Dynamic Scheduling of Scientific Workflow Applications on the Grid: A Case Study*

Radu Prodan and Thomas Fahringer
Institute for Computer Science, University of Innsbruck
Technikerstraße 13, A-6020 Innsbruck, Austria
Tel: ++43 512 507 6441
{radu,tf}@dps.uibk.ac.at

ABSTRACT

The existing Grid workflow scheduling projects do not handle recursive loops which are characteristic to many scientific problems. We propose a hybrid approach for scheduling Directed Graph (DG)-based workflows in a Grid environment with dynamically changing computational and network resources. Our dynamic scheduling algorithm is based on the iterative invocation of classical static Directed Acyclic Graphs (DAGs) scheduling heuristics generated using well-defined cycle elimination and task migration techniques. We approach the static scheduling problem as an application of a modular optimisation tool using genetic algorithms. We report successful implementation and experimental results on a pilot real-world material science workflow application.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Heuristic methods, Scheduling*

General Terms

Algorithms, Management, Measurement, Performance, Design, Experimentation.

Keywords

Grid computing, Scientific workflows, Optimisation, Scheduling, Genetic Algorithms, Performance steering.

1. MOTIVATION

Computational Grids have become an important asset that enable the application developers to aggregate resources scattered around the globe for large-scale scientific research. Recently, the workflow model has gained increased interest as the potential state-of-the-art paradigm for programming loosely-coupled Grid applications.

One important activity the users commonly perform before to executing a Grid application is to find appropriate *schedules*. The present literature on DAG scheduling comprises a variety of heuristics which aim to produce good solutions in a reasonable time [7]. These heuristics, however, have restricted applicability and lack of scalability since in general can not be parallelised. The complexity of most heuristics is of the order of $\mathcal{O}(n^2 \cdot m)$, where n is the number of tasks and m is the number of resources, which gets rather critical for large workflows solving big problem sizes in world-wide Grid infrastructures.

Beyond simulation, there has been little technology transfer that investigates existing heuristics for scheduling of real-world workflow applications on the Grid. The Pegasus system [2] promotes the use of Artificial Intelligence planning techniques and employs Greedy randomised adaptive search techniques for scheduling data Grid workflows. The GrADS project [3] optimises DAG-based workflows using Min-Min, Min-Max, and Suffrage heuristics. Both projects, however, exclusively concentrate on DAG-based workflows and report no results on the quality of the solutions delivered. Moreover, while DAG-based workflows are sufficient for modelling business processes, lots of scientific applications simulate recursive problems with dynamic convergence criteria, which are commonly implemented in an imperative programming language through backward *loops*.

In this paper we propose a hybrid approach to the DG-based workflow scheduling problem in a Grid environment with dynamically changing computational and network resources. Our dynamic scheduling algorithm is based on the iterative invocation of classical DAG scheduling heuristics at certain scheduling events, based on well-defined cycle elimination and task migration techniques. We approach the static scheduling problem as an application of an automatic experiment management tool using genetic algorithms.

2. WORKFLOW MODEL

Defining a new language to model scientific Grid applications is beyond the scope of this work. Rather, we adopt a low-level workflow representation which we believe to constitute the minimal but sufficient foundation to which any higher-level workflow specification needs to be compiled.

*This research was supported by the Austrian Science Fund as part of the Aurora project under contract SFBF1104.

SAC'05 March 13-17, 2005, Santa Fe, New Mexico, USA

Definition 1 A *workflow application* is a DG denoted as $\mathcal{A} = (Nodes, Edges)$, where *Nodes* is the set of workflow *tasks* and *Edges* the set of *directed task dependencies*. Workflow tasks are classified in two categories: $Nodes = Nodes^{JS} \cup Nodes^{FT}$: (1) *job submission*, denoted as $JS(M) \in Nodes^{JS}$, where M is the *abstract machine* where the *JS* task runs; (2) *file transfer*, denoted as $FT(M_1, M_2) \in Nodes^{FT}$, where M_1 and M_2 are the source, respectively the destination abstract machines. Let $succ(N)$ denote the set of *successors* of one task $N \in Nodes$: $N_s \in succ(N) \iff \exists\ (N, N_s) \in Edges$. Similarly, let $pred(N)$ denote the set of *predecessors* of one task $N \in Nodes$: $N_p \in pred(N) \iff \exists\ (N_p, N) \in Edges$. Additionally, we refer to the predecessors and successors of rank p of a task N as: $pred^p(N) = pred(\dots pred(N))$ and $succ^p(N) = succ(\dots succ(N))$ (p calls).

If the same abstract machine appears in the definition of two distinct tasks, it defines a *static schedule dependency*. A typical use of static schedule dependencies is for data staging between two workflow tasks. For example, the task $FT_2(M_1, M_2)$ in Figure 1 defines a file transfer from the abstract machine M_1, where the producer $JS_1(M_1)$ runs, to the abstract machine M_2, where the consumer $JS_2(M_2)$ runs. This defines two static schedule dependencies: (JS_1, FT_2) through M_1 and (FT_2, JS_2) through M_2.

3. WIEN2K

The pilot application for our work is the WIEN2k [1] 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 orbitals (lo) method. The various programs that compose the WIEN2k package are organised in a workflow, as illustrated in Figure 1. The LAPW1 and LAPW2 tasks can be solved in parallel by a fixed number of so called *k-points*. A final task applied on several output files tests whether the problem convergence criterion is fulfilled. The number of recursive loops is statically unknown.

We have implemented the workflow representation as specified in Definition 1 within a Java package on top of the Globus toolkit. The *JS* tasks are implemented as clients of the Globus Resource Allocation Manager and the *FT* tasks are based on the GridFTP communication protocol. The Grid Security Infrastructure is employed for secure communication across the workflow nodes and the Globus services. An excerpt of the Java WIEN2k workflow implementation is depicted in Example 1.

Example 1 (Parameterised Java workflow (WIEN2k))

```
//ZEN$ SUBSTITUTE lapw0_host = { machine{1:300} }
//ZEN$ SUBSTITUTE lapw1_host1 = { machine{1:300} }
//ZEN$ SUBSTITUTE lapw1_host2 = { machine{1:300} }
. . .
Task lapw0 = createJS("lapw0_host", "lapw0");
Task lapw1_1 = createJS("lapw1_host1", "lapw1 2");
Task lapw1_2 = createJS("lapw1_host2", "lapw1 1");
Task k1 = createFT("k1","lapw0_host","lapw1_host1");
Task k2 = createFT("k2","lapw0_host","lapw1_host2");
. . .
TaskGraph taskGraph = new TaskGraphImpl();
taskGraph.add(lapw0);
taskGraph.add(lapw1_1);
taskGraph.add(lapw1_2);
taskGraph.add(k1);
taskGraph.add(k2);
```

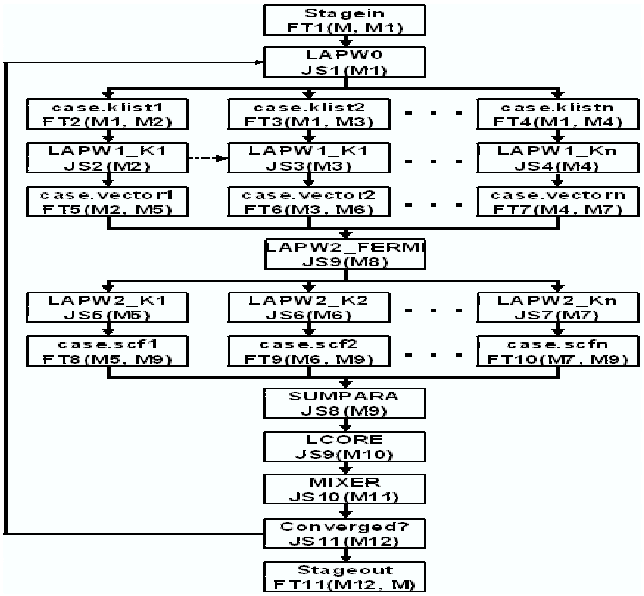


Figure 1: The WIEN2k workflow.

```
. . .
Dependency dependency = new DependencyImpl();
dependency.add(lapw0.getId(), k1.getId());
dependency.add(lapw0.getId(), k2.getId());
dependency.add(k1.getId(), lapw1_1.getId());
dependency.add(k2.getId(), lapw1_2.getId());
. . .
taskGraph.setDependency(dependency);
```

4. ZENTURIO OPTIMISATION TOOL

ZENTURIO [8] is an experiment management tool designed to perform automatic cross-experiment performance studies for parallel applications. A simple *directive-based language* (Fortran HPF and OpenMP-like) called ZEN is used to annotate any application files and define value ranges for arbitrary application parameters, including program variables, file names, compiler options, target machines, machine sizes, array and loop distributions, interconnection networks, or software libraries. For example, the *ZEN directives* illustrated in Example 1 specify the possible instantiation values (i.e., the set of concrete Grid machines: { *machine1, . . . , machine300* }) for the workflow abstract machines, which represent the generic application parameters (i.e., `lapw0_host`, `lapw1_host1`, `lapw1_host2`).

ZENTURIO has been designed as a distributed service-oriented architecture, depicted in Figure 2. The user inputs an application annotated with ZEN directives that defines the generic parameter space subject to automatic multi-experimental performance study. A point in the space that instantiates each application parameter is called *experiment*. The generation of experiments is performed by the Experiment Generator service either exhaustively, or according to a *heuristic search engine* that attempts to maximise an input objective function by visiting a fraction of the search space points defined through ZEN directives.

Definition 2 Let $\mathcal{A}(M_1, \dots, M_n)$ denote a parameterised application, where M_i are the application parameters defined through ZEN directives, and $\mathcal{V}^{M_i}$ the full set of in-

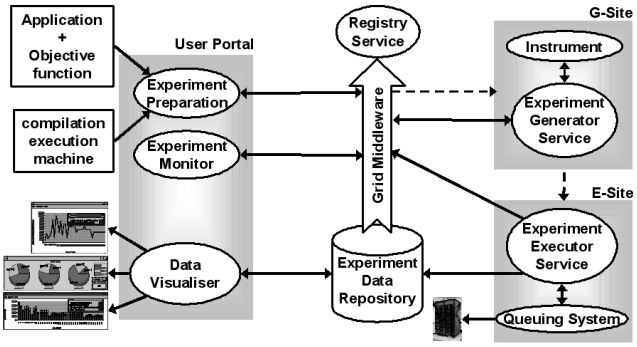


Figure 2: The ZENTURIO Tool Architecture.

stantiation values for the parameter M_i, $\forall\ i \in [1..n]$. Let

$$\mathcal{V}^{\mathcal{A}} = \left\{ \mathcal{I}(m_1, \ldots, m_n) \mid \forall\ (m_1, \ldots, m_n) \in \mathcal{V}^{M_1} \times \ldots \times \mathcal{V}^{M_n} \right\}$$

denote the full set of *experiments* that can be generated from the parameterised application. The *objective function* to be maximised by the search engine must implement a problem independent interface defined by the function: $\mathcal{F} : \mathcal{V}^{\mathcal{A}} \rightarrow \mathbb{R}_+$, where $\mathbb{R}_+$ the set of positive real numbers.

The isolation of the objective function under a problem independent interface enables the plug-and-play instantiation of new optimisation problems. In our approach, the objective function is represented either by a metric for performance tuning of parallel applications [8], or by a prediction function for the scheduling problem addressed in this paper.

For this paper we consider a problem independent search engine instantiation through a genetic algorithm, while further general purpose heuristics (including gradient descent and simplex methods, or other evolutionary algorithms like simulated annealing) are targeted for future work.

After each experiment has been generated, an Experiment Executor service takes over the execution on the target machine and stores the performance and the output data into a relational data repository upon the experiment completion. Through a graphical portal, the user can map application parameters to the axes of a variety of visualisation diagrams (i.e., linechart, barchart, piechart, surface) and automatically formulate SQL queries that display the evolution of the computed performance metrics or output results across multiple experiments (see Figure 6).

4.1 Static DAG Scheduling

Scheduling a DAG of n tasks onto m computational Grid resources is a known NP-complete optimisation problem of $\mathcal{O}(m^n)$ complexity.The present literature on DAG scheduling comprises a variety of heuristics which aim to produce good solutions in a reasonable amount of time [7]. These heuristics, however, have restricted applicability and lack of scalability since they are not parallelisable. The complexity of most heuristics is of the order of $\mathcal{O}(n^2 \cdot m)$, where m is the number of resources and n the number of tasks to be scheduled. This complexity gets rather critical for large workflows solving big problem sizes on a world-wide Grid infrastructure aggregating a potentially unbounded amount of computing resources.

Genetic algorithms are well known as a powerful problem-independent heuristic for randomised search of high-quality solutions within large search spaces. A peculiarity of genetic algorithms, which makes them suitable for being considered for scheduling in a Grid environment, is that they incrementally deliver improved (potentially satisfactory even in the early stages) solutions across generations, which could shield the users from the complexity of other heuristics.

The problem independent realisation of the genetic search engine in ZENTURIO encodes a generic application parameter M as a *gene*. The complete string of genes builds a *chromosome*. An *experiment* $\mathcal{I}(m_1, \ldots, m_n)$ is obtained by instantiating each gene M_i with a concrete *allele* $m_i \in \mathcal{V}^{M_i}$, $\forall\ i \in [1..n]$. We employ a classical *generational genetic search algorithm* that uses the reminder stochastic sampling with replacement *selection* mechanism and the single point *crossover* and *mutation* operators, as described in the existing literature [5].

For the DAG scheduling problem, a gene or a generic parameter of a workflow $\mathcal{A}(M_1, \ldots, M_n)$ represents an abstract Grid machine (see Definition 1). An experiment is instantiated by a workflow schedule. The manual annotation of the workflow with ZEN directives for static scheduling purposes, as shown (for the sake of explanation) in Example 1, is obviously impractical and of little use in large-scale dynamic Grid systems. Instead, our static scheduler annotates the workflow program with ZEN directives using a special run-time instrumentation library by retrieving the concrete machine information from the Globus information service.

4.2 Performance Prediction

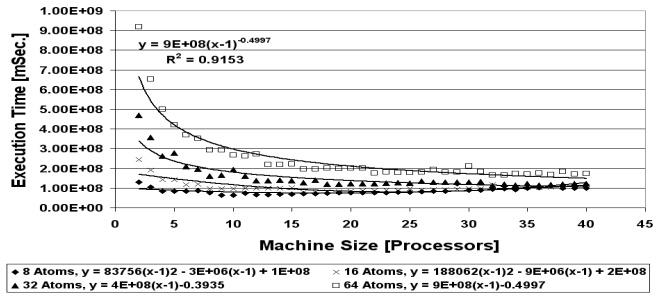
The computation of workflow metrics for scheduling purposes relies on existing *prediction models* for each individual task, which is a difficult research topic on its own that goes beyond our work for this paper. Together with the WIEN2k physicists, we developed for this case study appropriate ad-hoc cost functions for the most critical workflow tasks. For instance, we approximate the execution time of an LAPW1 k-point as: $T_{LAPW1} = \frac{W_{LAPW1}}{v}$, where $W_{LAPW1} = 7 \cdot A \cdot N^2 + N^3$ stands for work, A represents the number of atoms, N represents the matrix size, 7 is a scaling factor, and v is a quantification for the machine speed. Similarly, $T_{LAPW2} = 10\% \cdot T_{LAPW1}$.

For LAPW0 we use existing measurements of a previous exhaustive scalability study which we described in [8], automatically conducted using ZENTURIO. We generate regression functions of various types (i.e., linear, polynomial, logarithmic, exponential, power) to approximate the results on space points that have not been measured and choose the one with the best regression coefficient (i.e., closest to one). Figure 3(a) displays the scalability regression functions for four representative LAPW0 problem sizes executed on a homogeneous Beowulf cluster (with 700 MHz CPU clock rates). Figure 3(b) calculates the regression function for the work expressed in floating point operations based on which we approximate the LAPW0 execution time on different machines.

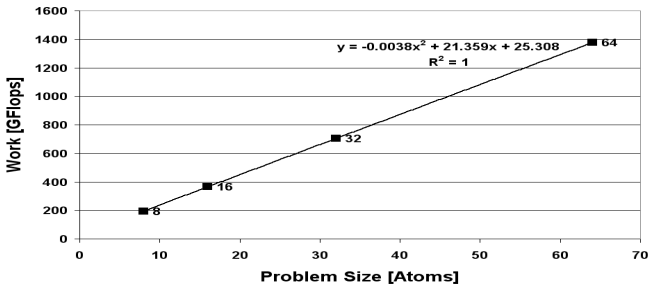
We approximate the file transfer (`case.vector`) time between two LAPW1 and LAPW2 k-point computations as: $T_{12} = \frac{200 \cdot N \cdot A}{v_{12}}$, where v_{12} represents the bandwidth of a TCP stream between the source and the destination machines.

4.3 Workflow Performance Metrics

Let $\mathcal{A}(M_1, \ldots, M_n) = \left(Nodes^{JS} \cup Nodes^{FT}, Edges\right)$ denote a workflow application and $\mathcal{I}(m_1, \ldots, m_n)$ a workflow



(a) LAPW0 machine size scalability.



(b) LAPW0 problem size work (Flops).

Figure 3: Regression performance prediction functions for LAPW0.

schedule (see Definition 2). Let $M = \bigcup_{i=1}^{n} m_i$ denote the set of underlying concrete machines of $\mathcal{I}$, and $|M|$ the cardinality of the set M. Based on our workflow representation, we propose a range of useful workflow performance metrics that implement the interface defined in Definition 2, which can be plugged in as objective functions to be maximised by the static scheduler. Since some of the workflow metrics require minimisation, they had to be subtracted from a large enough constant *Const*.

- *Execution time:* $\mathcal{F}(\mathcal{I}) = Const - T_{\mathcal{I}}$, $T_{\mathcal{I}} = end(N_p)$, where $\{N_1, \ldots, N_p\}$ is the *critical schedule path* and $end(N_p)$ if the termination time of task N_p;
- *Speedup:* $\mathcal{F}(\mathcal{I}) = S_{\mathcal{I}}$, $S_{\mathcal{I}} = \frac{T_{\mathcal{I}}^{seq}}{T_{\mathcal{I}(m_1,\ldots,m_n)}}$, where $T_{\mathcal{I}}^{seq} = \min_{\forall i \in [1..n]} \{T_{\mathcal{I}(m_i,\ldots,m_i)}\}$, $\mathcal{I}(m_i, \ldots, m_i)$ is the sequential schedule on the machine m_i, and $T_{\mathcal{I}(m_i,\ldots,m_i)} = \sum_{\forall JS \in Nodes^{JS}} T_{JS}$;
- *Efficiency:* $\mathcal{F}(\mathcal{I}) = E_{\mathcal{I}}$, $E_{\mathcal{I}} = \frac{S_{\mathcal{I}}}{|M|}$; where $|M|$ denotes the cardinality of set M (i.e., number of concrete Grid machines used);
- *Communication:* due to file transfer tasks on the critical path: $\mathcal{F}(\mathcal{I}) = Const - C_{\mathcal{I}}$, $C_{\mathcal{I}} = \sum_{\forall N \in \rho \cap Nodes^{FT}} T_N$, where ρ is the critical schedule path;
- *Synchronisation:* due to task dependencies on the critical path: $\mathcal{F}(\mathcal{I}) = Const - SY_{\mathcal{I}}$, $SY_{\mathcal{I}} = T_{\mathcal{I}} - \sum_{\forall N \in \rho} T_N$, where ρ is the critical schedule path;
- *Total Overhead:* $\mathcal{F}(\mathcal{I}) = Const - O_{\mathcal{I}}$, where $O_{\mathcal{I}}$ is given by Amdahl's law: $O_{\mathcal{I}} = T_{\mathcal{I}} - \frac{T_{\mathcal{I}}^{seq}}{|M|} = \sum_{N \in \rho \cap Nodes^{JS}} T_N + C_{\mathcal{I}} + SY_{\mathcal{I}} - \frac{T_{\mathcal{I}}^{seq}}{|M|}$;
- *Loss of Parallelism:* due to heterogeneity and task dependencies on the critical path: $\mathcal{F}(\mathcal{I}) = Const - LP_{\mathcal{I}}$, $LP_{\mathcal{I}} = O_{\mathcal{I}} - C_{\mathcal{I}} - SY_{\mathcal{I}} = \sum_{N \in \rho \cap Nodes^{JS}} T_N - \frac{T_{\mathcal{I}}^{seq}}{|M|}$;
- *Efficiency + Execution time:* Maximising the efficiency combined with the minimising execution time is a good metric for high *throughput scheduling*, in the context of multiple workflows (super- or meta-scheduling).

5. DYNAMIC SCHEDULING

We have already explained that the traditional static workflow scheduling approach suffers of two limitations: (1) loops are not comprised in the DAG-based workflow model; (2) the Grid is not considered as a dynamic environment where resources can change run-time load and availability.

algorithm *dynamic scheduler*;
input: *workflow:* $\mathcal{A} = (Nodes, Edges)$;
  cycle elimination: $\mathcal{A}_0 = (Nodes, Edges - Edges_{Queued})$,
    where $Edges_{Queued}$ is defined in Section 5.2);
  static schedule: $\mathcal{I}$ = *genetic optimiser*($\mathcal{A}_0$);
  submit workflow: execute($\mathcal{A}, \mathcal{I}$);
  repeat
    t = **sleep** until next scheduling event;
    select tasks for migration:
      $Nodes_{Migr} = \{N \in Nodes \mid state(N,t) = failed \vee state(N,t) = running \wedge PC(N,t) > f_N\}$;
    $\mathcal{A}_t$ = *generate static DAG*($\mathcal{A}, \mathcal{I}, t, Nodes_{Migr}$);
    cancel(N), $\forall N \in Nodes_{Migr}$;
    static reschedule: $\mathcal{I}$ = *genetic optimiser*($\mathcal{A}_t$);
  until $state(N,t) = completed$, $\forall N \in Nodes \wedge succ(N) = \phi$.

Figure 4: The dynamic scheduling algorithm.

We assume that a task $N \in Nodes$ of a running workflow $\mathcal{A} = (Nodes, Edges)$ can be at a certain time instance t in one of the states *queued*, *running*, *completed*, or *failed*, denoted as $state(N,t)$. Figure 4 depicts the dynamic scheduling algorithm in self-explanatory pseudo-code.

5.1 Task Migration

Let N be a running task, W_N its underlying work assigned (as given by our ad-hoc prediction models outlined in Section 4.2), T_N its estimated execution time, and $start(N) = end(N) - T_N$ its start timestamp. We define the *performance contract* of a task N at the time instance $start(N) \leq t < end(N)$ as: $PC(N,t) = \frac{W_N}{W_N(t) \cdot T_N} \cdot (t - start(N)) - 1$, where $W_N(t)$ is the work completed by task N in the time interval $[start(N), t]$. We migrate the task N iff $PC(N,t) > f_N$, where f_N is the *performance contract elapse factor* of the task N. We statically associate appropriate f_N factors with each task (i.e., percentage from T_N) as part of the workflow specification.

Computing $W_N(t)$ is obviously task dependent. Currently we compute $W_N(t)$ based on trace data that we periodically output from each task, which gives us an online indication of the amount of work computed. For tasks for which $W_N(t)$ is not available, we approximate the ratio $\frac{W_N}{W_N(t)}$ to 1, which translates the performance contract into a less accurate task static timeout operation. The disadvantage is that there is no online indication about the progress of the task, which will not be migrated before the timeout even if the execution has been fatally altered in early stages.

5.2 Static DAG Generation

Let (*Nodes*, *Edges*) denote a statically scheduled DG-based workflow running at the time instance t. The dynamic scheduling algorithm outlined in Figure 4 is based on iterative invocations of the (genetic) static scheduling algorithm. The *static DAG* $\mathcal{A}_t = (Nodes_t, Edges_t)$ given as input to the static scheduler at the scheduling event t is constructed using the following rules:

• $Nodes_t$ comprises the executing tasks $Nodes_{Migr}$ that require migration and the tasks $Nodes_{Queued}$ which are queued (properly running tasks like N_3 in Figure 5 are eliminated): $Nodes_t = Nodes_{Migr} \cup Nodes_{Queued}$, where:

• $Nodes_{Migr}$ comprises the executing tasks that require migration due to failures or performance contract violation:

$$Nodes_{Migr} = \{N \cup Nodes_N \mid \forall\, N \in Nodes \wedge state(N,t) = failed \vee (state(N,t) = running \wedge PC(N,t) > f_N)\}$$

(e.g., see the tasks FT_4 and JS_5 in Figure 5, assuming that the task JS_5 violated its performance contract), where:

• $Nodes_N$ comprises the tasks already completed which need to be reexecuted upon the migration of the task $N = JS(M) \in Nodes^{JS} \vee N = FT(M, M'') \in Nodes^{FT}$ due to the static schedule dependencies induced by the abstract machine M (the static schedule dependency has been defined in Section 2):

$$Nodes_N = \{JS'(M) \in pred^p(N)\} \cup \{FT'(M', M) \in pred^p(N)\}$$

(e.g., see the task $FT_4(M_1, M_2)$ in Figure 5 which contains M_2 as a static schedule dependency to the task $JS_5(M_2)$);

• $Nodes_{Queued}$ comprises the tasks which are queued and have not yet been executed:

$$Nodes_{Queued} = \{N \in succ^p(N_s) \mid \forall\, N_s \in Nodes \wedge state(N_s) = running\}.$$

The completed tasks which are part of workflow loops are therefore included for the next iteration (e.g., see the tasks N_1, N_2, FT_4, N_6 in Figure 5);

• $Edges_t$ comprises the edges that connect the subworkflow tasks from $Nodes_t$ and *eliminate the workflow cycles*:

$$Edges_t = Edges_{DAG} - Edges_{Migr} - Edges_{Queued}, \text{ where :}$$

• $Edges_{DAG}$ comprises the entire subset of edges that connect the workflow tasks from $Nodes_t$:

$$Edges_{DAG} = \{(N, N') \in Edges \mid \forall\, N \in Nodes_t \wedge \forall\, N' \in Nodes_t\}.$$

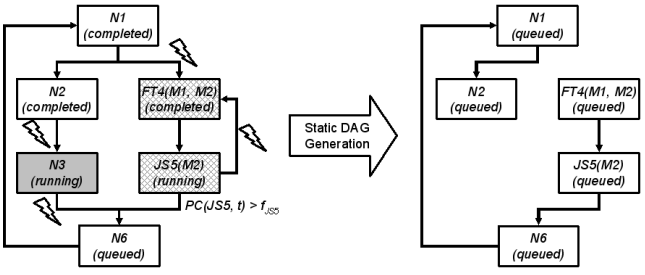


Figure 5: Sample static DAG generation where the task JS_5 violated its performance contract.

The subset $Edges_{DAG}$ eliminates the edges which contain properly running tasks that fulfill their performance contract (e.g., see the edges (N_2, N_3) and (N_3, N_6) within the loop $(N_1, N_2, N_3, N_6, N_1)$ in Figure 5);

• $Edges_{Migr}$ eliminates the cycles within loops which contain tasks that require migration (e.g., see the edge (N_1, FT_4) within the loop $(N_1, FT_4, JS_5, N_6, N_1)$ in Figure 5):

$$Edges_{Migr} = \{(N, N') \in Edges \mid \forall\, N \in Nodes_{Queued} \wedge \forall\, N' \in Nodes_{Migr}\};$$

• $Edges_{Queued}$ breaks the remaining cyclic execution paths ρ by eliminating the edges that violate the node topological order (e.g., see the edge (JS_5, FT_4) within the loop (FT_4, JS_5, FT_4) in Figure 5):

$$Edges_{Queued} = \{(N_p, N_1) \mid \forall\, \rho = (N_1, \ldots, N_p, N_1)\}.$$

6. STATIC SCHEDULER TUNING

We tested our scheduling algorithms on the WIEN2k workflow in a Grid testbed consisting of over 314 machines distributed across four cross-country Austrian Grid sites. Only 16 machines in the testbed have high 3 GHz CPU rates which are optimal for executing the CPU intensive tasks of the WIEN2k workflow. High performance Myrinet interconnections only exist between 64 processors (700 MHz) each organised in a local Beowulf cluster Grid site. We used for this experiment a typical WIEN2k workflow of over 100 nodes (i.e., 50 parallel k-points).

The solutions delivered by the genetic algorithm severely depend on several parameters, which we parameterised with ZEN directives for automatic tuning using ZENTURIO: *population size* $(50 : 200 : 50)$, *crossover* $(0.4 : 1 : 0.2)$ and *mutation* $(0.001, 0.01, 0.1)$ probabilities, *maximum generation* $(100 : 500 : 100)$, *steady state generation percentage* $(10\%, 20\%$, meaning earlier stop of the algorithm if no improvement is made), *fitness scaling factor* $(1, 1.5, 2$ – see [5]), and use of the *elitist model* (i.e., select or not the best individual in the next generation). The cross product of the parameter value sets describe a set of 2880 experiments exhaustively generated and conducted by ZENTURIO. Every experiment represents an instance of the static scheduling algorithm configured using a different genetic parameter combination. Each static scheduling experiment annotates the application with ZEN directives that define the possible instantiations for each abstract machine before starting the genetic algorithm, as explained in Section 4.1 and Example 1. All the experiments use Grid resource information collected at the same time instance (i.e., Grid snapshot) from the Globus information service.

We instantiate the objective function with the predicted workflow execution time. For the purpose of evaluating the quality of the solutions produced by the algorithm, we premeasured the workflow execution on a set of idle (unperturbated) 3 GHz machines, to which we refer as *optimal fitness* $\mathcal{F}_o$. From each experiment we collect three metrics that characterise the performance of the genetic algorithm: (1) *precision* P of the best individual $\mathcal{F}_b$ compared to the artificial optimum $\mathcal{F}_o$, defined as: $P = \frac{\mathcal{F}_b - \mathcal{F}_o}{\mathcal{F}_o} \cdot 100$; (2) *visited points* representing the total set of individuals (i.e., schedules) which have been evaluated by the algorithm during the search process; (3) *improvement* I in the fitness $\mathcal{F}_b$ of the last generation best schedule compared to the first

generation best schedule $\mathcal{F}_f$: $I = \frac{\mathcal{F}_f - \mathcal{F}_b}{\mathcal{F}_b} \cdot 100$. To attenuate the stochastic errors to which the randomised algorithms are bound, we repeat each scheduling experiment for 30 times and report the arithmetic mean of the results in each run.

Due to the large search space (i.e., over $300^{100} = 3 \cdot 10^{200}$ points) and difficult Grid setup (i.e., mostly average and low quality resources), large populations above 50 individuals are required for converging to good solutions (see Figure 6(a)). As expected, the precision improves with the number of generations. Lower population sizes (e.g., 50) do not ensure enough variety in the genes and converge pre-maturely. Larger populations (e.g., 200) converge to good solutions in fewer generations, however, the number of visited points may be unnecessarily large which increases the algorithm duration. The number of visited points (i.e., schedules computed) required for converging to good solutions is of the order of 10^4, which represents a fraction from the overall search space of 10^{200} points (see Figure 6(b)). The improvement in the best individual is remarkable of up to 700% over 500 generations for large populations (see Figure 6(c)). A value of 20% from the maximum generation number is a good effective estimate for checking whether the algorithm reached a steady state (see Figure 6(d)). The higher the crossover probability, the faster the algorithm converges to local maxima (see Figure 6(e)). A correct low mutation probability is crucial for escaping from local maxima and for obtaining good solutions (see Figure 6(f)). In our experiments this had to be surprisingly low (0.001%) due to the rather large population sizes and genes per individual (i.e., 45). Higher mutation probabilities produce too much instability in the population and chaotic jumps in the search space, that do not allow the algorithm to converge to local maxima through crossover. Fitness scaling is crucial for steady improvement over large number of generations (see Figure 6(g)) and produces up to 10 fold improvement in solution. The use of the elitist model (see Figure 6(h)) is beneficial due to the high heterogeneity of the search space and delivers in average 33% better solutions.

As a consequence of this performance tuning experiment, we are currently using the following parameter configuration for the genetic algorithm: population size: 150, crossover probability: 0.9, mutation probability: 0.001, maximum generation: 500, steady state generation percentage: 20%, fitness scaling factor: 2, elitist model: yes. In this configuration, the algorithm constantly produces 25% precision and a remarkable 700% improvement in solution, by visiting a fraction (i.e., $5 \cdot 10^4$) of the entire search space points. The most sensitive parameter that needs be tuned to the workflow characteristics is the mutation probability (i.e., inversely proportional with the population size times the workflow size). The other parameter values have to be tuned to the Grid resource characteristics and are less dependent on the particular workflow.

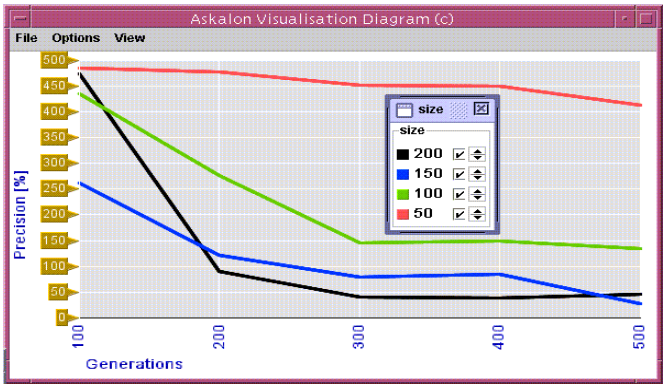
The execution time of the genetic algorithm is proportional with the number of visited points. Considering $7ms$ per workflow evaluation on a 3 GHz Intel Pentium processor, this translates to an average of 7 minutes per serial algorithm execution. To decrease the execution time, we have parallelised the genetic algorithm using the methodology presented in [6] and achieved a parallel speedup of 59 on 64 700 MHz processors. Assuming 16 parallel processors, the parallel genetic algorithm converges to potentially good-enough solutions by visiting a number of search points an order of magnitude lower compared to the existing heuristics from the Min-Min family [3] (i.e., 10^3 versus 10^4).

7. DYNAMIC SCHEDULER EVALUATION

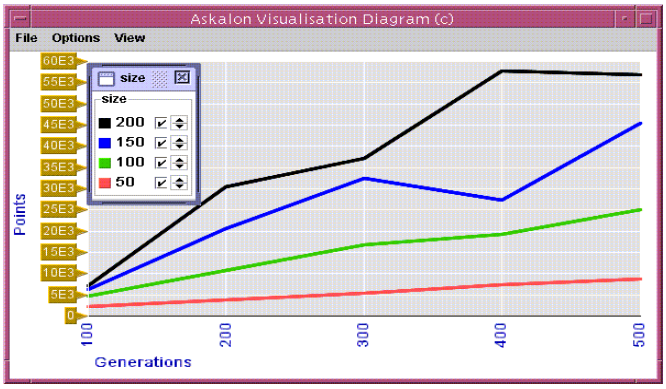
To validate the dynamic scheduling algorithm, we used three WIEN2k cases (two DAG and one DG-based) that correspond to different problem sizes (i.e., number of atoms and matrix size) with different parallelisation sizes (i.e., number of k-points). To achieve an effective evaluation of the algorithm, we have artificially introduced random CPU perturbations to the Grid machines at random time intervals (following a uniform distribution). We use a static value of 50% as a satisfactory performance contract elapse factor for all the workflow tasks. Only idle machines for which prediction models are available are considered by the static scheduler in the optimisation scheduling process. We generate a scheduling event immediately after a static scheduling iteration finished (maximum frequency), since we run the dynamic scheduler on a dedicated front-end machine.

Figure 7(a) traces the optimum static DAG execution time delivered by the genetic static scheduler at consecutive scheduling events during the execution of each experimental workflow. As the workflow tasks are scheduled, execute, and terminate, the predicted static execution time of the remaining DAG1 and DAG2 subworkflows obviously decreases with the number of scheduling events. The abrupt decreases happen after the submission of all LAPW1 k-points (the most time consuming workflow tasks) which no longer need to be considered by the static scheduler. The abrupt increases are due LAPW1 tasks that violate their performance contract and need to be reconsidered by the static scheduler for rescheduling, migration, and restart. In the case of the DG-based workflow, the static scheduler always receives the complete workflow as input, but with a different topological node order. This explains why the static execution time does not decrease with the scheduling events.

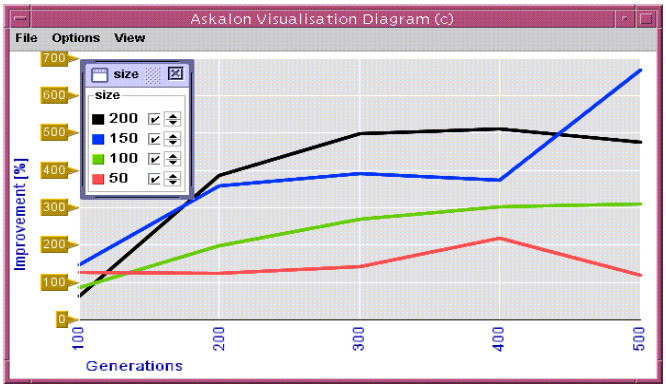
Figure 7(b) traces the overall predicted dynamic workflow execution at consecutive scheduling events during the workflow execution. There are several high peaks in the histogram which are due to severe high perturbations applied to the machines running LAPW1 k-points. As a consequence of the performance contract violation, the scheduler migrated the critical tasks to new machines at the next scheduling event, which drops the next predicted execution time close to the previous value. Through migration, an estimate improvement of about two fold in the overall execution time is achieved (see Figure 7(c)). Since the workflow referred as DAG2 represents a larger problem size than DAG1, the benefit obtained through rescheduling and task migration is higher. The final execution time of the DAG-based workflows is, however, about twice as large as it has been originally predicted by the static scheduler. The performance loss is comes from two sources: (1) a portion under 10% is due to task *rescheduling* using the serial genetic algorithm; (2) the remainder is *replicated work* due to task restarts after migration. For the DG-based workflow, we could not estimate the execution time of the entire workflow (i.e., beyond the execution of one iteration), since the number of loop iterations is statically unknown. As a consequence, in Figure 7 we represent the DG execution time of one workflow iteration only (i.e., entire WIEN2k DAG), which has been successfully kept relatively constant through task migration in two critical occasions.



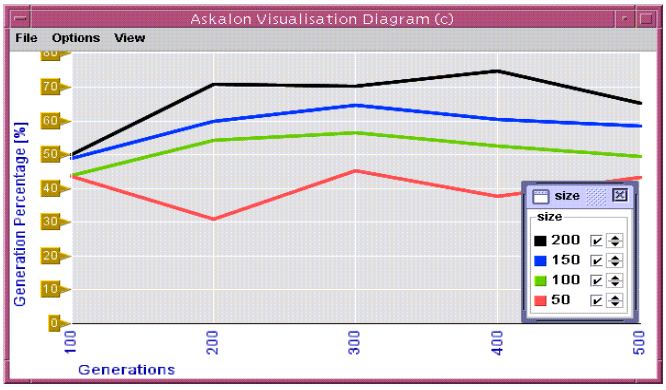
(a) Population size.



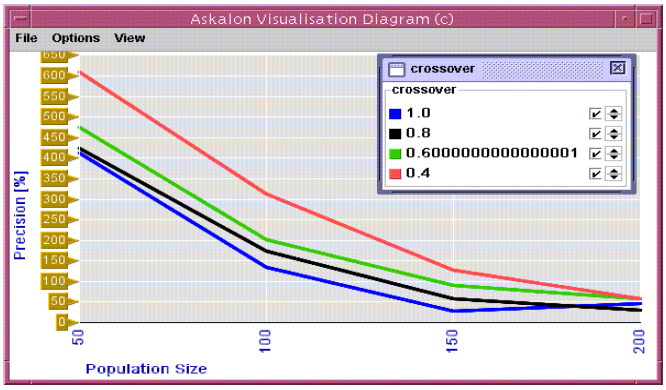
(b) Visited points.



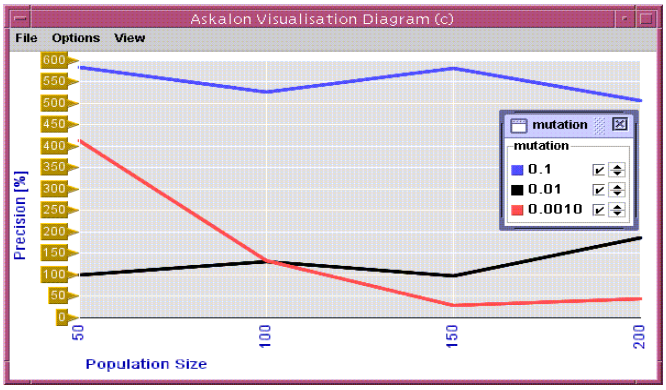
(c) Best individual improvement.



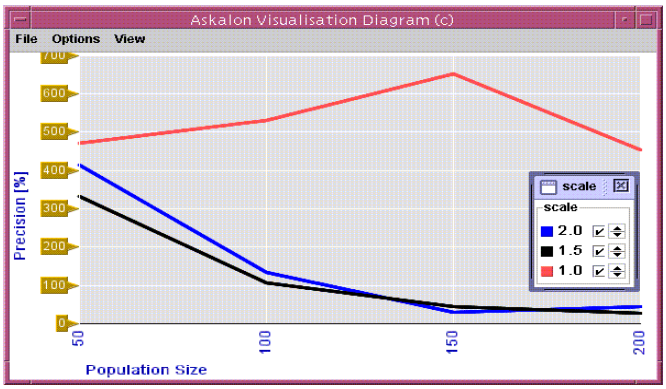
(d) Generation percentage.



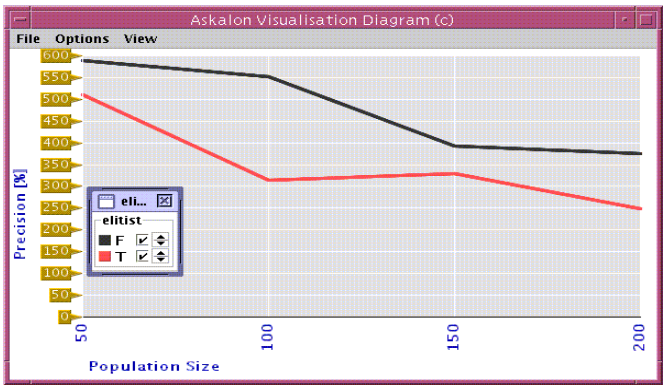
(e) Crossover probability.



(f) Mutation probability.

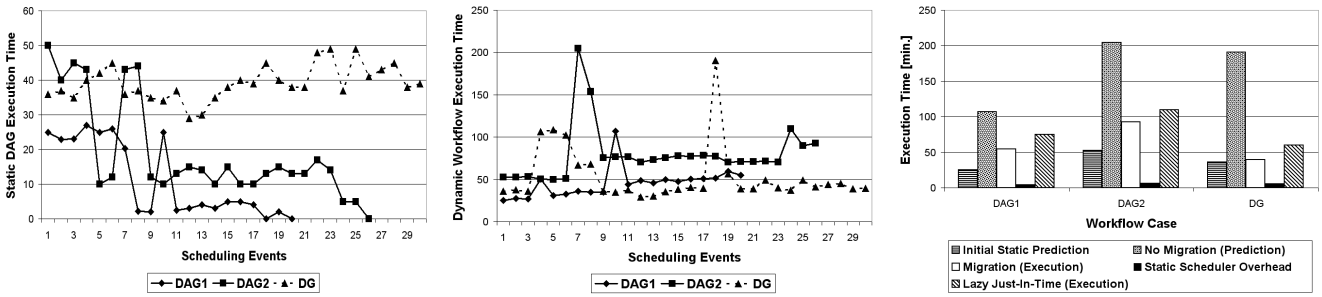


(g) Fitness scaling factor.



(h) Elitist model.

Figure 6: Genetic static scheduler tuning results.



(a) Static DAG execution. (b) Dynamic workflow execution. (c) Schedule comparison.

Figure 7: Dynamic scheduler experimental workflow executions.

In Figure 7(c) we also compare our algorithm against a lazy just-in-time approach that bypasses the static scheduler. The lazy experiments have been performed on the same workflow cases using idle Grid resources. After each task completes, the dependent tasks are scheduled on the resources that deliver the lowest execution times ($\mathcal{O}(m \cdot n)$ complexity). The execution times obtained were in average three times higher than the statically optimised ones and even 25% higher than the workflow executions with task perturbation and migration.

8. CONCLUSIONS

The contributions of this paper can be summarised as follows. (1) We have presented a new hybrid approach for dynamic scheduling of DG-based workflows that include recursive loops, which are centric to many scientific problems. The dynamic scheduler is based on the iterative invocation of classical DAG-based optimisation heuristics, generated using well-defined task migration and cycle elimination rules. (2) We have designed a modular optimisation tool that integrates general purpose heuristics for the maximisation of arbitrary objective functions to be plugged in through a well-defined platform independent interface. (3) We have used this tool to implement the static DAG-based scheduler using genetic algorithms, as a transfer of technology to the Grid domain. We have shown a systematic methodology for application parameter tuning using our optimisation tool which can be easily applied to other similar tuning problems.

Genetic algorithms prove to be quite effective in finding good static workflow schedules within large computational Grids. In our experiments, a carefully tuned genetic algorithm delivered 700% improvement and 23% accuracy for a 100 node workflow in a Grid consisting of over 300 machines by visiting less than 10^5 from 10^{200} search space points. Fitness scaling, a correct mutation probability (inversely proportional with the population size times the number of genes per individual) and a large enough number of visited points (population size times the number of generations) are the most crucial parameters that need appropriate tuning for fast convergence of the genetic algorithm to good solutions. A parallel implementation of the genetic algorithm converges to satisfactory solutions faster than other heuristics (up to an order of magnitude).

We have shown three experiments in which the dynamic scheduler monitored the execution of three sample workflow instantiations and improved the execution time by a factor of two through task migration. The statically optimised execution times were about three times better than the ones delivered by a lazy just-in-time dynamic scheduling algorithm that schedules only one workflow task at a time.

In the future we will investigate new general purpose heuristics for scheduling and automatic performance tuning of Grid workflow, including gradient descent and simplex methods, or other evolutionary algorithms like simulated annealing. ZENTURIO is part of the ASKALON programming environment and tool-set for cluster and Grid computing [4].

9. REFERENCES

[1] 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.

[2] Ewa Deelman et. al. Mapping abstract complex workflows onto grid environments. *Journal of Grid Computing*, 1:25–39, 2003.

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

[4] 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*, To appear. http://dps.uibk.ac.at/askalon/.

[5] David E. Goldberg. *Genetic Algorithms in Search, Optimization 6 Machine Learning*. Reading. Addison-Wesley, Massachusetts, 1989.

[6] Yu-Kwong Kwok and Ishfaq Ahmad. Efficient scheduling of arbitrary task graphs to multiprocessors using a parallel genetic algorithm. *Journal of Parallel and Distributed Computing*, 47(1):58–77, 1997.

[7] Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471, December 1999.

[8] Radu Prodan and Thomas Fahringer. ZENTURIO: A Grid Middleware-based Tool for Experiment Management of Parallel and Distributed Applications. *Journal of Parallel and Distributed Computing*, 64/6:693–707, 2004.