International Conference on Computational Science, ICCS 2010

Integrating scheduling policies into workflow engines

G. Martinez, E. Heymann, M. Senar

Departament d’Arquitectura de Computadors i Sistemes Operatius
Universitat Autònoma de Barcelona
Barcelona, Spain

Abstract

Workflow applications running on distributed environments are a promising solution for resource and computing intensive problems. However, the heterogeneity of resources in these kind of environments may turn scheduling of such applications into a complicated enterprise. Although there is research in sophisticated scheduling policies for workflows they are of little actual use, as they must be ported or implemented into each and every workflow engine they intend to support. The problem becomes an $m \times n$ effort (for $m$ policies and $n$ workflow engines). In response to such a problem we present schedflow, a system that provides a simple interface between existing workflow engines and scheduling policies. We conducted experiments that demonstrate schedflow’s usefulness when confronted with different workflow engines and their default scheduling policies.

Keywords: scheduling policies, workflow management, distributed environments.

1. Introduction

Workflow management is an important field of research in Grid computing. A workflow application is a collection of jobs to be executed in a partial order determined by control and data dependencies. A directed acyclic graph (dag) is a simple model that can be used for the representations of workflows. For executing this type of application two components are required: a workflow scheduling policy for determining on which resources tasks belonging to a workflow are to be executed, and a workflow engine for ensuring the correct execution order of these tasks.

On the one hand, several workflow engines can currently be found, including Condor DAGMan [1] [2], Taverna [3] [4] Triana [5], Karajan [6] and Pegasus [7], for supporting the execution of workflow applications on clusters and Grid systems. On the other hand, significant effort has been put into developing scheduling policies for workflows, such as heterogeneous earliest finish time (HEFT) [8] [9], balanced minimum completion time (BMTC) [10], Min-Min [11] and DAGmap [12]. Nevertheless, little attention has been paid to linking workflow scheduling policies with existing workflow engines.

Although scheduling policies may work well, most studies are theoretical, performed through simulation and their impact is not significant when applying them practically, since it is very complicated to include them in existing...
workflow engines. This complexity leads to modifications in the workflow engine architecture in order to implement a different scheduling policy than the one provided by default in the workflow engine.

In this paper, we present schedflow, a framework for transparently integrating scheduling policies on different workflow engines. We draw an analogy between schedflow and MAUI [13], where MAUIs function is acting as a meta-scheduler for sequential and parallel applications that can transparently link to different queuing systems.

Its usefulness comes from the ability the user has to schedule its application with no regard for the underlying local queuing system. schedflow extends MAUIs philosophy to the world of workflow applications. This framework introduces the following capabilities to the user:

1. Allows scheduling developers to test policies in real environments.
2. Turns policy selection into a reality for the end user.

Experimentation shows improvements in makespan when a workflow application is run using schedflow, compared to the case where the default scheduling policy for each engine is used. We ran the Montage application using different workflow engines (Condor DAGMan, Taverna and Karajan). The scheduling policies integrated were Random, Min-Min, HEFT and BMTC. These results also demonstrate the flexibility of our system to integrate different scheduling policies and link them to different workflow engines.

The remainder of this paper is organized as follows. Section 2 presents the related work. Section 3 describes the framework architecture and their interfaces. Section 4 explains the experimentation performed and results obtained. Finally, Section 5 concludes our work.

2. Related Work

In this section, we mention different alternatives for running workflow applications. We also describe their functional characteristics, and their default scheduling policies.

Condor DAGMan [2] is a meta-scheduler for Condor. It manages job dependencies at a higher level than the Condor scheduler. DAGMan uses a random scheduling policy by default, and its architecture does not provide methods for integrating new scheduling policies. DAGMan has two different mechanisms for fault-tolerance: (a) tasks retry: if a task fails for any reason, DAGMan runs it again, and (b) task migration: it allows the user to manually schedule a task to run somewhere else.

In Taverna [4], the workflow manager allows users to build complex analysis workflows from components located on both remote and local machines, run these workflows with their own data, and visualize the results. The scheduling policy is based on performance models (CPU speed), which means that tasks are sent to the machines with better performance. Its scheduling policy is tied to the processors speed. There is not mechanism for adding new scheduling policies. Taverna has a centralized retry mechanism for fault tolerance.

Karajan [6] is an extensible workflow framework derived from GridAnt [14] which provides additional capabilities such as workflow structure and error handling. Additionally, it supports choices and loops of workflow structures. The default scheduler cycles through a list of available resources and uses the first resource suitable for the given task. The resource search for the next task begins with the resource immediately following the last resource used in the list. If the end of the list is reached, the search continues from the beginning of the list. Its fault-tolerance scheme is based on task retries in alternating resources. Karajan does not support the integration of new scheduling policies.

Triana engine [5] provides a service capable of executing complete or partial tasks and graphs locally, or by distributing the code to other servers based on the specified distribution policy for the supplied task-graph. Triana is also designed in a just-in-time scheduling schema, and uses the random default policy. It does not support the integration of new scheduling policies, and its task migration scheme is operated manually.

A complete taxonomy and classification according to the main function and architecture of workflow systems is described in [7].

The aforementioned workflow engines have a lack of flexibility when the user wants to execute a workflow application with a scheduling policy different to the one provided by default.

WenGrid [15] is a Grid infrastructure based on distributed, autonomic self-organizing workflow engines, called Ws-engines, for processing workflow-based applications. WenGrid provides a scheduler service which is responsible
for job scheduling and dispatching jobs using a first-in-first-out policy for selecting the next job to be processed from the jobqueue.

This system is a first step in the research of linking different workflow engines. Its main disadvantage is that it only supports workflow engines with web interfaces and it does not accept different scheduling policies. It does not provide any fault tolerance mechanism either.

As it can be seen from the characteristics of the workflow engines mentioned, they share a common point: none of those support the integration of different scheduling policies than those built-in.

2.1. Workflow Engine and Policies Requirements

A workflow management system should deal with five issues [16]: workflow design, information retrieval, workflow scheduling, fault tolerance and data movement. Every aforementioned system has those components, so they are able to run workflow applications.

However, what happens when a user designs a new scheduling policy and wants to test it over those systems? The simple answer is that the user cannot do it. Actually it is easier to design a simulator and test the designed scheduling policy over it, as can be seen throughout the literature. So, why not design a system able to link scheduling policies and workflow engines, solving this problem for the user?

Therefore, it is important to know the needs of scheduling policies on workflow applications. Traditionally those policies refer to three basic services: task management, resource management, and mapping. We now proceed to detail each of them.

When a scheduling policy manages a task, it gathers information related to this task, such as dependencies, precedence computation time and communication time, in order to schedule it.

The policy also manages resource-related data: it gathers information about the execution environment, such as available architectures, performance, benchmark results and so on. This data helps the policy to better schedule the tasks.

When the scheduling policy has the data regarding both tasks and environment, it proceeds to the third service: map the tasks to the resources, according the criteria inherent from the algorithm. It is worth noting that no matter how complex a policy is, the basic services will always be the same.

Once having analyzed the functionalities of scheduling policies and describe the features of the current workflow engines, our system is designed to provide the much needed link between these two worlds. For one side, we transparently integrate scheduling policies. On the other, we developed interfaces for linking it with different workflow engines.

3. Framework Architecture

This section describes the architecture and available interfaces in our system. We also explain how users can integrate new scheduling policies without changing the workflow engine architecture. fig. 1 shows the three modules (Controller, Observer and Scheduler), and the two interfaces Scheduling and Workflow Engine) that allow integrating scheduling policies with workflow engines.

Controller: this module is responsible for storing information of the tasks in the task-list, and sending these tasks to the Scheduler module for scheduling. This module remains active waiting for the Scheduler module to send the result of mapping the tasks to the machines. Once received, task mappings are sent to the workflow engine to be executed.

Observer: this module is our resource and event manager. It is responsible for obtaining and managing the resource list used by the Scheduler module to schedule tasks. Additionally, it also monitors the events that affect tasks being executed, and informs the Controller module about tasks that have finished correctly. With this information another task is sent to the workflow engine if the scheduling is static or to the Scheduler if the scheduling is dynamic. If there is a failure in the execution of a task, the Observer module will signal the Scheduler to remap the whole failed tasks sub-tree in the static case, or only remap the failed tasks in the dynamic case.

Scheduler: this module is responsible for mapping tasks into machines. As shown in Fig. 1, the Scheduler module interacts with the user policy throughout our API. This interface returns a list of mapped tasks according to the scheduling policy integrated by the user to run the workflow. This mapping is sent to the Controller module so it can send the mapped tasks to the workflow engine to be run.
3.1. Engine Interface

Schedflow also provides an interface that allows the connection of different workflows engines such as dagman, taverna, or karajan. This interface includes a series of adaptors that converts the information into a format suitable for our system for scheduling. Once the tasks have been scheduled, another adaptor performs the conversion to a format suitable for the workflow engine to operate on it. The functions comprised in the adaptors are related to the services required by our system. fig. 2 shows the basic services used by schedflow to connect to any workflow engine.

Our system consists of three adaptors, responsible for the conversion of data between schedflow and the workflow engine being used at a given time, as shown in Figure 2. They are the task management adaptor, the resource management adaptor and the event management adaptor.

1. Task Management: This adaptor is necessary for the Controller module. It sends out the tasks mapped to the corresponding workflow engine.
   (a) For DAGMan we use the condor_submit command.
   (b) For Taverna, the job_submission command is used.
   (c) For Karajan the gridExecute command is used.
2. Resource Management: This adaptor is necessary for the Observer module in order to obtain the resources available from the execution environment as seen by each workflow engine.
(a) For DAGMan, the condor_status command is used.
(b) For Taverna, the g_resource command is used.
(c) For Karajan, we use the Rhost command.

3. Event Management: This adaptor is used by the Observer and Scheduler modules to perform the rescheduling of tasks. Rescheduling is performed when a predefined event occurs in our system. As of this moment, schedflow handles two kinds of events: fail and suspend. The former happens when the resource suffers a failure, while the latter happens when the task is put on hold by the execution environment due to the existence of a higher priority task. These events are detected through information provided by each workflow engine.
(a) For DAGMan events are obtained from the information log indicating the task status.
(b) For Taverna, the logbook-data content allows us to know the task status.
(c) For Karajan, we use the mapget function that provides the task status at runtime.

3.2. Scheduling Interface

SchedFlow provides users with an API that allows the integration of scheduling policies for workflows without the need to modify the workflow engine. This interface integrates the scheduling policy as a dynamic library, so that when scheduler module needs to map a task interacts with it. Users have to implement their scheduling policies algorithms as a C++ function that will call the appropriate methods from our API.

When a workflow is submitted to schedflow, translates the workflow into an internal structure that contains the workflow tasks and their dependencies. Similarly, available resources, as seen by the workflow engine, are obtained by means of the get_resource() function, which will translate the information provided by each workflow engine into an internal format.

In order to a run-time matching and scheduling policy to make the mapping either statically or dynamically, an accurate set of estimates of the execution time of the task on each potential machine is needed, as well as an accurate estimate of the communication time incurred between each pair of connected tasks in the workflow.

It is well known that the execution time of a task is a function of the size and properties of the input data and communication times depend on the volume of data transferred. In case of all machines very homogeneous, it can be assumed that each particular task performs identically on each target machine. Therefore, a single estimate of the execution time of each task is required, and this is fairly easy to obtain.

This, however, is not true for heterogeneous systems since an execution time estimate is required for each task-machine pair, and there are many factors unique to heterogeneous systems which can affect the execution time.

Unfortunately, current workflow engines do not provide such estimates (only some synthetic performance information is provided for available machines). Therefore, execution and communication time estimates must be computed by external mechanisms.

Our system includes two functions that can be used to include estimates of execution and communication times (get_comp_t() and get_comm_t(), respectively). Those functions return the computation and communication time from different computing resources of the execution environment. These times are obtained by means of a history of executions, where their average value is assigned as the tasks initial value for each task. This history is update every time this workflow is run. In the case this task was never run before, we run it into a local resource first to obtain this time. It is worth noting that the user needs no external tools to perform those estimations.

SchedFlow currently includes a simple mechanism based on historical information from past executions and we are currently working on the integration of a more sophisticated method based on nonparametric regression techniques [17].

There is no common notation to represent application workflows. Each workflow engine has its particular notation, but schedflow is capable of reading the original workflow through the corresponding adaptor (by using the set_workflow() function). Alg. 1 shows a simplified pseudo-code that illustrates the HEFT algorithm implemented with our API functions. In any case, our system loads the user-chosen scheduling policy as a dynamic library every time it is necessary to map some workflow tasks.

A complete and detailed description of all the methods included in our API cannot be included here due to space limitations. However, fig. 3 shows a summary of the main functions, including a brief description, arguments passed to the functions, and returned values as well. Additionally our system supports the Fail and Suspend events. We now explain each of them:
1. Suspended Tasks: a task sent to a computing resource may have its priority lowered or be suspended due to other processes executed directly on the same machine (local load), and the task is suspended for a random period of time. The consequence is a delay in the conclusion of the whole application. SchedFlow uses the Observer module to verify if this time is no bigger than this tasks estimated execution time. When this event occurs, our system removes this task from the current resource and the task is rescheduled.

2. Task Failure: a task that was already sent to a machine and is running might fail, due to different reasons, e.g. it was evicted from the system due to a machine or network failure. schedflow is not aware of the reason of failure instead, it just reschedules the task to a new available resource. The difference between these events lies in their detection. However, their solution is similar.

Our main efforts up to now have focused on the functionality of schedflow. However, with the existing components and API it would be very easy to integrate other mechanisms that could provide more accurate estimates of execution and communication times, which will result in better scheduling decisions.

```
1. schedflow::set_workflow(file, int)
2. schedflow::get_resource (void)
3. Compute nodes weight of the DAG.
4. Compute rank for all nodes by traversing.
5. Sort the nodes in a task_list by nonincreasing order of rank value, assigns to each node of the DAG an identifier (task_id).
6. While(there are unscheduled nodes in the task_list){
   6.1 Select the first task in the task_list and remove it.
   6.2 Find the machine_id that minimize the EFT value of task_id.
   6.3 schedflow::map(task_id, machine_id)}
*The event management*
1. If(any event is detected during execution){
   1.1schedflow::unmap(task_id)
2.1. Foreach failed node until the last node in the DAG.
   2.1.1 schedflow::get_resource(void)
   2.1.2 schedflow::map(task_id, machine_id)}
```

Algorithm 1. HEFT algorithm with the proposed API.

**4. Experimental Desing & Results**

This section we show some experiments and results obtained with our system. We ran the Montage [18] application with different scheduling policies, over different workflow engines. In our experiments we used the following scheduling policies: Random, Min-Min, Heterogeneous Earliest Finish Time (HEFT), and Balanced Minimum Completion Time (BMTC).The results obtained with scheduling policies proceeded by the different workflow engines (Condor DAGMan, Taverna, and Karajan).

**4.1. Experimental Environment and Aplication**

Our experiments were carried out on an opportunistic and non-dedicated environment, composed of 140 Intel-based computing nodes executing Linux Fedora Core 5, 768Mb of RAM and running Condor as a local resource management system. According to the data benchmark provided by Condor, machine performance in this environment ranged from 0.25 to 0.75 GFlops.

Montage is a toolkit for assembling flexible image transport system (FITS) images into custom mosaics. The Montage workflow application is divided into levels ( the levels 1, 2 and 5 have 12, 23 and 12 nodes respectively, while other levels have only one node. In order to execute this application it is necessary to include the input images files in FITS format (this is the standard format used by the astronomical community), while a head file should also be included that specifies the mosaic type that is to be built.
This workflow application operates in three steps. Firstly, the re-projection of the input images, second, the refinement of the re-projected images, and finally, the superimposition of the re-projected images and their refinement to obtain the mosaic in jpeg format.

4.2. Experimentation and Results

We carried out four sets of experiments to test schedflow using the execution environment described above. The experiments carried out were intended to illustrate schedflows capabilities as a tool for integrating different scheduling policies over different workflow engines.

Furthermore, these experiments show also some issues that constitute potential ways for developing new scheduling policies that provide good performance on real heterogeneous environments, in which many runtime events may affect significantly the overall execution time of a given workflow.

In the first scenario, we executed the Montage application integrating the HEFT, BMTC, and Min-Min scheduling policies with Condor DAGMan workflow engine. The Montage application was mapped once in a static way at the before beginning of the execution and it was later run in ideal conditions, where no events, such as failure or suspensions of tasks ever occur. The results are shown in 4(a), where in the x-axis are the different scheduling policies used, while the y-axis is the average (from 50 executions) execution time (makespan) in seconds. The makespan is computed as the time when the application was initially submitted to schedflow, until the last node of the application finishes.

In our second scenario, we used the same scheduling policies, but we injected three tasks with higher priority on random resources. As a consequence, some of the tasks were suspended temporarily (according to the policies applied by Condor), which affected the overall execution of the whole workflow. This is a very usual situation that usually happens in a non-dedicated environment.

Unfortunately, our three scheduling policies were applied statically and no correction action was taken. Suspended tasks were retried automatically by Condor on the same machine because the original mapping was not modified. The results of those experiments are shown in 4(b) Not surprisingly, all scheduling policies obtained worse execution times compared to the previous case.

These results obtained, when suspensions do not occur (first scenario), we can see that the used scheduling policies that take into account more factors will reduce the makespan. This shows us that if we have a tool such as the proposed one, it will help the end-user to select the policy which performs best performance of his application.
The result obtained in the second scenario, the effects of suspensions affected differently each scheduling policy. As a consequence, a very simple strategy such as min-min which, in contrast to the other two, does not take into account any information about the workflows critical path, achieved execution times very similar to BMTC.

HEFT was significantly affected by run-time events that modified the execution estimates used for the mapping process (up to 26% deviation were observed in some worst cases). This has to do with HEFT being static. This means that when a computing node fails, HEFT reschedules all the remaining tasks.

In the third scenario, we used the same scheduling, and the aforementioned task injection mechanism. However, we introduced some slight variations on the scheduling techniques and they were augmented with an event management function that was invoked for dealing with the occurrence of events, such as suspensions or failures. Under the occurrence of suspensions at run-time, schedflow called the event management function and the corresponding scheduling policy was applied to remap the suspended task and, eventually, some of the other tasks that were still waiting to be ready. The results are shown in 4(c).

The results observed in the third scenario were satisfactory, and demonstrate that the inclusion of a dynamic rescheduling mechanism that reacts in case on dynamic events is beneficial to all strategies. The average makespan is very close to that of the ideal case.

In most cases the difference was less than a 5% This is a good result if we consider the fact that each suspension required a whole execution of each strategy to reschedule all the tasks that were still remaining. It is important to point out that this extra overhead was small also because the number of suspensions was only three. In a different scenario where more suspensions of failures might occur, we expect that higher overheads will be obtained due to the high computational complexity of both HEFT and BMTC.

These results are by no means a benchmark of different scheduling policies. Instead, our objective is to show that users may choose among different scheduling policies and/or different workflow engines, and combine them according to their specific needs.

Once the different tested scenarios have been executed and analyzed, we now combined different scheduling policies with different workflow engines, to be able to verify the flexibility of our system. An additional insight of this experiment is showing that we are able to improve the applications performance when comparing to those workflow engines default scheduling policies by using more sophisticated policies. In this experiment, schedflow was connected to Taverna, DAGMan and Karajan, while using the random, Min-min, HEFT and BMTC policies.It must be noted that in this experiments, no higher priority tasks - which would cause suspension - were injected.

The results of last scenario are summarized in table 1, which shows maximum, minimum, average and standard deviation of the execution time for each scenario and with the different scheduling policy used.

In order to validate the engine interface of our system, we compared DAGMans default policy (random), with a random policy created for the experiment, and their results are similar, as can be seen in table 1, with a difference of only 4 sec.

Another validation is that of using the same policy over different workflow engines. The hypothesis was that the same policy should attain similar results independent of the workflow engine being used. As seen in table 1, each policy under different workflow engines performs very similarly. The only variation we found was when using the default policy, which is not necessarily the same for each workflow engine.

| Scenario | Default Policy | Random Policy | Min-Min Policy | HEFT Policy | BMTC Policy |
|----------|----------------|---------------|----------------|-------------|-------------|
| Condor   | mean:9781      | mean:9777     | mean:6677      | mean:6034   | mean:5986   |
|          | stdev:13       | stdev:9       | stdev:19       | stdev:17    | stdev:13    |
| Taverna  | mean:10229     | mean:9781     | mean:6682      | mean:6112   | mean:5998   |
|          | stdev:14       | stdev:8       | stdev:15       | stdev:19    | stdev:14    |
| Karajan  | mean:12415     | mean:9782     | mean:6694      | mean:6119   | mean:6013   |
|          | stdev:15       | stdev:9       | stdev:17       | stdev:18    | stdev:15    |

Table 1: Makespan summary.

In any case, these results are encouraging, as they show that schedflow constitutes a valuable tool for developing, using and evaluating in practice new scheduling policies for workflow applications. It is a flexible tool that simplifies
the integration of new scheduling policies into a workflow engine. Additionally, schedflow allows more coherent comparisons between different scheduling policies, both static and dynamic, as they can be executed with the same underlying workflow engine.

5. Conclusions

We have described schedflow, a system that allows an easy integration of scheduling policies into existing workflow engines. SchedFlow provides a simple API, for scheduling developers, that isolates many details of the underlying workflow engine.

A first prototype has been implemented for condor dagman, taverna, and karajan. The modular structure of our system simplifies the portable effort because only a small number of methods must be adapted to link schedflow to a new engine. Moreover, no changes are required to the workflow engine or to schedflow when the user wants to use a different scheduling policy.

We performed several experiments with our schedflow prototype, which was used to schedule a montage application running on a Condor pool. These experiments showed the potential of our tool. On the one hand, integration of
simple policies was done in an easy and quickly way, about different workflow engines. On the other hand, our results with the montage application highlighted the benefits of dynamic policies that take into account run-time events that affect the execution of the workflow.

We believe that schedflow could be a valuable tool to explore new scheduling strategies that, in contrast to many theoretical approaches are only evaluated through simulation. By doing so, we will be able to understand the behavior of different scheduling policies under real conditions imposed by existing workflow engines.

Acknowledgments

Special thanks to Daniel S. Katz for his valuable help regarding the montage application.

References

[1] Dagman engine. URL http://www.cs.wisc.edu/condor/dagman
[2] E. Deelman, J. Blythe, et al., Pegasus: Planning for execution in grids, Tech. rep., GriPhyN Project (2002).
[3] T. Oinn, M. Addis, et al., Taverna: a tool for the composition and enactment of bioinformatics workflows, Bioinformatics Journal, London-UK (2004) pp. 3045–3054.
[4] D. Hull, K. Wolsencroft, et al., Taverna: a tool for building and running workflows of services, Nucleic Acids Research (2006) pp. 729–732.
[5] I. Taylor, M. Shields, et al., Visual grid workflow in triana, Journal of Grid Computing, Netherlands (2006) pp. 153–169.
[6] Karajan engine. URL http://www.cogkit.org/release/4.0_a1/manual/workflow/workflow.html
[7] J. Yu, R. Buyya., Taxonomy of workflow management systems for grid computing, Sigmod, New York-USA (2005) pp. 44–49.
[8] H. Topcuoglu, S. Hariri, M. Wu., Performance- effective and low-complexity task scheduling for heterogeneous computing, IEEE Trans on Parallel and Distribution System (2002) pp. 260–274.
[9] M. Wieczorek, R. Prodan, T. Fahringer., Scheduling of scientific workflows in the askalon grid environment, ACM SIGMOD Record (2005) pp. 56–62.
[10] R. Sakellariou, H. Zhao., A hybrid heuristic for dag scheduling on heterogeneous system, In Proceeding of 18th International Parallel and Distributed Processing Symposium, New Mexico-USA (2004) pp. 111.
[11] H. Xiaoshan, X. Sun, G. Laszewski., Qos guided min-min heuristic for grid task scheduling, International Journal of Computer Science and Technology, Beijing- China (2003) pp. 442–451.
[12] H. Cao, H. Jin, et al., Dagmap: Efficient scheduling for dag grid workflow job, In Proceeding of the CCGRID, Francia (2008) pp. 17–24.
[13] B. Bode, D. Halstead, R. Kendall, Z. Lei, D. Jackson., The portable batch scheduler and the maui scheduler on linux clusters, In Proceedings of the 4th Annual Linux Showcase & Conference, Atlanta (2000) pp. 117.
[14] K. Amin, M. Hategan, et al., Gridant: A client- controllable grid workflow system, In Proceeding of the 37th Hawaii International Conference on System Science (2004) pp. 5–8.
[15] Y. Huang, Q. Huang, et al., Grid infrastructure based on multiple workflow engines, In Proceeding of 27th International Conference on Information Technology Interface (2005) pp. 59–64.
[16] H. Truong, P. Brunner, et al., Dipas: A distributed performance analysis service for grid service-based workflows, Future Generation Computer Systems (2009) pp. 385–398.
[17] M. Iverson, G. Follen., Run-time statistical estimation of task execution times for heterogeneous distributed computing, In Proceeding of the HPDC Conference (1996) pp. 263–270.
[18] G. Berriman, A. Laity, et al., Montage: The architecture and scientific applications of a national virtual observatory service for computing astronomical image mosaics, In. Proceeding. of Earth Sciences Technology Conference, Maryland-USA.