Title
Tigres Workflow Library: Supporting Scientific Pipelines on HPC Systems

ISBN
978-1-5090-2453-7

Authors
Hendrix, Valerie
Fox, James
Ghoshal, Devarshi
et al.

Publication Date
2016-05-01

DOI
10.1109/ccgrid.2016.54

Peer reviewed

Tigres Workflow Library: Supporting Scientific Pipelines on HPC systems

Valerie Hendrix, James Fox, Lavanya Ramakrishnan
Lawrence Berkeley National Laboratory
Berkeley, CA
Email: {vchendrix, jsfox, lramakrishnan}@lbl.gov

***Abstract*—The growth in scientific data volumes has resulted in need for new tools that enable users to operate on and analyze data on large-scale resources. In the last decade, a number of scientific workflow tools have emerged. These tools often target distributed environments and often need expert help to compose and execute the workflows. Data-intensive workflows are often ad-hoc and involve an iterative development process that includes users composing and testing their workflows on desktops and scaling upto larger systems. In this paper, we present the design and implementation of Tigres, a workflow library that supports the iterative workflow development cycle of data-intensive workflows. Tigres provides an application programming interface to a set of programming templates (i.e., sequence, parallel, split, merge) that can be used to compose and execute computational and data pipelines. In this paper, we a) present the design and implementation of Tigres b) evaluate number of scientific and synthetic workflows to show Tigres performs with minimal template overheads (mean of 13 seconds over all experiments) and c) study the factors that affect the performance of scientific workflows on HPC systems.**

I. Introduction

Scientific collaborations and experiments are increasingly generating large data sets that need to be processed. Scientists often develop their algorithms on their desktops but it is no longer practical to download the data to a desktop to operate on them. Large scale systems such as High Performance Computing (HPC) centers and clouds are needed to run these workflows. With current technologies, scaling an analysis to an HPC center is difficult even for experts.

We need tools that can support the iterative development process of data-intensive workflows that allow easy composition and seamless execution on multiple platforms. In this paper, we present Tigres that addresses the needs of data-intensive workflows. Tigres is a programming library that allows one to compose large-scale scientific workflows in a programming language and execute on multiple platforms including desktops and supercomputers.

Tigres addresses the challenge of enabling collaborative analysis of scientific data through a concept of reusable *"templates"* that enable scientists to easily compose, run and manage collaborative computational tasks. These templates define common computation patterns used in analyzing a data set. Tigres currently supports four templates sequence, parallel, split and merge. Tigres can run on a variety of different platforms including desktops, clusters and supercomputers and supports various execution mechanisms including thread, process and distribute.

Workflows from various scientific domains including astronomy, bioinformatics and earth sciences have been composed in Tigres. These workflows vary in complexity as well as computational, memory, I/O and storage requirements; they were composed from existing executable scripts and binaries.

Scientists usually start with existing binaries or functions and realize they need more workflow style capabilities. Tigres is designed to allow users to *design* workflows, *develop* using compose-able templates, *run* initial designs on their desktops and receive *feedback* that will inform further workflow design. This cycle of design, develop, run and feedback into further design is *iterative workflow development*. Tigres is implemented in python. The focus of this paper is python workflow implementations.

In this paper, our contributions are a) the design and implementation of Tigres workflow library, b) evaluation of three scientific applications and various synthetic workflows that show Tigres has minimal overheads c) study of the factors that affect the performance of scientific workflows on HPC systems.

The rest of this paper as organized as follows. In Section II, we discuss related work. In Section III, we describe Tigres design and implementation. We layout our experiment setup, workflows, evaluation metrics as well as discussing the experiment results in Section IV. We present our conclusions in Sections V.

II. Related Work

There are a number of workflow tools for building scientific workflows (e. g. Kepler [1], Taverna [2], Pegasus [3], Galaxy [4], Trident [5], Makeflow [6] and Triana[7]) They come with tools for data movement, orchestrating compute and data-centric tasks, provenance tracking, monitoring progress and error handling.

Fireworks [8], Swift [9] and Qdo [10] support executing workflows on HPC. Fireworks is a high throughput workflow system designed for HPC but needs to be installed and managed by an expert. Swift, a parallel scripting language, requires learning a domain specific language. Qdo is a python tool designed to combine a many small tasks into a single HPC batch job but lacks constructs to chain the batch jobs together into a workflow.

The HPC community has been utilizing common patterns [11] such as those in the thirteen dwarfs [12] and Hoare's Communicating Sequential Processes [13]. OpenMP [14] and Unified Parallel C (UPC) [15] used for share-memory programming adhere to the Message Passing Interface(MPI) [16] standard; they require some effort to master the common data-analysis operations (e. g. scatter, gather, reduction).

Tigres *templates* are inspired by concepts introduced by the MapReduce [17] programming model. MapReduce is good for handling big parallel data analysis but falls short for scientific workflows [18], [19]. Workflow technologies that support building a sequence of MapReduce jobs, such as CloudWF [20] and Oozie [21], don't offer more patterns beyond MapReduce. Spark [22] supports MapReduce tasks with it's Resilient Distributed Datasets (RDD) APIs allowing faster access to data in memory.

The Tigres library captures the common programming patterns at a higher-level than the aforementioned models in a programming interface. Many of these previously mentioned tools require expert knowledge for setting up and writing initial workflows and execution is tied to one set of resources. Also, taking a set of existing binaries and scripts and transforming them into a workflow is not always straightforward or possible in other tools.

III. Architecture

Figure 1 shows that Tigres has five major components in a layered architecture: *Templates API*, *Base API*, *State Management*, *Execution Management* and *Monitoring*.

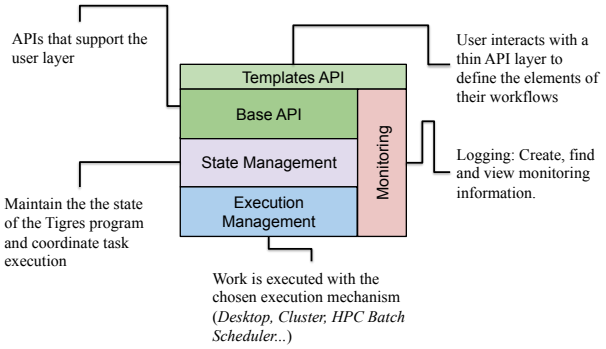


Fig. 1: Tigres provides a thin templates API to define the elements of workflows, log messages and monitor their program execution.

The users interacts with the user layer where users work with basic templates, manage template and task dependencies and monitor workflow progress. The user layer in turn interacts with the inter layers for state, execution and monitoring.

A. Basic Templates

The Tigres templates API allows users to programmatically create *workflows* using the *basic templates* as building blocks. A Tigres *workflow* is a python program that uses the Tigres templates API to build and execute a workflow. There are four basic Tigres template functions: *sequence*, *parallel*, *merge* and *split*. The Tigres program can contain one or more of these template functions. *Templates* are composed of individual *tasks* that are units of work from the end-user that needs to be executed.

Table I defines the Tigres data model. A *Task* is the most basic unit of execution and can be defined as a python function internal to the Tigres program or a separate executable. The inputs to each *Task* execution can be statically defined or retrieved from the results of previously executed *Tasks* or *Templates*.

Task	The atomic unit of execution. (*tigres.Task*)
Templates	A collection of tasks. (*tigres.TaskArray*)
Task Array	Patterns of execution from a combination of tasks. (*tigres.TaskArray*)
Input Types	The characteristic of the task the defines the type of the inputs. (*tigres.InputTypes*)
Input Values	The values used in a task execution. (*tigres.InputValues*)
Input Array	A collection of input values for a number of tasks. (*tigres.InputArray*)

TABLE I: Tigres data model

Each *template* function minimally takes two named collections: *Task Array* and *Input Array*. The *Task Array* is an ordered collection of tasks to be executed together, in sequence or parallel depending on the execution flow of the particular template. The *Input Array* defines the inputs for each *Task* in the corresponding *Task Array* and is a collection of *Input Values*.

The *Task*, the atomic unit of execution in Tigres, has a collection of *Input Types* that specifies the type of inputs a task may take. A task's *Input Values* is an ordered list of task inputs and are passed to the task during execution. They are are not included in the task definition which allows for task reuse and late binding of data elements to the Tigres program execution.

B. Dependencies

Tigres has both data and execution dependencies. The execution dependencies are simple and automatic. Each template is executed in the the order it appears in the workflow. The data dependencies are defined by the user and can be either implicit or explicit in nature.

Program in execution dependencies. The subsequent templates must wait for the previous template to finish its execution. Tasks inside a template have the same behavior. A single task must complete before the next task or set of parallel tasks is executed. Similarly a set of parallel tasks must complete before the next task is executed.

Tigres uses a special syntax called *PREVIOUS* syntax to create implicit and explicit data dependencies between tasks. The user can specify the output of a previously executed task as input for a subsequent task.

Implicit Data Dependencies. Data dependencies are implicit when there are no input values for a task execution. For example, a template uses the entire output of the immediately preceding task or template if any of the tasks inside are missing input values. A set of parallel tasks, as in merge, split or parallel templates, that are missing inputs will only use the

immediately preceding outputs if the results from the previous task or template is a list and can be iterated over.

Explicit Data Dependencies. Explicit data dependencies in Tigres are defined using the *PREVIOUS* syntax. This syntax allows the user to reference data output before it has been created and to define data dependencies between a task and any of its preceding tasks. Additionally, if the output is from parallel or split template, the *PREVIOUS* syntax provides a way to index into the parallel results list.

C. Execution Management

The execution layer can work with one or more resource management techniques including HPC and cloud environments. In addition, the separation of the templates API and the execution layer allows us to leverage different existing workflow execution tools while providing a native programming interface to the user.

Tigres can be executed in several different environments from batch queues to local threads and processes. By using the appropriate *execution plugin*, a Tigres program can be executed on a single node or deployed without additional infrastructure to department clusters and batch processing queues on supercomputers. A program is written once and only the execution plugin is changed at run time. This allows users to easily scale from development (desktop) to production (department clusters and HPC centers).

Tigres currently supports four execution plugins: local threads, local processes, distribute processes, job manager.

Distribute, process and thread plugins use the relevant Python concurrent programming libraries. Thread and process are limited to executing on a single node while distribute can scale across several nodes. Each execution mechanism launches a worker for each core on a node.

Thread workers use Python *threading*. Python threads share data with each other. There is little overhead on start up. Python threads are real POSIX threads (pthread) but Python's Global Interpreter Lock (GIL) only allows a single thread at a time to execute in the interpreter at any one time. There is no real concurrency with Python threads. A thread must acquire the GIL in order to execute.

Process workers use Python *multiprocessing* processes. Python processes do not share any data implicitly and avoid the GIL bottleneck by using subprocesses instead of pthreads. Processes are slower to initialize because it must create and maintain its own address space. Data shared among processes must be serializable. Process execution uses Python *JoinableQueue* for both the task and results queue which use pipes internally to transmit data.

Distribute workers use Python *multiprocessing* manager for distributing Python processes across multiple nodes. This mechanism shares data by having a server which manages shared queues among clients. The clients which launch the worker processes access the shared data on the server with a proxy to the shared data. The client processes communicate with the server through Transmission Control Protocol (TCP) protocol.

Manager plugins support Sun Grid Engine (SGE) and Simple Linux Utility for Resource Management(SLURM). Manager execution submits tasks as manager jobs. The evaluation in the paper does not include these execution plugins.

D. Monitoring and Logging

Monitoring is used by most Tigres components to log system or user-defined events and to monitor the program with a set of entities for querying those events. All the monitoring information, both automatic and user-provided, is semi-structured, i. e. it is broken into name/value pairs. In general, the monitoring follows the Logging Best Practices [23] that arose from the NetLogger [24] project.

Monitoring information in Tigres is produced at two levels: system and user. All timestamped log events are captured in a single location (a file). The user is provided with an API for creating user-level events, checking for the status of tasks or templates and searching the logs with a special query syntax. System-level events provide information about the state of a program such as when a particular task started or what is its latest status (e.g., did task x fail?).

E. State Management

State management encompasses different execution management aspects of the workflow. For instance, it validates the user input to confirm they result in a valid workflow both prior to start as well as during execution. It maintains state as each template is invoked and integrity of the running Tigres program.

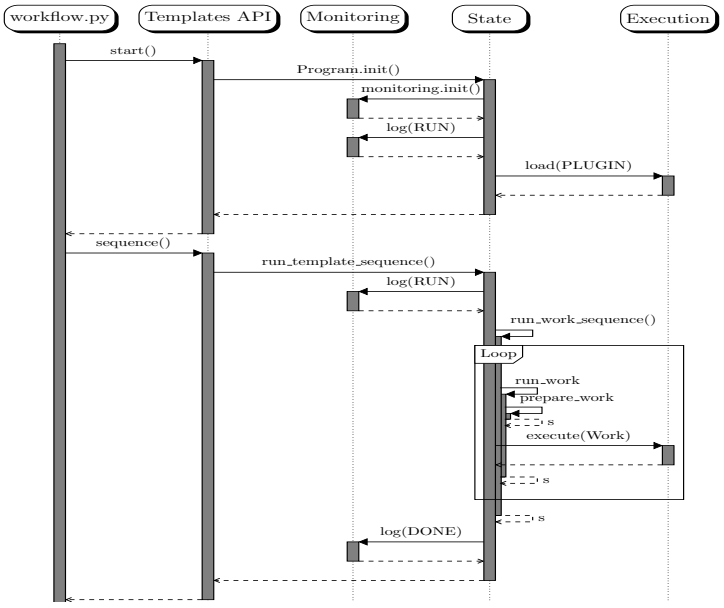


Fig. 2: Event sequence diagram of a Tigres workflow execution

Underneath the hood Tigres translates tasks and templates from the user-facing API into *Work* objects (Table II). Work objects used by the Tigres state management layer models to track a workflow's progress. There are three types of Work classes: *WorkUnit*, *WorkSequence* and *WorkParallel*.

The most basic class is WorkUnit which represents an individual task with inputs. Task state (i.e., RUN DONE FAIL) and results are stored in WorkUnit. The execution plugin receives the WorkUnit contents, executes the defined task and then updates it with the results (e.g, task x's output) and state (i.e., DONE FAIL). WorkParallel represents parallel execution of tasks. It contains an ordered list of WorkUnits. The status of a Work class is determined from the WorkUnits. WorkSequence represents a sequential execution of tasks. It contains an ordered list of WorkUnits in the case of sequence. For split and merge is contains one WorkUnit and one WorkParallel.

Figure 2 is a simplified sequence of events for a Tigres workflow (workflow.py) composed of a single sequence template. It shows the interaction between the workflow and the Tigres components (templates API, monitoring, state and execution management) by walking through workflow initialization (*start*) and sequence template execution (*sequence*). We omit creation of Tasks (using Templates and Base APIs) and corresponding *Work* objects (State Mgmt) for simplicity. Workflow finalization is left out in the interest of space.

start(): Workflow initialization is either explicitly invoked by the user code or implicitly when the first Tigres object (e.g Task, InputValue,...) is created. At this time state management, creates the workflow *Program* object, initializes monitoring, logs the workflow in a RUN state and loads the requested execution plugin. *Program* caches all the *Work* executed by the workflow.

sequence(): After all tasks have been prepared in the workflow, they are passed to the sequence template. The templates API calls upon state management layer which marks the template in a RUN state via monitoring. During *run_work_sequence* the Work objects are created from the tasks and input values and then they are iterated over and run (*run_work*). Inside the *Loop*, the inputs are validated and any results from previous tasks or templates are evaluated. If this completes successfully, the execution plugin is invoked and waits for task completion. If the work completes successfully the template is marked in a DONE state.

WorkBase	Base Work object inherited by all.
WorkUnit	Represents an individual task with inputs and it's state (READY,RUN,DONE,FAIL) and results.
WorkParallel	Represents parallel execution of tasks. It contains an ordered list of WorkUnits. State is determined from the WorkUnits.
WorkSequence	Represents a sequence execution. It contains an ordered list of WorkUnits and WorkParallel.

TABLE II: Tigres Work classes

IV. Evaluation

A. Experiment Setup

System. All workflows were executed on Edison, a Cray XC30 system at the National Energy Research Scientific Computing Center (NERSC). The workflows, supporting libraries, input and output data used Edison's local filesystem (*scratch*) where possible. Edison has a peak performance of 2.57 petaflops/sec, 133,824 compute cores and 357 terabytes of memory. All python code was executed with Python 2.7.9.

We also used NERSC's global *scratch* and *project* filesystems as specified. Global *scratch* and *project* are based on IBM's General Parallel File System (GPFS) and available on most NERSC systems. Global *scratch* can temporarily store large amounts of data and Global *project* is permanent storage shared across a team.

The workflows were run on *single-core*, *multi-core* or *multi-node*. Single-core is used by workflows with only *sequence* templates. Multi-core workflows executed on a single Edison node with 24 cores. Our multi-node workflows ran on 2 to 75 nodes (i.e., 48 to 1800 cores). In the case of multi-core and multi-node compute resources, the concurrent task execution is equivalent to the total number of available cores.

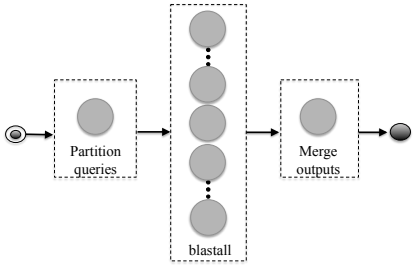
Workflows The workflows in the experiments are a combination of actual science applications and those with synthesized characteristics. The synthetic workflows are divided into two main characteristics: I/O and compute bound. Additionally, these workflows were designed to exercise all Tigres templates (sequence, parallel, merge, split).

BLAST BLAST, a bioinformatics application, allows comparison of biological sequences for different proteins against a sequence database. We measured input size by the total number of database protein sequence queries between 7500 and 45,000, in this case. Data was partitioned into files that contained 25 sequences each based on previous studies that shows bunching of inputs helps with performance [25]. Each task execution received a single input file which was used to query against the same reference database.

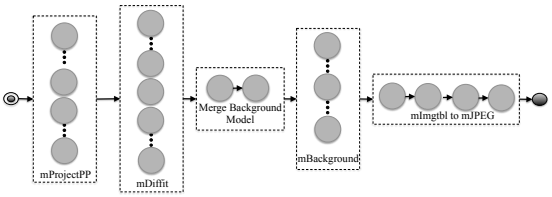
Figure 3a shows the BLAST workflow. It consists of two single sequence templates and a parallel template in-between. The first sequence template runs a single task, a shell script, that partitions an input sequences file into smaller files of 25 sequences each. The parallel template then runs NCBI's BLAST application, blastall on these smaller files and stores the results in-memory. Finally, the sequence template runs a single a task, a Python function, to reduce all the outputs into a single output file, finishing the workflow. All data sets are on Edison's *scratch* file system unless noted otherwise.

Montage. Montage is a software toolkit for assembling Flexible Image Transport System (FITS) images into custom mosaics. We implemented the Montage workflow in Tigres as documented in previous literature [26], [27]. Our test cases assembled images of sky survey M17 on band j from the Two Micron All Sky Survey (2MASS) Atlas images. FITS input files were pre-fetched to the the *scratch* filesystem. The main input to the workflow is used to control coverage of the sky by square degrees.

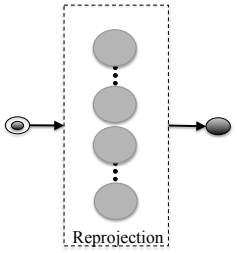
Figure 3b shows the Montage workflow. It consists of three parallel and two sequence templates. The sequence templates, *Merge Background Model* and *mImgTbl to mJPEG*, maintain their task counts of two and four respectively as the square sky degree is increased. The parallel templates, *mProjectPP*, *mDiffFit* and *mBackground*, task counts increase as the degree increases. Task implementations are compiled C programs that



(a) **BLAST workflow** performs queries through NCBI's BLAST application ("blastall")



(b) **Montage workflow** assembles an image for survey M17 on Band j from the 2MASS Atlas images.



(c) **CAMP workflow** processes NASA MODIS satellite data.

Fig. 3: Tigres Workflows

take FITS files as both input and output. The final task takes a FITS input and outputs a JPEG of the assembled image.

CAMP. Community Access MODIS Pipeline (CAMP) [28] is a toolkit for reprojecting satellite data products from NASA's Moderate Resolution Imaging Spectroradiometer (MODIS) instrument. A CAMP Tigres workflow, shown in Figure 3c, uses a single parallel template, called Reprojection, to process NASA MODIS satellite data. Each task is a python script that reprojects a day's observations for a specific location on the earth. Tigres was used to execute CAMP workflows in production and the data presented here is from those runs. The parallel template converts daily satellite observations into 194 land locations for an entire month which is ~5800 total parallel tasks for a single workflow execution.

The database was located on Edison's local *scratch* filesystem and the observation files were on a global *scratch*. The resulting output files were written to Edison *scratch* filesystem.

Synthetic Workflows We cover two main types of synthetic workflows: I/O and compute bound. Both workflows are implemented with each of the four Tigres templates (sequence, parallel, merge, split). In the case of the merge, split and parallel templates the I/O and light compute tasks are parallelized. The number of tasks in the experiments is equivalent to the number of physical cores requested. Table III summarizes the four synthetic template versions.

Synthetic Compute (SC) workflows consist mainly of a single task that performs five million multiplication operations on a random set of inputs. Each SC template version was evaluated for three task implementations: C executable, python function and python executable.

Synthetic I/O (SIO) workflows' main task uses the linux *dd* program to write files of varying sizes. Table III describes the four template versions. Each SIO template version was evaluated for varying file sizes.

B. Metrics

The metrics used in our experiment evaluations are *workflow turnaround*, *template turnaround*, *template overhead* and *task time*. All metrics are mined from the Tigres logs for start and end times.

Workflow turnaround is the time for the execution of the entire workflow inside the Tigres python program. It includes the execution between *start()* and *end()* which maybe implicitly or explicitly invoked.

Parallel	Prepares inputs and invokes a parallel template. A random list of integers is initialized for the tasks in SC workflow and inputs are prepared for task dd, that is executed in parallel.
Merge	Prepares inputs and executes a merge template. A Tigres merge template executes a parallel template followed by a single task. SC merge task sums the results of the merge parallel and SIO reads the outputs from the parallel I/O tasks combines them into a single file.
Split	Prepares the inputs in the split task for the parallel tasks. A Tigres split template executes a single split task followed by a parallel template. The split workflow generates the inputs in the split task instead of outside the workflow as in the Parallel and Merge workflows.
Sequence	Performs the tasks in sequence. SC computes a result in each task and then passes the result to the next task. SIO uses dd for each task to write a file.

TABLE III: Summary of Synthetic Workflow Template Implementations.

Template turnaround is the time for the execution of a single Tigres template. This time includes the execution inside a template function (*sequence()*, *parallel()*, *split()*, *merge()*).

Template overhead is the time during template execution that no tasks are running. This includes the time it takes for Work object initialization, execution plugin setup and task preparation. It is the time before the first task starts, after the last task ends and time between a sequence of tasks as in sequence, merge and split templates.

Task time is the time for the execution of a single template task. It includes Tigres execution mechanism overhead (distribute, process, thread) plus the actual wall time of the task execution.

C. Template Overheads

Template overhead was measured for all application experiments (BLAST, CAMP, Montage, Synthetic).

Figure 4 shows template overhead (in seconds) by varying task count. As template task count increases so does the template overhead. The figure compares distribute, process and thread execution for all workflow experiments. Distribute mechanism has higher template overhead compared to the other methods with lots of variability as seen by the numerous data points above 100 seconds. Distribute mechanism has

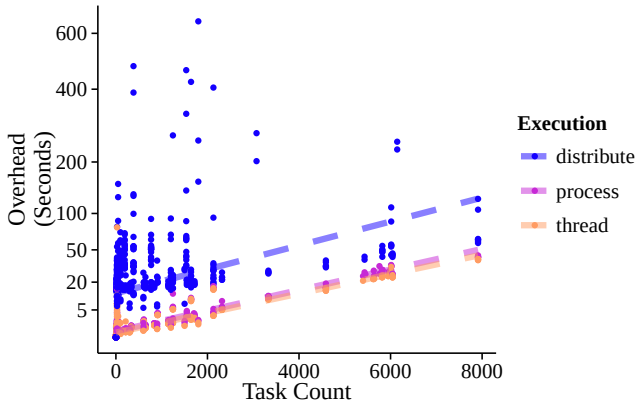


Fig. 4: **Template Overhead by Task Count** comparing distribute, process and thread execution for all experiments. Template overhead increases with task count. Distribute mechanism due to its high setup and communication costs has the most overhead and variability in overhead.

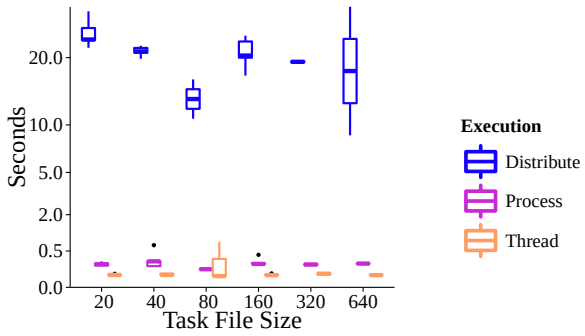


Fig. 5: **Synthetic I/O Multi-core Parallel Template Overhead** by task file size (MB) comparing distribute, process and thread execution. It is demonstrated here that when task count is consistent and file size is changed there is little demonstrated affect on template overhead.

higher setup and communication costs than thread and process mechanisms as discussed in Section III-C. Figure 4 also shows that process execution has consistently higher template overhead than thread.

Distribute execution has higher variability in template overhead for all task counts. Since all templates should have similar costs for internal task initialization we infer that this reflects the variability of the network communication versus those of pipes (process) and shared address space(thread).

Table IV shows the mean template overhead times (mean, median, max and min) for Synthetic I/O and sequence templates. All overheads, no matter the number of tasks or task implementation is less than a one second with one exception. Python executable task implementation shows a max overhead of ~3 seconds. Figure 5 shows template overhead for Synthetic I/O multi-core workflows by task file size comparing distribute, process and thread execution for parallel templates. This box plot demonstrates that when task count is consistent and file size being written per task increases that there is no

Task Impl	Mean	Median	Max	Min
C Exe	179 ms	154 ms	362 ms	66 ms
Python Exe	375 ms	125 ms	2972 ms	65 ms
Python Fn	174 ms	143 ms	368 ms	69 ms

File Size	Mean	Median	Max	Min
20 MB	9 ms	9 ms	10 ms	5 ms
40 MB	9 ms	9 ms	12 ms	5 ms
160MB	194 ms	9 ms	929 ms	6 ms
200 MB	18 ms	12 ms	29 ms	11 ms
400 MB	11 ms	12 ms	14 ms	7 ms

TABLE IV: **Synthetic Sequence Template Overhead** is negligible as evidences by times mostly under 1 sec.

demonstrated effect on template overhead. Similar behavior was seen for merge and split (Figures not shown due to space constraint)

In general, it was found that scaling up the parallel tasks increased the template overhead. We observed that template overheads are minimal with a large variance for distribute execution. We observe a mean template overhead of 13 seconds over all our experiments.

D. Comparison of Execution Mechanisms

CAMP Each CAMP workflow execution had different inputs which results in different execution times. Thus, it is not practical to directly compare turnaround time across all runs. However, we can compare execution mechanisms by a subset of task *types*.

The place on the earth (sinusoidal tile) that a task is acting on determines the number of input files processed. This in turn increases the memory and computation requirements. A sinusoidal tile is composed of longitude (`hN`) and latitude (`vN`) where `N` is an integer. For our analysis, four latitude ranges of sinusoidal tiles were chosen with two tiles in each latitude range. These different latitude ranges of tiles have a similar number of inputs. The ranges start at the equator (`v08`) and goes up to the North Pole (`v01`). Equitorial tiles would have the least (0-3) number of input files on average and the polar tiles would have the most (15-20). The other two tile classes chosen, `v05` and `v02`, would see an increase in the number of inputs files as they get away from the equator and closer to the North Pole as indicated by the decrease in `N`.

Figure 6 shows the CAMP task turnaround comparing sinusoidal tile tasks by execution mechanism. We see that each pair of tiles in latitude range has similar box plots. This validates that CAMP task input size governs much of the task turnaround time. As expected the equatorial tiles have the lowest median turnaround time and the polar tiles the greatest. Thread execution has the highest mean time (2 minutes) and largest variance of 3.4 minutes over all latitude ranges. The large variance in task times could be due thread execution being not truly concurrent. Process execution has a faster mean time than distribute until the North Pole tiles `v01`. Distribute has the least variability for all latitude ranges. This is due to the fact the TCP protocol (distribute) has lower overhead than pipes (process). The execution plugin overhead for tasks does not include initialization and task preparation costs as in

template overhead; it only includes the cost of communicating task definitions and results.

The CAMP workflow task turnaround shows that distribute execution is ideal for a large number of tasks that have I/O, compute and memory intensive activities. The median times and variability in task turnaround are acceptable for small and large task inputs.

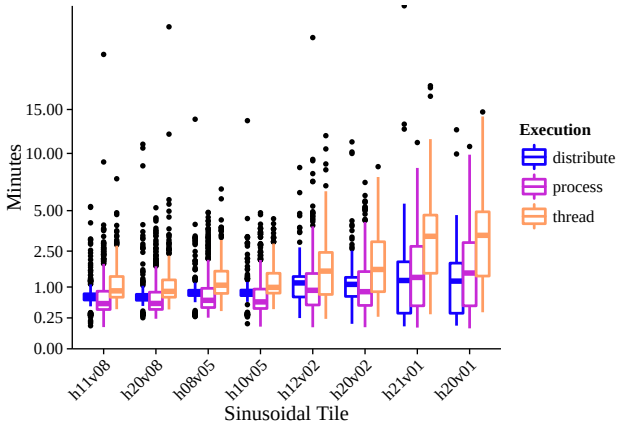


Fig. 6: **CAMP Task Turnaround (Sinusoidal Tile by Execution)** shows CAMP task turnaround on a subset of tasks. Distribute execution is ideal for a large number of tasks that have I/O, compute and memory intensive activities.

Montage. Figure 7 shows the workflow turnaround time for multi-core Montage runs. It shows that as degree increases so does workflow turnaround for all Tigres execution mechanisms. Workflow turnaround for thread and process execution are very similar. Distribute starts out slower but the gap between it and both thread and process workflow turnaround times closes as the degree increases. The initial setup cost is high for distribute (Section III-C) but this cost is a smaller percentage of execution time as the task count increases with the increase in degree. The parallel template in Montage dominates the trend of the workflow.

Figure 8 shows parallel task turnaround time by execution mechanism. As task count increases, so does the reliability of a quick task turnaround (<1 sec). The medians for all execution mechanisms are virtually equivalent. However, the difference in variability in times between execution plugins becomes larger as the task count increases. Distribute task times are less variable as can be see by the closer clustering of slower outlier times (black dots). Process has the most variability in task times which could be due to the communication cost of pipes as compared to TCP. Thread execution task time variability is between process and distribute showing that even though there is no true concurrent execution the communication through a shared address space performs better than pipes (process) on average.

E. Effects of Filesystem Performance

BLAST. Figure 9 shows the results of running three iterations of BLAST weak-scaling experiments using multi-node, distribute execution. Weak-scaling is achieved by appropriately

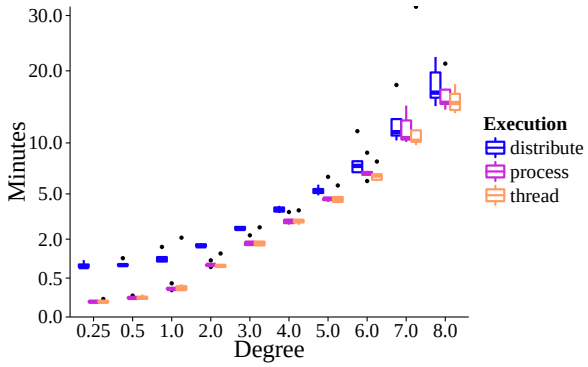


Fig. 7: **Montage Multi-core Workflow Turnaround** plot shows the initial setup cost higher for distribute than process and thread but this cost difference becomes negligible as the task count increases with the increase in degree.

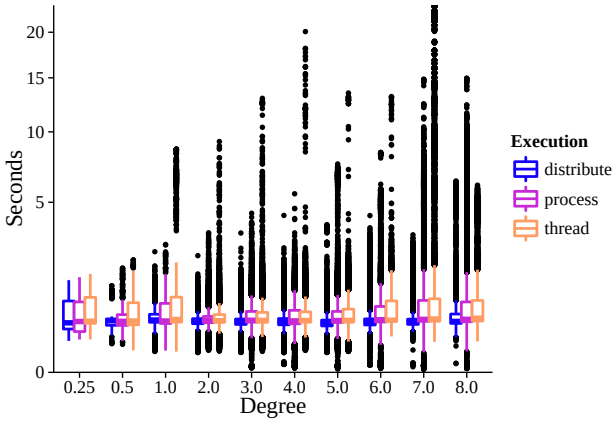


Fig. 8: **Montage Parallel Task Turnaround** shows that variability in task time increases with task count and that process communication is least reliable.

scaling both core count and total query count. The initial sequence template partitions the original queries into smaller input files of 25 queries each, and each partition gets its own core. Input files larger than 7500 query counts are synthesized from recycling queries from the 2500-query input file, implying relatively uniform work load.

The ideal trend for weak-scaling would be a constant slope, as the execution time would stay the same when the available parallelism never outpaces available cores. The experiment seem to indicate this constant trend is realized in BLAST while using Edison's "scratch" file system. BLAST workflows on all tested input sizes finished in a 10-20 minute window. This indicates that that distribute execution for BLAST is quite scalable up to the 1800 core. This demonstrates that Tigres is able to scale with application needs.

In contrast, the turnaround times for *project* file system indicate that the overhead of additional input queries outpaces parallel performance from additional cores. Interestingly, the execution time difference going from 15000 to 22500 queries, and 1200 to 1512 or 1200 to 1800 queries, is even worse than would be expected on single-node execution. Further investigation revealed that IO is a limiting factor.

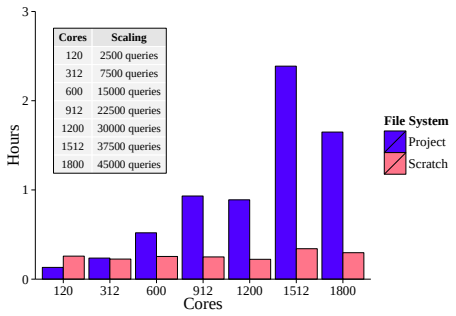


Fig. 9: **BLAST Multi-node** experiments results for distribute-execution weak scaling, by file system. In comparison to figure b), turnaround times for multi-node distribute are a fraction of the the times of their single-node counterparts. However, file system made a big difference, as the scaling is almost constant, using Edison's "*scratch*" file system.

Each BLAST task, although executing independently, reads the same database, and writes an estimated 1.4 MB to disk, prior to a final merge sequence. For reference, running BLAST for 22500 queries produces roughly 1.3 GB of final output. Edison's *scratch* file system is optimized for run-time I/O needs of workflows. *scratch*'s Lustre file system is designed for running I/O intensive jobs, with 4x the peak throughput of *project*'s GPFS, which is designed for permanent, shared storage. For instance, 37500 queries correspond to 1512 tasks running on 1512 cores, and potentially reading from the same database simultaneously. On *project* the IO-contention may be manageable for lower degrees of parallelism, but will become increasingly less so past a certain point, which would explain the trends observed. Another factor that affects the performance was number of jobs currently running in queue. To minimize interference, during our experiments with *scratch* we made sure only one job was active at a time. During the experiments with *project*, multiple jobs might have been active at a time causing further slowdown.

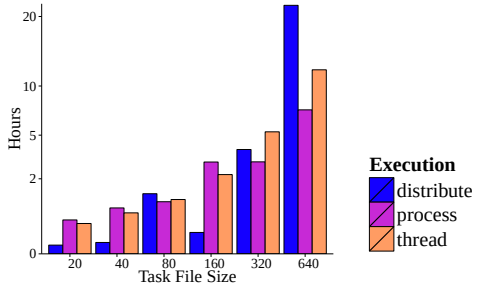


Fig. 10: **Synthetic I/O Multi-core Workflow Turnaround** for parallel templates was evaluated by template, task file size and Tigres Execution. A trend of increase in turnaround time as file size per task increases. There are two exceptions to this for Tigres distribute execution. Writing 160MB task file sizes was much faster than 80MB. Which leads us to question whether there is a variability in file system performance that is affecting these file write dominated workflows.

Synthetic I/O workflows in Figure 10 show the turnaround by Tigres execution and task file size. The parallel tasks writes 24 files concurrently at varying sizes. Figures 10 shows a trend of increase in turnaround time as file size per task increases. There are two exceptions to this for Tigres distribute execution. In these plots, writing 160MB task file sizes was much faster than 80MB. This is likely due to *scratch* filesystem variability in performance and that variability affects the Synthetic I/O workflow turnaround times.

Figure 11 has two plots that relate the Synthetic I/O task turnaround time with the variability in the *scratch* filesystem write performance. Figure 11a shows the task turnaround times for all Synthetic I/O parallel tasks by Tigres execution and task file size. Figure 11b uses the NERSC health data for file creation on Edison's *scratch* filesystem. This plot ties the mean Edison *scratch* create file time to each task execution in the left plot; it plots the NERSC create file time by Tigres execution and task file size.

Figure 11a shows that for 20, 40 and 160 MB distribute experiments there was a very low mean task turnaround with much variability in the 3rd quartile as evidenced by the large upper boxes. Figure 11b shows that the mean create file times at NERSC were comparatively low (~2.5 secs) when compared to 80MB distribute (~4 secs). While comparing these two plots will not exactly predict workflow performance it demonstrates that there is much variability in the filesystem which makes it difficult to predict workflow and task performance.

F. Comparison of Task Implementations

Synthetic Compute. Figure 12 shows the multi-core results for all four template versions. These results demonstrate the effectiveness of three compute bound task implementations: C executables, Python executable and Python function.

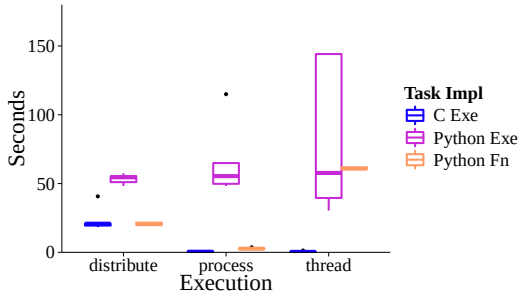
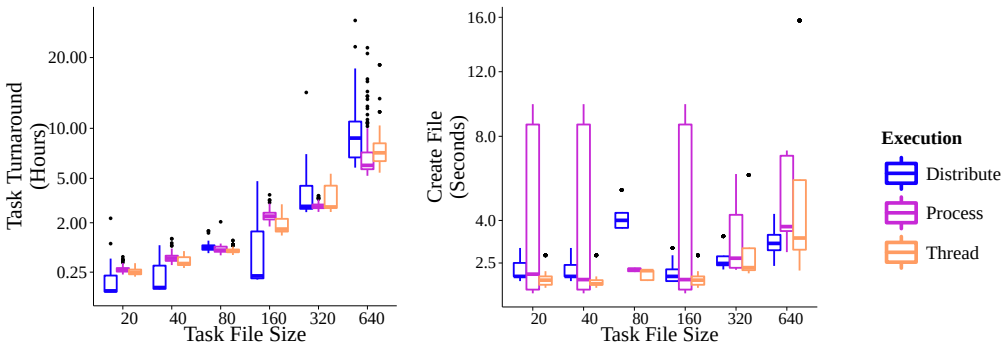


Fig. 12: **Synthetic Compute Multi-core Workflow Turnaround** for parallel templates demonstrates that compute bound python executable task implementations perform poorly for all templates. C programs and inline python functions perform similarly in plot a except for thread execution which performs poorly in comparison. This is most likely due to the fact that python thread execution does not actually run concurrently.

Figure 12 omits a single outlier for readability. The relationship between workflow turnaround times for the three task implementations is the same for all templates (merge and split plots omitted due to space constraint). The synthetic compute tasks are CPU bound with low use of memory or



(a) Task Turnaround Time (b) NERSC Create File Time

Fig. 11: **Synthetic I/O Tasks.** These plots demonstrated Edison's *scratch* filesystem variability for the Synthetic I/O experiments. The left plot shows the Task Turnaround times for all Synthetic I/O parallel tasks by Tigres execution and task file size. The right plot uses the NERSC health data for file creation on Edison's *scratch* filesystem. It plots create file time in relation to each Synthetic I/O experiment.

communication. In the case of process and distribute, both *C Exe* and *Python Fn* perform the same with distribute execution being slower over all.

Thread execution gives us different behaviors for *Python Exe* and *Python Fn* execution. While the workflow turnaround for *Python Exe* has the same median there is more variability in times as seen by the large size of the box in Figure 12. *Python Fn* turnaround times have little variability for thread execution as evidenced by the small boxes in Figures 12. The reason *Python Fn* performs worse in thread execution is that tasks are not truly concurrent because they are hindered by the GIL. However, *Python Exe* and *C Exe* tasks are concurrent for the duration of their execution because they are executed outside of the python interpreter. There is more variability in *Python Exe* thread execution for all Tigres templates which could be due to the bottleneck of launching from worker threads slowed down by Python's GIL. We saw a similar result for CAMP tasks (Figure 6). This demonstrates initialization of the python interpreter is highly variable. Thread execution can achieve almost true parallelism when invoking executables and is most efficacious for binary executables.

Figure 13 demonstrates the effect of scaling up synthetic compute workflows from 24 to 1536 cores for Synthetic Compute parallel templates. Merge and split templates had similar results but were omitted to to space constraint. The core count is equivalent to the task count and the distribute execution mechanism is used because it can distribute tasks across multiple nodes. As expected, *Python Exe* performs poorly becoming extremely slow at the higher core counts. Since we are comparing distribute executions we know that the communication cost is similar. Neither *C Exe* nor *Python Fn* initializes a new python interpreter for each task execution as *Python Exe* does. The cost of Python interpreter initialization causes *Python Exe* to perform poorly as the core count rises. *Python Fn* and *C Exe* execution start out even as we see in Figure 12. At 1536 cores, *Python Fn* performance degrades which is understandable as compiled languages perform better than

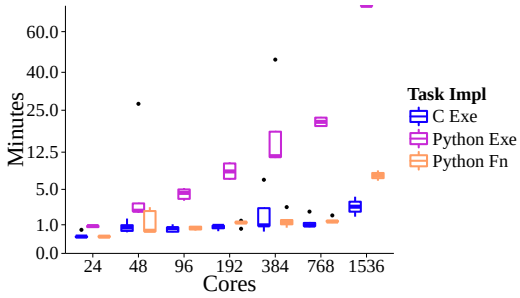


Fig. 13: **Synthetic Compute Multi-node Workflow Turnaround** for parallel templates shows that workflow turnaround for Python Exe task implementation increases dramatically as the core count increases. Turnaround times for C Exe and Python Fn have a relatively flat increase in time until 768 cores. This is due to the increase overhead in communication for the internal queues of the execution mechanisms.

interpreted ones and there is more data being communicated than *C Exe* because the python function is serialized and sent to the Tigres clients in addition to the task parameters.

G. Summary

Our findings from the synthetic and science applications demonstrate several factors that affect the performance of scientific workflows. In this section, we summarize our evaluation results.

Template overheads and task count. Tigres has minimal template overhead (mean of 13 seconds over all experiments). Tigres template overhead increases with task count, is unaffected by I/O, and is negligible for sequence templates. Distribute execution communication and setup costs for parallel execution is more costly than process and thread.

Task turnaround and task count. As parallel task count increases, the initial overhead cost of distribute is gained back by better communication performance during task execution than process and thread execution as seen in the Montage

multi-core experiments. The multi-core CAMP application, where task count was ~5800, saw the distribute execution plugin out perform in all classes of task complexity.

Task Implementation and execution plugin. Task implementation and execution plugin are important considerations when designing a Tigres workflow. Python executable tasks are the worst performing especially when used in conjunction with thread execution. Binary executables tasks have good workflow performance for all execution plugins.

Filesystem and node count. It is critical to choose the right filesystem when scaling up application task and node count for applications with significant I/O, such as BLAST. The BLAST experiment showed that choosing Edison *scratch* which is designed for job's I/O saw relatively no increase in time.

During our experimentation process, we saw a number of failures. For example, out of 1730 workflows (Montage, CAMP and Synthetic) that ran between May and July 2015, 8% failed due to wall clock exceeding from filesystem variability, communication errors, etc.

V. Conclusions

In this paper, we presents the design and implementation of Tigres, a library that supports scientific workflow pipelines on HPC systems. We evaluated three scientific and several synthetic applications and show that Tigres has minimal overheads. Additionally, we evaluated various factors that might affect workflow performance including execution mechanisms, task implementations and file system performance. Our results show that scientific applications must carefully select various application and resource mechanisms for optimal performance. Tigres enables scientists to compose their workflows and port them to various environments seamlessly.

Acknowledgements

This work and the resources at NERSC are supported by the Director Office of Science, Office of Advanced Scientific Computing; of the U.S. Department of Energy under Contract No. DEAC0205CH11231.

References

[1] B. Ludscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. Lee, J. Tao, and Y. Zhao, "Scientific Workflow Management and the Kepler System," 2005.

[2] T. Oinn *et al.*, "Taverna: lessons in creating a workflow environment for the life sciences," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 10, pp. 1067–1100, 2006.

[3] E. Deelman, G. Singh, M. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. Berriman, J. Good *et al.*, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming*, vol. 13, no. 3, pp. 219–237, 2005.

[4] J. Goecks, A. Nekrutenko, J. Taylor, and The Galaxy Team, "Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences," *Genome Biology*, vol. 11, no. 8, p. R86, 2010.

[5] R. Barga, J. Jackson, N. Araujo, D. Guo, N. Gautam, and Y. Simmhan, "The trident scientific workflow workbench," in *eScience, 2008. eScience '08. IEEE Fourth International Conference on*, dec. 2008, pp. 317 –318.

[6] P. Bui, L. Yu, and D. Thain, "Weaver: integrating distributed computing abstractions into scientific workflows using python," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. ACM, 2010, pp. 636–643.

[7] D. Churches, G. Gombas, A. Harrison, J. Maassen, C. Robinson, M. Shields, I. Taylor, and I. Wang, "Programming Scientific and Distributed Workflow with Triana Services," *Concurrency and Computation: Practice and Experience (Special Issue: Workflow in Grid Systems)*, vol. 18, no. 10, pp. 1021–1037, 2006.

[8] A. Jain and et. al., "FireWorks: a dynamic workflow system designed for high-throughput applications," *Concurrency and Computation: Practice and Experience*, p. n/a, May 2015.

[9] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster, "Swift: A language for distributed parallel scripting," *Parallel Computing*, vol. 37, no. 9, pp. 633–652, 2011.

[10] "QDO." [Online]. Available: {https://bitbucket.org/berkeleylab/qdo}

[11] M. Wei, C. Jiang, and M. Snir, "Programming patterns for architecture-level software optimizations on frequent pattern mining," in *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, april 2007, pp. 336 –345.

[12] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from berkeley," EECS Department, University of California, Berkeley, Tech. Rep., Dec 2006, http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html.

[13] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, Aug. 1978.

[14] L. Dagum and R. Menon, "OpenMP: an industry standard api for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[15] K. Yelick and et. al., "Productivity and performance using partitioned global address space languages," in *Proceedings of the International workshop on Parallel symbolic computation*. ACM, 2007, pp. 24–32.

[16] M. Snir, S. Otto, D. Walker, J. Dongarra, , and S. Huss-Lederman, *MPI: The complete reference*. MIT press, 1995.

[17] S. Ghemawat and J. Dean, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI04)*, 2004.

[18] L. Ramakrishnan, P. T. Zbiegel, S. Campbell, R. Bradshaw, R. S. Canon, S. Coghlan, I. Sakrejda, N. Dešai, T. Declerck, and A. Liu, "Magellan: Experiences from a science cloud," in *Proceedings of the 2nd international workshop on Scientific cloud computing*, ser. ScienceCloud '11. ACM, 2011, pp. 49–58.

[19] E. Dede, M. Govindaraju, D. Gunter, and L. Ramakrishnan, "Riding the elephant: managing ensembles with hadoop," in *Proceedings of the 2011 ACM international workshop on Many task computing on grids and supercomputers*, 2011, pp. 49–58.

[20] C. Zhang, H. De Sterck, M. Jaatun, G. Zhao, and C. Rong, "CloudWF: A Computational Workflow System for Clouds Based on Hadoop," in *Cloud Computing*, ser. Lecture Notes in Computer Science, M. G. Jaatun, G. Zhao, and C. Rong, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, vol. 5931, pp. 393–404.

[21] "Oozie: Workflow engine for Hadoop," http://yahoo.github.com/oozie/.

[22] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10.

[23] Lawrence Berkeley National Lab , "Logging Best Practices." [Online]. Available: {https://docs.google.com/a/lbl.gov/document/d/1oeW_l_YgQbR-C_7R2cKl6eYBT5N4WSMbvz0AT6hYDvA/edit}

[24] "Netlogger." [Online]. Available: {http://netlogger.lbl.gov/}

[25] S. Canon, S. Cholia, J. Shalf, K. Jackson, L. Ramakrishnan, and V. Markowitz, "A performance comparison of massively parallel sequence matching computations on cloud computing platforms and hpc clusters using hadoop," *Using Clouds for Parallel Computations in Systems Biology Workshop, Held at SC09*, 2009.

[26] Pegasus, University of Southern California Information Sciences Institute , "Pegasus Montage Tutorial." [Online]. Available: {https://confluence.pegasus.isi.edu/display/pegasus/Montage}

[27] J. Balderrama, M. Simonin, L. Ramakrishnan, V. Hendrix, C. Morin, D. Agarwal, and C. Tedeschi, "Combining workflow templates with a shared space-based execution model," in *Workflows in Support of Large-Scale Science (WORKS), 2014 9th Workshop on*, Nov 2014, pp. 50–58.

[28] V. Hendrix, L. Ramakrishnan, Y. Ryu, C. van Ingen, K. R. Jackson, and D. Agarwal, "Camp: Community access MODIS pipeline," *Future Generation Computer Systems*, vol. 36, pp. 418 – 429, 2014.