Specification of Grid Workflow Applications with AGWL: An Abstract Grid Workflow Language*

Thomas Fahringer, Jun Qin, Stefan Hainzer
Institute of Computer Science, University of Innsbruck
Technikerstraße 13, 6020 Innsbruck, Austria
{Thomas.Fahringer, Jun.Qin, Stefan.Hainzer}@uibk.ac.at

Abstract

Currently Grid application developers often configure available application components into a workflow of tasks that they can submit for executing on the Grid. In this paper, we present an Abstract Grid Workflow Language (AGWL) for describing Grid workflow applications at a high level of abstraction. AGWL has been designed such that the user can concentrate on specifying scientific applications without dealing with either the complexity of the Grid or any specific implementation technology such as Web/Grid services, software components or Java classes. AGWL is an XML-based language which allows a programmer to define a graph of activities that refer to computational tasks or user interactions. Activities are connected by control and data flow links. A rich set of constructs is provided to simplify the specification of Grid workflow applications which includes basic control flow constructs such as `if`, `foreach` *and* `while` *loops as well as advanced control flow constructs including parallel sections, parallel loops and collection iterators. Moreover, AGWL supports a generic high level access mechanism to data repositories. AGWL is the main interface to the ASKALON Grid application development environment and has been applied to numerous real world applications. We demonstrate one example of a material science workflow that has been successfully ported to a Grid infrastructure based on an AGWL specification.*

1. Introduction

Grid computing [1] enables the virtualization of distributed and heterogeneous resources such as CPUs, storage systems, sensor networks and scientific instruments thus a unified view of resources, authentication, authorization and accounting can be represented to applications and end users. With the advent of Grid technologies, scientists and engineers are building more and more complex applications to manage and process large data sets and execute scientific experiments on distributed Grid resources. It appears believed that Grid workflow applications are emerging as one of the most important and challenging Grid application classes.

A Grid workflow application can be seen as a collection of activities (computational tasks) that are processed in a well-defined order to accomplish a specific goal. These activities are expected to be executed on heterogeneous resources which are geographically distributed. Many resources may be involved in one workflow execution. Furthermore, there is no central ownership and control in a typical Grid environment. The computational and networking capabilities can vary significantly over time. These distributed and dynamic characteristics of the Grid bring many challenges to the execution and the management of Grid workflow applications.

In this paper, we describe AGWL, an XML-based high level Abstract Grid Workflow Language, which shields the details of the underlying Grid infrastructure and allows programmers to compose scientific workflow applications in an intuitive way. AGWL has been carefully designed to include important workflow constructs as well as some advanced workflow constructs to simplify the execution of activities in parallel. In addition, programmers can specify some high-level constraints and properties on activities or on data flows for the underlying workflow enactment engine to optimize the execution of workflow applications.

The paper is organized as followed. In section 2, an overview of the development process of the Grid workflow application is introduced. Section 3 is the main part of this article, which defines AGWL and its

*The work described in this paper is partially supported by the Austrian Grid Project, funded by the Austrian BMBWK (Federal Ministry for Education, Science and Culture) under contract GZ 4003/2-VI/4c/2004.

language constructs. A material science workflow application with AGWL is discussed in the fourth section. Finally, based on the discussions in the previous sections, some conclusions are drawn and future work is presented in the last section.

2. Related Work

Although workflow applications have been extensively studied in areas like the business process modeling [2] and web services [3,4,5], it is relatively new in the Grid computing area.

Several efforts toward Grid workflow applications have been made. The Workflow Enactment Engine (WFEE) [6] is based on a workflow language xWFL that comprises three parts: parameter definitions, task definitions and data link definitions. The Grid Service Flow Language (GSFL) [7] supports the specification of workflow descriptions for Grid services in the OGSA framework. However, both WFEE and GSFL miss some important control flow constructs such as branches and loops. Grid workflow in the GriPhyN [8] project are constructed by using partial or full abstract descriptions of "components", which are executable programs that can be located on some resources. Their abstract workflow is limited to acyclic graphs. Moreover, a reduced data flow model is supported, and there is no customization allowed and no advanced constructs for the parallel execution of "components". DAGMan [9], which was developed to schedule jobs for the Condor system in an order represented by a directed acyclic graph (DAG), is also limited to acyclic graph workflows. Other Grid workflow applications include SycFlow [10], GridFlow [11] and Triana [12].

Existing work commonly commonly suffers by one or several of the following drawbacks: control flow limitations (e.g. no branches or loops), limited mechanism for expressing parallelism (e.g. no parallel sections or loops), restricted data flow mechanisms (e.g. limited to files), implementation specific (focus on Web services, Java classes, software components, etc.), and low level constructs (e.g. start/stop tasks, transfer data, queue task for execution, etc.) that should be part of the workflow enactment engine.

In contrast to much existing work, AGWL provides an advanced and user-oriented Grid workflow language which shields most complexity of the underlying Grid infrastructure and enactment engine from the application developer. AGWL supports a reasonable and important set of control and data flow constructs to build Grid workflow applications. AGWL is also a modularized Grid workflow language which supports the declaration of sub-workflows that can be invoked by other workflows. Most existing workflow languages are based on a simple I/O system (e.g. files) whereas AGWL also enables a generic access mechanism to data repositories which is important for many scientific applications. AGWL includes several constructs that are crucial to steer an enactment engine for the performance-driven execution of workflow applications. For instance, properties and constraints are AGWL constructs to optimize the workflow execution on a Grid.

3. System Overview

Figure 1 shows an overview of the development process of Grid workflow applications in the ASKALON Grid application development environment [13] from an abstract representation to an actual execution on a Grid infrastructure. The development process consists of three fundamental procedures: Composition, Reification and Execution.

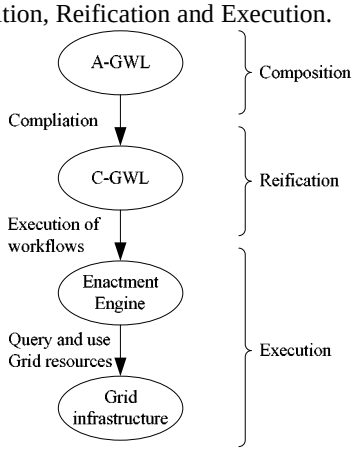


Figure 1: The development process of Grid workflow applications in ASKALON

Composition: The user composes the Grid workflow by using a visual composition tool Teuta [14] to define a graph of activities or writing an AGWL source file directly. At this level, activities correspond mostly to computational tasks and the specification of their input and their output data. There is no notion of how the input data is actually delivered to activities and how activities are implemented, invoked and terminated. AGWL contains all the information specified by the user during workflow composition.

Reification: A transformation system in ASKALON compiles AGWL to a Concrete Grid Workflow Language (CGWL) which represents an executable workflow. CGWL contains all information specified by the user and some additional information, such as data types provided by the software infrastructure, which is necessary to execute the workflow. CGWL also considers data conversion and pre-processing.

Execution: CGWL is interpreted by the underlying workflow enactment engine of ASKALON to construct

and execute the Grid workflow application on a Grid infrastructure.

4. AGWL Specification

In AGWL, a workflow consists of *activities*, which can produce results from the given input data, *control flow constructs*, which define control flow among activities, and *data flow* through which data packages can be exchanged among activities. The composition of an application is done by specifying both the control flow and the data flow among activities.

In addition, properties and constraints can be specified to optimize the execution of workflow applications.

4.1. Basic Concepts

4.1.1. Activity

An activity represents a specific computational function (e.g. to multiply two matrices). Its implementation can be derived from its activity type. Its input/output behavior can be specified by data-in/data-out ports. The definition of an activity is shown in Figure 2.

```
<activity name="name" type="type">
  <dataIn name="name" (source="source")?>
    (<value> value as XML </value>)?
  </dataIn>*
  <dataOut name="name" />*
</activity>
```

Figure 2: Activity

Activity Name: The activity name serves an identifier for the activity. Activities must be organized in an `AGWL-workflow` or a `sub-workflow` which define a scope for them. In the scope, the name of each activity is unique.

Activity Type: The activity type is an abstract description of a computational entity available in the Grid. For instance, an activity type *MatrixMult* can be used to describe a software component that multiplies two matrices. The activity type must be defined and registered before crafting AGWL workflows. Activity types shield the implementation details from the AGWL programmer. Locating activity type implementations and invoking activity types are done by the enactment engine for AGWL.

Data-In/Data-Out Ports: The number and types of the data-in/data-out ports are determined by the chosen activity type. The data-in port can be specified by (1) setting its `source` attribute to the name of a data-out port of another activity in the same workflow in the form of `activity-name/data-out-port-name`, (2) setting its `source` attribute to the name of an abstract data container *repository*, or (3) specifying an XML constant in the `value` element, if the needed data is an XML constant and no `source` is specified. We will address the *repository* in detail in Section 3.3.6. Linking the data-in ports and data-out ports of different activities through the `source` attributes of the data-in ports defines the data flow of workflow applications.

4.1.2. Data Package

The data contained in one data-in or data-out port is denoted as a data package. The data in activities can be interchanged through the exchange of data packages along the specified data flow. When an activity is executed, there must be exactly one data package for each data-in port, and when the activity finishes, it produces exactly one data package for each data-out port.

Data Package Collection: In some cases it is necessary to wrap several data packages in a container. For example, transferring data packages one by one over a network is often slower as compared to creating a data package container that holds all of the data packages and transferring a single data package collection. This is similar to packing files into a zip or a tar archives. We call this data package container `AGWL-Collection`. To support operations on `AGWL-collections`, several activity types are predefined, e. g. to add a data package into a collection, to remove a data package, to remove all data packages, etc. Each data package in an `AGWL-collection` can be accessed through a zero bounded index or by its optional name if it exists. `AGWL-collection` is a data package as well, thus it in turn can be contained in other `AGWL-collections`.

4.2. Control Flow Constructs

Control Flow Constructs (CFCs) are used to specify control flows in AGWL. A CFC can be described as a composed activity with a certain control flow structure. In the remainder of this paper, the term *basic activity* will be used to denote an activity such as the one in Figure 2 and the term *activity* denote either a *basic activity* or a CFC. In a CFC, the specifications for its name attribute, its data-in/data-out ports and the source attributes of its data-in ports are similar to the identical attributes of a *basic activity*. To avoid redundancy, we will not separately explain these attributes in the following sections.

There are two kinds of CFCs in AGWL: basic CFCs, which are similar to some well known constructs of high-level languages such as `if`, `for` or `while`, and advanced CFCs, which enable the expression of parallelism at a high level, such as parallel loops.

4.2.1. Basic Control Flow Constructs

- Sequence Construct

The `sequence` construct (Figure 3) imposes a sequential control flow on all of its contained *activities*.

```
<sequence name="name">
  <dataIn name="name" (source="source")?>
    (<value> value as XML </value>)?
  </dataIn>*
  activity+
  <dataOut name="name" source="source">*
</sequence>
```

Figure 3: `sequence` construct

Data-Out Ports: The `source` attributes specify internal data flows from data-out ports of inner *activities* to data-out ports of the `sequence` construct.

- Parallel Construct

The `parallel` construct (Figure 4) indicates that all contained *activities* can be executed simultaneously.

```
<parallel name="name">
  <dataIn name="name" (source="source")?>
    (<value> value as XML </value>)?
  </dataIn>*
  activity+
  <dataOut name="name" source="source">*
</parallel>
```

Figure 4: `parallel` construct

Data-Out Ports: The `source` attributes specify internal data flows from data-out ports of inner *activities* to data-out ports of the `parallel` construct.

- Conditional Constructs

There are two conditional constructs in AGWL: `if` (Figure 5) and `switch` (Figure 6). Both of them enable the conditional execution of *activities* which depends on the condition expression of these constructs.

```
<if name="name">
  <dataIn name="name" (source="source")?>
    (<value> value as XML </value>)?
  </dataIn>*
  <condition> condition </condition>
  <then>
    activity+
  </then>
  (<else>
    activity+
  </else>)?
  <dataOut name="name" source="source"/>*
</if>
```

Figure 5: `if` construct

Condition: In the `if` or `switch` construct, the condition is an XSLT [15] expression. Correspondingly, it requires that the values of the data-in ports must be interpretable as XML.

Data-Out Ports: The control flow outcome of the `if` or `switch` construct is commonly unknown at compile time. Therefore, it is not allowed to connect a data-out port of an inner activity of a conditional construct to a data-in port of an activity outside of this construct. Instead, data-out ports are defined for the `if` or `switch` construct which can be connected to data-in ports of other activities outside of the `if` or `switch` construct. The `source` attribute specifies the possible internal data flow from data-out ports of internal activities to data-out ports of the enclosing conditional construct. It contains a comma separated list of data-out ports of some inner activities. This list must contain one entry for each possible branch in the `if` or `switch` construct. Especially, if the optional `else` branch in the `if` construct or the optional `default` branch in the `switch` construct is not specified, the name of data-in ports must be given to ensure that the data-out ports are set with valid data packages.

```
<switch name="name">
  <dataIn name="name" (source="source")?>
    (<value> value as XML </value>)?
  </dataIn>*
  <case condition="condition">
    activity+
  </case>+
  (<default>
    activity+
  </default>)?
  <dataOut name="name" source="source"/>*
</switch>
```

Figure 6: `switch` construct

- Loop Constructs

There are three loop constructs in AGWL: `while` (Figure 7), `for` (Figure 8) and `forEach` (Figure 9). The `while` construct can be used to execute the loop body zero or more times. The `for` construct is provided to execute its body multiple times controlled by a counter. The `forEach` construct has been included in AGWL to enable the iteration across a data package collection. The loop body is executed once for each element in the data collection.

```
<while name="name">
  <dataIn name="name" (source="source")?
    (loopSource="loopSource")?>
    (<value> value as XML </value>)?
  </dataIn>*
  <condition> condition </condition>
  <loopBody>
    activity+
  </loopBody>
  <dataOut name="name" source="source"/>*
</while>
```

Figure 7: `while` construct

```
<for name="name">
  <dataIn name="name" (source="source")?
    (loopSource="loopSource")?>
    (<value> value as XML </value>)?
  </dataIn>*
  <loopCounter name="name" from="from" to="to"
    (step="step")? />
  <loopBody>
    activity+
  </loopBody>
  <dataOut name="name" source="source"/>*
</for>
```

Figure 8: `for` construct

Data-In Ports: In these loop constructs, the optional `loopSource` attribute of data-in ports is used to express a cyclic data flow which is commonly linked to a data-out port of an activity inside the loop body, in the form of `activity-name/data-out-port-`

name. After each execution of the loop body, the data-in ports of the loop constructs receive the new values specified by the loopSource attributes. Such data-in ports receive the final results of the loop body after the loop terminates. For the forEach construct, the first data-in port must refer to a collection over which this construct iterates.

```
<forEach name="name">
  <dataIn name="name" (source="source")?>
    (<value> value as XML </value>)?
  </dataIn>
  <dataIn name="name" (source="source")?
    (loopSource="loopSource")?>
    (<value> value as XML </value>)?
  </dataIn>*
  <loopElement name="name"/>
  <loopBody>
    activity+
  </loopBody>
  <dataOut name="name" source="source"/>*
</forEach>
```

Figure 9: forEach construct

Condition: The condition (expressed as an XSLT expression) of the while construct controls how often the loop body is executed. This condition is evaluated before the loop body is executed. The loop is executed until this condition is evaluated to false.

Loop Counter: The loopCounter in the for construct controls how often the loop body is executed. It provides an implicit data-in port for the activities in the loop body and can be accessed by its name. The value of the loopCounter is initially assigned to the value specified by the variable from and is increased by the value of step until it reaches the value of to or larger. The values of from, to, step can be expressed as constants or as an XPATH [16] expression where the values of data-in ports, which must be interpretable as XML, can be used. The values of from, to and step are only evaluated once at the beginning of an invocation of the for construct.

Loop Element: The loopElement in the forEach construct specifies the name by which the activities in the loop body can access the current iteration element of the collection. The loopElement value is assigned to each element in the data collection exactly once following the order of elements in the collection. If the collection is empty, then the loop body is never executed.

Data-Out Ports: In these loop constructs, the source attributes of the data-out ports are set to the names of the data-in ports with the loopSource attribute. Such data-in ports are set to the final results after the execution of the loop construct. Activities outside of a loop construct can access the final results through the data-out ports of the loop construct after the loop terminates.

- Directed Acyclic Graph (DAG) Construct

For specifying more complex control flow, AGWL includes the dag construct (Figure 10). The control flow of a dag construct is described as a directed acyclic graph. In order to specify the execution order of *activities*, a special wrapper element dagNode is introduced. An activity in a dag can be executed as soon as all of its predecessors terminated. Thus a dag construct has a potential for parallel execution of its contained activities.

```
<dag>
  <dataIn name="name" (source="source")?>
    (<value> value as XML </value>)?
  </dataIn>*
  <dagNode name="name" predecessor="names"/>
    activity
  </dagNode>+
  <dataOut name="name" source="source">
</dag>
```

Figure 10: dag construct

Dag Node Name: Each dagNode must have a unique name in the scope defined by the dag. Each dagNode element contains a single *activity*.

Dag Node Predecessor: The predecessor attribute of a specific dagNode *dn* is a comma separated list of names of some other dagNode elements which must be executed before *dn* can be executed. Those dagNode elements without predecessors are root elements of the dag construct. AGWL allows multiple roots in a dag construct. The root elements, as well as those dagNode elements whose predecessors have finished, can be executed immediately possibly in parallel with other dagNode elements.

Data-Out Ports: The source attribute specifies dag internal data flow from data-out ports of inner *activities* to data-out ports of the dag construct.

4.2.2. Advanced Control Flow Constructs

Advanced control flow constructs are supported by AGWL to express that multiple activities can be executed in parallel which includes: parallelFor (Figure 11) and parallelForEach (Figure 12).

```
<parallelFor name="name">
  <dataIn name="name" (source="source")?>
    (<value> value as XML </value>)?
  </dataIn>*
  <loopCounter name="name" from="from" to="to"
    (step="step")?/>
  <loopBody>
    activity+
  </loopBody>
  <dataOut name="name" source="source"/>*
</parallelFor>
```

Figure 11: parallelFor construct

The parallelFor/parallelForEach construct is similar to the for/forEach construct with the difference that in parallelFor/parallelForEach all loop iterations can be executed simultaneously. It can be assumed that the data input of any iteration is independent of data produced by other iterations of the same construct. The semantics of the attributes in

parallel loop constructs is identical to those of the sequential loop constructs.

```
<parallelForEach name="name">
  <dataIn name="name" (source="source")?>
    (<value> value as XML </value>)?
  </dataIn>
  <dataIn name="name" (source="source")?>
    (<value> value as XML </value>)?
  </dataIn>*
  <loopElement name="name"/>
  <loopBody>
    activity+
  </loopBody>
  <dataOut name="name" source="source"/>*
</parallelForEach>
```

Figure 12: `parallelForEach` construct

4.3. Data Flow in AGWL

The data flow, which describes the flow from data-out ports to data-in ports, is an inherent part of several CFCs. To fully understand the AGWL data flow model, some aspects are examined in more detail in the following.

4.3.1. Well-defined Data-in and Data-out Ports

For each *activity* in AGWL it must be guaranteed that whenever the control flow reaches the *activity*, all the data-in ports of the *activity* have been assigned to well-defined values (valid data packages). When the control flow leaves, all its data-out ports must be well-defined as well. For constructs with a trivial control flow (one predecessor and one successor), the definition of this constraint is straightforward. For all other constructs with a more complex control flow (e.g. conditional constructs, sequential loop constructs, parallel loop constructs) the corresponding data flow can become less trivial which will be explored in the following sections. Note that data can flow to inner activities of CFCs via the data-in port of the CFC or directly from activities outside of the CFC. This flexibility reduces the overall number of data-in ports if data must be transferred to inner activities from outer activities. However, data flow can leave a CFC only via its data-out ports.

4.3.2. Data Flow in Conditional Constructs

Data flow is more complex for conditional constructs. In the following we discuss the `if` construct to the explain data flow issues for conditional constructs. Figure 13(a) and 13(b) illustrate two possible data flows of the `if` construct. Figure 13(a) shows an illegal data flow that is not allowed in AGWL. If the `then` branch is executed, then the data-in port of activity `A4` is not defined. Therefore, the `then` branch must define the data-out ports of the `if` construct as shown in Figure 13(b). Activity `A4` is not allowed to be linked to a data-out port of an inner activity of the `if` construct. Instead, it must be linked to a data-out port of the `if` construct. In this case, it is ensured that each data-out port is well-defined with a valid data package for each possible execution path.

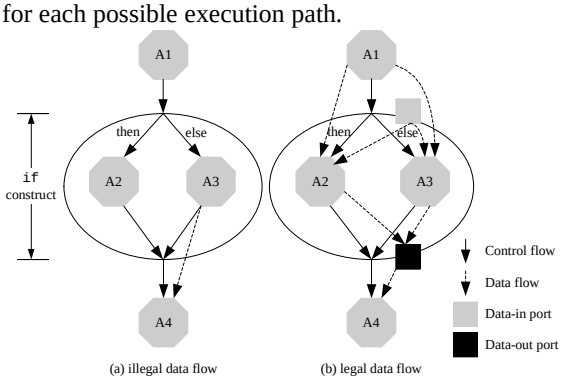


Figure 13: The data flow in conditional constructs

4.3.3. Data Flow in Sequential Loop Constructs

The data flow in sequential loop constructs occurs in `while`, `for` and `forEach` constructs. We explain the data flow model for the `while` construct which is similar for all other sequential loop constructs. For the `while` construct, we have to describe the flow of data from one iteration to next one, i.e. the output of one iteration serves as the input of the subsequent iteration. To model the data flow for the `while` construct, we have to consider two cases:

(1) The loop body is never executed if the loop condition is never evaluated to true.
(2) The loop body is executed multiple times which implies a cyclic data flow.

Figure 14 illustrates the data flow model for the `while` construct in AGWL. The control flow inside the loop body is not shown in order to avoid overloading this figure.

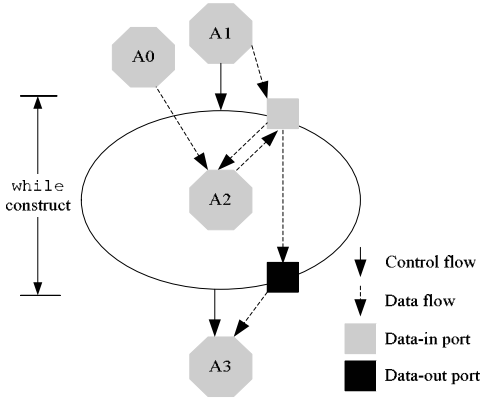


Figure 14: The data flow of sequential loop constructs

(1) The data-in ports of the `while` construct are assigned initial values specified by the `source` attribute (e.g. a source activity and its port).

(2) If the loop condition is evaluated to false, the data flow continues at step (4). If true, the inner activity A2 is executed. A2 can obtain data from either the data-in ports of the while construct or from some outside activities such as A0 executed.
(3) Once A2 has been executed, data packages from its data-out ports can be transferred to the data-in ports of the surrounding while construct, specified by its loopSource attribute. Then the loop condition is evaluated again with the new data packages arrived at the data-in ports.
(4) When the loop condition is evaluated to false, the current data packages of the data-in ports are mapped to the data-out ports of the while construct.
(5) Activities outside the while construct can obtain the data packages from the data-out ports of the while construct only.

For each possible execution path inside the while construct, it is ensured that the data flow is well-defined and valid data packages are available at all data-in and data-out ports of all executed activities. In case that A2 receives data packages from an activity outside of the while construct, these data packages remain constant and accessible for all loop iterations. Note that the number of possible data-out ports of the while construct must be smaller than or equal to the number of its data-in ports.

4.3.4. Data Flow in Parallel Loop Constructs

In the following we describe the data flow mechanism for parallel loop constructs exemplified by the parallelFor construct (Figure 15). The data flow for the parallelForEach construct is similar. In contrast to sequential loop constructs, there is no need for a cyclic data flow. Since in general it cannot be decided at compile time how many times the loop body will be executed, a data collection is used to hold all result data produced by all the loop iterations when execution of the loop construct is finished.

Before any iteration of the parallelFor construct is executed, the number of iterations is evaluated. Then for each iteration an instance of the loop body (activity A2 in the Figure 15) is invoked concurrently. Each instance (A2.1, A2.2, A2.3 or A2.4) receives a unique iteration index from the loop counter of the parallelFor construct. In the case of parallelForEach, each instance would receive an element of the first data-in port of the parallelForEach construct which is an AGWL-collection. Every loop instance may also receive some other input data from either the data-in ports of the parallelFor construct or some outside activities such as A1. Each instance may also have its own data-in ports. At the end of its execution, each instance of the loop body writes its results into the data collection specified by the data-out ports of the parallelFor construct. This data collection can be accessed through the data-out ports of the parallelFor construct by subsequent activities such as A3.

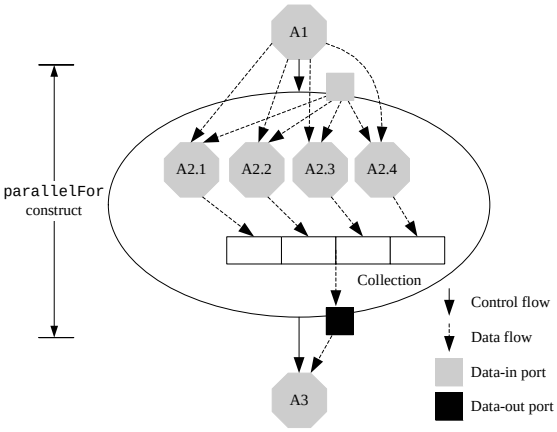


Figure 15: The data flow in parallel loop constructs

4.3.5. Data Flow in the DAG Construct

Figure 16 illustrates a possible data flow in a dag construct. The control flow inside the dag construct is not shown to avoid overloading the figure. A3, A5 are the root elements of the dag construct. In the dag construct, the user can define an arbitrary complex acyclic data flow as long as the activities associated with the data source are executed before the ones associated with the data destination according to the defined control flow.

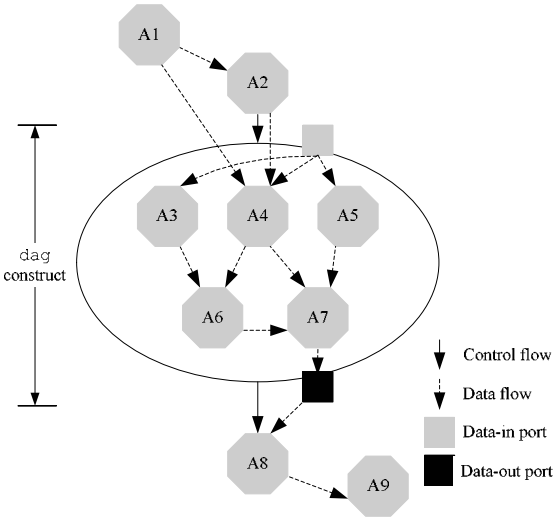


Figure 16: The data flow in the DAG construct

4.3.6. Advanced Data Flow

In addition to the data flow among activities, AGWL supports the data flow between activities and special entities called *repositories*, which are abstractions for data containers. They are used to model, for instance, saving intermediate results or querying data resources without knowing any details about how repositories are actually implemented, e.g. file servers, databases, etc. A repository has a unique name in the context of a workflow application. The retrieval/insertion of information is made by specifying the `source`/`saveto` attribute with the repository name in data-in/data-out ports. Figure 17 illustrates that an activity stores its data-out port (named `outDat`) into a repository (named `R`).

```
<dataOut name="outDat" saveto="R">
```

Figure 17: Save a data-out port into a repository

4.4. Properties and Constraints

In AGWL, properties and constraints can be defined by the user to provide additional information for a workflow enactment engine to optimize and steer the execution of workflow applications. Properties provide hints about the behaviour of activities, e.g. the expected size of the input data, the estimated computational complexity, etc. Constraints should be complied by the underlying workflow enactment engine, e. g. to minimize execution time, to provide as much memory as possible, to run on the specific host architecture, etc. The user can define properties and constraints elements for activities and for data-in and data-out ports (Figure 18).

```
<activity name="name" type="type">
  <dataIn name="name" (source="source")?>
    <properties>...</properties>
    <constraints>...</constraints>
  </dataIn>*
  <properties>...</properties>
  <constraints>...</constraints>
  <dataOut name="name">
    <properties>...</properties>
    <constraints>...</constraints>
  </dataOut>*
<activity>
```

Figure 18: Properties and constraints

4.5. The Structure of an AGWL file

Grid workflow applications are described in so-called AGWL source files (Figure 19) which can import activity type definitions and workflow definitions, and invoke sub-workflows. We address each part in the following sections.

4.5.1. Importing Activity Type Definitions

Before using an activity type in AGWL workflows, it is required to import the definition of the activity type which is always organized in an ATD file. The ATD files may be created and published by anyone providing implementations of activities which can be reused for Grid workflow construction. Line 2-4 in Figure 19 illustrates how to import activity types by using the `importATD` element.

```
 1 <agwl-workflow>
 2   <importATD url="url" name="name">
 3     (<alternativeUrl> url </alternativeUrl>)*
 4  </importATD>*
 5  <importWD url="url" name="name">
 6    (<alternativeUrl> url </alternativeUrl>)*
 7  </importWD>*
 8  <subWorkflow name="name">
 9    <dataIn name="name" />*
10    <body>
11      activity+
12    </body>
13    <dataOut name="name" source="source"/>*
14  </subWorkflow>*
15  activity*
16 </agwl-workflow>
```

Figure 19: An AGWL file

URL: The `url` attribute specifies the location of the ATD file. Due to the dynamic nature of the Grid infrastructure, some Grid sites may be unavailable or unreachable unexpectedly. For this reason, we provide alternative URLs for the ATD files which can be accessed in an arbitrary order until correct ATD files are found.

ATD Name: The `name` attribute defines an identifier which is used to avoid naming conflicts. It is not unlikely that the same activity type name (e.g. `MatrixMult`) occurs in different ATD files. In order to distinguish them, we associate different ATD files (e.g. `d1.atd`, `d2.atd`) with unique identifiers (e.g. `d1`, `d2`) in different `importATD` elements. When referring to an activity type, it must be specified with the unique identifier, followed by a colon and the activity type name.

```
<agwl-workflow>
  <importATD url="http://.../d1.atd" name="d1"/>
  <importATD url="http://.../d2.atd" name="d2"/>
  <activity name="a1" type="d1:MatrixMult">
    <dataIn name="in1" source="matrix1"/>
    <dataIn name="in2" source="matrix2"/>
    <dataOut name="out" />
  </activity>
  <activity name="a2" type="d2:MatrixMult">
    <dataIn name="mat1" source="matrix3"/>
    <dataIn name="mat2" source="a1/out"/>
    <dataOut name="resMatrix" />
  </activity>
</agwl-workflow>
```

Figure 20: Importing Activity Type Definitions

In the example shown in Figure 20, the two matrices in the repositories `matrix1` and `matrix2` are multiplied based on the activity type defined in `d1.atd`. The result is then multiplied with the matrix in repository `matrix3` according to the activity type defined in `d2.atd`.

4.5.2. Importing and Invoking Workflows

In order to modularize and reuse workflows, we provide the `importWD` element (Line 5-7 in Figure 19) to import a workflow in another workflow. A workflow can be invoked in another workflow only after it is imported in that workflow. For efficiency reasons, the import elements always refer to compiled AGWL files, that is, CGWL representations. The `name` attribute and the `alternativeUrl` element have the same semantic as that of the `importATD` element described in section 3.5.1.

4.5.3. Invoking Sub-Workflows

Line 8-14 in Figure 19 defined a sub-workflow. Sub-workflows are similar to procedures in other high-level languages. They are used to modularize, encapsulate, and reuse a code region. Each `subWorkflow` has a unique name and well-defined data-in and data-out ports in an AGWL source file. Inside a sub-workflow, only data flow among inner activities is allowed. Sub-workflows can be invoked from different locations. To invoke a sub-workflow in an imported workflow, the name defined in the `importWD` element, followed by a colon and the name of the sub-workflow, must be specified. To invoke a sub-workflow which is defined in the same AGWL file, only the name of the sub-workflow must be indicated (Figure 21).

```
<agwl-workflow>
  <importWD url="http://.../wf.cgwl" name="wf"/>

  <subWorkflow name="subwf1">
    <dataIn name="in1" ...</dataIn>
    <dataIn name="in2" ...</dataIn>
    <body>
      <activity name="in_act" type="type">
        <dataIn name="param1" source="subwf1/in1"/>
        <dataIn name="param2" source="subwf1/in2"/>
        <dataOut name="res" />
      </activity>
    </body>
    <dataOut name="out" source="in_act/res"/>
  </subWorkflow>

  <activity name="a2" type="subwf1">
    <dataIn name="in1" source="inmat1"/>
    <dataIn name="in2" source="inmat2"/>
    <dataOut name="out" />
  </activity>
  <activity name="a1" type="wf:subwf2">
    <dataIn name="in1" source="inmat1"/>
    <dataIn name="in2" source="inmat2"/>
    <dataOut name="out" />
  </activity>

</agwl-workflow>
```

Figure 21: Invoking sub-workflows

5. Modeling a Real World Material Science Workflow with AGWL

WIEN2k [17] is a program package for performing electronic structure calculations of solids using density functional theory. The programs which compose the WIEN2k package are typically organized in a workflow illustrated in Figure 22. The `LAPW1` and `LAPW2` tasks can be executed in parallel with a number of *k-points* (atoms). A final task applied to several output files tests whether the problem convergence criterion is fulfilled (i.e. test whether the first number of the result of the `Mixer` task is "1"). The number of iterations for the convergence loop is unknown at compile time.

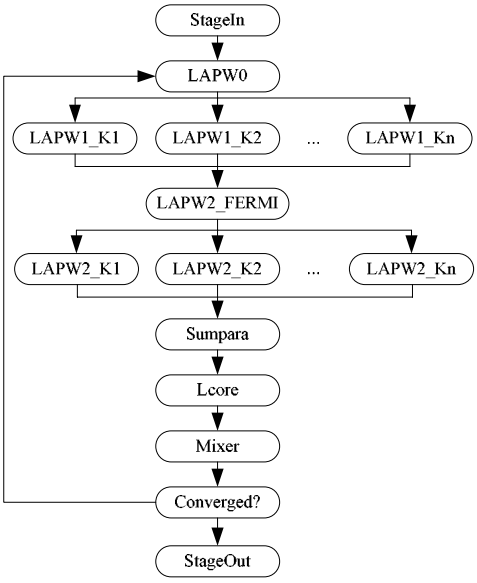


Figure 22: The WIEN2k workflow

We developed an AGWL representation for the workflow and describe a representative excerpt of it in Figure 23. Firstly, the ATD file `atd.xml` is imported, in which the definition of activities like `LAPW0`, `LAPW1`, etc. are included. Next, the activity `StageIn` is invoked to prepare for the execution of the while loop `whileConv`. In this while loop, the activity `actLAPW1`, the `parallelFor` loop `pfLAPW1`, the activity `actLAPW2_FERMI`, the `parallelFor` loop `pfLAPW2` and the activities `Sumpara`, `LCore` and `Mixer` are invoked sequentially. The value of `val` to exit the loop can be changed after each loop by the data-out port of the activity `Converged`, which is refered by the `loopSource`. In the `parallelFor` loop `pfLAPW1` and `pfLAPW2`, the activities are executed in parallel. Finally, the `outVal` of the workflow `wfWien2k` is returned as the result.

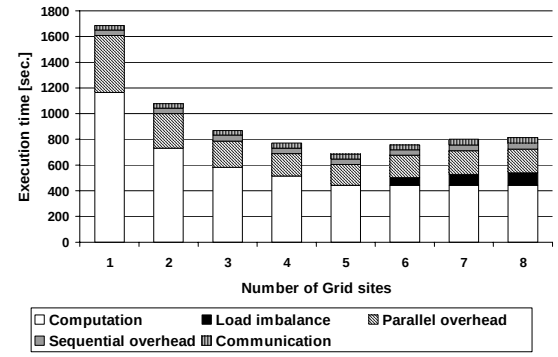
The AGWL representation of WIEN2k is compiled into a CGWL representation and executed in the ASKALON Grid environment [13], which is currently developed by the Distributed and Parallel Systems Group at the University of Innsbruck. ASKALON supports the performance-oriented execution of workflows specified in AGWL/CGWL by providing a rich set of services, which includes *resource broker*,

resource monitor, information service, workflow executor, (meta-) scheduler, performance prediction, and performance analysis services. All of these services are developed on top of a low-level Grid infrastructure implemented by the Globus toolkit, which provides a uniform platform for secure job submission, file transfer, discovery, and resource monitoring. ASKALON is deployed on the Austrian Grid infrastructure that aggregates several Grid sites across the country of Austria. We tested the performance of the WIEN2k Grid workflow respectively on 1, 2, 3, 4, 5, 6, 7 and 8 Grid sites. A large problem case (called *atype*) with three different problem sizes identified by the number of parallel *k-points* (100, 200 and 250) has been selected. Gescher, a local cluster used by the Wien2k scientists for their experiments, is used as the reference measurement for the single site execution. Figure 24 shows the performance results.

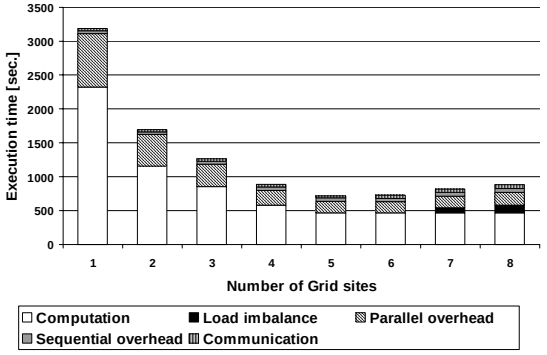
```
<agwl>
  <importATD url=".../wien2k/atd.xml" name="w2kAtd"/>
  <workflow name="wfWien2k">
    <dataIn name="fileIn0" source="a repository"/>
    <!-- some other "dataIn"s for workflow "wfWien2k" -->
    <body> <sequence>
      <activity name="Stagein" type="w2kAtd/Stagein">
        <!-- some "dataIn"s for activity "Stagein" -->
        <dataOut name="numOfProc" />
      </activity>
      <while name="whileConv">
        <!-- some "dataIn"s for while loop "whileConv" -->
        <dataIn name="val" loopSource="actConv/outVal">
          <value>true</value>
        </dataIn>
        <condition>val='true'</condition>
        <loopBody>
          <activity name="actLAPW0" type="w2kAtd:LAPW0">
            <!-- some "dataIn"s/"dataOut"s -->
            <dataOut name="outFileVsp"/>
          </activity>
          <parallelFor name="pfLAPW1">
            <dataIn name="fVsp" source="actLAPW0/outFileVsp"/>
            <!-- some "dataIn"s -->
            <loopCounter name="i" from="0"
                         to="Stagein/numOfProc" step="1"/>
            <loopBody> <sequence>
              <activity name="actLAPW1" type="w2kAtd:LAPW1">
                <!-- some "dataIn"s/"dataOut"s -->
              </activity>
            </sequence> </loopBody>
            <!-- some "dataOut"s for parallelFor "pfLAPW1" -->
          </parallelFor>
          <activity name="actLAPW2_FERMI"> ... </activity>
          <parallelFor name="pfLAPW2"> ... </parallelFor>
          <!-- some activities: Sumpara, ..., Mixer -->
          <activity name="Converged" type="w2kAtd/Converged">
            <!-- some other "dataIn"s and "dataOut"s -->
            <dataOut name="outVal"/>
          </activity>
        </loopBody>
        <dataOut name="outVal" source="whileConv/val"/>
      </while>
    </sequence> </body>
    <dataOut name="outVal" source="whileConv/outVal" />
  </workflow>
</agwl>
```

Figure 23: Excerpt from the WIEN2k AGWL representation

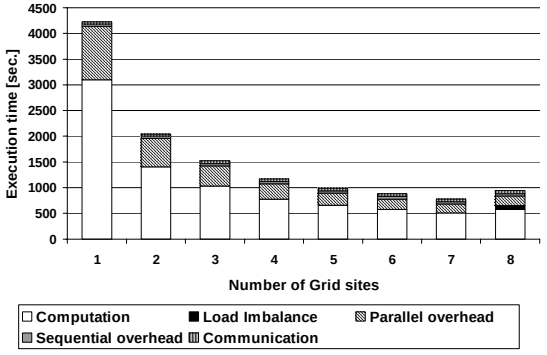
Performance results demonstrate that by migrating from the local Gescher cluster to a distributed Grid environment, good performance results are achieved. The speedup improves with larger problem sizes indicated by the parallel *k-points* (see Figure 24 (d)). The improvement comes from the parallel execution of *k-points* on multiple Grid sites that significantly decreases the computation time. The parallel overhead



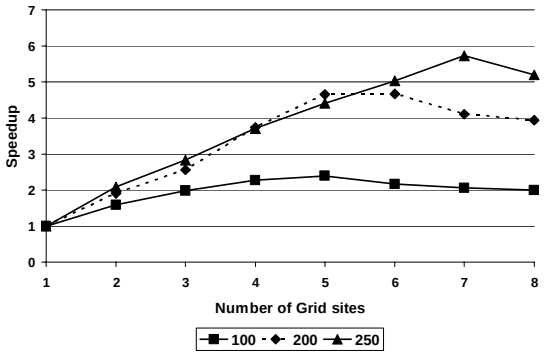
(a) 100 parallel *k-points*



(b) 200 parallel *k-points*



(c) 250 parallel *k-points*



(d) Speedup

Figure 24: Performance results of WIEN2k

decreases by increasing the number of Grid sites with constant number of k-points because fewer tasks are scheduled to a single Grid site. The sequential overhead remains relatively constant, but its ratio to the overall execution time is smaller for large problem sizes. The communication overhead of this application is negligible since we schedule `LAPW1` and `LAPW2` to Grid sites with a single NFS file system. The communication overhead becomes more significant with increasing number of Grid sites hosting different file systems. Typically for a scalability study, the workflow performance deteriorates beyond a certain machine size (e.g., 6 for 100 or 200 *k-points*, and 8 for 250 *k-points*). This is due to sites containing slower processors that are included into the Grid infrastructure, which causes load imbalance of the workflow application.

6. Conclusions and Future Work

In this paper, we presented our work on the Abstract Grid Workflow Language (AGWL), which is a novel XML-based language for the specification of Grid workflow applications at a high level of abstraction. AGWL allows the user to concentrate on describing scientific workflow without dealing with implementation or level details of the underlying Grid infrastructure. It is a powerful and user-oriented workflow language that has been tailored for scientific performance-oriented Grid workflow applications. AGWL provides advanced workflow constructs to facilitate the parallel execution of workflows, to retrieve and store data from and to data repositories, and to modularize and reuse workflows. Properties and constraints can be specified to optimize and steer workflow execution by the underlying Grid middleware.

We have integrated AGWL in the ASKALON Grid Application development environment. AGWL has been extensively used for the specification of Grid workflow applications in the field of material science, river modeling, astrophysics, and finance modeling.

Currently, we are in the process to further enhance AGWL to alleviate automatic transformation from AGWL to CGWL. Moreover, we are working to improve our workflow enactment engine for AGWL/CGWL representations.

References

[1] I. Foster and C. Kesselman (editors), The Grid Blueprint for a Future Computing Infrastructure, Morgan Kaufmann Publishers, USA, 1999.

[2] Layna Fischer, *The Workflow Handbook 2004*, Published by Future Strategies Inc., Lighthouse Point, FL, USA. 2004

[3] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana, Business Process Execution Language for Web Services Specification, version 1.1. IBM, Microsoft, BEA, SAP and Siebel Systems, May 5, 2003.

[4] Y. Huang, SWFL: Service Workflow Language. Technical report, Welsh e-Science Centre, Cardiff University, 2003.

[5] F. Leymann, Web Services Flow Language (WSFL 1.0), Technical report, IBM Software Group, May 2001.

[6] Jia Yu and Rajkumar Buyya, A Novel Architecture for Realizing Grid Workflow using Tuple Spaces, *Proceeding of Fifth IEEE/ACM International Workshop on Grid Computing*, Pittsburgh, PA, November 2004.

[7] P. Wagstrom, S. Krishnan, and G. von Laszewski, GSFL: A Workflow Framework for Grid Services, Argonne National Laboratory, August 2002.

[8] P. Avery, I. Foster, GriPhyN: Grid Physics Network, Technical report, August 2004.

[9] Condor Team, The directed acyclic graph manager, http://www.cs.wisc.edu/condor/dagman

[10] McCann, Karen M., Maurice Yarrow, Adrian DeVivo, and Piyush Mehrotra, *ScyFlow:An Environment for the Visual Specification and Execution of Scientific Workflows*, In Proceedings of Workflow in Grid Systems Workshop in GGF10, at Berlin, Germany,March, 2004.

[11] JunWei Cao, et al., *GridFlow: Workflow Management for Grid Computing*, 3rd International Symposium on Cluster Computing and the Grid, Tokyo, Japan, May 12 - 15, 2003.

[12] Ian Taylor, Matt Shields, Ian Wang and Roger Philp, *Distributed P2P Computing within Triana: A Galaxy Visualization Test Case*, IPDPS 2003 Conference, April 2003

[13] Thomas Fahringer, Alexandru Jugravu, Sabri Pllana, Radu Prodan, Clovis Seragiotto Junior, and Hong-Linh Truong. ASKALON: A Tool Set for Cluster and Grid Computing. Concurrency and Computation: Practice and Experience. http://dps.uibk.ac.at/askalon/

[14] S. Pllana, T. Fahringer, J. Testori, S. Benkner, and I. Brandic. Towards an UML Based Graphical Representation of Grid Work Applications. In The 2nd Eu-ropean Across Grids Conference, Nicosia, Cyprus, January 2004. Springer-Verlag.

[15] XSL Transformations (XSLT) version 1.0, http://www.w3.org/TR/xslt

[16] XML Path Language (XPath) version 1.0 http://www.w3.org/TR/xpath

[17] 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, Vienna University of Technology, 2001.