Title

Clustered Workflow Execution of Retargeted Data Analysis Scripts**This work is supported by the National Science Foundation under grants ATM-0231380 and IIS-0431203.

ISBN

978-1-4244-4237-9

Authors

Wang, Daniel L
Zender, Charles S
Jenks, Stephen F

Publication Date

2008-05-01

DOI

10.1109/ccgrid.2008.69

Peer reviewed

Clustered Workflow Execution of Retargeted Data Analysis Scripts *

Daniel L. Wang, Charles S. Zender, and Stephen F. Jenks
University of California, Irvine
Irvine, California, USA
{wangd, zender, stephen.jenks}@uci.edu

Abstract

Supercomputing advances have enabled computational science data volumes to grow at ever increasing rates, commonly resulting in more data produced than can be practically analyzed. Whole-dataset download costs have grown to impractical heights, even with multi-Gbps networks, forcing scientists to rely on server-side subsetting and limiting the scope of data they can analyze on a workstation.

Our system supplements existing scientific data services with lightweight computational capability, providing a means of safely relocating analysis from the desktop to the server where clustered execution can be coordinated, exploiting data locality, reducing unnecessary data transfer, and providing end-users with results several times faster. We show how dataflow and other compiler-inspired analyses of shell scripts of scientists' most common analysis tools enables parallelization and optimizations in disk and network I/O bandwidth. We benchmark using an actual geoscience analysis script, illustrating the crucial performance gains of extracting workflows defined in scripts and optimizing their execution. Current results quantify significant improvements in performance, showing the promise of bringing transparent high-performance analysis to the scientist's desktop.

1 Introduction

Now that multi-core processors have flooded the mainstream, parallel programming and applications have become necessities in delivering ever-increasing performance. While scientists are promised virtual supercomputers by computational grids and universal, locatable, replicated, and distributed filesystems in data grids, few solutions address the problem of how to analyze this new avalanche of data. Simple terabyte data analyses are too computationally simple to submit to compute grids, and too bulky to download from data grids, which expect transfer sizes of 100 megabytes or less–sizes that are orders of magnitude too small.

Our *medium-scale* system bridges the gap between tera- or petascale data production and hundred-megabyte desktop data inspection and visualization. In this paper we show how the Script Workflow Analysis for MultiProcessing (SWAMP) system enables generation of directed acyclic graph (DAG) workflows automatically from unmodified shell scripts, and the resulting parallelization and execution of these workflows on commodity cluster installations. We explain the case for using script-defined workflows, the special importance of locality that motivates integrating computation with data service, and problems with scheduling algorithms not fundamentally influenced by I/O considerations. Our results show large performance improvements, which, coupled with a nearly transparent interface, should enable scientists to analyze datasets of far broader scope. By drastically reducing the barrier to data-intensive data analysis, we enable scientists to ask questions of a different, more expansive nature.

2 Scripted Workflows

2.1 Background

Grid computing implementations have focused on two approaches. Computational grids expose raw computational power and data grids expose distributed, replicated storage. Both provide a unified interface to heterogeneous resources.

Workflow scheduling is a well-studied topic in the form of business workflows[20] and constraint systems [8]. Research in computational workflows has focused on optimizing performance and resource utilization in machines, clusters, batch systems, and grids. Current research, however, has shown the difficulty of applying workflow technology in the sciences without careful understanding of scientists' usage.

*This work is supported by the National Science Foundation under grants ATM-0231380 and IIS-0431203.

978-0-7695-3156-4/08 $25.00 © 2008 IEEE
DOI 10.1109/CCGRID.2008.69

2.2 Existing Systems

Grid workflow systems such as Pegasus[9] and Kepler[2] require workflows to be specified using XML [3] or other custom workflow languages, usually providing a GUI design tool to assist users in workflow construction. These systems allow workflows to be defined flexibly and explicitly so that they may be efficiently scheduled with a workflow-aware grid scheduler, and are suited for operational workflows, i.e. workflows whose parameters may change rapidly, but whose task graphs are approximately constant for particular analyses. In some systems e.g. [9], users create abstract workflows which are subsequently transformed into concrete workflows by a planner, whereas other systems execute workflows directly[1]. Examples of workflow specification standards include Grid Services Flow Language (GSFL) [15], IBM's Web Services Flow Language (WSFL) [16], and the Open Grid Services Architecture Basic Execution Standard (OGSA-BES) [12]. The Job Submission Description Language (JSDL) [4] standardizes methods for single job submission, and can be used as part of workflows. While these workflow languages employ similar semantics, they are mutually incompatible, with few exceptions. Because of such customization, these workflows are difficult to develop and debug outside of their own systems.

An important characteristic common to many existing workflow systems is their target task size. Indeed, computational workflow systems are almost universally targeted at the highest computational tasks. Thus, they provide rich, full-featured workflow definition, allowing developers to fine-tune their workflows for optimum performance. Indeed, the significant effort required to define such workflows is amortized over execution times often measured in days, and over many workflow executions. For scientists who conceive of an idea in the morning and want answers in the afternoon, however, such features make those systems unsuitable for use with simple one-off data analyses.

2.3 Workflow Definition Alternatives

An alternate method of defining workflows utilizes existing non-workflow-specific programming and scripting languages. This complicates the task of mapping tasks to processors, and likely requires a new implementation of the language compiler or interpreter. These languages also lack the ability to explicitly specify workflow-specific constructs. However, the benefits of established debugging practices, existing familiarity and comfort, and transparency of execution make such a workflow system far more usable to many computational scientists. Such a system could be wrapped in an otherwise vanilla shell interpreter that seamlessly and automatically invokes the scripted workflow engine where appropriate, completing the transparency of a distributed, scripted workflow system.

Workflows are, by definition, sets of tasks bound by causal/temporal dependencies. Dataflow languages such as SISAL [11], SAC [19], or other declarative or functional languages are natural candidates for workflow use, but their somewhat limited acceptance in the programming world makes them no more familiar than grid workflow languages to most users. Existing parallel processing models such as skeletons in structured parallel programming [7] could be used, but are as unfamiliar as dataflow languages to non-computer scientists.

Other, more mainstream programming languages specify programs in the form of causally and temporally dependent expressions and statements that can be considered low-level workflows. One project, GRID Superscalar [5] provides a way to specify and execute workflows defined in a Perl language-based syntax, illustrating the practicality of using a mainstream imperative language for workflow definition.

Most general-purpose programming languages are designed with semantics operating at granularities too fine to be practically retargeted on grid and distributed systems. High-level "glue" languages like shell-scripting languages, on the other hand, exist to connect sequences of dependent operations, such as applications written in common programming languages.

2.4 Scripted Scientific Analysis

Because they are commonly used to bind sequences of applications together, shell-scripting languages are well-suited to defining workflows. Indeed, scientists often analyze datasets using sequences of executables operating at the file or dataset levels, bound together in shell scripts, where more flexible tools as Matlab [13] or custom Fortran code are too cumbersome or too slow because of their fine granularity. Shell-scripting's flexibility in invoking *any* binary executable is also the primary difficulty in optimization and parallelization. Argument syntax and semantics of program invocations pose the greatest problem, thus forcing any prospective optimizer to treat invocations as opaque functions with unpredictable side-effects, and preventing basic optimization. Although the practical impossibility of extracting a program's argument syntax and semantics in the general case seems to present an insurmountable barrier to optimization, a workable approach is to define limits in syntax while maintaining the familiar usability of shell-scripting.

We have observed that many shell-scripts, especially those specifying long workflows, utilize a small subset of the full syntax to invoke a similarly limited set of programs. Large and meaningful data analyses involving hundreds of gigabytes can be specified using sequences of invo-

```
srca=model1.nc
srcb=model2.nc
ncdiff \$srca \$srcb a-b.nc
ncbo --mult a-b.nc a-b.nc sqr.nc
ncwa sq.nc msqab.nc
```

Figure 1. Scripted computation of mean squared difference

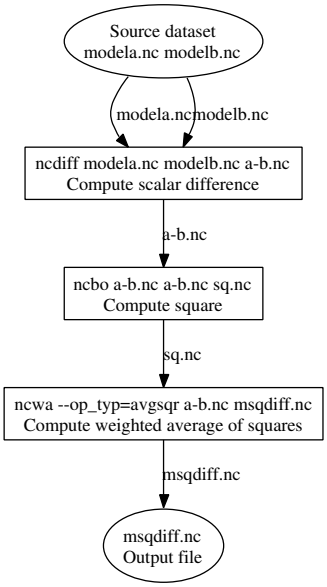

Figure 2. Resultant DAG from script in Figure 1

cations of handful of executables, each representing a well-defined processing component. These scripts seldom utilize control-flow constructs and rarely involve syntax more complex than simple looping. Geoscience data analysis, in particular, frequently involves relatively simple computations that iterate over large datasets to produce summary statistics such as spatio-temporal averaging. The reduced syntax and program set forms a *domain-specific* language which is tractable to implement. The code in Figure 1 illustrates a sequence of NCO (see section 4.1) commands that compute the mean squared difference between all variables in two different files. This code is a simple example of a script that can be submitted to our system. Its DAG workflow representation is illustrated in Figure 2.

Other scientific workflow systems such as Taverna[18] and Triana[21] have recognized the immense utility in providing scientists with easy ways to compose and execute workflows constructed with well-defined data sources and a limited set of operations. In both systems, the interface is graphical, and simple enough to empower scientists with the ability to construct ad-hoc workflows without worrying about execution.

A disadvantage of shell-script specification for workflows is its inability to explicitly specify traditional business and grid workflow characteristics. Shell-scripts may therefore be unsuitable for those applications. However, by transforming ordinary shell scripts into workflows, a system can bring high-performance concurrent execution to the daily data analysis tasks of an individual scientist. By operating at the granularity of program invocation, this syntax could allow for workflows to be generically constructed from existing and future programs, provided that the system were extended to support their particular argument formats. Additionally, providing workflow parallelism *independent* of programs means that an additional level of parallelism beyond what the programs themselves provide, whether they be multithreaded, process-forked, MPI- or OpenMP-enabled.

2.5 Relocation

The execution of an arbitrary program depends on its context. Executables are isolated by virtual memory, but their behavior often affects and depends on an underlying filesystem. Indeed, in considering a sequence of program invocations in a workflow as an application, the filesystem becomes a context much as physical memory is to an application. Just as traditional compiled programs must not directly access memory with real physical addresses, script workflows cannot be safely executed in general using directly specified filenames. An execution engine must safely remap script filenames to physical filenames in order to safely execute. Mapping workflows specified in grid workflow languages have similar problems and rely on explicit dependency specification and well-defined semantics.

Mapping script filenames can be considered similar to register renaming in modern out-of-order dynamically-scheduled processor architectures. Script filenames, like register specifiers, can be dynamically mapped to physical equivalents in order to eliminate write-after-write (WAW), and write-after-read (WAR) dependencies. Script workflows however, due to their typically low frequency of control-flow dependencies, stand to benefit far more from such renaming.

3 I/O-Driven Scheduling

The data produced by ever advancing high performance computing continues to grow faster than such data can be analyzed and digested. Grid workflow projects recognize data movement as a crucial factor in scheduling workflow

execution, but continue to target computationally-intense workflows where bandwidth costs remain small in comparison to computation costs. With computational performance increasing rapidly and long-haul network bandwidth increasing relatively slowly in comparison, overall time for end-user data analysis depends increasingly on network bandwidth. In some classes of applications, such as geoscience data analysis, these network transfer times have become dominant. Because of the enormous cost of downloading data produced remotely, scientists only infrequently perform such analyses, restricting their scope to small subsets when downloading. On-demand dataset subsetting has subsequently become a common feature in data centers such as the National Center for Atmospheric Research's Community Data Portal and the HUGO Genomic Nomenclature Database, sometimes comprising 99% of download requests [10].

In the class of applications exemplified by geoscience data analysis, data movement costs often exceed computational costs, so a different approach to scheduling is needed. Traditional scheduling methods typically consider data dependence as an ordering constraint that affects correctness, not as a primary determiner of performance. Instructions or tasks that can be parallelized are issued in parallel as soon as their inputs are available, moving input data to idle processing units appropriately. When the input data to be moved is measured in gigabytes or more, it is clear that it may be more expensive to parallelize a series of processing tasks among multiple nodes than it is to allow execution on a single node.

I/O-driven scheduling implies that scheduling decisions are made to minimize data movement costs rather than to maximize processor utilization. Cases such as the above where parallelization is unprofitable should ideally be detected by such a scheduler. Unfortunately, such I/O costs are difficult to determine *a priori* in the general case, similarly as it is difficult to predict a program's runtime heap memory usage. In workflow execution, input data sizes and thus approximate transfer costs for a future task are known once the task is ready, thus allowing a dynamic scheduler to utilize such information "just-in-time," for future scheduling, but less optimally compared to knowing transfer costs in advance.

Although transfer costs are difficult to predict in the general case, certain workflows are composed of operations or tasks whose output characteristics, including size, are deterministic from input parameters and data characteristics. Such operations are common in application domains like geoscience data analysis. Representing workflows as directed acyclic graphs, this added information allows approximation of transfer costs and possibly computation costs to form edge weights, facilitating the search for an efficient schedule as a form of graph partitioning.

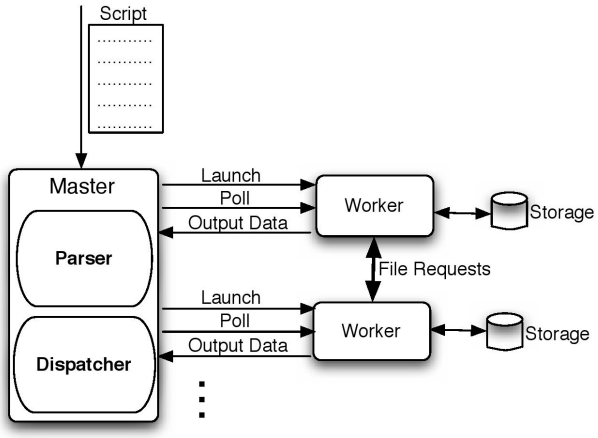

Figure 3. Block diagram of our system

Computer-aided design of VLSI layouts has historically used such partitioning algorithms to minimize wire count in layout, due to increasing on-chip signal latency. In some of these algorithms, graph vertices have been duplicated to reduce edge count. Similarly, an I/O driven workflow scheduler may find it more efficient to duplicate an operation rather than incur transfer costs.

In recognition of the high cost of data movement and the explosion of data production, it is clear that data-intensive workflow execution should almost always occur near the source of production. Since end-user (i.e. scientist) data analysis is almost always representable as a data-intensive workflow, this makes data-computation co-location an inevitability.

4 Architecture

We implemented our system, a workflow execution framework to provide computation services integrated with data services. Our first target application is geoscience data analysis, whose data-intensive and computationally light characteristics exemplify a class of computing which is not well-served by existing solutions and will grow in importance as data production continues to explode. A block diagram of our architecture is shown in Figure 3.

4.1 Interface

We chose a shell-script syntax for workflow specification, motivated by the heavy use of shell-scripting in geoscience data analysis. The syntax supports invocation of a set of executables, the netCDF Operators (NCO)[23] each of which operate on variables at a file-level granularity. These operators are used individually interactively, and are

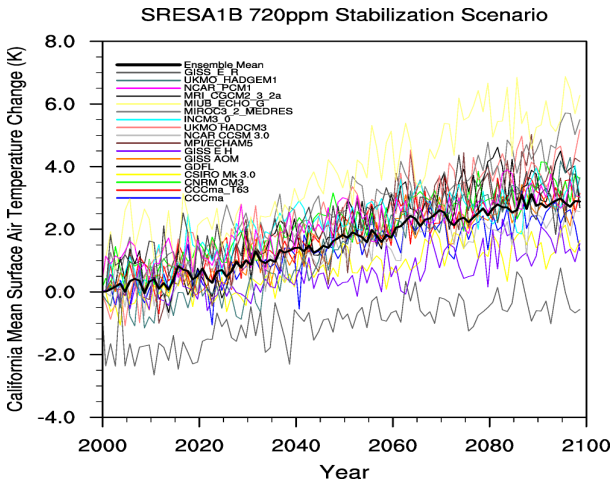

Figure 4. A comparison of 16 different IPCC climate simulations

often composed into sequences which define a rich set of analyses on atmospheric, oceanic, and other gridded data. Scripts of these operators commonly involve input data sizes ranging from gigabytes to terabytes and result in output data sizes far smaller. The chart in Figure 4 exemplifies our target class of applications. This plot compares the predicted surface air temperature change in California from 2000 to 2100 according to 16 different Intergovernmental Panel on Climate Change [14] models. To create this plot, more than a terabyte of data was downloaded from remote sources and analyzed using the same operators and scripts we target in our system. The resultant data was just a few megabytes.

Our shell-script syntax supports shell/environment variables and "#"-delimited comments, as well as standard single and double-quoting. Control flow constructs, such as for-loops and if-statements, and limited backquoted expressions are also supported. A large fraction of existing scripts require no modification beyond correcting file paths. Output files are differentiated from temporary files, or intermediate results, using a simple heuristic– output files are defined as files produced at leaves of the workflow's DAG.

4.2 Execution Engine

Execution begins with a scientist's use of a desktop client to submit of the script to the SWAMP service at the data center. The syntax is straightforward, e.g. `swamp_client.py -u http://data.center.org:8080/SWAMP analysis.swamp`, and the submitted script is easy to debug, because it can be executed on a personal workstation.

Scripts submitted to the system are parsed for shell syntax and to apply filename remapping, resulting in a directed acyclic graph of the workflow. Each line must be parsed for variable definition and interpolation, executable recognition, and comment handling. Executable recognition verifies program invocations against the custom set of allowed executable calls, exploiting knowledge of argument syntax to determine inputs and outputs from raw argument lists. Inputs are matched against outputs of previous lines or the available source data, performing appropriate filename globbing to handle wildcards in both cases. Parsing shell and environment variable definition and interpolation is currently the most computationally intensive part of parsing, forming roughly 80% of overall parsing time. We attribute this performance to the use of a pure Python parsing library that favored readability and ease of implementation over performance.

After the parse step, the system searches the parsed list of commands that are *ready to run*, in other words, commands whose inputs do not belong to any unfinished commands, creating a ready list. After initialization, the engine begins its execute phase, in which it removes commands from the ready list and dispatches them to worker nodes with idle CPU slots. Commands in the ready list are sorted according to their line number in the original script, allowing overall execution to stay close to a sequential schedule and provide partial results earlier to end-users. For each command, the engine selects an idle slot from nodes that host one or more of the launching command's input files, falling back to round-robin if none of the hosting nodes have free slots. This simple scheme significantly reduces I/O transfer, although more complex schemes will be explored in the future.

Each node hosts a replica of the data server's data set on local disk, reducing input data contention. This is a reasonable limitation, because worker nodes are expected to have fast interconnections with each other and the master. However, since most commands in a workflow operate on results of other commands, some amount of I/O transfer is almost always required. Worker nodes must, therefore, fetch inputs of commands assigned to them on-demand, requesting them from the peer workers that produced them. Because the system favors spreading load among available nodes, no single node becomes a bottleneck for source data. To further reduce disk contention, worker nodes write all command output to a Linux tmpfs RAM disk, switching to local disk after a "high water mark" has been reached in memory usage. Exploiting RAM is crucial in parallelizing data-intensive operations and the RAM disk operates as an explicit caching mechanism to prevent temporary files from being written to disk. This dramatically reduces physical disk contention, shifting disk access behavior towards se-

quential rather than random.

Communication between nodes occurs using a combination of SOAP [6] and HTTP. Worker nodes operate as SOAP and HTTP web services, using SOAP to service command dispatching, status, and other requests, while deferring to HTTP for bulk data transfer. Although each worker node has its own replica of the entire script-referencable dataset, no shared filesystem is used. We chose to restrict data movement to HTTP to ease portability to an environment where worker nodes are distributed geographically on the Internet.

For the scientist, execution ends when the client code downloads and writes the output files to disk on her workstation. Because the inputs do not need to be downloaded, she is able to get results to an analysis of data much larger in scope and size than what her workstation could handle.

4.3 Limitations and Comparisons

The system implementation has significant limitations that limit its applicability, but are acceptable for our target domain. Because it is impossible to automatically determine a program's argument format, our system is limited to handling the set of executables it is programmed to parse. Extension to support additional programs is possible by with added parsing code. A comparison to GRID superscalar [5], which is similar in goals and approach, may be instructive. In contrast, GRID superscalar supports arbitrary programs, as long as their task parameters are implemented in an interface definition language, but user workflows must be specified in a Perl-like language. Since our system is intended to be deployed for general access at scientific data centers and reduced numbers of programs are easier to secure, this is not a serious limitation. Indeed, data center operators have remarked to us that a system that allowed client-side specification of arbitrary programs would be unsuitable for public deployment. If additional program support is required, we anticipate that programmer effort to support new programs is comparable to that of other workflow systems such as GRID superscalar, requiring between a dozen lines of Python code to support programs with simple argument formats to a few dozen for more complex programs.

In clustered execution, our system assumes that the entire data set is replicated among executing cluster nodes. Although the system can be easily configured so that nodes may access the same data via NFS from the master node(or equivalent), this comes at a heavy performance cost, considering that our application domain is frequently performance-limited by disk I/O. Remote data sources can be accessed in the existing system, but since we have not yet implemented outward task migration, such accesses may incur significant download cost. We are also investigating support for enviroments where each executing node has different sets of input data. Worker nodes currently exchange intermediate data (but not original input data) with each other as necessary, although the scheduler attempts to minimize this by preferentially scheduling tasks where their input data resides. Our system's dispatcher is aware of data locality, and could be modified to handle workers without data replicas, but we have currently deferred such a feature in favor of enhancing scientist usability.

Task coordination is completely done by the master node, which may limit scalability for large scripts and large numbers of workers. Parsing incurs significant costs for extremely long (tens of thousands of lines) scripts, but is difficult to parallelize or distribute, just as it is difficult for a C-compiler to parallelize the compilation of a single file. An earlier implementation utilized communicating peer workers which scheduled work and reported progress through a single database, but suffered in throughput due to contention on the shared data.

5 Experimental Results

5.1 Benchmark

We tested our system with a shell script that applies custom subsampling to an input dataset of 8230MB, producing 228MB of output and 26GB in intermediate (temporary) files. Geoscience data analysis represents a class of data intensive applications which are not computationally intensive and not *search*-based in nature, as in computational high energy physics. Rather, they are analyses involving iteration over entire datasets, reducing raw values to summary statistics that illustrate large trends. The benchmark case we have chosen illustrates features common to a large segment of geoscience analysis– large input datasets reduced by computationally simple operations to summary statistics that are a small fraction of the input size. Figure 5 summarizes the benchmark workflow.

The benchmark was tested on a cluster of dual Opteron 270s with 16GB of memory and dual 500GB SATA drives, running CentOS 4.3 Linux. We tested execution with one master and one, two, or three worker worker nodes, all configured identically. We varied the number of executable slots (number of parallel processes) allowed on each node in order to study the benefit of using using more cores (four available per-node) versus the increased I/O contention. Table 1 and Figure 6 summarize the test results. Transfer times listed are estimated assuming 3MBytes/s ($3 * 2^{20}$) bandwidth, based on NPAD *pathdiag* [17] measurement of 30Mbits/s bandwidth between UCI and the National Center for Atmospheric Research(NCAR). In our example, a scientist can avoid downloading nearly 8GB, obtaining just 228MB of output rather than the entire input dataset and

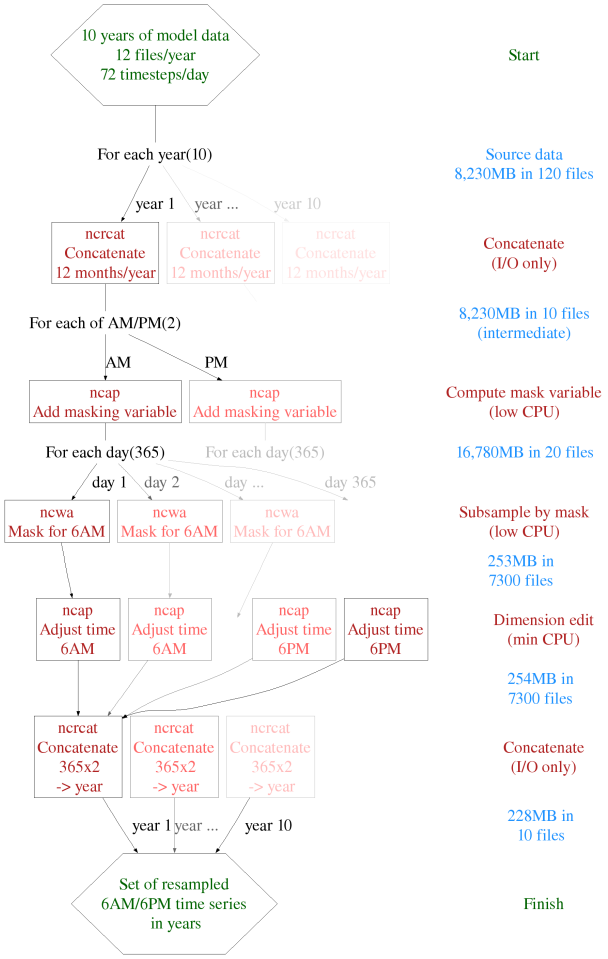

Figure 5. DAG of Benchmark Workflow, greatly simplified

Table 1. Overall Performance Table(times in mm:ss)

	Baseline	1 worker x 1 slot	3 workers x 4 slots
Parsing Time	-	4:29	4:29
Computation Time	53:18	86:27	15:06
Transfer Time	45:43	1:16	1:16
Total Time	99:01	87:43	20:51
Normalized Speed	1.00	1.12	4.75

saving 44 minutes of transfer time. Our baseline case shows the execution time of the original shell script and the time to download the input data, and takes 99 minutes overall.

The parsing stage incurs significant overhead, due to shell and environment variable handling in pure Python code. In our benchmark case consisting of 14,000 commands in 22,000 lines, parsing accounts for approximately 5 minutes of overhead. This benchmark represents a practical upper-bound on the length of script expected and can be considered a "stress-test." We expect the parsing overhead to be reduced to insignificance for most scripts in future versions by implementing an early start mechanism that dispatches commands as soon as they are discovered as ready, before parsing completes.

We first compare our system's end-to-end performance against the baseline case of non co-located operation, illustrating the enormous speedup from co-locating computation with data. Secondly, we compare performance of our system with a single worker node, while varying the number of available slots, illustrating the degree of parallelism available in a typical geoscience workflow and our system's ability to extract and exploit it. Thirdly, we compare system performance while varying the number of worker nodes, illustrating the performance potential despite increased I/O traffic.

5.2 Overall Performance

In Table 1 and graphically in Figure 6, we see that savings in transfer time dominate the overall benefit from using our system. Even when configured to use only one slot on one worker, transfer time savings make up for the considerable overhead. Recalling that transfer time was calculated using 30Mbit/s as effective bandwidth and that actual effective bandwidth to the data center may be far less, the importance of reducing unnecessary network transfer cannot be understated.

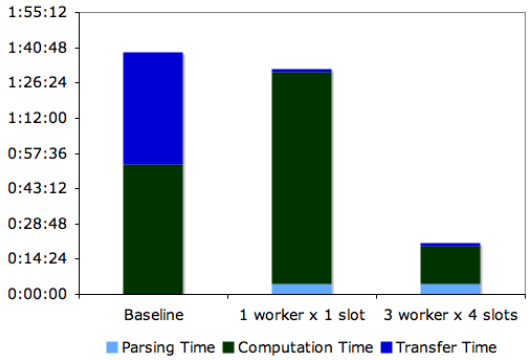

Figure 6. Overall Performance

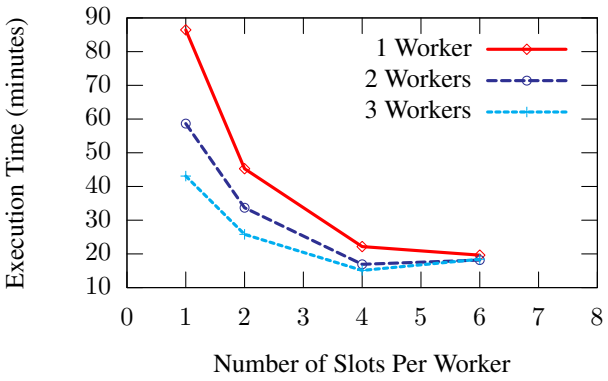

Figure 8. Computational Time: Multi-worker performance is limited by I/O

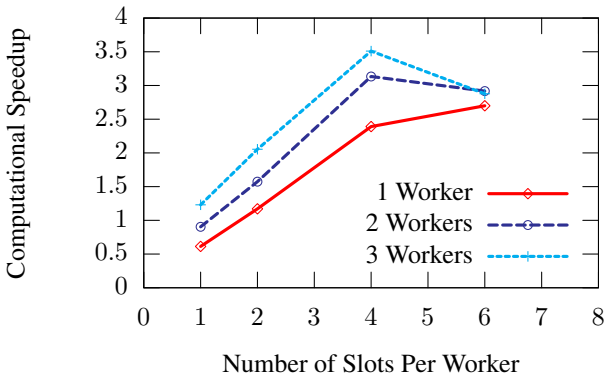

Figure 7. Relative Computational Speed: Single-worker performance saturates while multi-worker performance experiences I/O latency and contention issues

The single-worker single-slot performance in Table 1 illustrates the maximum overhead incurred by using our system. Parsing overhead can be largely eliminated as described in Section 5.1 and coarser-grained scheduling should significantly reduce the remaining overhead. Aside from parsing overhead, we see that the computational performance is unusually slow. This can be explained due to communication overhead from *dispatching commands to* and *polling from* workers. Since the master sleeps 100ms between polling nodes for completion, idleness from polling cycles may contribute significant overhead. With 14640 commands to be executed in the benchmark script, and assuming that polling inaccuracy is uniformly distributed between 0 to 100ms, we estimate that coarse polling adds $50ms \times 14640 = 732s$ or 12.2 minutes of slack computational time. Discounting the time for each poll, we can therefore estimate the polling overhead in each configuration as 12.2 minutes divided by the average number of commands in flight. We initially implemented the polling mechanism as an alternative to keeping open connections between master and worker, or allowing workers to initiate communication to the master (as a sort of back-channel). Polling may need to be reconsidered in light of this observed overhead. In our benchmark script, all slots are full during the entire run time, except near workflow completion, when there are fewer unfinished jobs than the total number of slots. The case of 3 workers with 6 slots each, given in Figures 7 and 8, thus has an approximate overhead of 41 seconds.

Figures 7 and 8 and Table 2 provide details of the computational performance of the system. Times include scheduling and execution overhead as well as peer worker file trans-

Table 2. Computational Time (times in minutes)

	Slots per worker			
	1	2	4	6
1 worker	86.4	45.3	22.2	19.6
2 workers	58.6	33.7	16.9	18.2
3 workers	43.1	25.8	15.1	18.4

fer overhead, but not parsing overhead or end-user transfer time. Speedup is normalized against the baseline case where a standard shell (GNU `bash`) is used instead of our system.

5.3 Single-worker Performance

To illustrate the scalability of the system when multiple processing cores are available on a single node, consider the single worker case, whose performance is tracked as the solid red line in Figures 7 and 8. In the case with only one slot, performance is slow at ≈ 0.6 speedup, owing to polling slack and task management overhead, but the system readily scales to use additional processing cores. Because our test hardware has 4 cores per node, performance saturates with 4 configured slots, gaining slightly with 6 slots in the single node case due to reduced polling slack and other dispatch and finishing latencies. It is worth noting that without the I/O optimization for intermediate files discussed in Section 4.2, disk contention outweighs benefits from parallelism [22].

5.4 Multi-worker performance

Our system's behavior with multiple worker nodes illustrates the importance of I/O management. Looking at Figure 7 and Figure 8 again, a number of notable features emerge. First, we see that adding more worker nodes has limited benefit. Though three nodes, each with a single slot, have similar capabilities to a single node with three slots, we see that the former displays performance only slightly better than a single node with two slots. This 3x1 case illustrates distribution of data among nodes, increasing the likelihood that intermediate files need to be transferred. For this benchmark, much of the workflow operates on intermediate files (which, again, are optimized as described in Section 4.2, so the cost of intermediate file transfers and additional scheduling overhead was greater than the additional disk bandwidth available.

Another notable feature is the degradation of performance in both the two- and three-node cases when more than 4 slots are used. Increased I/O from peer data fetching was observed in system logs, and we believe that overprovisioning the nodes increases the potential for poorer distribution of input data, allowing lopsided data distribution to cause one or more nodes to spend too much time waiting for data. Another likelihood is that increased contention for both disk and processor at each node impacts input data fetch transfers between workers– a fetching node waits longer for its input data, and a source node suffers even greater I/O contention.

Because the number of commands in a script is almost always many times the worker node count, we are implementing coarser grained scheduling, i.e. scheduling blocks of dependent commands to nodes instead of single commands. This will reduce the communications between master and worker and should reduce file exchange between workers. This additional workflow partitioning is similar to flow partitioning in the VLSI layout design community and may be able to adapt similar methods. The coarsely-divided work could be dispatched to groups of workers, each with their own delegated master, resulting in tree-structured workflow management.

6 Conclusion

SWAMP is designed to provide scientists a simple means to safely execute medium-scale scientific data analysis at a remote data center rather than transferring terabytes of data to analyze. Because scientists are adept at composing simple analysis shell-script workflows that execute on workstations, our decision to adopt this syntax provides them with the ability to reuse their existing analysis scripts for remote computation/data services that exploit data locality. Our limitations on syntax and analysis program choice limit end-user flexibility, but are safer to deploy to a public data service.

Our performance measurements have shown how our system generates workflows automatically from commonly-used shell scripts, how the resulting workflow can be executed on a cluster of nodes as might be available at a data center, and the enhanced performance we have achieved in a variety of configurations. Our lightweight scheduler illustrated the potential overhead pitfalls of status reporting and monitoring in a web/grid-service environment. Our experiments also showed the importance of optimizing scheduling algorithms to account for I/O costs for these I/O-intensive workflows, even in systems with Gigabit Ethernet connectivity. As a result, we have illustrated the viability of compiling, parallelizing, optimizing, and executing workflows defined by ordinary shell scripts as domain-specific languages.

References

[1] The OpenSymphony project. http://www.opensymphony.com.

[2] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock. Kepler: an extensible system for design and execution of scientific workflows. In *Proceedings of the 16th Intl. Conference on Scientific and Statistical Database Management(SSDBM)*, pages 423–424, Santorini Island, Greece, June 2004.

[3] K. Amin, G. von Laszewski, M. Hategan, N. J. Zaluzec, S. Hampton, and A. Rossi. Gridant: A client-controllable grid workflow system. In *HICSS '04: Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 7*, page 70210.3, Washington, DC, USA, 2004. IEEE Computer Society.

[4] A. Anjomshoaa, F. Brisard, M. Drescher, D. Fellows, A. Ly, S. McGough, D. Pulsipher, and A. Savva. Job Submission Description Language (JSDL) Specification. *Version*, 1:15, 2005.

[5] R. Badia, J. Labarta, R. Sirvent, J. Pérez, J. Cela, and R. Grima. Programming Grid Applications with GRID Superscalar. *Journal of Grid Computing*, 1(2):151–170, 2003.

[6] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer. Simple Object Access Protocol (SOAP) 1.1. Technical report, W3C, May 2000. http://www.w3.org/TR/2000/NOTE-SOAP-20000508/.

[7] J. Darlington, M. Ghanen, and H. W. To. Structured Parallel Programming. In *Working Conference on Massively Parallel Programming Models: Suitability, Realization, and Performance*, Berlin, Germany, September 20–23, 1993.

[8] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artif. Intell.*, 49(1-3):61–95, 1991.

[9] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–238, 2005.

[10] T. A. Eyre, F. Ducluzeau, and T. P. Sned. The hugo gene nomenclature database, 2006 updates. *Nucleic Acids Research*, 34:D319–D321, 2006.

[11] J.-L. Gaudiot, T. DeBoni, J. Feo, W. Bo̊hm, W. Najjar, and P. Miller. The sisal model of functional programming and its implementation. In *PAS '97: Proceedings of the 2nd AIZU International Symposium on Parallel Algorithms / Architecture Synthesis*, page 112, Washington, DC, USA, 1997. IEEE Computer Society.

[12] A. Grimshaw, S. Newhouse, D. Pulsipher, and M. Morgan. OGSA Basic Execution Service Version 1.0. *Open Grid Forum*, 2006.

[13] D. C. Hanselman and B. L. Littlefield. *Mastering Matlab 7*. Prentice Hall, October 2004.

[14] Climate Change 2007: The IPCC 4th Assessment Report. http://www.ipcc.ch/, 2007.

[15] S. Krishnan, P. Wagstrom, and G. von Laszewski. GSFL: A Workflow Framework for Grid Services. *Preprint ANL/MCS-P980-0802, Argonne National Laboratory, August*, 2002.

[16] F. Leymann. WSFL–Web Services Flow Language. *IBM Software Group, Whitepaper*, 2001.

[17] M. Mathis, J. Heffner, and R. Reddy. Web100: extended tcp instrumentation for research, education and diagnosis. *SIGCOMM Comput. Commun. Rev.*, 33(3):69–79, 2003.

[18] T. Oinn, M. Greenwood, M. Addis, M. N. Alpdemir, J. Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, D. Marvin, P. L. P. Lord, M. R. Pocock, M. Senger, R. Stevens, A. Wipat, and C. Wroe. Taverna: lessons in creating a workflow environment for the life sciences. *Concurrency and Computation: Practice and Experience*, 18(10):1067–1100, 2006.

[19] S.-B. Scholz. Single assignment c – efficient support for high-level array operations in a functional setting. *Journal of Functional Programming*, 13(6):1005–1059, 2003.

[20] P. Senkul, M. Kifer, and I. H. Toroslu. A logical framework for scheduling workflows under resource allocation constraints. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 694–705, Hong Kong, China, 2002.

[21] I. Taylor, I. Wang, M. Shields, and S. Majithia. Distributed computing with triana on the grid. *Concurrency and Computation: Practice and Experience*, 17(9):1197–1214, 2005.

[22] D. L. Wang, C. S. Zender, and S. F. Jenks. Server-side parallel data reduction and analysis. In *Advances in Grid and Pervasive Computing: 2nd International Conference, GPC 2007, Paris, France, May 2-4, 2007, Proceedings*, volume 4459 of *Lecture Notes in Computer Science*, pages 744–750, Heidelberg, Germany, May 2007. Springer-Verlag.

[23] C. S. Zender. netCDF Operators (NCO) for analysis of self-describing gridded geoscience data. *Submitted to Environ. Modell. Softw.*, 2006.