Distributed and Adaptive Execution of Condor DAGMan Workflows

Selim Kalayci[1], Gargi Dasgupta[2], Liana Fong[3], Onyeka Ezenwoye[4], S. Masoud Sadjadi[1]

[1]Florida International University, Miami, FL, USA,{skala001,sadjadi}@cs.fiu.edu
[2]IBM India Research Lab, New Delhi, India, gdasgupt,@in.ibm.com
[3]IBM Watson Research Center, Hawthorne, NY, USA, llfong@us.ibm.com
[4]South Dakota State University, Brookings, SD, USA, onyeka.ezenwoye@sdstate.edu

***Abstract*— Large-scale applications, in the form of workflows, may require the coordinated usage of resources spreading across multiple administrative domains. Scalable solutions need a decentralized approach to coordinate the execution of such workflows. At runtime, adjustments to the workflow execution plan may be required to meet Quality of Service objectives. In this paper, we provide a decentralized execution approach to large-scale workflows on different resource domains. We also provide a low overhead, decentralized runtime adaptation mechanism to improve the performance of the system. Our prototype implementation is based on standard Condor DAGMan workflow execution engine and does not require any modifications to Condor or its underlying system.**

Keywords: Application workflow, resource domain, execution, decentralization, distributed system.

I. INTRODUCTION

Large-scale applications, in the form of workflows, may require a coordinated usage of resources spreading across multiple administrative domains [3, 4]. Current systems mostly provide a centralized approach to the execution of such workflows across resource domains. This may induce scalability problems for large workflows executing in many domains. In addition, the execution may not be able take advantage of the flexibility of site autonomy. In our previous study [5], we provided a solution to decentralize the execution of workflows to address these issues. Another desired capability with the execution of large-scale workflows in dynamic distributed environments is the runtime steering of the workflow in order to meet the desired Quality of Service objectives. In this paper, we will focus on the runtime adaptation aspect on decentralized workflow execution.

Typically, scientific workflows are composed as abstract workflows, in which tasks and data are specified without bindings to the actual physical resources. Workflow management systems [2, 6, 7, 10, 11] perform the execution of abstract workflows in two basic steps. In the first step, tasks and data comprising the workflow are mapped on to the available physical resources using some method or heuristic [14, 15]. This results in a concrete workflow, which then can be enacted. The second step carries out the execution of the concrete workflow in compliance with the dependency relationships between tasks. But, the inherently dynamic characteristics of the resources, and the occasional unpredictable behavior of the workflow tasks, may require variations in the original workflow execution plan to meet the desired Quality of Service objectives.

Most of the existing proposals [8, 22, 2] tackle this problem by re-mapping the rest of the workflow, halting the execution of the original plan and enacting a new execution plan. This approach is very costly to perform, and also requires the interruption of the whole workflow execution even if the variations at the workflow execution plan does not require variations to execution plan of all the tasks. In this paper, we propose a new adaptation approach that is low-overhead, because it operates at the sub-workflow level and requires only the involvement of the affected tasks. Also, the variations to the execution plan do not interrupt the execution of the entire workflow.

Our prototype implementation focuses on adapting an existing workflow management system to make it more resilient to changes in its execution support. Hence, in this paper, we present an adaptation mechanism to a widely-used centralized workflow management system called Condor DAGMan [12]. Since Condor DAGMan is a very popular system, the adaptation mechanism we provide is transparent, which means that it does not require modifications to the Condor DAGMan infrastructure, so no changes is required to the existing installations. Our prototype accomplishes both the decentralized execution and the dynamic adaptation at runtime in a decentralized and non-intrusive manner.

The technical contributions detailed in this paper include:

- The novel concept of *adaptive sub-workflow* during runtime of workflow execution in peer domains.
- The design of *adaptive sub-workflow* run-time support with event notification across peer domains.
- The prototype implementation of *adaptive sub-workflow* as a transparent enhancement to a conventional workflow execution engine.

The rest of the paper is organized as follows. In Section 2, a background on workflows and our decentralization approach is provided. In Section 3, we present our adaptive workflow

execution approach. In Section 4, we provide implementation-level details. In Section 5, the related work is reviewed. In Section 6, the paper is concluded and some future work is discussed.

II. BACKGROUND

In this section, we briefly introduce the concept of scientific workflow management. We provide an overview of our decentralized design of workflow management, upon which our adaption of workflow execution is designed.

A. *Application workflow and execution*

Fig. 1 shows a very simple workflow represented as a **directed acyclic graph (DAG)** G = (V, E), where V = {V_1, V_2, …, V_8} is the set of vertices that correspond to the tasks and E = {E_1, E_2, …, E_{10}} is the set of edges that correspond to the dependencies between the tasks. Workflow managers execute the tasks in V in the order specified by the dependencies between them. For example, if there is an edge in the graph from vertex V_x to vertex V_y, V_y can start its execution only after V_x has finished its execution.

For scientific workflow applications, tasks in the graph need to map unto required resources (e.g, computation or data) before execution begins. For example, Pegasus system [6] provides the mapping functions as a planner for workflows and Condor DAGMan system executes the workflow using the required resources.

B. *Decentralized workflow*

Based on our approach for decentralized workflow execution [5], the home workflow manager (HWM) first decides on a mapping of workflow tasks to its peer domains. Based on the workflow in Fig. 1, a sample mapping for this workflow is shown in Fig. 2. Tasks are first assigned to domains, and then actual binding of tasks to specific resources is made autonomously by each domain.

1) *Mapping of workflow*

Graph partitioning techniques [17-20] were used in the literature to map the workflow tasks to peer domains. We used partitioning tools provided by the package called METIS [1], which includes multi-level partitioning tools for graphs and hyper-graphs. When partitioning the graph, basic workflow information such as structure, computation-communication costs associated with tasks and dependencies are taken into consideration. As the result of workflow partitioning, each task will be associated with one of the peer domains.

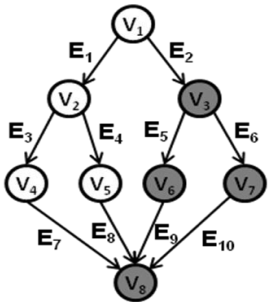

Figure 1. A Sample workflow specification represented as a DAG.

2) *Decentralized workflow execution*

When a workflow is submitted to a domain, it is assigned a HWM. After the mapping, the HWM extends the mapping information to the original workflow by associating each task with its corresponding execution domain information, including the corresponding end-point reference (EPR). With this information, each peer knows where each task is going to execute, and peers are able to communicate during execution. The HWM distributes this extended workflow definition to all the participating peers by using the operation:

sendWorkflow *(workflowID, workflowDef)*

The *workflowID* includes a unique identifier given by the HWM, and the workflow manager's EPR. The *workflowID* uniquely identifies a workflow instance. The *workflowDef* is the extended workflow specification. Each task in the original workflow specification is also identified with a unique ID as part of the workflow specification before the distribution.

Upon receiving the extended workflow, each workflow manager executes only the tasks that have been assigned to it. After a workflow manager executes a task, it checks whether the task has any dependent task(s). If a dependent task is assigned to a different peer, the workflow manager sends a trigger message to the responsible peer to start execution of the dependent task(s). Using Fig. 2 as an example, when task #1 is completed, the workflow manager in domain 1 sends a trigger message to the workflow manager in domain 2 to start execution of task #3. The signature of this operation is:

sendTrigger(*homeEPR,workflowID,taskID,status,data)*

The *homeEPR* and *workflowID* parts correctly identify the workflow instance. The *taskID* identifies the specific preceding task in the workflow being executed. The *status* part gives an indication of the status of the preceding task. A *done* status indicates that this task has executed successfully and the output data has been generated successfully. A *failed* status message would indicate to the next peer workflow manager that the preceding task did not complete successfully. The *data* in this message refers to the output data generated by the task, and can be a reference to a storage system or an RLS (replica locating service). The middleware for the underlying Grid infrastructure [17, 13] can be used to transfer data between peers.

Fig. 3 illustrates the execution control and peer interactions of the sample workflow, based on the mapping given in Fig. 2. Workflow Manager 1 (WFM1) is the HWM, and WFM2 is the peer workflow manager. There are 4 trigger operations made between peers.

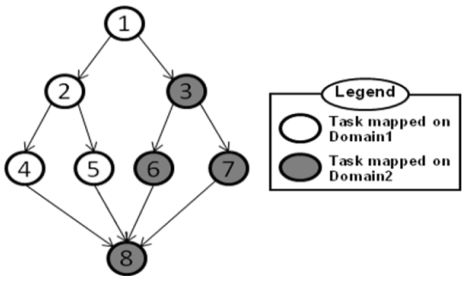

Figure 2. Sample workflow mapped on two domains. For simplicity, the labels for edges are dropped and vertices are merely numbered.

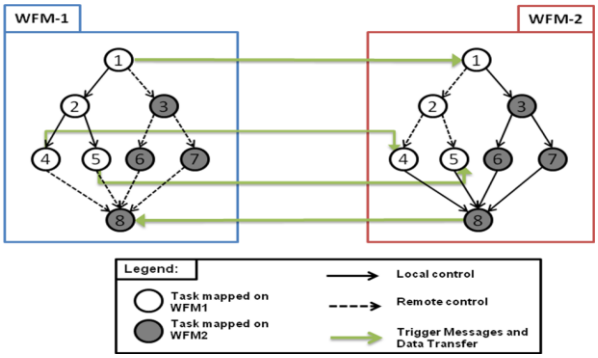

Figure 3. Execution controls and peer interactions during the execution of the sample workflow.

III. ADAPTIVE EXECUTION

Our adaptive workflow execution approach builds upon the distributed execution mechanism that is briefly explained in Section 2. So, the assumption in this section is that, a workflow has been mapped on multiple domains, and it is being executed in a peer-to-peer manner according to our decentralization mechanism. At runtime, the original workflow execution plan may need to be changed in order to meet Quality of Service objectives. Reasons for change may vary from inaccurate runtime predictions made during the mapping process to dynamic changes in computing environments due to resources being overloaded.

Fig. 4 shows the architectural diagram of our adaptation approach. Gray-colored components and artifacts denote those introduced by us. As our approach is based on peer-to-peer communication and collaboration of workflow managers (WFM), rather than coming up with a whole new mapping for the entire workflow, we let individual workflow managers detect any problems in their own domain that may prevent QoS objectives from being met. In such a case, tasks may then be migrated to other available domains for execution. The execution of the rest of the workflow is not affected by this adaptation. Only the peers directly involved within this adaptation process need to perform some additional steps.

A. Monitoring and Detection

Each workflow manager independently monitors its resource queues for normal operations, as well as detection of problems and troubleshooting. A backed-up resource queue is almost always an indication of some underlying problem. In this work, we consider *readyQ* that queues all the tasks that are *ready* for execution at the WFM (a ready task is one for which all preceding dependencies have been satisfied). Under normal operating conditions, tasks from the *readyQ* are steadily dispatched for execution to the underlying scheduler resources (i.e., computes nodes). A steadily building up *readyQ* signifies that there is some problem with the underlying infrastructure resources (e.g., resource outages, surge in background traffic, etc). It also almost always signifies that some QoS objective of response time or makespan will be compromised unless some corrective action is taken. We use *readyQ*-length as the indicator for the onset of congestion and proactively make runtime reconfigurations to deal with this.

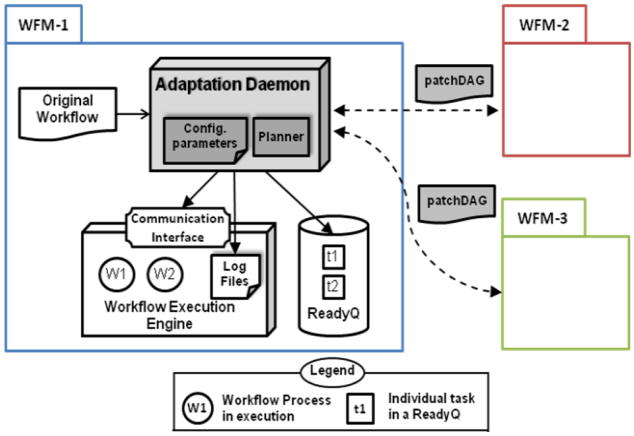

Figure 4. Architecture of the Distributed Adaptive Workflow Execution Approach

Detecting a congested queue: Each WFM continually monitors its *readyQ*-length. For every *readyQ*, we assign two thresholds *low* and *high*. The average queue length is calculated, using an exponential weighted moving average, given by:

$$avg_t = (1 - w)* avg_{t-1} + w * qlen_t,$$

where $qlen_t$ is the length of *readyQ* at time *t*, and *w* takes values between [0,1].

At time *t*, the following decisions are taken:

- if the queue length is below *low*, no corrective action is taken.
- if the queue length is between *low* and *high*, we probabilistically pick *some* tasks for remapping. The next subsection outlines how we select the tasks for remapping.
- if the queue length is above high, we pick *all* tasks for remapping.

B. Planning

Selecting jobs to move: A runtime reconfiguration in the original mapping involves the movement of yet-to-start computation or data management jobs to another domain. Our approach is based on RED [21], albeit accounting for data transfer involved in remapping the jobs. The probability, $p_j^{\wedge}$, of selecting a particular job *j* for moving depends on the average queue length, last time a job was moved from the queue, and also on data characteristics of the job.

Since the average length varies at a queue (belonging to site *i*) from *low* to *high*, the probability that a new compute job *j* is moved varies linearly from 0 to *P* (a small fraction). The final moving probability $p_j^{\wedge}$, however, also increases slowly with the number of jobs seen since last move (*count*), and decreases as the local data of *j* at *i* increases. We define the probability,

$$p_j^{\wedge} = \{p_b/(1 - count * p_b)\} * p_{ij}\text{ , where}$$

$$p_b = \{(avg_q - low_q)/(high_q - low_q)\}*P,$$

$$p_{ij} = (\Delta_{max} - \Delta_{i,j})/ (\Delta_{max} - \Delta_{min})$$

where Δ_{max},, Δ_{min} denote the maximum, minimum data requirements of *j*, respectively, and $\Delta_{i,j}$ is the size of local data

present for *j* at site *i*. Intuitively, this is the *stickiness* factor of a job to a site, with higher values indicate lower probability of movement.

Selecting a target domain to move to: Having selected a job to move, a target domain for the job needs to be identified. The target domain can be selected randomly from among the peer WFMs or be based on some characteristics of the peer domain. For instance, a job can be moved to an alternate domain that has the maximum link bandwidth, or processing capacity. A simple negotiation process occurs between the source domain and the target domain candidates. A candidate target domain accepts to receive the tasks being pushed only if it is able to sustain its ready*Q*-length below low threshold after the intended move.

C. *Execution: Adaptation Mechanism*

As mentioned earlier, our adaptation mechanism is based on a peer-to-peer push-based mechanism, without interrupting the execution of the rest of the workflow. In subsections A and B, we discussed methods for identifying the set of tasks to push, and the peer(s) to push those tasks to them. Here, we explain the mechanism for performing task migration and execution without interrupting the execution in-progress, while maintaining the integrity of the execution of the whole workflow.

When a set of tasks need to be moved from one domain to another, we capture those tasks and the associated trigger messages and create a new workflow, which we call a *patchDAG*. The patchDAG is a fragment of the whole that encapsulates the business logic of the tasks that are selected for migration. This patchDAG is transferred to the proper domain and executed independent of the original workflow. Even though it executes separately from the original workflow, by means of the trigger messages embedded within it, the patchDAG achieves the necessary interactions with the original workflow. This way, execution of the original workflow is sustained without making any changes to the rest of the workflow.

Fig. 5 shows a sample mapping scenario of a simple workflow that is to be executed in collaboration between WFM-1 and WFM-2 with our decentralized workflow execution approach. Consider a case where after the execution of task A in WFM-1, WFM-1 detects an overload on its resources and decides to push task B and task C to WFM-2.

Assuming that WFM-2 is able to receive those tasks, Fig. 6 shows the new outlook of the execution of the workflow after the adaptation. The patchDAG in this case includes task B, task C, and individual trigger messages from each of these tasks to synchronize with task F in the original workflow. This patchDAG is constructed at WFM-1 and transferred to WFM-2 for execution. WFM-2 starts the execution of this patchDAG independent from the original workflow, but as the tasks in the patchDAG finish and trigger messages are sent out, synchronization with the original workflow is achieved and task F in the original workflow can be scheduled for execution in WFM-2.

We will give details of how a patchDAG is constructed and we will show a sample patchDAG specification in Section 4.

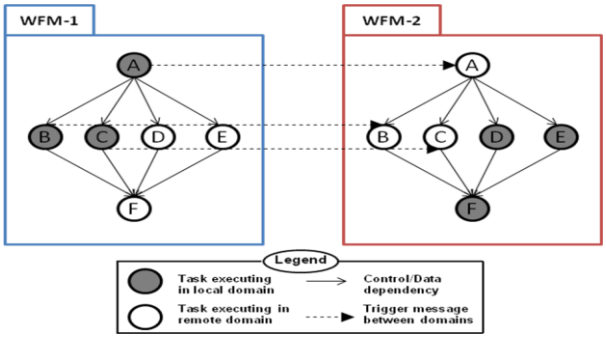

Figure 5. Before Adaptation

IV. IMPLEMENTATION

In this section, first we will introduce the implementation of decentralizing a centralized workflow specification. Then, we will explain how we achieve adaptation over decentralized execution. Our implementation is based on Condor DAGMan workflow execution engine, which is a meta-scheduler on top of Condor workload scheduler [9]. The systems we have tested our prototype implementation had standard Condor versions of 6.7, on which and further versions.

A. *Decentralization*

Let us consider the sample mapping of a DAG spanning across multiple domains as in Fig. 5. The standard centralized Condor DAGMan specification on WFM-1 is provided in Listing 1. In this case, the whole workflow is orchestrated from the Condor DAGMan instance at WFM-1. Cross-domain task submissions (e.g., task D) would be facilitated through the employment of Condor-G [16] and those tasks would be constructed accordingly. The details pertaining the individual task construction and modification are outside the scope of this paper.

The decentralization of an original DAG specification for our Condor DAGMan-based prototype occurs in two stages. In the first stage, HWM (in this case, WFM-1) aggregates the original DAG specification with mapping and site-specific contact information. Mapping information denotes the site that is responsible for each task's execution. Site-specific contact information includes the GRAM (Grid Resource Allocation and Management) [3] end-point reference to facilitate trigger message communication, and the GridFTP [17] server and URL information to facilitate the transfer of data items among sites. HWM distributes this aggregated DAG specification to all the peer WFMs involved in the execution of the workflow.

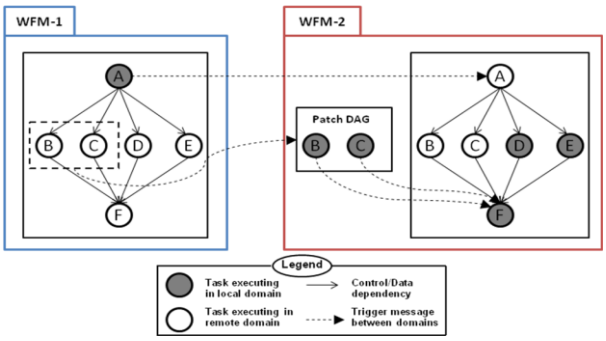

Figure 6. After Adaptation

```
Job     A     A.submit
Job     B     B.submit
Job     C     C.submit
Job     D     D.submit
Job     E     E.submit
Job     F     F.submit

PARENT  A   CHILD  B C D E
PARENT  B C D E  CHILD   F
```

Listing 1. Original DAG specification for the DAG given in Figure 5.

The second stage of decentralization process occurs at individual sites upon receiving the aggregated DAG specification. Each WFM modifies its own copy of the DAG specification based on the information received from the HWM. Listing 2 and Listing 3 displays the DAG specifications at WFM-1 and WFM-2 respectively, after the modification. There are 3 basic functionalities involved within this modification process:

1. If a task is mapped on a different domain, this task is labeled as *DONE* in the DAG specification. (This will cause local Condor DAGMan instance to skip this task during the execution.)
2. If a task has child task(s) that is mapped on a different domain, insert a *Post Script* in the DAG specification to synchronize with the child task(s).
3. If a task has parent task(s) that is mapped on a different domain, insert a *Pre Script* in the DAG specification for the reception of a matching synchronization message.

Our Post Script prototype simply generates a file that is specifically named (e.g., file: A_D to trigger task D at WFM-2) and puts it in the shared folder of the receiving end. The Pre Script simply checks the existence of the specifically named file in the shared folder.

Hence, the final set of DAG specifications at each site differs, but the collaborative execution of these DAGs results in the complete and exact execution of the original DAG specification.

```
Job     A     A.submit
Job     B     B.submit
Job     C     C.submit
Job     D     D.submit    DONE
Job     E     E.submit    DONE
Job     F     F.submit    DONE

SCRIPT POST  A   trigger.sh   A_D A_E
SCRIPT POST  B   trigger.sh   B_F
SCRIPT POST  C   trigger.sh   C_F
…
```

Listing 2. Modified DAG specification for WFM-1.

```
Job     A     A.submit     DONE
Job     B     B.submit     DONE
Job     C     C.submit     DONE
Job     D     D.submit
Job     E     E.submit
Job     F     F.submit

SCRIPT PRE  D   synch.sh   A_D
SCRIPT PRE  E   synch.sh   A_E
SCRIPT PRE  F   synch.sh   B_F  C_F
…
```

Listing 3. Modified DAG specification for WFM-2.

B. Run-time Adaptation

As mentioned in Section 3, we provide run-time adaptation through an adaptation daemon running at each site. The adaptation daemon constantly monitors the execution progress of the workflow and the length of ready queue. It also has a thread that listens for any push requests that might come from other peers.

The specific functionality of an adaptation daemon in case of an adaptation process varies depending on its role. The algorithm employed at the adaptation daemon for the pushing role is shown in Listing 4. After the task(s) to push is chosen, and a site to push the patchDAG is picked successfully, the patchDAG is sent to the other site. Then, the pusher site needs to perform some additional operations at the local site. First, it needs to remove those pushed tasks from the local Condor queue (step 5). But, Condor DAGMan requires all tasks to complete successfully for the whole DAG to succeed. Therefore, step 5 will cause the local DAG execution to fail eventually (it fails when the workflow execution cannot proceed any further due to dependencies). When a DAG fails execution, Condor DAGMan automatically generates a rescue DAG (step 6). Adaptation daemon monitors and detects the generation of this rescue DAG. Then, it adapts this rescue DAG to conform to the new mapping scenario. First, it labels those tasks that are pushed to other sites as "DONE" in the rescue DAG (step 7.a). Also, if there are any tasks that are locally mapped and have at least one parent task among the pushed tasks, a Pre Script is inserted into the rescue DAG specification to achieve synchronization (step 7.b.) Then, this modified rescue DAG is submitted again to the local Condor DAGMan engine.

Listing 5 shows the algorithm employed at the adaptation daemon for the receiving role. The only operation that the adaptation daemon needs to perform in this case is to receive the patchDAG and submit it to the local Condor DAGMan engine.

Listing 6 shows the sample specification for the patchDAG illustrated in Fig. 6. In this specification, the only different item than the previous specifications is the transferData script. There is a transferData Pre Script before both task B and task C, and it performs the transfer of any input data required for the execution of these tasks at its new location.

1. Detect the need for adaptation
2. Select the task(s) to push
3. Pick a site among candidates
    a. if negotiation is successful, move to step 4
    b. if negotiation is unsuccessful, pick another site, repeat step 3
    c. if tried all the sites, abort the adaptation
4. Construct the patchDAG and send it to the receiver site
5. Remove pushed tasks from the local queue (*condor_rm taskID*)
6. Wait for a rescue DAG to be created by Condor DAGMan
7. Modify the rescue DAG
    a. label the pushed tasks as DONE
    b. insert a Pre Script for the task(s) which is mapped locally and is a child of a pushed task
8. Submit the modified rescue DAG to Condor DAGMan.

Listing 4. Pushing algorithm employed within the Adaptation Daemon.

1. Receive the request to push tasks
2. Check the status of the local queue and respond to the request
    a. if response is negative, go back to step 1
    b. if response is positive, go to step 3
3. Receive the patchDAG
4. Submit the patchDAG to the local Condor DAGMan

Listing 5. Receiving algorithm employed within the Adaptation Daemon.

```
Job     B     B.submit
Job     C     C.submit

SCRIPT PRE  B   transferData.sh
SCRIPT PRE  C   transferData.sh
SCRIPT POST  B   trigger.sh   B_F
SCRIPT POST  C   trigger.sh   C_F
```

Listing 6. PatchDAG specification for the scenario given in Figure 6.

V. RELATED WORK

Pegasus [6] and ASKALON [11] are two significant workflow management systems for the grid environment. Pegasus receives an incoming workflow and then partitions it into a collection of sub-workflows. Pegasus uses the Condor DAGMan [12] as the workflow execution engine. A single central DAGMan instance works as a meta-scheduler to submit and monitor jobs on local and remote domains of a grid. This is a centralized model of workflow execution. ASKALON has its own design of the workflow engines. For each workflow, the execution enactment consists of a master and many slave engine for sub-workflows. In other words, ASKALON supports a distributed model of workflow execution. Unlike Pegasus and ASKALON, the workflow system we propose for a grid environment consists of one or more peer domain workflow managers [5].

Partitioning an incoming workflow for execution can be multiple phases and multiple iterations with a phase in ASKALON [10], described as its initial partition phase and run-time optimization phase. Re-partitioning of a workflow to achieve run-time optimization phase can have one or more iterations. ASKALON re-partition is done by the master engine for the workflow. The approach described in this paper is different from ASKALON as each peer workflow manager is responsible for a sub-workflow repartitioning.

Many studies examine the efficiency and effectiveness of run-time workflow rescheduling algorithm. The studies presented in [22, 2] propose modifications to the initial workflow schedule at run-time via some modifications to existing workflow scheduling algorithms. The closest study [8] to the work presented in this paper is provided as an adaptation mechanism on top of Pegasus workflow planner [6]. In this study, the mapping of a workflow executing on top of Condor DAGMan workflow execution engine is monitored periodically, and if there is a substantial increase or decrease in queue wait times per site, Pegasus workflow planner tools are called to perform a new mapping. After an improved mapping is generated, execution of the current workflow is stopped, and the workflow is deployed again according to the new mapping. All the phases (monitoring, analysis, planning, and execution) involved in this adaptation process are performed in a centralized manner, and this is the main difference between this work and our approach. Also, our approach does not necessitate the halting of the current execution of the workflow; hence it is expected to be lower-cost and less intrusive under normal conditions.

VI. CONCLUSION

In this paper, we addressed the scalability and adaptation problems of large-scale workflows in distributed environments. Contrary to many existing solutions, our decentralized approach has a low performance overhead and is transparent. Our push-based mechanism does not require the use of any other WFMs other than those that are directly involved with the adaptation process. Also, our devised patchDAG-based solution does not require the halting and re-deployment of a workflow at runtime, but rather, acts as a bridge between the remapped portions of the workflow. Our prototype implementation on Condor DAGMan shows the feasibility and transparency of our approach.

Our future studies include the enhancement of the adaptation with a *pull-based* mechanism. With such as mechanism, a lightly-loaded WFM can opportunistically pull some of the tasks from relatively heavy-loaded sites. We also would like to explore the application of our approach to other existing workflow management systems.

ACKNOWLEDGMENT

This work was supported in part by the NSF (grants OISE-0730065, OCI-0636031, and HRD-0833093), and in part by IBM.

REFERENCES

[1] Karypis, G and Kumar, V., "Multilevel k-way Partitioning Scheme for Irregular Graphs." J. Parallel Distrib. Comput. , 1998, Issue 1, Vol. 48, pp. 96-129.

[2] Sakellariou, R. and Zhao, H. 2004. “A low-cost rescheduling policy for efficient mapping of workflows on grid systems.” *Sci. Program.* 12, 4 (Dec. 2004), 253-262.

[3] *Globus Toolkit.* [Online] http://www.globus.org/toolkit/.

[4] *EGEE.* [Online] http://www.eu-egee.org/.

[5] Kalayci, S. et al. “A Peer-to-Peer Workflow Mapping and Execution Framework for Grid Environments.” Technical Report FIU-SCIS-2010-01-01, Florida International University.

[6] Deelman, E, et al., "Pegasus: A framework for mapping complex scientific workflows onto distributed systems." s.l. : Scientific Programming, 2005, Issue 3, Vol. 13.

[7] Zhao, Y., Hategan, M., et al. "Swift: Fast, Reliable, Loosely Coupled Parallel Computation." Salt Lake City, UT, 2007. 2007 IEEE Congress on Services.

[8] K. Lee, N.W. Paton, R. Sakellariou, E. Deelman, A. A. A. Fernandes, and G.Mehta. “Adaptive Workflow Processing and Execution in Pegasus.” In *3rd International Workshop on Workflow* Management and Applications in Grid Environments (WaGe08) (in Proceedings of the Third International Conference on Grid and Pervasive Computing Symposia/Workshops, May 25-28 2008, Kunming, China), 2008, pp. 99-106.

[9] M. Litzkow, M. Livny, and M. Mutka, "Condor - a hunter of idle workstations" in *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[10] Duan, R and Prodan, R. Fahringer, T., "Run-time Optimisation of Grid Workflow Applications." 2006. Grid Computing, 7th IEEE/ACM International Conference on Grid. Barcelon, Spain, 2006.

[11] Wieczorek, M, Prodan, R and Fahringer, T., "Scheduling of scientific workflows in the ASKALON Grid environment." s.l. : ACM, 2005, Issue 3, Vol. 34.

[12] Condor team, "The directed acyclic graph manager", www.cs.wisc.edu/condor/dagmang, 2002.

[13] Allcock B., Foster I., Madduri R., "Reliable data transport: a critical service for the grid", In Building service based grids workshop, Global Grid Forum 11, June 2004.

[14] Topcuouglu, H, Hariri, S and Wu, M., "Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing." IEEE Transactions on Parallel and Distributed Systems, IEEE Press, 2002, Issue 3, Vol. 13.

[15] Yu, J and Buyya, R., *Workflow Schdeduling Algorithms for Grid Computing.* Grid Computing and Distributed Systems Laboratory, The University of Melbourne. s.l. : GRIDS-TR-2007-10, 2007.

[16] Frey, J, et al., "Condor-G: A Computation Management Agent for Multi-Institutional Grids." Kluwer Academic Publishers, 2002, Issue 3, Vol. 5.

[17] Allcock B., et al., "Secure, efficient data transport and replica management for high-performance data-intensive computing." In IEEE Mass Storage Conference, San Diego, CA, April 2001.

[18] Hendrickson, B and Kolda, T.G., "Graph partitioning models for parallel computing." Parallel Computing, s.l. : Elsevier Science Publishers B. V., 2000, Issue 12, Vol. 26.

[19] Simon, Horst D., "Partitioning of unstructured problems for parallel processing." Computing Systems in Engineering, 1991, Vol. 2..

[20] Kernighan, B and Lin, S., "An Efficient Heuristic Procedure for Partitioning Graphs." s.l. : Bell Systems Technical Journal, 1970, Issue 2, Vol. 49.

[21] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Trans. Netw.*, 1(4), 1993.

[22] Yu Z, Shi W. "An adaptive rescheduling strategy for grid workflow applications." *IPDPS*, IEEE Press, 2007; 1–8.