Partitioning and Scheduling Workflows across Multiple Sites with Storage Constraints

Weiwei Chen and Ewa Deelman

Information Sciences Insitute,
University of Southern California, Marina del Rey, CA, 90292, USA
{wchen,deelman}@isi.edu

Abstract. This paper aims to address the problem of scheduling large workflows onto multiple execution sites with storage constraints. Three heuristics are proposed to first partition the workflow into sub-workflows. Three estimators and two schedulers are then used to schedule sub-workflows to the execution sites. Performance with three real-world workflows shows that this approach is able to satisfy storage constraints and improve the overall runtime by up to 48% over a default whole-workflow scheduling.

Keywords: workflow scheduling, partitioning, storage constraints

1 Introduction

Scientific workflows [1] have been widely applied in astronomy [2], seismology [3], genomics [4], etc. A scientific workflow has a sequence of jobs that perform required functionality and it has control or data dependencies between jobs. Workflows in systems such as Pegasus [5] are defined at an abstract level, devoid of resource assignments. A key problem that needs to be addressed is the mapping of jobs in the workflow onto resources that are distributed in the wide area. This is especially challenging for data-intensive workflows that require significant amount of storage. For these workflows, we need to use multiple execution sites and consider their available storage. For example, the CyberShake [3] workflow has 80 sub-workflows and each sub-workflow has more than 24,000 individual jobs and 58 GB data. A sub-workflow is a workflow and also a job of a higher-level workflow. We use Condor [6] pools as execution sites.

In this paper, we have developed a three-phase scheduling approach integrated with the Pegasus Workflow Management System [5] to partition, estimate, and schedule workflows onto distributed resources. Pegasus is a workflow-mapping and execution engine that is used to map large-scale scientific workflows onto the available resources. Our contributions include three heuristics to partition workflows respecting storage constraints and internal job parallelism. We utilize three methods to estimate runtime of sub-workflows and then we schedule them based on two commonly used algorithms (MinMin[7] and HEFT[8]).

The reason that we partition workflows into sub-workflows instead of scheduling individual jobs is that this approach reduces the complexity of the workflow

mapping. For example, the entire CyberShake workflow has more than 1.9×10^5 tasks, which is a large number for workflow management tools. In contrast, each sub-workflow has 24,000 tasks, which is acceptable for these tools.

We model workflows as Directed Acyclic Graphs (DAGs), where nodes represent computation and directed edges represent data flow dependencies between nodes. Such workflows comprise sequences of fan-out (where the output of a job is input to many children), fan-in (where the output of several jobs is aggregated by a child), and pipeline nodes (1 parent, 1 child). We assume that the size of each input file and output file is known, and that they are much smaller than the storage constraints of a site. To estimate the runtime of sub-workflows, runtime information for each job is required. In our proposed heuristics, we use historical performance information.

2 Related Work

Considerable amount of work tried to solve workflow-mapping problem using DAG scheduling heuristics such as HEFT[8], Min-Min[7], etc. Sonmez [10] extended them to multiple workflows in multi-cluster grids. Duan [9], Wieczorek [16] have discussed the scheduling and partitioning of scientific workflows in dynamic grids. These algorithms do not take storage constraints into consideration and they need to check every job and schedule it, while our algorithm only needs to check a few particular types of jobs (see Section 3).

Singh[11] optimized disk usage and runtime performance by removing data files when they're no longer required. We distinguish our work in three points. First, there is an upper bound of the amount of data that can be cleaned up. If the workflow with data clean up is still too large for a single site to run, our work tries to find a valid partitioning if it exists. Second, our algorithm only needs to check a few particular jobs instead of the entire workflow. Third, simply applying scheduling algorithms to this problem and grouping jobs at the same sites into sub-workflows may result in invalid workflows with cross dependencies (see Section 3). Data clean up can be simply added to our approach.

Workflow partitioning can be classified as a network cut problem [12] where a sub-workflow is viewed as a sub-graph. But there are two differences with our approach. First, we must consider the problem of data overlap when a new job is added to a sub-workflow. Second, valid workflows require no cross dependencies although it is possible to make that cut in network cut problem.

3 System Design

Our approach (shown in Fig.1) has three phases: partition, estimate and schedule. The partitioner takes the original workflow and site catalog (containing information about available execution sites) [5] as input, and outputs various sub-workflows that respect the storage constraints—this means that the data requirements of a sub-workflow are within the data storage limit of a site. The site catalog provides information about the available resources. The estimator

provides the runtime estimation of the sub-workflows and supports three estimation methods. The scheduler maps these sub-workflows to resources considering storage requirement and runtime estimation. The scheduler supports two commonly used algorithms. We first try to find a valid mapping of sub-workflows satisfying storage constraints. Then we optimize performance based on these generated sub-workflows and schedule them to appropriate execution sites if runtime information for individual jobs is already known. If not, a static scheduler maps them to resources merely based on storage requirements.

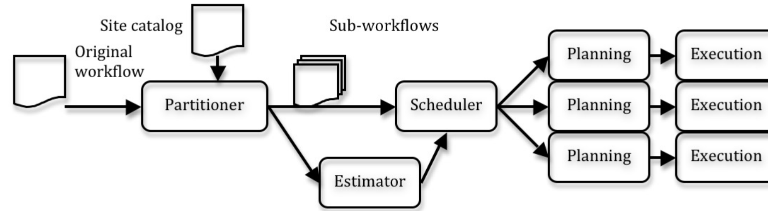


Fig. 1. The steps to partition and schedule a workflow

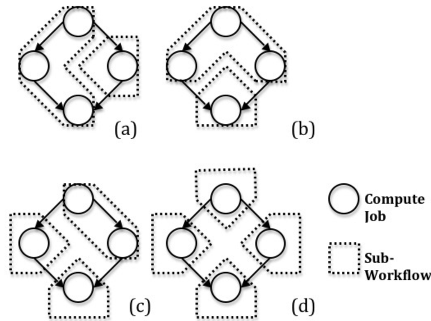


Fig. 2. Four Partitioning

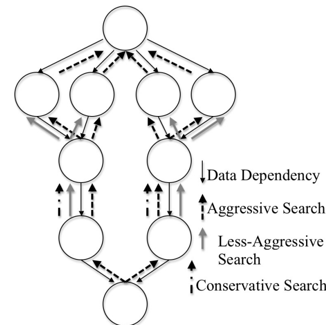


Fig. 3. Three Steps of Searches

Cross Dependency The major challenge in partitioning workflows is to avoid cross dependency, which is a chain of dependencies that forms a cycle in the graph (in this case cycles between sub-workflows). With cross dependencies, workflows are not able to proceed since they form a deadlock loop. For a workflow depicted in Fig.2, we show the result of four different partitioning. Partitioning (a) does not work in practice since it has a deadlock loop. Partitioning (c) is valid but not efficient compared to Partitioning (b) or (d) that have more parallelism.

Partitioner Usually jobs that have parent-child relationships share a lot of data since they have data dependencies. It is reasonable to schedule such

jobs into the same partition to avoid extra data transfer and also to reduce the overall runtime. Thus, we propose Heuristic I to find a group of parents and children. Our heuristic only checks three particular types of nodes: the fan-out job, the fan-in job, and the parents of the fan-in job and search for the potential candidate jobs that have parent-child relationships between them. The check operation means checking whether one particular job and its potential candidate jobs can be added to a sub-workflow while respecting storage constraints. Thus, our algorithm reduces the time complexity of check operations by n folds, while n is the average depth of the fan-in-fan-out structure. The check operation takes more time than the search operation since the calculation of data usage needs to check all the data allocated to a site and see if there is data overlap. Similar to [8], the algorithm starts from the sink job and proceeds upward.

```
Input: Workflow G
Input: Site List SL[index], which stores all
information about a compute site.
Output: Sub-workflow List SWL[index], which has
all the sub-workflows scheduled to a compute site.
Let index = 0;
Q = new Queue()
Add the sink job of G to Q
S = new subworkflow()
While Q is not empty
  Let j be the last job in Q
  //for fan-in job
  Do Aggressive-Search from j
  Let C be the list of potential candidate jobs to be
added to S on SL[index]
  Let P be the list of parents of all candidates
  Let D be the data size on SL[index] with C
  If(D > storage constraint of SL[index])
  Do Less-Aggressive-Search from j, update C, P, D
    If(D > storage constraint of SL[index])
      Do Conservative-Search from j, update C, P, D
    EndIf
  EndIf
  // for other types of job
  ...
  If(S causes cross dependency at SL[index])
    S = new subworkflow()
  EndIf
  Add all the jobs in C to S
  Add all the jobs in P to the head of Q
  add S to SWL[index]
  If(SL[index] has no enough space left)
    index ++
  EndIf
  //for other situations
  ...
  Remove j from Q
EndWhile
Return SWL
```

Fig. 4. Pseudo-code of partitioning. Not all the situations are listed here

To search for the potential candidate jobs that have parent-child relationships, the partitioner tries three steps of searches. For a fan-in job, it first checks if it is possible to add the whole fan structure into the sub-workflow (aggressive search). If not, similar to Fig.2(d), a cut is issued between this fan-in job and its parents to avoid cross dependencies and increase parallelism. Then a less aggressive search is performed on its parent jobs, which includes all of its predecessors until the search reaches a fan-out job. If the partition is still too large, a conservative search is performed, which includes all of its predecessors until the search reaches a fan-in job or a fan-out job. Fig.3 depicts an example of three steps of search while the workflow in it has an average depth of 4. Pseudo-code of Heuristic I is depicted in Fig.4.

The partitioner starts by picking an execution site from site catalog and forming a sub-workflow with the heuristic above. Users can specify the order of execution sites to be picked or the partitioner will sort them in the order of storage constraints. If the execution site does not have sufficient storage to host

any more jobs, a new execution site is selected. For the dependencies between jobs across multiple sub-workflows, they form the new dependencies between sub-workflows and are added to the final graph. The partitioner guarantees to satisfy storage constraints since in each step it assures the size of all sub-workflows assigned to a site is smaller than its storage constraint.

To compare the approach we propose two other heuristics. The motivation for Heuristic II is that Partitioning (c) in Fig.2 is able to solve the problem. The motivation for Heuristic III is an observation that partitioning a fan structure into multiple horizontal levels is able to solve the problem. **Heuristic II** adds a job to a sub-workflow if all of its unscheduled children can be added to that sub-workflow without causing cross dependencies or exceed the storage constraint. **Heuristic III** adds a job to a sub-workflow if two conditions are met: 1) for a job with multiple children, each child has already been scheduled; 2) after adding this job to the sub-workflow, the data size does not exceed the storage constraint.

Estimator To optimize the workflow performance, runtime estimation for sub-workflows is required assuming runtime information for each job is already known. We provide three methods. **Critical Path** is defined as the longest depth of the sub-workflow weighted by the runtime of each job. **Average CPU Time** is the quotient of cumulative CPU time of all jobs divided by the number of available resources. The **HEFT** estimator uses the calculated earliest finish time of the last sink job as makespan of sub-workflows assuming that we use HEFT to schedule sub-workflows.

Scheduler The scheduler selects appropriate resources for the sub-workflows satisfying the storage constraints and optimizes the runtime performance. We select **HEFT**[8] and **MinMin**[7]. There are two differences compared to their original versions. First, the data transfer cost within a sub-workflow is ignored since we use a shared file system in our experiments. Second, the data constraints must be satisfied for each sub-workflow. The scheduler selects a near-optimal set of resources in terms of available Condor slots since its the major factor influencing the performance. Although some more comprehensive algorithms can be adopted, HEFT or MinMin are able to improve the performance significantly in terms that the sub-workflows are already generated since the number of sub-workflows has been greatly reduced compared to the number of individual jobs.

4 Experiments and Evaluations

In order to quickly deploy and reconfigure computational resources, we use a cloud computing resource in FutureGrid [17] running Eucalyptus [13]. Eucalyptus is an infrastructure software that provides on-demand access to Virtual Machine (VM) resources. In all the experiments, each VM has 4 CPU cores, 2 Condor slots, 4GB RAM and has a shared file system mounted to make sure data staged into a site is accessible to all compute nodes. In the initial experiments we build up four clusters, each with 4 VMs, 8 Condor slots. In the last experiment of site selection, the four virtual clusters are reconfigured and each

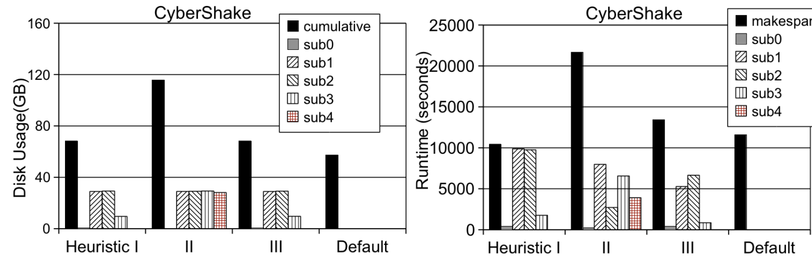


Fig. 5. Performance of the three heuristics. The default workflow has one execution site with 4 VMs and 8 Condor slots and has no storage constraint

cluster has 4, 8, 10 and 10 Condor slots respectively. The submit host that performs workflow planning and which sends jobs to the execution sites is a Linux 2.6 machine equipped with 8GB RAM and an Intel 2.66GHz Quad CPUs. We use Pegasus to plan the workflows and then submit them to Condor DAGMan [14], which provides the workflow execution engine. Each execution site contains a Condor pool and a head node visible to the network.

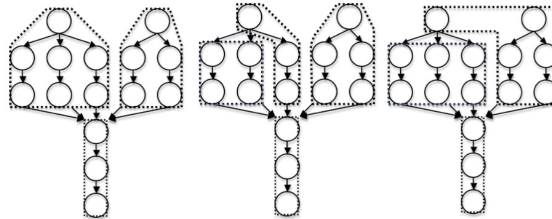
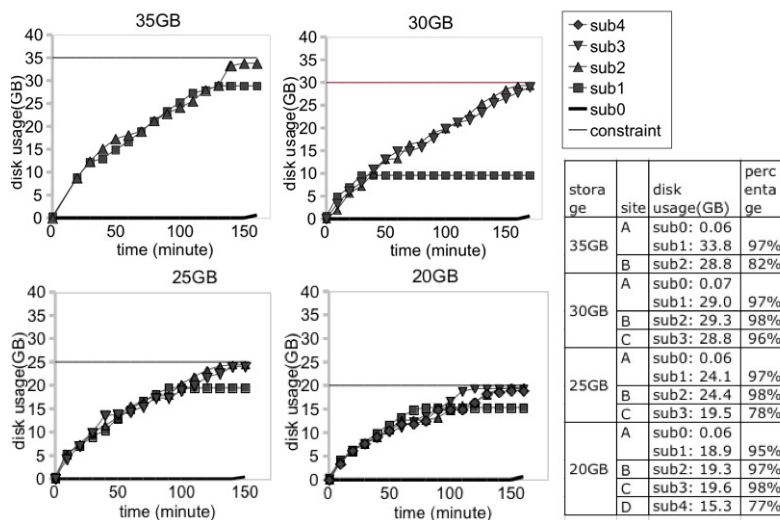


Fig. 6. From left to right: Heuristic I, Heuristic II, Heuristic III

Performance Metrics To evaluate the performance, we use two types of metrics. **Satisfying the Storage Constraints** is the main goal of our work in order to fit the sub-workflows into the available storage resources. We compare the results of different storage constraints and heuristics. **Improving the Runtime Performance** is the second metric that is concerned with to minimize the overall makespan. We compare the results of different partitioners, estimators and schedulers.

Workflows Used We ran three different workflow applications: an astronomy application (Montage), a seismology application (CyberShake) and a bioinformatics application (Epigenomics). They were chosen because they represent a wide range of application domains and a variety of resource requirements [15]. For example, Montage is I/O intensive, CyberShake is memory intensive, and Epigenomics is CPU intensive. The goal of the CyberShake Project [3] is to calculate Probabilistic Seismic Hazard curves for locations in Southern California



storage	site	disk usage(GB)	percentage
35GB	A	sub0: 0.06 sub1: 33.8	97%
	B	sub2: 28.8	82%
30GB	A	sub0: 0.07 sub1: 29.0	97%
	B	sub2: 29.3	98%
	C	sub3: 28.8	96%
25GB	A	sub0: 0.06 sub1: 24.1	97%
	B	sub2: 24.4	98%
	C	sub3: 19.5	78%
20GB	A	sub0: 0.06 sub1: 18.9	95%
	B	sub2: 19.3	97%
	C	sub3: 19.6	98%
	D	sub4: 15.3	77%

Fig. 7. CyberShake with storage constraints of 35GB, 30GB, 25GB, and 20GB. They have 3, 4, 4, and 5 sub-workflows and require 2, 3, 3, and 4 sites to run respectively

area. We ran one partition that has 24,132 tasks and 58GB of overall data. Montage [2] is an astronomy application that is used to construct large image mosaics of the sky. We ran a Montage workflow with a size of 8 degree square of sky. The workflow has 10,422 tasks and 57GB of overall data. Epigenomics [4] maps short DNA segments collected with gene sequencing machines to a reference genome. The workflow has 1,586 tasks and 23GB of overall data. We ran each workflow instance 5 times to assure the variance is within 10%.

Performance of Different Heuristics We compare the three proposed heuristics with the CyberShake application. The storage constraint for each site is 30GB. Heuristic II produces 5 sub-workflows with 10 dependencies between them. Heuristic I produces 4 sub-workflows and 3 dependencies. Heuristic III produces 4 sub-workflows and 5 dependencies. The results are shown in Fig.5 and Heuristic I performs better in terms of both runtime reduction and disk usage. This is due to the way it handles the cross dependency. Heuristic II or Heuristic III simply adds a job if it does not violate the storage constraints or the cross dependency constraints. Furthermore, Heuristic I puts the entire fan structure into the same sub-workflow if possible and therefore reduces the dependencies between sub-workflows. The entire fan structure is defined as a set of jobs and begins from a fan-out job and merges into a fan-in job. In Fig.6 with a simplified CyberShake workflow, Heuristic I runs two sub-workflows in parallel while the other two have to run them in sequence. From now on, we only use Heuristic I in the partitioner in our experiments.

Performance with Different Storage Constraints Fig.7 depicts the disk usage of the CyberShake workflows over time with storage constraints of 35GB,

30GB, 25GB, and 20GB. They are chosen because they represent a variety of required execution sites. Fig.8 depicts the performance of both disk usage and runtime. Storage constraints for all of the sub-workflows are satisfied. Among them sub1, sub2, sub3 (if exists), and sub4 (if exists) are run in parallel and then sub0 aggregates their work. The CyberShake workflow across two sites with a storage constraint of 35GB performs best. The makespan (overall completion time) improves by 18.38% and the cumulative disk usage increases by 9.5% compared to the default workflow without partitioning or storage constraints. The cumulative data usage is increased because some shared data is transferred to multiple sites. Adding more sites does not improve the makespan because they require more data transfer even though the computation part is improved.

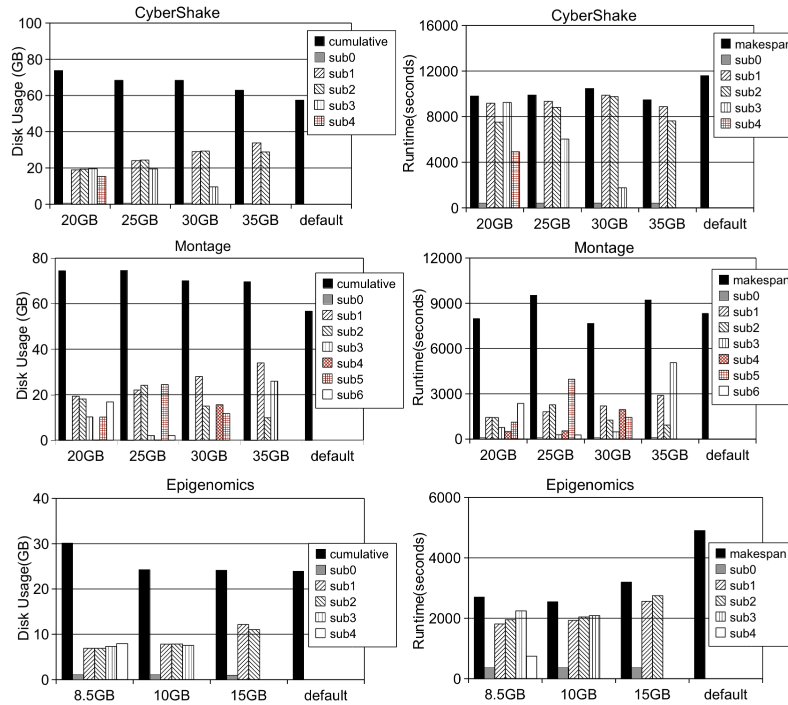
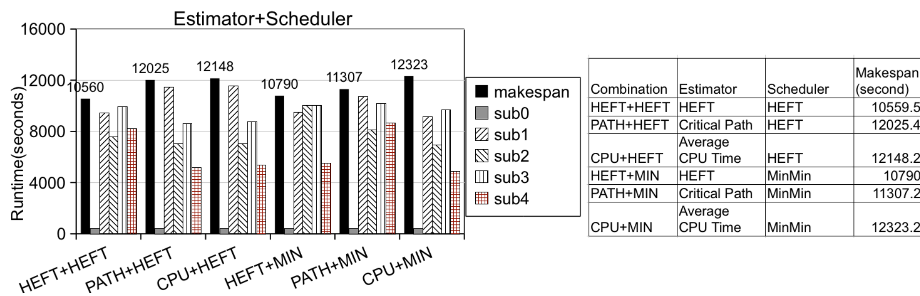


Fig. 8. Performance of CyberShake, Montage and Epigenomics with different storage constraints.

Fig.8 depicts the performance of Montage with storage constraints ranging from 20GB to 35GB and Epigenomics with storage constraints ranging from 8.5GB to 15GB. The Montage workflow across 3 sites with 30GB disk space performs best with 8.1% improvement in makespan and the cumulative disk usage increases by 23.5%. The Epigenomics workflow across 3 sites with 10GB storage constraints performs best with 48.1% reduction in makespan and only

Fig. 9. The Montage (left) and the Epigenomics (right) workflows. For simplicity, only a few branches are shown

1.4% increase in cumulative storage. The reason why Montage performs worse is related to its complex internal structures. Montage has two levels of fan-out-fan-in structures and each level has complex dependencies between them as shown in Fig.9. Our heuristic is not able to untie them thoroughly and thereby the cost of data transfer increases and the sub-workflows are not able to run in parallel.



Combination	Estimator	Scheduler	Makespan (second)
HEFT+HEFT	HEFT	HEFT	10559.5
PATH+HEFT	Critical Path	HEFT	12025.4
CPU+HEFT	Average CPU Time	HEFT	12148.2
HEFT+MIN	HEFT	MinMin	10790
PATH+MIN	Critical Path	MinMin	11307.2
CPU+MIN	Average CPU Time	MinMin	12323.2

Fig. 10. Performance of estimators and schedulers

Site selection We use three estimators and two schedulers described in Section 3 together with the CyberShake workflow. We build four execution sites with 4, 8, 10 and 10 Condor slots respectively. The labels in Fig.10 are defined in a way of Estimator+Scheduler. For example, HEFT+HEFT denotes a combination of HEFT estimator and HEFT scheduler, which performs best. The Average CPU Time (or CPU in Fig. 10) does not take the dependencies into consideration and the Critical Path (or PATH in Fig.10) does not consider the resource availability. The HEFT scheduler is slightly better than MinMin scheduler (or MIN in Fig.10). Although HEFT scheduler uses a global optimization algorithm compared to MinMins local optimization, the complexity of scheduling sub-workflows has been greatly reduced compared to scheduling a vast number of individual tasks. Therefore, both of them are able to handle such situations.

5 Conclusions

This paper provides a solution to address the problem of scheduling large workflows across multiple sites with storage constraints. Three heuristics are proposed and compared to show the close relationship between cross dependency and runtime improvement. The performance with three workflows shows that this approach is able to satisfy the storage constraints and reduce the makespan significantly especially for Epigenomics, which has fewer fan-in (synchronization) jobs.

6 Acknowledgements

This work is supported by NFS under grant number IIS-0905032, and OCI-0910812 (FutureGrid). We would like to thank G. Juve, K. Vahi and M. A. Amer for their help.

References

1. I. J. Taylor, E. Deelman, et al. Workflows for e-Science. Scientific Workflows for Grids. Springer, 2007.
2. G. B. Berriman, et al., Montage: a grid-enabled engine for delivering custom science-grade mosaics on demand, in Proc. of SPIE vol. 5493, ed, 2004, pp. 221-232.
3. P. Maechling, E. Deelman, et al., SCEC CyberShake Workflows–Automating Probabilistic Seismic Hazard Analysis Calculations, Workflows for e-Science. Scientific Workflows for Grids. Springer, 2007.
4. USC Epigenome Center, http://epigenome.usc.edu.
5. E. Deelman, et al., Pegasus: A framework for mapping complex scientific workflows onto distributed systems, Sci. Program., vol. 13, pp. 219-237 2005.
6. M. Litzkow, et al., Condor–A Hunter of Idle Workstations. ICDCS, Jun 1988.
7. J. Blythe, et al. Task Scheduling Strategies for Workflow-Based Applications in Grids. CCGrid, 2005.
8. H. Topcuoglu, et al., Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing, IEEE TPDS, Vol 13, No.3, Mar 2002.
9. R. Duan, et al., Run-time Optimisation of Grid Workflow Applications, 7th IEEE/ACM Intl. Conf. on Grid Computing, Sep 2005.
10. O.O. Sonmez, Application-Oriented Scheduling in Multicluster Grids, http://www.pds.ewi.tudelft.nl/˜sonmez/research.htm, June 2010.
11. G. Singh, et al., Optimizing Workflow Data Footprint, Special issue of the Scientific Programming Journal dedicated to Dynamic Computational Workflows: Discovery, Optimisation and Scheduling, 2007
12. C.H. Papadimitriou, et al. (1998), Combinatorial Optimization: Algorithms and Complexity, Dover. pp. 120128. ISBN 0486402584
13. Eucalyptus Systems. http://www.eucalyptus.com/
14. Condor Team. http://www.cs.wisc.edu/condor/dagman.
15. G. Juve, et al., Scientific Workflow Applications on Amazon EC2, E-Science Workshops, Oxford UK, Dec 2009.
16. M. Wieczorek, et al., Scheduling of scientific workflows in the ASKALON grid environment, SIGMOND Record, Vol 34 Issue 3, Sep 2005.
17. FutureGrid. https://portal.futuregrid.org