Makeflow: A Portable Abstraction for Cluster, Cloud, and Grid Computing

Michael Albrecht, Patrick Donnelly, Peter Bui, and Douglas Thain
Department of Computer Science and Engineering
University of Notre Dame

ABSTRACT

In recent years, there has been a renewed interest in languages and systems for large scale distributed computing. Unfortunately, most systems available to the end user use a custom description language tightly coupled to a specific runtime implementation, making it difficult to transfer applications between systems. To address this problem we introduce Makeflow, a simple system for expressing and running a data-intensive workflow across multiple execution engines without requiring changes to the application or workflow description. Makeflow allows any user familiar with basic Unix Make syntax to generate a workflow and run it on one of many supported execution systems. Furthermore, in order to assess the performance characteristics of the various execution engines available to users and to assist them in determining which engine to use we introduce Workbench, a suite of benchmarks designed to compare the performance of various execution engines. We evaluate Workbench on two physical architectures – a storage cluster and a high performance computing cluster – using a variety of execution engines. We conclude by demonstrating three applications that use Makeflow to execute data intensive applications consisting of thousands of jobs.[1]

1. INTRODUCTION

Many problems in both science and industry ranging from web indexing to genome analysis can be expressed as a graph of small sequential programs with a high degree of parallelism. A number of *workflow systems* [11, 17, 19, 22, 12, 9, 34] have been created to express and execute such programs. While these systems have many virtues, they typically couple a custom language to a custom runtime implementation, making it difficult to move applications across systems, or even to evaluate which system is most appropriate for a given application.

We argue that there has long existed a portable and effective language for data parallel computing. Traditional Make [13], although most commonly used for compiling and linking programs, is also very effective at expressing highly parallel data intensive applications. To this end, we present a new implementation called *Makeflow* (Make + Workflow) that can portably run the same applications across multicore processors, dedicated clusters (SGE), cycle scavenged grids (Condor), storage clouds (Hadoop), and combinations of the above (Work Queue), without requiring any changes to the application or the workflow description. This makes it possible to develop an application on a personal computer, and them seamlessly move it between institutional clusters and commercial clouds without any restructuring.

Of course, there have certainly been many *parallel* implementations of Make presented previously [5, 24, 4, 26]. These all assume a homogeneous set of reliable processors, all connected to a common shared filesystem. Makeflow goes beyond these previous systems with a truly *distributed* implementation that runs on heterogeneous, failure-prone distributed systems, taking advantage of data locality when the implementation allows it. To enable this, we require a small but important change to the semantics of Make: data dependencies must be completely elaborated.

Because Makeflow provides transparent portability of applications across systems with significantly different properties, it allows us to perform an objective comparison of the relative capabilities of each system for different types of workloads. To this end, we have created *Workbench*, a system-independent set of workflow benchmarks that measures dispatch latency, job throughput, I/O throughput, and interprocess communication. We evaluate Workbench on two distinct architectures – a storage cluster and a high performance computing cluster – using each of the execution engines supported by Makeflow. These results help the end user to select the right type of execution system for the workload at hand.

Makeflow is open source software that is currently in production use by a number of scientific communities. (To be clear, Makeflow is designed for data intensive scientific applications, and is not particularly suited for compiling and linking programs.) We conclude with a selection of bioinformatics applications using Makeflow, currently in use by a cloud computing portal at Notre Dame.

Several previous publications have mentioned Makeflow in passing. A journal article [33] and a book chapter [28] briefly discuss Makeflow as an example of one of several kinds abstractions for distributed computing. This is the first publication to discuss Makeflow in detail, to present the Workbench benchmarks, and to evaluate workloads across multiple implementations.

2. THE MAKEFLOW LANGUAGE

The Makeflow language is very closely related to the traditional language of Make [13]. A valid Makeflow program consists of a sequence of assignments and rules. An **assignment** indicates the name and value of an environment variable, which applies to all

[1] This work was supported in part by National Science Foundation grants CCF-0621434 and CNS-0643229.

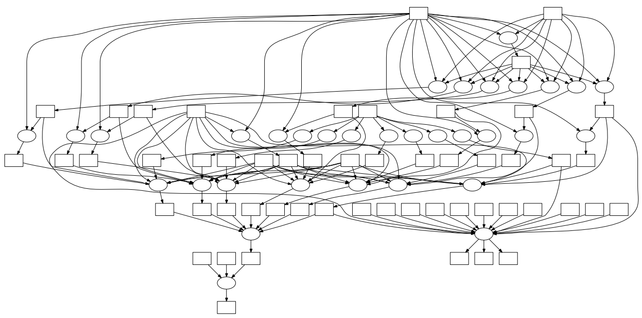


Figure 1: Example of a Bioinformatics Application in Makeflow

succeeding items in the file. A **rule** indicates a command line to be executed, along with the input files required by that command, and the output files that it will create.

A Makeflow rule has slightly different semantics than traditional Make. Traditional Make simply requires that a rule state only the files that may have changed, but assumes that any other file in the filesystem is available for use by the command. In contrast, a Makeflow rule must accurately specify *all of the files* that a command requires as both input and output, because this is used to create the correct execution environment.

For example, the following is an **incorrect** Makeflow rule:

```
out.dat:
   simulate.exe in.dat -o out.dat
```

The following rule is **correct**, because it correctly specifies all of the input dependencies, including the executable and data file:

```
out.dat: simulate.exe in.dat
   simulate.exe in.dat -o out.dat
```

Figure 1 is a visualization of a relatively small bioinformatics job expressed in Makeflow, consisting of 33 jobs (circles) and the interdependent files (squares). In practice, workflows are often very large, consisting of thousands to millions of jobs processing terabytes of data. (The largest are difficult to present in graphical form.) As one may see, the graphs may be highly irregular and thus not easily expressed in a fixed abstraction such as Map-Reduce [11].

Once a workflow is expressed with fully elaborated data dependencies, a number of opportunities for executing the workflow efficiently and scalably become possible:

Job migration. When the full dependencies of each job are known, the single job may be moved to a remote execution site, without requiring any particular runtime support on a shared filesystem. This enables harnessing of resources that are outside the immediate execution environment.

Workflow migration. With the inputs and outputs of the whole workflow are known, it becomes easy to move the entire workflow to another site. For example, one might allocate a cluster on a commercial cloud, send the inputs, execute the entire workflow remotely, and then retrieve the outputs.

Workflow decomposition. Given sufficient information about jobs and data, it may make more sense to partition the graph, and run sub-graphs in distinct systems. As we show below, some execution systems are more effective at partitioned data, and others at shared data.

Co-location of computation and data. When operating on distributed data, it is often beneficial to move computation to where

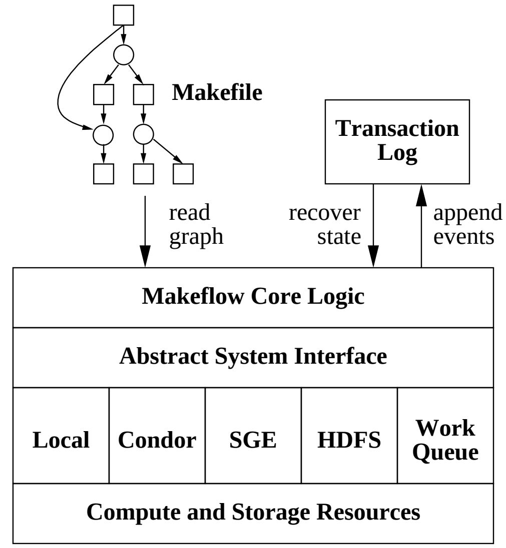


Figure 2: Architecture of Makeflow

the data is located, or vice versa. With information about the data needs of each application, an implementation of Makeflow can seek to put them together.

Resource management. Cloud computing environments in particular require the user to make resource management decisions: allocating more machines always costs more money, but may not necessarily improve performance. Accurate information about the computation and data needs of a workload makes it feasible to select appropriate resources for an execution.

In order to exploit these management and performance properties, we have very deliberately chosen **not** to include any of the higher-order constructs found in other versions of Make, such as implicit and pattern rules. For example, if we include pattern rules, then it is no longer possible to measure the width or depth of the graph without actually executing it. Makeflow is a static declarative specification and nothing more.

We have not found this to be a serious limitation in practice: where more dynamic behavior is needed, it is accomplished by writing a script which generates the desired workflow, which can then be processed as an independent step. (See our earlier paper on Weaver [6] for examples.) By analogy, HTML has achieved universal success by aiming simply to be a declarative statement of the structure of a web page; programmability has been obtained by a variety of languages that generate HTML.

3. IMPLEMENTATION

We have created an initial implemenation of Makeflow that executes complex workflows effectively on several different execution platforms. Our implementation is open source software under active development, with a growing user community. [2] The work here describes version 3.3.1 of the software, and exploits many (but not all) of the opportunities made possible by the graph representation.

Figure 2 shows the architecture of the current implementation. The user provides a workflow in the form of a Makefile, then executes the `makeflow` program. The Makeflow core logic manages the graph of processes and data, submits jobs to the abstract system interface, and records events in a transaction log. The abstract system interface provides an API for submitting a single job at a

[2] http://www.cse.nd.edu/~ccl/software

time, indicating input and output dependencies, and the ability to wait for the completion (or failure) of any previously submitted job. Drivers for multiple execution systems are hidden behind the abstract interface, and include Local execution, Condor, Sun/Oracle Grid Engine, Hadoop, and Work Queue, which we describe below. Events in the life of the workflow are recorded in a transaction log, which is used for both failure recovery and system monitoring.

3.1 System Drivers

Two details of the system drivers are worth noting. First, each driver is a combination of a computation environment with a tightly-coupled storage system. Local is actually *local processes and a local filesystem*, while Condor is actually *Condor execution with file transfer*, and so forth. Each represents a signficantly different method of accessing data at runtime. We will elaborate on these properties in each section below, but use the short name for clarity.

Second, we have found that many execution environments have poor support for monitoring the status and completion of many jobs asynchronously. All provide some interactive command or web page for observing system or job status, but this is unwieldy to access from within Makeflow, because the information may be presented poorly, or the call may be very slow to invoke. Thus in each case we describe the method by which we monitor the status of executing jobs, and in many cases it involves a creative workaround to bypass the limitations of the system.

Local Driver. In Local execution mode, all jobs with satisfied data dependencies are forked as new processes that execute on the local machine. Makeflow then monitors the status of each child process and marks the job description as completed or failed once the child exits. The local engine uses the local filesystem as its storage component. Where multiple cores are available, multiple processes can run simultaneously. Local execution is often used for pre-testing workflow executions to ensure they are constructed properly before scaling up.

Users can recommend local execution for specific jobs even when executing on a distributed system by prefixing the rule with the `LOCAL` keyword. This informs Makeflow that a particular command is optimally run locally, due to filesize, permissions, or configuration constraints, while still allowing the command to be managed as part of the workflow.

Condor Driver. Condor [29] is a distributed batch computing system that can be used across hardware ranging from desktop machines to high performance clusters. Condor provides a comprehensive matchmaking system to match jobs to their hardware requirements as well as to ensure fair usage of shared resources without inconveniencing their owners.

Figure 3 shows the interaction between Makeflow and Condor. For each job to be executed, Makeflow creates a job submission file and invokes `condor_submit`, which queues the job with the local `condor_schedd` daemon. The job submission file indicates the input and output files required for the job. `condor_schedd` communicates with the matchmaker to find a compatible execution machine. At the execution site, the input files are retrieved from the submission site, the job is executed, and output files are moved back. When multiple jobs execute simultaneously, the transfers may happen concurrently.

To provide lightweight notification of job status, Condor produces a *user log file* that indicates when a job starts, completes, migrates, and so forth. Makeflow monitors job status by periodically looking for new data appended to the file.

SGE Driver. Sun/Oracle Grid Engine (SGE) [14] is a batch system for managing large clusters. SGE schedules jobs across the nodes in a cluster, typically requiring the use of a shared filesys-

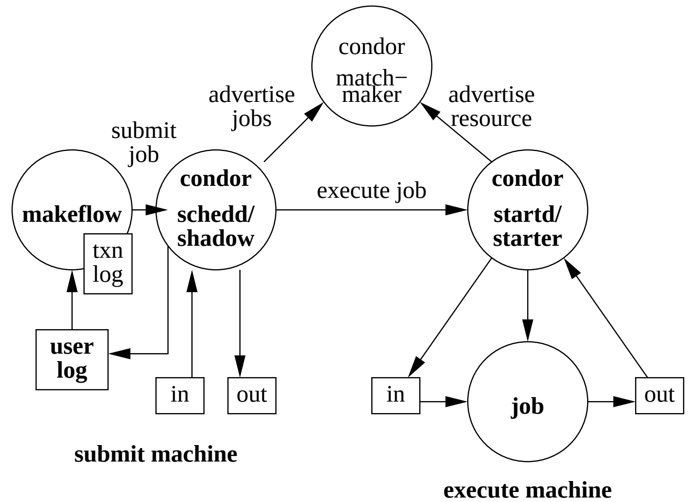


Figure 3: Makeflow on Condor

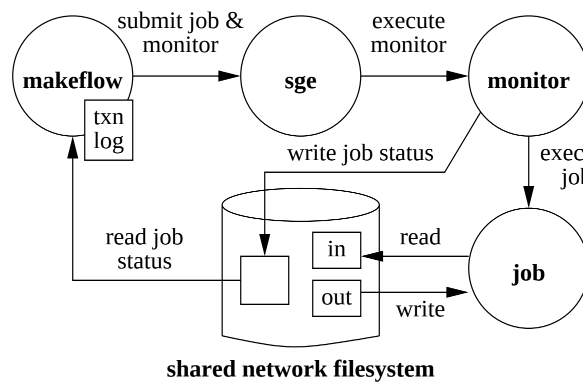


Figure 4: Makeflow on SGE

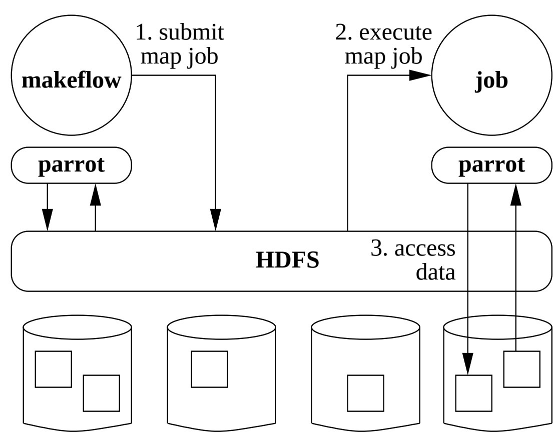


Figure 5: Makeflow on Hadoop

tem such as NFS or AFS to provide uniform data access across the cluster. Unlike Condor, the queue of jobs in SGE is stored in a centralized location, which exercises absolute control over each node in the cluster.

Figure 4 shows the interaction between Makeflow and SGE. To submit a job, Makeflow invokes the `qsub` command, indicating a `wrapper` script and an executable. No information about data needs is communicated. The job is run in the shared filesystem where the `wrapper` script may be executed by the node. The job's executable and its data are also accessed through the shared filesystem. The `wrapper` script also writes to a log file the job's start time, completion time, and exit status. Makeflow monitors the log files of all running jobs to observe completion events.

Hadoop Driver. Hadoop [17] is an open source implementation of the Map-Reduce [11] distributed computing concept. HDFS is the file system component, corresponding to Google GFS [15]. Normally, to interact with Hadoop, one submits a Java program with a Map component and a Reduce component. This program is distributed to all nodes simultaneously and used to construct the graph of communicating processes.

It can be challenging to convert an existing application to a Map-Reduce form. A Map-Reduce program accepts input data as it is fed in, and does not normally open arbitrary files at run-time, so I/O must be converted into a streaming form, if possible. The programmer must decide how to partition code among the Map and the Reduce components and how to partition data horizontally. Multiple invocations of Map-Reduce may be necessary to achieve the desired goal.

With Makeflow, our objective is to make it possible to run existing, unmodified applications within the Hadoop environment, achieving acceptable parallelism and performance with minimal effort by the user. If all the user's code and data are already in Hadoop, Makeflow provides a way to execute applications that are not obviously of a Map-Reduce form.

Figure 5 shows how this is achieved. Makeflow passes its jobs to Hadoop as single-node Map-only jobs. Each job consists of the streaming wrapper program, the desired executable and the name of its input dependencies. Hadoop will then execute the job, ideally on a node close to the input data. The job is completely unaware of the Map-Reduce framework and will attempt to access files in the normal way through the filesystem. To enable this, we rely on the Parrot [27] interposition agent to transparently convert the application's system calls into the corresponding operations on HDFS. (FUSE [1] can also be used for the same purpose, but requires administrator privileges to install.) The progress of each job is tracked by forking a child process to monitor the standard output and standard error from the Hadoop streaming wrapper. The monitoring process writes the status into a file which the main process reads.

Work Queue Driver. Most of the available job execution engines are managed via a queue system, where jobs are submitted to a central manager which then dispatches jobs based on the available resources and system policies. As we show below, this can result in long dispatch times with jobs sitting in a queue for minutes or longer. For long-running or data-intensive jobs this dispatch time may be subsumed by the computation time of the job itself; for a job which consists of huge numbers of short-running jobs this type of system may impose 10-100 or more seconds of wait time for each second of computation.

To address this problem, we have created Work Queue, a master-worker framework designed to work natively with Makeflow. Work Queue provides fast invocation times, local storage management, and a means to rapidly deploy an execution environment for the user that is not already invested in a batch system.

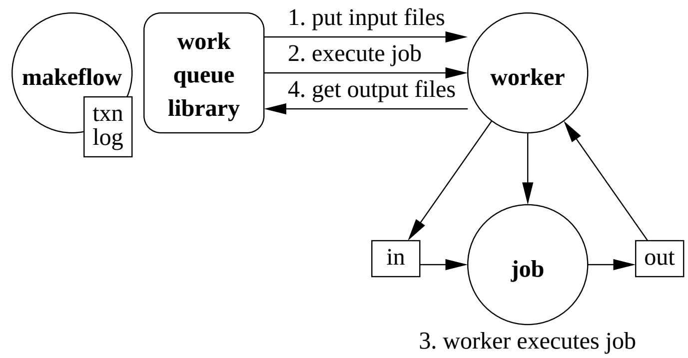


Figure 6: Makeflow on Work Queue

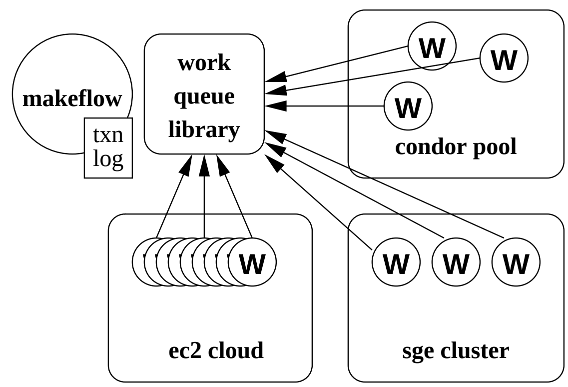


Figure 7: Building a Private Cloud with Work Queue

Figure 6 shows the relationship between Makeflow and Work Queue. The system consists of a lightweight worker process and a master library which is linked into Makeflow. Workers may be submitted as jobs to one or more distributed systems (such as Condor or SGE), or run by hand on machines accessible to the user. To execute a job, the master simply transmits the required input files, tells the worker to execute a command line, and then retrieves the outputs. Both inputs and outputs can be cached at the worker with the knowledge of the master, so that jobs can be automatically located with their input data.

The Work Queue system achieves fault tolerance in the following manner: The connection between each worker and the master is via TCP. If the worker or the network fails the master drops its internal record of the worker, assumes any running job failed, and reschedules it for another worker. If the master or the network fails the worker stops the running job, cleans up any cached data, and then attempts to connect again, up to a configurable timeout. The system safely accepts workers joining and leaving at runtime, albeit with some performance penalty.

Figure 7 shows how Work Queue can be used to create a private cloud environment. The Worker process can be submitted as a batch job to systems such as Condor and SGE, or executed on resources allocated from a commercial cloud. Regardless of how the worker is started, it contacts the master process, and begins to execute jobs and cache data. From the perspective of Makeflow, the set of workers forms a cloud of both computation and storage resources.

3.2 Transaction Log

As Makeflow processes a workflow, it produces a transaction log.

This is a complete description of the life of the workflow, including all job submissions, completions, and failures. The log file serves two purposes: it facilitates recovery from failures, and it provides data for troubleshooting and performance analysis. The log is kept by Makeflow at the submission site, and is independent of the various logs produced by the batch systems.

Fault tolerance is critical in a distributed computing environment. Makeflow must operate with multiple distributed computing systems over the wide area, each with their own peculiar failure modes. It is not uncommon for the network or a batch system to fail during a job execution. A job within a workflow might run for hours or days, during which time Makeflow may become disconnected from the batch system, or killed outright. A careless approach to job submission may result in multiple jobs being submitted unnecessarily, or orphaned jobs left running in the system with no corresponding record in Makeflow.

To address these problems, every event in the lifetime of a job is recorded in the transaction log. Rather than rely on the presence of files (as a traditional Make would), Makeflow examines the transaction log to recover the state of running jobs. This allows Makeflow to crash and restart while allowing its jobs to continue to run. If Makeflow receives a signal to abort, it works to remove jobs from the batch system, logging as it goes. Interacting with the batch system can take time, so if Makeflow crashes and restarts during an abort, it will pick up the current state, and then continue to abort cleanly.

Ideally, batch systems would facilitate this by providing an interface for *two phase commit*. This would allow Makeflow to first request a job number from the batch system, record it to the transaction log, and then commit the job within the system. To our knowledge, no batch system provides this through a public interface. (Condor does this internally, but does not expose an API.) As such, our implementation has a short time window in which a failure could result in duplicate job submission.

The use of the log has the side effect that Makeflow can be killed and moved to a completely different machine, and as long as the corresponding workflow, transaction log, and dependencies are available, the workflow can be restarted without duplicating work. It also allows the user to switch execution engines mid-workflow. If the initial execution engine bogs down under external strain or part of the system fails permanently the user can stop the workflow and restart it using a different engine while preserving all of the work done up to that point.

The provenance information provided by the transaction log also makes debugging, performance analysis, and job status monitoring much easier. By maintaining the logs of which jobs fail, how many times they are retried, and what times the failures occur, it makes tracking down the problem easier. The details of the log file allows us to keep track of which jobs are unexpected bottlenecks or how close to completion the workflow is. The real-time timestamps allow the user to correlate job events with the logs maintained by many of the Makeflow subsystems in order to identify problems with the subsystem and to correctly identify which failures are due to the application itself and which are caused by the execution engine.

An example transaction log created from this simple workflow:

```
b.dat: simulate.exe a.dat  # node 0
simulate.exe a.dat -o b.dat
c.dat: simulate.exe b.dat # node 1
simulate.exe b.dat -o c.dat
```

looks like this:

```
1301517576747953 0 1 15739 1 1 0 0 0 2
1301517576803854 0 2 15739 1 0 1 0 0 2
1301517576806569 1 1 15742 0 1 1 0 0 2
...
```

Each event in the log describes a change in state of a single job. The first four columns indicate the state change by the current time, the node number, the new state of the node, and the job id. The next five columns are a convenience that show the total number of nodes in each state: waiting, running, completed, failed, and aborted. The final column shows the total number of nodes in the workflow.

The log is designed to be easily machine readable. In order to monitor the progress and analyze the output of our workflows, we have developed a handful of useful example scripts to parse the transaction log and generate statistics about the workflow's execution. These scripts, `makeflow_monitor` and `makeflow_analyze`, are made available alongside Makeflow to provide both useful analysis tools to researchers using Makeflow as well as to provide a basis for custom analysis scripts.

4. WORKBENCH

Makeflow allows one to run workflows across several different execution engines and physical architectures, including multicore systems, shared-nothing clusters, shared-filesystem clusters, and distributed transient caches. As a result, one would expect that different kinds of workflows would be better suited to different kinds of architectures.

To evaluate the essential performance characteristics of a system, we created Workbench, a set of simple workflow benchmarks. Figure 8 shows the basic patterns, which are parameterized to run at various scales. Of course, none of these patterns is representative of a complete workflow, but each exercises a different aspect of the system, including dispatch overhead, job throughput, I/O throughpot, and interprocess communication. By understanding these basic parameters, we can better judge the expected performance of a real workflow, which consists of many of these patterns put together.

We note that Workbench very deliberately exercises the *pathological cases* in workflow performance. We expect that any execution system would be effective at running a large number of independent processes that each run for hours with minimal input and output. The differences between systems are apparent only in the more difficult cases that stress I/O and/or large numbers of short jobs.

The five Workbench patterns are:

Chained(J,T) consists of a chain of J jobs running for T time, each producing one empty output file, which is consumed by the next. When $T = 0$ the chained benchmark measures the average latency to submit a job, which puts a lower bound on the execution of any job. As we show below, many systems have a surprisingly high job latency.

Concurrent(J,T) consists of a set of J independent running for T time and producing no input or output. This benchmark measures the throughput of job dispatch and completion. If $T > 0$, then the throughput is likely to vary with the number of available processors.

FanIn(J,T,F,S) consists of a set of J jobs running for T time, each reading F files of size S as input. This benchmark measures the ability of the system to deliver input data. At large F but small S, it stresses the number of file transactions, while at small F and large S, it stresses the total data throughput.

FanOut(J,T,S) consists of a set of J jobs running for T time, with a common file of size S as an input and independent files of size S as outputs. This type of workflow is common in simulations, where the input file represents the starting data and each job is one

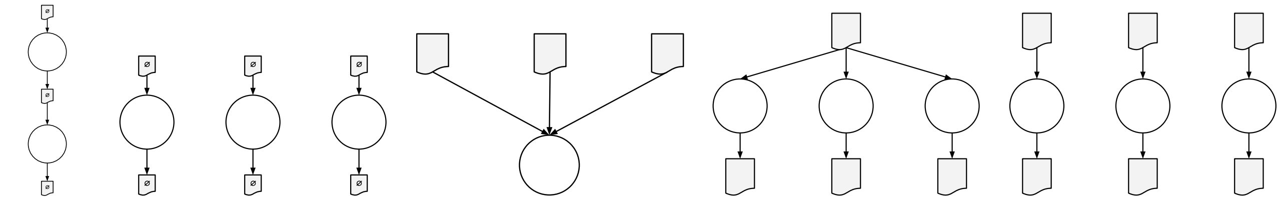


Figure 8: The Five Workflow Benchmark Patterns

run of the simulator with varying input parameters. The benchmark measures how well a system can exploit commonality of input data, typically when S is large and J is less than the number of cores.

Map(J,T,S) consists of J independent jobs running for T time, each of which reads one input file of size S and writes one output file of size S. Maps occur in many transformative workflows, where a series of operations are performed on each of many input files, such as video transcoding or data exploration workflows. The Map benchmark also allows us to measure the aggregate I/O capacity of the system on independent tasks.

5. EVALUATION

We evaluate the Workbench benchmarks on two different physical architectures and five software systems.

The first architecture is a Storage Cluster (SC) that reflects the hardware philosophy of commercial clouds: large numbers of shared-nothing machines with large local disks and cost-effective commodity processors. Our SC cluster consists of 32 machines with 2.4GHz dual-core AMD Opteron CPUs, 4GB of RAM, and 750GB local SATA disks, all connected by a 1Gb/s Ethernet. On this cluster, we evaluated Makeflow using the Local driver (on one machine) and Condor, Hadoop, and Work Queue, which are all designed to exploit the local disks in different ways.

The second architecture is a High Performance Cluster (HPC) that reflects a more traditional cluster architecture: large numbers of machines that share a common central filesystem and have minimal scratch disk on each machine. This cluster consists of 2.6 GHz six-core AMD Opteron CPUs with 1-2GB RAM per core, connected via 10Gb links to a high performance NFS file server. The HPC is a campus-owned resource and thus we are only able to evaluate the installed SGE, along with Local and Work Queue.

The results of these benchmarks are a function of both the physical architecture and the design of the software system in use. As such, we caution the reader not to compare the absolute results across clusters. Rather, the results reveal to what extent each software system can exploit the unique properties of each cluster.

5.1 System Overhead and Throughput

To establish a baseline for I/O operations, we measured the maximum throughput achievable by a single client on each cluster, shown in Table 1.

System	Storage	Read (MB/s)	Write (MB/s)
SC	Local	51.08	61.40
SC	HDFS	24.06	24.04
HPC	Local	110.73	290.50
HPC	NFS	93.14	133.29

Table 1: Basic I/O Throughput

To measure the latency of job submission we ran **Chained**$(128, 0)$ and to measure the throughput of job submission we ran **Concurrent**$(128, 0)$ on all configurations. The results are shown in Table 2.

System	Engine	Latency Sec/Job	Throughput Jobs/Sec
SC	Local	0.012	126.392
SC	Condor	37.807	2.936
SC	Hadoop	17.405	0.648
SC	WorkQueue	0.049	124.000
HPC	WorkQueue	0.016	115.711
HPC	SGE	7.659	6.261
HPC	Local	0.016	229.688

Table 2: Chained and Concurrent Results

The results of these tests demonstrate that the real bound on job throughput is the submission time of the system. Systems such as Work Queue and Local that are controlled and scheduled programmatically by Makeflow have relatively short dispatch times and can push out 100+ jobs/second. In contrast, systems like Condor and Hadoop that use an external scheduler have dispatch times orders of magnitude larger. Constrained as they are by the speed of the system's native dispatcher they can only start at best a few jobs each second.

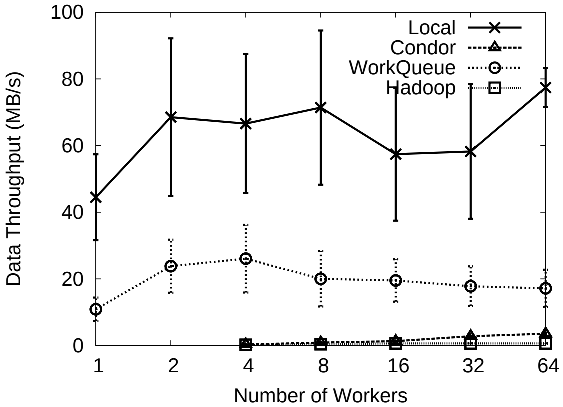
To measure the ability of each system to deliver files to jobs, we run **FanIn**$(1, 0, 1-256, 1\text{MB})$ with the number of files varying in powers of two. The results are shown in Table 3.

Again, we see significant differences in the ability of each system to complete small file transactions. All of the distributed execution systems are an order of magnitude worse than local storage. Work Queue is about three times faster than Condor or Hadoop on the storage cluster, while about half as fast as SGE with NFS on the HPC cluster.

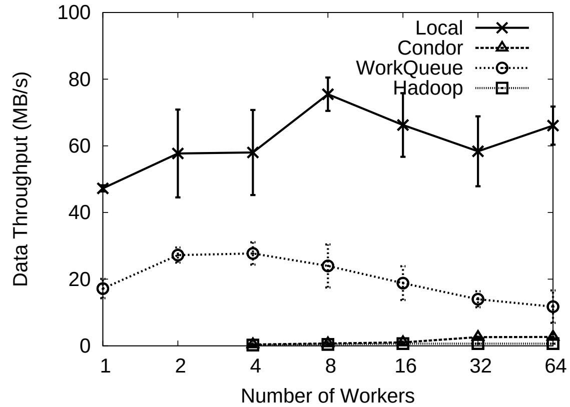
To measure aggregate I/O performance on large workflows, we ran **FanOut**$(128, 0, 1\text{MB}/16\text{MB}/64\text{MB})$ and **Map**$(128, 0, 1\text{MB}/16\text{MB}/64\text{MB})$ on both clusters. The SC results are shown in Figure 9 and the HPC results are shown in Figure 10

The differences between the Local tests on the SC and HPC clusters are not due to hardware differences. Rather, when the data for the test is created it gets cached in memory. On the SC we are able to flush the cache before beginning the tests, so the performance is unaffected by this. On the HPC cluster flushing the cache is not permitted, so the runs on smaller data sizes are aided by the benchmark set-up. Once the data sizes becomes larger than the cache size (see Fig 10c vs Fig 10e) this artificial prefetching does not occur and the performance drops to disk bandwidth rather than memory bandwidth.

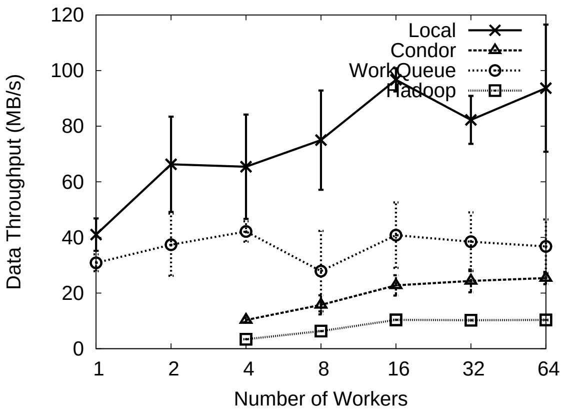
Overall, the results indicate that when dealing with small files (a few MB or less) the systems with minimal overhead and short dispatch times like Local and Work Queue tend to dominate, achieving relatively steady throughput regardless of the number of workers



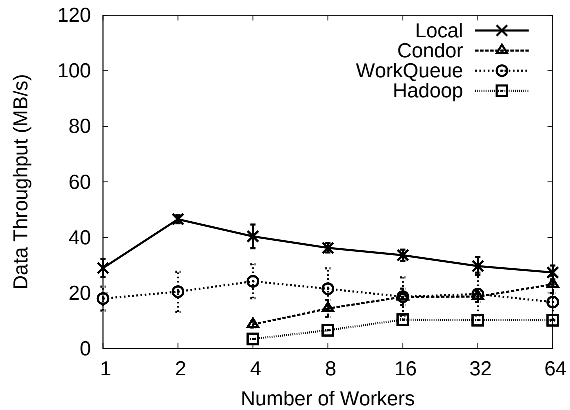
(a) FanOut 1MB



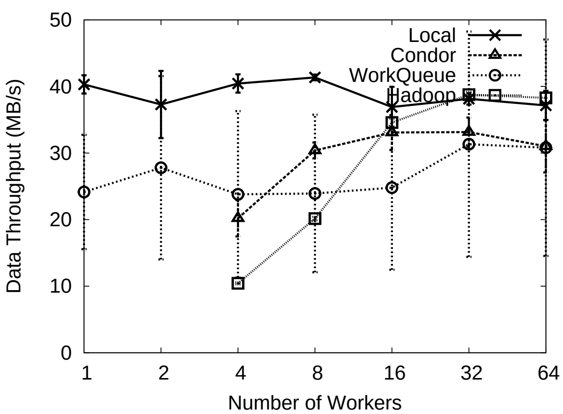
(b) Map 1MB



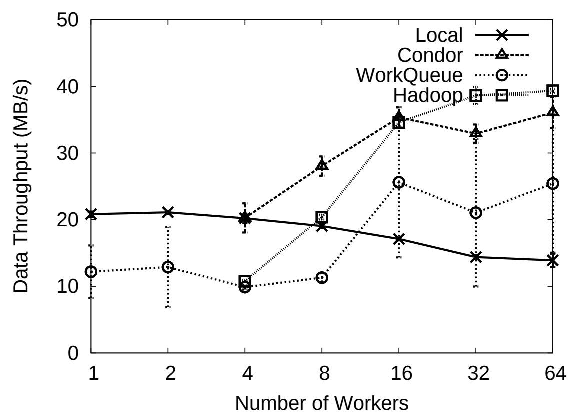
(c) FanOut 16MB



(d) Map 16MB

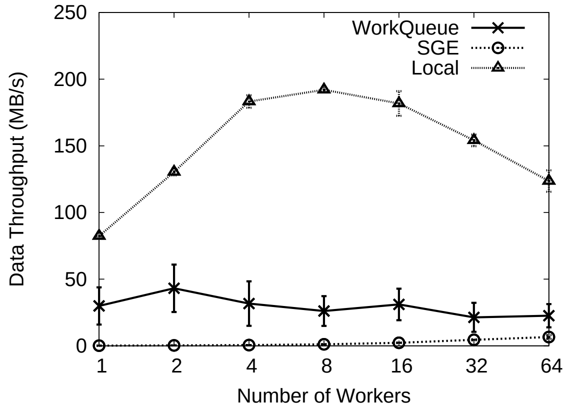


(e) FanOut 64MB

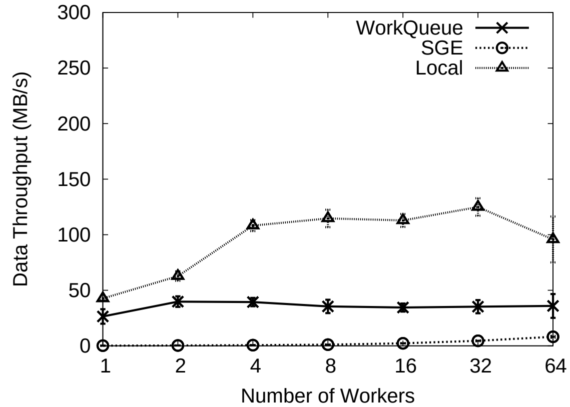


(f) Map 64MB

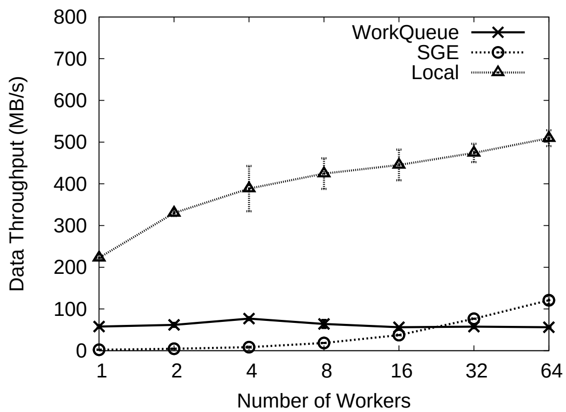
Figure 9: **Storage Cluster: Fan-Out and Map** These graphs display the average and standard deviation of data throughput for data intensive workflows, Map and FanOut. The workers execute the *cat* function on the input to create a copy output file. The data throughput is calculated based on the size of the input data being *read* . The time used to calculate this value also includes the time taken to return output, that is, *write* data. When running the Local engine on this storage cluster, we cleaned the disk cache before execution so all data had to be read from disk. (This is different from the Local engine benchmarks on the HPC cluster.) To keep runtimes reasonable, the benchmarks omit numbers for 1-2 workers on Condor and Hadoop.



(a) FanOut 1MB

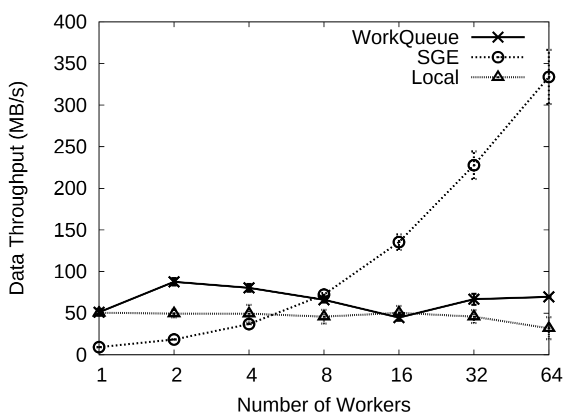


(b) Map 1MB

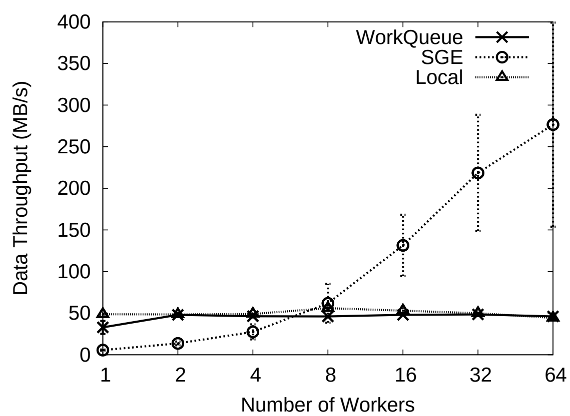


(c) FanOut 16MB

(d) Map 16MB



(e) FanOut 64MB



(f) Map 64MB

Figure 10: **High Performance Cluster: Fan-Out and Map** These graphs display the average and standard deviation of data throughput for data intensive workflows, Map and FanOut. The workers execute the *cat* function on the input to create a copy output file. The data throughput is calculated based on the size of the input data being *read* . The time used to calculate this value also includes the time taken to return output, that is, *write* data. The Local engine on this cluster takes advantage of the hot in-memory cache from moving the data into a local sandbox (/tmp). When the data size is 64 MB, the dataset reaches 8 GB (128 files each 64 MB). At this point the cache can no longer be hot so the performance is indicative of the disk bandwidth. (This is different from the Local engine benchmarks on the SC cluster.)

System	Engine	Files / s
SC	Local	234.07
SC	Condor	5.54
SC	WorkQueue	17.23
SC	Hadoop	5.99
HPC	SGE	23.53
HPC	Local	475.75
HPC	WorkQueue	13.00

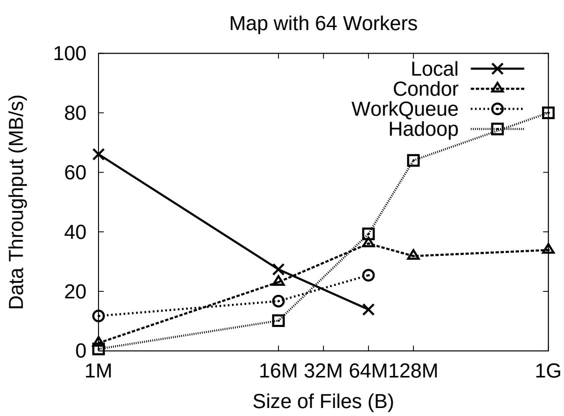
Table 3: FanIn 256 1MB Files

used. In fact, despite the slight decline in data throughput of Work Queue as the number of workers increase, it still outperforms both Condor and Hadoop by a factor of 10 for 1MB files.

As the size of the files increases, the gap closes. The caching that both Local (in memory) and Work Queue (both on disk and in memory on the remote node) can perform in the case of Fan-Out allows them to remain dominant across all worker numbers, but in the case of Map, where each file must be loaded from disk or transferred fresh across the network, both Condor and Hadoop begin to catch up. At filesizes around 16MB Condor consistently achieves better performance than Hadoop, which is expected given Condor's substantially better job throughput for independent jobs.

Once the filesize reaches the largest of these tests we see both Condor and Hadoop overtake Local and Work Queue. At low numbers of workers, Condor still outperforms Hadoop in terms of data throughput, though it plateaus at around 35 MB/s, likely due to the network bandwidth. In contrast Hadoop continues to improve.

Given the performance trend Hadoop began to show we were curious to see how it scaled to larger filesizes. We therefore decided to run the additional tests **FanOut**$(128, 64, 0, 128\text{MB}/1\text{GB})$ and **Map**$(128, 64, 0, 128\text{MB}/1\text{GB})$ on both Hadoop and Condor to see how it performed.



Hadoop's performance increases as the file size gets larger until it begins to plateau around 1GB files. In contrast, Condor maintains stable at about 35 MB/sec. This difference is not really attributable to the locality aspects of Hadoop, as only the first portion of the file is guaranteed to be stored on the same machine. Instead, it is likely that the higher-speed network and limited network congestion present in the Hadoop cluster allows higher speed file transfer between the nodes than Condor achieves across a campus network.

The results on the HPC also demonstrate the dominance of short submission times with low file sizes. Even when compared to SGE, which can fetch all of the files in parallel from the shared filesystem, Work Queue still manages to perform the best of the distributed systems for small filesizes.

SGE's advantage in parallel acquisition of files begins to emerge earlier than Condor or Hadoop. At large numbers of workers for even 16MB files SGE begins to perform better than WorkQueue, reaching twice the data throughput in the **Map**$(128, 64, 0, 16\text{MB})$ case.

At the largest filesizes SGE quickly dominates both other systems. Lacking the bottleneck of sending all of the requisite data through a single master process, SGE is able to satisfy each worker's data dependencies faster and thus achieve much better performance than Work Queue.

6. APPLICATIONS

Makeflow is available as an open source project, and has a growing community of users building scalable applications in fields such as bioinformatics, image procesing, data mining, and molecular dynamics. In this section we will give some examples of non-trivial makeflows and our experience in executing them at large scale.

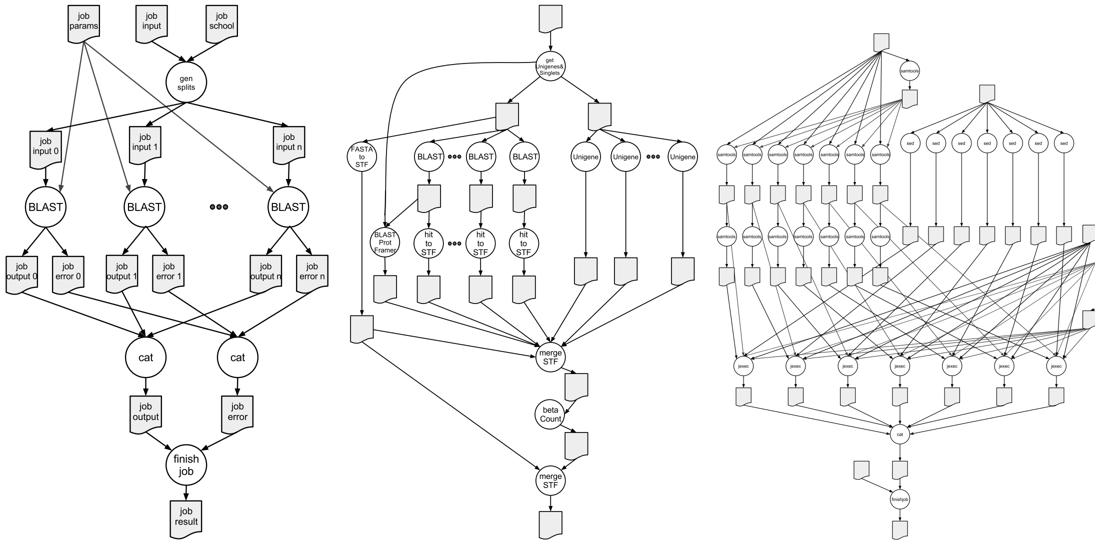
An early adopter of Makeflow was Biocompute [7], which is a bioinformatics data analysis facility at the University of Notre Dame. Users interact with the system via a web portal to select various applications, choose parameters and input files, and then run various data analysis jobs.

The first implementation of Biocompute ran only BLAST [3] jobs using custom-designed scripts to partition the data, distribute it to a limited number of execution nodes, and then execute jobs on each data partitions via Condor. Unfortunately, this first implementation was complex to implement, difficult to modify, and very failure prone because of the large number of parts that interlocked in a complex way. The second implementation of Biocompute switched to Makeflow to express these workloads, which dramatically simplified the implementation, leaving the data management and fault tolerance aspects to makeflow itself. Once this switch was made, it became easy to implementation more complicated workflows harnessing tools such as SHRiMP, SSAHA, SNPexp without developing dedicated tools for each. The infrastructure for submitting, running, monitoring and reporting the status of each job can be maintained independently of the workflows themselves, and new types of workflows can be introduced by merely setting up an interface page and writing a module to generate the necessary workflow. Figure 11 shows a selection of these applications.

BLAST*(Fig 11a)*. The primary use of Biocompute is to allow users to run their BLAST jobs against large reference databases in a reasonable amount of time. Most attempts to parallelize BLAST take one of two approaches. The first approach, taken most famously by mpiBLAST [10], is to segment the database and run the entire set of query sequences against each segment in parallel. The second approach, used by AzureBLAST [20] amongst others, is to distribute multiple copies of the reference database(s) and split up the set of query sequences, running each query sequence against the entire reference database in parallel. Biocompute takes the second approach, relying on prestaged reference databases for performance reasons but allowing any of the underlying systems supported by Makeflow to run the jobs.

EST Pipeline*(Fig 11b)*. One of Makeflow's major advantages is that it is easy to use by researchers who are experts in a field other than distributed systems. One example of this is the work done by Thrasher, et al. [30] in developing a pipeline for the analysis of Expressed Sequence Tags, or ESTs.

The EST pipeline was originally developed as a series of Ruby scripts, each of which had multiple dependencies on Ruby libraries both public (but not usually installed) and custom-built. The pipeline had been designed to be run by hand on a single machine which was



(a) BLAST Diagram (b) EST Diagram (c) SNPExp Diagram

Figure 11: Biocompute Workflows

set up with all of the relevant dependancies. Makeflow, with its low barrier to usage and easy dependency tracking, was able to exploit the fair amount of natural parallelism within the EST pipeline, as well as provide a scalable solution portable between the Condor, SGE, and Work-Queue based systems available at Notre Dame.

The EST Pipeline workflow is fairly complicated and is actually generated by the Weaver workflow compiler [6]. The workflow consists of a pair of fan-out patterns, one of which is mapped to another set of intermediate files, and then combined with the output of the other into a single fan-in. The output from that operation is then run through a few more transformations before being emitted as the EST comparison results.

SNP Exploration*(Fig 11c)*. Another example of a bioinformatics analysis pipeline made easily distributable by Makeflow is the SNP Exploration (SNPExp) pipeline. SNPExp was developed to help researchers identify interesting regions of assembled genomes through analysis of Single Nucleotide Polymorphisms, or SNP's.

The SNPExp pipeline starts by taking the user's job specification and input files and running them through the SAMtools bioinformatics analysis package to create the files necessary for the analysis step. The data is then broken up and sent out to be analyzed by the same SAMtools package in small chunks. Then the results are parsed and collated into a report returned to the researcher. Much of the complexity in the pipeline comes from the preparation steps necessary to massage the input data into a format readable by the analysis package, which were automated by Makeflow rather than requiring the user to perform them by hand.

The SNPExp is a good example of how Makeflow can allow researchers to rapidly prototype complicated workflows for solving intermediate-sized problems. The SNPExp pipeline is of a moderate size, small enough that it could theoretically be run on a single core in a manageable amount of time, but large enough that each test run could take an hour or longer. By using Makeflow to exploit the nascent parallelism, this turnaround time was reduced to minutes, allowing the researcher to quickly test and refine the system without having to worry about most of the complications imposed by distributed systems.

6.1 Real Performance

While Makeflow's simplicity makes it useful as a prototyping tool or a mechanism for domain scientists to write their own distributed applications, it is also useful for running large workflows in an efficient manner. Regularly, Makeflows will run harnessing hundreds or thousands of workers to process hundreds of gigabytes of image, video, and/or genomic data in support of biometrics research, bioinformatics research, and more. Some of the more extreme examples of these workflows can be seen in Figure 12.

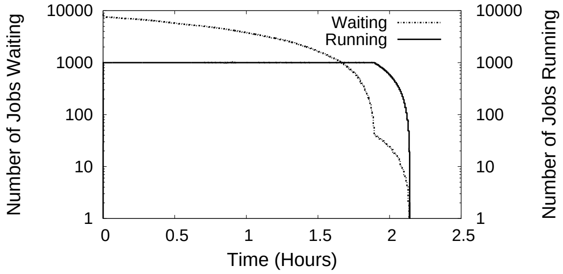
The ND BXGrid repository generates a lot of Makeflow-managed traffic on the campus grid. The workflow in Figure 12a represents a subset of the daily transcoding workflows BXGrid requires. The BXGrid workflows are large, independent map operations which take full-sized images as inputs and generate thumbnails. The given example shows the smooth performance of a large workflow under optimal conditions running on a large portion of the campus grid.

For practical reasons, the BLAST workflows are constrained by the number of nodes we have preinstalled the large BLAST databases on. This limits the number of workers potentially available to any BLAST makeflows but prevents the campus network and storage systems from being overwhelmed. This also means that the largest BLAST workflows can run for days or even weeks, as is the case in Figure 12b. This also means that large BLASTs are subject to system disturbances. Figure 12b provides a good example, as around days 4, 8, 12, and 16 the system hiccups and the number of workers running drops to zero. Makeflow recovers from these disruptions, returns quickly to full operating capacity and completes the job without losing much, if any, work.

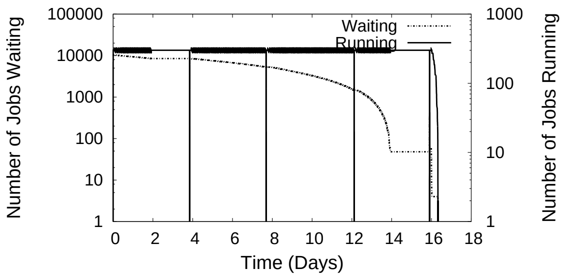
Figure 12c provides a good example of a workflow taxing the boundaries of the systems we have available. The EST pipeline shows a reasonable queueing period before jumping to well over 2000 concurrently running jobs, a number that slowly degrades as the remaining work in the queue is exhausted. The EST Pipeline, by harnessing an extreme amount of parallelism, manages to complete within an hour, and avoids any system failures.

7. RELATED WORK

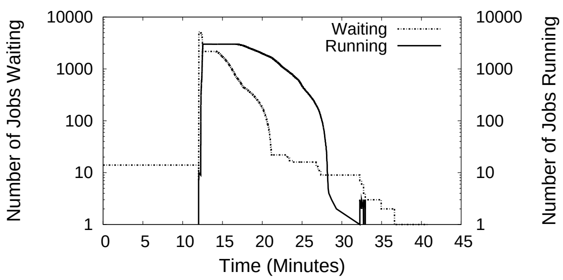
Stu Feldman presented the original Make [13] as a means for maintaining dependencies in compilation. In the years since, there



(a) BXGrid Example



(b) BLAST Example



(c) EST Example

Figure 12: Performance Data

have been many variations on Make that exploit *parallel* computing systems which assume a fault-free environment and a global filesystem namespace, which is required for the traditional Make semantics. For example, Amoeba pmake [5] made minimal modifications to make to allow jobs to be distributed across an Amoeba [21] cluster, relying on the central fileserver for a common namespace. ISIS pmake [24] took the unusual approach of forking commands for all rules simultaneously, relying on a object space similar to Linda [2] to block rules whose dependencies were not yet satisfied. PGMAKE [4] employed PVM to fork processes on a large cluster, and NFS to provide access to data files. The SGE batch system provides qmake [14], which dispatches each rule as an SGE job, again relying on a shared filesystem. GXP make [26] uses the standard GNU make [13], but interposes on the `SHELL` variable to dispatch commands to various remote filesystems, relying on a shared filesystem for data access. Makeflow builds upon this body of work by showing how a slight twist to the semantics allow Make to function correctly on fault-prone, shared-nothing distributed systems constructed from clusters, clouds, and grids.

Explicit statement of dependencies is not the only way of managing workflows. DAGMan [9], the workflow engine provided with Condor [29], states the control dependencies between batch jobs, and has no direct information about data needs. This can be useful if the jobs have side effects that are reflected in some external device such as a database or a physical actuator. Another technique is transparent result caching [32], in which the actual dependencies of a program are observed by interposition on file operations. This is useful for accelerating the performance of a repeated and highly uniform job, but offers no assistance in resource management on the first execution of a workflow. In a cloud environment where data movement has real costs, we believe that explicit statement of dependencies is of greater utility.

In recent years, cloud computing systems have emphasized the Map-Reduce programming model, popularized by Google [11] and the Hadoop [17] open source implementation of the same ideas. Map-Reduce is, by design, less flexible than a generalized graph programming model, but this makes it much more tractable to solve problems of data partitioning, data locality, and fault tolerance. The concept of Map-Reduce is portable to other architectures such as multicore [25] and graphics processors [18]. Systems such as Dryad [19] and Sphere [16] provide a more generalized graph processing model than Map-Reduce, but also focus on the storage cluster architecture.

It is common to view Map-Reduce programs from the perspective of query processing. A number of small domain specific languages have been created to simplify multiple invocations of Map-Reduce for this purpose. For example, Pig [23] consists of a series of simple query-like operators that are all implemented as calls to Map-Reduce. Cascading [8] is a similar idea, but is able to pipeline multiple operations together for higher efficiency. Hive [31] layers a table structure on top of files within Hadoop, and then applies Map-Reduce programs to implement a query language.

8. CONCLUSIONS

Makeflow. Many common problems in science and industry can be easily expressed as a graph of small sequential programs exhibiting a high degree of natural parallelism. These can be run on a variety of dedicated systems, but each requires the user to have some expertise with a custom language and runtime system. This makes it difficult for users to learn new systems, limiting the resources available to them.

To solve this problem we have introduced Makeflow, a fault tolerant workflow manager based on the traditional Unix Make. Makeflow uses Make syntax, modified only by requiring explicit data dependencies, to describe workflows. Makeflow provides both portability amongst many execution systems and a language whose basic syntax is already well known to most potential users. We've shown many examples of complex workflows developed by users expert in their field but novices in distributed programming, with performance comparable to custom-designed distributed solutions.

Workbench. Makeflow's inherent portability has also made it easy to run the same workflow on each of the supported systems which prompted our introduction of Workbench. Workbench is a workflow benchmark suite whose purpose is to measure the performance characteristics of common workflow patterns on a variety of systems. When running Workbench we observed that workflows with small files are most affected by dispatch latency rather than data throughput. We also noticed that asynchronous file transfer provides the greatest benefit to workflows with large filesizes and large numbers of workers, but that for low-data jobs the infrastructure necessary to run the job slows dispatch to the point of reducing performance. Workbench is also useful for characterizing the performance of new systems and highlighting unexpectedly good or bad performance in specific cases for individual systems.

Future Work There are a variety of directions we can take this work. Makeflow's knowledge about the data dependencies of a workflow offers many exciting opportunities to optimize execution. Data-local computation, resource management, and graph decomposition are three areas we've already begun investigating.

9. REFERENCES

[1] Filesystem in user space. http://sourceforge.net/projects/fuse.

[2] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, August 1986.

[3] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 3(215):403–410, Oct 1990.

[4] Andrew Lih and Erez Zadok. PGMAKE: A Portable Distributed Make System. Technical Report CUCS-035-95, Computer Science Department, Columbia University, 1994.

[5] E. H. Baalbergen. Design and implementation of parallel make. *COMPUTING SYSTEMS*, 1:135–158, 1988.

[6] P. Bui, L. Yu, and D. Thain. Weaver: Integrating Distributed Computing Abstractions into Scientific Workflows using Python. In *Challenges of Large Applications in Distributed Environments at ACM HPDC 2010*, 2010.

[7] R. Carmichael, P. Braga-Henebry, D. Thain, and S. Emrich. Biocompute: Toward a Collaborative Workspace for Data Intensive Bio-Science. In *Workshop on Emerging Computational Methods for Life Sciences at ACM HPDC 2010*, pages 489–498, 2010.

[8] Cascading. http://www.cascading.org/, 2010.

[9] Condor Team. The directed acyclic graph manager. http://www.cs.wisc.edu/condor/dagman, 2002.

[10] A. E. Darling, L. Carey, and W. chun Feng. The design, implementation, and evaluation of mpiblast. In *In Proceedings of ClusterWorld 2003*, 2003.

[11] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large cluster. In *Operating Systems Design and Implementation*, 2004.

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

[13] S. Feldman. Make – A Program for Maintaining Computer Programs. *Software: Practice and Experience*, 9:255–265, November 1978.

[14] W. Gentzsch. Sun grid engine: Towards creating a compute power grid. In *CCGRID '01: Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, 2001.

[15] S. Ghemawat, H. Gobioff, and S. Leung. The Google filesystem. In *ACM Symposium on Operating Systems Principles*, 2003.

[16] Y. Gu and R. L. Grossman. 1 Sector and Sphere: The Design and Implementation of a High Performance Data Cloud.

[17] Hadoop. http://hadoop.apache.org/, 2007.

[18] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: a mapreduce framework on graphics processors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 260–269, New York, NY, USA, 2008. ACM.

[19] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data parallel programs from sequential building blocks. In *Proceedings of EuroSys*, March 2007.

[20] W. Lu, J. Jackson, and R. Barga. Azureblast: a case study of developing science applications on the cloud. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 413–420, New York, NY, USA, 2010. ACM.

[21] S. Mullender, G. van Rossum, A. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba: A distributed operating system for the 1990s. *IEEE Computer*, 23(5):44–53, 1990.

[22] T. Oinn, M. Greenwood, M. Addis, M. N. Alpdemir, J. Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, D. Marvin, P. Li, P. Lord, M. R. Pocock, M. Senger, R. Stevens, A. Wipat, and C. Wroe. Taverna: lessons in creating a workflow environment for the life sciences: Research articles. *Concurr. Comput. : Pract. Exper.*, 18:1067–1100, August 2006.

[23] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110, New York, NY, USA, 2008. ACM.

[24] A. Polze. Using the object space: A distributed parallel make. *4th IEEE Workshop on Future Trends of Distributed Computing Systems*, 1993.

[25] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *In HPCA âĂŹ07: Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, pages 13–24. IEEE Computer Society, 2007.

[26] K. Taura, T. Matsuzaki, M. Miwa, Y. Kamoshida, D. Yokoyama, N. Dun, T. Shibata, C. S. Jun, and J. Tsujii. Design and implementation of gxp make – a workflow system based on make. *IEEE Conference on eScience*, 2010.

[27] D. Thain and M. Livny. Parrot: An Application Environment for Data-Intensive Computing. *Scalable Computing: Practice and Experience*, 6(3):9–18, 2005.

[28] D. Thain and C. Moretti. Abstractions for Cloud Computing with Condor. In S. Ahson and M. Ilyas, editors, *Cloud Computing and Software Services: Theory and Techniques*, pages 153–171. CRC Press, 2010.

[29] D. Thain, T. Tannenbaum, and M. Livny. Condor and the grid. In F. Berman, G. Fox, and T. Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley, 2003.

[30] A. Thrasher, R. Carmichael, P. Bui, L. Yu, D. Thain, and S. Emrich. Taming Complex Bioinformatics Workflows with Weaver, Makeflow, and Starch. In *Workshop on Workflows in Support of Large Scale Science*, pages 1–6, 2010.

[31] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy. Hive - a petabyte scale data warehouse using hadoop. In *International Conference on Data Engineering*, pages 996–1005, 2010.

[32] A. Vahdat and T. Anderson. Transparent result caching. *Proceedings of the 1998 USENIX Technical Conference*, 1998.

[33] L. Yu, C. Moretti, A. Thrasher, S. Emrich, K. Judd, and D. Thain. Harnessing Parallelism in Multicore Clusters with the All-Pairs, Wavefront, and Makeflow Abstractions. *Journal of Cluster Computing*, 13(3):243–256, 2010.

[34] Y. Zhao, J. Dobson, L. Moreau, I. Foster, and M. Wilde. A notation and system for expressing and executing cleanly typed workflows on messy scientific data. In *SIGMOD*, 2005.