Parallel High-resolution Climate Data Analysis using Swift

Matthew Woitaszek
National Center for
Atmospheric Research
1850 Table Mesa Drive
Boulder, CO 80305
matthew@ucar.edu

John M. Dennis
National Center for
Atmospheric Research
1850 Table Mesa Drive
Boulder, CO 80305
dennis@ucar.edu

Taleena R. Sines
Frostburg State University
Frostburg, MD
trsines0@frostburg.edu

ABSTRACT

Advances in software parallelism and high-performance systems have resulted in an order of magnitude increase in the volume of output data produced by the Community Earth System Model (CESM). As the volume of data produced by CESM increases, the single-threaded script-based software packages traditionally used to post-process model output data have become a bottleneck in the analysis process. This paper presents a parallel version of the CESM atmosphere model data analysis workflow implemented using the Swift scripting language.

Using the Swift implementation of the workflow, the time to analyze a 10-year atmosphere simulation on a typical cluster is reduced from 95 to 32 minutes on a single 8-core node and to 20 minutes on two nodes. The parallelized workflow is then used to evaluate several new data-intensive computational systems that feature RAM-based and flash-based storage. Even when constraining parallelism to limit the amount of file system space used by intermediate temporary data, our results show that the Swift-based implementation significantly reduces data analysis time.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; J.2 [**Physical Sciences and Engineering**]: Computer Applications—Earth and atmospheric sciences

General Terms

Performance

Keywords

data-intensive computing, workflow orchestration, many-task computing, climate modeling

1. INTRODUCTION

Earth System modeling involves the study of the processes and interactions of earth system components such as the ocean, atmosphere, land, sea ice, biogeochemistry, and other terrestrial systems. These models are typically run to simulate long periods of time ranging from 10 to 10,000 years. The Community Earth System Model (CESM) [3] can now utilize over 115,000 cores and generate in excess of 1 TB of output data per simulation per wall-clock day of execution on current supercomputers. While modest diagnostic functions are performed when the models are running, more comprehensive comparisons with existing control simulations are only possible as a post-processing step. However, the post-processing data analysis scripts have typically remained single-threaded and have become a bottleneck to scientific investigations. The ability to effectively parallelize data analysis will allow for faster comparisons and re-analyses, resulting in faster evaluation of model operations and subsequent improvements.

In this paper, we present a parallel implementation of the computational portion of the CESM Atmospheric Model Working Group (AMWG) diagnostic package [1]. This diagnostic package is typically run every few days to perform analysis on a time segment of atmosphere model output data that was produced on a separate supercomputer. Historically, with lower-resolution data sets that have a separation of 2.0° (∼200 km) between neighboring grid-points, the analysis scripts may have taken 10 to 15 minutes to execute. However, with the higher resolutions (e.g., 0.5°) now in use, the serial analysis scripts can require overnight or longer to process a recently completed segment. Our goal for this work is not to achieve the same level of parallelism as the climate applications that generate the data or to fundamentally rearchitect the manner in which we perform analysis, but to reduce the time to perform analysis to historical standards and to understand how architectural features of *data-intensive* systems being deployed on the TeraGrid can be leveraged by our workflow.

We parallelized our data analysis software using the Swift scripting language [15]. Swift's data-centric approach that treats data in files as variables and automatically derives workflow dependencies greatly simplified the creation of the parallel analysis script. Moreover, Swift introduced a useful layer of abstraction between the workflow, the underlying system, and the executables that perform tasks. The analysis script is now platform-agnostic, so the Swift environment can be configured to use the execution methods, storage technologies, and processing components that are most appropriate for each computational platform without modifying the workflow definition.

We also investigate the ability of three different data-intensive compute platforms to execute the analysis script. The test platforms include the shared-memory SGI system “Nautilus” at the National Institute for Computational Science (NICS), the “Dash” cluster with a flash-based file system at the San Diego Supercomputing Center (SDSC), and the “Polynya” large-memory analysis server prototype at the National Center for Atmospheric Research (NCAR). We demonstrate that it is possible to achieve 7× speedup by utilizing 4 nodes of Dash, making it again possible to perform analysis jobs during a coffee break or lunch using only a very modest increase to the computational resources required. Further reductions in execution time are observed when using flash or RAM-based file systems as temporary storage for the workflow. However, as flash and RAM-based storage is often limited in capacity more so than high-performance spinning disk, we examine mechanisms to constrain the parallelism and thus the temporary storage requirements of the workflow in order to effectively utilize high-performance storage resources.

The remainder of this paper is organized as follows. Section 2 introduces Swift and related work. Section 3 describes the climate analysis workflow, the design of our parallel implementation, and Swift features relevant to the workflow. Section 4 presents results, first describing scalability on clusters and then highlighting performance using new storage technologies. The paper concludes with a discussion of Swift, the data-intensive platforms, and future work.

2. BACKGROUND AND RELATED WORK

A variety of tools and technologies exist to define, parallelize, and automate the computational, data transfer, and analysis and visualization components of scientific workflows. We considered two – Kepler [9, 4] and Swift [15, 12] – as possible frameworks to implement a parallel version of the AMWG diagnostic workflow. These tools take dramatically different approaches to defining and structuring execution. Kepler uses a workflow-centric approach where tasks are explicitly ordered as the workflow is constructed. Swift uses a data-centric approach where tasks are executed based on detected dependencies in declared data structures and procedures. For our workfow, which runs tasks based on the cascade of data files in multi-step processing sequences, we found Swift’s data-centric paradigm to be a better match.

Swift consists of a scripting language and execution system that provides a simple mechanism to define data objects and rules (“transformations”) that consume or produce data objects. Swift provides basic variables, such as integers, floats, and strings, as well as opaque objects that map to external data sets such as files on a disk device. File mappers can point to explicitly named files, such as input or output data, or to anonymous files used to link processing stages with temporary files. Transformation procedures, also referred to as atomic procedures, define how external applications transform opaque objects. Based on the input and output objects of transformation procedures, Swift derives dependencies, and procedures are executed (“triggered”) when all of their input objects are available.

Swift has been previously used to automate workflows in a variety of disciplines including molecular dynamics, economics, and computational neuroscience [12]. A precursor to Swift, GriPhyN, was used to automate ensembles of climate model runs [10], allowing the investigators to perform experiments that were impractical without automated task management. Most of the prior work in the literature involves workflows that are characteristic of “many-task computing” [11], requiring the efficient execution of large numbers – from thousands to millions – of short-running tasks. As used for the iterative model execution/validation cycles motivating this project, AMWG diagnostic runs execute substantially fewer tasks (204-511) that manipulate large files (820 MB each), making it a data-intensive task-oriented workflow. However, one benefit of the parallelization is that it can easily be used to analyze multiple decades or even centuries, simply generating more tasks as required by the data dependency flow.

3. DESIGN AND IMPLEMENTATION

The AMWG diagnostic package has been used to analyze the output of the Community Atmosphere Model (CAM), the atmospheric model component of CESM and its predecessors including Community Climate System Model (CCSM), since 1999 [1]. Written as a C-shell script, the AMWG diagnostic package performs two phases of processing. First, the statistics generation phase executes a series of netCDF Operator (NCO) [5] commands that read the monthly CAM output files and perform weightings and summations to produce statistics files. The second phase of processing executes NCAR Commmand Language (NCL) [2] scripts that read the intermediate statistics files and generate a variety of graphical plots arranged in an HTML tree ready for publication on a webserver.

Preliminary timing of the serial AMWG script indicated that the cost of the statistics generation phase relative to the plot generation phase increases with resolution. For example, in the case of the high-resolution 0.5° data motivating this project, statistics generation consumed 70% of the execution time. It is for this reason that we focused on the statistics generation portion of the workflow. We analyzed the C-shell-based AMWG script to extract the workflow, implemented the workflow using Swift, and then produced a modified version of the C-shell script that can use the Swift implementation instead of the built-in serial implementation if selected by the user.

The AMWG diagnostics statistics generation workflow consists of four independent analysis chains for summer (June, July, August; JJA), winter (December, January, February; DJF), annual (ANN), and monthly (MON) statistics (see Figure 1). The figure shows the number of task executions, as well as the volume of data produced at each step, where N is the number of years in the analysis and each file is equal in size to a monthly output file from CAM. For our benchmark 0.5° 10-year analysis, each input file is 820 MB (0.8 GB). The size of input files as well as total

Table 1: Model output (analysis workflow input) data size for AMWG diagnostic benchmarking

Resolution	Monthly File Size	10-Year Analysis Size
0.50°	820.1 MB	96.11 GB
1.00°	232.6 MB	27.26 GB
2.00°	51.3 MB	6.01 GB

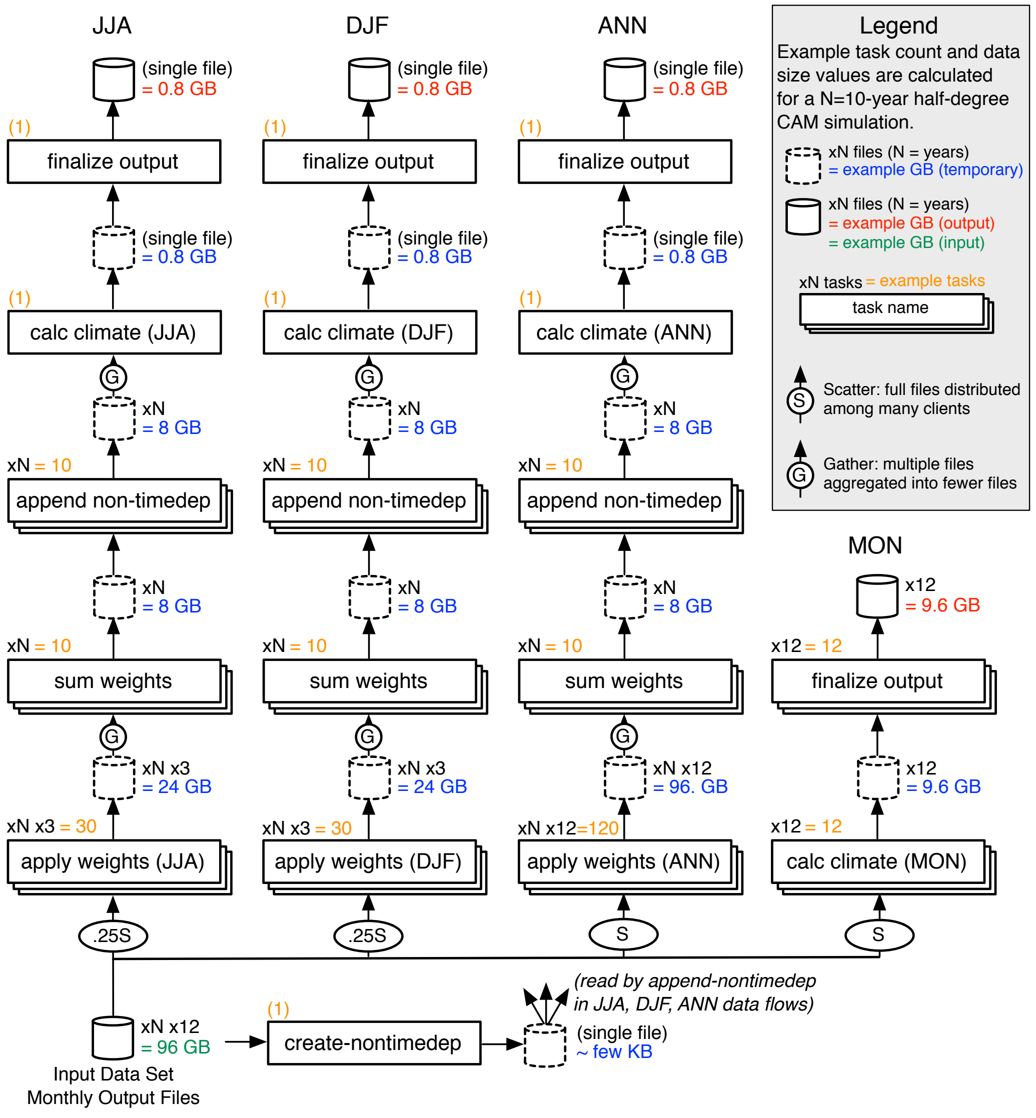


Figure 1: AMWG diagnostics statistics generation workflow (presented using representation based on Wozniak and Wilde [14]). Example task count and data size values are for a 0.5° 10-year atmosphere (CAM) analysis.

input data size for other resolutions is provided in Table 1. (Note that the size of 0.5° data is not quite four times larger than the 1.0° because several unused variables were identified and eliminated from the 0.5° output files.) In the 0.5° example, the workflow starts with 96 GB of input data, requires 271 task executions, and produces 204 GB of intermediate data and 12 GB of output data. In practice, a real analysis may consist of many invocations (or a single invocation) of this workflow on 30-100 TB of data, such as running a decadal analysis for every decade in multiple hundred-year simulations for purposes of comparison.

To illustrate how the workflow is implemented in Swift, we examine the *apply weights (JJA)* component of the workflow from Figure 1. This component applies weighting factors to input files from the summer months using the NCO operator *ncflint*. Figure 2 is the simplified Swift code that implements this component of the workflow and consists of three code blocks. The first code block defines the transformation procedure *ApplyWeights* that calls the NCO operator *ncflint* to transform the input file *f* to the output file *fout* by applying weights defined by *weights*. The second code block constructs the list of names of the input files on which to perform the *ApplyWeights* transformation. The third code block calls the transformation procedure for each month. Note that the third code block is contained in another foreach loop (not illustrated in Figure 2) that iterates over each year in the analysis. Because there are no dependencies between each call to the *ApplyWeights* procedure, all 30 calls can be executed in parallel as independent tasks.

```
#----------------------------------------------
# Transformation procedure ApplyWeights
#----------------------------------------------
(file fout)ApplyWeights(string n_t_var, string weights, string f)
{
  app {ncflint "-O" "-C" "-x" "-v" n_t_var "-w" @strcat(weights,",0.0") f f @filename(fout);}
}

#----------------------------------------------
#  Construct the list files for transformation
#   tp:  absolute pathname
#   yr:  year string
#----------------------------------------------
string names[]= [@strcat(tp,yr,"-06.nc"),@strcat(tp,yr,"-07.nc"),@strcat(tp,yr,"-08.nc") ];
file files[]<array_mapper;files=names>;

#------------------------------------------------------------------
# Loop over the months June, July, and August to apply weights.
#  (for input, force Swift to read the real file, not a staged copy)
#------------------------------------------------------------------
string jja_weights[] = [".3260869681835", ".3369565308094",".3369565308094"];
int m[] = [1, 2, 3];
foreach mth in m {
   # Apply transformation procedure to each month's input
   wgtf[mth-1]=ApplyWeights(ntVars, jja_weights[mth-1], @strcat("/",@files[mth-1]));
}
```

Figure 2: Example Swift code that implements the *apply weights (JJA)* component of the workflow.

For this workflow, it is important to highlight the expansion factor of intermediate data from the input (model output) data. The first step in each analysis chain is the application of weights based on the average number of months in that analysis, so each of the four chains must apply different weights to every file. This results in 2.5× the input data volume being read and stored at the very first step. Next, the weighted month files are combined to produce aggregate statistics for the particular data product (3 months for JJA or DJF, or 12 months for annual). The remaining processing executes on substantially fewer files. It is this expansion factor that stresses the storage and I/O throughput components of most underlying computational systems, as the workflow's tasks read ∼444 GB of data and write ∼194 GB of data to turn ∼100 GB of input into ∼10 GB of output.

3.1 Relevant Swift Features

Several of Swift's features, including support for multi-site execution through multiple submission providers and automatic data staging, is particularly relevant to the implementation and performance of the workflow. It is necessary to choose an execution provider appropriate for the target environment (e.g., one node or a cluster) and configure the file access methodology to match the file systems available for input, temporary, and final output data.

Execution providers. Swift can execute tasks on multiple resources, referred to as "sites", simultaneously using a variety of execution providers. Common providers include just running tasks on the local node (localhost) or submitting tasks to a cluster's batch scheduler either locally (qsub) or via Globus GRAM. Instead of submitting every task to a queue individually, Swift can also execute tasks using the Java Commodity Grid (CoG) Kit's coasters lightweight task dispatching framework. Coasters can in turn be run on a single node locally or on multiple nodes through a queue. The selection of the dispatch method is independent from the script itself. We generally chose to use coasters because it reliably limits the number of executing tasks, allowing data describing workflow execution time vs. number of cores to be easily collected. In the typical use case on a cluster, the user starts the analysis in an interactive shell on a login node, and the Swift script takes care of submitting jobs to the back-end using the system's queue. On other systems, such as large shared-memory systems and clusters with a few large nodes, the user submits the Swift script to the queue and coasters runs a limited number of tasks on the allocated resource directly.

Data staging. To make multi-site runs possible, along with restart and recovery capabilities in the case of site execution or file transfer faults, Swift uses a hierarchy of storage locations. First, the directory on the computer from which the user is running Swift serves as a home base for files, and all data is staged from and to this run directory. Second, every execution site has a directory used by all tasks run by Swift on that particular site; this directory is thus intended to be a high-performance file system shared by all nodes at that site. Finally, in the case where individual execution nodes at a site have independent storage (rather than site-shared storage), Swift also supports staging to the individual nodes themselves. This staging hierarchy is combined with file and directory name mangling to ensure that independent tasks, when run in parallel, execute in separate directories and do not produce conflicts.

In the case of the AMWG workflow, the use of independent staging directories (one for the Swift run, and one for the target site) is problematic because in practice the workflow is run on only one system – the analysis system attached to the file system with the data. In that case, the directory from which Swift is run and the Swift site directory used for staging are on the same parallel file system. With independent staging directories, the volume of data that

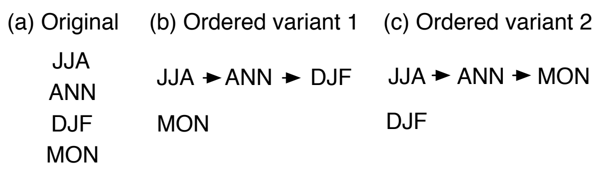


Figure 3: Parallelism constraint variants for the Swift-based AMWG diagnostic statistics workflow. Version (a) removes no files and enforces no processing order; versions (b) and (c) remove intermediate files and order processing phases.

must be stored on the same file system doubles from 204 GB to 408 GB for our workflow, and extensive amounts of time are spent copying data between two directories on the same file system. To avoid this unnecessary duplication, we used Swift's support for Collective Data Management (CDM) [13, 14] to configure Swift to access intermediate files directly with no staging. Use of Swift's CDM capability had a profound positive impact on both disk space usage and execution time for our workflow.

3.2 Workflow Temporary Data Management

As an execution framework designed to coordinate the execution of file-processing tasks, Swift takes the safe approach of not deleting any files, including anonymous intermediate files. Swift also executes procedures in a trigger-like fashion, where any elegible task can be run, rather than scheduling tasks on subsequent dependencies. Because the AMWG workflow can require hundreds of gigabytes of temporary storage space if all temporary files are retained through the duration of the analysis, the volume of temporary data can become a problem on systems with limited or quota-controlled temporary storage. Moreover, it makes it very difficult to take advantage of systems with special high-performance intermediate storage in small capacities.

With the goal of constraining intermediate data storage, reducing file system contention, and cleaning up unneeded temporary space while a run is in progress, we designed two alternate versions of the analysis script. These alternate versions introduce extra dependencies that artificially order some of the task chains and remove all of the intermediate files after each task chain is complete (see Figure 3). The original version (a) has no forced ordering and does not remove any files. In the alternate versions, the analysis steps are followed by dependencies that remove intermediate output files that are no longer needed. To further constrain the generation of intermediate files, the processing of each year in each chains is ordered. The first alternate version (b) forces each year in the JJA, ANN, and DJF analysis chains to run in sequence so that an entire year must be completed in JJA before that year begins processing in ANN. This effectively changes the task elegibility search from breadth-first to depth-first for a portion of the workflow, constraining the parallelism and reducing the use of storage space. Functional testing shows that the alternate versions do in fact remove temporary files and constrain intermediate storage space from 204.2 GB (version a) to 123.3 GB and 130.5 GB for versions (b) and (c). However, as will be described in the Results section, these constraints do not reduce the parallel performance with statistical significance for up to 64 processors.

3.3 Workflow Parallelism and Efficiency

To illustrate the maximum parallelism characteristics of the workflow, we use Gantt charts combined with disk utilization plots that show the execution of tasks and the use of storage as the workflow executes. The Gantt charts describe how the workflow's data dependencies constrain the available parallelism and in turn limit the efficiency achievable with an increasing number of processors. For example, when Swift is configured to queue all ready tasks as soon as they are triggered, and 64 processors are available, large gaps of unused processors emerge (see Figure 4(a)). However, when Swift is configured to submit no more than 8 tasks for execution to 8 available cores, all processors remain busy (see Figure 4(b)). Thus, processors sit idle if the number of available processors exceeds the least upper bound of parallelism among all of the workflow's stages, which occurs between 8 and 64 cores for this workflow.

4. RESULTS

We evaluate the performance of the Swift-based parallel analysis workflow on ten years of output data from a 0.5° run of CAM on a variety of computer platforms. All timing exeriments were performed at least three times, with standard deviations plotted. The workflow is executed on the following platforms:

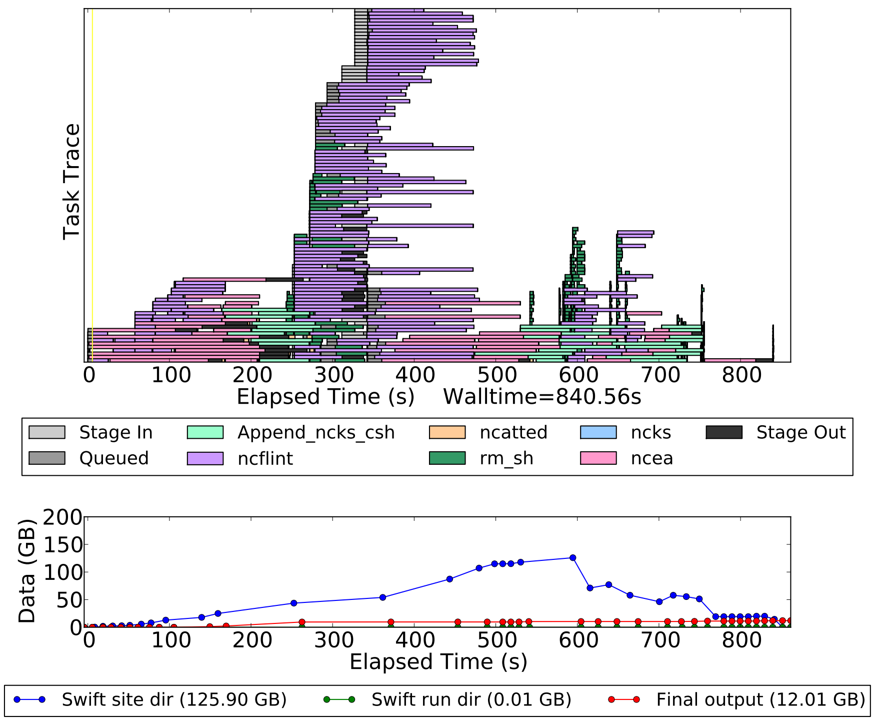
- **Dash** [7], a cluster targeted for data-intensive computing at SDSC, consisting of 32 nodes each with dual quad-core 2.4 GHz Intel Nehalem processors and 48 GB of RAM per node. Half of Dash is configured as a traditional 16-node cluster but with a single 64 GB flash drive in each node. The other half of the system is aggregated into a single system image using ScaleMP's virtual shared memory (vSMP) product, providing 960 GB of flash storage and a RAM-based file system as well. Persistent storage on Dash is provided by TeraGrid's GPFS-WAN, hosted at SDSC.

- **Nautilus**, an SGI Altix UV 1000 at NICS, containing 1024 Nehalem EX cores at 2.0 GHz with 4 GB RAM/core. Persistent storage on Nautilus is provided by a local GPFS file system.

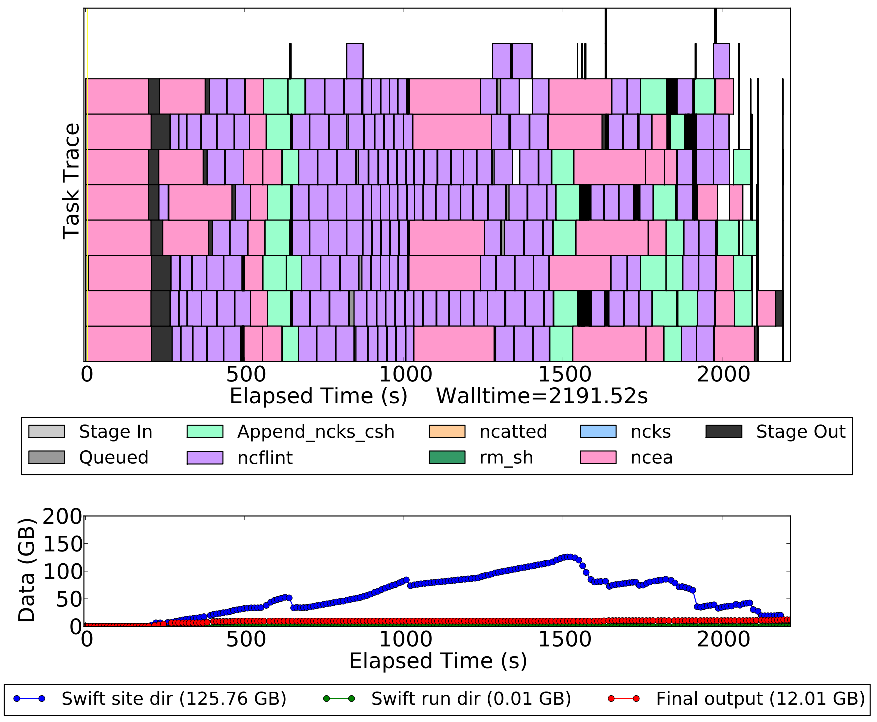
- **Polynya**, a single-node prototype for a large-memory data analysis and visualization cluster at NCAR. The server used for this experiment contains 32 Intel Nehalem cores and 1 TB of RAM. Half of the memory has been used to create a 512 GB RAM disk using `tmpfs`. Persistent storage is provided by a GPFS file system shared with NCAR's Blue Gene/L supercomputer.

4.1 Serial and Variant Performance

We first established the baseline performance of the Swift-based workflow, as well as the performance of the variants that constrain workflow parallelism and remove intermediate files, using the portion of Dash that is configured as a typical Linux cluster. In this hardware configuration, Dash uses the GPFS disk-based file system to store input, intermediate,



(a) 64 cores available, no throttle



(b) 8 cores available, 8-task throttle

Figure 4: Task execution Gantt charts and disk usage plots for a 64-core system with no Swift throttling (4(a)) and an 8-core system throttled to 8 tasks (4(b)). The 64-core plot shows the achievable parallelism limited only by dependencies; the 8-core plot shows core allocation efficiency by limiting the resource request.

and output data, and does not make use of flash memory or the vSMP single-system-image technologies. This provides a useful characterization of the parallel performance achievable on typical commodity clusters absent specialized data-intensive hardware components.

The use of Swift introduces a small amount of task management overhead that is visible in runs that have been constrained to only one processor. The original serial C-shell version requires 96 minutes to run on Dash. The version written in Swift, constrained to one processor, requires an average of 102 minutes, a slowdown of 6%. This result demonstrates that Swift overhead is modest for our workflow.

Constraining workflow parallelism and removing intermediate temporary files using the variants described in Section 3.2 do not reduce the performance with statistical significance at the 95% confidence level. The mean execution time of the version without the file removals is always faster than the versions that remove the files – it is, after all, extra work – but this difference is not always statistically significant. The extra time required to remove the files within the workflow is thus "in the noise", and removing them within the workflow avoids the need to clean-up the temporary files manually after the workflow completes. We therefore chose to use version (b) for the remainder of our timing experiments.

4.2 Parallel Performance

The primary result of our work is an examination of the parallel performance of the Swift-based workflow on three current platforms–SDSC Dash (without vSMP), NICS Nautilus, and NCAR Polynya–all configured to use their high-capacity disk-based storage systems for input, output, and intermediate data (see Figure 5(a)). The results show that Dash and Nautilus generally have similar performance at 1, 8, and 16 tasks, reducing execution time from ∼102 minutes to ∼30 minutes, with Nautilus having slightly better performance at 8 tasks and Dash for higher processor counts.

The single-core results suggest that single processes on Dash and Nautilus have approximately the same throughput to disk, which is also significantly better than can be achieved on Polynya. This is consistent with the fact that both Dash and Nautilus are both newly deployed systems and that Polynya uses disk subsystem components that are approximately four years old. The parallel performance shows that while Nautilus has slightly better execution time than Dash on 8 cores (one Dash node), the trend reverses at greater than 16 cores. We suspect that at small core counts, Nautilus benefits from access to a larger number of network connections to disk than Dash due to its single-system-image and high-performance shared-memory interconnect. However, once Dash begins to utilize multiple nodes and thus more connections to the disk system, its execution time is lower.

To better understand the interaction of the workflow, its parallelism, and the storage technology on performance, we ran additional tests on the Polynya server using a RAM-based file system in three configurations that vary the usage of disk and RAM for storage (see Figure 5(b)). The first configuration (*Spinning Disk*), described previously, uses GPFS for all storage. The second configuration (*Hybrid*) reads input from GPFS, but stores all temporary and intermediate data as well as final output on a RAM disk.

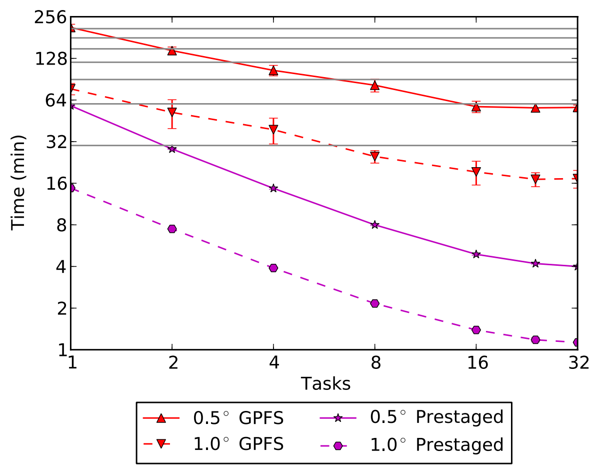


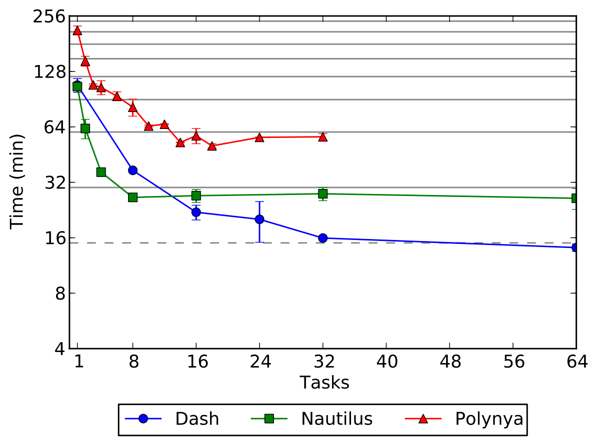
Figure 6: Execution time of the parallel workflow for 0.5° and 1.0° data sets on the NCAR Polynya server using GPFS and prestaged RAM storage.

This represents the expected use case for the Swift-based implementation in production. For the third configuration, we prestaged the input data to RAM disk (*RAM Disk*). The prestaged configuration does not represent a typical use case, because multiple consumers of the data are not typically co-scheduled. However, it is useful to emphasize the performance impact of reading data from disk. On 32 cores of Polynya, the execution time of our workflow is reduced from 56 minutes for the disk configuration to 16 minutes for the hybrid configuration. In the prestaged case, the workflow executes in 4 minutes.

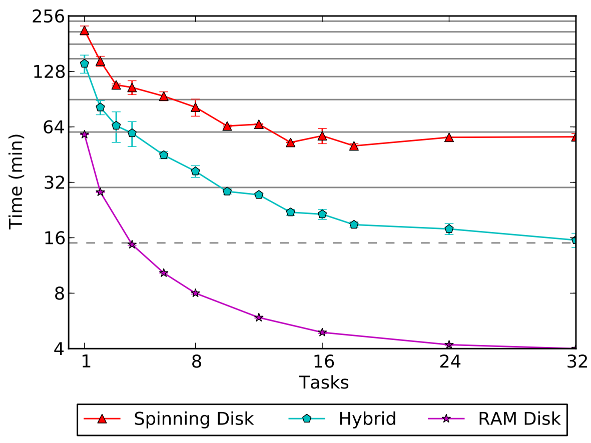
We interpret overall performance as a combination of node performance, aggregate I/O performance, and workflow parallelism. At small processor counts (e.g., np $\leq$ 8), all processors are busy, so the results are based on standard computational and I/O performance. As the processor count becomes large, the executing tasks are bound by the underlying platform's I/O performance. The shape of the curve thus describes the workflow's parallelism and its asymptotically decreasing efficiency. We therefore observe that on Polynya, the curves reduce to asymptotes based on the workflow's use of the high-performance storage, such that Polynya's older storage is outperformed by Dash and Nautilus.

4.3 Resolution, Throughput, and Performance

While most of our work on the AMWG workflow has emphasized the high-resolution 0.5° data, a substantial amount of production analysis is currently performed using data at a lower 1.0° resolution that is approximately four times smaller. We therefore compared the performance of the workflow with both resolutions using all-disk and prestaged configurations on Polynya to highlight the effect of resolution on performance (see Figure 6). The results show that for the two resolutions, the shapes of the curves using each storage system are similar, but the lower resolution approaches a lower asymptote. This is not surprising, as the workflows execute exactly the same pattern and number of tasks for each resolution thus producing the same



(a) Systems using GPFS Disk Storage



(b) Polynya Server using GPFS and RAM Storage

Figure 5: Parallel performance of the AMWG workflow on three major platforms using disk-based storage (5(a)) and on a single server using GPFS and RAM storage (5(b)). The hybrid configuration reads raw data from spinning disk and writes temporary and output data to RAM disk – the typical production configuration.

parallel efficiency characteristics, but the lower resolution case manipulates less data and therefore requires less I/O time leading to the offset between the resolution cases.

4.4 Preliminary Results with Dash Features

The results from the previous three subsections demonstrate that our workflow has similar performance on Nautilus and the portion of Dash configured as a Linux cluster, and that Polynya's RAM-based file system provides substantial performance benefit over both. However, the RAM-based storage solution that Polynya provides is potentially cost prohibitive. Dash's prototype architecture includes two features that might provide performance improvements in a more cost-effective solution: a flash-memory (solid-state disk) file system, and ScaleMP software-based single-system-image virtualization that includes a distributed RAM-based disk storage.

Given the high efficiency demonstrated with 8 processors, we first tested the effect of using flash memory storage on a single node of Dash for temporary and output data, while reading the input data from disk (a *hybrid* configuration similar to production use). We used the 1.0° test case because even with the constrained parallelism, the temporary data for the 0.5° test case could not fit in a single 64 GB flash disk. The use of flash disk reduced the time to execute the analysis from 9.5 minutes to 7.3, a speedup of 1.3. A similar speedup (1.5) is obtained on the lower-resolution 2.0° data set.

We also attempted to run the 0.5° and 1.0° workflows on Dash's vSMP node that would allow access to much more flash memory as well as a large RAM-based file system. Unfortunately, the external I/O throughput available on the early-evaluation prototype node was insufficient to execute the workflow in a realistic fashion using either an all-disk or hybrid configuration. We prestaged the data to the vSMP-based RAM disk manually (requiring ∼45 minutes) and were able to execute the workflow, but execution times were not competitive with the previous results on Dash without vSMP. Even without the special flash or memory file systems available in the vSMP configuration, Dash already provided competitive performance compared to the other systems.

As we concluded development on Dash, we worked with SDSC to evaluate preliminary network, I/O, and file system configurations in search of additional performance improvements. For example, we ran the workflow using SDSC's prototype of the RNA Networks "MVX memory cloud" technology used to create a RAM disk on one node by aggregating memory from multiple separate nodes. Using the remote memory, the 1.0° test case required 172.6 seconds on MVX compared to 129.4 on Polynya, and the 0.5° test case required 646.8 seconds on MVX compared to 481.9 on Polynya. These tests show that the MVX-provided remotely aggregated RAM disk is competitive with the single large-RAM system; in the future, we are planning on running tests in other configurations to further examine the application of this technology to the workflow.

5. DISCUSSION

The parallel implementation of the AMWG diagnostic workflow highlights the different factors influencing performance and scalability of typical computation-bound tightly-coupled parallel applications (such as the climate model generating the data used here) versus data-intensive analysis workflows. Our data analysis workflow is heavily dependent on each node's throughput to storage, including factors such as the type and number of network connections, their throughput, and the ability of the system's expansion bus to achieve that throughput. The scalability of our data-intensive workflow is limited by I/O performance versus computation-bound problems which are limited by floating-point and memory subsystem performance. Scalability of this workflow is also bound by the number of available tasks and their dependencies. Our workflow, which represents one particular invocation of the AMWG diagnostics scripts, consists of only 204 short data transformation tasks grouped in analysis chains with limited internal parallelism. We

```
rule .*.nc DEFAULT
rule .* DIRECT /scratch/34568/swift-tmp
```

Figure 7: A Swift CDM configuration file for controlling data management.

believe that the most important consideration is that the parallel workflow can use even small numbers of nodes and processors to achieve a significant improvement in execution time. Moreover, the same script and infrastructure can be used to execute the diagnostic for multiple years, with scalability limited only by the number of data files and the size of the system's scratch space.

Based on our experience using Swift to automate a data-intensive workflow, we have identified several features that may be beneficial to the language and execution system. First, our workflow generates a large number of anonymous temporary files that are used to connect two workflow steps together and are then never referenced again. Swift only deletes temporary files after the workflow has completed. In the case of our workflow, this means that twice the input data size is generated and left on disk in temporary directories. While we added artificial dependencies to our scripts to trigger procedures that call `rm` to remove files after they are no longer needed, the automatic deletion of temporary files could be provided by Swift as a garbage collection feature. Indeed, this feature has been added to the Swift source code repository between our experiments and this publication.

Another possible improvement in the Swift runtime is support for customizable, or dependency-aware, task scheduling. In our application, we occasionally encountered situations where Swift would execute tasks that generated few dependencies, leaving processors idle before triggering a task that generated more work. It may be beneficial if the execution system could estimate the task branching factor among the tasks elegible for execution and run those that might produce more work first, or use profile-based scheduling, to avoid situations where the number of running tasks artificially drops due to dependency ordering.

Finally, Swift may benefit from improved integration of language support for data management. Swift's collective data management (CDM) feature, described in Section 3.2, was critical to achieving the obtained performance and scalability of our workflow. our of eight cores on a node. Removing the However, its usage is not integrated with the declaration of opaque file types in SwiftScript itself. For example, a CDM configuration file similar to those used for our workflow contains two rules (see Figure 7). The first rule indicates that all files that match that pattern $*.nc$ should be staged or copied from their original location to the site specific directory. This rule is necessary to make sure that all output files are returned to the desired final output location. The second rule indicates that all other files should not be staged but should use the direct method from the site run directory "/scratch/34568/swift-tmp". An unwanted side-effect of the first rule is that it matches the input files in addition to the output files. This causes all input to be staged-in from the input directory to the Swift site run directory before task execution. We avoided the unnecessary stage-in of read-only data in the Swift code in Figure 2 by treating read-only input files as strings containing file names, and explicitly prepending a leading forward slash. More transparent and integrated data management control would be beneficial.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we described the parallelization of a workflow that analyzes the output of the atmospheric model component of a coupled earth system model. The parallelization of the AMWG diagnostic package statistics generation workflow was greatly simplified through our use of the Swift data-centric scripting language. To the best of our knowledge, our workflow represents the most data-intensive use of Swift currently attempted. In particular, the size of our input files are quite large relative to the computational cost of each task in the workflow, so our workflow places a premium on precise control of data movement. We found Swift to be a powerful and efficient means to describe and execute the workflow, and based on our experiences, we provided several suggestions on how the Swift language could be improved through enhanced data management support for input and temporary files. Our manual addition of remove statements to reduce temporary data storage requirements was a reminder of how the construction of explicit dependencies within a workflow can be quite painful and time consuming.

We also evaluated the ability of several data-intensive systems to execute our data analysis workflow. Our most important results demonstrated that this workflow, which executes only 511 tasks in the tested configuration, can obtain substantial reductions in execution time through parallelism with only 8 or 16 processors on typical cluster nodes. Moreover, we identified a hybrid execution mode where input is read from disk and temporary data is placed on high-performance RAM-based storage, that can further reduce execution time even in realistic production scenarios.

In the future, we plan to finalize the integration of this work with the broader AMWG diagnostic suite such that it may be easily used by other researchers for realistic production scenarios. First, we are preparing Swift configurations for a variety of systems at resource providers popular with our colleagues and other climate research groups. Second, we are working with collaborators in the ParViz project [8] to expand the parallelization of the AMWG diagnostic suite to include the plot generation stage. These changes will make the Swift-parallel AMWG diagnostic suite quite useful for use by a broad community of researchers. Given our positive experience with Swift, we believe this will spur greater adoption of Swift for analysis workflows in our community.

We are also investigating methods to insert the parallel AMWG diagnostic in production workflows while reducing the use of spinning disk. For example, when the data is prestaged to Polynya's RAM disk, a speedup of 52× can be obtained versus the original serial implementation. We envision reading the data from the supercomputer's disk just once to stream it to Polynya's RAM disk. Once on Polynya, the data would be simultaneously stored to the long-term archive while being analyzed using the parallel diagnostics suite. This would reduce the use of disk storage steps in the overall workflow to just one, and the cost of the analysis would be absorbed by the cost of the transfer and upload to archive.

Finally, this work automated only the data analysis component of a much larger scientific workflow that includes execution at one supercomputer center, analysis at another, storage at both, and data transfers between the two. In the future, we plan to broaden our investigation of automation to the overarching workflow itself. In particular, we plan to examine strategies and tools, such as Kepler and NCAR's existing Earth System Grid infrastructure [6], that could be used to orchestrate all of the components of the scientific workflow.

ACKNOWLEDGMENTS

We would like to thank Michael Wilde, Justin Wozniak, Mihael Hategan, and Allan Espinosa at the University of Chicago Computational Institute and Argonne National Laboratory for their extensive assistance using the Swift workflow management system, including help with CDM and quick bug fixes over weekends. Thanks to Allan Snavely, Shawn Strande, Adam Jundt, Jeffrey Bennett, and Eva Hocks at the San Diego Supercomputer Center (SDSC) for their assistance with Dash.

Computational time was provided on Dash and Nautilus by the San Diego Supercomputer Center and the National Institute of Computational Sciences, respectively, through TeraGrid grant TG-ASC100040 from the National Science Foundation. This work was supported by National Science Foundation (NSF) grants #OCI-0749206 and #CCF-0937939, as well as NSF sponsorship of the National Center for Atmospheric Research.

7. REFERENCES

[1] AMWG Diagnostics Package, August 2004. http://www.cgd.ucar.edu/cms/diagnostics/.

[2] CISL's NCAR Command Language (NCL), December 2010. http://www.ncl.ucar.edu/.

[3] Community Earth System Model (CESM) 1.0, November 2010. http://www.cesm.ucar.edu/models/cesm1.0/.

[4] The Kepler Project, December 2010. http://kepler-project.org/.

[5] NCO Homepage, November 2010. http://nco.sourceforge.net/.

[6] D. Bernholdt, S. Bharathi, D. Brown, K. Chanchio, M. Chen, A. Chervenak, L. Cinquini, B. Drach, I. Foster, P. Fox, J. Garcia, C. Kesselman, R. Markel, D. Middleton, V. Nefedova, L. Pouchard, A. Shoshani, A. Sim, G. Strand, and D. Williams. The earth system grid: Supporting the next generation of climate modeling research. *Proceedings of the IEEE*, 93(3):485–495, 2005.

[7] J. He, A. Jagatheesan, S. Gupta, J. Bennett, and A. Snavely. DASH: a recipe for a flash-based data intensive supercomputer. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, New Orleans, LA, November 2010.

[8] R. Jacob. Personal Conversation, November 2010. Argoone National Laboratory.

[9] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the Kepler system. *Concurr. Comput. : Pract. Exper.*, 18:1039–1065, August 2006.

[10] V. Nefedova, R. Jacob, I. Foster, Z. Liu, Y. Liu, E. Deelman, G. Mehta, M.-H. Su, and K. Vahi. Automating climate science: Large ensemble simulations on the TeraGrid with the GriPhyN Virtual Data System. In *Second IEEE International Conference on e-Science and Grid Computing*, page 32, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[11] I. Raicu, I. Foster, and Y. Zhao. Many-task computing for grids and supercomputers. In *Proceedings of the IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS08)*, 2008.

[12] M. Wilde, I. Foster, K. Iskra, P. Beckman, Z. Zhang, A. Espinosa, M. Hategan, B. Clifford, and I. Raicu. Parallel scripting for applications at the petascale and beyond. *Computer*, 42:50–60, 2009.

[13] J. M. Wozniak. Swift-user mailing list: handling prestaged data on multiple sites, May 2010. http://mail.ci.uchicago.edu/pipermail/swift-user/2010-May/001479.html.

[14] J. M. Wozniak and M. Wilde. Case studies in storage access by loosely coupled petascale applications. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, PDSW '09, pages 16–20, New York, NY, USA, 2009.

[15] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, V. Nefedova, I. Raicu, T. Stef-Praun, and M. Wilde. Swift: Fast, reliable, loosely coupled parallel computation. In *IEEE Congress on Services*, pages 199–206, 2007.