consumption, S = amount of data transferred, N = number of nodes, M = number of files, m = memory on each node, a = latency overhead/file transfer, b = bandwidth overhead/byte, c = file system overhead/file, and d = processing overhead/byte.

Figure 4 shows AMFS's collective file movement performance. As we discussed previously [35], Multicast shows O(log(N)) scalability regardless of file size. Gather, Scatter, and Allgather performance show O(log(N)) scalability when the data size is small; they are dominated by latency. For other sizes, Gather, Scatter, and Allgather show linear scalability.

The time consumption of Shuffle has three elements, due to latency overhead ($(N-1) * a$), bandwidth ($\frac{S}{N} * b$), and processing ($\frac{S}{N} * d$). It requires the total input storage, processing, and output storage data size on each node to be less than its available memory. Thus, the shuffle operations requires $N \geq \frac{3*S}{m}$ compute nodes.

On the other hand, for large N, $N-1$ approaches N and the time consumption can be estimated as $T = (N) * a + \frac{S}{N} * (b + d)$. The minimum value of this equation is $2 * \sqrt{S * a * (b+d)}$, when $N = \sqrt{\frac{S*(b+d)}{a}}$. In other words, using more than $\sqrt{\frac{S*(b+d)}{a}}$ compute nodes for a shuffle of files of size S won't improve performance. These two inequalities indicate that to enable an efficient shuffle operation on files of size S, the number of compute nodes has to meet the following condition:

$$\frac{3 * S}{m} \leq N \leq \sqrt{\frac{S * (b+d)}{a}} \qquad (1)$$

To validate AMFS_shuffle, we fixed the data size of the shuffle operations at 4 GB, and ran the shuffle function on

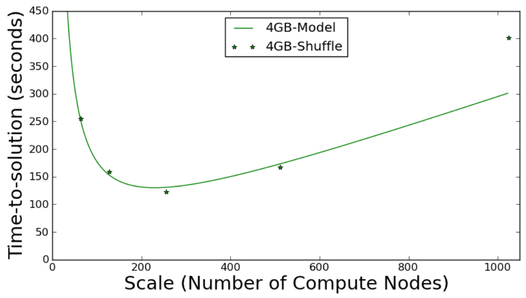

Figure 5: Shuffle performance

{64, 128, 256, 512, 1024} AMFS servers, as shown in Figure 5. As discussed in §3.4, a shuffle operation has four steps, and the partition and transfer steps dominate the time consumption. With fixed data size, increasing the number of servers results in a shorter partition time as the data size for each server decreases. On the other hand, the transfer step consumes more time due to the time complexity of the ring algorithm. We use the measured performance at 64 and 128 servers to regress the parameters in a time consumption model. Equation 1 predicts the optimal number of servers is 231 servers, while the best performance in our measurements runs on 256 servers. We see that the real system performance is far off on 1024 servers, because as the scale increases, the transfer congestion is more likely, which results in unmodeled delay.

5.2 MTC Envelope of AMFS Shell

We measured the MTC Envelope of an IBM Blue Gene/P with the ZeptoOS operating system and the GPFS shared file system on {64, 128, 256, 512, 1024, 2048} compute nodes. Then we measured the same performance metrics with AMFS replacing GPFS, as shown in Table 1. Figure 6 shows the comparison, where numbers greater than one show improvements, and numbers less than one show slowdown.

Most of metrics show improvements with AMFS Shell, as the file system I/O is cached in memory. Exceptions are open throughput on 64 compute nodes, and N-1 read throughput and bandwidth in general. On 64 compute nodes, though we spread the metadata across all compute nodes, the aggregated throughput is still lower than that of the GPFS metadata server, as the compute node is 850 MHz while the GPFS server is 2.6 GHz. The N-1 read is a two step procedure with AMFS Shell: a multicast followed by a group of synchronized concurrent reads to the local data replica. Since the concurrent local reads and the 1-1 read both access local memory, the N-1 read performance can be estimated as the sum of the multicast time and a group of concurrent 1-1 reads. Thus 1-1 read performance is an upper bound of N-1 read performance.

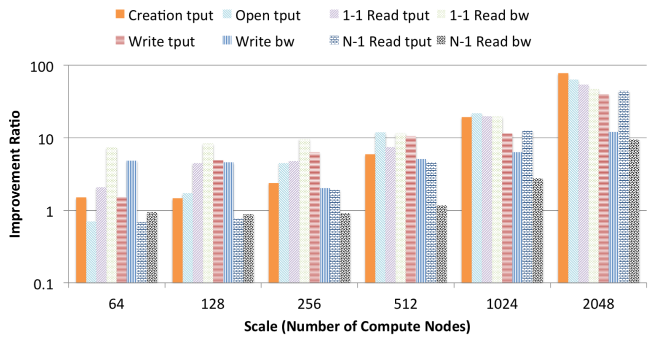

Figure 6: MTC Envelope improvements

Examining Table 1, we see that open throughput, 1-1 read throughput, 1-1 read bandwidth, write throughput, and write bandwidth increase close to linearly as the number of compute nodes increases. This linear scalability is the result of the combination of distributed metadata server topology, local write, and data-aware scheduling. Create throughput scales less well, as the file name distribution over all metadata servers is not uniform. The N-1 read is a two-step operation: multicast followed by read. The speedup of N-1 read throughput and bandwidth is a mix of the scalability of the multicast and read. Thus when the multicast stage dominates the N-1 read, the scalability of N-1 read is close to the scalability of multicast, otherwise it is close to the scalability of 1-1 read.

5.3 Application Performance

We run four applications—Montage, PageRank, K-means and Monte Carlo—with AMFS Shell on a Blue Gene/P supercomputer. Montage is an astronomy image processing application with a complex dataflow pattern. We run the parallel stages of a Montage workload that produces a 6×6 degree mosaic. The input, output, and intermediate data sizes are 3.2 GB, 10.9 GB, and 45.5 GB, respectively. mProjectPP reads one input file, and reprojects it. mDiffFit fits a plane to overlapped image pairs. mBackground reads one reprojected image and applies correction coefficients to it.

In PageRank, we run with the Wikipedia 5.7m-page page link dataset (1.1 GB size), to produce an output of 120 MB, with 4 GB of intermediate files. PageRank-Distribution distributes the score of the current page to the pages it links to, and produces output files with the link and its score. PageRank-Shuffle reorganizes those files so that all records of the same page are in a single file. PageRank-Sum adds the scores for each page to produce a new score for it.

In K-means, we randomly generate one million coordinates, and cluster them into 2,048 groups. This test has 19 MB input, 38 MB output and 76 MB intermediate data. KMeans-Group first computes each point's distance to all candidate centroids, and writes a pair of points with the nearest centroid in the first column. KMeans-Shuffle then reorganizes those output files based on the candidate centroids. KMeans-Centroid reads all point pairs with the same candidate centroid as key and calculates the new candidate centroid.

In Monte Carlo, we compute π by counting random coordinates in a square that fall in the circle with diameter equal to the square's width. MonteCarlo-Sim generates one billion random coordinates within a square, counts those that fall in the target circle, and writes this to an output file. MonteCarlo-Sum reads those files and adds them together.

In Figure 7, we first fixed each problem size, and ran the workloads on an increasing number of AMFS servers to study the scalability of the applications using AMFS Shell. In each application performance test, we compare each application stage performance on AMFS Shell against that on GPFS, except the shuffle stages of PageRank and K-Means, as shuffle is a built-in function in AMFS Shell.

5.4 Application Performance Observations

AMFS Shell both performs well and scales well for application stages where tasks access distinct files, such as Montage-mProjectPP, Montage-mBack, and KMeans-Centroid. The Montage-mDiffFit stage has a similar pattern; each task has two input files and one output file. PageRank-Distribution, PageRank-Sum, KMeans-Group and MonteCarlo-Sim also have this pattern, but the AMFS Shell improvement in these stages is marginal, because the long task running time hide the improvement in the small file I/O.

AMFS Shell works well when the I/O size is large but does not exceed the memory space of any AMFS server. Montage-mDiffFit's tasks each read in two 4 MB input file, and write a 1 MB output file. Our test case has 3,883 tasks. With AMFS Shell, all writes are local to memory, and data-aware scheduling guarantees at least one read file is in memory. That results in a 3.6x speedup of this stage.

For computations that involve the shuffle operation, larger scale does not imply shorter time-to-solution. The optimal scale for the problems size of our PageRank and K-means cases are 256 and 512, respectively.

AMFS Shell eliminates GPFS I/O traffic by caching intermediate files in memory. This not only offers better performance, but also removes load on the shared resources (GPFS) that might be used by other applications on the same system.

AMFS Shell is slower than GPFS on {64, 128, 256, 512} compute nodes for MonteCarlo-Sum, which has a single task that reads many input files. Even though we move the input files into one node using AMFS_Gather, the lower CPU frequency on the compute nodes than on the I/O nodes and the FUSE latency make AMFS Shell slower than GPFS on less than 1,024 nodes. However, we see that on 1024 nodes, AMFS Shell outperforms GPFS, as the task hits the GPFS concurrency wall, while the AMFS Shell's performance is almost consistent (O(log(N)) time, where N is the number of nodes.)

6. RELATED WORK

Many parallel scripting applications leverage the parallelism that MapReduce offers, including for example CloudBLAST [23], which runs the BLAST sequence alignment computation on clouds using Hadoop [4]. Hadoop, Amazon EMR [3], and Azure MapReduce [24] have been evaluated [19] as parallel frameworks for CAP3 [20], a sequence

Table 1: AMFS MTC Envelope Scalability. tput is throughput (in op/s), and bw is bandwidth (in Mb/s).

Scale	Create tput	Open tput	1-1 read tput	1-1 read bw	N-1 read tput	N-1 read bw	Write tput	Write bw
64	255.5	170.8	174.6	2215.3	139.7	537.3	129.6	499.8
128	1.5x	2.0x	2.2x	2.2x	2.1x	1.8x	2.0x	1.9x
256	2.9x	4.1x	4.5x	4.8x	4.2x	3.3x	4.1x	3.3x
512	6.8x	9.8x	10.1x	8.8x	8.6x	5.9x	8.2x	7.9x
1024	11.6x	15.7x	17.3x	16.9x	14.9x	11.0x	15.8x	16.0x
2048	19.4x	29.4x	29.6x	27.9x	26.6x	20.0x	29.0x	31.7x

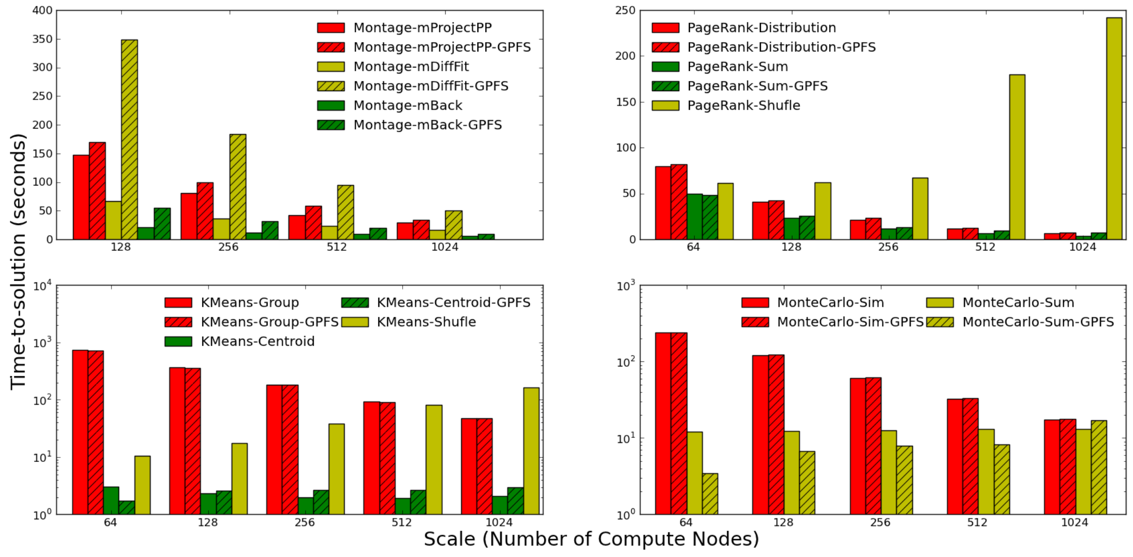

Figure 7: Application performance

assembly application. High energy physics text processing [19] is run in parallel with MapReduce.

Twister [15] and HaLoop [9] incorporate iteration support to accommodate iterative applications. Hadoop Streaming has been used for parallel execution of the parallel scripting applications, because it has the flexibility to run user-defined mapper and reducer functions as executables regardless of implementation language. However, this approach requires that all communications occur via standard input and standard output. MARISSA [12] runs Hadoop on a POSIX file system to eliminate this limitation.

Swift [32] and Makeflow [2] take a functional programming approach to compose and execute parallel scripting applications. Programmers can declare application tasks as functions and compose execution logic with control flow.

Bash over shell pipes [31] has been studied to enable parallel execution of Bash scripts. The programming abstraction here is key-value pairs. Computation stages exchange information through key-value pair aggregation. AMFS Shell uses files and directories as intermediate data format, which is more general than key-value pairs.

Locality has long been a key concern for performance purposes. HDFS [6] exposes data locality through a customized remote procedure call to the Hadoop execution engine. MosaStore [1, 30] takes the approach of embedding data locality in the extended attributes of the POSIX standard. Both approaches require that the execution engine process locality information in order to make it transparent to programmers.

Collective communications have been well studied in the MPI community [29, 28]. These communication patterns could speed up parallel scripting applications if properly implemented at the file system level.

As more parallel scripting applications have been enabled on clouds and large-scale computers, people have observed that the single metadata server design of file systems such as GPFS [27], PVFS [10], Lustre [14], and HDFS [6] is problematic at large scale. GIGA+ [25], ZHT [22], and other systems have explored a scalable metadata server architecture.

7. CONCLUSION AND FUTURE WORK

We have shown that a simple scripting language such as Bash can express many parallel scripting applications, including those that use the MapReduce model, using AMFS Shell. The AMFS Shell interface eases the programming of applications with simple dataflow patterns without breaking the application tasks' interfaces to the file system. The AMFS Shell data management scheme reduces congestion to provide faster collective file movement and it provides functional file content reorganization. AMFS Shell's measured collective file movement and functional file management features perform as expected. AMFS Shell extends the MTC Envelope (benchmark of parallel scripting applications) on the same machine by factors of {78, 64, 54 47, 40, 12, 44, 10} for {create-throughput, open-throughput, 1-1 read throughput, 1-1 read bandwidth, write throughput, write bandwidth, N-1 read throughput, N-1 read bandwidth}, respectively. Application performance confirms that AMFS Shell runs the parallel stages of Montage 2.2x faster on average on between 64 and 1024 compute nodes. AMFS Shell runs PageRank without shuffle 1.4x faster than GPFS on 1024 compute nodes.

In future work, we will deploy AMFS Shell on different platforms, such as supercomputers with other architectures and commodity clusters. We will comprehensively compare AMFS Shell with MapReduce models, along with their applications and frameworks. We will examine additional parallel scripting applications with AMFS Shell. We will develop better algorithms for the shuffle functionality of AMFS to make it more scalable and predictable.

Acknowledgments

This work was supported in part by the U.S. Department of Energy under the ASCR X-Stack program (DE-SC0005380) and DE-AC02-06CH11357. Computing resources were provided by the Argonne Leadership Computing Facility. Work by Katz was supported by the National Science Foundation while working at the Foundation. Any opinion, finding, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

8. REFERENCES

[1] S. Al-Kiswany, A. Gharaibeh, and M. Ripeanu. The case for a versatile storage system. *SIGOPS Oper. Syst. Rev.*, 44:10–14, March 2010.

[2] M. Albrecht, P. Donnelly, P. Bui, and D. Thain. Makeflow: A portable abstraction for cluster, cloud, and grid computing. Technical Report TR-2011-02, Department of Computer Science and Engineering, University of Notre Dame, 2011.

[3] Amazon. Amazon Elastic MapReduce. `http://aws.amazon.com/elasticmapreduce/`.

[4] Apache. Apache Hadoop. `http://hadoop.apache.org/`.

[5] T. G. Armstrong, Z. Zhang, D. S. Katz, M. Wilde, and I. Foster. Scheduling many-task workloads on supercomputers: Dealing with trailing tasks. In *Proc. of Many-Task Comp. on Grids and Supercomputers, 2010*, 2010.

[6] D. Borthakur. HDFS architecture. `http://hadoop.apache.org/hdfs/docs/current/hdfs_design.pdf`.

[7] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra. DAGuE: A generic distributed DAG engine for high performance computing. *Parallel Computing*, 38(1):37–51, 2012.

[8] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.

[9] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: Efficient iterative data processing on large clusters. *Proceedings of the VLDB Endowment*, 3(1-2):285–296, 2010.

[10] P. H. Carns, W. B. Ligon, III, R. B. Ross, and R. Thakur. PVFS: a parallel file system for linux clusters. In *Proc. of the 4th annual Linux Showcase & Conf. - Volume 4*, pages 28–28. USENIX Association, 2000.

[11] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. 8th USENIX Symposium on Operating Systems Design and Implementation*, pages 137–150, 2004.

[12] E. Dede, Z. Fadika, J. Hartog, M. Govindaraju, L. Ramakrishnan, D. Gunter, and R. Canon. MARISSA: MApReduce Implementation for Streaming Science Applications. In *Fourth IEEE International Conference on eScience (eScience '08)*, pages 1–8. IEEE, 2012.

[13] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahl, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming Journal*, 13(3):219–237, 2005.

[14] S. Donovan, G. Huizenga, A. J. Hutton, C. C. Ross, M. K. Petersen, and P. Schwan. Lustre: Building a file system for 1000-node clusters, 2003. Proceedings of the 2003 Linux Symposium.

[15] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: a runtime for iterative MapReduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 810–818. ACM, 2010.

[16] J. Ekanayake, S. Pallickara, and G. Fox. MapReduce for data intensive scientific analyses. In *IEEE Fourth International Conference on eScience*, pages 277–284. IEEE, 2008.

[17] FUSE Project. Fuse: Filesystem in userspace. `http://fuse.sourceforge.net`.

[18] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Par. Comp.*, 22(6):789 – 828, 1996.

[19] T. Gunarathne, T.-L. Wu, J. Qiu, and G. Fox. MapReduce in the Clouds for Science. In *IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom 2010)*, pages 565–572. IEEE, 2010.

[20] X. Huang and A. Madan. CAP3: A DNA sequence assembly program. *Genome research*, 9(9):868–877, 1999.

[21] J. C. Jacob, D. S. Katz, G. B. Berriman, J. C. Good, A. C. Laity, E. Deelman, C. Kesselman, G. Singh, M.-H. Su, T. A. Prince, and R. Williams. Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking. *Intl. J. of Comp. Sci. and Eng.*, 4(2):73–87, 2009.

[22] T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, and I. Raicu. ZHT: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table. In *27th International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE, 2013. to appear.

[23] A. Matsunaga, M. Tsugawa, and J. Fortes. CloudBLAST: Combining MapReduce and virtualization on distributed resources for bioinformatics applications. In *Fourth IEEE*

International Conference on eScience (eScience '08), pages 222 –229, 2008.

[24] Microsoft. Windows Azure. `http://www.windowsazure.com/`.

[25] S. Patil and G. Gibson. Scale and concurrency of GIGA+: file system directories with millions of files. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, pages 13–13. USENIX Association, 2011.

[26] C. Ramey. Bash, the bourne-again shell. 1994.

[27] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proc. of 2002 Conf. on File and Storage Technologies FAST*, pages 231–244, 2002.

[28] R. Thakur and W. Gropp. Improving the performance of collective operations in MPICH. In J. Dongarra, D. Laforenza, and S. Orlando, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 2840 of *Lect. Notes in Comp. Sci.*, pages 257–267. Springer, 2003.

[29] R. Thakur and R. Rabenseifner. Optimization of collective communication operations in MPICH. *Intl. J. of High Perf. Comp. Applications*, 19:49–66, 2005.

[30] E. Vairavanathan, S. Al-Kiswany, L. Costa, M. Ripeanu, Z. Zhang, D. S. Katz, and M. Wilde. A workflow-aware storage system: An opportunity study. In *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2012.

[31] E. Walker, W. Xu, and V. Chandar. Composing and executing parallel data-flow graphs with shell pipes. In *Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science*, page 11. ACM, 2009.

[32] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster. Swift: A language for distributed parallel scripting. *Par. Comp.*, pages 633–652, September 2011.

[33] J. M. Wozniak, T. G. Armstrong, M. Wilde, D. S. Katz, E. Lusk, and I. T. Foster. Swift/T: Scalable data flow programming for distributed-memory task-parallel applications. In *Proceedings of the 2013 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2013)*, 2013.

[34] Z. Zhang, D. Katz, M. Wilde, J. Wozniak, and I. Foster. MTC Envelope: Defining the capability of large scale computers in the context of parallel scripting applications. In *Proceedings of the 22nd ACM International Symposium on High Performance Distributed Computing.* ACM, 2013.

[35] Z. Zhang, D. Katz, J. Wozniak, A. Espinosa, and I. Foster. Design and analysis of data management in scalable parallel scripting. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 85. IEEE Computer Society Press, 2012.