Online Performance Monitoring and Analysis of Grid Scientific Workflows*

Hong-Linh Truong[1] and Thomas Fahringer[2]

[1] Institute for Software Science, University of Vienna
truong@par.univie.ac.at
[2] Institute for Computer Science, University of Innsbruck
Thomas.Fahringer@uibk.ac.at

Abstract. While existing work concentrates on developing QoS models of business workflows and Web services, few tools have been developed to support the monitoring and performance analysis of scientific workflows in Grids. This paper describes a Grid service for performance monitoring and analysis of Grid scientific workflows. The service utilizes workflow graphs and various types of performance data including monitoring data of resources, execution status of activities, and performance measurement obtained from the dynamic instrumentation, to provide a rich set of monitoring and performance analysis features. We store workflows and their relevant information, devise techniques to compare constructs of different workflows, and support multi-workflow analysis.

1 Introduction

Recently many interests have been shown in exploiting the potential of the Grid for scientific workflows. Scientific workflows [12] are normally more flexible and diverse than production and administrative business workflows. As the Grid is diverse, dynamic and inter-organizational, even with a particular scientific experiment, there is a need of having a set of different workflows because (i) one workflow mostly is suitable for a particular configuration of underlying Grid systems, and (ii) available resources allocated for a scientific experiment and their configuration are changed in each run on the Grid. This requirement is a challenge for the performance monitoring and analysis of workflows (WFs) because very often the client of performance tools wants to compare the performance of different WF constructs with respect to the resources allocated in order to determine which WF construct should be mapped onto which topology of the underlying Grid. Therefore, multi-workflow analysis, the analysis and comparison of the performance of different WF constructs, ranging from the whole WF to a specific construct (e.g. a fork-join construct), is an important feature. Moreover, the performance monitoring and analysis of Grid scientific workflows must be conducted online. Even though numerous tools have been developed for constructing and executing scientific workflows on the Grid, e.g. [9, 14, 4], there is a lack of tools that support scientists to

* The work described in this paper is supported by the European Union through the IST-2002-511385 project K-WfGrid.

monitor and analyze the performance of their workflows in the Grid. Most existing work concentrates on develop QoS models of business workflows and Web services [8, 3, 1].

To understand the performance of WFs on the Grid, we need to collect and analyze a variety of types of data relevant to the execution of the WFs from many sources. In previous work, we have developed a middleware which supports services to access and utilize diverse types of monitoring and performance data in a unified system named SCALEA-G [16]. This paper presents a Grid performance analysis service for scientific WFs. The analysis service, utilizing the unified monitoring middleware, collects monitoring data from the WF control and invocation services, and performance measurements obtained through the dynamic instrumentation of WF activities, and uses WF graphs to monitor and analyze the performance of WFs during the runtime. Relevant data of WFs including WF graphs and performance metrics are stored, and we develop techniques for comparing the performance of different constructs of WFs.

The rest of this paper is organized as follows: Section 2 outlines the Grid performance analysis service. Performance analysis for WFs is presented in Section 3. We illustrate experiments in Section 4. Section 5 discusses the related work. Finally we summarize the paper and outline the future work in Section 6.

2 Grid Performance Analysis Service

Figure 1 presents the architecture of the Grid monitoring and performance analysis service for WFs. The WF is submitted to the *Workflow Invocation and Control* (WIC) which locates resources and executes the WF. Events containing execution status of activities, such as *queuing, processing*, and information about resources on which the activities are executed will be sent to the monitoring tool. The *Event Processing* processes these events and the *Analysis Control* decides which activities should be instrumented, monitored and analyzed. Based on information about the selected activity instance and its consumed resources, the Analysis Control requests the *Instrumentation and Monitoring Control* to perform the instrumentation and monitoring. Monitoring and measurement

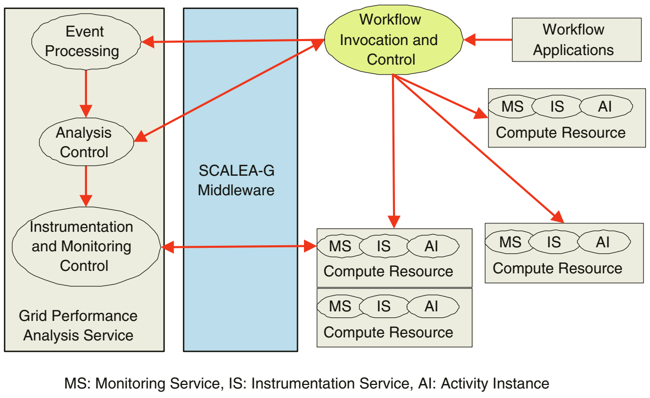

Event Processing
Analysis Control
Instrumentation and Monitoring Control
Grid Performance Analysis Service
SCALEA-G Middleware
Workflow Invocation and Control
Workflow Applications
MS IS AI
Compute Resource
MS IS AI
Compute Resource
MS IS AI
Compute Resource
MS IS AI
Compute Resource
MS: Monitoring Service, IS: Instrumentation Service, AI: Activity Instance

Fig. 1. Model of monitoring and performance analysis of workflow-based application

data obtained are then analyzed. Based on the result of the analysis, the Analysis Control can decide the next step. The performance monitoring and analysis service uses SCALEA-G as its supportive monitoring middleware. The monitoring service (MS) and Instrumentation Service (IS) are provided by SCALEA-G [16].

3 Performance Monitoring and Analysis of Grid Workflows

3.1 Supporting Workflow Computing Paradigm

Currently we focus on the WF modeled as a DAG (Direct Acyclic Graph) because DAG is widely used in scientific WFs. A WF is modeled as a DAG of which a node represents an activity (task) and an edge between two nodes represents the execution dependency between the two activities. An invoked application of an activity instance may be executed on a single or multiple resources.

We focus on analyzing (i) *fork-join* model and (ii) *multi-workflow* of an application. Figure 2(b) presents the fork-join model of WF activities in which an activity is followed by a parallel invocation of n activities. There are several interesting metrics that can be obtained from this model such as load imbalance, slowdown factor, and synchronization delay. These metrics help to uncover the impact of slower activities on the overall performance of the whole structure. We also concentrate on fork-join structures that contain *structured block* of activities. A structured block is a single-entry-single-exit block of activities. For example, Figure 2(c) presents structured blocks of activities.

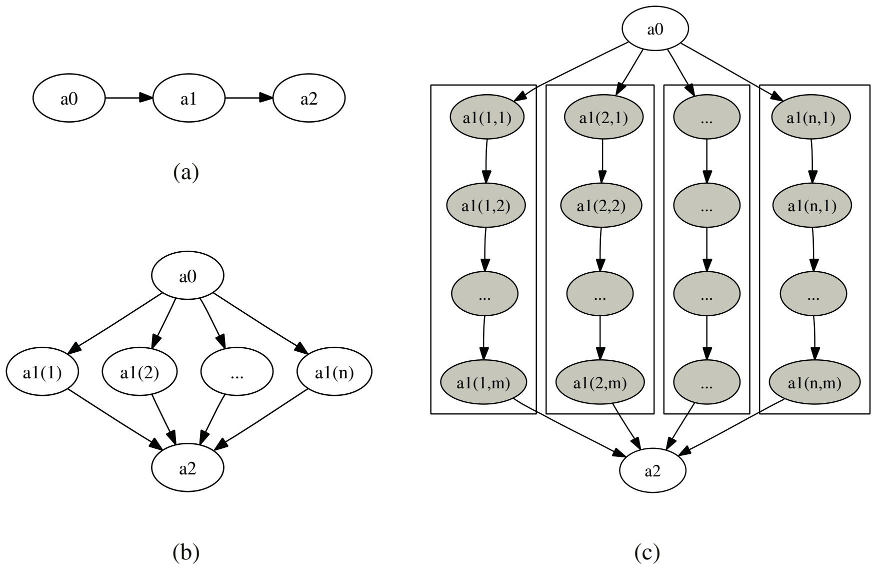


Fig. 2. Multiple workflows of an workflow-based application: (a) Sequence workflow, (b) Fork-join workflow, and (c) Fork-join of structured blocks of activities

A workflow-based application (WFA) can have different versions, each represented by a WF. For example, Figure 2 presents an application with 3 different WFs, each may be selected for execution on specific underlying resources. When developing a WFA, we normally start with a graph describing the WF. The WFA is gradually developed in a step-wise refinement that creates a new WF. In a refinement step, a subgraph may be replaced by another subgraph, resulting in a set of different constructs of the WF. For example, the activity $a1$ in Figure 2(a) is replaced by set of activities $\{a1(1), a1(2), \cdots, a1(n)\}$ in Figure 2(b).

We focus on the case in which a subgraph of a DAG is replaced by a another subgraph in the refined DAG. This pattern occurs frequently when developing WFs. Let G and H be DAG of WF WF_g and WF_h, respectively, of a WFA. G and H represent different versions of the WFA. H is said to be a *refinement* of G if H can be derived by replacing a subgraph SG of G by a subgraph SH of H. The replacement can be controlled by the following constraints:

- Every edge $(a, b) \in G$, $a \notin SG$, $b \in SG$ is replaced by an edge $(a, c) \in H$, $\forall c \in SH$ satisfies no $d \in H$ such that $(d, c) \in SH$.
- Every edge $(b, a) \in G$, $a \notin SG$, $b \in SG$ is replaced by an edge $(c, a) \in H$, $\forall c \in SH$ satisfies no $d \in H$ such that $(c, d) \in SH$.

SH is said to be a *replaced refinement graph* of SG. Note that SG and SH may not be a DAG nor a *connected graph*. For example, consider the cases of Figure 2(a) and Figure 2(b). Subgraph $SG = \{a1\}$ is replaced by subgraph $SH = \{a1(1), a1(2), \cdots, a1(n)\}$; both are not DAG, the first is a trivial graph and the latter is not connected graph. Generally, we assume that a subgraph SG has n components. Each component is either a DAG or a trivial graph. Comparing the performance of different constructs of a WFA can help to select and map WF constructs to the selected Grid resources in an optimal way.

Graph refinement is a well-established field and it is not our focus. We do not concentrate on the determination of refinement graphs in WFs, rather, the WF developers and/or WF construction tools are assumed to do this task. In this paper, (a_i, a_j) indicates the dependency between activity a_i and a_j, and *pred(*a_i*)* and *succ(*a_i*)* denote sets of the immediate predecessors and successors, respectively, of a_i.

3.2 Activities Execution Model

We use discrete process model [13] to represent the execution of an activity a. Let $P(a)$ be the discrete process modeling the execution of activity a. A $P(a)$ is a directed, acyclic, bipartite graph (S, E, A), in which S is a set of nodes representing *activity states*, E is a set of nodes representing *activity events*, and A is a set of edges representing ordered pairs of activity state and event. Simply put, an agent (e.g. WIC, activity instance) causes an event (e.g. submitted) that changes the activity state (e.g. from queuing to processing), which in turn influences the occurrence and outcome of the future events (e.g. active, failed). Figure 3 presents an example of a discrete process modeling the execution of an activity.

Each state s is determined by two events: leading event e_i, and ending event e_j such that $e_i, e_j \in E$, $s \in S$, and $(e_i, s), (s, e_j) \in A$ of $P(a)$. To denote an event *name* of $P(a)$

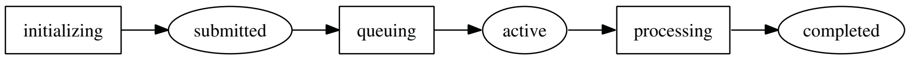


Fig. 3. Discrete process model for the execution of an activity. □ represents an activity state, ◯ represents an activity event

we use $e_{name}(a)$. We use $t(e)$ to refer to the timestamp of an event e and t_{now} to denote the timestamp at which the analysis is conducted. Because the monitoring and analysis is conducted at the runtime, it is possible that an activity a is on a state s but there is no such $(s, e) \in A$ of $P(a)$. When analyzing such state s, we use t_{now} as a timestamp of the current time of state s. We use $\rightarrow$ to denote the *happened before* relation between events.

3.3 Intra-activity and Inter-activity Performance Metrics

Performance data relevant to a Grid WF are collected and analyzed at two levels: *activity* and *workflow* level.

Activity Level. Firstly, we dynamically instrument code regions of the invoked application of the activity. We capture performance metrics of the activity, for example its execution status, performance measurements of instrumented code regions (e.g. wall-clock time, hardware metrics), etc. Performance metrics of code regions are incrementally provided to the user during the execution of the WF. Based on these metrics, various analysis techniques can be employed, e.g. load imbalance, metric ratio. We extend our overhead analysis for parallel programs [15] to WFAs. For each activity, we analyze *activity overhead*. Activity overhead contains various types of overheads, e.g. communication, synchronization, that occur in an activity instance.

Secondly, we focus on analyzing response-time of activities. *Activity response time* corresponds to the time an activity takes to be finished. The response time consists of waiting time and processing time. Waiting time can be queuing time, suspending time. For each activity a, its discrete process of execution model, $P(a)$, is used as the input for analyzing activity response time. Moreover, we analyze synchronization delay between activities. Let consider a dependency between two activities (a_i, a_j) such as $a_i \in pred(a_j)$. $\forall a_i \in pred(a_j)$, when $e_{completed}(a_i) \rightarrow e_{submitted}(a_j)$, the synchronization delay from a_i to a_j, $T_{sd}(a_i, a_j)$, is defined as

$$T_{sd}(a_i, a_j) = t(e_{submitted}(a_j)) - t(e_{completed}(a_i)) \tag{1}$$

If at the time of the analysis, $e_{submitted}(a_j)$ has not occurred, $T_{sd}(a_i, a_j)$ is computed as $T_{sd}(a_i, a_j) = t_{now} - t(e_{completed}(a_i))$. Each activity associates with a set of the synchronization delays. From that set, we compute maximum, average and minimum synchronization delay at a_j. Note that synchronization delay can be analyzed for any activity which is dependent on other activities. This metric is particularly useful for analyzing synchronization points in a WF.

Workflow Level. We monitor and analyze performance metrics that characterize the interaction and performance impact among activities. Interactions between two activities can be file exchanges, remote method invocations or service calls. In the analysis phase, we compute load imbalance, computation to communication ratio, activity usage, and success rate of activity invocation, average response time, waiting time, synchronization delay, etc. We combine WF graph, execution status information and performance data to analyze load imbalance for fork-join model. Let a_0 be the activity at the fork point. $\forall a_i, i = 1 : n, a_i \in succ(a_0)$, load imbalance $T_{li}(a_i, s)$ in state s is defined by

$$T_{li}(a_i, s) = T(a_i, s) - \frac{\sum_{i=1}^{n} T(a_i, s)}{n} \tag{2}$$

We also apply load imbalance analysis to a set of selected activities. In a WF, there could be several activities whose functions are the same, e.g. `mProject` activities in Figure 4, but are not in fork-join model.

3.4 Multi-workflow Analysis

We compute *slowdown factor* for fork-join model. Slowdown factor sf is defined by

$$sf = \frac{max_{i=1}^{n}(T_n(a_i))}{T_1(a_i)} \tag{3}$$

where $T_n(a_i)$ is the processing time of activity a_i in fork-join version with n activities and $T_1(a_i)$ is the processing time of activity a_i in the version with single activity. We also extend the slowdown factor analysis to fork-join structures that contain structured block of activities. In this case, $T_n(a_i)$ will be the processing time of a structured block of activities in a version with n blocks.

For different replaced refinement graphs of WFs of the same WFA, we compute *speedup* factor between them. Let SG be a subgraph of WF WF_g of a WFA; SG has n_g components. Let $P_i =< a_{i1}, a_{i2}, \cdots, a_{in} >$ be a critical path from starting node to the ending node of the component i, C_i, of SG. The processing time of SG, $T_{cp}(SG)$, is defined by

$$T_{cp}(SG) = max_{i=1}^{n_g}(T_{cp}(C_i)), T_{cp}(C_i) = \sum_{k=1}^{n} T(a_{ik}) \tag{4}$$

where $T(a_{ik})$ is the processing time of activity a_{ik}. Now, let SH be the replaced refinement graph of SG, SG and SH are subgraphs of WF WF_g and WF_h, respectively, of a WFA. Speedup factor *sp* of SG over SH is defined by

$$sp = \frac{T_{cp}(SG)}{T_{cp}(SH)} \tag{5}$$

The same technique is used when computing the speedup factor between WF_g and WF_h.

In order to support multi-workflow analysis of WFs, we have to collect and store different DAGs of the WF, performance data and machine information into an experiment repository powered by PostgreSQL. Each graph is stored with its associated performance metrics; graph can be DAG of the WF or a subgraph. We use a table to represent relationship between subgraphs. Currently, for each experiment, the user can select subgraphs, specifying refinement relation between two subgraphs of two WFs. The performance tool uses data in the experiment repository to conduct multi-experiment analysis.

4 Experiments

We have implemented a prototype of the Grid performance analysis service with WIC is based on JavaCog [10]. JGraph [6] and JFreeChart [5] are used to visualize WF DAGs and performance results, respectively. In this section, we illustrate experiments of different WFs of the Montage application in the Austrian Grid [2].

Montage [11] is a software for generating astronomical image mosaics with background modeling and rectification capabilities. Based on the Montage tutorial, we develop a set of WFs, each generates a mosaic from 10 images without applying any background matching. Figure 4 presents experimental WFs of the Montage application. In Figure 4(a), the activity `tRawImage` and `tUncorrectedMosaic` are used to transfer raw images from user site to computing site and resulting mosaics from computing site to user site, respectively. `mProject` reprojects input images to a common spatial scale. `mAdd` coadds the reprojected images. `mImgtbl1` is used to build image table which is accessed by `mProject`, `mAdd`. In WFs executed on multiple resources, we have several subgraphs $tRawImage \rightarrow mImgtbl1 \rightarrow mProject1 \rightarrow tProjectedImage$, each subgraph is executed on a resource. The new `tProjectedImage` activity is used to transfer projected images produced by `mProject` to the site on which `mAdd` is executed. When executed on n resources, the subgraph $mImgtbl2 \rightarrow mAdd \rightarrow tUncorrectedMosaic$ is allocated on one of that n resources.

We conduct experiments on sites named GUP (University of Linz), UIBK (University of Innsbruck), AURORA6 (University of Vienna) and VCPC (University of Vienna) of the Austrian Grid. Due to the space limit, we just present a few experiments of online performance analysis of Montage WFs.

Figure 5 presents the performance analysis GUI when analyzing a Montage WF executed on two resources in UIBK. Performance analysis component retrieves profiling data through the dynamic instrumentation of invoked applications. The left-pane shows the DAG of the WF. The middle-pane shows the dynamic code region call graph (DRG) of invoked applications of activities. We can examine the profiling data of instrumented code region on the fly. The user can examine the whole DRG of the application, or DRG of an activity instance. By clicking on a code region, detailed performance metrics will be displayed in the right-pane. We can examine historical profiling data of a code region, for example window *Historical Data* shows the execution time of code region `computeOverlap` executed on `hafner.dps.uibk.ac.at`. The user also can monitor resources on which activities are executed. For example, the window *Forecast CPU Usage* shows the forecasted CPU usage of `hafner.dps.uibk.ac.at`.

Figure 6(a) presents the response time and synchronization delay analysis for activity *mImgtbl2* when the Montage WF, presented in Figure 4(c), is executed on 5 machines, 3 in AURORA6 and 2 in GUP. The synchronization delay from *tProjectedImage3, 4,*

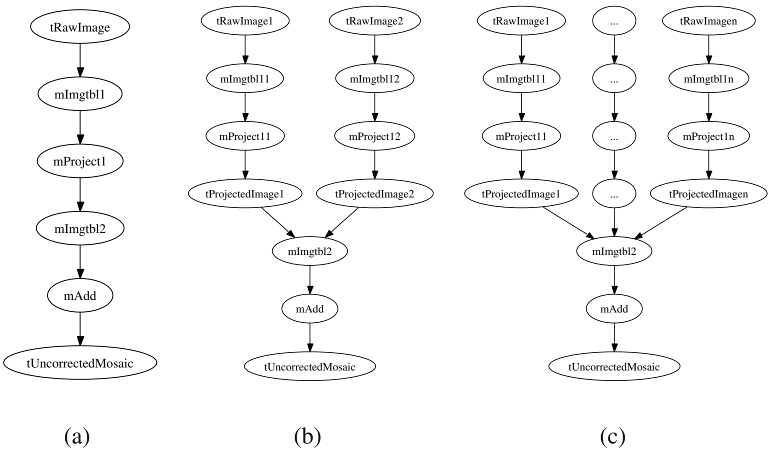
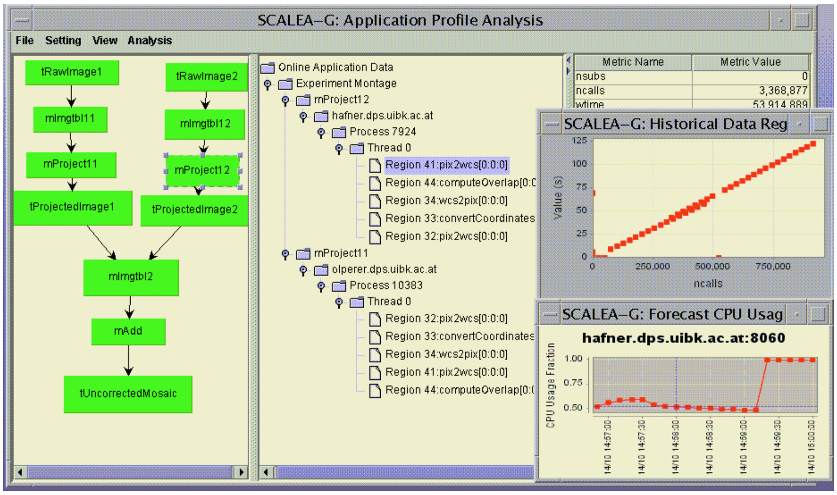


Fig. 4. Experimental workflows of the Montage application: (a) workflow executed on single resource, (b) workflow executed on two resources, and (c) workflow executed on n resources



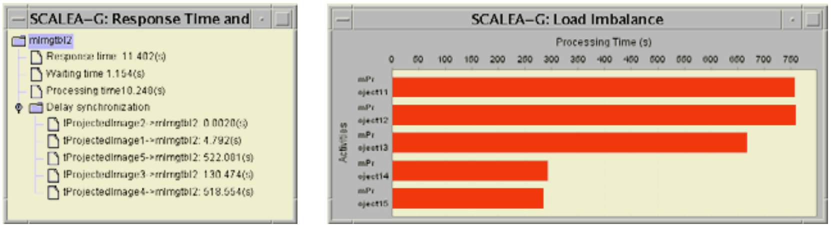
Online Application Data
Experiment Montage
mProject12
hafner.dps.uibk.ac.at
Process 7924
Thread 0
Region 41:pix2wcs[0:0:0]
Region 44:computeOverlap[0:0
Region 34:wcs2pix[0:0:0]
Region 33:convertCoordinates
Region 32:pix2wcs[0:0:0]
mProject11
olperer.dps.uibk.ac.at
Process 10383
Thread 0
Region 32:pix2wcs[0:0:0]
Region 33:convertCoordinates
Region 34:wcs2pix[0:0:0]
Region 41:pix2wcs[0:0:0]
Region 44:computeOverlap[0:0

Metric Name	Metric Value
nsubs	0
ncalls	3,368,877
wtime	53,914,889

Fig. 5. Online profiling analysis for WF activities

5 to *tImgtbl2* are very high. This causes by the high load imbalance between *mProject* instances, as shown in Figure 6(b). The two machines in GUP can process significantly faster than the rest machines in AURORA6.

Over the course of the WF development process, subgraph named `mProjectedImage` which includes $tRawImage \rightarrow mImgtbl1 \rightarrow mProject1$ in single resource version is replaced by subgraphs of $tRawImage \rightarrow mImgtbl1 \rightarrow$



SCALEA–G: Response Time and

mImgtbl2
- Response time 11.402(s)
- Waiting time 1.154(s)
- Processing time10.240(s)
- Delay synchronization
  - tProjectedImage2->mImgtbl2: 0.0020(s)
  - tProjectedImage1->mImgtbl2: 4.792(s)
  - tProjectedImage5->mImgtbl2: 522.001(s)
  - tProjectedImage3->mImgtbl2: 130.474(s)
  - tProjectedImage4->mImgtbl2: 518.554(s)

(a) (b)

Fig. 6. Analysis of Montage executed on 5 machines: (a) response time and synchronization delay of `mImgtbl`, (b) load imbalance of `mProject`

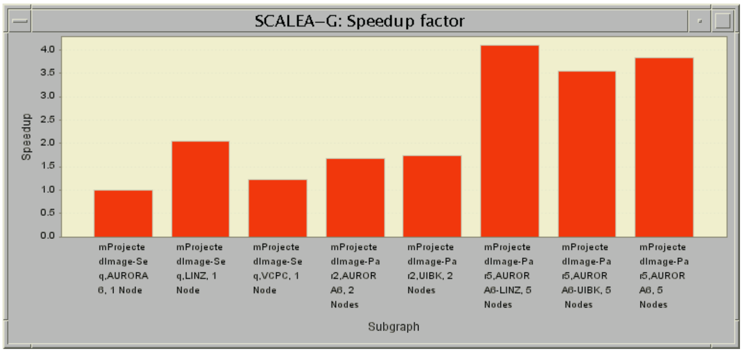


Fig. 7. Speedup factor for refinement graph `ProjectedImage` of Montage WFs

$mProject1 \rightarrow tProjectedImage$ in a multi-resource version. These subgraphs basically provide projected images to the `mAdd` activity, therefore, we consider they are replaced refinement graphs. We collect and store performance of these subgraphs in different experiments. Figure 7 shows the speedup factor for the subgraph `mProjectedImage` of Montage WFs executed on several experiments. The execution of `mProjectedImage` of the WF executed on single resource in LINZ is faster than that of its refinement graph executed on two resources (in AURORA6 or UIBK). However, the execution of `mProjectedImage` of WF executed on 5 resources, 3 of AURORA6 and 2 of LINZ, is just very slightly faster than that executed on 5 resources of AURORA6. The reason is that the slower activities executed on AURORA6 resources have a significant impact on the overall execution of the whole `mProjectedImage` as presented on Figure 6(b).

5 Related Work

Monitoring of WFs is an indispensable part of any WfMS. Therefore it has been discussed for many years. Many techniques have been introduced to study quality of service and performance model of WFs, e.g., [8, 3], and to support monitoring and analysis of the execution of the WF on distributed systems, e.g. in [1]. We share them many common ideas and concepts with respect to performance metrics and monitoring techniques of the WF model. However, existing works concentrate on business WFs and Web services processes while our work targets to scientific WF executed on Grids. We support dynamic instrumentation of activity instances and online monitoring and performance profiling analysis of WFs, and integrate resources monitoring with WF monitoring.

Most effort on supporting the scientist to develop Grid workflow-based applications concentrates on WF language, WF construction and execution systems, but not focuses on monitoring and performance analysis of the Grid WFs. P-GRADE [7] is one of a few tools that supports tracing of WF applications. Instrumentation probes are automatically generated from the graphical representation of the application. It however limits to MPI and PVM applications. Our Grid WF monitoring and performance analysis service supports monitoring execution of activities and online profiling analysis. Also the dynamic instrumentation does not limit to MPI or PVM applications.

6 Conclusion and Future Work

This paper introduces a Grid performance analysis service that can be used to monitor and analyze the performance of scientific WFs in the Grid. The Grid performance analysis service which combines dynamic instrumentation, activity execution monitoring, and performance analysis of WFs in a single system presents a dynamic and flexible way to conduct the performance monitoring and analysis of scientific WFs. We believe techniques for comparing performance of subgraphs of WFs and for supporting multiple-workflow analysis are very useful for optimizing WF structures and mapping WF constructs onto selected underlying Grid resources.

In the current prototype, we manually instrument WIC in order to get execution status of activities. We can extend WF specification language with directives specifying monitoring conditions. These directives will be translated into code used to publish the status to the monitoring middleware. WIC can also offer an interface for the monitoring service to access that status. Meanwhile, the process of analysis, monitoring and instrumentation is controlled by the end-user. The future work is to automate that process.

References

1. Andrea F. Abate, Antonio Esposito, Nicola Grieco, and Giancarlo Nota. Workflow performance evaluation through wpql. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 489–495. ACM Press, 2002.
2. AustrianGrid. http://www.austriangrid.at/.
3. Jorge Cardoso, Amit P. Sheth, and John Miller. Workflow quality of service. In *Proceedings of the IFIP TC5/WG5.12 International Conference on Enterprise Integration and Modeling Technique*, pages 303–311. Kluwer, B.V., 2003.
4. Ewa Deelman, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, Kent Blackburn, Albert Lazzarini, Adam Arbree, and Scott Koranda. Mapping abstract complex workflows onto grid environments. *Journal of Grid Computing*, 1:25–39, 2003.

5. JFreeChart. http://www.jfree.org/jfreechart/.
6. JGraph. http://www.jgraph.com/.
7. P. Kacsuk, G. Dozsa, J. Kovacs, R. Lovas, N. Podhorszki, Z. Balaton, and G. Gombas. P-GRADE: a Grid Programming Environment. *Journal of Grid Computing*, 1(2):171–197, 2003.
8. Kwang-Hoon Kim and Clarence A. Ellis. Performance analytic models and analyses for workflow architectures. *Information Systems Frontiers*, 3(3):339–355, 2001.
9. Sriram Krishnan, Patrick Wagstrom, and Gregor von Laszewski. GSFL : A Workflow Framework for Grid Services. Technical Report, Argonne National Laboratory, 9700 S. Cass Avenue, Argonne, IL 60439, U.S.A., July 2002.
10. G. Laszewski, I. Foster, J. Gawor, and P. Lane. A java commodity grid kit. *Concurrency and Computation: Practice and Experience*, 13(643-662), 2001.
11. Montage. http://montage.ipac.caltech.edu.
12. Munindar P. Singh and Mladen A. Vouk. Scientific workflows. In *Position paper in Reference Papers of the NSF Workshop on Workflow and Process Automation in Information Systems: State-of-the-art and Future Directions*, May 1996.
13. John F. Sowa. *Knowledge Representation: logical, philosophical, and compuational foundations*. Brooks/Cole, Pacific Grove, CA, 2000.
14. The Condor Team. Dagman (directed acyclic graph manager). http://www.cs.wisc.edu/condor/dagman/.
15. Hong-Linh Truong and Thomas Fahringer. SCALEA: A Performance Analysis Tool for Parallel Programs. *Concurrency and Computation: Practice and Experience*, 15(11-12):1001–1025, 2003.
16. Hong-Linh Truong and Thomas Fahringer. SCALEA-G: a Unified Monitoring and Performance Analysis System for the Grid. *Scientific Programming*, 2004. IOS Press. To appear.