Monitoring and Performance Analysis of Workflow Applications in Large Scale Distributed Systems

Dragos Sbirlea, Alina Simion, Florin Pop, and Valentin Cristea

Abstract. The chapter presents the design, implementation and testing of the monitoring solution created for integration with a workflow execution platform. The monitoring solution is a key for modeling and performance analysis of Grid systems considered as a networking and collaborative systems. The monitoring solution constantly checks the system evolution in order to facilitate performance tuning and improvement. The novelty of the work presented in this chapter is the improvement of distributed application obtained using the real-time information to compute estimates of runtime which are used to improve scheduling. Monitoring is accomplished at application level, by monitoring each job from each workflow and at system level, by aggregating state information from each processing node. The scheduling performance in distributed systems can be improved through better runtime estimation and the error detection can automatically detect several types of errors.

1 Introduction

Achieving performance in the environment of the Grid is difficult because of the heterogeneity implied by such a Large Scale Distributed System (LSDS) and because the Grid itself is not a stable platform. One of the promises of the Grid is

Dragos Sbirlea · Alina Simion
Computer Science Department, Rice University,
Houston, Texas, USA
e-mail: alina.gabriela.simion@rice.edu, dragos@rice.edu

Florin Pop · Valentin Cristea
University Politehnica of Bucharest, 313 Splaiul Independentei,
060042 Bucharest, Romania
e-mail: {florin.pop@cs.pub.ro, valentin.cristea}@cs.pub.ro

S. Caballé, F. Xhafa & A. Abraham (Eds.): INCoS 2009, SCI 329, pp. 171–191.
springerlink.com

reliable operation at a small cost, compared with high-end conventional computer systems, because the reliability could come from software, rather than expensive hardware [1]. Because of its geographical dispersion, Grids are not prone to many kinds of failures that would make other systems unusable, such as power or network failures). The Grid management software can send automatically resubmit jobs to other machines when a failure is detected, but failure detection, which is based on profiling and end-to-end monitoring, turns these features into requirements in order to make the Grid promise or reliability possible.

In a dynamic, heterogeneous environment such as the Grid, monitoring is the first step towards building a reliable system form an unreliable one. An end-to-end monitoring system that takes into account the various possible uses in the Grid environment of the monitored data is a solution for application like satellite images processing. The main focus using monitoring is have a mechanism for performance analysis based on behavior models, in order to handle both current complexity of systems design and collaborative interactions. Validation of the analysis using simulation tools or real testbed infrastructures proves the correctness of proposed solution.

The chapter is structured as follows. Section 2 presents the related work and introduces the gProcess architecture. In section 3 the goals of the proposed monitoring solution are presented. Monitoring system architecture is described in section 4 and the components description is presented in section 5. Section 6 presents the application runtime estimation, process based on monitoring information and very important in workflow execution. Testing and results for the proposed solution is analyzed in section 7. In section 8 we conclude on the presented solution and experimental results.

2 Related Work

The proposed monitoring solution is built on top of a workflow-based satellite image processing engine which is being built in the context of the SEEGRID research project. This engine, named gProcess, decomposes image processing workflows into operators which are then run on clusters [2][3].

The initial architecture of gProcess was design based on client-server application paradigm (see in Figure 1)[12]. A client sends a image processing request specifying a workflow and the appropriate inputs to the gProcess web services (1) through a locally installed client application. The web services separate the workflow in operators; jobs are created and then send by the scheduler to be executed on a gLite worker nodes from a gLite cluster (2). The output follows the reverse path, until it reaches the user [13][14].

Two workflow types are already available for use in gProcess, the Enhanced Vegetaion Index (EVI) [15] and the Difference Drought Index (NDDI) [16].

A workflow instance adds the corresponding input files so that the processing can be started. The workflow works with multi-spectral satellite images. Basic transformations operate on a single spectral band and define operators in a workflow. They

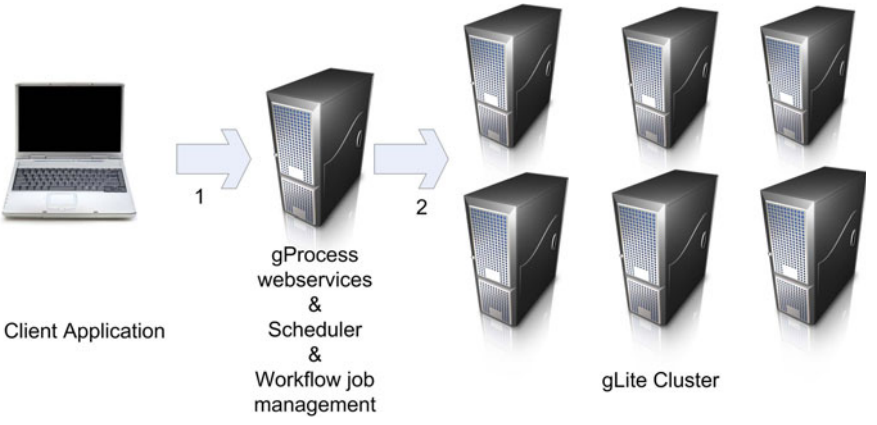

Fig. 1 The initial gProcess architecture; it was changed while implementing the monitoring solution

can compute arithmetic transformations on two images, like the Subtraction (Sub) and Addition (Add) operators, or on one image and a floating point number, like the Multiplication with constant (MultFloat) and Division operators. Some arithmetic transformations require only an image, without a second parameter, like the Complement (Compl) operator. Basic transformations can compute geometric transformations, like rotation, scale and translation, or visualization transformations, such as edge detection or spot detection. These transformations map to operators in gProcess terms, which means they are similar to jobs.

Monitoring Agents using a Large Integrated Services Architecture (MonALISA) [5] is a distributed monitoring system built using a dynamic distributed architecture based on autonomous agents that interact an collaborate in order to perform information gathering and processing. The advantage of such architecture is its scalability. This tool has a good integration with existing monitoring tools and utilities and allows collection of various types of parameters about computational nodes, networks as well as custom application parameters. Based on Java, JINI and web services, MonALISA is used non-stop in more than 350 sites; it's perfectly suited for around the clock operation because it can easily adapt to dynamic changes in the environment. The real-time data visualization feature of MonALISA is an important feature which has proven valuable during the course of this project. A MonALISA repository, which includes a powerful data visualization and analysis engine is used for retrospective data analysis. The repository is built using enterprise java technology running on a Tomcat web server, around a scalable and fast database engine (PosgreSQL). As far as programming language integration, MonALISA allows sensor creation using Java, C, C++, Perl and Python.

3 Goals of the Monitoring Solution

The goals of the proposed monitoring framework are: offer sufficient error management (used for fault recovery), help scheduling improvement (through relevant statistics), do thorough resource utilization accounting and be easily extensible and reusable in another application.

A goal of the monitoring solution is to allow easy visualization of current (real-time) and historical data. Accessibility of these visualization tools is important in the context of the Grid: the more accessible the better. The perfect solution would mean remote access to text based and graphical representations of the data gathered.

It is a primary goal of the monitoring solution to discover, log and announce any condition that might affect the proper functioning of the system. The system should also attempt automatic recovery after an error condition occurs (for example: set a number of retries for a job that was not successful the first time). The monitoring solution should make it easier for the scheduler part of the system to adapt to such conditions.

Another important contribution of the monitoring framework in workflow application concerns scheduling performance. Statistics gathered by the monitoring subsystem can be successfully used to improve the quality of the scheduling [6] [7] [8]. Furthermore, "the quality of the estimated [run] times is essential for the quality of the schedule" [9]. Because it offers real time information on the job runtime in Grid environments, the monitoring system plays an important role in estimating execution time. These estimates need not be very accurate to see an improvement in schedule performance [10].

The monitoring data should be aggregated in such a way as to make it easier for the workflow or application developers to identify bottlenecks and, once identified, to alleviate the problems. In the heterogeneous and rapidly changing context of the Grid, extensibility and reuse are of utmost importance. The solution should be portable to as many operating systems as possible and not be dependent of additional technologies.

4 Monitoring System Architecture

This paragraph describes the improved gProcess architecture, that integrates the monitoring solutions component with the gProcess components.

The proposed, improved gProcess Architecture, consists of Client Application, Web services, Scheduler, Job Management Unit, customized MonALISA Repository, Monitoring Database Server, Automated Error Detection System and Statistics Computing Unit. The client application is the application that requests the execution of a satellite image processing workflow. It can be located anywhere, but must have network access to the GreenView web services, where it sends the request and from where it gets the results back [13]. The gProcess web services are web services built using Java, and deployed in GlassFish Enterprise Application Server and contain scheduling and job management components which decompose the

workflow into tasks, schedule the tasks for running on the gLite cluster, checking that all dependencies are met and get the final output.

The MonALISA repository and database server collect the monitoring information generated by the running jobs and store it for later visualization and accounting. Monitoring information generated by the running jobs is collected and stored using a MonALISA repository and database server, for accounting, visualization and post-processing.

The relationship between all these components is shown in the following system information flow diagram (see Figure 2). A client sends a image processing request to the gProcess web services (1). These services separate the workflow in operators; jobs are created and then send by the scheduler to be executed on a gLite worker node from a gLite cluster (2). The worker nodes that execute the jobs (operators) send monitoring data to a MonALISA Repository for visualization over the Internet and also to a database server for extraction of statistical information (3). The automated error detection system and the statistics computing unit query the database and update it to discover the errors and to reflect the updated statistics values (4). The statistics are used by the scheduler to improve the scheduling process (5).

5 Components Description

The system components that required modification and adaptation to the new monitoring solution are: Job Management Unit, Scheduler, Operator and Workflow Engine. New components that were added are: Monitoring Sensors, Monitoring Engine, MonALISA Repository, Monitoring Database Server, Automated Error Detection System, Estimates Computing Unit, Scheduling Costs Wrapper Classes and Notification Service. In this section a description of each of these components, presented (colored) in figure 3, will be offered.

5.1 Monitoring Sensors

Monitoring Sensors are the components that accomplish the actual extraction of system and application parameters. Although MonALISA offers access to numerous and diverse parameters, these are only system parameters, that do not characterize evolution of the running job, but that of the system. Correlating the job and system behavior remains is a difficult task without job parameters.

The proposed implementation contains custom sensors for systems parameters (these are used if MonALISA does not supply a particular system sensor, or if the sensor implementation is not compatible with the operating system used). A more important feature of the monitoring solution is the possibility for the operator creator to implement their own sensors, that may transmit job related information. Correlated with the system sensors information, this allows better results in remote profiling and system tuning using the proposed monitoring solution.

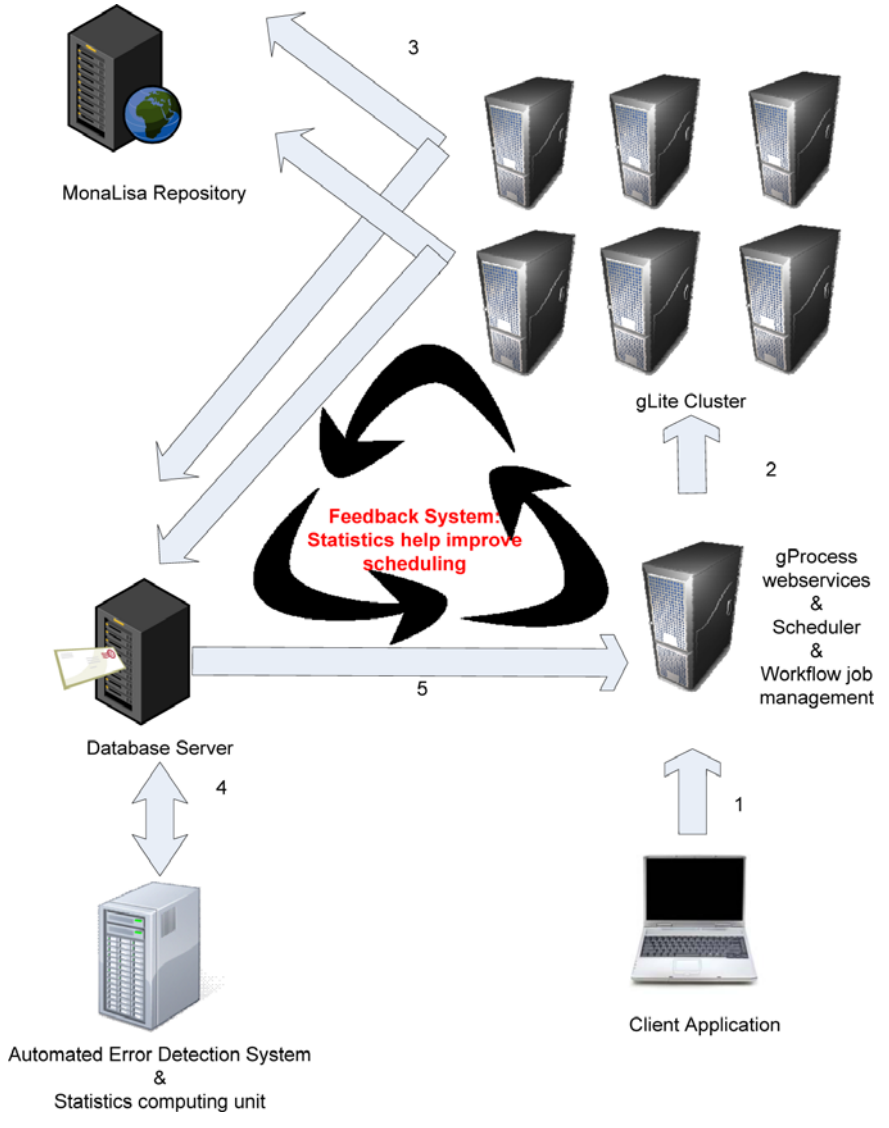

Fig. 2 The improved architecture of gProcess, showing the implemented components of the monitoring system and their integration with the original components. The diagram shows the information flow through the system.

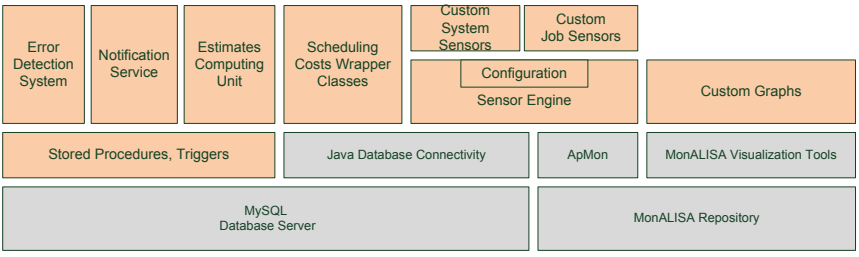

Fig. 3 Monitoring System Components, represented as a stack. The components implemented for the proposed monitoring solution are colored and the external libraries and frameworks used are grayed out.

A small framework that allows for an easy to use, extensible sensors system has been created and is running in the context of the operators that are in fact jobs in a gLite cluster. Because gProcess is developed in Java, the easiest way to interact with it is by writing all code in Java, which was the approach we took. Each operator sent to be executed is wrapped by a special MonitoringOperator object that adds monitoring capabilities and forwards all actual processing done to its wrapped object. Practically, by adding this Monitoring Operator class and, with a small modification in the Operator loading code, monitoring capabilities have been added to a pre-existing application. The goal of this approach was to be minimally invasive in the pre-existing code and keep an elegant design nevertheless.

Because one of the goals of the monitoring solution was extensibility, the actual sensors used are separate Java classes. These are dynamically loaded through Java Reflection API and offer a very powerful extensibility feature to the system. It is to be noted that the current implementation does not even require that these classes implement a certain interface, only that the actual function that does the sensor data retrieval taken no parameters and returns a double (unlike an interface, the function name is not important). Each sensors class, being loaded on demand at runtime can be sent to the worker nodes only when an operator uses it. This creates the possibility of in place upgrades, without need of restart and thus without imposing downtime to the system. In services where the Service Agreement requires a certain uptime, this is especially important.

From the application (job) sensors implemented, the most interesting is the life cycle step sensor, which allows knowing remotely the evolution of the running operator, if it supports this. Such information may be useful in estimating the runtime of that operator, in error checking and debugging (when an operator is stuck in a certain life cycle stage) or in system tuning (the n-th stage of operator Add takes up too many resources which shows developers where to optimize that operator).

The life cycle step of Jobs is a custom job (application) sensor type. In the graphic presented in Figure 4, on the left, we see the evolution of an Add operator (life cycle steps 1 to 4) and on the left, a simple operator (a RGB filter) who uses only steps 1 and 3. The values, scale and type of application sensors depend only on the application developer. In the background, the main window of the MonALISA Web

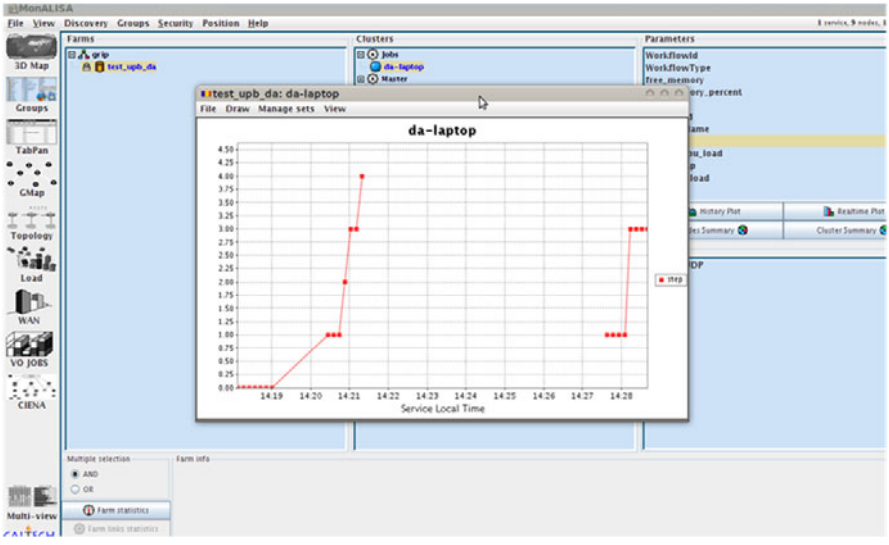

Fig. 4 Monitored life cycle steps

Client is shown; in the right we see the sensor selection list which allows access to the graphs for every sensor offered by the system.

5.2 Monitoring Engine

The Monitoring Engine handles the configuration of monitoring sensors and their arrival at available data repositories. Because each operator might require different sensors, each operator executed comes with a configuration file that indicates the classes and function names of the sensors used. This configuration file can be easily changed, for example to allow the same operator to use different sensors according to which workflow it belongs or to allow the system to enter a debug mode where the monitoring level is increased. These configuration files are in XML format for easy processing by machines and humans and allow setting the interval between consecutive sensor readings.

In the following example, a configuration file for monitoring sensors of operators Sub and Add are displayed; the active sensors are located in a Java class called SystemData (which contains most of the system parameters).The sensor frequency is set to 5 seconds, meaning each 5 seconds, sensors are read and the data obtained is sent to repositories. The sensor name is a string that identifies the values read in the database, it might be different for different operators using the same sensor and can be used as desired by the application administrator. The fragment shows a possible configuration for the Add operator. This configuration is used by gProcess

for workflow execution. The sensors used are listed in the operator node. The other operator XML node (with the value of the name attribute "Operator") is used in case the configuration file is sent (possibly by mistake) as configuration for another operator. These values are used as a failsafe (not having them means no monitoring data is sent).

```
<monitoringData frequency="5">
   <operator name="Add" class="SystemData">
      <sensor name="user_cpu_load"
              method="getCpu"/>
      <sensor name="system_cpu_load"
              method="getSysCpu"/>
      <sensor name="used_swap"
              method="getSwapUsed"/>
      <sensor name="free_memory"
              method="getFreeMemory"/>
      <sensor name="free_memory_percent"
              method="getFr[...]"/>
   </operator>
</monitoringData>
```

The Monitoring Engine sends the sensors data to the repositories. To do this, another problem must be solved, and that is assigning correct operator identity. Matching a job with its appropriate sensor data is done based using an operator configuration file which contains unique identifiers for an operator in the workflow (Add-1), the workflow type ("EVI"), the workflow instance to which it belongs (for example "EVI-24"), like in the following:

```
<operator name="Add"
          workflowTypeId="EVI"
          workflowId="EVI-24"
          operatorWorkflowId="Add-1">
```

The workflow for EVI is sent by user to gProcess execution platform. The input data for the workflow operators is accessed by gProcess in Grid using GridFTP protocol.

5.3 ApMon Usage

The Monitoring Engine uses ApMon, a Java library that works in conjunction with a MonALISA farm. ApMon helps to send the parameter values to the MonALISA farm which is installed on the gLite gateway. Because the farm is installed on the cluster gateway, it can access (using the pbsnodes command) system statistics for each node, including CPU and RAM usage. Several default sensors can be used to monitor systems, using ApMon.

5.4 The Choice of Data Repositories

The repositories included with the system are a database running on a MySQL Server and a MonALISA Repository. Although MonALISA is the de-facto standard for large scale monitoring systems, we opted for a hybrid approach because of the difficulty of using MonALISA to compute the statistics we desired. MonALISA is used, in our solution, for visualization of system parameters in general. Because application parameters are in general characteristic to the particular operator that is running, relevant visualization is more complicated to do using the fixed 3-level nesting level offered by MonALISA.

A main argument for not relying only on MonALISA is redundancy. Data is kept in the MonALISA and MySQL databases increasing the data safety. Further arguments for choosing this hybrid approach is that by doing this we obtain independence on the MonALISA Version. To use MonALISA to calculate the required statistics would have meant to directly access the MonALISA database (a clear warning about the dangers of doing this is given by the developers) or to access them through the supplied web services, which could potentially lock the monitoring framework into using a certain MonALISA version. Furthermore, choosing another solution for data visualization is feasible with the current architecture.

Because one of the goals of the system was easy visualization and flexibility, we used a customized MonALISA repository to create our own graphs of the monitoring data gathered. A repository is also useful for controlling the data gathered (relying on data stored in a third-party server, which you cannot control, is not wise). The repository helps to achieve the goal of data visualization. The solution of using MonALISA for this task meant that time is not wasted by writing code that already exists and as an added advantage is the minimal number of bugs introduced (MonALISA code is used 24/7 in many large scale systems and its reliability is proven).

The customization of the default MonALISA repository was done in three different aspects: customizations affecting the interface, security customizations and graphs creation. The most important change was the creation of new graphs. Each graph has a title, a legend showing what signification has each data series and, according to the graph type, some graphs offer a calendar control for the time interval that should be plotted, a selection list o choose the farm that provides the data that is plotted and a possibility to select only certain series to be represented in the actual graph. Few graphs also offer links to related graphs (see Figure 5).

The graphs are structured in three categories, according to their purpose, as shown in the following screenshot of the MonALISA customized menu. Some graphs are used to monitor the sensors form all nodes; these graphs allow visual representation of CPU History, Memory Usage History and combined view of both of the above. The second category of graphs shows the values of the runtime estimates for the available workflows. These show both the real time values (as pie-chart) for these estimates and the history (as x-y charts). The third graphs category is the one that allows visualization of server sensors only; these graphs are useful in server profiling and performance tuning. From the MonALISA perspective, the created graphs are classified as: pie charts (real-time runtime estimates for EVI

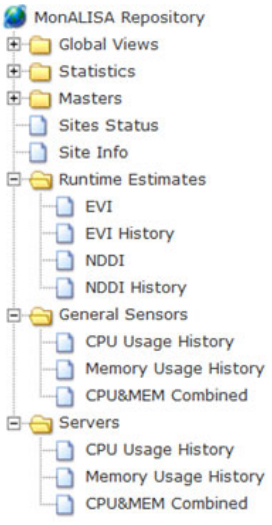

Fig. 5 MonALISA Repository customized menu

workflow), history (CPU, memory usage), real-time (server CPU usage in real-time) and combined (CPU and memory combined).

A important part of the graph creation effort was spent on making the current implementation to be flexible enough to support the further development of the gProcess platform. Here, we find that the servers used (scheduler and web services servers) are discovered dynamically. When adding a new server to the system, there is no need to modify this graph; just start the server monitoring and the data will appear in the servers CPU history graph.

5.5 Monitoring Database Server

The second type of data repository used in the solution is a MySQL database server; the server is used to compute statistics later used for scheduling improvement, to identify errors and to more easily search through the monitoring data (without the three level limitation of MonALISA).

Because the statistics use the computation of operator runtime, the database contains a table for logging the start and stop time for all operators, as well as their identification information. When all operators from a workflow instance are finished, the statistics for the workflow instance can be calculated. The single per - workflow statistic is the perceptual runtime estimates for operators in a workflow, described in the next chapter. This estimate is computer using a trigger and stored procedures.

5.6 Automated Error Detection System

Currently, the system detected errors are timeout and job duplicate errors. To mark timeout jobs, the operators whose end time is not set in the corresponding database table after a set amount of time are moved to a special table. The simplest fault tolerance method that can be built upon such error detection mechanism, is the job retry method (the scheduler can attempt to reschedule them). This creates a simple error detection system and allows fault recovery system at the same time. Other types of errors can be added to the same table but specific methods of identification should be devised for each of them. A version of such method is already implemented by gLite but it does not always work, because job errors are not detected by gLite. This is a major advantage for the error detection system, but it further work is needed to identify possible job-related errors and add the detection mechanism to the other error detection database functions.

5.7 Notification Service

The Notification Service is a program, created to be run periodically (maybe as a cron job in Linux environment) which send notifications to the administrator concerning the errors that have been discovered. Now, it supports ending email (plain text and html with images) and is implemented as Java classes.

5.8 Estimates Computing Unit

The Estimates Computing Unit updates the runtime estimates to the latest values after a workflow is finished. This is accomplished through a trigger that checks, on insert, in the table that logs the start and end times of operators, that a workflow has been successfully finished. It uses the previous statistics and the current values obtained from the newly executed workflow to obtain new estimates (statistics) for the next execution of that particular type of workflow. These estimates values are the sent to the MonALISA Repository for storage and visualization.

5.9 Scheduler Costs Wrapper

The Scheduler Costs Wrapper classes allow the scheduler to access the runtime statistics computer by the Statistics Computing Unit and saved in the Monitoring Database Server. These Java classes ease the job of the scheduler by aggregating and simplifying access to the runtime estimates. These are kept in a type safe (generic) hash map and are extracted atomically from the database. The atomicity needs to be maintained in order for the percentages for all operators extracted to have the sum of 100. For the scheduler, this is not an absolute requirement, but worse schedules

might still appear in rare cases, which might lead to longer total execution times. This wrapper over the statistics database might also be used as a base class for other scheduler cost computing classes if the scheduling algorithm uses other kind of costs than those provided.

5.10 Goal Related Issues

The generality of the solution was a main concern. The solution was tested inside and outside(in large distributed systems) of the Grid environment, where it performed according to specifications. It is expected that reuse and extensibility are more than enough to assure fast development times for applications that choose the proposed monitoring solution.

6 Runtime Estimation

Previous solutions to assigning a cost to each Grid job involved estimating the actual runtime of a job. However, this is a difficult problem to solve, even if that job has been run already, in the past. These difficulties arise from job parameters being different (an image of a much larger size as input would lead to a larger runtime) or because of the dynamic nature of the Grid environment.

6.1 Using Statistical Data to Estimate Runtime for Jobs

Some batch queuing systems require the user to provide job runtime estimates, instead of trying to statistically compute them. Research has show that these estimates are wildly incorrect, with up to 60% of jobs using less than 20% of the estimated time [11]. Many factors contribute to this discrepancy between estimated time and real job execution time. One such factor might be that many jobs crash immediately as execution starts; the inherent difficulty of estimating runtime and the varying load of the systems used can be another.

A contribution of this chapter is proposing new, relative estimates of the runtime of related jobs, in the context of workflows. The proposed solution offers a operator cost computing component which associates a certain cost to each operator used in a certain position in a workflow. The cost assigned to each operator is an estimate of the runtime of the operator divided by the runtime of the workflow and scaled to 100. This information is computed statistically using data from all instances of that particular operator in that particular type of workflow, that were run in the system.

In general, the cost associated with the execution of an operator in the $N+1$ execution of a certain workflow wf should be estimated using a particularization of the following formula: $Cost_{op}^{'wf}(N+1) = f(Cost_{op}^{wf}(1), Cost_{op}^{wf}(2), \ldots, Cost_{op}^{wf}(N))$, where: $Cost_{op}^{wf}(k)$ is the measured cost of the execution of operator named op in

the k-th execution of workflow wf in the system. It is not estimated, but computed, because the running time of the operator and its workflow is known. $Cost_{op}^{lwf}(N+1)$ is an estimate of the cost of the next operator, used in the scheduling of the op operator in the wf workflow. For example, a cost is associated with the first Subtract operator in the EVI workflow type: $Cost_{Sub-1}^{EVI} = 1.25$ [13].

The initial cost of each operator in a workflow can be set by the programmer, but default values are assigned automatically if this is not done. Default values assign the same cost to each operator in the workflow. Although this might not be a good approximation, the statistical engine will update these parameters to more realistic values as soon as the workflow is run a few times.

Notice that the operator cost is in relation only to the runtime of other operators from the same workflow (dependent on the same inputs) and not to those in other workflow. This is an improvement over other estimation methods, because the relationship between runtime of operators in different workflow types is uncertain. Also, this cost is updated with each new workflow execution, which means the cost is adjusted when changes caused by the dynamic nature of the Grid occur (for example, other type of machines are installed, and they execute some operators faster and other slower; this might happen when the IO bandwidth is decreased, but the FLOPS are increased on the new hardware).

Because the number of executions of a workflow can potentially be very large, using data from all the past executions to calculate an estimate can thus cause performance problems. This is why we preferred to use a simpler formula, one that does not need the cost of every execution of that operator. We have chosen to use only the previous estimate and the last actual execution cost to computer the next estimate of the cost: $Cost_{op}^{lwf}(N+1) = f(Cost_{op}^{lwf}(N), Cost_{op}^{wf}(N))$.

Because the updated cost is not needed instantly, it is also feasible to use the complete formula and just choose a particular function f. However, the computing cost is not justified, for reasons discussed below.

Table 1 Relative runtime (out of 10) for the first 5 executions of the EVI workflow

Workflow ID	EVI-135	EVI-136	EVI-137	EVI-138	EVI-139
Input Images	Romania	Japan	Italy	France	South America
Images Size	2480*3508	5800*7200	3933*4717	5200*4000	5940*8520
MultFloat-5	1.03	1.06	0.84	1.02	1.06
MultFloat-8	1.03	1.03	0.90	1.02	1.06
Sub-3	1.61	1.41	1.81	1.58	1.58
AddFloat-11	1.26	1.03	0.90	1.02	0.95
Add-9	1.49	1.70	1.94	1.64	1.60
MultFloat-14	1.03	1.03	0.84	1.07	1.08
Add-12	1.49	1.06	0.84	0.96	1.06
Div-15	1.03	1.06	0.84	0.96	1.06

Considering the simplification above, the problem remains to choose the function f so as to balance the following requirements:

- Quick adaptation to workflow pattern change.
- Good resistance to few statistically aberrant results
- Proven to respect the fundamental characteristic that $\sum_{op} Cost(op) = k$ (The sum of the estimated cost of all operators in a single workflow wf, estimated at any moment, should be equal to k, where k is a fixed constant).

The sum of the costs associated to the actual runtime of the operators for a workflow is considered to be k. This is consistent to a cost that represents a percentage of the workflow runtime, if run on a single machine. We propose two potentially good formulas for runtime estimates, considering the previously mentioned restrictions.

The FPE Formula. The first formula considered has a *fixed percentile adjustment rate*:

$$Cost_{op}^{\prime wf}(N+1) = Cost_{op}^{\prime wf}(N) * (1-p) + Cost_{op}^{wf}(N) * p.$$

Proof of Correctness of FPE Formula. For simplification, we will consider $op_1' \ldots op_n'$ to be the estimated costs for the t execution of a workflow wf. We presume that these operator costs respect the formula $\sum_{op'} Cost(op') = k$. We consider that $op_1 \ldots op_n$ are the actual costs for these operators and we want to prove that $op_1'' \ldots op_n''$ (the estimated costs for the $t+1$ execution of the same workflow), respect the same formula, $\sum_{op''} Cost(op'') = k$. The actual costs, because of the way they are computed, respect the formula $\sum_i Cost(op_i) = k$. Using these terms, the FPE formula can be expressed as follows: $Cost(op_i'') = Cost(op_i') * (1-p) + Cost(op_i) * p$. By adding the previous relation, for each operator in the considered workflow, we get:

$$\sum_{op''} Cost(op'') = \sum_{op'} Cost(op') * (1-p) + \sum_{op} Cost(op) * p$$

By replacing the sums with their values (presented before), $\sum_{op''} Cost(op'') = k * (1-p) + k * p = k$.

The DPE Formula. The second formula considered has a *decreasing percentile adjustment rate*:

$$Cost_{op}^{\prime wf}(N+1) = Cost_{op}^{\prime wf}(N) * \left(1 - \frac{p}{N}\right) + Cost_{op}^{wf}(N) * \frac{p}{N}.$$

Proof of Correctness of DPE formula. The proof is similar with the proof for FPE, but in this case p is variable and equal with $p' = p/N$. By replacing in the previous proof p with p' we get the desired proof.

6.2 Implementation of Runtime Estimates

The estimates are updated using a trigger on the database's table that stores the starting and ending time of operators. When the operator is the last one of its workflow, the percentages of time spent running each operator from that workflow are computed and these values are used to update the runtime estimates. After being updated, the statistics values are sent to the MonALISA Repository for visualization.

7 Testing and Results

The testing of the proposed solution was done on a gLite production cluster from University POLITEHNICA of Bucharest set-up for SEEGRID Project. The cluster is a high performance cluster built around two cores: one with 32 dual-Xeon computers with 2GB RAM and one with 48 P4 HT computers at 3GHz, connected with a Gigabit network. Besides these components there are also storage servers and auxiliary applications, as well as the pre-production cluster (24 P4 computers at 3GHz).

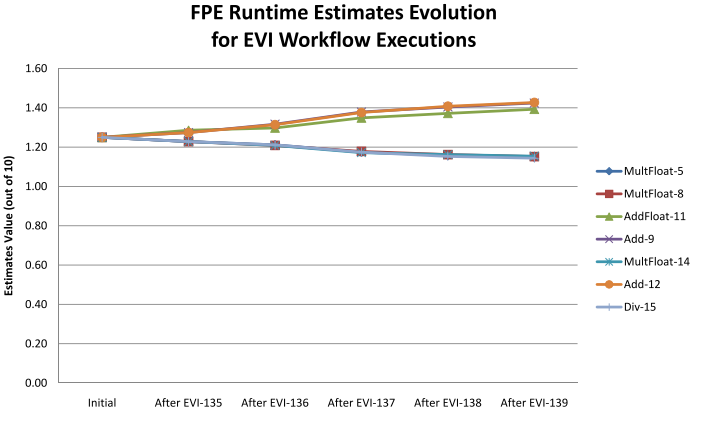

Fig. 6 The FPE estimates get close to their stable values after only a small number of executions of the EVI workflow

7.1 General Improvements

After the monitoring solution was installed in the cluster, the first improvement noticed by the development team was that monitoring information about running jobs was available with several (1-10) minutes earlier from the monitoring system than from the gLite middleware. This might lead to an improvement of job performance

if, for example, the scheduler policy for resubmitting jobs that fail took into account the error identified by the monitoring system and not those provided by gLite. The monitoring solution proved useful for the developers in identifying a bug that allowed the jobs to remain on the worker nodes indefinitely after completion. Also, it helped to show that the worker nodes were not fully stressed during operator execution, so further optimization of the image processing implementation is useful.

Because the monitoring system is not available at the time, a certain class of monitoring data is not available. For example, monitoring during the decompressing part of the script that contains the job that will be run, before the loading of the Java operator that does the processing is not implemented. Thus, detecting missing Java environment is not possible; a workaround might be found by using other MonALISA bindings, such as those for Python, but this would only mean changing one dependency problem with another. A disadvantage of the runtime estimation system is that it presumes identical machines in the cluster where a certain workflow is run; the system adjusts well when replacing all processing nodes, but does not work properly on heterogeneous clusters.

7.2 Reliability

The implementation used features such as: threads for isolation of the monitoring code, UPD to mitigate timeout errors, MonALISA for scalability and redundancy. The tests shows no sign of decreased reliability and, as expressed above, the monitoring system helped identify irreversible and unknown error types.

7.3 Runtime Estimates Testing

Tests were conducted using instances of the EVI operator showed that the estimates tend to stabilize after a few workflows are executed. This is true for tests are run in one job per worker node situation. Some clusters are configured to accept multiple jobs at the same time, which might lead to bad estimates.

Tests performed on the SEEGRID cluster at night (low load) showed almost constant estimates. Daytime tests (heavy load) lead to variations in run times of over 1000% which mean the data from executions that take place under uncontrolled load should not be used to improve the estimates (we disabled the updating of the estimates when the system load is larger than a set limit). The input images for testing the runtime estimates consisted of satellite images of medium size, mentioned in Table 1 and 2. This tables presents the estimation cost for the first five executions of the EVI workflow.

The FPE estimates, presented in Figure 6 get close to their stable values after only a small number of executions of the EVI workflow. This show that we have a short overhead for estimation step and the costs for workflows could be used successfully in a scheduling system.

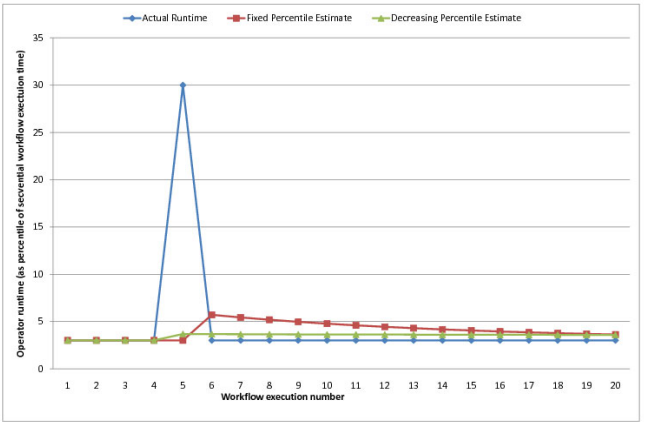

Fig. 7 The evolution of the two proposed estimated of the operator runtime relative to the workflow, in the case of a single execution. DPE has the smallest variation, possibly leading to better scheduling performance than FPE.

Because the test data showed no change in workflow profile, both estimates were quick to adapt to the workflow profile. The DPE ($p = 0.1$) was much slower than FPE, which can be attributed to a low value of p (the influence of the new runtime value is small). A greater value for p would have helped in this case. The DPE tests with $p = 0.3$ confirmed this, but the better adjustment rate proves to be a disadvantage if the first few operators are not representative (typical) of the others, so this makes high values of p usable only if the first executions of newly introduced workflows are run in a controlled environment. Lower p values have the advantage that workflow types can be introduced at any time.

The Fixed Percentile Estimate (FPE) formula has the advantage of a rapid adaptation to changes in the execution profile of a workflow. The disadvantage is that it will not reach a stable level, but will keep alternating, according to the variation of actual execution results (see Figure 7).

The Decreasing Percentile Estimate (DPE) formula leads to better stability considering a stable workflow execution profile (after a number of execution, a new, aberrant execution time will cause less change to the estimate) but this has the disadvantage that if workflow profile do change (an operator execution time changes drastically compared to other operators in the same workflow) adaptation to the new cost will be very slow. Because such changes appear rarely after a workflow is created and it is easy to create another workflow with another name instead of using the same one, the disadvantage becomes less important (see Figure 8).

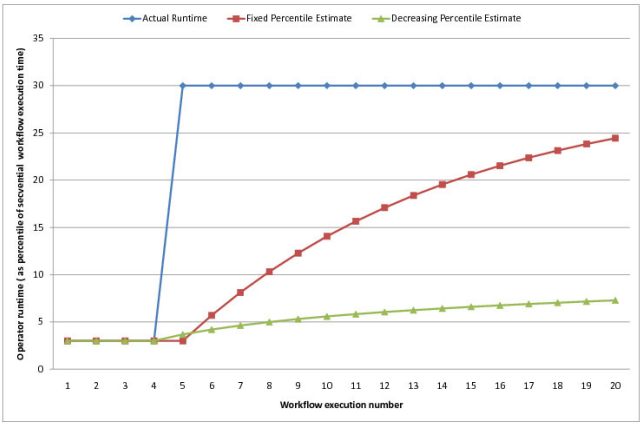

Fig. 8 The evolution of the two proposed estimated of the operator runtime relative to the workflow, in the case of a workflow execution profile change. FPE is the quickest to adapt, possibly leading to better scheduling performance than DPE.

Table 2 The FPE Evolution with $p = 0.1$ for the first five executions of the EVI workflow

	Initial	EVI135	EVI136	EVI137	EVI138	EVI139
MultFloat-5	1.25	1.23	1.21	1.17	1.16	1.15
MultFloat-8	1.25	1.23	1.21	1.18	1.16	1.15
Sub-3	1.25	1.29	1.30	1.35	1.37	1.39
AddFloat-11	1.25	1.25	1.23	1.20	1.18	1.16
Add-9	1.25	1.27	1.32	1.38	1.40	1.42
MultFloat-14	1.25	1.23	1.21	1.17	1.16	1.15
Add-12	1.25	1.27	1.31	1.38	1.41	1.43
Div-15	1.25	1.23	1.21	1.17	1.15	1.14

8 Conclusion

A modern trait in Grid monitoring systems is the increasing number of uses for the collected data. One domain in which this data can have a significant improvement on the performance of Grid application is using the real-time information to compute estimates of runtime which are used to improve scheduling. The proposed estimates proved were successfully used for the workflows tested.

Another such domain is automated error detection systems, which can improve the robustness of Grid by enabling fault recovery mechanisms to be used. Both these aspects can benefit from the particularization of the monitoring system for a workflow-based application: the scheduling performance can be improved through better runtime estimation and the error detection can automatically detect several

types of errors. Of a fundamental importance in building such components for a distributed application is using a real-time monitoring framework such as MonALISA, used for scalability.

The importance and value of this project rest not only in drawing the conclusions mentioned above, but also in more concrete facts. The solution is used in the SEE-GRID-SCI project and will be a part of the satellite image processing engine that is being built.

References

1. Zanikolas, S., Sakellariou, R.: A taxonomy of grid monitoring systems. Future Gener. Comput. Syst. 21(1), 163–188 (2005)
2. Bacu, V., Gorgan, D.: Graph Based Evaluation of Satellite Imagery Processing over Grid. In: ISPDC 2008: Proceedings of the 2008 International Symposium on Parallel and Distributed Computing, pp. 147–154. IEEE Computer Society, Washington (2008)
3. Gorgan, D., Stefanut, T., Bacu, V.: Grid Based Training Environment for Earth Observation. In: GPC 2009: Proceedings of the 4th International Conference on Advances in Grid and Pervasive Computing, pp. 98–109. Springer, Heidelberg (2009)
4. Simion, B., Leordeanu, C., Pop, F., Cristea, V.: A Hybrid Algorithm for Scheduling Workflow Applications in Grid Environments (ICPDP). In: Meersman, R., Tari, Z. (eds.) OTM 2007, Part II. LNCS, vol. 4804, pp. 1331–1348. Springer, Heidelberg (2007)
5. Cirstoiu, C.C., Grigoras, C.C., Betev, L.L., Costan, A.A., Legrand, I.C.: Monitoring, accounting and automated decision support for the alice experiment based on the MonALISA framework. In: GMW 2007: Proceedings of the 2007 workshop on Grid monitoring, pp. 39–44. ACM, New York (2007)
6. Xhafa, F., Carretero, J., Barolli, L., Durresi, A.: Immediate mode scheduling in grid systems. Int. J. Web Grid Serv. 3(2), 219–236 (2007)
7. Streit, A.: A Self-Tuning Job Scheduler Family with Dynamic Policy Switching. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2002. LNCS, vol. 2537, pp. 1–23. Springer, Heidelberg (2002)
8. Xhafa, F., Abraham, A.: Computational models and heuristic methods for Grid scheduling problems. Future Gener. Comput. Syst. 26(4), 608–621 (2010)
9. Iordache, G., Boboila, M., Pop, F., Stratan, C., Cristea, V.: A decentralized strategy for genetic scheduling in heterogeneous environments. Multiagent Grid Syst. 3(4), 355–367 (2007)
10. Armstrong, R., Hensgen, D., Kidd, T.: The Relative Performance of Various Mapping Algorithms is Independent of Sizable Variances in Run-time Predictions. In: HCW 1998: Proceedings of the Seventh Heterogeneous Computing Workshop, p. 79. IEEE Computer Society, Washington (1998)
11. Cirne, W., Berman, F.: A comprehensive model of the supercomputer workload. In: WWC 2001: Proceedings of the Workload Characterization, IEEE International Workshop on WWC-4, pp. 140–148. IEEE Computer Society, Washington (2001)
12. Gorgan, D., Stefanut, T., Bacu, V.: Grid Based Training Environment for Earth Observation. In: Abdennadher, N., Petcu, D. (eds.) Proceedings of the 4th International Conference on Advances in Grid and Pervasive Computing, Geneva, Switzerland, May 04-08. LNCS, vol. 5529, pp. 98–109. Springer, Heidelberg (2009)

13. Mihon, D., Bacu, V., Meszaros, R., Gelybo, G., Gorgan, D.: Satellite Image Interpolation and Analysis through GreenView Application. In: Proceedings of the 2010 International Conference on Complex, Intelligent and Software Intensive Systems, February 15-18. CISIS, pp. 514–519. IEEE Computer Society, Washington (2010)
14. Bacu, V., Gorgan, D.: Graph Based Evaluation of Satellite Imagery Processing over Grid. In: Proceedings of the 2008 International Symposium on Parallel and Distributed Computing, July 01-05. ISPDC, pp. 147–154. IEEE Computer Society, Washington (2008)
15. Zhang, L., Furumi, S., Muramatsu, K., Fujiwara, N., Daigo, M., Zhang, L.: A new vegetation index based on the universal pattern decomposition method. Int. J. Remote Sens. 28(1), 107–124 (2007)
16. Kendall, W., Glatter, M., Huang, J., Peterka, T., Latham, R., Ross, R.: Terascale data organization for discovering multivariate climatic trends. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC 2009, Portland, Oregon, November 14-20, pp. 1–12. ACM, New York (2009)