Scientific Programming 13 (2005) 219–237
IOS Press

Pegasus: A framework for mapping complex scientific workflows onto distributed systems

Ewa Deelman[a,*], Gurmeet Singh[a], Mei-Hui Su[a], James Blythe[a], Yolanda Gil[a], Carl Kesselman[a], Gaurang Mehta[a], Karan Vahi[a], G. Bruce Berriman[b], John Good[b], Anastasia Laity[b], Joseph C. Jacob[c] and Daniel S. Katz[c]
[a]*University of Southern California Information Sciences Institute, CA, USA*
[b]*Infrared Processing and Analysis Center, California Institute of Technology, CA, USA*
[c]*Jet Propulsion Laboratory, California Institute of Technology, CA, USA*

Abstract. This paper describes the Pegasus framework that can be used to map complex scientific workflows onto distributed resources. Pegasus enables users to represent the workflows at an abstract level without needing to worry about the particulars of the target execution systems. The paper describes general issues in mapping applications and the functionality of Pegasus. We present the results of improving application performance through workflow restructuring which clusters multiple tasks in a workflow into single entities. A real-life astronomy application is used as the basis for the study.

1. Introduction

Many applications today are being built by large scientific collaborations such as those in physics [1,2], astronomy [3], biology [4], earthquake science [5], and many others. The applications often involve the processing of large data sets in many discrete steps (from calibration of the raw data, various data transformations, visualization, etc.) To support the scale of the applications, many resources are needed in order to provide adequate performance. These resources are often drawn from a heterogeneous pool of geographically distributed compute and data resources. The resources are often contributed by various institutions that are part of the collaborations. Running the large-scale, collaborative applications in such environments has many challenges. Among them are: systematic management of the applications, their components and the data, as well as successfully and efficiently running on the distributed resources.

In order to manage the process of application development and execution, it is often convenient to separate the two concerns. For example the application can be developed independently of the target execution system using a high-level representation. Then, once the target execution environment is identified, the application can be mapped onto it. One paradigm that has been explored in recent years is the notion of a workflow which can capture the behavior of the application. At the application-level the workflows are *abstract* in the sense that the workflow describes only the application components and their dependencies (reflecting the data dependencies in the application). The data and computations in an abstract workflow are also described at an abstract, or *logical* level in that their names refer to a logical entity that can then be mapped to one or more physical instance. The abstract workflow representation simplifies the application development process. It enables a systematic approach to application description; it provides flexibility in that individual application components can be replaced with alternative implementations; and it forms the basis for managing the resulting data products by supplying a provenance chain [6] that can be examined at a later date. However, since the abstract workflow description does not indicate which resources will be used for the execution, it is insufficient for the execution of the workflow.

*Corresponding author. E-mail: deelman@isi.edu.

ISSN 1058-9244/05/$17.00

At the execution-level, the workflows need to be *concrete or executable* and describe not only the specific tasks to be executed but also the resources that would be used in the execution of the tasks. As we will describe in this paper, the process of mapping from the abstract to the executable workflow can be automated. During that mapping the original workflow undergoes a series of refinements geared towards transforming the workflow to an executable description and towards optimizing the performance of the overall application. Making the workflow executable may for example involve data stage-in and stage out steps, whereas improving the performance of the workflow may involve reducing the workflow to the minimum number of steps or scheduling several workflow tasks as one unit.

In this paper, we concentrate on the workflow mapping aspect of the problem. We assume that the application is already represented in an abstract workflow form that identifies the application components and their dependencies, as well as the data they use and produce, but that does not specify particular resources to be used. We examine various aspects of the mapping problem from generalizing the type of mapping decisions that need to be made (Section 2) through the description of the mapping software Pegasus (Section 3). We also touch upon the operational complexities of the system and its design. In Section 4 we examine an astronomy application in detail and analyze how the performance of the application can be improved via task clustering techniques. Section 5 presents related work and Section 6 summarizes the benefits of the workflow approach and the Pegasus system.

2. Decisions that need to take place in workflow mapping

We can define an abstract workflow as a directed acyclic graph (DAG) composed of tasks and data dependencies between them. In our work we have used three different methods to create an abstract workflow. The first technique is appropriate for application developers who are comfortable with the notions of workflows and have experience in designing executable workflows (workflows already tied to a particular set of resources. They may choose to design the abstract workflows directly according to a predefined schema. The second method uses Chimera [7] to build the abstract workflow based on the user-provided partial, logical workflow descriptions specified in Chimera's Virtual Data Language (VDL) [7]. Thirdly, abstract workflows may also be constructed using assistance from intelligent workflow editors such as the Composition Analysis Tool (CAT) [8,9]. CAT uses formal planning techniques and ontologies to support flexible mixed-initiative workflow composition that can critique partial workflows composed by users and offer suggestions to fix composition errors and to complete the workflow templates. Workflow templates are in a sense skeletons that identify the necessary computational steps and their order but do not include the input data. When using the CAT software, an input data selector component uses a Metadata Catalog Service (MCS) [10,11] to populate the workflow template with the necessary data. MCS performs a mapping from specific metadata attributes to particular data instances. The three methods of constructing the abstract workflow can be viewed as appropriate for different circumstances and scientist backgrounds, from those very familiar with the details of the execution environment to those that wish to reason solely at the application level.

In any case, all three workflow creation methods result in an abstract workflow representation that needs to be mapped onto the available resources to facilitate execution. The workflow mapping problem can be defined as finding a mapping of tasks to resources that minimizes the overall workflow execution time. The workflow execution consists of the running time of the tasks and the data transfer tasks that stage data in and out of the computation. In general, this mapping problem is NP-complete, so heuristics must be used to guide the search towards a solution.

2.1. Scheduling and mapping horizon

Scientific workflows are often large, consisting of thousands or hundreds of thousands of individual tasks. At the same time the availability and characteristics of the execution resources may vary over time. Clearly, mapping the entire workflow and then committing all the tasks to the selected resources may not be beneficial. For example, by the time the latter portions of the workflow are ready to execute, the resource assignments may no longer be efficient or even feasible. Instead one can plan out the entire workflow but submit only the portions of the workflow that can be run in the near future. We can denote how far into the future to release the workflow as the *scheduling horizon*. This horizon encompasses tasks that can be sent to the execution system. As the execution progresses and the execution environment changes, the initial workflow mapping may need to be adjusted.

Mapping the entire workflow ahead of the execution may be very costly and not appropriate for cases where the execution environment changes rapidly. In such cases it may be beneficial to derive a *mapping horizon* indicating how far into the future (how far into the workflow) to map the tasks. As the workflow is executed, the workflow horizon is increased further and the mapping of the resulting portions of the workflow is being conducted.

The horizons can be expressed in a number of ways, for example as the number of tasks to be released for execution (or mapped) or the set of tasks that fall within a certain time interval on the predicted execution timeline. As can be seen in Section 3.3 the horizons can also be set based on the workflow levels.

In general we can imagine that one can dynamically set the mapping and scheduling horizons based on the cost of mapping the workflow onto the resources, which could be related to the mapping algorithm used and/or to the size of the workflow and the behavior of the execution environment. Figure 1 depicts possible horizon setting in four different situations. Based on the cost of the mapping, one may set a long or short mapping horizon. For example, if the cost is high and not linear with the number of tasks, we may want to map only small portions of the workflow at one time. If the execution environment is fairly static, then it is safe to release the mapped portion of the workflow as soon as they are ready to execute. The horizons may also differ with a greater mapping horizon allowing for a possibly better overall schedule and the shorter scheduling horizon improving the execution time of the workflow as in the case for dynamic environments where the cost of the mapping is manageable.

2.2. Resource allocation

An important parameter of the problem is the information available to conduct the mapping. This information is obtained dynamically from the execution environment. Among such information is:

- The set of available resources, their characteristics (load, job queue length, job submission servers, data transfer servers, etc.)
- The location of the data referenced in the workflow (the data may be replicated and available at several locations)
- The location and characteristics of the software (including the environment that needs to be set up for the software, any libraries that need to be present, etc.)

Given this information, the mapping needs to consider which resources to use to execute the tasks in the workflow as well as from which locations to access the data. These two issues are inter-related because the choice of execution site may depend on the location of the data and the data site selection may depend on where the computation will take place. If the data sets involved in the computation are large, one may favor moving the computation close to the data. On the other hand if the computation is significant compared to the cost of data transfer the compute site selection could be considered first.

The choice of execution location is complex and involves taking into account two main things: feasibility and performance.

Feasibility: (Is a site suitable for execution?)

- Does the user have access to that site? This question could be simple, for example, can the user authenticate now? Or it could be more complex, will the user have access to a resource for the duration of the run of the workflow tasks?
- Does the resource have the necessary software or can the software be staged in?
- Does the resource have enough disk space, memory, etc?

Given the answer to these questions, we can construct a set of feasible resources. Then, given this set we can start analyzing the performance tradeoffs.

Performance tradeoffs:

- *Data reuse*. Is it better to re-produce the data or access them? For example, some intermediate data products or even the final products may be already available on some storage system, so we need to evaluate whether it is more efficient to access that data or to recompute it.
- *Which site to access the data from*? As already mentioned, data may be replicated, the decision about which location to use to retrieve the data may depend on the bandwidths between the data source and the execution site, the performance of the storage system and the performance of the destination data transfer server.
- *Software stage-in*. Is it better to use a site which already has the software or pre-stage the software? For example, there may be a site that has high-availability and performance, but that does not have the necessary software, would it be worthwhile to stage in the software.

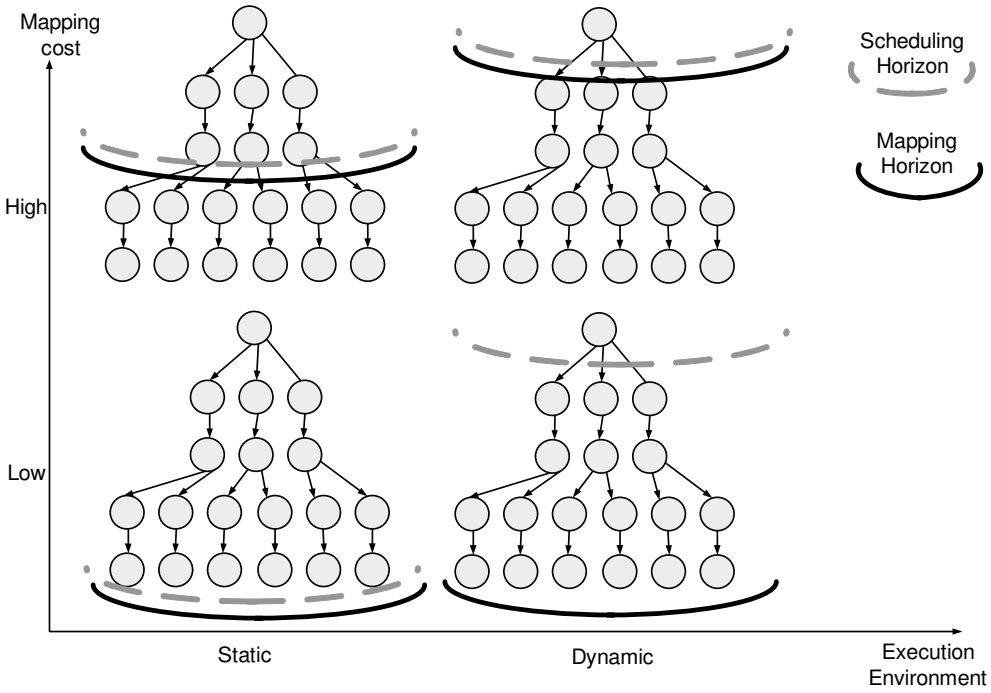

Fig. 1. Setting scheduling and mapping horizons based on mapping costs and the behavior of the execution environment.

– *Which compute resource to use for the computation*? In some sense the decision of which compute resource to use is simpler than which data to access. There has been much work of the years in scheduling tasks onto resource, using analytical models [12], empirical models based on past performance [13] and simulation [14]. There are however a few aspects of a distributed environment such as the Grid [15] and of the applications that make the problem more complex. Among them are the sharing of resources among many users, the dependency between tasks and the possible use and production of large data sets. As part of the resource allocation problem one may also need to consider parallel applications and their special needs such as how many processors and what type of network interconnects are needed to obtain good performance.

2.3. *Optimizing workflow execution through workflow restructuring*

Once the target systems for a portion of the workflow are identified, there are still optimizations that can be taken into consideration to improve the overall workflow performance. These involve restructuring the workflow so that the jobs can be run on the systems as efficiently as possible.

In some cases it is possible and efficient to reduce the workflow based on the availability of the intermediate data products. For example, it is possible that several scientists within a collaboration are running similar workflows and thus some of the data products referred to in the workflow may already exist. In that case the workflow can be automatically reduced by removing the redundant computations.

Another possible restructuring aims at increasing the granularity of computation and thus reducing the scheduling overhead. The granularity can be increased by combining (clustering) several tasks and treating it as a single unit for the purposes of mapping and scheduling. The question then is: how many tasks destined for a specific location should be clustered together?

The third type of restructuring involves scheduling jobs onto multi-processor systems. On these systems it may be more efficient to request more than one processor at a time, since the delay in the scheduling queues may be significant. If there are multiple tasks scheduled for the mutli-processor system, it may be beneficial to cluster them together and run them as one schedulable unit. This would result for example in allocating a

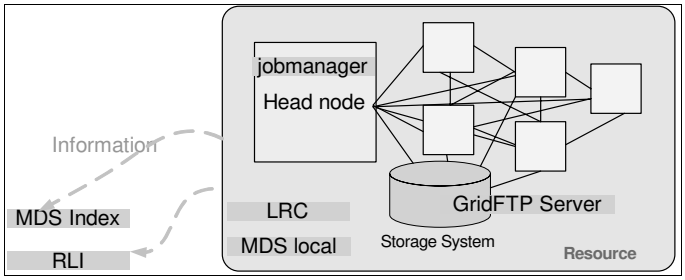

Fig. 2. An example execution host configuration.

given number of processors and using them to run the tasks possibly in a master/slave fashion.

For the latter two optimizations, it may be desirable to cluster the jobs prior to or along with the resource assignment. We examine the benefits of task clustering in Section 4.

In general the amount of decisions one would like to make in order to optimize workflow execution is great so in practice, in current workflow mapping systems such as Pegasus, only a subset of decisions is taken into account at any one time.

3. Pegasus design

Pegasus [1,4,16], which stands for Planning for Execution in Grids, is a framework that maps complex scientific workflows onto distributed resources such as the Grid. Since no single system can optimize a wide variety of workflows and environments, we designed the framework in a way that allows users to customize various aspects of the system.

3.1. Target execution system overview

In order to understand the functionality of Pegasus it is important to describe the execution environment in which the workflows are to be executed. We assume that the environment is a set of heterogeneous hosts connect via a network, often a wide area network. The hosts can be single processor machines, multi-processor clusters and high-performance parallel systems.

Figure 2 shows a typical execution host, with a head node visible to the network, possibly some other hosts that form a pool of resources and a storage system. In order to be able to schedule jobs remotely, the resource needs to have appropriate software deployed. In our work, we use the Globus Toolkit [17] to provide:

- Remote job submission and management (via the GRAM jobmanager [18]).
- Remote data stage-in and stage out (via GridFTP [19]). GridFTP allows for high-performance, secure data transfer in the wide area networks.
- Information about the state of the resources (via the Monitoring and Discovery Service (MDS) [20]). MDS provides information about the number and type of available resources, static characteristics such as the number of processors and dynamic characteristics such as the amount of available memory.
- Information about the data available at the resource (via Replica Location Service's (RLS) [21] Local Replica Catalog (LRC)). RLS is a distributed replica management system consisting of local catalogs that contain information about logical to physical filename mappings and distributed indexes that summarize the local catalog content.

In order to collect and organize information about multiple sites, we use the indexing capabilities of MDS and RLS (the Replication Location Index – RLI). This collective information is utilized by Pegasus in the resource and replica selection decisions.

In order to use Pegasus in such an environment, a resource, which could be a user's desktop, needs to be setup to provide Pegasus itself, DAGMan and Condor-G [22], the latter two provide the workflow execution engine and the capability to remotely submit jobs to a variety of Globus-based resources. We name this resource a *submit host*. The submit host also maintains information about the user's application software installed at the remote sites (in the Transformation Catalog (TC) [23]) and about the execution hosts of interest to the user (in the Pool Configuration file). The Pool Configuration file is dynamically constructed using data provided by MDS and additional information provided by the user. In addition to the general resource

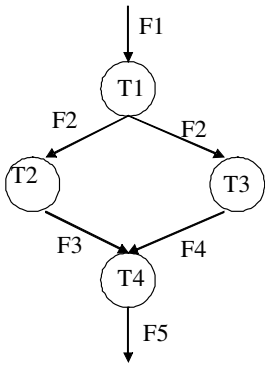

Fig. 3. An Example Abstract Workflow Composed of Four Tasks. Ti stands for a logical transformation (task). Fi is a logical filename.

information usually found in MDS it contains the information about the remote GridFTP, RLS servers, etc. The submit host can also serve as a local execution platform for small workflows or for debugging purposes.

3.2. Pegasus functionality

Pegasus transforms an abstract workflow to an executable (concrete) workflow through a series of refinements. The abstract workflow (Fig. 3 shows an example abstract workflow) is composed of tasks described in terms of logical transformations and logical input and output filenames. The abstract workflow is independent of the resources. Pegasus' goal is to find a good mapping of the tasks to the available resources necessary for execution. In Section 2 we described the many choices that can be made when allocating resources. Here we detail the approach taken by Pegasus.

Figure 4 depicts the steps taken by Pegasus during the workflow refinement process.

3.2.1. Defining the set of available and accessible resources

First, Pegasus consults MDS and the Pool Configuration file to check which resources are available. Additionally, Pegasus may try to authenticate to these resources using the user's credentials. Thus, the possible set of resources may be reduced to a minimum.

3.2.2. Workflow reduction

The next step may modify the structure of the abstract workflow based on the available data products. Pegasus consults the Replica Location Service to determine which intermediate data products are already available. Based on this information, Pegasus may reduce the workflow to contain only the tasks necessary to generate the final data products. In the extreme case, if the final data products are already available, no tasks will be scheduled except perhaps a data transfer (to the user-specified location) and registration of the desired data products. An example reduction is discussed as part of Section 3.4.

3.2.3. Resource selection

At this point we have the minimal abstract workflow in terms of the number of tasks. The workflow reduction was made based on the assumption that it is more efficient to access the data then to recompute it. Given the minimal workflow, a site (resource) selection is performed. This selection can be done based on the available resources and their characteristics as well as the location of the required input data. The type of site selection performed is customizable as a pluggable components within Pegasus. The system incorporates a choice of a few standard selection algorithms: random, round-robin and min-min. These algorithms can be applied to the selection of the execution site as well as the selection of the data replicas. The selection algorithms make use of information available in MDS, the Pool Configuration file (resource characteristics), the Transformation Catalog (the location of the application software components), and RLS (the location of data). It is also possible to delay data replica selection until a later point, in which case RLS is not consulted at this time. Additionally, users may wish to add their own algorithms geared towards their application domain and execution environment. These algorithms may also rely on additional or different information services and these can be plugged into Pegasus as well.

3.2.4. Task clustering

Pegasus provides an option to cluster jobs together in cases where a number of small granularity jobs is destined for the same computational resource. In Section 4 we examine the value of clustering, here we briefly described its mechanics. During clustering we consider only independent tasks, so that they can be viewed by the remote execution system as a single entity. These tasks also need to be destined for the same execution system. The task clusters can be executed on a remote system either in a sequence or if feasible and appropriate they can be executed using MPI [24] in a master/slave mode. In the latter case an initial number of processors is requested and the clustered tasks are being dispatched (sent to the remote site) to them as the constituent task execution is completed.

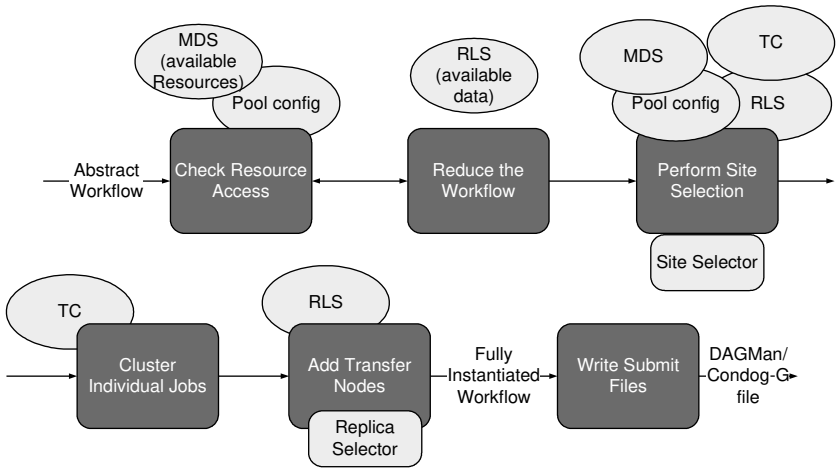

Fig. 4. Pegasus' Logic. Corresponds to the "gencdag" command.

3.2.5. Adding data stage-in, stage-out and registration tasks

The abstract workflow contained only nodes representing computations. Since the workflow can be executed across multiple platforms and since data need to be staged in and out of the computations, Pegasus augments the workflow with tasks that explicitly perform data transfers. If during site selection data replica selection was not performed it can be done at this point. Again, the user has the option of using Pegasus provided algorithms or supply their own. These algorithms are used to determine which of possibly many data replicas will be used as a data access locations. Once the location is determined, a node is placed in the workflow and a dependency to the corresponding computation is added. Additionally, where appropriate, intermediate and final data products may be registered in the data catalogs such as the RLS or a metadata catalog to enable subsequent data discovery and reuse. The data registration is also represented explicitly by the addition of registration tasks and dependencies to the workflow.

3.2.6. Submit file generation

At this point all the compute resource and data selection has been performed and the workflow has the structure corresponding to the ultimate execution structure that includes computation, data transfer and registration. The final step is to write it out in a form that can be interpreted by a workflow execution engine such as for example DAGMan. Once this is accomplished, the resulting submit files can be given to DAGMan and Condor-G for execution. DAGMan will follow the dependencies in the workflow and submit available tasks to Condor-G which in turn will dispatch the tasks to the target resources.

The sequence of refinements depicted in Fig. 5 is currently static, but one can imagine constructing the sequence dynamically based on user and/or application requirements.

3.3. Setting the mapping horizon

As we mentioned in Section 2.1, it is often beneficial to set a mapping and/or scheduling horizon which determines which parts of the workflow will be refined and which tasks scheduled at any given time. Although the space of possible solutions is large, we implemented a basic horizon setting mechanism within Pegasus. In this case the scheduling horizon is equal to the mapping horizon. In our initial implementation the mapping horizon is set statically based on the structure of the workflow. The mapping horizon is simply determined by the levels of the workflow as described below. Clearly this type of horizon definition is not efficient for all applications so the user can provide their own horizon setting function that partitions the workflow into subworkflows and maintains the dependencies between them.

Once the subworkflows are set, Pegasus and DAGMan are then used to refine the partitions in the order of dependencies. Figures 5 and 6 illustrate the process for a level-based partitioning. The levels refer to the depth of the tasks in the workflow. Figure 5 shows a 3-level workflow being partitioned. The resulting new workflow, which we term a *MegaDAG* consists of three partitions sequentially dependent on each other. Intuitively we would like to refine the first partition and

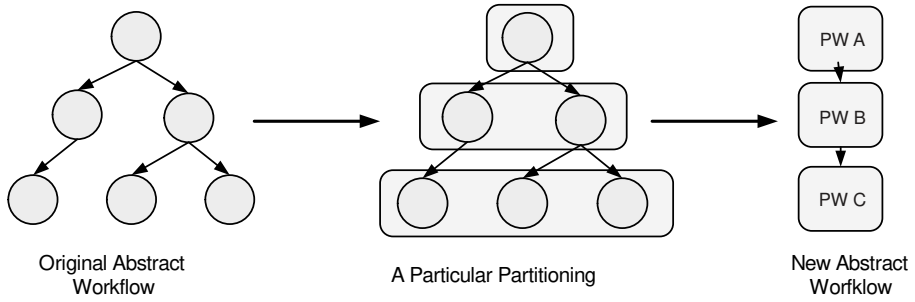

Fig. 5. Level-based Partitioning.

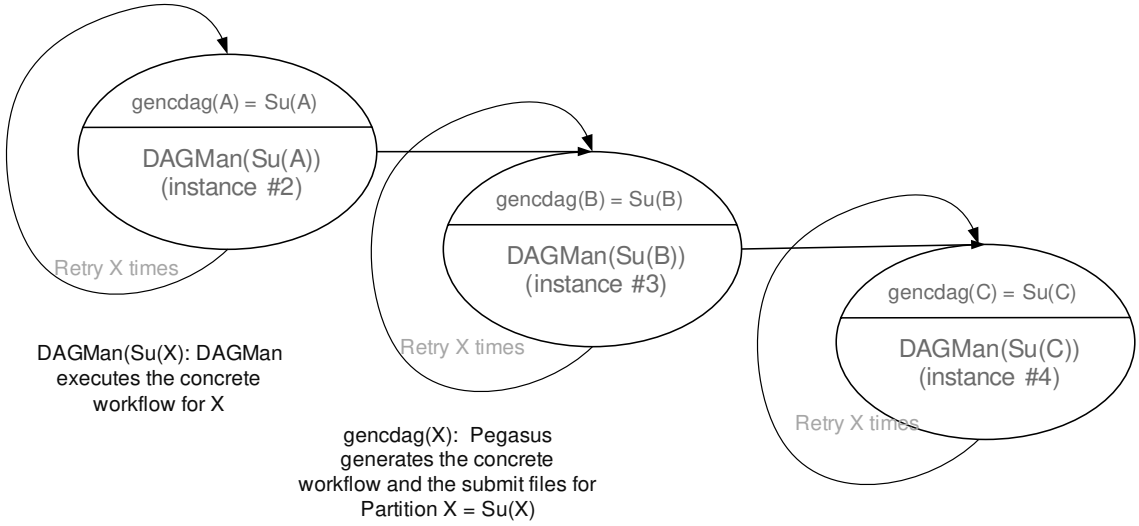

Fig. 6. The MegaDAG guides workflow refinement. It is submitted to the #1Instance of DAGMan.

then schedule and execute it before proceeding to refine the remaining partitions. In the example in Fig. 5 the partitioning is static, but we could also generate the first partition (PW A), refine and execute it and only then recursively partition the remainder of the workflow.

In order to support the static partitioning and refinement, Pegasus constructs a MegaDAG, shown in Fig. 6. This DAG is a "recipe" of the refinement and execution of the original workflow. The ovals correspond to the workflow partitions. The directives in each oval indicate the actions to be taken when a workflow execution system (in this case DAGMan) processes the tasks. First, we see a call to gencdag on the first partition (A). Gencdag stands for the concrete (executable) DAG generator and corresponds to the Pegasus' workflow refinement process (as illustrated in Fig. 4). Once Pegasus/gencdag generates the submit files for the particular partition, DAGMan is called to execute that partition. If the process of refinement or execution fails, it can be retried a given number of times. After DAGMan successfully finishes the execution of the refined partition A, the next directives are followed (refinement and execution of partition B), etc.

In order to assure that the directives in the MegaDAG are followed, we use DAGMan (instance #1, representing the first invocation of DAGMan). It calls gencdag on the workflow partitions (for example partition A) and then invokes another instance of DAGMan (instance #2) to execute the newly generated executable workflow (Su(A)). Once the second instance of DAGMan successfully completes the execution of the refined partition A, the first instance of DAGMan continues with the invocation of gencdag on partition B and so on.

Figure 7 demonstrates the process from the point of view of the user. The user submits the abstract workflow to the system, which in turns generates the MegaDAG and submits it to DAGMan for execution. As a result tasks are released to Condor-G which submits them to the remote resources for execution.

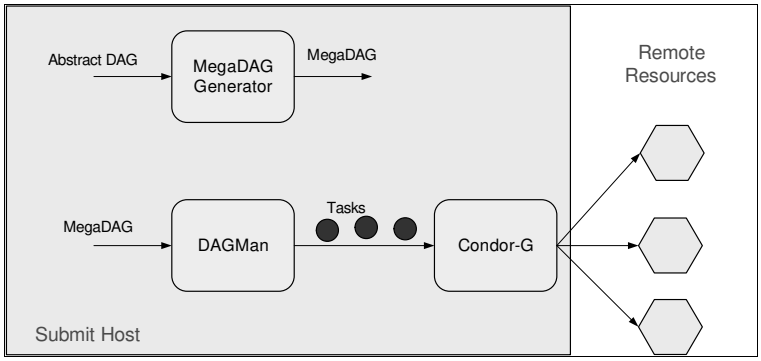

Fig. 7. Overall partition-based workflow refinement process as seen by the user.

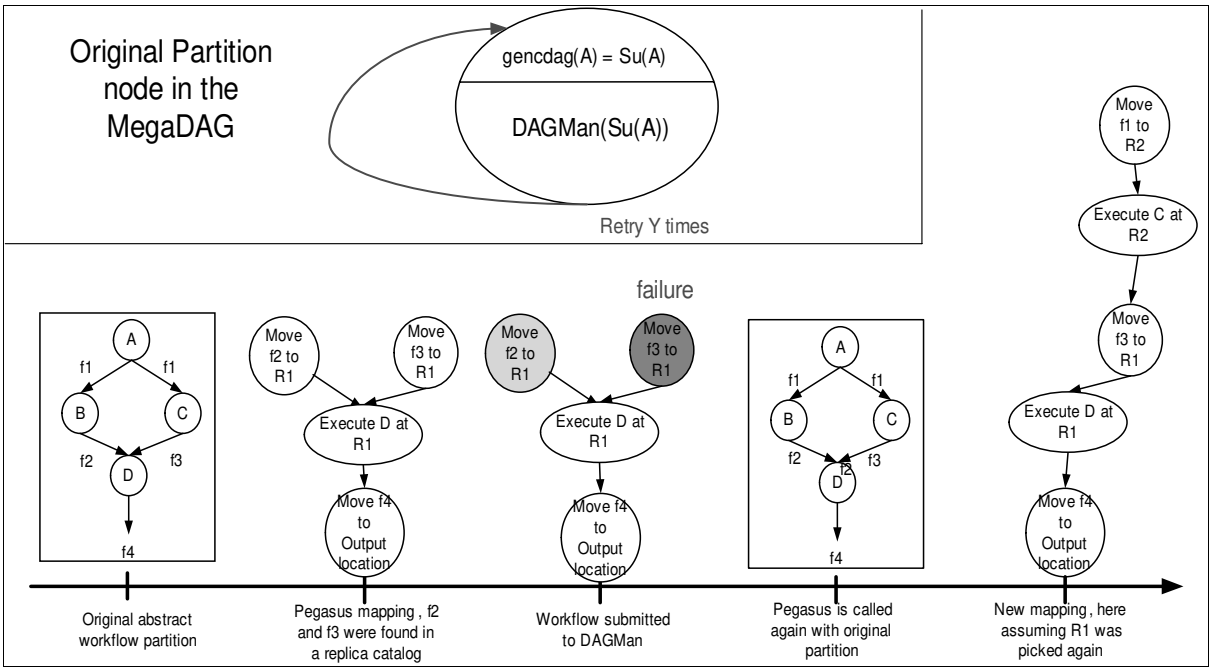

Fig. 8. Recovery from failure. The top left shows the MegaDAG node that is being refined and executed. The bottom of the figure shows (left to right) the progression of the refinement and execution process.

3.4. Partition-level failure recovery

Pegasus and DAGMan can be a powerful combination that enables a certain degree of remapping in case of failure. As explained above, in the MegaDAG each task consists of a workflow partition mapping step followed by a DAGMan execution of the mapped workflow. If either of these steps fails due to a mapping failure or due to the execution, the entire task can be retried. An example of a situation where this is particularly useful is shown in Fig. 8. We start off with a partition containing a subworkflow in a shape of a diamond, consisting of 4 tasks. As mentioned before, Pegasus reduces the workflow based on the available data products. In this case Pegasus found that file f_2 and f_3 are already available. Because the two files are available, tasks B and C do not need to be executed and consequently neither does task A. The resulting executable workflow is shown next. It consists of four nodes, the first two stage in files f_2 and f_3 to the execution location R_1. Then task D is to be executed at location R1 and finally the data is to be staged out to the user-specified location. Given this mapping, DAGMan proceeds with the execution of the workflow. Let's assume that file f_2 is successfully staged in, but for some reason there is a failure when accessing or

Table 1
Characteristics of the tasks/transformations in the montage workflow

Level	Transformation Name	Description	No. of jobs at the level	Runtime of each job at the level (in seconds)
1	mProject	Reprojects a single image to the image parameters and footprints defined in a header file.	180	6
2	mDiffFit	Finds the difference between two images and fits a plane to that difference image	1010	1.4
3	mConcatFit	Does a simple concatenation of the plane fit parameters from multiple mDiffFit jobs into a single file	1	44
4	mBgModel	Models the sky background using the plane fit parameters from mDiffFit and computes planar corrections for the input images that will rectify the background across the entire mosaic	1	32
5	mBackground	Rectifies the background in a single image	180	0.8
6	mImgtbl	Extracts the FITS header geometry information from a set of files and stores it in an image metadata table	1	3.5
7	mAdd	Co-adds a set of reprojected images to produce a mosaic as specified in a template header file	1	60

transferring f_3 and the data transfer software (in our case GridFTP) returns an error Given this failure the DAGMan execution of the partition fails as does the entire original MegaDAG node representing the refinement and execution of the partition. Upon this failure the MegaDAG node is resubmitted for execution and the refinement (gencdag(A) and execution(Su(A)) are redone. In the final step we see the executable workflow that resulted from the Pegasus/gencdag mapping. We notice that Pegasus took into account that f_2 was already successfully staged in and at the same time, the reduction step did not reduce task C because f_3 needs to be regenerated (assuming there was only one copy of f_3 available). In this case we also assume that f_1 is available thus task A still does not need to be executed. Given this new mapping DAGMan is invoked again to perform the execution.

4. Application study

4.1. Montage workflow

Montage [3,25] is an application that constructs custom science-grade astronomical image mosaics on demand. Figure 9 shows the structure of a small Montage workflow. The figure only shows the graph of the abstract workflow. The concrete workflow would contain data transfer and registration nodes in addition to those shown in the figure.

The workflow can be divided into levels as mentioned in Section 3.3. The numbers inside the vertices in the graph show the level number of the job in the workflow. Table 1 gives a description of the workflow and the number of the jobs for a representative 2 degree mosaic centered around the celestial object M16. The inputs to the workflow include the input images in standard FITS format (a file format used throughout the astronomy community), and a "template header file" that specifies the mosaic to be constructed. The workflow can be thought of as having three parts, including reprojection of each input image to the coordinate space of the output mosaic, background rectification of the reprojected images, and coaddition to form the final output mosaic.

4.2. Target execution system model

In Section 3 we described the functional aspects of the target execution system. Here we examine the system from the point of view of remote job scheduling and execution performance. In particular, we study how to improve the overall workflow performance by reducing the scheduling overheads incurred by the workflow tasks. The system consists of a user submitting an application for execution on multiple grid resources (sometimes referred to as sites below) which are possibly geographically distributed and belong to different administrative domains as shown in Fig. 10.

Each site in the Fig. 10 belongs to a different administrative domain and consists of a cluster of (in this case) homogeneous machines. In this configuration, only one particular machine (called the head node) on each site is used for submitting jobs to that site. For each site, the big oval on the left hand side depicts the head node for the site and the smaller circles depict the worker nodes for the site. Each site might be shared by many users.

As we described previously, in order to develop an application for the Grid environment, the user con-

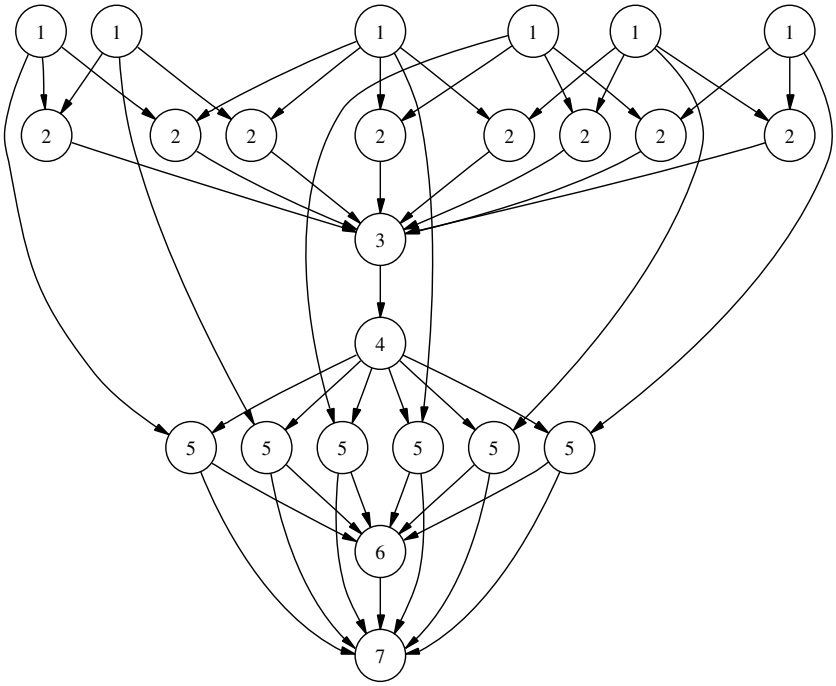

Fig. 9. A small montage workflow.

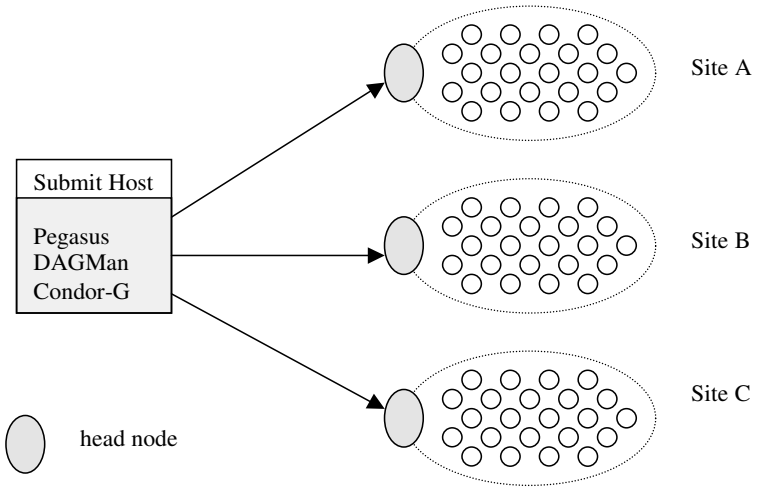

Fig. 10. Pegasus and DAGMan schedule and submit jobs to multiple sites.

structs an application workflow in the form of an abstract workflow, then uses Pegasus to do the resource allocation for the jobs in the workflow and to generate the Condor submit files. These submit files specify the head node of the remote site to which the job has to be submitted and any required input files. Condor DAGMan takes the workflow specification and submits the ready jobs to the local Condor queue while maintaining the dependencies between the jobs in the workflow. Condor-G is used to schedule the submitted jobs in the workflow on the remote resources. In this scenario, the Condor software has to be installed on the user's local machine and the Globus software has to be installed on the head nodes of the various sites. The Globus installation at the remote sites takes care of receiving the job specification and submitting it to the local resource management system such as PBS [26], Condor, LSF [27], etc.

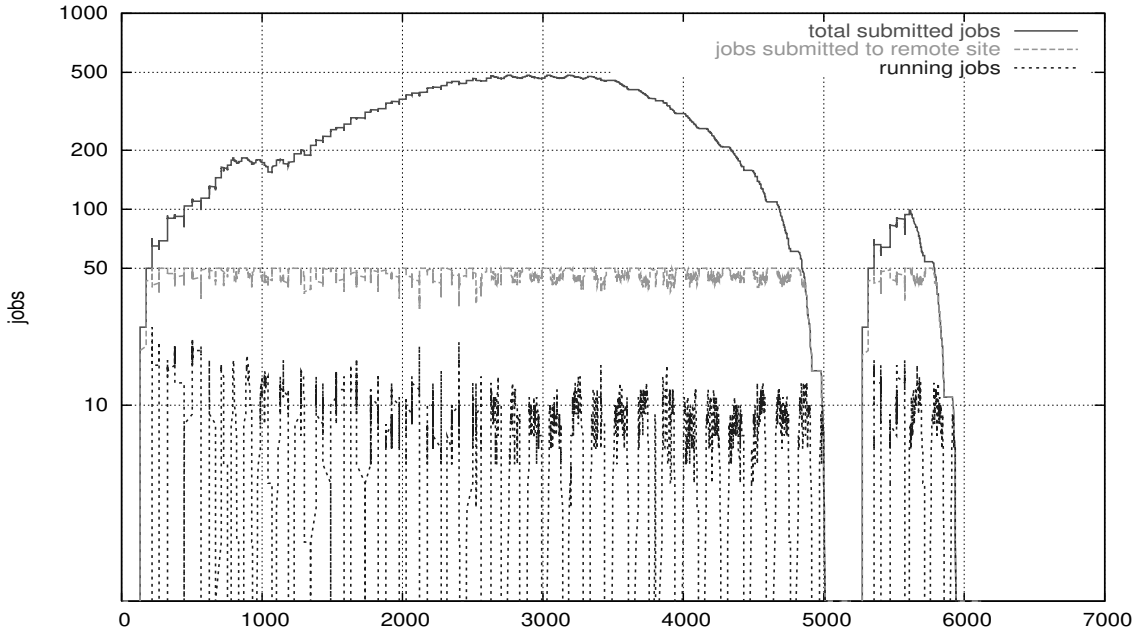

Fig. 11. The total number of submitted jobs in the system (top line), the number of jobs submitted to the remote site (middle line) and the number of jobs actually running (bottom line) as time progresses (in seconds). The jobs are shown on a logarithmic scale.

In this model, the delay that a job encounters after being submitted to the local Condor queue and before starting execution on a remote resource is composed mainly of the following two components:

1. The time spent waiting in the local Condor queue before being submitted to a remote site.
2. The time spent waiting in the remote site queue.

One issue that comes up in real Grid deployments is the overloading of the head node with too many jobs. To avoid this situation, one can limit the number of jobs submitted to a particular remote site. Typically, in our experiments we tell Condor to limit the number of jobs sent to a remote site to 50. Ideally, this limit would be dynamic and based on the load on the head node. Setting the limit too low can increase the workflow completion time dramatically and setting it too high can cause the head node of the remote site to crash because of overload. During our experiments we observed that the limit of 50 worked well in most cases. However, due to this limit, a job destined for a particular remote site may have to wait in the local Condor queue until the number of jobs that can be submitted to that particular site falls below the maximum allowed. Each remote site uses a resource management system such as PBS, LSF, or Condor to queue up the submitted jobs and start their execution as the resources become available. Thus, the job may have to wait in the remote queue before the resources become available. In the following section, we study the effect of the above-mentioned delays on the completion time of the Montage application and examine ways to reduce the delays to optimize the overall workflow execution.

4.3. Experiment

4.3.1. Standard execution

In our experiments, the submit machine was located at ISI and the jobs were scheduled to execute on the TeraGrid's NCSA cluster. In our work we use only one remote cluster in order to eliminate the effects of data transfer between remote clusters. The NCSA Teragrid Linux cluster consists of 887 cluster nodes running the SuSE linux OS. 256 nodes are dual 1.3 GHz Intel Itanium 2 processors and 631 are dual 1.5 GHz Intel Itanium 2 processors. The cluster nodes are managed by the PBS resource management system and the Maui [28] scheduler is used to schedule tasks onto the nodes. The NCSA Teragrid cluster contains a GRAM gatekeeper that can be used for submitting tasks remotely to the PBS queue and several GridFTP servers for transferring data. Shared file systems that are visible from all the nodes in the cluster can be used for sharing data between tasks. DAGMan submits the ready jobs in the workflow to the local Condor queue and waits for any further events. Condor-G submits the specified

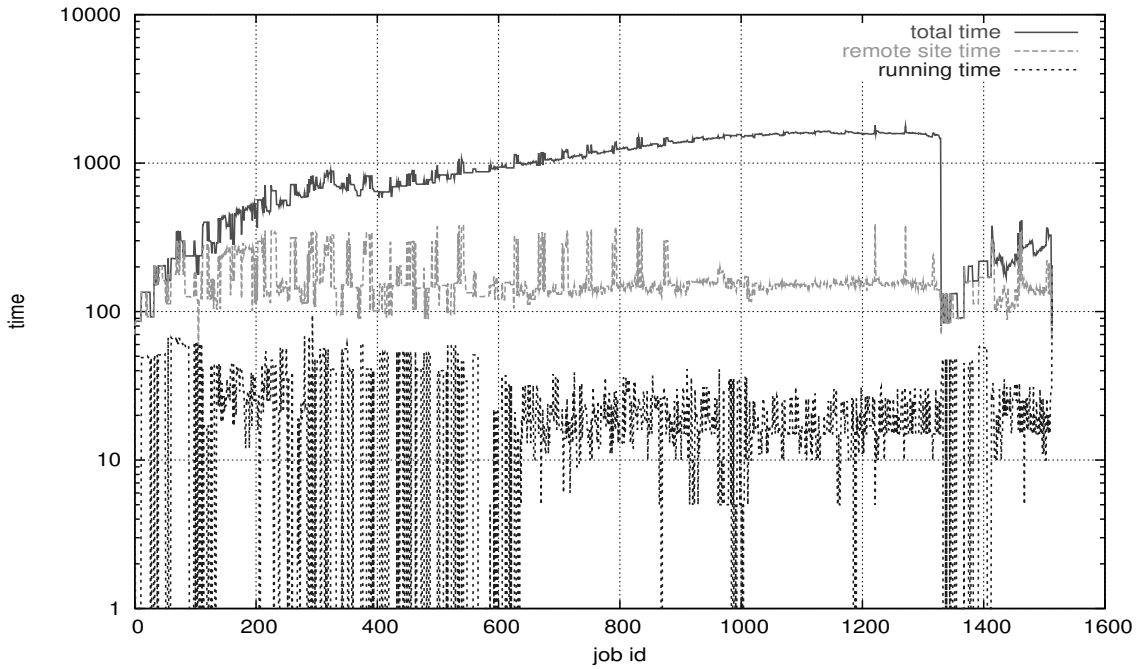

Fig. 12. The time (in seconds) each job spends in the system, on the remote site and the actual execution time of the job. The time axis is shown on a logarithmic scale.

number of jobs from the local queue to the Globus jobmanager on the head node of the remote site. For example, there are 180 top-level jobs in the workflow that are ready for execution but only 50 of them are submitted to the remote cluster. The remaining 130 have to wait in the local condor queue until any already submitted job finishes execution. Figure 11 shows the total number of jobs in the system, the number of jobs submitted to the remote system, and the number of jobs actually executing as time progresses. This figure shows the wide difference between the total number of submitted jobs, the number of jobs actually submitted to the remote site, and the number of jobs actually running at a given time. This difference is due to the limit on the maximum number of jobs that can be submitted to the remote site and the limited number of machines that are available at the remote site. Figure 11 also shows that the number of submitted jobs to the remote site is always less than or equal to the specified limit of 50 even though there may be 10 times more jobs ready for execution in the local Condor queue.

Figure 12 shows the total time each job had to spend in the system (indicated by the line marked total time – top line in the figure), the time it spends on the remote site (indicated by the line marked remote site time – middle line), the time it is actually running on a machine on the remote site (indicated by the line marked running time – bottom line). The time each job spends in the system is composed of the time the jobs spent waiting in the local queue and the time it spends on the remote site. The time each job spends on the remote site consists of the time it spends waiting for a machine to become available and the time it is actually running. The time spent in the local queue is not explicitly marked in the figure but can be calculated as the difference between the total time and the time spent on the remote site. The execution time is very small in comparison to the total time and is barely noticeable at the bottom of the graph. The X-axis in Fig. 12 consists of the job identifiers and the Y-axis is the time in seconds (on the logarithmic scale).

4.3.2. Sequential clustering

As we see from Fig. 12, a majority of the jobs in the workflow spent most of their time (upto 90%) waiting in the local Condor queue before being submitted to the remote site for execution. Since the jobs have to wait in the local Condor queue because we set a limit of 50 jobs being sent to a remote site at any given point of time, one possible alternative to reduce this waiting time is to cluster the jobs in the workflow so that we reduce the number of jobs as seen by Condor. This also reduces the load on the head node of the remote site. By clustering, we merge two or more jobs in the original

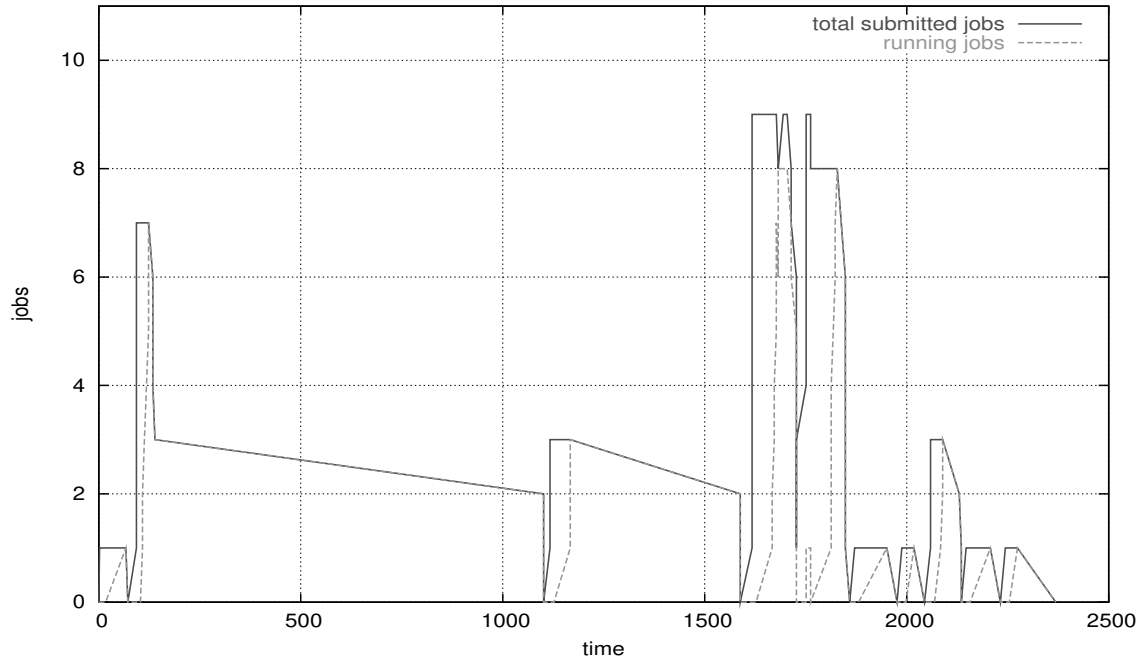

Fig. 13. The total number of submitted jobs in the system, the jobs submitted to the remote site and the number of jobs actually running at any particular time.

workflow into a single cluster. This cluster is submitted to Condor as a single job. When it starts executing on the remote resource, it executes all its constituent jobs sequentially.

Clustering of jobs in the workflow increases the computational granularity of jobs submitted to Condor. Clustering effectively modifies the workflow graph. However, all the dependencies of the original workflow should be preserved in the new graph. Each cluster should be a convex subgraph of the original workflow i.e. each directed path between the jobs in the cluster should be fully included in the cluster.

Our current approach to clustering consists of assigning levels to the jobs in the workflow and forming clusters from the jobs at the same level. The jobs in the workflow that have no parent jobs are assigned level 1. The jobs that become ready for execution when the level 1 jobs have completed successfully are assigned level 2 and so on. The jobs within each level are independent of each other and can be clustered together without violating any of the dependencies in the workflow. Figure 9 shows the jobs in the Montage workflow and the levels assigned to the jobs in the workflow (indicated by the numbers inside the vertices).

In our next experiment, we took the same workflow and clustered 60 jobs at the same level in a single cluster. The clustered workflow now has 35 jobs instead of about 1500 in the original workflow. Figure 13 shows the total number of submitted jobs in the system, and the number of jobs actually running at any particular instant of time after the clustered workflow has been submitted to DAGMan for execution. Since the number of ready jobs at any point of time now is less than the limit of 50, they do not have to wait in the local Condor queue. Consequently the total number of submitted jobs in the system is equal to the number of jobs submitted to the remote site. Each job in Fig. 13 is a cluster of jobs from the original workflow.

There are a few important differences between the timing results in Figs 11 and 13. Most importantly, the workflow now completes in 2400 seconds whereas earlier it took more than 6000 seconds to complete even though the number of jobs running, and hence the number of allocated machines at any particular instant of time are roughly the same. Thus, the resource availability at the remote site has not changed but the time each job spends in the system has reduced as seen in Fig. 14. The second most important observation is that since the number of jobs as seen by Condor is less as compared to the earlier case and less than the job submission limit of 50 jobs per remote resource, the jobs now do not have to wait in the local Condor queue. Each job is submitted to the remote site as soon as it becomes ready for execution. As we had seen earlier, most of the jobs

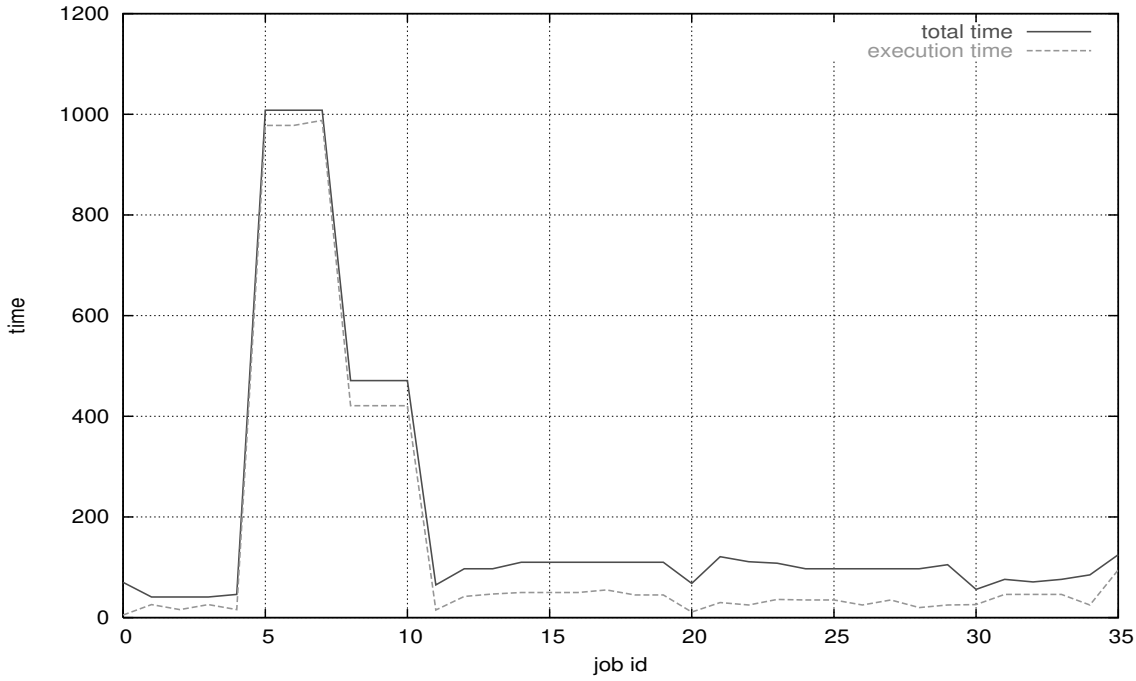

Fig. 14. The distribution of time each job or cluster spends in the system.

in the workflow spent a large portion of their time in the system waiting in the local Condor queue. Thus, by eliminating this wait time, we were able to reduce the workflow completion time by more than 50%. Figure 14 shows the distribution of time each job spends in the system (similar to what was shown in Fig. 12). Each job now spends all of its time on the remote site, either waiting in the queue or executing.

As Figs 13 and 15 show that clustering helps in reducing the completion time of the workflow. It does so by reducing the number of jobs in the workflow so the waiting time for jobs in the local queue is eliminated. This also helps in decreasing load on the head node of the remote site since it takes some CPU and memory resource to track each submitted job. It also helps in decreasing the load on the local machine since instead of hundreds or thousands of jobs in the Condor queue; there are now only few of them. Increasing the computational granularity of jobs improves the efficiency with which the remote resources are used.

4.3.3. MPI clustering

Even after clustering we still have the limit of 50 submitted jobs per remote resource (or some other set limit that prevents the remote head node from overloading). This implies that the system can only submit 50 clusters for execution on the remote resource even though there may be more resources available. In order to utilize all the available resources on the remote site (and keeping in mind that the target system is a cluster that supports parallel execution), we make each cluster an MPI job. Therefore, each cluster can use more than one resource for execution. In our clusters all the jobs in a cluster are independent of each other and so it is simple to write a MPI wrapper which can execute the jobs in the cluster using the master/slave approach.

For the next experiment, we cluster the jobs in the original workflow with 60 jobs per cluster as before. In this case, each cluster is an MPI job that uses 10 processors for execution. Thus n running clusters would use $n * 10$ remote processors for execution. Figure 15 shows the number of jobs in the system and the number of running jobs as time progresses. In this figure, we do not differentiate between the total number of jobs in the system and the number of jobs submitted to the remote site since there is little difference between the two. As we can see, the maximum number of jobs running simultaneously is 8 and therefore 80 processors or machines (assuming 1 processor per machine) were being used for workflow execution at that point. This would not have been possible earlier as we would have been able to use only 50 processors for 50 submitted clusters. In addition, the workflow completion time reduces to about 1420 seconds.

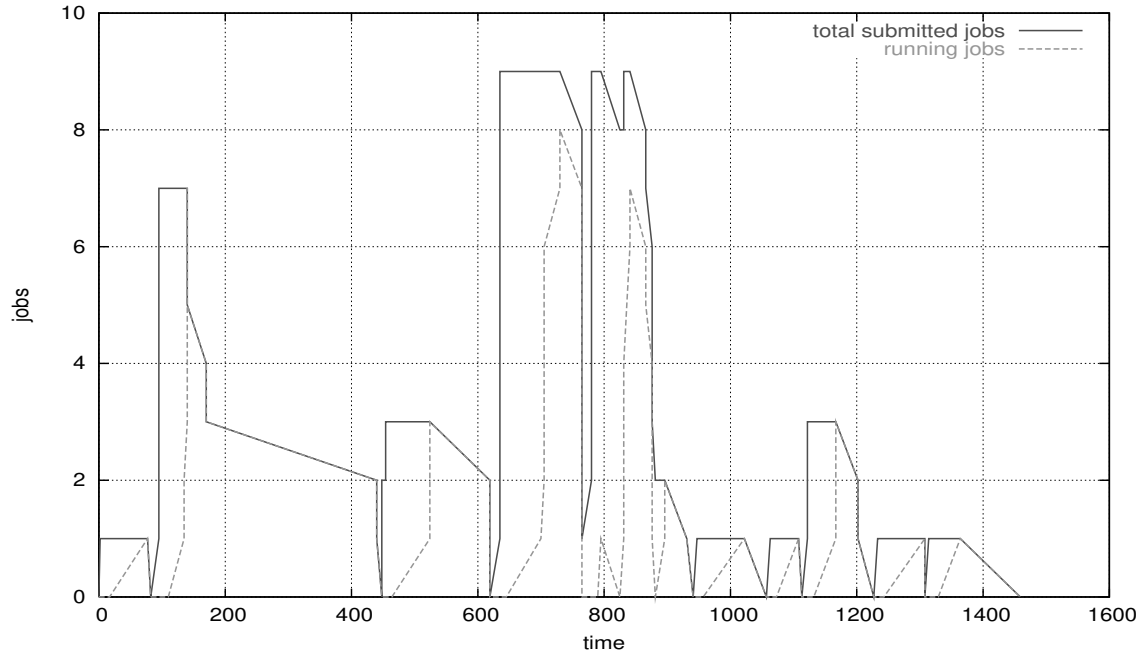

Fig. 15. The number of total and running clusters in the system.

Figure 16 shows the total time spent and the running time for each cluster in the workflow. In the earlier case of sequential clustering, the average wait time for each cluster on the remote site was approximately 50 seconds but in this case of clustering with MPI, the wait time is around 100 seconds. This increase in wait time is expected since now each cluster is requesting 10 processors whereas earlier it was requesting only a single processor. In case of sequential clustering, each cluster requires only a single processor for execution and so may get scheduled faster when the remote scheduler takes advantage of the backfilling opportunities.

4.4. Discussion

We studied the overhead associated with executing an application workflow over the Grid resources using standard Grid Middleware components such as Condor and Globus. We found that for small-grained application such as Montage this overhead is mostly composed of the queuing delay each job in the workflow encounters on the local machine as well as on the remote pool. We presented an approach which clusters jobs in the workflow as a possible solution to eliminate or reduce these delays. In our results, the clusters were formed based on the level of a task in a workflow. We used two possible execution modes for the resulting clusters: sequential and MPI. The MPI-based clustering is able to utilize more resources on the remote site and hence should be used if more resources are available than the number of jobs that can be submitted to the remote site.

Clustering can also improve or even make feasible the execution of very large workflows which normally cannot execute efficiently because of the lack or overloading of resources. For example, there are benefits to clustering in cases where sites have a limit on the number of machines that a particular user can use at any particular instant of time. Clustering can be very helpful in that case because it reduces the number of jobs that are sent to a cluster without reducing the amount of work done. Clustering can also be used to reduce the load on the local machine and the head node of the remote site. Because clustering reduces the number of workflow nodes that the execution system needs to manage, it also enables the execution of very large workflows. For example Montage workflows containing thousands of nodes at the various levels are almost impossible to execute without using clustering. The Montage workflow used in Section 4.2 is used to create a 2 degree mosaic and has about 1500 nodes. A concrete Montage workflow that creates a 6 degree mosaic can contain more than ten thousand nodes. Due to the shared nature of the resources, such a large workflow can take days to complete in the absence of failures. However when properly clustered, the workflow can be completed in

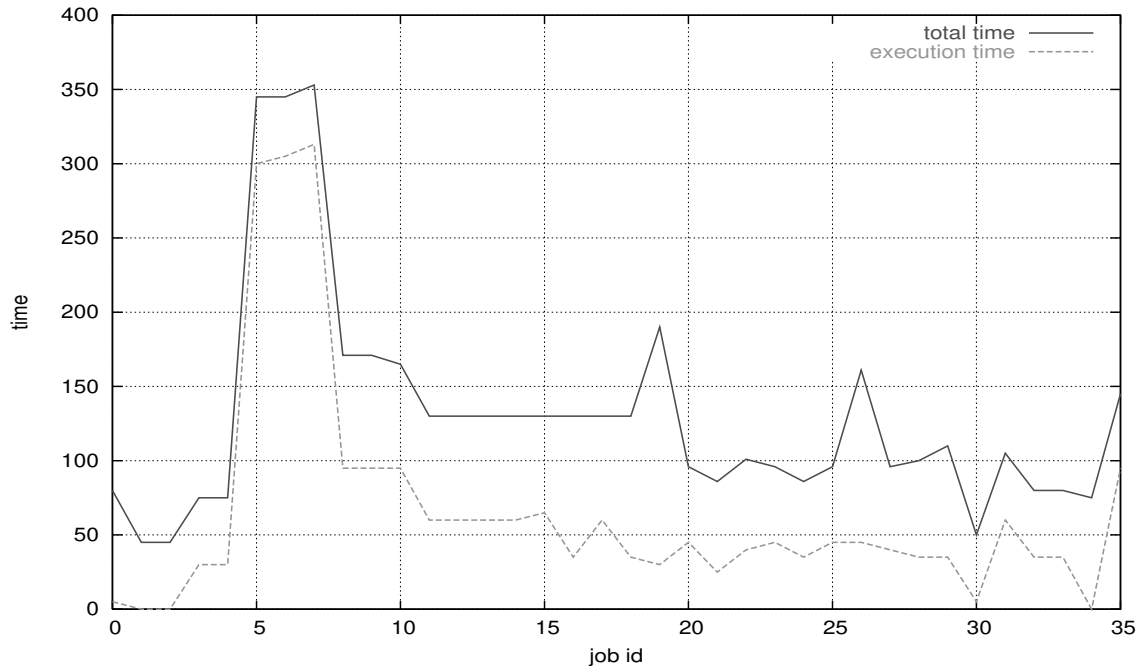

Fig. 16. The distribution of time spent by clusters in the workflow.

less than two hours. Fewer jobs in the workflow also mean less opportunity for failure.

We studied the effect of clustering on a Montage workflow that is a fine computational granularity workflow. The runtime of jobs in the Montage workflow is very small as listed in Table 1. The effect of clustering on coarse-grained workflow is not yet clear although we suppose that the benefits of clustering will be dimished. In addition, clustering increases the run times of jobs submitted to the remote sites and thus can lose scheduling opportunities provided by backfilling. Therefore, the number of jobs per cluster should be determined based on the runtimes of the jobs and the availability of the remote resources.

5. Related work

There have been a number of efforts within the Grid community to develop general-purpose workflow management solutions.

WebFlow [29] is a multileveled system for high performance distributed computing. It consists of three layers. The top layer consists of a web based tool for visual programming and monitoring. It provides the user the ability to compose new applications with existing components using a drag and drop capability. The middle layer consists of distributed web flow server implemented using java extensions to httpd servers. The lower layer uses the Java CoG Kit to interface with the Grid [30] for high performance computing. Webflow uses GRAM as the interface between webflow and the Globus Toolkit. Thus, Webflow also provides a visual programming aid for the Globus toolkit.

GridFlow [31] has a two-tiered architecture with global Grid workflow management and local Grid sub workflow scheduling. GridAnt [32] uses the Ant [33] workflow processing engine. Nimrod-G [34] is a cost and deadline based resource management and scheduling system. The Accelerated Strategic Computing Initiative Grid [35] distributed resource manager includes a desktop submission tool, a workflow manager and a resource broker. In the ASCI Grid software components are registered so that the user can ask "run code X" and the system finds out an appropriate resource to run the code. Pegasus uses a similar concept of virtual data [2] where the user can ask "get Y" where Y is a data product and the system figures out how to compute Y. Almost all the systems mentioned above except GridFlow use the Globus Toolkit for resource discovery and job submission. The GridFlow project will apply the OGSA [36] standards and protocols when their system becomes more mature. Both ASCI Grid and Nimrod-G uses the Globus MDS service for resource discovery

and a similar interface is being developed for Pegasus. GridAnt, Nimrod-G and Pegasus use GRAM for remote job submission and GSI [37] for authentication purposes. GridAnt has predefined tasks for authentication, file transfer and job execution, while reusing the XML-based workflow specification implicitly included in ant, which also makes it possible to describe parallel and sequential executions.

The main difference between Pegasus and the above systems is that while most of the above system focus on resource brokerage and scheduling strategies, Pegasus uses the concept of virtual data and provenance to generate and reduce the workflows based on data products which have already been computed. It prunes the workflow based on the assumption that it is always more costly to compute the data product than to fetch it from an existing location. Pegasus also automates the job of replica selection so that the user does not have to specify the location of input data files. Pegasus can also map and schedule only portions of the workflow at a given time, using partitioning techniques. In combination with DAGMan, Pegasus can provide partition-level failure recovery capabilities.

6. Conclusions

In this paper we described the Pegasus framework, its ability to be customized to accommodate various scheduling and replica selection algorithms, and its ability to provide partition-level failure recovery. We evaluated the benefits of task clustering for an application with a relatively low computational granularity. This paper has shown experimental results of using Pegasus in the astronomy domain in the context of running on the TeraGrid. We selected this particular domain because it posses challenges to the workflow mapping system. The Montage workflows are typically large, often with hundreds and thousands of tasks and the tasks have a low computational granularity which exposes the overheads of the job submission and scheduling systems. Pegasus is currently being used in a number of other application domains including gravitational-wave physics [38], high-energy physics [1], biology [4], earthquake science and others [39]. The details about the various domains as well as additional details on Pegasus' functionality can be found in [4].

From the point-of-view of the user, Pegasus can run workflows across multiple heterogeneous resources distributed in the wide area, while at the same time shielding the user from the Grid details. From the point-of-view of performance, there are great benefits to the workflow and Pegasus approach to application description, mapping, and execution. The workflow exposes the structure of the application and its maximum parallelism. Pegasus can then take advantage of the structure to set the mapping horizon to adjust to the volatility of the target execution system. This feature is beneficial both in cases where resources or data may become suddenly unavailable and in cases where new resources come online. In the latter case, Pegasus can opportunistically take advantage of these newly available resources. The exposure of the maximum parallelism also enables Pegasus to cluster tasks together to reduce the overheads of target scheduling systems. Pegasus' workflow reduction capabilities can also improve overall workflow performance.

Pegasus is an evolving system. We are continuously improving the decision-making capabilities as well as developing algorithms for resource and replica selection, and task clustering. One direction that is of particular interest is resource reservation. Although it is not currently supported by many systems, as resource management technologies improve, the ability to reserve resources will become an important tool in not only improving performance of workflows but also in enabling new, time critical and/or interactive applications.

Acknowledgments

Pegasus is supported by NSF under grants ITR-0086044 (GriPhyN), ITR AST0122449 (NVO) and EAR-0122464 (SCEC/ITR). Montage is supported by the NASA Earth Sciences Technology Office Computing Technologies program, under Cooperative Agreement Notice NCC 5-6261. Part of this research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. Use of TeraGrid resources for the work in this paper was supported by the National Science Foundation under the following NSF programs: Partnerships for Advanced Computational Infrastructure, Distributed Terascale Facility (DTF), and Terascale Extensions: Enhancements to the Extensible Terascale Facility.

References

[1] E. Deelman et al., Mapping abstract complex workflows onto grid environments, *Journal of Grid Computing* **1** (2003), 25–39.

[2] E. Deelman et al., *GriPhyN and LIGO, Building a Virtual Data Grid for Gravitational Wave Scientists*, Proceedings of 11th Intl Symposium on High Performance Distributed Computing, 2002.
[3] G.B. Berriman et al., *Montage: A Grid Enabled Engine for Delivering Custom Science-Grade Mosaics On Demand*, Proceedings of SPIE Conference 5487: Astronomical Telescopes, 2004.
[4] E. Deelman et al., *Pegasus: Mapping Scientific Workflows onto the Grid*, Proceedings of 2nd EUROPEAN ACROSS GRIDS CONFERENCE, Nicosia, Cyprus, 2004.
[5] Southern California Earthquake Center (SCEC), 2004. http://www.scec.org/.
[6] P. Bruneman et al., *Why and where: A characterization of data provenance*, Proceedings of 8th International Conference on Database Theory, 2001.
[7] I. Foster et al., *Chimera: A Virtual Data System for Representing, Querying, and Automating Data Derivation*, Proceedings of Scientific and Statistical Database Management, 2002.
[8] J. Kim et al., *A Knowledge-Based Approach to Interactive Workflow Composition*, Proceedings of Workshop: Planning and Scheduling for Web and Grid Services at the 14th International Conference on Automatic Planning and Scheduling (ICAPS 04), Whistler, Canada, 2004.
[9] J. Kim et al., *An Intelligent Assistant for Interactive Workflow Composition*, Proceedings of 2004 International Conference on Intelligent User Interfaces (IUI-2004), Madeira Islands, Portugal, 2004.
[10] G. Singh et al., *A Metadata Catalog Service for Data Intensive Applications*, Proceedings of Supercomputing (SC), 2003.
[11] E. Deelman et al., *Grid-Based Metadata Services*, Proceedings of Statistical and Scientific Database Management (SSDBM), Santorini, Greece, 2004.
[12] D. Sundaram-Stukel and M.K. Vernon, *Predictive Analysis of a Wavefront Application Using LogGP*, Proceedings of 7th ACM SIGPLAN Symp. on Principles and Practices of Parallel Programming (PPoPP '99), Atlanta, GA, 1999.
[13] V. Taylor et al., *Using Kernel Couplings to Predict Parallel Application Performance*, Proceedings of 11th IEEE International Symposium on High-Performance Distributed Computing (HPDC 2002), Edinburgh, Scotland, 2002.
[14] V.S. Adve et al., POEMS: End-to-end performance design of large parallel adaptive computational systems, *IEEE Transactions on Software Engineering* **26** (2000), 1027–1048.
[15] I. Foster and C. Kesselman, *The Grid: Blueprint for a New Computing Infrastructure*, (2nd ed.), Morgan Kauffmann, 2004.
[16] E. Deelman et al., Workflow Management in GriPhyN, in: *Grid Resource Management*, J. Nabrzyski, J. Schopf and J. Weglarz, eds, Kluwer, 2003.
[17] "Globus". http://www.globus.org/.
[18] K. Czajkowski et al., *A Resource Management Architecture for Metacomputing Systems*, in 4th Workshop on Job Scheduling Strategies for Parallel Processing: Springer-Verlag, 1998, 62–82.
[19] W. Allcock et al., *Secure, Efficient Data Transport and Replica Management for High-Performance Data-Intensive Computing*, Proceedings of Mass Storage Conference, 2001.
[20] K. Czajkowski et al., *Grid Information Services for Distributed Resource Sharing*, Proceedings of 10th IEEE International Symposium on High Performance Distributed Computing, 2001.
[21] A. Chervenak et al., *Giggle: A Framework for Constructing Sclable Replica Location Services*, Proceedings of Proceedings of Supercomputing 2002 (SC2002), 2002.
[22] J. Frey et al., Condor-G: A computation management agent for multi-institutional grids, *Cluster Computing* **5** (2002), 237–246.
[23] E. Deelman et al., Transformation Catalog Design for GriPhyN, Technical Report GriPhyN-2001-17, 2001.
[24] MPI: A Message-Passing Interface Standard, May 1994.
[25] "Montage". http://montage.ipac.caltech.edu.
[26] R. Henderson and D. Tweten, *Portable Batch System: External Reference Specification*, 1996.
[27] S. Zhou, *LSF: Load Sharing in Large-Scale Heterogeneous Distributed Systems*, in Proc. Workshop on Cluster Computing, 1992.
[28] B. Bode et al., *The Portable Batch Scheduler and the Maui Scheduler on Linux Clusters*, Proceedings of 4th Annual Linux Showcase & Conference, Atlanta, 2000.
[29] E. Akarsu et al., *WebFlow – High-Level Programming Environment and Visual Authoring Toolkit for High Performance Distributed Computing*, 1998. http://www.supercomp.org/sc98/TechPapers/sc98 FullAbstracts/Akarsu809/Index.htm.
[30] G. v. Laszewski et al., A java commodity grid toolkit, *Concurrency: Practice and Experience* **13** (2001), 643–662.
[31] J. Cao et al., *GridFlow: WorkFlow Management for Grid Computing*, Proceedings of 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'03), 2003.
[32] G. v. Laszewski et al., *GridAnt – Client Side Grid Workflow Management with Ant*, 2003. http://www-unix.globus.org/cog/projects/gridant/gridant-whitepaper.pdf.
[33] "ANT." http://ant.apache.org.
[34] R. Buyya et al., *Nimrod-G: An Architecture for a Resource Management and Scheduling System in a Global Computational Grid*, Proceedings of HPC ASIA'2000, 2000.
[35] J. Beiriger et al., *Constructing the ASCI Grid*, Proceedings of Proc. 9th IEEE Symposium on High Performance Distributed Computing, 2000.
[36] "Globus Toolkit 3". http://www.globus.org/ogsa/.
[37] V. Welch et al., *Security for Grid Services*, Proceedings of Twelfth International Symposium on High Performance Distributed Computing (HPDC-12), 2003.
[38] E. Deelman et al., *Pegasus and the Pulsar Search: From Metadata to Execution on the Grid*, Proceedings of Applications Grid Workshop, PPAM 2003,, Czestochowa, Poland, 2003.
[39] "Pegasus". http://pegasus.isi.edu.