Scientific Programming 15 (2007) 193–211
IOS Press

Specification and runtime workflow support in the ASKALON Grid environment[1]

Radu Prodan
Institute of Computer Science, University of Innsbruck, Technikerstraße 21a, A-6020 Innsbruck, Austria
Tel.: +43 512 507 6445; Fax: +43 512 507 2758; E-mail: radu@dps.uibk.ac.at

Abstract. We describe techniques to support the runtime execution of scientific workflows in the ASKALON Grid environment. We present a formal model and three middleware services that support in combination the effective execution in heterogeneous and dynamic Grid environments: performance prediction, scheduling, and enactment engine. We validate our techniques with concrete experimental results for two real-world applications executed in the Austrian Grid environment.

Keywords: Grid computing, scientific workflows, performance prediction, scheduling, enactment engine

1. Introduction

The workflow model based on the loosely-coupled coordination of atomic activities emerged as one of the most interesting paradigms in the Grid community for programming or porting scientific applications to Grid environments. To meet this need numerous efforts [1,4,7,9,12,13,16,17,21], among which is the ASKALON project [10], are currently developing integrated environments to support the development cycle of scientific Grid workflows through graphical modelling tools, XML-based specification languages, or middleware services for advanced resource management, scheduling, prediction, reliable execution, and monitoring.

In this paper, we present new techniques developed within the ASKALON project to support the development of scientific workflows in Grid infrastructures. First of all, we present a model that tries to formally cover the most important constructs that we encountered in several real-world applications which we designed as scientific workflows (see Section 2). A performance prediction service (see Section 3) performs an offline training phase based on a well-defined experimental design to predict the execution times of workflow activities on Grid sites with a reduced number of experiments. A scheduling service (see Section 4) converts complex hierarchical workflows in flat Directed Acyclic Graphs (DAG) that can be effectively mapped onto the Grid using optimisation heuristics such as genetic or list scheduling algorithms. An enactment engine (see Section 5) simplifies the scheduled workflow using a partitioning algorithm to reduce the middleware overheads required for achieving improved performance. We compare our approach with the relevant related work in Section 6 and conclude in Section 7.

2. Workflow model

We present in this section a generic abstract model for formally representing large scale and complex scientific workflows in Grid environments. Our representation is generic and independent of any language or grammar as the underlying implementation platform. For example, we implemented our model through the XML-based Abstract Grid Workflow Language (AGWL) [11].

[1]This work is supported by the European Union through IST-004265 CoreGRID and IST-034601 edutain@grid projects.

Definition 1. We define a *workflow* as a DAG: $\mathcal{W} = (Nodes, C\text{-}edges, D\text{-}edges, IN\text{-}ports,\ OUT\text{-}ports)$, where $Nodes$ is the set of activities, $C\text{-}edges = \bigcup_{N_1,N_2 \in Nodes} (N_1, N_2)$ is the set of *control flow dependencies*, $D\text{-}edges = \bigcup_{N_1,N_2 \in Nodes} (N_1, N_2, D\text{-}port)$ is the set of *data flow dependencies*, $IN\text{-}ports$ is the set of workflow *input data ports*, and $OUT\text{-}ports$ is the set of *output data ports*. An *activity* $N \in Nodes$ is a mapping from a set of input data ports $IN\text{-}ports^N$ to a set of output data ports $OUT\text{-}ports^N$: $N : IN\text{-}ports^N \rightarrow OUT\text{-}ports^N$. A *data port* $D\text{-}port \in IN\text{-}ports^N \times OUT\text{-}ports^N$ is an an association between a unique *identifier* (within the workflow representation) and a well-defined *type*: $D\text{-}port = (identifier,\ type)$.

The type of a data port is instantiated by the type system supported by the underlying implementation language, e.g. the XML schema. The most important data type in our experience that shall be supported for Grid workflows is *file* along side other basic types such as integer, float, or string. An activity $N \in Nodes$ can be of two kinds. (1) *Computational activity* or *atomic activity* represents an atomic unit of computation such as a legacy sequential or parallel application. (2) *Composite activity* is a generic term for an activity that aggregates multiple (atomic and composite) activities according to one of the following four patterns: (i) *parallel loop activity* allows the user to express large-scale workflows consisting of hundreds or thousands of atomic activities in a compact manner; (ii) *sequential loop activity* defines repetitive computations with possibly unknown number of iterations (e.g. dynamic convergence criteria that depend on the runtime output data port values computed within one iteration); (iii) *conditional activity* models `if` and `switch`-like statements that activate one from its multiple successor activities based on the evaluation of a boolean condition; (iv) *workflow activity* is introduced for modularity and reuse purposes, and is recursively defined according to Definition 1.

For this particular paper, of special importance is the *parallel loop* (similar to a parameter sweep) which we represent as a tuple: $N_{par} = (N_{body}, IN\text{-}ports^{N_{par}},\ OUT\text{-}ports^{N_{par}})$, where: (1) $\exists(D\text{-}port_{card},$ *integer*$) \in IN\text{-}ports^{N_{par}}$ a predefined *cardinality input port* of type integer that defines the runtime cardinality of the parallel loop (i.e. number of parallel activities), denoted as $|N_{par}|$; (2) N_{body} is an atomic or composite activity representing the parallel loop body of which $|N_{par}|$ independent instances are executed. The cardinality port can be instantiated either statically or at runtime, for example from one output port of a predecessor activity through a data flow dependency.

3. Performance prediction

The performance prediction service is responsible for predicting the execution times of atomic activities onto given Grid sites needed for scheduling purposes. We employ a prediction model based on historical data collected through a well-defined *experimental design* and *training phase*. Specifically in our work, the general purpose of the experimental design phase is to set a strategy for experiments to get the maximum performance training information to support its prediction later in minimum numbers of experiments.

The factors affecting the response variable which we currently consider are the problem size p incorporating the range of instances for each parameter variable, the Grid size g comprising all the Grid sites (i.e. parallel computers), and the machine size m including all different processor numbers on a Grid site. To reduce the experimental space from $p \times g \times m$, we introduce a *Performance Sharing and Translation* (*PST*) mechanism based on several multi-parameter performance relativity properties, experimentally observed for our case study applications. For example, embarrassingly parallel applications that scale linearly with the machine and problem size benefit from the following inter- and intra-platform performance relativity properties.

Inter-platform PST specifies that the performance behaviour $P_g(A,p)$ of an application A for a problem size p relative to another problem size r on a Grid site g is the same as that of the same problem sizes on another Grid site h: i.e. $\frac{P_g(A,p)}{P_g(A,r)} \simeq \frac{P_h(A,p)}{P_h(A,r)}$.

Similarly, *intra-platform PST* specifies that the performance behaviour of an embarrassingly parallel application A on a Grid site g for a machine size m relative to another machine size n for a problem size p is similar to that for another problem size q, i.e. $\frac{P_g(A,p,m)}{P_g(A,p,n)} \simeq \frac{P_g(A,q,m)}{P_g(A,q,n)}$.

We choose one Grid site (the fastest based on previous runs) as the reference site and execute the complete set of experiments on it based on the cross product of the input problem size parameters. Later, we make one single experiment on each of the other Grid sites and use the reference values to calculate the predictions for other platforms using the inter-platform PST and thus minimise problem size combinations with the Grid size. Similarly, to minimise machine size combinations with the Grid size, we make complete set of experiments with one reference machine size and later make one single experiment each for each of the other machine sizes to translate the reference performance values for other machine sizes using intra-platform PST.

By means of inter-platform PST, the total number of experiments reduces from $p \times m \times g$ to $p \times m + (g - 1)$ for parallel computers, and from $p \times m$ to $p + g - 1$, for single processor machines. By introducing intra-platform PST, we reduce total number of experiments for parallel machines as Grid sites further to a linear complexity of $p + (m - 1) + (g - 1)$.

3.1. Experiments

We report experiments for two real-world applications, WIEN2k [20] and Invmod [23], in the Austrian Grid environment. We will present these workflows in detail in Sections 4.3 and 5.3, while in this section we concentrate on the two most computational expensive activities of these workflows: LAPW1 and WasimB2C.

We analysed the scalability of our experimental design strategy by varying the problem size from 10 to 200 for fixed values of the remaining factors: 10 Grid sites with machine size of 20 and 50 single processor machines. We observed a reduction in the total number of experiments (i.e. previously denoted as $p \times g \times m$) from 96% to 99%, as shown in Fig. 1a. A reduction from 77% to 97% in the total number of experiments was observed when we varied the machine size from one to 80, for fixed factors of 10 parallel machines, 50 single processor Grid sites and problem size of five. From another perspective, we observed that the total number of experiments increased from 7% to 9% when the Grid size was increased from 15 to 155, for the fixed factors of five parallel machines with machine size of 10 and problem size 10. We observed an overall reduction of 78% to 99% when we varied all factors simultaneously: five parallel machines with machine size from 1 to 80, single processor Grid sites from 10 to 95, and problem size from 10 to 95.

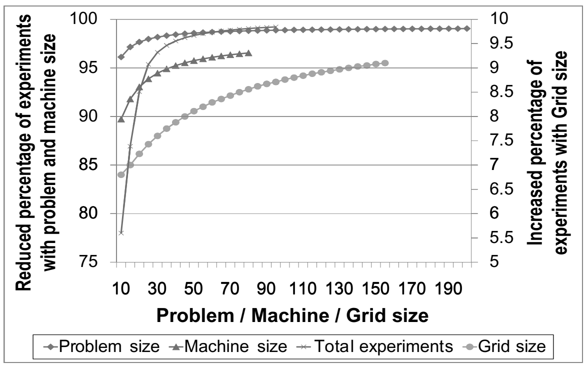
We comparatively show the predicted results using the inter-platform PST method versus the measured values for LAPW1 and iWasimB2C in Figs 1b and 1c, respectively. The lowest curve represents the execution values on the base Grid site whose values are used in the PST mechanism. In both cases the curves of the measured and predicted values are very similar, however, we can see that they are closest to each other near to the reference problem size and they distance from each other as the distance from the reference problem size increases. Due to this reason and whenever possible, we take the reference problem size as close as possible to the target value to be predicted. We observed that the average variation in the predicted values from the measured value, if made on the basis of maximum available value, is at most 10% which yields 90% accuracy in the prediction. As we get more data during the actual runs, the probability of finding closer parameter values other than the one calculated in the training phase increases which further improves the prediction accuracy.

4. Scheduler

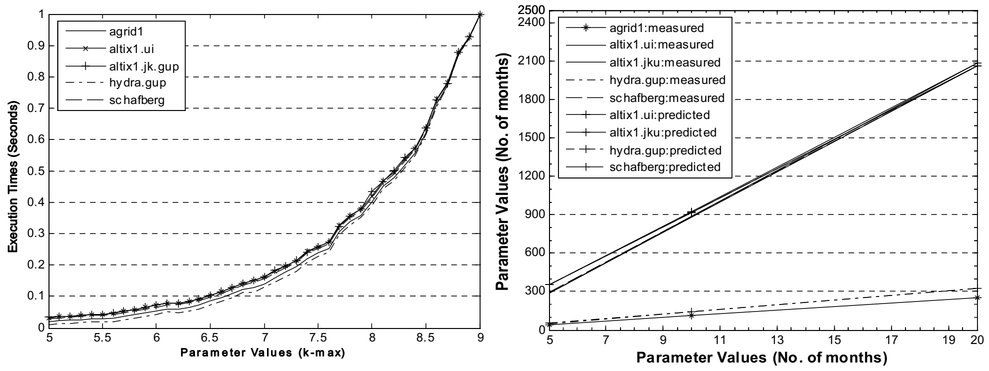
One major difficulty of the scheduling service is the fact that the objective function (i.e. execution time) cannot be precisely calculated for dynamic workflows comprising conditional activities or sequential and parallel loops with unknown number of iterations. We therefore approach the workflow scheduling problem in two phases implemented by two modules: (1) *workflow converter* (see Section 4.1) transforms compact hierarchical workflow representations into flat static DAGs; (2) *scheduling engine* (see Section 4.2) implements heuristic algorithms for achieving good mappings of the generated DAGs onto the Grid resources.

4.1. Workflow converter

The purpose of the *workflow converter* is to transform a hierarchical Directed Graph (DG)-based scientific workflows containing sequential loops into a flat DAG of atomic activities suitable for optimised scheduling using heuristic algorithms. There are four constructs corresponding to the four composite activities described in Section 2 which are handled by the converter through four corresponding transformations: conditional activities through branch



(a) Experiment reduction with problem, machine, and Grids ize.



(b) Relative values of LAPW1.

(c) Relative values of WasimB2C.

Fig. 1. Performance prediction results.

expansion, sequential and parallel loops through loop unrolling, and sub-workflows through workflow inlining. A complete specification of the conversion algorithm can be found in [19]. These transformations usually require additional prediction information such as the probability of execution of each branch in conditional activities or the number of iterations within sequential and parallel loops, which we compute from historical data. Transformations based on correct assumptions can imply substantial performance benefits, while incorrect assumptions require appropriate runtime adjustments such as undoing existing optimisations or rescheduling based on the new information available (see Section 5.2.1).

4.2. Scheduling engine

The *scheduling engine* is responsible for the actual mapping of a workflow application converted into a DAG onto the Grid resources such that the execution time objective function is minimised. The scheduling engine employs the performance prediction service to obtain expected execution times of individual activities.

Definition 2. Let $\mathcal{W} = (Nodes, C\text{-}edges, D\text{-}edges, IN\text{-}ports, OUT\text{-}ports)$ denote a scientific workflow application. We evaluate a workflow schedule by constructing the *Gantt chart* where the end timestamp of each activity $N \in Nodes$ is recursively defined according to the function:

$$end : Nodes \to \mathbb{R}_+^*,$$

$$end(N) = \begin{cases} T_N, & pred(N) = \emptyset \\ \max_{\forall N' \in pred(N)} \left\{ end(N') + \sum_{\forall (N', N, D\text{-}port) \in D\text{-}edges} T_{D\text{-}port} \right\} + T_N, & pred(N) \neq \emptyset, \end{cases}$$

Table 1
The HEFT weight and rank calculations for the sample workflow depicted in Fig. 2

(a) Computational activity ranks.					(b) Data transfer activity ranks.				
	P_1	P_2	P_3	$\overline{\omega}$		(P_1, P_2)	(P_1, P_3)	(P_2, P_3)	$\overline{\omega}$
N_1	5	8	8	7	*D-port*$_1$	6	4	5	5
N_2	9	13	11	11	*D-port*$_2$	4	2	3	3
N_3	3	4	5	4	*D-port*$_3$	7	4	7	6
N_4	7	10	10	9	*D-port*$_4$	1	1	4	2

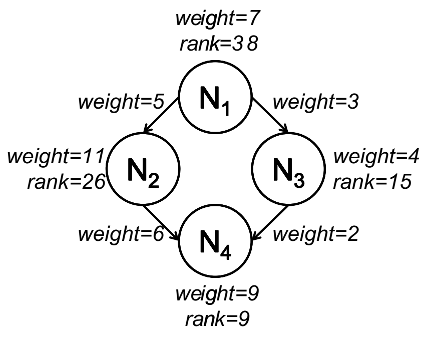


Fig. 2. The HEFT weights and ranks for a sample workflow.

where $\mathbb{R}_+^*$ denotes the set of real positive numbers, $pred(N) = \bigcup_{\forall(N',N)\in C\text{-}edges} N'$ is the set of predecessors of activity N, and $\emptyset$ the empty set.

In this section we present two heuristics that we use to implement the scheduling engine, along side a genetic algorithm that we described in [18]: (1) *Heterogeneous Earliest Finish Time* (*HEFT*) [27] algorithm that is a list scheduling heuristic purposely tuned for scheduling complex DAGs in heterogeneous environments; (2) a *myopic* just-in-time algorithm acting like an opportunistic resource broker similar to Condor DAGMan [1].

4.2.1. *Heterogeneous Earliest Finish Time Algorithm* (*HEFT*)

The HEFT algorithm, illustrated in pseudocode in Algorithm 1, is an extension of the classical list scheduling algorithm for heterogeneous environments which consists of three distinct phases: (1) the *weighting phase* (lines 3–8); (2) the *ranking phase* (lines 9–20); (3) the *mapping phase* (lines 21–24). We explain these three phases through a concrete example depicted in Fig. 2.

Weighting during the weighting phase (lines 3–8) adjusted for heterogeneous Grid environments, we assign weights to the workflow activities equal to their predicted execution time which we estimate based on the training phase that we presented in Section 3. Afterwards, we calculate the weight associated with a computational activity $N \in Nodes$ as the average value of the predicted execution times $T_N^{\mathtt{P}}$ on every individual processor $\mathtt{P}$ available on the Grid (lines 3–5): $\overline{w}_N = avg_{\forall \mathtt{P}\in\mathtt{GRID}}\left\{T_N^{\mathtt{P}}\right\}, \forall N \in Nodes$. Similarly, we compute the weight associated to a data dependency as the average of the predicted transfer times across all pairs of Grid sites (rather than processors – lines 6–8): $\overline{w}_{D\text{-}port} = avg_{\forall(M_1,M_2)\in\mathtt{GRID}}\left\{T_{D\text{-}port}^{(M_1,M_2)}\right\}, \forall\,(N_1, N_2, D\text{-}port) \in D\text{-}edges$.

In the example depicted in Fig. 2 and Table 1, the Grid consists of three processors $\mathtt{P}_1$, $\mathtt{P}_2$, and $\mathtt{P}_3$, therefore, the weight of activity N_1 is calculated as follows: $\overline{w}_{N_1} = \frac{T_{N_1}^{\mathtt{P}_1}+T_{N_1}^{\mathtt{P}_2}+T_{N_1}^{\mathtt{P}_3}}{3} = \frac{5+8+8}{3} = 7$, and similarly: $\overline{w}_{D\text{-}port_1} = \frac{T_{D\text{-}port_1}^{(\mathtt{P}_1,\mathtt{P}_2)}+T_{D\text{-}port_1}^{(\mathtt{P}_1,\mathtt{P}_3)}+T_{D\text{-}port_1}^{(\mathtt{P}_2,\mathtt{P}_3)}}{3} = \frac{6+4+5}{3} = 5$. Table 1 displays the weights of all workflow activities calculated using the same formulas.

Ranking The ranking phase (lines 9–20) is performed by traversing the workflow graph upwards and assigning a rank value to each activity equal to the weight of the activity plus the maximum rank value of all the successors (line 13): $\overline{R}_N = \max_{\forall N_{succ}\in succ(N)}\left\{\overline{w}_N + \overline{R}_{N_{succ}}\right\}$, where $succ(N) = \bigcup_{\forall(N,N')\in C\text{-}edges} N'$. For example, the rank of the activity N_1 is calculated as follows: $\overline{R}_{N_1} = \max\left\{\overline{w}_{N_1} + \overline{w}_{(N_1,N_2)} + \overline{R}_{N_2}, \overline{w}_{N_1} + \overline{w}_{(N_1,N_2)} + \overline{R}_{N_3}\right\} =$

Algorithm 1 The HEFT algorithm.

```
 1: function HEFT($\mathcal{W}$, GRID)
 2:     $\mathcal{W} = (Nodes, C\text{-}edges, D\text{-}edges, IN\text{-}ports, OUT\text{-}ports)$          ▷ Precondition
 3:     for all $N \in Nodes$ do                                                  ▷ Weighting phase
 4:         $\overline{w}_N \leftarrow \frac{\sum_{\forall P \in \mathrm{GRID}} T_N^P}{|\mathrm{GRID}|}$              ▷ |GRID| = no. of processors in GRID
 5:     end for
 6:     for all $IN\text{-}ports \bigcup_{\forall (N_1, N_2, D\text{-}port) \in D\text{-}edges} D\text{-}port \cup OUT\text{-}ports$ do
 7:         $\overline{w}_{D\text{-}port} \leftarrow \frac{\sum_{\forall P_1 \neq P_2 \in \mathrm{GRID}} T_{D\text{-}port}^{(P_1,P_2)}}{C^2_{|\mathrm{GRID}|}}$ ▷ $C^2_{|\mathrm{GRID}|}$ = combination of |GRID| elements taken 2 at a
    time
 8:     end for
 9:     $List_{C\text{-}edges} \leftarrow C\text{-}edges$                                          ▷ Ranking phase
10:     $List_{Nodes} \leftarrow Nodes$
11:     while $List_{C\text{-}edges} \neq \emptyset$ do
12:         for all $N \in List_{Nodes} \wedge (succ(N) \cap List_{C\text{-}edges} = \emptyset)$ do
13:             $\overline{R}_N \leftarrow \overline{w}_N + \max_{\forall N_{succ} \in succ(N)} \{\overline{w}_{Nsucc} + \overline{w}_{Dsucc}\},$
                            where $\overline{w}_{Dsucc} = \begin{cases} \overline{w}_{D\text{-}port}, & (N, Nsucc, D\text{-}port) \in D\text{-}edges; \\ 0, & (N, Nsucc, D\text{-}port) \notin D\text{-}edges \end{cases}$
14:
15:             $List_{C\text{-}edges} \leftarrow List_{C\text{-}edges} \setminus (pred(N) \times N)$
16:
17:             $List_{Nodes} \leftarrow List_{Nodes} \setminus N$
18:         end for
19:     end while
20:     $RL \leftarrow \mathrm{SORT}(Nodes, \overline{R}_N)$                          ▷ Sort the activities based on ranks
21:     for all $i \in [1..|RL|]$ do                                              ▷ Mapping phase
22:         $N \leftarrow RL_i$
23:         $\mathcal{S}_N \leftarrow \mathrm{P}$, where $end(N, \mathrm{P}) = \min_{\forall P \in \mathrm{GRID}} \{end(N, P)\}$       ▷ end function was defined in
    Definition 2
24:     end for                                                         ▷ $\mathcal{S}_N$ = schedule of activity $N$
25:     return $\mathcal{S}_\mathcal{W}$                                           ▷ $\mathcal{S}_\mathcal{W}$ = Schedule of workflow $\mathcal{W}$
26: end function
```

$\max\{7+5+26, 7+3+15\} = 38$. The activity list is then sorted in a descending order of the activity ranks (line 20), i.e. N_1, N_2, N_3, and N_4.

Mapping Finally in the mapping phase (lines 21–24), each ranked activity N is scheduled onto the processor $\mathcal{S}_N$ that delivers its earliest completion time according to Definition 2, i.e.:

$$\begin{aligned}
&end\,(N_1) = \min\{5, 8, 8\} = 5 && \Rightarrow \mathcal{S}_{N_1} = \mathtt{P}_1; \\
&end\,(N_2) = \min\{5+0+9, 5+6+13, 5+4+11\} = 14 && \Rightarrow \mathcal{S}_{N_2} = \mathtt{P}_1; \\
&end\,(N_3) = \min\{14+0+3, 5+4+4, 5+2+5\} = 12 && \Rightarrow \mathcal{S}_{N_3} = \mathtt{P}_3; \\
&end\,(N_4) = \min\{\max\{14+0, 12+1\}+7, \max\{14+7, 12+4\}+10, \\
&\qquad\qquad \max\{14+4, 12+0\}+10\} = 21 && \Rightarrow \mathcal{S}_{N_4} = \mathtt{P}_1.
\end{aligned}$$

4.2.2. *Myopic algorithm*

To compare the effectiveness of the scheduling heuristics, we developed a simple and inexpensive method which makes the mapping based on local optimal decisions similar to the matchmaking mechanism performed by a resource broker like Condor DAGMan [1] (see Algorithm 2). The algorithm traverses the workflow in the top-down direction (lines 5 and 6), analyses every activity separately, and assigns it to the processor which delivers the earliest completion time (line 7).

4.2.3. *Layered partitioning*

Layered partitioning combines HEFT and the myopic algorithms and considers as input to the conversion algorithm only a sub-workflow of a given depth of n activities, calculated for a workflow $\mathcal{W} = (Nodes, C\text{-}edges, D\text{-}edges, IN\text{-}ports^{\mathcal{W}}, OUT\text{-}ports^{\mathcal{W}})$ as follows: $\mathcal{W}_n = (Nodes_n, C\text{-}edges_n, D\text{-}edges_n, IN\text{-}ports^{\mathcal{W}_n},$

Algorithm 2 The myopic scheduling algorithm.

```
1:  function MYOPIC($\mathcal{W}$, GRID)
2:      $\mathcal{W} = (Nodes, C\text{-}edges, D\text{-}edges, IN\text{-}ports, OUT\text{-}ports)$      ▷ Precondition
3:      $List_{Nodes} \leftarrow Nodes$
4:      $List_{C\text{-}edges} \leftarrow C\text{-}edges$
5:      while $List_{Nodes} \neq \emptyset$ do
6:          for all $N \in List_{Nodes} \wedge (pred(N) \cap List_{C\text{-}edges} = \emptyset)$ do
7:              $S_N \leftarrow$ P, where $end(N, \mathrm{P}) = \min_{\forall P \in \mathrm{GRID}} \{end(N, P)\}$      ▷ end function was defined in
    Definition 2
8:              $List_{Nodes} \leftarrow List_{Nodes} \setminus N$
9:              $List_{C\text{-}edges} \leftarrow List_{C\text{-}edges} \setminus (N \times succ(N))$
10:         end for
11:     end while
12:     return $S_{\mathcal{W}}$      ▷ Workflow schedule
13: end function
```

$OUT\text{-}ports^{\mathcal{W}_n}$), where: (1) $Nodes_n \subseteq Nodes$; (2) $succ^m(N) \in Nodes_n, \forall N \in Nodes_n \wedge pred(N) = \emptyset \wedge \forall m \in [1..n]$; (3) $succ^{n+1}(N) \notin Nodes_n, \forall N \in Nodes_n \wedge pred(N) = \emptyset$; (4) $D\text{-}edges_n = \bigcup_{\substack{\forall (N_1, N_2, D\text{-}port) \in D\text{-}edges \\ \wedge N_1, N_2 \in Nodes_n}} (N_1, N_2, D\text{-}port)$; (5) $IN\text{-}ports^{\mathcal{W}_n} = IN\text{-}ports^{\mathcal{W}}$; (6) $OUT\text{-}ports^{\mathcal{W}_n} = \bigcup_{\substack{\forall N \in Nodes_n \wedge \\ succ(N) \notin Nodes_n}} OUT\text{-}ports^{N}$.

This method is similar to the one described informally in [7] and is more suitable for workflows with regular structures and a large number of activities (see Section 5.3), since it needs less scheduling time to compute optimised mappings of smaller sub-workflows while preserving the overall quality of the solution.

4.3. *Invmod*

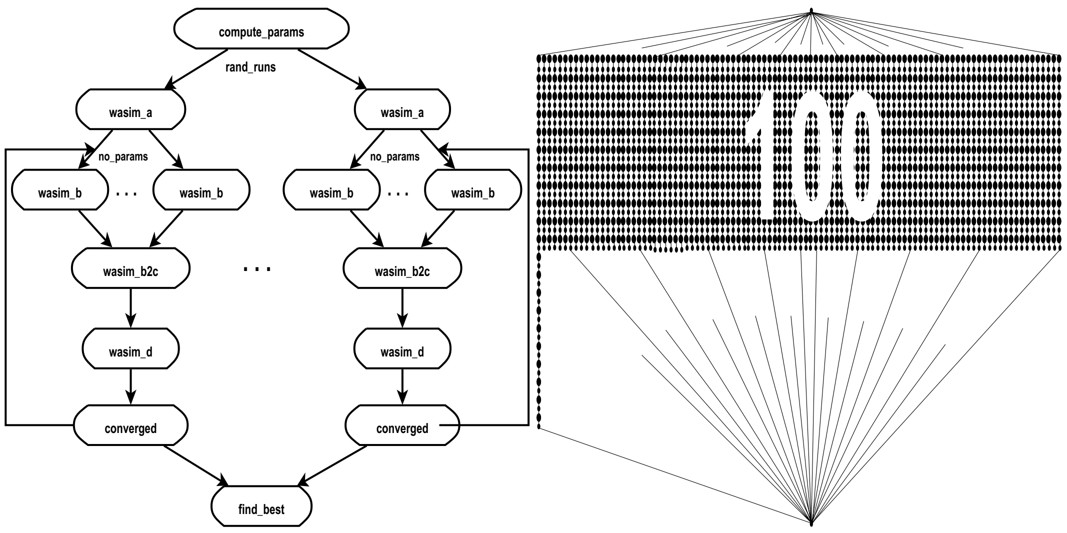
Invmod [23] is a hydrological application designed at the University of Innsbruck for calibration of parameters of the WaSiM tool developed at the Swiss Federal Institute of Technology Zurich. Invmod uses the Levenberg-Marquardt algorithm to minimise the least squares of the differences between the measured and the simulated runoff for a determined time period. We re-engineered the monolithic Invmod application into a Grid-enabled scientific workflow consisting of two levels of parallelism as depicted in Fig. 3: (1) each iteration of the outermost parallel loop called *random run* performs a local search optimisation starting from an arbitrarily chosen initial solution; (2) alternative local changes are examined separately for each calibrated parameter, which is done in parallel in the inner nested parallel loop. The number of sequential loop iterations is variable and depends on the actual convergence of the optimisation process, however, it is usually equal to the input maximum iteration number.

We performed the experiments on seven heterogeneous Grid sites of the Austrian Grid [22] infrastructure illustrated in Table 2, aggregating 116 processors in total. The Invmod workflow is a common case of strongly imbalanced workflows for which one of the outermost parallel loop iterations is significantly longer than the others due a different number of inner sequential loop iterations. In our case, the converted DAG consists of 100 parallel iterations, one of which contains 20 sequential iterations of the inner optimisation loop, while the other 99 iterations only contain 10 optimisation iterations each (see Fig. 3b). This means that one parallel iteration needs approximately approximately twice the execution time of the others.

The experimental results for the Invmod workflow illustrated in Fig. 4 explain how each of the three algorithms deals with such strongly imbalanced workflow structures. As expected, the myopic algorithm provides the worst results which are approximately 32% worse than HEFT. The genetic algorithm produces quite good results, however, it is worse than HEFT since it does not consider in the optimisation process the execution order of parallel activities scheduled on the same processor. In addition, we applied incremental scheduling using 10, 20, and 30 partitioning layers and compared the results against the full-ahead workflow scheduling consisting of 44 layers. For such strongly imbalanced workflows, the activities on this much longer critical path should be given priority which is

Table 2
Austrian Grid testbed for Invmod scheduling experiments

Site	*Architecture*	*Size*	*Processor*	*GHz.*	*Location*
agrid	NOW, Fast Ethernet	20	Pentium 4	1.8	Innsbruck
hydra	COW, Fast Ethernet	16	AMD 2000	1.6	Linz
agrid1	NOW, Fast Ethernet,	16	Pentium 4	1.8	Innsbruck
altix1.jku	ccNUMA, SGI Altix 3000	16	Itanium 2	1.6	Innsbruck
altix1.uibk	ccNUMA, SGI Altix 350	16	Itanium 2	1.6	Linz
schafberg	ccNUMA, SGI Altix 350	16	Itanium 2	1.6	Salzburg
gescher	COW, Gigabit Ethernet	16	Pentium 4	3	Vienna



(a) Original workflow. (b) Converted DAG.

Fig. 3. The Invmod scientific workflow.

well handled by the entire workflow scheduling strategy based on optimisation heuristics like HEFT and the genetic algorithm. Therefore, scheduling strategies based on workflow partitioning deliver worse results than those based on full workflow analysis, although their results are still better than the one found by the myopic algorithm. The genetic algorithm requires two orders of magnitude longer than list scheduling algorithms to converge to good solutions.

In addition, we applied the scheduling algorithms with and without prediction information to study the value and impact of the prediction availability. Scheduling with prediction information delivers between 25%–33% better results than without performance prediction when all activities are considered to have equal execution times.

5. Enactment engine

The enactment engine is a service responsible for the effective workflow execution in three major steps: (1) in the first step, the (XML-based) workflow representation is delivered to the scheduler for appropriate mapping onto the Grid, as presented in Section 4; (2) once the concrete workflow schedule is received, the engine simplifies the workflow through a partitioning algorithm (see Section 5.1); (3) during runtime, the workflow execution is dynamically improved by a dynamic steering algorithm (see Section 5.2).

5.1. Workflow partitioning

Our experience in running real-world applications in the Austrian Grid environment revealed that executing one computational activity on a remote Grid site contains on average about 10–20 seconds of overhead mainly due to

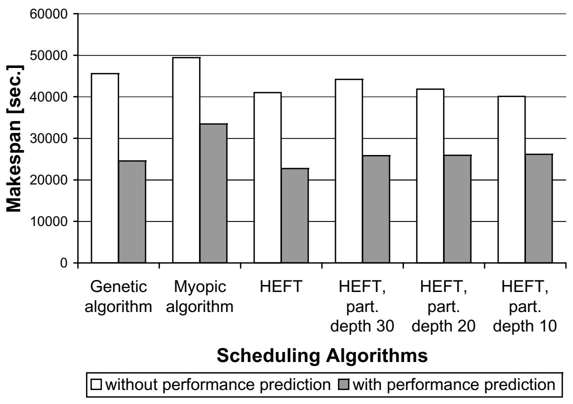


Fig. 4. The scheduling results for the Invmod workflow.

mutual authentication latency and polling for job termination. This overhead may be significantly larger if the access to Grid sites is performed through local job management systems and, therefore, becomes critical for large scientific workflows comprising hundreds to thousands of activities.

In this section we propose a *partitioning algorithm* to decrease the number of activities and data dependencies in a workflow. Determining the number of partitions of a set of n numbers is a classical NP-complete problem of combinatorial mathematics called the *n-th Bell number*. Some related partitioning approaches were already proposed to solve this problem although their algorithms have different goals [2,7].

Definition 3. We define a *workflow partition* as the largest sub-workflow $\mathcal{W}_P = (Nodes_P,\ C\text{-}edges_P, D\text{-}edges_P)$ with the following properties: (1) all activities are scheduled on the same Grid site: $\mathcal{S}_{N_1} = \mathcal{S}_{N_2}, \forall N, N_2 \in Nodes_P$; (2) there must be no control flow and data flow dependencies to / from activities that have predecessors / successors within the partition: $pred(N) = \emptyset \vee pred(N) \in Nodes_P, \forall N \in Nodes_P$.

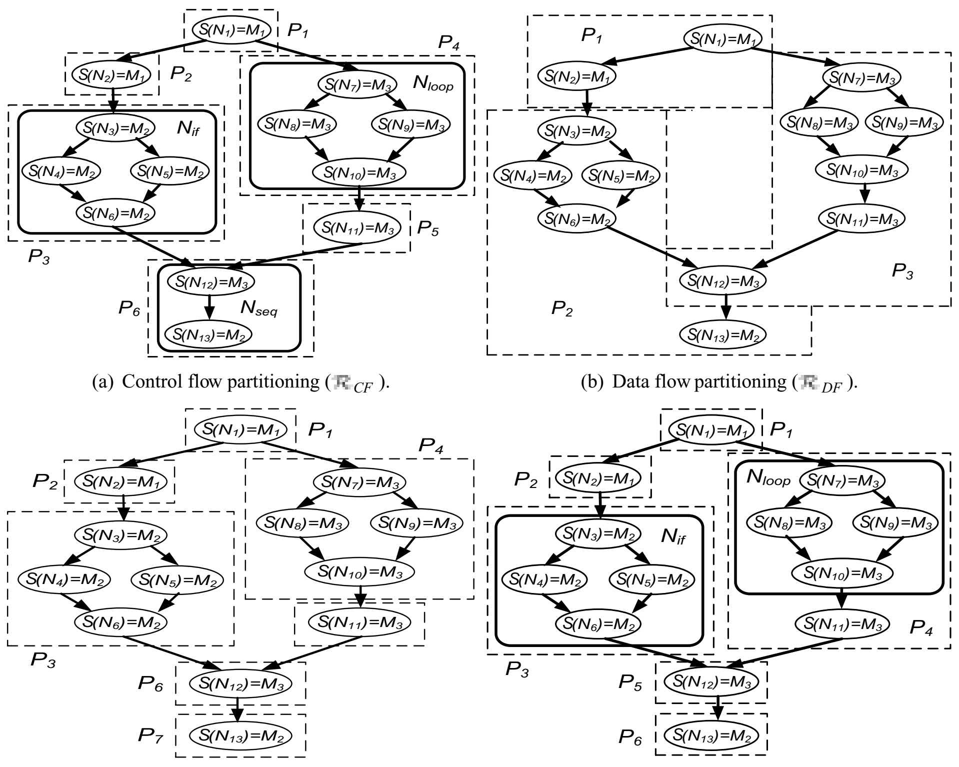
The goal of the partitioning algorithm is to generate a partitioned workflow denoted $\mathcal{W}_P = (Nodes_P, C\text{-}edges_P, D\text{-}edges_P)$ from a workflow $\mathcal{W} = (Nodes, C\text{-}edges, D\text{-}edges)$, where: $Nodes_P = \{P_1, \ldots, P_n\}$ is the set of workflow partitions that fulfil Definition 3, $\bigcap_{i=1}^{n} P_i = \emptyset$ (i.e. partitions are disjoint), $\bigcup_{i=1}^{n} P_i = Nodes$ (i.e. partitions cover all workflow activities), and n is minimum. We base our partitioning algorithm on graph transformation theory [3] as the formal background to rigourously express it. We define several rules for defining valid workflow partitions that aim to decrease the complexity of the algorithm (to polynomial) and create the set of cooperating workflow partitions.

Let $(\mathcal{W}, \mathcal{R})$ denote a workflow transformation system, where $\mathcal{R}$ denotes the set of graph transformation rules. We approach the workflow partitioning problem using a four step transformation sequence denoted as: $\left(\mathcal{W} \overset{\mathcal{R}_{CF}}{\Longrightarrow} \mathcal{W}_{CF}, \mathcal{W} \overset{\mathcal{R}_{DF}}{\Longrightarrow} \mathcal{W}_{DF}\right) \overset{\mathcal{R}_{M1}}{\Longrightarrow} \mathcal{W}' \overset{\mathcal{R}_{M2}}{\Longrightarrow} \mathcal{W}_P$, where: $\mathcal{W}_{CF} = (Nodes_{CF}, C\text{-}edges_{CF}, D\text{-}edges_{CF})$, $\mathcal{W}_{DF} = (Nodes_{DF},\ C\text{-}edges_{DF}, D\text{-}edges_{DF})$, $\mathcal{W}' = \left(Nodes', C\text{-}edges', D\text{-}edges'\right)$, and $\mathcal{W}_P$ are partition sets generated using different transformation rules that preserve the control and data flow dependencies of the original workflow $\mathcal{W}$. We omit the workflow input and output data ports for clarity reasons since they are irrelevant to our partitioning algorithm.

Step 1: $\mathcal{W} \overset{\mathcal{R}_{CF}}{\Longrightarrow} \mathcal{W}_{CF}$. Partition the workflow according to three control flow dependency rules $\mathcal{R}_{CF}$:

1. every activity of the workflow must belong to exactly one partition: $\forall N \in Nodes, \exists P \in Nodes_{CF} \wedge N \in P \wedge N \notin P' \wedge \forall P' \in Nodes_{CF} \setminus P$;
2. every partition is one composite or atomic activity. Currently we perform this step by using additional information provided by the user in the XML-based workflow representation [11] and mapping one composite activity (e.g. parallel activity) to one partition;
3. no control flow dependencies between intermediate activities in different partitions are allowed:

$$\forall N_1 \in P_1 \in Nodes_{CF} \wedge (pred\,(N_1) \in P_1 \vee succ\,(N_1) \in P_1) \wedge (\nexists\,(N_1, N_2) \in C\text{-}edges_{CF}$$
$$\wedge \nexists\,(N_2, N_1) \in C\text{-}edges_{CF}, \forall N_2 \in P_2 \in Nodes_{CF})\,,$$



(a) Control flow partitioning ($\mathcal{R}_{CF}$). (b) Data flow partitioning ($\mathcal{R}_{DF}$).

(c) Partition merge ($\mathcal{R}_{M1}$). (d) Partitioned workflow ($\mathcal{R}_{M2}$).

Fig. 5. A workflow partitioning example.

where *pred* and *succ* denote the predecessor, respectively the successor of an activity in the workflow;

4. the number of activities inside one composite activity must be more than the average number of processors on one Grid site. We introduce this rule to avoid too fine grained partitions in the workflow that would start slave engines on sites with little workload.

For example, in Fig. 5a we partition all atomic activities of the composite activities N_{if}, N_{loop}, and N_{seq} into one partition, respectively, which produces the following control flow partitioning: $Nodes_{CF} = \{\{N_1\}, \{N_2\}, \{N_3, \ldots, N_6\}, \{N_7, \ldots, N_{10}\}, \{N_{11}\}, \ \{N_{12}, N_{13}\}\}$.

Step 2: $\mathcal{W} \overset{\mathcal{R}_{DF}}{\Longrightarrow} \mathcal{W}_{DF}$. Partition the original workflow according to three data flow dependency rules $\mathcal{R}_{DF}$:

1. each activity of the workflow must belong to exactly one partition: $\forall N \in Nodes, \exists P \in Nodes_{DF} \wedge N \in P \wedge N \notin P', \forall P' \in Nodes_{DF} \setminus P$;
2. the data dependencies between activities scheduled on the same Grid site are eliminated: $D\text{-}edges_{DF} = D\text{-}edges \setminus (N_1, N_2, D\text{-}port), \forall N_1, N_2 \in Nodes \wedge \mathcal{S}_{N_1} = \mathcal{S}_{N_2}$;
3. activities scheduled on the same Grid site belong to the same partition: $\forall N_1 \in P \in \mathcal{W}_{DF} \wedge \forall N_2 \in P \wedge \mathcal{S}_{N_1} = \mathcal{S}_{N_2}$.

Figure 5b displays the result of the data flow partitioning according to the schedule of the activities: $Nodes_{DF} = \{\{N_1, N_2\}, \{N_3, \ldots, N_6, N_{13}\}, \{N_7, \ldots, N_{11}, N_{12}\}\}$.

Step 3: $(\mathcal{W}_{CF}, \mathcal{W}_{DF}) \overset{\mathcal{R}_{M1}}{\Longrightarrow} \mathcal{W}'$. Merge the two sets $Nodes_{CF}$ and $Nodes_{DF}$ of control and data flow-based partitions into one partition set while preserving the control and data flow dependencies: $\mathcal{W}' = \bigcup_{\substack{\forall Nodes_1 \in Nodes_{CF} \\ \forall Nodes_2 \in Nodes_{DF}}} \{Nodes_1 \cap Nodes_2\}$. For our example in Fig. 5c we obtain: $Nodes' = \{\{N_1\}, \{N_2\}, \{N_3, \ldots, N_6\}, \{N_7, \ldots, N_{10}\}, \{N_{11}\}, \{N_{12}\}, \{N_{13}\}\}$.

Step 4: $\mathcal{W}' \overset{\mathcal{R}_{M2}}{\Longrightarrow} \mathcal{W}_P$. Since the partitioning may have been done too fine grain, we merge the partitions connected through control flow dependencies using the following two merge rules:

1. merge the partitions that are connected through control flow dependencies but have no data flow dependencies (i.e. they are scheduled on the same site:

$$\mathit{Nodes}_P = \bigcup_{\forall P_i \neq P_j \in \mathcal{W}'} \{\{P_i \cup P_j\} \setminus \{P_i\} \setminus \{P_j\} \,|\forall N_1 \in P_i \wedge \forall N_2 \in P_j \wedge$$
$$\nexists\,(N_1, N_2, \mathit{D\text{-}port}) \in \mathit{D\text{-}edges} \wedge (P_i, P_j) \in \mathit{C\text{-}edges}'\big\}\,;$$

2. in the final partition, there must be no control and data flow dependencies to/from activities that have predecessors/successors within the partitions. This is achieved by iteratively applying the following formula within fixed point algorithm until nothing changes and the largest partitions are achieved:

$$\mathit{Nodes}_P = \bigcup_{\forall P_i \neq P_j \in \mathcal{W}'} \big\{\{P_i \cup P_j\} \setminus \{P_i\} \setminus \{P_j\} \,|\neg \big((P_i, P_j, \mathit{D\text{-}port}) \in \mathit{D\text{-}edges}'\big) \wedge$$
$$\big((P_i, P_j) \in \mathit{C\text{-}edges}'\big) \wedge \big(\big(\nexists P_x \neq P_j \in \mathcal{W}'|\, \big((P_i, P_x) \in \mathit{C\text{-}edges}'\big)\big) \wedge$$
$$\big(\nexists P_x \neq P_i \in \mathcal{W}'|\, \big((P_x, P_j) \in \mathit{C\text{-}edges}'\big)\big)\big)\big\}\,.$$

Therefore, $\mathit{Nodes}_P = \{\{N_1\}, \{N_2\}, \{N_3, \ldots, N_6\}, \{N_7, \ldots, N_{11}\}, \{N_{12}\}, \{N_{13}\}\}$.

Workflow partitioning allows the engine to aggregate the activities belonging to the same partition and execute them as one single job submission which drastically reduces the job management latencies of workflows with a large number of activities. In addition, the data dependencies between activities belonging to different partitions are archived, packed, and transferred as one file transfer job which drastically reduces the (GridFTP) connection latencies: $\mathit{D\text{-}edges}_P = \bigcup_{\forall P_1, P_2 \in \mathit{Nodes}_P} \{(P_1, P_2, \mathit{D\text{-}port}_{archive})\}$, where: $\mathit{D\text{-}port}_{archive} = \bigcup_{\substack{\forall\,(N_1, N_2, \mathit{D\text{-}port}) \in \mathit{D\text{-}edges} \\ \wedge N_1 \in P_1 \wedge N_2 \in P_2}} \{\mathit{D\text{-}port}\}$ is a compressed archive of all data dependencies between partitions P_1 and P_2 (typically instantiated during execution by files).

5.1.1. *Virtual single execution environment*

As a specialisation of workflow partitioning, we propose another simple technique called *Virtual Single Execution Environment* (VSEE) to reduce the management overheads of workflows characterised by a large (hundreds to thousands) number of activities with complex data dependencies which are relatively small in size. VSEE replaces the data dependencies between activities with the full data environment, recursively defined for a partition P as follows: $V_P = \bigcup_{\forall (P', P, \mathit{D\text{-}port}) \in \mathit{D\text{-}edges}_P} V_{P'} \bigcup_{\forall (P, P'', \mathit{D\text{-}port}) \in \mathit{D\text{-}edges}_P} \{\mathit{D\text{-}port}\}$. Clearly, the following property holds: $\exists\,(P', P, \mathit{D\text{-}port}) \in \mathit{D\text{-}edges}_P \iff V_{P'} \subset V_P$.

Upon executing a workflow partition on a Grid site, each slave engine automatically creates and removes one working directory that represents its execution environment. The VSEE mechanism transforms complex data dependencies between activities into one environment dependency between partitions that is packaged and transferred at runtime as one single data transfer activity. In addition to noticeably reducing the latencies and the number of data transfers for compute intensive Grid applications with large amounts of small sized data dependencies, VSEE presents two additional advantages: (1) it reduces the overhead of activity migration upon rescheduling (see Sections 5.2 and 5.3); (2) it shields the user from the complexity of the workflow definition and the error prone task of specifying tens or hundreds of input and output data ports between activities.

Figure 6 illustrates a sample workflow (see also Section 5.3) scheduled on three Grid sites $\{M_1, M_2, M_3\}$. First of all, the workflow is split into seven partitions: $\mathit{Nodes}_P = \bigcup_{i=1}^{7} P_i$, based on the algorithm presented in Section 5.1 (see Fig. 6a). Then, the data flow between partitions is optimised according to the VSEE-based relationships depicted in Table 3. For example, transferring data between partitions only according to the data flow dependencies requires P_6 to receive the data from: $V_{in} \cup V_1 \cup V_2 \cup V_3 \cup V_4 = V_4$, since $V_{in} \subset V_1 \subset V_2 \subset V_3 \subset V_4$. For certain compute intensive applications characterised by large numbers of small data dependencies like WIEN2k, the VSEE mechanism can drastically decrease the number of file transfers (up to orders of magnitude) as we will experimentally illustrate in Section 5.3.

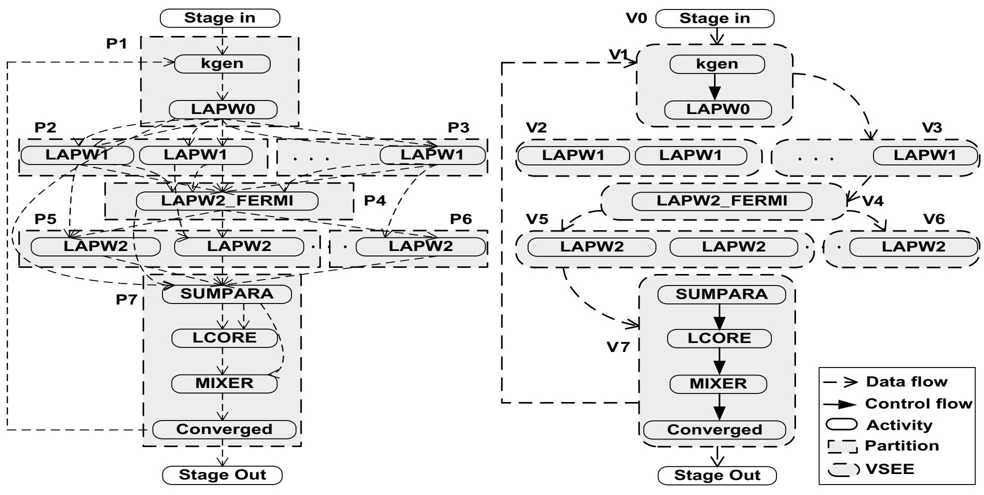
Table 3
The VSEE results for the WIEN2k workflow

(a) VSEE relationships.

$\mathcal{R}_V$	V_1	V_2	V_3	V_4	V_5	V_6	V_7	V_{out}
V_{in}	$\subset$	$\subset$	$\subset$	$\subset$	$\subset$	$\subset$		
V_1	–	$\subseteq$	$\subset$	$\subseteq$	$\subset$	$\subset$	$\subset$	
V_2		–		$\subseteq$	$\subset$	$\subset$		
V_3			–	$\subset$	$\subseteq$	$\subset$		
V_4				–	$\subset$	$\subset$	$\subset$	
V_5					–		$\subset$	
V_6						–	$\subseteq$	
V_7	$\subset$						–	$\supset$

(b) Minimum VSEE transfer set.

Transfer	P_1	P_2	P_3	P_4	P_5	P_6	P_7	*Output*
V_{in}	✓							
V_1			✓					
V_2								
V_3				✓				
V_4					✓	✓		
V_5							✓	
V_6								
V_7	✓							✓



(a) Original data flow. (b) Optimised VSEE data flow.

Fig. 6. A VSEE example.

5.2. *Workflow steering*

There may be many external factors that affect the execution of large workflows in dynamic Grid environments which no longer follow the original plan computed by the scheduler. Such unpredictable factors may include unpredictable queuing times, external load on processors (e.g. on Grid sites that also serve as student workstation rooms in our real Grid environment), unpredictable availability of processors on workstation networks (e.g. if a student shuts down a machine or reboots it in Windows operating system mode), jobs belonging to other users on parallel machines, congested networks, or simply inaccurate prediction information. Moreover, we often encountered in our real Grid environment sites that offer a reduced capacity for certain resources, for example a small number of input and output nodes that generate a denial of service attack if the number of concurrent file transfers (often used to increase bandwidth utilisation) exceeds a certain limit. The *steering* module of the enactment engine aims to minimise the losses due to such unpredictable situations that violate the optimised static mapping computed by the scheduler through appropriate rescheduling techniques.

5.2.1. *Rescheduling events*

The steering module of the enactment engine continuously monitors the workflow execution and triggers appropriate *rescheduling events* whenever any of the following situations occur:

– *cardinality port value change* which implies modifications in the workflow shape, in particular in the size of parallel loops (see Section 2 for the formal definition and Section 5.3 for a real-world example);
– *inaccurate prediction* of various workflow characteristics based on new execution performance data available, in particular branch probabilities in conditional activities, number of iterations in sequential and parallel loops, or more accurate execution time estimations of computational activities;
– *resource change*, in particular in the availability of Grid sites (i.e. number of processors available) where workflow activities are scheduled, or when new powerful parallel computers become available;
– *performance contract violation* caused by activity executions that no longer follow the original optimised plan computed by the scheduler.

Definition 4. Let N be a submitted activity, W_N its underlying work assigned (e.g. floating point operations), T_N its estimated execution time, and: $start(N) = end(N) - T_N$ its start timestamp, where the end timestamp *end*(*N*) was defined in Section 4 (see Definition 2). We define the *performance contract* [26] of an activity N at time instance t, such that $start(N) \leqslant t < end(N)$, as: $PC\left(N, \mathcal{S}_N, t\right) = \frac{W_N}{W_N(t) \cdot T_N} \cdot (t - start(N))$, where $W_N\left(t\right)$ is the work completed by activity N in the interval $[start(N), t]$.

The steering module of the enactment engine triggers a rescheduling event for activity N at time instance t whenever: $PC\left(N, \mathcal{S}_N, t\right) > f_N$, where f_N is the predefined *performance contract elapse factor* of activity N. Currently, the value of the performance contract elapse factor f_N needs to be statically defined by the user for each activity (as activity properties in the workflow specification [11]) that represents a certain percentage from its predicted activity execution time T_N. After rescheduling, the workflow activities are restarted.

5.2.2. *Steering algorithm*

An activity $N \in Nodes$ of the running workflow can be at a certain time instance t in one of the following *states*: *queued*, *running*, *completed*, or *failed*, denoted as $state(N, t)$.

In this section we propose a simple *steering algorithm* depicted in Algorithm 3 that is based on the repeated invocation of the static scheduling algorithm, as informally outlined by the following execution steps: (1) the algorithm receives as input a scientific workflow compliant with the model presented in Section 2 (lines 1–2); (2) the workflow is converted into a DAG and scheduled onto the Grid using optimisation heuristics as presented in Section 4 (lines 3–4); (3) the workflow is submitted for execution based on the initial schedule (line 5); (4) the workflow is monitored until it completes its execution (lines 6–14); (5) whenever one of the events presented in the previous section occur, a rescheduling event is triggered (line 7); (6) all activities that violate their performance contract are cancelled and reported as failed (lines 8–11); (7) the workflow is converted once again based on the new runtime information and rescheduled (lines 12–13).

To efficiently handle workflow rescheduling at runtime, we extended the workflow conversion algorithm introduced in Section 4.1 (see [19]) with a new time axis that only considers the relevant (i.e. still to be executed) part of the workflow as part of the optimisation process (lines 17–30). More specifically, the following activities are eliminated and not considered for rescheduling (lines 26–27): (1) all properly running activities that fulfill their performance contract; (2) all completed activities that do not have sequential loops as parents and, therefore, will not be re-executed.

5.3. *WIEN2k*

WIEN2k is a program package for performing electronic structure calculations of solids using density functional theory based on the full potential (linearised) augmented plane wave ((L)APW) and local orbital (lo) method. We first ported the application onto the Grid by splitting the monolithic code into several course grain activities coordinated in a workflow as illustrated in Fig. 6. The LAPW1 and LAPW2 activities can be solved in parallel by a fixed number

Algorithm 3 The workflow steering algorithm.

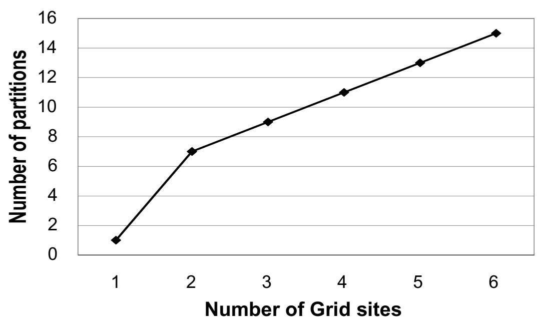
```
1:  function STEERING($\mathcal{W}$, GRID)
2:      $\mathcal{W} = (Nodes, C\text{-}edges, D\text{-}edges, IN\text{-}ports, OUT\text{-}ports)$        ▷ Precondition
3:      $\mathcal{W}' \leftarrow$ WF-CONVERTER($\mathcal{W}, \mathcal{W}, 0$)        ▷ Workflow conversion
4:      $\mathcal{S}_{\mathcal{W}'} \leftarrow$ SCHEDULE($\mathcal{W}'$)        ▷ Workflow scheduling
5:      EXECUTE($\mathcal{S}_{\mathcal{W}'}$)        ▷ Workflow execution
6:      repeat
7:          $t \leftarrow$ SLEEP($event$)        ▷ Until scheduling event
8:          for all $N \in Nodes \wedge state(N, t) = running \wedge PC\,(N, \mathcal{S}_N, t) > f_N$ do
9:              CANCEL($N$)        ▷ Performance contract violation
10:             $state(N) \leftarrow failed$
11:         end for
12:         $\mathcal{W}' \leftarrow$ WF-CONVERTER($\mathcal{W}, \mathcal{W}, t$)        ▷ Runtime workflow conversion
13:         $\mathcal{S}_{\mathcal{W}'} \leftarrow$ SCHEDULE($\mathcal{W}'$)        ▷ Workflow rescheduling
14:     until $state(N, t) = completed, \forall N \in Nodes \wedge succ(N) = \emptyset$
15: end function
16:
17: function WF-CONVERTER($\mathcal{W}_{root}, N, t$)
18:     if $N$ is a $N_{if}$ then
19:         $\mathcal{W}_{root} \leftarrow$ BRANCH-EXPANSION($\mathcal{W}_{root}, N$)
20:     else if $N$ is a $N_{loop}$ then
21:         $\mathcal{W}_{root} \leftarrow$ SEQ-LOOP-UNROLLING($\mathcal{W}_{root}, N$)
22:     else if $N$ is a $N_{par}$ then
23:         $\mathcal{W}_{root} \leftarrow$ PAR-LOOP-UNROLLING($\mathcal{W}_{root}, N$)
24:     else if $N$ is a $\mathcal{W}$ then
25:         $\mathcal{W}_{root} \leftarrow$ WF-INLINING($\mathcal{W}_{root}, N$)
26:     else if $(state(N, t) = running \wedge PC\,(N, \mathcal{S}_N, t) \leq f_N) \vee$
            $(state(N, t) = completed \wedge (\nexists n \in \mathbb{N} \wedge Parent^n(N)$ is a $N_{loop}))$ then
27:         ACTIVITY-ELIMINATION($N$)        ▷ Completed or properly running
28:     end if
29:     return $\mathcal{W}_{root}$
30: end function
```

of so called *k-points*. A final activity called *Converged* applied to several output files tests whether the problem convergence criterion is fulfilled. The number of sequential loop iterations is statically unknown.

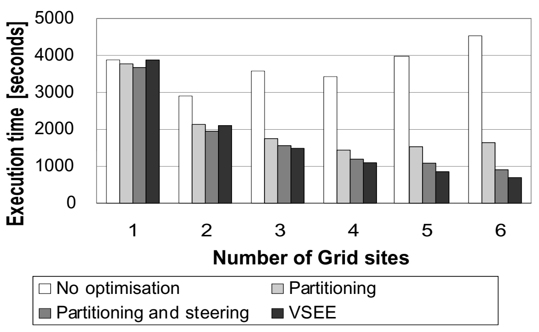
We executed the WIEN2k workflow in a subset of the Austrian Grid infrastructure [22] consisting of a number of parallel computers and workstation networks accessible through the Globus toolkit and local job managers, as depicted in Table 4. We chose a problem size that produces at runtime 250 parallel k-points which means a total of over 500 workflow activities. We first executed the workflow application on the fastest site available (i.e. altix1.jku in Linz) that gives an indication of what can be achieved for this application by using only local compute resources. Then we incrementally added the next fastest sites for this application (i.e. top-down order in Table 4) and observed the benefits or losses obtained by executing the same problem size in a larger Grid environment. We compared the performance delivered by three of our workflow enactment techniques: partitioning, partitioning and steering, and VSEE.

Figure 7a presents the number of WIEN2k partitions computed by the partitioning algorithm for each Grid site configuration. The number of partitions depends on the workflow structure and the execution plan computed by the scheduler and is proportional to the number of sites used in each execution. Figure 7b shows the execution times for running the same WIEN2k problem on different Grid size configurations ranging from one to six aggregated sites. Similarly, Fig. 7c displays the workflow *speedup* computed as the ratio between the fastest single site execution time T_{seq}^M (altix1.jku in Linz) and the current Grid execution time $T_{\mathcal{W}}$: $S = \frac{\min_{\forall M \in \mathtt{GRID}}\left\{T_{seq}^M\right\}}{T_{\mathcal{W}}}$. Without any optimisation, the performance and the speedup deteriorate with the increase in the number of Grid sites used for scheduling and running the workflow. With optimisation and steering, the WIEN2k execution time improves because of the simplified data flow and balanced execution of the LAPW1 and LAPW2 parallel loops.

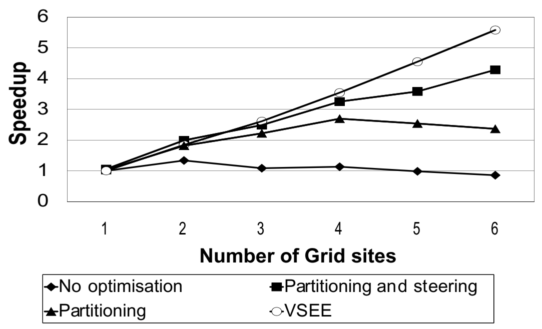
Figures 7d and 7e show that the number of file transfers, respectively remote job submissions, are considerably reduced when optimisation is applied, which explains the performance results obtained. Figure 7f shows that the



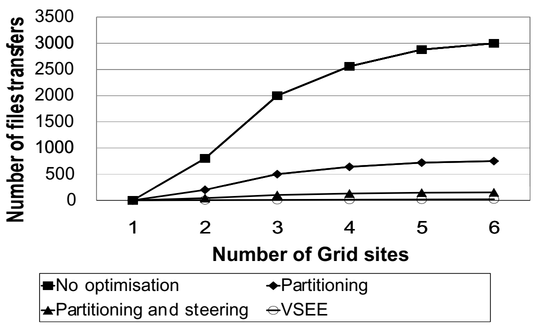
(a) Number of partitions.



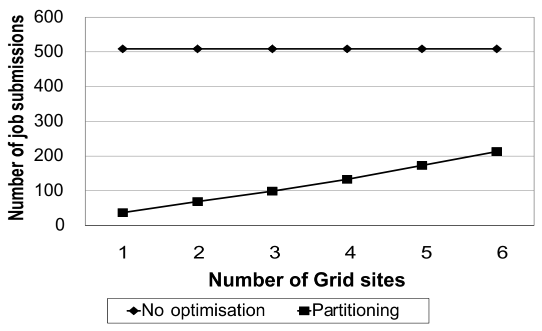
(b) Execution time.



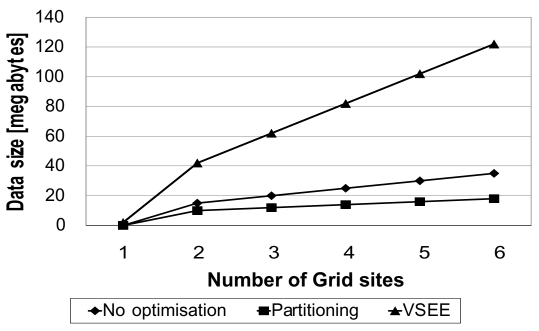
(c) Speedup.



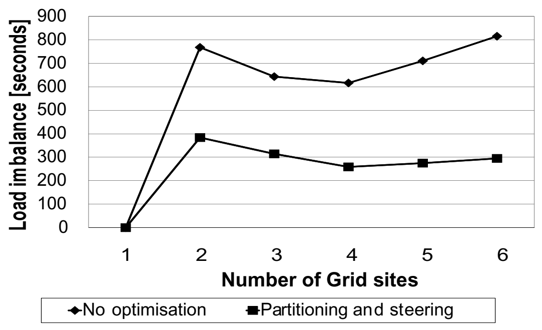
(d) Number of file transfers.



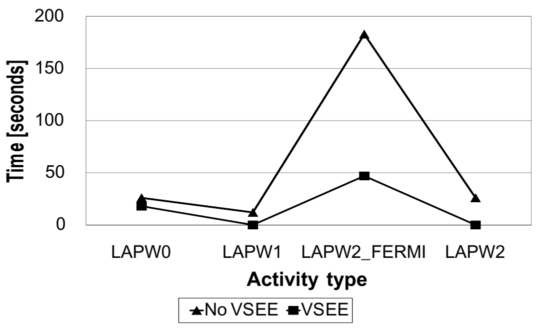
(e) Number of job submissions.



(f) Size of transferred files.



(g) Load imbalance.

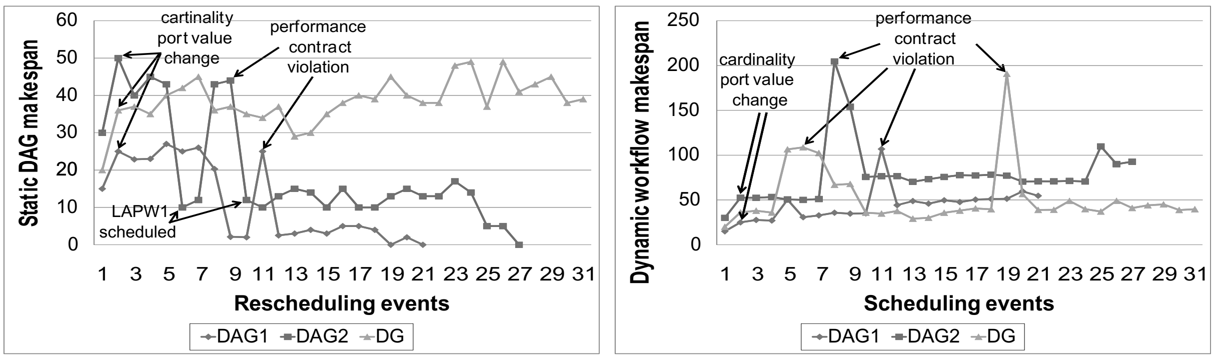


(h) Overhead of activity migration.

Fig. 7. The WIEN2k execution results.

Table 4
The Austrian Grid testbed for WIEN2k experiments

Site	*Architecture*	*Size*	*Processor*	*GHz*	*Job Mgr.*	*Location*
altix1.jku	ccNUMA, SGI Altix 3000	16	Itanium 2	1.6	Fork	Linz
altix1.uibk	ccNUMA, SGI Altix 350	16	Itanium 2	1.6	Fork	Innsbruck
schafberg	ccNUMA, SGI Altix 350	16	Itanium 2	1.6	Fork	Salzburg
agrid1	NOW, Fast Ethernet	16	Pentium 4	1.8	PBS	Innsbruck
arch19	NOW, Fast Ethernet	20	Pentium 4	1.8	PBS	Innsbruck
arch21	NOW, Fast Ethernet	20	Pentium 4	1.8	PBS	Innsbruck



(a) Static DAG makespan. (b) Dynamic workflow makespan.

Fig. 8. The WIEN2k steering traces.

size of transferred data under VSEE is obviously larger than in the other cases, however, VSEE offers the biggest execution improvement since it reduces the number of file transfers by three orders of magnitude, which drastically reduces the latencies (i.e. mutual authentication to the GridFTP service) while effectively utilising the bandwidth.

The steering improvement is caused by a large load imbalance in the workflow parallel loops caused by external load on the Altix shared memory machines and external jobs on the workstation networks. As a consequence, the execution deviated too much from the schedule which triggered a rescheduling event. Since the number of activities in the parallel loop exceeds the number of processors on the Grid testbed, the scheduler was able to reduce the load balance to half (see Fig. 7g) the loop by rescheduling the jobs not yet executed. We exhibit a slow down from five to six Grid sites using control and data flow optimisation because of the increased communication time across six distributed sites.

Figure 7h compares the data transfer overheads of the activity migration upon steering with and without the VSEE mechanism. One important aspect is that the data transfer overhead upon migrating LAPW1 and LAPW2 activities is zero when using the VSEE mechanism. The reason is that the sequential activities LAPW0 and LAPW2_FERMI replicate all their output files to the sites where the following LAPW1 and LAPW2 parallel loop activities are scheduled. Therefore, these activities will find their inputs already prepared on the sites where they are migrated which eliminates the data transfer overhead.

To better understand the steering algorithm, we generated three experimental WIEN2k workflows (i.e. two DAG and one DG-based containing one sequential loop) that correspond to different application input cases (i.e. the number of atoms and matrix sizes) with different parallelization sizes (i.e. number of k-points). To achieve a more fine-grained execution trace, we generated periodical rescheduling events in addition to those presented in Section 5.2.1.

Figure 8a traces the value of the makespan objective function optimised by the scheduling algorithm at consecutive scheduling events during the execution of each experimental workflow. A characteristic of the WIEN2k workflow is that the cardinality of the LAPW1 and LAPW2 parallel loops is determined by a cardinality port generated by LAPW0. Since the value of this port is statically unknown, the scheduler assumes one activity iteration which serialises the workflow activities onto the fastest Grid site. As soon as the cardinality port is instantiated, the workflow is rescheduled and the predicted makespan increases. As the remaining workflow activities are scheduled, execute, and complete, the makespan of the remaining DAG1 and DAG2 sub-workflows obviously decreases with the number of scheduling events. The abrupt decreases of the makespan happen after the submission of all the LAPW1 k-points, which are the most time consuming workflow activities that no longer need to be considered by the scheduler.

At this point, we artificially created external load by submitting several external jobs to the parallel machines. This caused abrupt increases in the makespan due to LAPW1 activities that violate their performance contract and need to be reconsidered by the scheduler for rescheduling, migration, and restart. In case of the DG-based workflow, the scheduler always receives the complete workflow as input but with a different control precedence relation between activities which explains why the makespan stays relatively constant.

Figure 8b traces the overall predicted workflow makespan (i.e. the time the entire workflow is expected to execute) at consecutive scheduling events during the workflow execution. The peaks are again due to the same performance contract violations of several LAPW1 activities which, after being rescheduled, bring the next predicted makespan close to the original predicted value. We achieve through rescheduling an estimate of about two fold improvement in the overall makespan. Since the workflow referred as DAG2 represents a larger problem size than DAG1, the benefit obtained through rescheduling and activity migration is higher. The final makespan of the DAG-based workflows is, however, about twice as large as originally predicted by the scheduler. For the DG-based workflow, we could not estimate the makespan of the entire workflow (i.e. beyond the execution of one sequential loop iteration) since the number of loop iterations is statically unknown. As a consequence, Fig. 8 represents the DG makespan of one workflow iteration only, which was successfully kept relatively constant through activity migration in two critical occasions.

6. Related work

The GrADS project pioneered the idea of runtime adaptation based on performance contracts through online monitoring and analysis performed by the Autopilot tool [24]. The work focused primarily on Grid support for numerical libraries, high performance parallel applications [5], and parameter studies, which is complementary to our research on scientific workflows.

Gridbus [6] provides an XML-based language oriented towards parametrisation and Quality of Service requirements. No branches and loops are supported. Gridbus provides a scheduler supporting deadline and budget constraints based on genetic algorithms. The work is based on the a simulated environment, while our work targets real-world applications. Performance prediction is not addressed.

ICENI [16] workflow specification contains low-level enactment engine-specific constructs such as `start` and `stop` activities. Scheduling is done using random, best of n-random, simulated annealing, and game theory algorithms. Prediction work focuses on improving Grid predictability through advance reservation.

Similar to our approach, Karajan [25] can specify hierarchical workflows using an XML language that includes sequential and parallel loops. Workflows can be modified at runtime through interaction with a workflow repository or schedulers for dynamic association of resources to tasks. Karajan applies opportunistic round-robin or lookahead scheduling policies based on the maximum number of jobs allowed for a Grid site. No workflow optimisation or steering techniques are addressed.

Kennedy et al. [14] compared several task-based and workflow-based approaches to resource allocation for workflow applications based on simulation rather than real executions as we do. They also study the impact of uncertainty on the overall workflow schedule but do not propose runtime optimisation or steering techniques.

Kepler [15] extends the Ptolemy system with new features and components for scientific workflow design such as branches, sequential loops, and data dependencies. Parallel loops are not supported. Automatic scheduling based on performance prediction is not supported.

Pegasus [8] uses Condor DAGMan [1] as its enactment engine enhanced with data derivation techniques that simplifies the workflow at runtime based on data availability. Pegasus provides a layered workflow restructuring method which takes place before the scheduling phase. Our approach partitions and optimises the workflow after scheduling and uses this mapping information to further improve the partitioning.

Taverna [17] was originally designed to provide Grid support for bioinformatics applications with recent focus on semantic annotations, provenance, and workflow reuse. Taverna is based on Web services and uses the Simple Conceptual Unified Flow Language for workflow choreography, which is limited to DAGs. Taverna enables users to construct, share, and enact workflows using a customized fault-tolerant enactment engine based on opportunistic just-in-time scheduling.

Triana [21] uses the Grid Application Toolkit interface to the Grid through JXTA and Web services. It provides support for conditional activities and sequential loops, but lacks compact mechanisms for expressing large parallel loops. Scheduling is done just-in-time with no optimizations or performance estimates. Recent preliminary work targets a generic architecture for monitoring and steering legacy applications.

In [12], a pattern-based software engineering tool for Grid environments is described. The authors propose a broad set of structural patterns classified in topological (e.g. pipeline, ring, star) and non-topological (e.g. adapter, facade, proxy) categories. Additionally, they propose a set of structural operators to modify and manipulate patterns including increase, decrease, extend, reduce, embed, or extract. The approach is validated through an implementation of the structural patterns as an extension to the Triana tool. The approach is generic and at a high level of abstraction that could be used to express ASKALON workflows too.

UNICORE [9] provides a graphical composition of directed graph-based workflow with no support for parallel loops. The scheduling is done manually by allowing the user to allocate resources and specify data transfer tasks though the graphical user interface.

JISGA [13] is a Jini-based service-oriented architecture for Grid computing that, in addition to the basic functionalities of a general Jini system, supports workflow applications by implementing the XML-based Service Workflow Language (SWFL). SWFL extends IBM's Web Services Flow Language (WSFL) through improved conditional and loop control constructs, extended data flow patterns, more data mappings including arrays, and support for assignment statements. Optimisation is supported through discovery of a set of semantically equivalent services and selection based on real-time or historical data. Global workflow optimisations are not supported.

7. Conclusions

In this paper, we have presented techniques used in the ASKALON project for supporting effective modelling and high-performance execution of scientific workflows in Grid environments. Our approach is new or different from existing approaches in several aspects. As part of our abstract and generic hierarchical model, we introduced the concept of cardinality input port of parallel loops that changes the workflow structure dynamically at runtime. A performance prediction service supports the scheduler with accurate execution time information based on a well-defined training phase with a reduced number of experiments. A modular scheduling service employs advanced heuristics such as list scheduling, matchmaking, and genetic algorithms to find good mappings of workflow activities onto the Grid resources. An enactment engine service ensures scalable execution of large scientific workflows through techniques such as partitioning, control and data flow optimisation, and runtime steering adaptation. In contrast to related work often based on simulation, we validated our techniques by modelling, scheduling, and analysing the scalability of two real-world applications from material science and hydrological fields on our national Grid infrastructure.

References

[1] DAGMan: Directed acyclic graph manager. http://www.cs.wisc.edu/condor/dagman/. Condor project, University of Wisconsin-Madison.

[2] Daniel Barbara, Sharad Mehrotra, and Marek Rusinkiewicz. INCAS: a computation model for dynamic workflows in autonomous distributed environments. Technical report, Matsushita Information Technology Laboratory, 2 Research Way, 3rd Floor, Princeton, N.J. 08540 USA, 1994.

[3] L. Baresi and R. Heckel, Tutorial introduction to graph transformation: A software engineering perspective, in: *1st International Conference on Graph Transformation*, Springer Verlag, 2002, 402–429.

[4] F. Berman, H. Casanova, A. Chien, K. Cooper, H. Dail, A. Dasgupta, W. Deng, J. Dongarra, L. Johnsson, K. Kennedy, C. Koelbel, B. Liu, X. Liu, A. Mandal, G. Marin, M. Mazina, J. M. Mellor-Crummey, C. Mendes, A. Olugbile, M. Patel, D. Reed, Z. Shi, O. Sievert, H. Xia and A. YarKhan, New Grid scheduling and rescheduling methods in the GrADS project, in: *International Journal of Parallel Programming* **33** (2005), 209–229.

[5] L.S. Blackford, J. Choi, A. Cleary, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker and R.C. Whaley, ScaLAPACK: a linear algebra library for message-passing computers. in: *Conference on Parallel Processing*, Society for Industrial and Applied Mathematics, 1997.

[6] Rajkumar Buyya and Srikumar 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*, IEEE Computer Society Press, 2004, 19–36.

[7] Ewa Deelman, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, Kent Blackburn, Albert Lazzarini, Adam Arbree, Richard Cavanaugh, and Scott Koranda, Mapping abstract complex workflows onto Grid environments, *Journal of Grid Computing* **1**(1) (2003), 25–39.

[8] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vaki, G. Bruce Berriman, John Good, Anastasia Laity, Joseph C. Jacob and Daniel S. Katz, Pegasus: A framework for mapping complex scientific workflows onto distributed systems, *Scientific Programming* **13**(3) (2005), 219–237.

[9] Dietmar W. Erwin, UNICORE – a Grid computing environment. *Concurrency and Computation: Practice and Experience* **14**(13–15) (2002), 1395–1410.

[10] Thomas Fahringer, Radu Prodan, Rubing Duan, Jürgen Hofer, Farrukh Nadeem, Francesco Nerieri, Jun Qin Stefan Podlipnig, Mumtaz Siddiqui, Hong-Linh Truong, Alex Villazon and Marek Wieczorek, Askalon: A development and Grid computing environment for scientific workflows, in: *Scientific Workflows for Grids*, I.J. Taylor, E. Deelman, D.B. Gannon and M. Shields, eds, Workflows for e-Science, chapter Frameworks and Tools: Workflow Generation, Refinement and Execution. Springer Verlag, 2007. ISBN: 978-1-84628-519-6.

[11] Thomas Fahringer, Jun Qin, and Stefan Hainzer, Specification of Grid workflow applications with AGWL: An abstract Grid workflow language, in: *International Symposium on Cluster Computing and the Grid*. IEEE Computer Society Press, 2005.

[12] M. Cecilia Gomes, Jose C. Cunha and Omer F. Rana, Pattern operators for grid environments, *Scientific Programming* **11** (2003), 237–261.

[13] Yan Huang, JISGA: A Jini-based service-oriented Grid architecture, *International Journal of High Performance Computing Applications* **17**(3) (2003), 317–327.

[14] Ken Kennedy, Jim Blythe, Sonal Jain, Ewa Deelman, Yolanda Gil, Karan Vahi and Anirban Mandal, Task scheduling strategies for workflow-based applications in Grids, in: *International Symposium on Cluster Computing and the Grid*, IEEE Computer Society Press, 2005.

[15] Bertram Ludascher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A. Lee, Jing Tao and Yang Zhao, Scientific workflow management and the Kepler system, *Concurrency and Computation: Practice and Experience* **18**(10) (2006), 1039–1065.

[16] A. Mayer, S. McGough, N. Furmento, W. Lee, S. Newhouse and J. Darlington, (ICENI) dataflow and workflow: Composition and scheduling in space and time, in: *UK e-Science All Hands Meeting*, Nottingham, UK, 2003, pp. 627–634.

[17] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver adn K. Glover, M.R. Pocock, A. Wipat and P. Li, Taverna: a tool for the composition and enactment of bioinformatics workflows, *Bioinformatics* **20**(17) (2004), 3045–3054.

[18] Radu Prodan and Thomas Fahringer, ZENTURIO: A Grid service-based tool for optimising parallel and Grid applications, *Journal of Grid Computing* **2**(1) (2004), 15–29.

[19] Radu Prodan and Thomas Fahringer, *Grid Computing. Experiment Management, Tool Integration and Scientific Worflows*, volume 4340 of *Lecture Notes in Computer Science*. Springer Verlag, 2007.

[20] K. Schwarz, P. Blaha and G.K.H. Madsen, Electronic structure calculations of solids using the WIEN2k package for material sciences, *Computer Physics Communications* **147**(71) (2002).

[21] Ian Taylor, Matthew Shields, Ian Wang and Rana Rana, Triana applications within Grid computing and peer to peer environments, *Journal of Grid Computing* **1**(2) (2003), 199–217.

[22] The Austrian Grid Consortium. http://www.austriangrid.at.

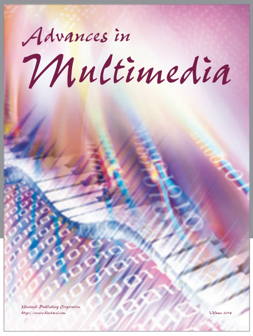
[23] Dieter Theiner and Peter Rutschmann, An inverse modelling approach for the estimation of hydrological model parameters, In *Journal of Hydroinformatics*. IWA Publishing, 2005.

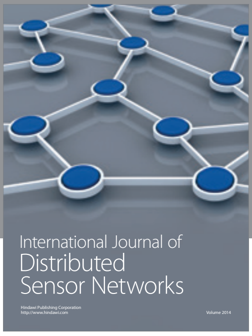
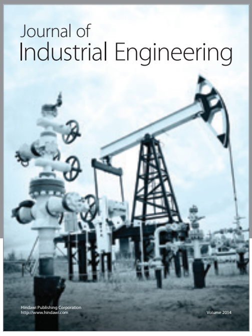
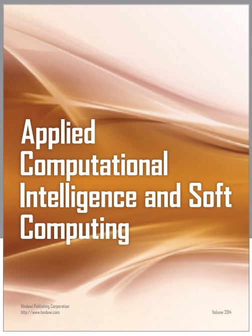
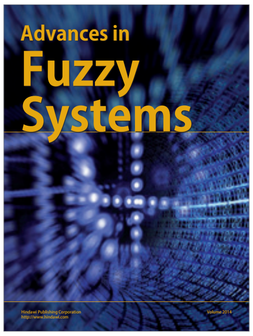
[24] Jeffrey S. Vetter and Daniel A. Reed, Real-time performance monitoring, adaptive control, and interactive steering of computational Grids, *International Journal of High Performance Computing Applications* **14**(4) (2000), 357–366.

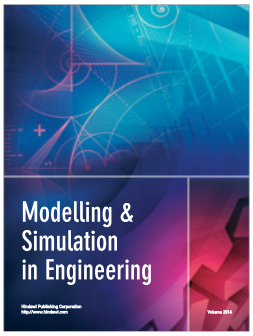
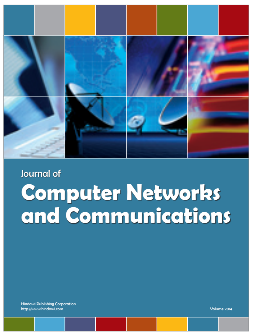
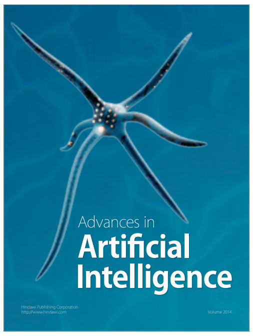
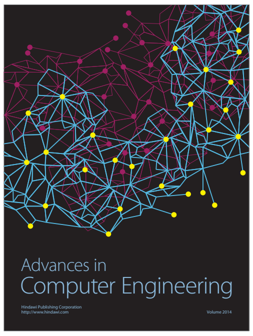
[25] Gregor von Laszewski and Mihael Hategan, Workflow concepts of the java coG kit, *Journal of Grid Computing* **3**(3-4) (2005), 239–258.

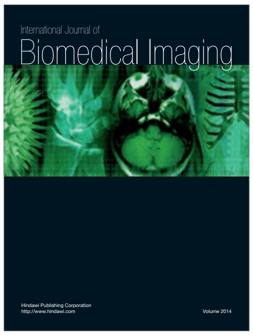
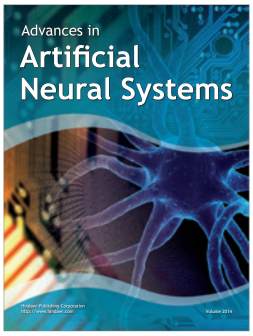
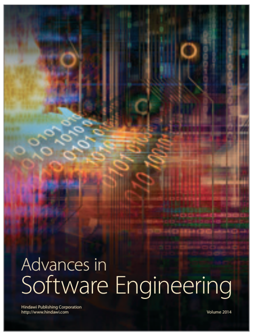
[26] Fredrik Vraalsen, Ruth A. Aydt, Celso L. Mendes and Daniel A. Reed, Performance contracts: Predicting and monitoring Grid application behavior, in: *2nd International Grid Computing Workshop*, volume 2242 of *Lecture Notes in Computer Science*, Springer Verlag, 2001, pp. 154–166.

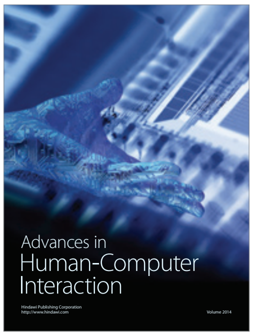
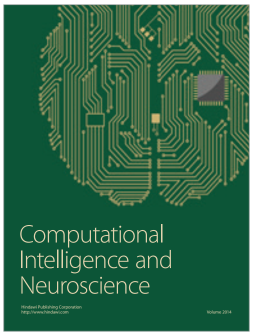
[27] Henan Zhao and Rizos Sakellariou, An experimental investigation into the rank function of the heterogeneous earliest finish time scheduling algorithm. in: *Euro-Par Conference*, 2003, pp. 189–194.


Advances in
Multimedia

The Scientific World Journal

International Journal of
Distributed Sensor Networks

Journal of
Industrial Engineering

Applied Computational Intelligence and Soft Computing

Advances in
Fuzzy Systems


Modelling & Simulation in Engineering

Journal of
Computer Networks and Communications

Advances in
Artificial Intelligence

Advances in
Computer Engineering

International Journal of
Computer Games Technology

International Journal of
Biomedical Imaging

Advances in
Artificial Neural Systems

Advances in
Software Engineering

Journal of
Robotics

Advances in
Human-Computer Interaction

Computational Intelligence and Neuroscience

International Journal of
Reconfigurable Computing

Journal of
Electrical and Computer Engineering