ASKALON: A Grid Application Development and Computing Environment

Thomas Fahringer, Radu Prodan, Rubing Duan, Francesco Nerieri, Stefan Podlipnig,
Jun Qin, Mumtaz Siddiqui, Hong-Linh Truong, Alex Villazón, Marek Wieczorek
Institute of Computer Science, University of Innsbruck, Technikerstraße 21A, A-6020 Innsbruck, Austria
Email: tf@dps.uibk.ac.at

***Abstract*— We present the ASKALON environment whose goal is to simplify the development and execution of workflow applications on the Grid. ASKALON is centered around a set of high-level services for transparent and effective Grid access, including a Scheduler for optimized mapping of workflows onto the Grid, an Enactment Engine for reliable application execution, a Resource Manager covering both computers and application components, and a Performance Prediction service based on training phase and statistical methods. A sophisticated XML-based programming interface that shields the user from the Grid middleware details allows the high-level composition of workflow applications. ASKALON is used to develop and port scientific applications as workflows in the Austrian Grid project. We present experimental results using two real-world scientific applications to demonstrate the effectiveness of our approach.**

I. Introduction

A computational Grid is a set of hardware and software resources that provide seamless, dependable, and pervasive access to high-end computational capabilities. Resources on the Grid are usually geographically distributed and commonly belong to different administrative domains. While Grid infrastructures can provide massive compute and data storage power, it is still an art to effectively harness the power of Grid computing. Most existing Grid application development environments provide the application developer with a non-transparent Grid. Commonly application developers are explicitly involved in tedious tasks such as selecting software components deployed on specific sites, mapping applications onto the Grid, or selecting appropriate computers for their applications. Moreover, many programming interfaces are either implementation technology specific (e.g. based on Web services [23]) or force the application developer to program at a low-level middleware abstraction (e.g. start task, transfer data [2], [15]). In this paper we describe the ASKALON Grid application development and computing environment whose final goal is to provide an invisible Grid to the application developers.

In ASKALON (see Fig. 1), the user composes Grid workflow applications at a high-level of abstraction using an XML-based language (AGWL) that shields the application developer from the Grid. The AGWL representation of a workflow is then given to the middleware services (run-time system) for scheduling and reliable execution.

ASKALON provides the following set of middleware services that support the execution of scientific workflows on the Grid. The *Resource Manager* service is responsible for negotiation, reservation, and allocation of resources, as well as automatic deployment of services required to execute Grid applications. The *Enactment Engine* service targets reliable and fault tolerant execution of workflows through techniques such as checkpointing, migration, restart, retry, and replication. *Performance analysis* [24] supports automatic instrumentation and bottleneck detection (e.g. excessive synchronization, communication, load imbalance, inefficiency, or non-scalability) within Grid workflow executions. The *Performance Prediction* service currently focuses on estimating execution times of workflow activities through training phase and statistical methods using the Performance analysis service. The *Scheduler* is a service that determines effective mappings of single or multiple workflow applications onto the Grid using graph-based heuristics and optimization algorithms on top of the Performance Prediction and Resource Manager services. Additionally, the Scheduler aims to provide Quality of Service (QoS) by dynamically adjusting the optimized static schedules to meet the dynamic nature of Grid infrastructures through execution contract monitoring [18].

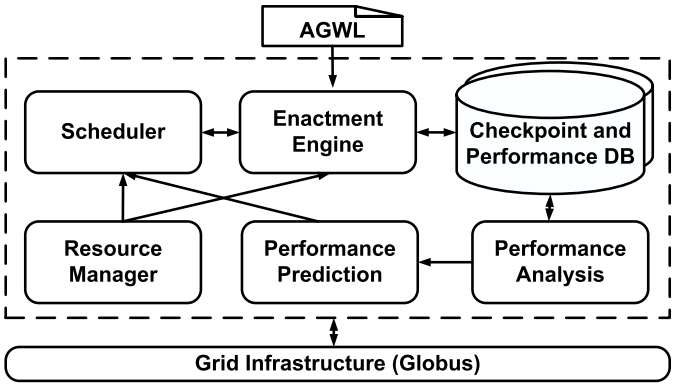


Fig. 1. The ASKALON architecture.

Due to space limitations we limit the scope of this paper to a set of distinguishing new features of ASKALON. Descriptions of other parts can be found at [10].

This paper is organized as follows. In the next section we describe the programming interface of ASKALON. In Sections III – VI we introduce the main middleware services: the Scheduler, the Enactment Engine, the Resource Manager,

and the Performance Prediction service. Section VII presents experimental results involving two real-world Grid workflow applications. Related work is summarized in Section VIII. In Section IX we conclude the paper with an outlook to the future work.

II. Grid Workflow Composition with AGWL

In contrast to related work [2], [8], [23], [21], [6], ASKALON enables the description of workflow applications at a high-level of abstraction that shields the user from the middleware complexity and dynamic nature of the Grid.

We designed the XML-based *Abstract Grid Workflow Language* (AGWL) [11] for composing a workflow application from atomic units of work called *activities* interconnected through control flow and data flow dependencies. Activities are represented at two abstract levels: *activity types* and *activity deployments*. An *activity type* is a simplified abstract description of functions or semantics of an activity, whereas an *activity deployment* (not seen at the level of AGWL but resolved by the underlying Resource Manager – see Section V) refers to an executable or a deployed Web service and describes how they can be accessed and executed on the Grid.

In contrast to most existing work, AGWL is not bound to any implementation technology such as Web services. The control-flow constructs include `sequences`, Directed Acyclic Graphs (`dag`), `for`, `forEach`, `while` and `do-while` loops, `if` and `switch` constructs, as well as more advanced constructs such as `parallel` activities, `parallelFor` and `parallelForEach` loops, and `collection` iterators. In order to modularize and reuse workflows, so called sub-workflows can be defined and invoked. Basic data flow is specified by connecting input and output ports between activities, while more advanced data flow constructs include access to abstract data repositories.

Optionally, the user can specify `properties` and `constraints` for activities and data flow dependencies that provide functional and non-functional information to the run-time system for optimization and steering of the Grid workflow execution. Properties define additional information about activities or data links, such as computational or communication complexity, or semantic description of workflow activities. Constraints define additional requirements or contracts to be fulfilled by the run-time system that executes the workflow application, such as the minimum memory necessary for an activity execution, or the minimum bandwidth required on a data flow link.

The AGWL representation of a workflow serves as input to the ASKALON run-time middleware services (see Fig. 1).

III. Scheduler

The Scheduler service prepares a workflow application for execution on the Grid. It processes the workflow specification described in AGWL, converts it to an executable form, and maps it onto the available Grid resources.

The scheduling process starts when the Enactment Engine sends a scheduling request with a workflow description. The

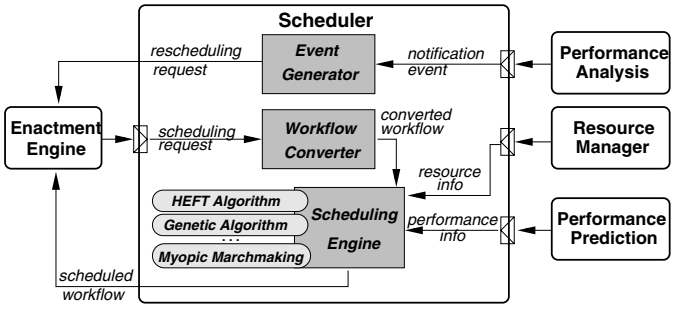


Fig. 2. The Scheduler architecture.

workflow consists of nodes representing activity types connected through control and data flow dependencies, as well as overall workflow input and output data. The Scheduler uses the Resource Manager (see Section V) to retrieve the current status of the Grid resources and to determine available activity deployments that correspond to the workflow activity types. In addition, the queries submitted by the Scheduler to the Resource Manager can contain constraints that must be honored, such as processor type, minimum clock rate, or operating system. The Performance Prediction service supplies predicted activity execution times and data transfer times required by the performance-driven scheduling algorithms.

The Scheduler consists of three main components, as illustrated in Fig. 2: workflow converter, scheduling engine, and event generator, which we describe in more detail in the following.

The *workflow converter* resolves all the ambiguities and transforms sophisticated workflow graphs into simple Directed Acyclic Graphs (DAGs – usually through loop unrolling) on which existing graph scheduling algorithms can be applied. Many transformations are based on assumptions which may change during the execution. Assumptions are made for conditions (e.g. `while`, `if`, `switch`) and parameters (e.g. number of parallel loop iterations) that cannot be evaluated statically before the execution begins. Transformations based on correct assumptions can imply substantial performance benefits, particularly if a strong imbalance in the workflow is predicted (see Section VII). Incorrect assumptions require appropriate run-time adjustments such as undoing existing optimizations and rescheduling based on the new Grid information available.

The *scheduling engine* is responsible for the actual mapping of a converted workflow onto the Grid. It is based on a modular architecture, where different DAG-based scheduling heuristics can be used interchangeably. The algorithms with varying accuracy and complexity are based on different metrics as optimization goals. We have currently incorporated three scheduling algorithms: Heterogeneous Earliest Finish Time (HEFT) [26], a genetic algorithm [18], and a ”myopic” just-in-time algorithm acting like a resource broker, similar to the Condor matchmaking mechanism used by DAGMan [6]. All algorithms receive as input two matrices representing the predicted execution time of every activity instance on each compute architecture, respectively the predicted transfer time

of each data dependency link on every Grid site interconnection network, and deliver a Grid schedule.

After the initial scheduling, the workflow is executed based on the current mapping until the execution finishes or any interrupting event occurs. The *event generator* module uses the Performance analysis service to monitor the workflow execution and detect whether any of the initial assumptions, also called *execution contracts*, has been violated. Execution contracts that we are currently monitor include structural assumptions made by the workflow converter, external load on processors, processors no longer available, congested interconnection networks, or new Grid sites available. In case of a contract violation, the Scheduler sends a rescheduling event to the Enactment Engine, which generates and returns to the Scheduler a new workflow based on the current execution status (by excluding the completed activities and including the ones that need to be re-executed). We have formally presented this approach in detail in [18].

IV. Enactment Engine

The Enactment Engine is the central service of the ASKALON middleware responsible for executing a workflow application based on the Grid mapping decided by the Scheduler. The main tasks performed by the Enactment Engine is to coordinate the workflow execution according to the control flow constructs (i.e. `sequence`, `if`, `switch`, `while`, `for`, `dag`, `parallel`, `parallelFor`) and to effectively resolve the data flow dependencies (e.g. activity arguments, I/O file staging, high bandwidth third-party transfers, access to databases) specified by the application developer in AGWL.

Among the novel features, the Enactment Engine provides flexible management of large collections of intermediate data generated by hundreds of parallel activities that are typical to scientific workflows (see Fig. 4). Additionally, it provides a mechanism to automatically track data dependencies between activities and performs static and run-time workflow optimizations, including archiving and compressing of multiple files to be transferred between two Grid sites, or merging multiple activities to reduce the job submission and polling for termination overheads.

The Enactment Engine provides fault tolerance at three levels of abstraction:

1) activity-level through retry and replication;
2) control flow-level using light-weight workflow checkpointing and migration (described later in this section);
3) workflow-level based on alternative task, workflow-level redundancy, and workflow-level checkpointing.

Checkpointing and recovery are fundamental techniques for saving the application state during normal execution and restoring the saved state after a failure to reduce the amount of lost work. The Enactment Engine provides two types of checkpointing mechanisms described in the following.

Light-weight workflow checkpointing saves the workflow state and URL references to intermediate data (together with additional semantics that characterize the physical URLs) at customizable execution time intervals. The light-weight checkpoint is very fast because it does not back-up the intermediate data. The disadvantage is that the intermediate data remains stored on possibly unsecured and volatile file systems. Light-weight workflow checkpointing is typically used for immediate recovery during one workflow execution.

Workflow-level checkpointing saves the workflow state and the intermediate data at the point when the checkpoint is taken. The advantage of the workflow-level checkpointing is that it completely saves backup copies of the intermediate data into a checkpoint database, such that the execution can restored and resumed at any time and from any Grid location. The disadvantage is that the checkpointing overhead grows significantly for large intermediate data.

Let $W = (AS, CF, DF)$ represent a Grid workflow application, where AS denotes the set of activities of the workflow, $CF = \{(A_{from}, A_{to}) \mid A_{from}, A_{to} \in AS\}$ represents the set of control flow dependencies, and $DF = \{(A_{from}, A_{to}, Data) \mid A_{from}, A_{to} \in AS\}$ the data flow dependencies, where $Data$ denotes the workflow intermediate data, usually instantiated by a set of files and parameters. Let $State : AS \rightarrow \{Executed, Unexecuted\}$ denote the execution state function of an activity. We formally define a *workflow checkpoint* as a set of tuples:

$$CKPT_W = \{(A, State(A), Data) \mid \forall\ A, A_{from} \in AS \wedge State(A) = Unexecuted \wedge State(A_{from}) = Executed \wedge (A_{from}, A, Data) \in DF\}.$$

As we can notice, there are two possible options for the checkpointed state of an executing activity for which we propose three solutions:

1) We let the job run and regard the activity as *Unexecuted*;
2) We wait for the activity to terminate and set the state to *Executed*, if the execution was successful. Otherwise, we set the state to *Unexecuted*. Both solutions are not obviously perfect, therefore, we propose a third compromise option that uses the predicted execution time of the job, as follows:
3) Delay the checkpoint for a significantly shorter amount of time CD and compute the state of an activity A using the following formula:

$$State(A) = \begin{cases} Unexecuted, & PT_A - CD \geq ET_A; \\ Executed, & PT_A - CD < ET_A, \end{cases}$$

where PT_A is the predicted execution time of the activity A as reported by the Performance Prediction service and ET_A is the elapsed execution time of the activity from the beginning until the current time. This solution reduces the checkpoint overhead and lets the checkpoint complete within a shorter time frame.

V. Resource Manager

Resource management is a key concern for the implementation of an effective Grid middleware and for shielding application developers from its low-level details. The Resource

Manager renders the boundaries of Grid resource management and brokerage and provides Grid resource discovery, advanced reservation and virtual organization-wide authorization along with a dynamic registration framework for the Grid activities [19], [20]. The Resource Manager covers both physical resources, including processors, storage devices, and network interconnections, as well as logical resources comprising Grid/Web services and executables.

Based on Scheduler requests, the Resource Manager discovers resources or software activities, performs user authorization to verify resource accessibility, optionally makes a reservation, and returns the result. The result could be a list of resources along with their specifications, a list of software components, or a reservation ticket, depending on the request type. In case of a failure, a Resource Manager can interact with other Resource Managers distributed in the Grid to recursively discover and allocate required resources. Moreover, the Resource Manager monitors the allocated resources and propagates exceptional situations to the client. The Resource Manager can also work as a co-allocation manager.

Grid resource discovery and matching is performed based on the constraints provided by the Scheduler in the form of a resource request (see Section III). The Resource Manager can be configured with one or more Monitoring and Discovery Services (MDS) [12] (of Globus versions 2 and 4) and the Network Weather Service (NWS) [25].

Advanced reservation of the Grid resources (including computers and software components) based on the constraints provided by the resource requester is a distinguishing feature of the Resource Manager. The Scheduler can negotiate for reservation based on time, cost, and QoS models. The essential attributes of a reservation include resource contact information, time frame, and resource requester and provider constraints. The acquisition of reserved resources by the Enactment Engine is only possible by providing a valid user credential based on which the reservation was made, or a valid reservation ticket.

The Resource Manager also provides a distributed framework for dynamic registration, automatic deployment, and on-demand provision of workflow activities. Automatic deployment is performed based on Autoconf / AutoBuild [3] and Expect [9] tools. The framework provides an effective mapping between high-level application descriptions (activity types) and actual installations (activity deployments) on specific Grid sites. Activity types are described in a hierarchy of abstract and concrete types. Concrete types may have activity deployments which are shielded from the Grid application developer. Fig. 3 illustrates a real-world example of a concrete activity type called *POVray* [17] which inherits generic *rendering* and *imaging* types. The activity type *POVray* has two activity deployments: a legacy executable *povray* and a Web Services Resource Framework (WSRF) [13]-compliant service called *WS-POVray*, both visible only internally to the registration framework. The registration framework performs on-demand installation of these activity deployments and maps them automatically to the activity types, thus shields the Grid from the application developers.

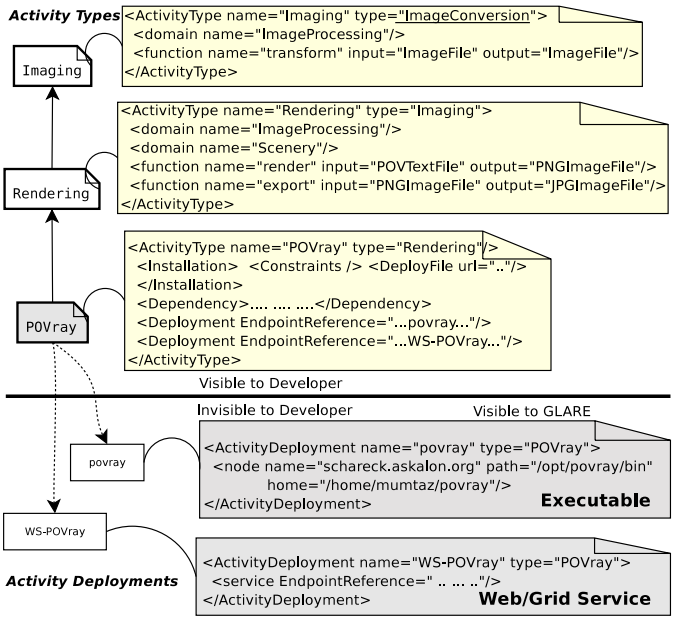
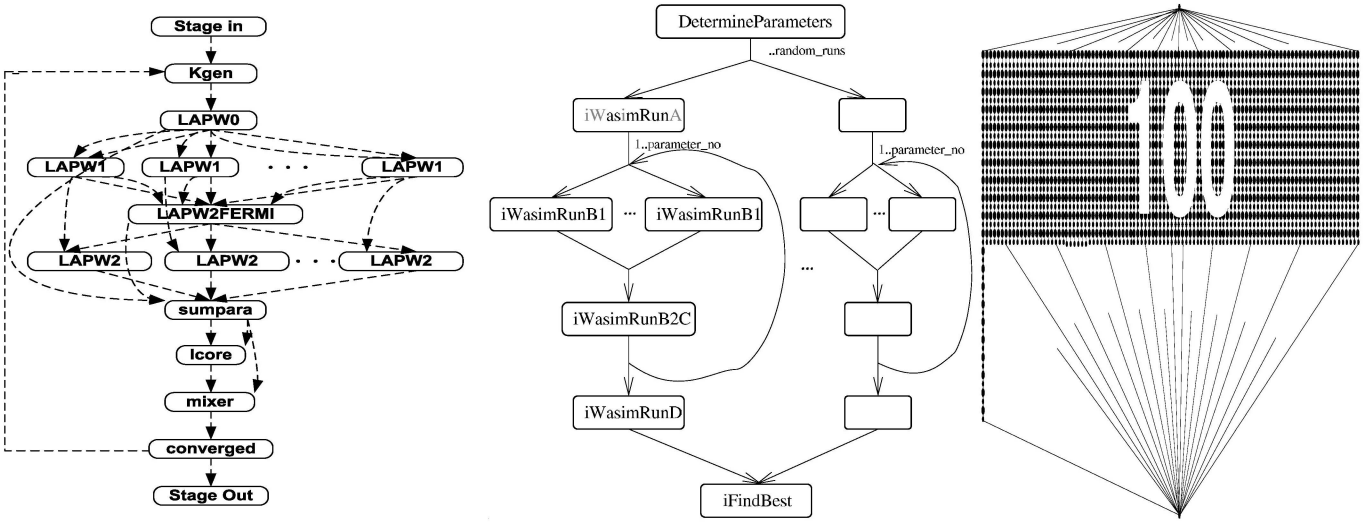


Fig. 3. Activity type hierarchy and type to deployment mapping.

VI. Performance Prediction

ASKALON provides a service for predicting the time required by activity executions and data transfers. Predicting the execution time of an activity on a given computer depends on various factors, like the problem size. While there exist several proposals for the execution time estimation problem, the most general form is statistical prediction based on historical data which relies on past measurements for different input parameter values on different Grid computers. Furthermore, such techniques are generic and do not need any direct knowledge of the internals of an algorithm or computer.

ASKALON employs active history-based predictions by introducing a training phase for each workflow application. Initially, all activity types are selected which usually represent a fraction of the total number of activity instances (i.e. number of activity executions) of a scientific workflow. For each activity type, different parameter combinations on different Grid computers are tested (usually all in parallel to minimize the duration of the training phase) and the execution times are stored in a performance database. This technique has the advantage that the quality of the data stored is improved compared to the passive prediction methods. In the latter case, the lack of measurements must be compensated by additional interpolation. To further minimize the training phase overhead, we divide it in two phases. The first phase finds the minimum number of measurements for each parameter value (i.e. minimizing the introduced error according to some threshold criterion) that are needed for an accurate prediction on a specific computer. In the second phase, we measure the execution time on all the available computer architectures on the Grid with the reduced number of measurements.



(a) The WIEN2k workflow. (b) The Invmod workflow. (c) Large unbalanced Invmod workflow.

Fig. 4. Real-world workflow applications.

VII. Experiments

We have implemented the ASKALON services described in this paper on top of the Globus toolkit [12] as the underlying infrastructure. Furthermore, ASKALON is used as the main application development and computing environment in the Austrian Grid project [1]. In this section we demonstrate several experiments that evaluate the Performance Prediction service, the Scheduler, the Enactment Engine, and the Resource Manager of ASKALON. Our experiments are centered around two real-world applications, executed on a subset of the Austrian Grid testbed depicted in Table I.

Site	*#*	*CPU*	*GHz*	*Job Manager*	*Location*
altix1.jku	16	Itanium 2	1.6	Fork	Linz
gescher.vcpc	16	Pentium 4	3	PBS	Vienna
altix1.uibk	16	Itanium 2	1.6	Fork	Innsbruck
schafberg	16	Itanium 2	1.6	Fork	Salzburg
agrid1	10	Pentium 4	1.8	PBS	Innsbruck
arch19	10	Pentium 4	1.8	PBS	Innsbruck

TABLE I
The Austrian Grid testbed.

Firstly, WIEN2k is a program package for performing electronic structure calculations of solids using density functional theory, based on the full-potential (linearized) augmented plane-wave ((L)APW) and local orbital (lo) method [4]. We have ported the WIEN2k application onto the Grid by splitting the monolithic code into several course-grain activities coordinated in a workflow (see Fig. 4(a)). The `LAPW1` and `LAPW2` activities can be solved in parallel by a fixed number of so-called *k-points*. A final activity `converged` applied on several output files tests whether the problem convergence criterion is fulfilled. The number of recursive loops is statically unknown.

Secondly, Invmod is a hydrological application for river modeling which has been designed for inverse modeling calibration of the WaSiM-ETH program [22]. Invmod has two levels of nested parallelism with variable number of inner loop iterations that depends on the actual convergence of the optimization process (see Fig. 4(b)). Fig. 5 gives a sample excerpt from the Invmod AGWL representation.

```
<agwl name="invmod"
  <dataIn name="nRuns" type="xs:integer" />
  <dataIn name="isFinish" type="xs:boolean" />
  <activity name="DeterParas" type="...">
    <dataIn name="nRunsIn" source="invmod/nRuns" />
    <dataOut name="numRuns" />
    <dataOut name="numParas" />
  </activity>
  <parallelFor name="parallelFor1">
    <loopCounter name="index" from="1"
                 to="DeterParas/numRuns"/>
    <loopBody>
      <activity name="WasimA" ... />
      <while name="whileLoop">
        <dataIn name="finish"
                source="invmod/isFinish"
                loopSource="WasimB2C/isFinish" />
        <condition> finish != true </condition>
        <loopBody>
          <parallelFor name="parallelFor2" .../>
          <activity name="WasimB2C" ... >
            <dataOut name="isFinish" />
          </activity>
        </loopBody>
        <dataOut name="dataTarball" ... />
      </while>
      <activity name="WasimD" ... />
    </loopBody>
    <dataOut name="dataTarball" type="collection"
             source="WasimD/wasimDOutput" />
  </parallelFor>
  <activity name="FindBest" ... />
  <dataOut name="result" .../>
</agwl>
```

Fig. 5. Invmod AGWL Excerpt

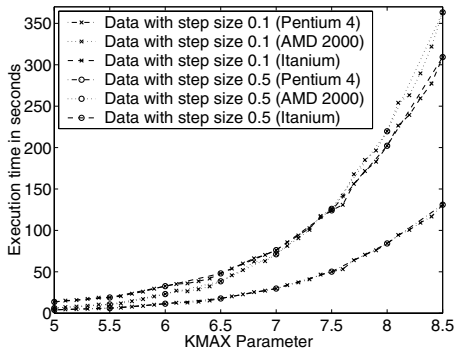
A. Prediction Service

We illustrate the applicability of our prediction approach for the WIEN2k application. We chose `LAPW1` in our example because it is computationally the most expensive activity type. The problem size and the execution time of all WIEN2k activities are influenced by one parameter called `KMAX` which represents the number of plane waves used (i.e. the size of the matrix to be diagonalized). We ran `LAPW1` with a `KMAX` parameter range between 5.0 and 8.5. We show in Fig. 6(a) the execution times for a parameter step of size 0.1 and 0.5 on some of the computer architectures from our Grid environment. We choose the minimum value from 10 different executions for each parameter value to obtain a lower bound for the execution times. The plots in this figure demonstrate that a reduction to one fifth of the original number of experiments through interpolation hardly changes the measured execution time. We thus can provide predictions at much smaller costs.

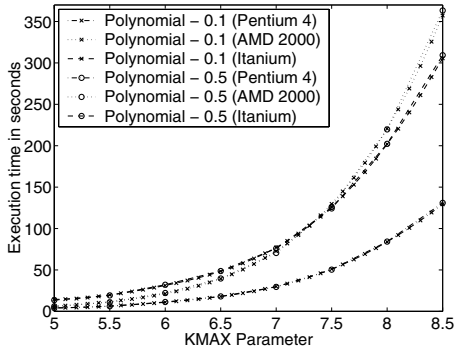
Our actual research concentrates on simple heuristics for minimizing the number of measurements needed. These heuristics are influenced by an error criterion that, for example, should drop below some threshold for providing reasonable accurate predictions.

The measured execution times are obtained in a training phase, stored in a database, and used for statistical predictions. Currently, we use polynomial fitting (of degree five) to obtain a model for our prediction. This model is used when the Scheduler asks for estimated execution times of a specific activity deployment (i.e. activity type plus computer architecture) with a specific parameter value (e.g. `LAPW1` with parameter 8.2). Fig. 6(b) tries to answer an interesting question about how the reduced number of experiments influences the polynomial fitting process. From this example we can see that the reduced number of measurements does not harm the model building process and delivers results similar to the original measurements. We will investigate in the future the ability to predict beyond the training phase range. At present we base our prediction on a useful maximal parameter range.

For data transfer predictions we use an approximation technique. From the training phase we know the size of the output data that is produced by an activity. If this output is the input to another activity scheduled on a different Grid site, we use the data size and the actual available bandwidth between the involved sites measured using NWS [25] to approximate the transfer time. We realized from our test runs that this procedure of prediction is not very accurate. Due to different heterogeneous overheads (e.g. Grid middleware, NFS overhead, multiple parallel connections with different transfer times), the network transfer time prediction is a very difficult problem. Currently, we are using pessimistic assumptions that force the Scheduler to perform Grid mappings with good data locality (e.g. `LAPW1` and `LAPW2` k-points in Fig. 4(a) that have a considerable data dependency). Therefore, future research will investigate more elaborated techniques for data transfer time prediction.



(a) Execution times.



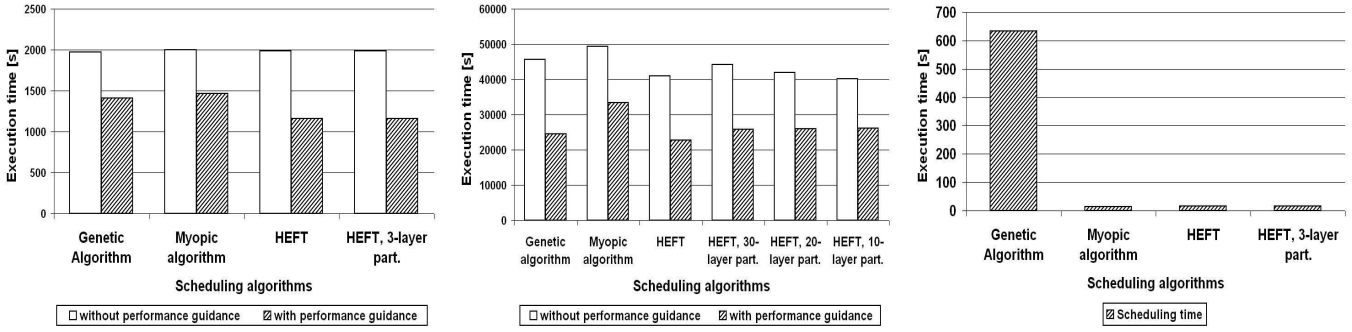
(b) Polynomial fitting.

Fig. 6. Performance prediction experiments.

B. Scheduling

For testing the Scheduler, we examined the WIEN2k and Invmod applications which represent two different classes of workflows considering the balance between the parallel branches. We considered a WIEN2k workflow with one iteration of the outermost loop, and 250 parallel activities in each of the two parallel sections (see Fig. 4(a)). The Invmod workflow represents a class of *strongly unbalanced workflows*, where some of the parallel iterations are considerably longer than the others. Fig. 4(c) displays a sample Invmod workflow generated by the workflow converter using a loop unrolling transformation (based on static speculative assumptions), where one parallel iteration is significantly longer than the other 99 iterations. The goal of our experiments was to compare three different heuristic algorithms supported by the Scheduler: the HEFT algorithm [26], a genetic algorithm [18], and a simple myopic algorithm similar to a resource broker such as the Condor DAGMan [6]. We also compared a full-graph scheduling strategy (i.e. schedule all the workflow activities in advance) against a workflow partitioning approach which resembles the one applied in the Pegasus project [7]. Partitioning means the division of a workflow into a set of sub-workflows (usually of a certain depth that is decided ad-hoc), that are executed in sequence. We performed the partitioning with various depths (i.e. 3, 10, 20 and 30) in different experiments to observe the difference in the results obtained.

The results depicted in Fig. 7 show that optimization algorithms such as HEFT and genetic search produce substantially



(a) WIEN2k execution time. (b) Invmod execution time. (c) WIEN2k scheduling time.

Fig. 7. Scheduling experimental results.

better schedules than the myopic matchmaking. HEFT is also superior to the genetic algorithm, since it is a workflow-specific heuristic highly suited to heterogeneous environments such as the Grid. The full-graph scheduling approach produces better results than the workflow partitioning strategy [7], the most obvious case being the strongly unbalanced Invmod workflow (see Fig. 7(b)). We can also notice that the genetic algorithm meta-heuristic executes two orders of magnitude longer than the others (see Fig. 7(c)).

C. Enactment Engine

In this section we present experimental results that evaluate the performance of the Enactment Engine for reliable execution of the WIEN2k workflow using the optimized mapping delivered by the Scheduler.

We have first investigated the scalability of a relatively modest problem (100 parallel k-points) by incrementally increasing the Grid size from one to six distributed Grid sites (see Table I. Fig. 8(a) shows that this modest WIEN2k problem size manages to scale until three distributed Grid sites because of a network bottleneck on the fourth Itanium site (i.e. schafberg in Salzburg).

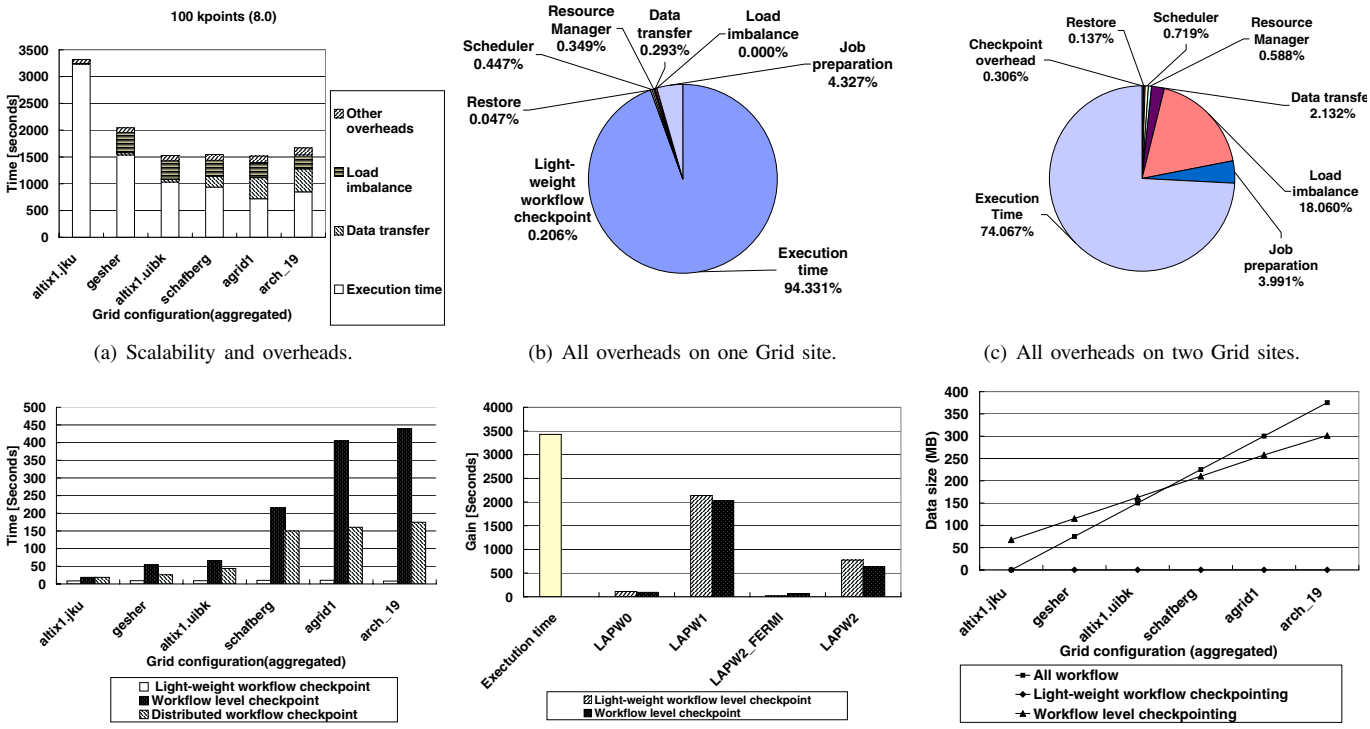
The distributed Grid-wide execution, the interaction with the ASKALON middleware services, as well as the fault tolerance mechanisms applied during the distributed execution, represent sources of performance overheads (losses) that delay the overall completion time of Grid applications. We configured the Enactment Engine to perform a light-weight checkpoint after each main phase of the WIEN2k execution: `LAPW0`, `LAPW1`, and `LAPW2`. In Fig. 8(b) and 8(c) we illustrate the performance overhead breakdown for one Grid site and two Grid site executions, which we describe in the following based on the importance of each overhead.

The *data transfer* overhead represented by third party (GridFTP [12]) file transfers naturally increases with the number of Grid sites, especially when the fourth Itanium site is added. The overhead due to *load imbalance* upon synchronization (see activity `LAPW2_FERMI` in Fig. 4(a)) increases with the number of sites, mainly because of heterogeneity. We define the load imbalance as the difference between the maximum and the average termination time of the activities in a workflow parallel section (e.g. `LAPW1` and `LAPW2` in Fig. 4(a)). The rest of the overheads are so small that we aggregate them as *other overheads* in Fig. 8(a). The *workflow preparation* overhead for creating workflow-level directory structures, as well as other global environment set-up tasks required by legacy applications, obviously increases with the number of Grid sites. The *job preparation* overhead (i.e. the time required to uncompress data archives or create job specific remote directory structures), in contrast, decreases with the number of Grid sites, since more activity instances can be executed in parallel. In case of smaller number of Grid sites, some parallel activity instances have to be serialized (to avoid processor sharing), which explains the high job preparation overhead (e.g. for one Grid site). The *Grid middleware* overhead, including the interaction with the Scheduler and the Resource Manager, remains relatively constant since it is done once using the same algorithms for every individual execution. The overhead due to *checkpointing* is negligible.

Fig. 8(d) compares the overheads of the light-weight workflow checkpointing and the workflow-level checkpointing for a centralized and a distributed checkpoint database. The overhead of the light-weight workflow checkpointing is very low and relatively constant, since it only stores the workflow state and URL references to the intermediate data. The overhead of the centralized workflow-level checkpointing increases with the number of Grid sites due to a larger intermediate data volume to be transferred to the checkpoint database. For a distributed checkpoint database, the workflow-level checkpointing overhead is much lower, since using a local repository on every site where the intermediate data is residing eliminates the wide area network transfers.

Fig. 8(e) presents the checkpoint gains obtained for a single site workflow execution. We define the *gain* as the difference between the timestamp when the last checkpoint is performed minus the timestamp of the previous checkpoint. The biggest gains are obtained after checkpointing the parallel regions `LAPW1` and `LAPW2`. The gain for the workflow-level checkpointing is lower, since it subtracts the time required to copy the intermediate data to the checkpoint database.

Fig. 8(f) shows that the size of the data checkpointed at the workflow level is bigger than the overall size of intermediate



(a) Scalability and overheads.

(b) All overheads on one Grid site.

(c) All overheads on two Grid sites.

(d) Checkpointing overhead comparison.

(e) Checkpoint gains.

(f) Size of data transferred.

Fig. 8. Enactment Engine experimental results.

data transferred for a small number of Grid sites (up to three, when scalability is achieved). Beyond four Grid sites, the size of the intermediate data exceeds the workflow-level checkpoint data size. The data size of the light-weight workflow checkpoint is insignificant.

D. Resource Manager

In this section we present experiences on using the Resource Manager for allocation of resources and automatic deployment of workflow activities.

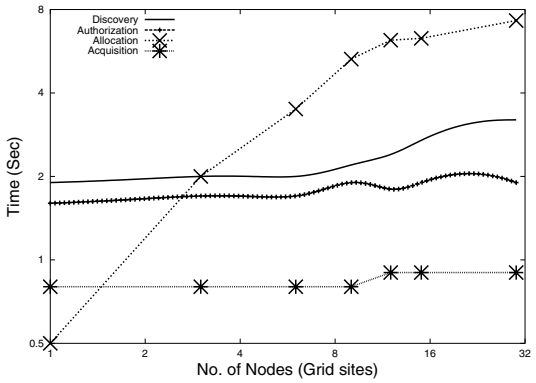
When serving a resource allocation request from the Scheduler, the Resource Manager performs four operations in a single transaction: *discovery*, *authorization*, *allocation*, and *acquisition*. All these operations introduce an overhead to the workflow execution, as already presented in Section VII-C. In the following, we further study and breakdown this overhead by benchmarking a series of requests which vary both the total number of required resources and their attributes. We measure the time for each request by starting a timer in the client program immediately before invoking the Resource Manager and stopping it upon the successful acquisition of the resource ensemble. The result of this experiment depicted in Fig. 9(a) shows that allocation is the most expensive overhead, which increases with the number of resources, as it involves negotiation for advanced reservation.

One of the important feature of the Resource Manager is the automatic deployment of software activities, implemented either as legacy applications or as WSRF services. We have evaluated this feature and calibrated the deployment overheads for two real world scientific applications (i.e. WIEN2k and Invmod), as well as a sample WSRF service (i.e. Counter). We have used two methods to perform the automatic deployment:

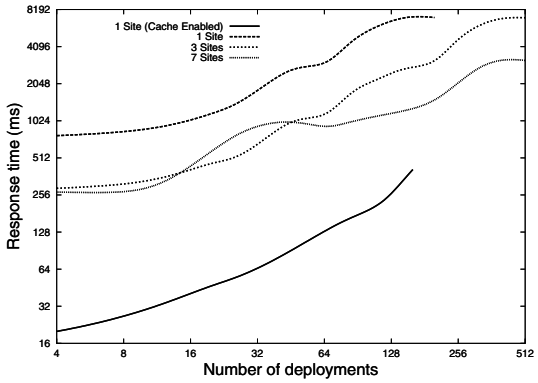
1) *Globus*, using the GridFTP protocol [12] to remotely transfer and the Globus Resource Allocation Manager (GRAM) [12] to remotely deploy the code;
2) *Expect*, by programmatically acquiring local shell on a target site and automatizing the installation [20].

Table II illustrates the overheads associated with different deployment operations. The communication overhead depends on the size of the installation files and is generally lower than the deployment overhead introduced by the code compilation. The registration of a new activity type and its deployments imply reasonable costs. The Expect-based deployment is more efficient than Globus because it accesses sites using built-in system shells.

We have also studied the scalability of the registration framework on one, three, and seven Grid site configurations, by enabling and disabling the caching of activity types and deployments on local sites for faster future access. Fig. 9(b) shows the response time per request for a list of deployments associated with an activity type. The deployment entries are equally distributed on all involved sites. We can observe that there is a significant improvement in performance by increasing number of sites and also by enabling the caching.



(a) Resource Manager overhead breakdown.



(b) Response time per activity deployment request with caching on one Grid site and without caching on one, three and seven Grid sites.

Fig. 9. Resource Manager experimental results.

Method	*Operation / Overhead*	*WIEN2k*	*Invmod*	*Counter*
Expect	Activity type registration	633	632	665
	Communication	1,667	1,381	1,279
	Installation / deployment	8,068	27,776	29,843
	Deployment registration	700	695	697
	Expect overhead	2,100	2,100	2,100
	Total overhead	11,068	30,484	32,484
Globus	Activity type registration	633	632	665
	Communication	5,600	2,500	2,400
	Installation / deployment	18,068	49,700	39,756
	Deployment registration	700	695	697
	Globus overhead	9,800	9,900	9,800
	Total overhead	25,001	53,527	43,518

TABLE II
TIME SPENT IN DIFFERENT DEPLOYMENT OPERATIONS (MILLISECONDS).

VIII. RELATED WORK

DAGMan [6] is a centralized meta-scheduler for Condor jobs organized in a directed acyclic graph. Important control flow constructs such as branches and loops are missing. Scheduling is done through matchmaking with no advanced optimization. Fault tolerance is addressed through rescue DAG, automatically generated whenever an activity instance fails. The ASKALON checkpointing, in contrast, addresses also the case when the Enactment Engine itself crashes.

Pegasus [7] uses DAGMan as enactment engine, enhanced with data derivation techniques that simplify the workflow at run-time based on data availability. Pegasus provides a workflow partitioning approach, which is problematic for strongly unbalanced workflows like our Invmod application. Research results report simulation-based scheduling using a weighted Min-min heuristic.

Triana [21] uses the Grid Application Toolkit interface to the Grid through Web services. It misses compact mechanisms for expressing large parallel loops. Scheduling is done just-in-time with no optimizations or performance estimates.

ICENI [15] contains low-level enactment engine-specific constructs such as `start` and `stop` in the workflow definition. Scheduling is done using random, best of n-random, simulated annealing, and game theory algorithms.

Taverna's [16] workflow language called SCUFL is also limited to DAGs. Scheduling is done just-in-time, while fault tolerance is addressed at activity-level through retries and alternate resources.

The GridAnt [2] centralized workflow engine extends the Ant commodity tool for controlling the application building processes in Java with low-level constructs such as `grid-copy` and `grid-execute`. Scheduling is done manually and fault tolerance is not addressed.

The GrADS project [14] restricts workflows to a DAG model and does not propose any workflow specification language. The architecture is centralized and does not consider any service-oriented Grid technology. Scheduling is done using Min-min, Max-min, and Sufferage algorithms traditionally used for independent tasks, which are less appropriate for workflows with large control flow depths and data flow dependencies. Like in ASKALON, performance prediction models are derived from historical executions based on processor operations and memory access patterns.

UNICORE [8] provides a graphical composition of directed graph-based workflows. It does not support parallel loop constructs. Scheduling is user-directed (manual) and fault tolerance is not addressed.

The workflow management in Gridbus [5] provides an XML-based workflow language, oriented towards parametrization and QoS requirements. No branches and loops are supported. The scheduler provides just-in-time mappings using Grid economy mechanisms. Fault tolerance is limited to activity-level using replication.

ASKALON differs in several aspects compared to the above mentioned related projects. AGWL allows a scalable specification of large numbers of parallel activities, typical to scientific workflows, by using compact parallel loops. The Enactment Engine effectively handles large data collections generated by large-scale control and data flow constructs. Additionally, it provides two levels of workflow checkpointing for restoring and resuming the execution in case of failures of

the engine itself. The HEFT and genetic search algorithms implemented by the ASKALON Scheduler are, to the best of our knowledge, not addressed by any of the related projects. ASKALON proposes novel architectural feature based on a clear separation between the Scheduler and the Resource Manager which covers both physical and logical resources and provides brokerage, constraint-based advanced reservations, and activity type to deployment mappings.

IX. Conclusions

In this paper we presented the ASKALON application development and computing environment. The new contributions of this work are centered around several integrated run-time middleware services.

In contrast to many existing systems, ASKALON supports a programming interface that shields the application developer from the low-level middleware technologies with the goal of providing an invisible Grid. Our Scheduler supports HEFT and genetic search as optimization algorithms which perform significantly better than a pure resource broker, in particular in the case of unbalanced workflows. The Scheduler significantly benefits from a Performance Prediction service that provides expected execution times based on a training phase and statistical methods. The Enactment Engine efficiently handles large collections of data dependencies produced by hundreds of parallel activities specific to scientific workflows. We have demonstrated significant performance gains through two checkpointing methods for saving and restoring the execution of Grid workflows upon engine and application failures. A separate Resource Manager, which covers both physical resources and workflow activities, renders the boundaries of resource brokerage, virtual organization-wide authorization, advanced reservation, and provides mechanisms for Grid resource discovery, selection, and allocation along with resource requester and provider interaction.

We have demonstrated the integrated use of the ASKALON services for two real-world scientific workflows executed in the Austrian Grid infrastructure. Our future work will focus on further optimization of workflow executions to increase their scalability on the Grid, scheduling based on QoS parameters to be negotiated between the Scheduler and the Resource Manager, and automatic bottleneck detection and steering based on on-line performance analysis.

Acknowledgement

This research has been partially supported by the Austrian Science Fund as part of the Aurora project under the contract SFBF1104 and the Austrian Federal Ministry for Education, Science and Culture as part of the Austrian Grid project under the contract GZ 4003/2-VI/4c/2004.

References

[1] "The Austrian Grid Consortium," http://www.austriangrid.at.

[2] K. Amin, M. Hategan, G. von Laszewski, N. J. Zaluzec, S. Hampton, and A. Rossi, "GridAnt: A Client-Controllable Grid Workfbw System," in *37th Hawai'i International Conference on System Science*, 5-8 2004.

[3] "GNU AutoConf Tool," http://www.gnu.org/software/autoconf.

[4] P. Blaha, K. Schwarz, G. Madsen, D. Kvasnicka, and J. Luitz, *WIEN2k: An Augmented Plane Wave plus Local Orbitals Program for Calculating Crystal Properties*. Institute of Physical and Theoretical Chemistry, TU Vienna, 2001.

[5] R. Buyya and S. Venugopal, "The Gridbus Toolkit for Service Oriented Grid and Utility Computing: An Overview and Status Report," in *1st International Workshop on Grid Economics and Business Models (GECON 2004)*. IEEE Computer Society Press, Apr. 2004, pp. 19–36.

[6] Condor Team, "Dagman (directed acyclic graph manager)," http://www.cs.wisc.edu/condor/dagman/.

[7] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M.-H. Su, K. Vahi, and M. Livny, "Pegasus: Mapping Scientifi c Workfbws onto the Grid," in *European Across Grids Conference*, 2004, pp. 11–20.

[8] D. W. Erwin and D. F. Snelling, "UNICORE: A Grid computing environment," *Lecture Notes in Computer Science*, vol. 2150, 2001.

[9] "Expect," http://expect.nist.gov.

[10] T. Fahringer, "ASKALON Grid Application Development and Computing Environment," Distributed and Parallel Systems Group, Institute for Computer Science, University of Innsbruck, http://dps.uibk.ac.at/askalon.

[11] T. Fahringer, J. Qin, and S. Hainzer, "Specifi cation of Grid Workfbw Applications with AGWL: An Abstract Grid Workfbw Language," in *International Symposium on Cluster Computing and the Grid 2005 (CCGrid 2005)*. IEEE Computer Society Press, May 9-12 2005.

[12] Globus Alliance, "The Globus Toolkit," http://www.globus.org.

[13] ——, "The Web Services Resource Framework," http://www.globus.org/wsrf.

[14] Ken Kennedy et.al., "New Grid Scheduling and Rescheduling Methods in the GrADS Project," in *International Parallel and Distributed Processing Symposium, Workshop for Next Generation Software*. IEEE Computer Society Press, Apr. 2004.

[15] A. Mayer, S. McGough, N. Furmento, W. Lee, S. Newhouse, and J. Darlington, "ICENI Datafbw and Workfbw: Composition and Scheduling in Space and Time," in *UK e-Science All Hands Meeting*, Nottingham, UK, Sept. 2003, pp. 627–634.

[16] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. C. adn K. Glover, M. Pocock, A. Wipat, and P. Li, "Taverna: a tool for the composition and enactment of bioinformatics workfbws," *Bioinformatics*, vol. 20, no. 17, pp. 3045–3054, 2004.

[17] POVray, http://www.povray.org/.

[18] R. Prodan and T. Fahringer, "Dynamic Scheduling of Scientifi c Workfbw Applications on the Grid using a Modular Optimisation Tool: A Case Study," in *20th Symposion of Applied Computing (SAC 2005)*. ACM Press, Mar. 2005.

[19] M. Siddiqui and T. Fahringer, "GridARM: Askalon's Grid Resource Management System," in *Advances in Grid Computing - EGC 2005 - Revised Selected Papers*, ser. Lecture Notes in Computer Science, vol. 3470. Amsterdam, Netherlands: Springer Verlag GmbH, ISBN 3-540-26918-5, June 2005, pp. 122–131.

[20] M. Siddiqui, A. Villazon, J. Hofer, and T. Fahringer, "GLARE: A grid activity registration, deployment and provisioning framework," in *Supercomputing Conference*. ACM Press, November 12-18 2005.

[21] I. Taylor, M. Shields, I. Wang, and R. Rana, "Triana applications within Grid computing and peer to peer environments," *Journal of Grid Computing*, vol. 1, no. 2, pp. 199–217, 2003.

[22] D. Theiner and P. Rutschmann, "An inverse modelling approach for the estimation of hydrological model parameters," in *Journal of Hydroinformatics*, 2005.

[23] Tony Andrews et.al., "Business Process Execution Language for Web Services (BPEL4WS)," OASIS," Specifi cation Version 1.1, May 2003.

[24] H.-L. Truong and T. Fahringer, "SCALEA-G: a Unifi ed Monitoring and Performance Analysis System for the Grid," *Scientific Programming*, vol. 12, no. 4, pp. 225–237, 2004, iOS Press.

[25] R. Wolski, N. T. Spring, and J. Hayes, "The network weather service: a distributed resource performance forecasting service for metacomputing," *Future Generation Computer Systems*, vol. 15, no. 5–6, pp. 757–768, Oct. 1999.

[26] H. Zhao and R. Sakellariou, "An experimental investigation into the rank function of the heterogeneous earliest fi nish time scheduling algorithm." in *Euro-Par*, 2003, pp. 189–194.