2009 Fifth IEEE International Conference on e-Science

A Fresh Perspective on Developing and Executing DAG-Based Distributed Applications: A Case-Study of SAGA-based Montage

Andre Merzky[1], Katerina Stamou[1,2], Shantenu Jha[1,2,3], Daniel S. Katz[4,1,5]

[1]*Center for Computation & Technology, Louisiana State University, USA*

[2]*Department of Computer Science, Louisiana State University, USA*

[3]*e-Science Institute, Edinburgh, UK*

[4]*Computational Institute, University of Chicago, USA*

[5]*Department of Electrical & Computer Engineering, Louisiana State University, USA*

Abstract

Most workflow based applications currently have to adapt to available tools. While this keeps the cost of development low, it can lead to performance and flexibility tradeoffs that the application developer and deployer must make. In this paper, we use the Montage astronomical image mosaicking application as prototypical DAG-based workflow application to layout the development and deployment decisions for distributed applications. We discuss and explain the lack of simple (easy-to-use), scalable, and extensible distributed applications. We then introduce SAGA as a technology that permits the construction of abstractions that aid the development and execution of the applications, and thus addresses some of common shortcomings of traditional distributed applications development. We use Montage together with SAGA to examine how legacy applications can be made to run on distributed infrastructures, to see if our reasons are valid, and to compare potential new methods for creating distributed applications with existing technologies that are currently used. We demonstrate the ability to (i) scale-out and (ii) use different production infrastructure, while maintaining performance comparable to established systems. Our hope is that by demonstrating the simplicity of development along with other advantages (performance, scalability, extensibility, and infrastructure independence), this example will encourage others to think more broadly about how distributed applications are created and how new programming models such as Dryad can be supported in an infrastructure independent way, thus eventually leading to more applications that can seamlessly scale-out.

I. Introduction

Most applications that utilize distributed systems such as Grids and Clouds were not developed explicitly for distributed cyberinfrastructure (CI). Many applications are developed with a local cluster or parallel system in mind, not distributed CI. However, once an application is developed for local and parallel systems with static resources and static application execution, re-tooling it to effectively deploy and execute it on distributed CI is not easy. It also remains difficult for the application to easily take advantage of the full set of capabilities that distributed CI offer. Developing applications without distributed systems and infrastructure in mind is simple, but this often comes at the price of performance, extensibility, flexibility, and the ability to exploit the strengths of distributed systems. For example, as a consequence of this legacy, most distributed applications are developed with the model of a set of resources that are fixed at start-up time, with limited capacity to expand the resources used during execution, or to use a different set of resources that may be more effective/efficient than the set on which execution started. Developing applications that can use a dynamic distributed resource pool is an important and powerful feature. There are inherent advantages of rethinking the basics of developing distributed applications that extend beyond the realm of distributed systems to high-end platforms.

An aim of this work is to highlight some tradeoffs that exist when developing applications to exploit distributed CI by design – so called (*a priori* or *first-principles*, rather than as an afterthought. While this work is some what descriptive and theoretical, it is not limited to theory (except in the application development phase). Rather, suggested approaches are backed up by specific implementations and performance measures. In this paper, we investigate a very well known application, Montage, and consider three different phases in which it can be re-developed to utilize distributed systems: application development, deployment, and execution. Underpinning all the implementations is the SAGA [1], [2] programming system, which provides the handles to implement the basic distributed functionality, the ability to develop the higher-level abstractions to support the development of distributed applications, and the mechanisms to implement them on a range of distributed systems.

II. Background

We begin by quickly comparing distributed applications and their better understood parallel counterparts. Although the comparison and the description of programming systems (for distributed applications) is purely pedagogical, it highlights several of the challenges associated with developing distributed applications that we will address later in the paper. In the latter half of this section, we will provide a brief description of Montage, which serves as a prototypical DAG-based application, and SAGA and extensions thereof, as the integrated API and programming system used to address some of the observed deficiencies in distributed programming, both in general and specifically with respect to DAG-based applications.

A. Distributed Applications

Distributed versus Parallel Applications: The complexity of developing applications for large-scale problems stems in part from the challenges inherent in both high-performance computing and large-scale distributed systems. In traditional high-performance computing, performance is paramount, systems are typically homogeneous and assumed to be failure free (at least historically), and applications operate within a single administrative domain. For distributed systems, performance, as measured by peak utilization, is often a secondary concern, not because it isn't important, but rather, because the concerns and objectives for distributed applications are broader, such as scalability, heterogeneity, fault-tolerance and security. In addition to these non-functional features of distributed systems, application execution must be managed across administrative domains with variable deployment support.

The usage requirements that drive distributed applications differ from typical parallel applications, as do the nature and characteristics of distributed applications. More often than not, parallel applications are monolithic entities where the challenge is to determine a scalable decomposition that maintains stringent requirements on communications. In contrast, distributed applications are mostly either naturally decomposable or are built from existing, loosely-coupled components (which may themselves be parallel), where the challenge is to effectively coordinate these components. In other words, for parallel applications skillful decomposition is required; for distributed applications skillful composition is required. Most parallel/cluster applications are developed with a model that assumes a specific execution environment with well-defined resource requirements/utilization and control over these resources. Thus, most parallel applications are not developed with the flexibility to use additional resources during execution, even though some may have varying resource requirements during execution (e.g., arising from some "interesting" computational result.)

Given the fact that the computational and scheduling requirements of the individual components can change, and that their execution environment is less controlled and more variable than traditional parallel environments, it is fair to assume that distributed applications should, more often than not, be developed to support dynamical execution. As we will discuss, there are many distributed applications, e.g., Dynamically Data Driven Applications, that would benefit from support for dynamic execution models. This is a complex issue, for distributed applications are developed using the programming systems and deployment tools that are available, but these systems and tools can lag the development requirements of distributed applications.

Viewed alternatively, most distributed applications are legacy applications that have been re-architected, re-deployed or re-tooled for execution in a distributed mode; they have retained the legacy static execution models, which is also a consequence of the fact that bulk of programming systems used to (re-)develop distributed applications have been somewhat slow to facilitate the uptake of dynamical applications. This provides another motivation for this work: to investigate how an application that has historically had a static resource model, can be modified to have dynamic (at runtime) and adaptive capabilities. We present trade-offs associated with each approach.

B. Programing Systems

A number of workflow systems currently support constructing scientific applications by combining components and services. Although they share a number of similarities, they differ in how they describe the workflow, or support the subsequent enactment process. This is one reason why many existing programming systems are not widely used by application developers, who often prefer to "roll their own" capabilities, even though many already exist in available programming systems. While using existing programming systems would let the developer focus on application specifics, rather than the implementation of such core infrastructure, only a few programming systems and tools are widely and extensively used. So why don't application developers use capabilities from an existing programming system to support required capabilities? This may be due to a number of factors [3]: (a) many programming systems lack robustness when used by developers in scientific applications – for instance, they may have scalability limitations; (b) the programming system's maturity and development schedule may not align with application developers' requirements; and (c) it may be difficult to isolate a specific capability from a programming system, thereby limiting use by an application developer requiring that capability. We will revisit these issues in the specific case of DAG-based applications.

C. SAGA

SAGA [4] provides a simple, POSIX-inspired API to the most commonly required distributed functionality at a sufficiently high-level of abstraction so as to be independent of the diverse and dynamic Grid environments.

In the absence of a formal theoretical taxonomy of distributed applications, Fig. 1 can act a guide. Using this classification system, there are three types of distributed applications: (i) Applications where local functionality is swapped for distributed functionality, or where distributed execution modes are provided. A simple but illustrative example is an ensemble of an application that uses distributed resources for bulk submission. Here, the application remains unchanged and even unaware of its distributed execution, and the staging, coordination, and management are done by external tools or agents. Most application in this category are classified as implicitly distributed. (ii) Applications that are naturally decomposable or have multiple components are then aggregated or coordinated by some unifying or explicit mechanism. DAG-based workflows are probably the most common example of applications in this category. Finally, (iii) applications that are developed using frameworks, where a framework is a generic name for a development tool that supports specific application characteristics (e.g., hierarchical job submission), and recurring patterns (e.g., MapReduce, data parallelism) and system functionality. SAGA has been used to develop system-level tools and applications of each of these types.

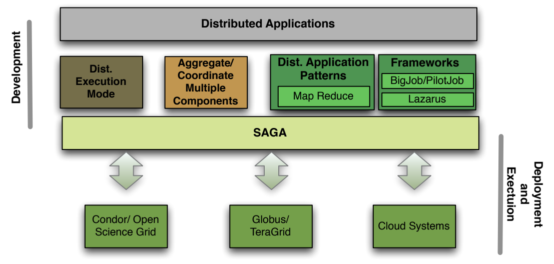

Fig. 1: Showing the ways in which SAGA can be used to develop distributed applications. The different shaded box represent the three different types; frameworks in turn can capture either common patterns or common application requirements/characteristics.

It is important to note that SAGA provides the basic API to implement distributed functionality required by applications (typically used directly by the first category of applications), and is also used to implement higher-level APIs, abstractions, and frameworks that, in turn, support the development and deployment of distributed applications [5], [6]. In this paper, we discuss a specific case of using SAGA to implement a higher-level API to support workflows.

D. Montage

Montage [7], [8] is an astronomical image mosaicking application that is also one of the most commonly studied workflow applications. It is designed to take multiple astronomical images (from telescopes or other instruments) and stitch them together into a mosaic that appears to be from a single instrument.

The initial version of Montage focused on making Montage scientifically accurate and useful to astronomers, without being concerned about computational efficiency, and is being used by many production science instruments and astronomy projects [9]. Montage was envisioned to be customizable, so that different astronomers could choose to use all, much, or some of the functionality, and so that they could add their own code if so desired. For this reason, Montage is a set of modules or tools, each an executable program, that can run on a single computer, a parallel system, or a distributed system. Montage initially used a script to run a series of these modules on a single processor, with some modules being executed multiple times on different data. An example of the Montage workflow is shown in Fig. 2, showing that a Montage run is a set of tasks, each having input and output data, and many of the tasks are the same executable run on different data, referred to as a stage.

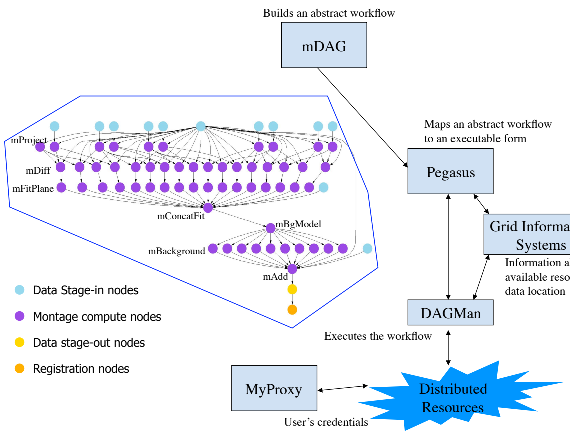

Fig. 2: Original Montage workflow processing

The second Montage release delivered two new execution modes in addition to sequentially execution[1], suitable for Grid and also Cloud environments [10]. First, each stage can be wrapped by an MPI executable that calls the tasks in that stage in a round-robin manner across the available processors. Second, the Montage workflow can be described as a directed acyclic graph (DAG), and this DAG can be executed on a grid, as shown in Fig. 2. In Montage, this is done by mDAG, a Montage module that produces an abstract DAG (or A-DAG, where abstract means that no specific resources are assigned to execute the DAG), Pegasus [11], [12], which communicates with grid information systems and maps the abstract DAG to a concrete resource assignment, creating a concrete DAG (or C-DAG), and DAGMan [13], which executes C-DAGs on their internally-specified resources.

There is an issue here, somewhat specific to Montage, in that the work that carried out by Montage is partially based on runtime information: some steps in the initial workflow may not need to be performed. The DAG that is fed into Pegasus is a worst-case scenario. If the DAG didn't need to be fixed in this early stage, the total amount of work to be carried out could sometimes be reduced.

The generality of Montage as a workflow application has led it to become an exemplar for those in the computer science workflow community, such as those working on: Pegasus, ASKALON [14], QoS-enabled GridFTP [15], SWIFT [16], SCALEA-G [17], VGrADS [18], etc. Most applications do not have the advantage of being studied by multiple CS groups. Most application developers can only make use of existing tools, which are typically not designed for extension, flexibility, etc. This leads to the application developers having to modify their application to suit the tools. If the application is being designed from scratch, this may not be

[1]All tasks are sequentially ordered, from start to end, and from left to right in each stage.

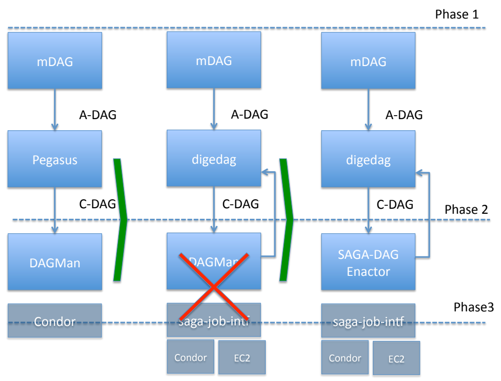

Fig. 3: A comparison of the three different ways in which Montage pipeline can be implemented. The green arrows (chevrons) indicate increasing generality of Montage on various infrastructures

an issue, but it can limit the reuse of existing applications in distributed systems, particularly if the reworking needed in the application is large.

Using three separate tools to run Montage has both advantages and disadvantages. Clearly, the separation of concerns means that each tool can focus on what it needs to do, and the developer of the tool does not need to be an expert on the other parts of the application process. However, in this paper, we want to examine how integrating the entire process can lead to improved performance, more simplicity for users, and more flexibility for developers.

III. Analyzing and Re-architecting DAG-Based Applications

Using Montage as a specific example, we analyze the typical three phases of DAG-based distributed applications and propose alternative approaches. Fig. 3 shows on what level these alternative approaches are applied. The leftmost column represents *traditional* Montage. The middle column shows changing the DAG Generation and Planning phase from the traditional approach to using a higher-level API to represent DAGs and operate on both the A-DAG and the C-DAG. The third column extends the approach from column 2 to use the SAGA-API to execute the DAG on multiple different backend infrastructures. Phase 1 remains the same in all three approaches and is thus not addressed here. The X in Column 2 illustrates the fact that although we wanted to re-architect DAGMan to be independent of Condor and use a general job-submission interface, we were unable to do so; details are explained below.

A. Application Phases

Given an application built of components that are run by a script on a single processor, as Montage started out, in order to use distributed resources, one must first (*Phase I*) express the work to be done as a modularized and interconnected workflow. Currently a DAG is generated; this should be done by a tool. The Montage team initially had difficulty writing out the abstract DAG file, not because the dependencies were unknown or unclear, but because the actual mechanics of what needed to be in this file and how it would actually be generated were not trivial. It was known that each input file needed to be run through mProject to reproject it, and then each pair of overlapping images needed to be run through mDiff, etc., but building the A-DAG that contains these dependencies for all of the input files currently takes about 4000 lines of C code. Ideally, an application developer would be able to express the dependencies in a general manner, and have an automated (runtime) system build the abstract DAG from these rules, without exposing this to the application developer, who would not need to write a code to build the DAG (mDAG in the case of Montage). Note that this DAG is *abstract* (A-DAG), in the sense that it describes an abstract workflow that is not yet mapped onto concrete resources.

The second step (*Phase II*) is the DAG execution planning, or generating the *concrete* DAG (C-DAG), which includes information about specific resources to be used to execute the DAG. Mapping the DAG nodes to specific resources is non-trivial; it may involve resource discovery, resource reservation, authentication and authorization for resource utilization, and potentially multivariant optimization, for cost and runtime. For Montage, that task is traditionally performed by Pegasus, which maps the DAG to Condor based resources.

The final step is the actual application execution (*Phase III*), where a particular problem is solved. Once the C-DAG is available, the grid version of Montage uses DAGMan to spawn tasks, and GridFTP (or similar) to move data files if needed.

We have re-architected these phases of Montage in several ways, using digedag – a prototype implementation of a SAGA-based workflow package, which has four basic components:

1) an API for programatically expressing workflows,
2) a parser for (abstract or concrete) workflow descriptions, (not discussed here in any detail),
3) an (in-time workflow) planner, and
4) a workflow enactor (using the SAGA engine)[2]

Phase 1: Application Development

In Montage, as in other workflow applications, a set of modules need to be run, and some dependency rules control when each modules can run. For example, mDiff can be run for an overlap of two images once both of the images have been reprojected by running mProject. This is an example of a dependency rule within Montage. Currently, the Montage mDAG module implicitly contains these rules. They were

[2]Although the enactor is currently part of digedag, it is intended to be separated out. The future enactor will, however, continue to use SAGA to interact with backend resources.

known by the mDAG developer, who used them to write this mDAG. An alternative would be for the developer to merely express the rules in a standardized format, e.g. similar to the specification of dependencies in Unix Makefiles. This would make the job of developing codes such as Montage easier.

Digedag API: Traditionally, workflows are rarely expressed programatically, at least not explicitly, so only a small number of APIs exist to do so. Digedag has such an API, to create and manage nodes and edges. A node (representing a job to be executed) can have multiple incoming and outgoing edges (representing data items written by one job and read by another.) An edge has exactly one source node, and one target node. The API also permits the management of the execution of the DAG, and of parts thereof. Internally, digedag uses the API to express and manage the C-DAG. This API offers an immediate gain in Montage application development, though not in execution, which is not a serious limitation, as the A-DAG creation time is similar to that of current mDAG Montage module – which in turn is a very small fraction of the Montage run time.

Phase 2: C-DAG Generation and Execution Planning

Montage traditionally uses Pegasus to translate the A-DAG into a C-DAG. Pegasus is tightly coupled to DAGMan and Condor; the concrete workflow descriptions created by Pegasus are not easily reusable by other tools. Digedag, on the other hand, provides similar functionality to Pegasus[3] but does not bind to specific underlying middleware/platforms, as it uses SAGA as an interface to different infrastructures.

Another feature of the digedag planner is that it is activated not only when creating the C-DAG, but also when *executing* the C-DAG. It is activated when a DAG is created, when a DAG has nodes and edges added/removed, when a DAG fires/completes successfully/unsuccessfully, when a node/edge fires/succeds/fails or edge's data transfer failed/-succeeds. Thus, the planner responds to runtime errors immediately, can observe all actions on a DAG and on its elements, and can globally optimize execution.

Phase 3: Execution

As shown in Fig. 3, the process of integrating DAGMan with SAGA can be separated into two logical parts: (i) using DAGMan as a DAG metascheduler, while decoupling its job submission and monitoring functionality from any specific grid resource management system (i.e., Condor in this case); (ii) replacing just the Job Submission mechanism of DAGMan.

One way for DAGMan to work transparently with any kind of grid middleware backend system would be to interface it to SAGA so that it could leverage a multitude of supported adaptors. The core functionality of DAGMan as a metascheduler requires a mechanism via which the system can query for the status of any submitted job, in order to schedule the execution of the subsequent tasks, which might be dependent on the successful completion and output of previously submitted jobs. DAGMan does this by specifying a particular job execution status log file (the 'log' directive in Condor submission files). The Condor Shadow daemon writes updates of the status of a job while in pool to the specified file. The messages follow a conventional Condor event format [19], managed by the (not formally defined and documented) "Condor User Log API". Various event types indicate the status of the job in pool. The most important are those that indicate a job is: (i) submitted in queue; (ii) running/executing; (iii) terminated/completed; (iv) under an error condition/job abort. There is no straightforward way to discern the event-type based job status monitoring system from DAGMan, which appears to be very tightly coupled with the various functions of its codebase.

We found DAGMan too intricately programmed and too interwoven with Condor's internal mechanisms to easily allow use of SAGA for backend interaction to enable the desired runtime properties. DAGMan is a native part of the Condor project; its design and implementation is very closely tied to the Condor ecosystem of cooperating components; and it lacks a generic programming API and well defined system-interfaces that would enable independent projects to take advantage of its metascheduler functionality to integrate it with external applications. This is not a criticism of DAGMAN, but a general statement of the way tools and programming systems are designed (as alluded to in §II-B.) Because it was difficult to disentangle the monitoring requirements of DAGMan from Condor, it didn't allow DAGMan to use SAGA's job-submission capability.

This gap motivated the development of a general purpose SAGA-based DAG enactor that executes and orchestrates DAGs on multiple distributed backends; the individual tasks are submitted to the assigned concrete physical resources, and their execution and communication is orchestrated according to the specified workflow logic. The pipeline is shown in the rightmost column of Fig. 3. This SAGA-DAG enactor is analogous to DAGMan in consuming concrete workflows, i.e. it consumes files that contain a description of the workflow elements (nodes and edges) and descriptions of the individual resources these workflow elements are mapped to and executed upon. For compatibility with usual Montage usage, we also implemented a parser for both A-DAGs and C-DAGs such as those created by Montage's mDag, both of which can be consumed by digedag and in turn use individual elements via SAGA middleware calls.

In order to implement autonomic and adaptive features as well as advanced runtime properties (fail safety, data co-scheduling, dynamic scheduling) we designed the SAGA-DAG enactor to *change* the C-DAG on-the-fly. (As mentioned above, the digedag planner can adjust the C-DAG on certain DAG execution events.) The need for this might

[3]Note that digedag is an experimental prototype, and doesn't provide all the advanced scheduling and mapping capabilities of Pegasus. Digedag implements the same functionality – the translation of abstract DAGs into concrete DAGs, but conceptually differs by operating on both the A-DAG and the C-DAG by design.

arise in remapping workflow elements to other, more suitable resources[4] The ability of the SAGA-DAG enactor to *dynamically* change the mapping of an A-DAG into a C-DAG representation, while having (a) the full A-DAG, (b) the current C-DAG, and (c) live system information available, allows for much more flexibility than the more static approach in the leftmost column in Fig. 3. In general, the advantages of this approach are (i) Extensible to multiple, heterogeneous distributed infrastructures (ii) Enabling additional runtime properties such as fault-tolerance, dynamic scheduling, and data-compute co-scheduling.

SAGA-Condor Adaptor: SAGA, via its job package API, provides job submission and management capabilities, interfacing with resources managed by the Condor system. The grid application programmer uses the well-defined SAGA API to construct the required functionality, while the middleware specifics are transparently handled by the appropriate matching adaptor, requiring virtually no configuration setup. The SAGA-Condor adaptor is implemented as a thin, lightweight layer that wraps the Condor distribution application utilities. The actual job submission is achieved through the condor_submit utility and the execution status event logfile, closely resembling how DAGMan operates. Furthermore, SAGA-specific attributes are dynamically mapped into their Condor-specific equivalents.

Additionally, the digedag enactor is, of course, able to use other SAGA adaptors to access backends based on Globus, GridSAM, ssh, etc. This gives the enactors runtime capabilities, compared to DAGman, which is more or less married to Condor.

IV. Experiments

A. Experimental Setup

The goal of our experiments is *not* to demonstrate performance gains for a specific application running in a specific distributed environment, but to demonstrate that the development of distributed applications with dynamic distributed execution services does not introduce additional development or runtime overhead as compared to developing the same application for a traditional parallel or a static distributed execution service.

Our experiments are, as discussed, based on Montage. We have chosen the Montage *m101* tutorial's workflow, which is large enough to demonstrate our objectives, but is also small enough to allow us to compare to runs on local resources. In fact, the small size of the data sets (640 MB) and short execution time (about 160 s sequential on a 2.6-GHz Intel Core-2 CPU) makes the distribution of the DAG execution dominantly communication-dominated.

The DAG representing the m101 workflow consists of 127 nodes (representing the 125 jobs to be executed, plus one input and one output node) in y stages, with implicit synchronization between some stages. These nodes are connected by 380 edges, representing 383 data exchanges via files. The respective files are rather small (the largest being about 35 MB, the bulk below 5 MB). As file transfer times are thus more or less negligible and not straight forward to compare between the different backends, which use a mixture of shared file systems and file exchange services with varying support for 3rd party file transfer, we present all numbers *without* file transfer times. The times do include, however, file management latencies. Within each stage, nodes are executed in parallel, but file management latencies may still add up due to middleware serialization.

[4]Even if Pegasus does a perfect job when mapping the nodes to specific resources, runtime events such as resource failures or dynamic scheduling fluctuations cannot be taken into account.

resource	CPUs	latency
apple	1 * Core-2 Duo, 2.6 GHz	0.1 ms
qb	8 * Xeon, 2.33 GHz	221.9 ms
EC2	1 * Xeon, 2,66 GHz virt.	153.6 ms
purdue	x.000 * Xeon	38.6 ms[5]

TABLE I: Properties of the resources used for the experiments.

The resources used for the experiments are a local Apple computer with a Core-2 Duo CPU, a headnode of LONI's Queen Bee cluster with 8 Xeon CPU's, and 'small' Amazon EC2 instances with virtualized hardware (single Xeon). The remote DAGMan experiment submitted to Purdue's condor pool, consisting of several thousands Xeon nodes. Otherwise, the local Apple machine was used for all submission. The local DAGMan experiment was run on a different hardware, but the measurements have been normalized to be comparable to the Apple used for the digedag runs. Latencies between the machines and their submission backends are given in table I.

Athough SAGA can submit to a larger set of middlewares, for simplicity we limited our experiments to fork, ssh and Condor.

B. Results

We intend to show that no *additional* runtime overhead is introduced, apart from communication and known software-layer overhead. To support this claim, we show that the overall execution time for a distributed run is the sum of the execution time of a local run plus communication time plus overhead introduced by the respective remote service (ssh, Condor, EC2). Compared to that overhead, the enactment and workflow control provided by digedag is not introducing any significant additional overhead.

1) Planning Phase: The performance numbers from table II include the planning times, which for digedag are an integral part of the DAG execution, as described earlier. Additional experiments show that for the simple (i.e. prototypical) scheduling mechanisms implemented in digedag right now, that overhead is close to zero. Even the rather intricate and intelligent Pegasus planning phase requires only about 2.0 seconds of runtime, which is negligible compared to the overall job runtime. (Planning time for the local resource was 2.0 s, with standard deviation of 0.1 s. For the Purdue

[5]Latency for the remote DAGMan run.

#	resources	middleware	walltime	std-dev.	diff to local
1	l	f	68.7 s	9.4 s	– –
2	l	s	131.3 s	8.7 s	62.6 s
3	l	c	155.0 s	16.6 s	86.3 s
4	l	f, s	89.8 s	5.7 s	21.1 s
5	l	f, c	117.7 s	17.7 s	49.0 s
6	l	s, c	133.5 s	32.5 s	64.8 s
7	l	f, s, c	144.8 s	18.3 s	76.1 s
8	q	s	491.6 s	50.6 s	422.9 s
9	e	a	354.2 s	23.3 s	285.5 s
10	e, q	s, a	363.6 s	60.9 s	294.0 s
11	l, q, e	f, s, a	409.6 s	60.9 s	340.9 s
12	l	d	168.8 s[6]	5.3 s	100.1 s
11	p	d	309.7 s	41.5 s	241.0 s

TABLE II: Execution measurements
resources: ***l****=local,* ***p****=Purdue,* ***q****=Queen Bee,* ***e****=aws/EC2*
middleware: ***f****=fork/SAGA,* ***s****=ssh/SAGA,* ***a****=aws/SAGA,* ***c****=Condor/SAGA,* ***d****=Condor/DAGMan,* ***p****=Pegasus*

Phase	Scale-Out	Sched.	Adaptivity	Simp.	Ext.
Phase 1	-	-	-	Y	Y
Phase 2	-	Y	Y	-	Y
Phase 3	Y	Y	-	Y	Y

*TABLE III: Advantages of re-architecting the traditional DAG-based pipeline, and the different levels at which these advantages manifest themselves. (**Scale-Out**: ability to use multiple distributed resources concurrently; **Sched.**: greater scheduling flexibility; **Adaptivity:** DAG Adaptivity and response to changes; **Simp.**: greater simplicity without sacrificing functionality and performance; **Ext.**: extensibility and general-purpose uptake)*

resource, we measured 1.8 s/0.3 s.) That observation supports our claim that the integration of dynamic scheduling decisions during the execution phase of the DAG will not add any significant overhead to the overall time-to-completion.

2) SAGA Overhead: Table II gives the mean and standard deviation over 10 consecutive runs for each experiment. Table II shows run times. The aws/EC2 backend times (#9, #10, #11) are cleared of the EC2 startup times – those are in detail discussed in [20]. If multiple backends are specified, the individual DAG nodes are mapped to the respective backends in round-robin fashion. Note that the table also gives the times for the traditional DAGMan execution to a local and a remote Condor pool (#10, #12).

The performance impact incurred by introducing the SAGA abstraction layer on top of the Condor submission system is relatively negligible, and in fact comparable to DAGMan, which is part of the Condor project (#3 vs. #12). Additional experiments reveal that the *condor_submit* binary startup and execution time is on the order of 0.040 to 0.050 s, while a SAGA linked submission utility takes approximately 0.45 s to execute. These timings are independent of the actual time that it takes for the submitted task to execute, which will by definition be equal for both approaches. It appears that the difference of the binary start-up times can be attributed to the fact that the SAGA-based layer is linked against a much larger number of shared libraries that are required for runtime execution than condor_submit is. For applications with short runtimes, such as 10 seconds, the overhead of SAGA is roughly 4-5%, and for applications that take more than 1 minute, the overhead of SAGA drops to 1%. A similar argumentation holds comparable numbers for SAGA's aws and ssh backends (#4).

3) Distribution Overhead: It is of course not surprising that the experiments for distributed backends imply a higher walltime than for the local runs: as our chosen problem is not really CPU nor data, but infrastructure bound, the additional infrastructure latencies add up directly.

[6]Timing normalized, see §IV-A.

For example, for local execution via fork/exec and via ssh (#1 vs. #2) differ by about 60 seconds. The estimated number of sequential interactions exchanged over ssh is about 200[7]. An execution of `'ssh localhost true'` shows an overhead of about 0.25 seconds. 200 exchanges would add up to about 50 seconds, which nicely explains the pure ssh overhead.

The remote ssh backend to Queen Bee adds another 360 seconds (#2 vs. #8). A single ssh interaction with QB is measured to take 3.3 seconds. The 200 interactions thus would add up to about 700 seconds – but in fact the file system interactions can utilize parallel access much better in that case[8], and due to that congruency increase we end up with about 100 sequential interactions, which add up to about 330 seconds – again an excellent match to the observed performance.

Similar observations can be made for the Condor and aws backends: condor_submit to a local pool adds about 5.0s, but local file access is not affected by additional latency; aws execution adds about 2.2 seconds, for jobs and files.).

The remote DAGMan experiment (#12) shows actually somewhat better performance than the remote digedag experiments (#8-#11). We expect that this is mostly caused by the lower latency of the infrastructure used for that experiment (see table I). Although we estimate latencies and neglect factors like socket-reuse, memory and process overhead, etc., this still serves to help understand to a first approximation the individual experiments.

Interesting is also the observation that the standard deviation for the Condor backend, and more so for the remote backends, is much larger than for the local fork and ssh backends: total execution times are thus somewhat less exactly to predict.

Mixing multiple backends and/or resources leads to the expected overall execution time which lies mostly in between the times for the individual backends (see for example #8 (ssh/qb) and #9 (aws/EC2) versus #10 (mixing both)). Again, the standard deviation for these times tends to increase with the number of backends used.

V. Conclusion and Future Work

Our goal was to highlight some of the development and execution issues and demonstrate that the development of DAG based applications can be both simple and efficient if the right tools are available and provide the right levels of abstractions. We showed that developing DAG-based applications can be greatly simplified by using an integrated API that does not require the manual building of DAGs, but that allows node and data dependencies to be specified and leaves the DAG generation to a tool. Our experiments also showed that the runtime overhead is negligible, compared to the latency overhead which all distributed application encounter (which is not addressed using abstract tooling).

Digedag represents a first example of a higher-level API supporting specific functionality to have been developed based on the SAGA API. The ability to support (abstract and concrete) DAGs adaptively and the ability to dynamically reschedule them provides the basis for implementing advanced programming models such as Dryad and other streaming-based approaches.

Our implementations of the digedag API and DAG enactor are currently experimental, and thus are not meant to be comparable in performance to other generic workflow tools. However, these implementations demonstrate design simplicity, flexibility, and the advantages of flexible backend support, letting us easily scale-out to a variety of middleware. Also, our ability to change the C-DAG during the DAG execution time comes with small to no overhead, at least for the DAG we used, and seems to open promising new pathways for DAG execution optimizations, such as fail safety, data-compute colocation, or node migration, which will come free to the application developers, as they will still code against a single, middleware-agnostic API.

We note that in our approach although there is feedback between execution phase and execution planing, our solution is just as modular as as the traditional mDAG/Pegasus/DAGMan path (or even more so, considering SAGA's modularity towards middleware bindings). We are really just dividing the same problem space in a different way in order to explore different aspects of the solution space.

Finally, the reader should be aware that an incredible large amount of research and implementation work went into Pegasus and DAGMan. Both very likely outperform our implementation for any but the most simple DAGs and backend configurations. We think, however, that our approach will allow a maturing of digedag, and this is what we hope to achieve in the near future, to demonstrate competitive performance for real-life DAGs (i.e. large DAGs with non-trivial resource requirements.)

[7]10 stages * two interactions per job * 3 round trips per interaction + 10 * stages * 3 file exchanges per job * 4 roundtrips = 10*2*3 + 10*3*4 = 180. This assumes perfect job parallelism in each and every stage.

[8]Digedag tries to run file accesses in parallel, but has some delay between them; short roundtrips negate the parallelism gain.

Acknowledgements

SJ acknowledges the e-Science Institute, Edinburgh for supporting the research theme, "Distributed Programming Abstractions," which provided the motivation for this work. This work would not have been possible without the efforts and support of the wider SAGA team. Important funding for SAGA specification and development has been provided by the UK EPSRC grant number GR/D0766171/1 (via OMII). This work has also been made possible thanks to the internal resources of the Center for Computation & Technology at Louisiana State University and computer resources provided by LONI. We thank Joao Abecasis for his work on the SAGA Condor adaptors, and Lukasz Lacinski and Yaakoub El Khamra for help with the Condor experiments.

References

[1] A. Merzky and S. Jha, "A Requirements Analysis for a Simple API for Grid Applications," Grid Forum Document GFD.71, 2006, Global Grid Forum.

[2] ——, "A Collection of Use Cases for a Simple API for Grid Applications," Grid Forum Document GFD.70, 2006, Global Grid Forum.

[3] S. Jha et al., *Programming Abstractions for Large-scale Distributed Applications*, submitted to ACM Computing Surveys; draft at http://www.cct.lsu.edu/~sjha/publications/dpa_surveypaper.pdf.

[4] [Online]. Available: {http://saga.cct.lsu.edu}

[5] Y. E. Khamra and S. Jha, "Title: Developing Autonomic Distributed Scientific Applications: A Case Study From History Matching Using Ensemble Kalman-Filters," in *Workshop on Grids Meet Autonomic Computing (GMAC), Sixth International Conference on Autonomic Computing, 2009. ICAC '09 (Barcelona).* IEEE, 2009.

[6] C. Miceli, M. Miceli, S. Jha, H. Kaiser, and A. Merzky, "Programming abstractions for data intensive computing on clouds and grids," in *Cluster Computing and the Grid, 2009. CCGRID '09. 9th IEEE/ACM International Symposium on*, May 2009, pp. 478–483.

[7] The Montage project, http://montage.ipac.caltech.edu/.

[8] G. B. Berriman, J. C. Good, D. Curkendall, J. Jacob, D. S. Katz, T. A. Prince, and R. Williams, "Montage: An on-demand image mosaic service for the NVO," in *Astronomical Data Analysis Software and Systems (ADASS) XII*, 2002.

[9] G. B. Berriman, J. C. Good, A. C. Laity, J. C. Jacob, D. S. Katz, E. Deelman, G. Singh, M.-H. Su, R. Williams, and T. Prince, "Science applications of Montage: An astronomical image mosaic engine," presented at IAU XXXVI General Assembly, 2006.

[10] E. Deelman, G. Singh, M. Livny, B. Berriman, and J. Good, "The cost of doing science on the cloud: the Montage example," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–12.

[11] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, and K. Vahi, "Mapping abstract complex workflows onto grid environments," *Journal of Grid Computing*, vol. 1, no. 1, 2003.

[12] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M.-H. Su, K. Vahi, and M. Livny, "Pegasus: mapping scientific workflows onto the grid," in *Proc. Across Grids Conf.*, 2004.

[13] DAGman, "Directed Acyclic Graph Manager," in *http://www.cs.wisc.edu/condor/manual/v7.0/condor-V7_0_1-Manual.pdf*, 2008.

[14] M. Wieczorek, R. Prodan, and T. Fahringer, "Scheduling of scientific workflows in the askalon grid environment," *ACM SIGMOD Record*, vol. 34, no. 3, pp. 52–62, 2005.

[15] M. Humphrey and S.-M. Park, "Data throttling for data-intensive workflows," in *Proc. IPDPS*, 2008.

[16] Yong Zhao et al., "Swift: Fast, reliable, loosely coupled parallel computation," *IEEE Congress on Services*, pp. 199–206, 2007.

[17] H.-L. Truong, T. Fahringer, , and S. Dustda, "Dynamic instrumentation, performance monitoring and analysis of grid scientific workflows," *J. Grid Comp.*, vol. 2005, no. 3, pp. 1–18, 2005.

[18] "VGRaDS: Montage, a project providing a portable, compute-intensive service delivering custom mosaics on demand." [Online]. Available: {http://vgrads.rice.edu/research/applications/montage}

[19] Condor Stable Release Version 7.2 Manual, Section 2 - Users' Manual, Subsection 2.6.6.

[20] S. Jha, H. Kaiser, A. Merzky, and O. Weidner, "Grid interoperability at the application level using saga," in *E-SCIENCE '07: Proceedings of the Third IEEE International Conference on e-Science and Grid Computing (e-Science 2007).* Washington, DC, USA: IEEE Computer Society, 2007, pp. 584–591.