Run-time Optimisation of Grid Workflow Applications

Rubing Duan, Radu Prodan, Thomas Fahringer

Institute of Computer Science, University of Innsbruck
Technikerstraße 21A, A-6020 Innsbruck, Austria
{rubing,radu,tf}@dps.uibk.ac.at

***Abstract*— The execution of workflow applications on the Grid is a complex issue because of its dynamic and heterogeneous nature. While the Grid provides good potential for achieving high performance, it also introduces a broad set of unpredictable overheads and possible failures. In this paper we present new methods for scalable and fault tolerant coordination of workflows in dynamic Grid environments, including partitioning, static and dynamic optimisation, as well as Virtual Single Execution Environment, incorporated into the ASKALON distributed workflow Enactment Engine. We demonstrate the effectiveness of our methods on a material science workflow application executed in a real-world Grid environment.**

I. Introduction

Workflows today play a very important role in the scientific and business communities. Grid applications are not monolithic entities anymore, but rather complex workflows that need to be distributed and executed efficiently in a highly heterogeneous and dynamic environment like the Grid which is a challenging problem. Although good workflow scheduling is the foundation for high-performance, the dynamic nature of Grid environments opens a great potential for run-time steering and fault tolerance.

To address these problems, we propose a distributed Enactment Engine for performance-oriented and fault tolerant execution of scientific workflows within the ASKALON Grid application development and computing environment [6]. ASKALON gives the user the opportunity to express workflow applications as a set of activities interconnected through control flow and data flow dependencies using two high-level abstractions: a graphical UML modeling tool and an abstract XML-based language. So called *composite activities* that include sub-workflows and parallel loops allow the user to express large-scale workflows consisting of hundreds to thousands of activities in a compact form. The XML representation of a workflow represents the input to the Enactment Engine that interacts with a Scheduler for appropriate mapping of the workflow activities onto the Grid resources.

Based on the hands-on experience with several real-world applications in a real Grid environment, we have identified three major types of performance overheads. The first type is related with the implementation of the individual workflow activities and is essentially independent of distributed nature of the workflow application (i.e. it is a sequential and parallel processing problem, rather than a Grid problem). For example, we observed that pre-compiled static binaries are often used by scientists as a convenient (but short sighted) way to deploy legacy applications on newly available Grid sites with compatible processors and operating systems but different parallel architectures. The second type of overheads is related to the workflow structure and distributed nature of the Grid application, and includes in principal job submission and file transfer latencies or inappropriate scheduling due to various unpredictable factors such as external load in dynamic Grids or simply inaccurate prediction information used in the mapping process. The third type of overheads is introduced by the ASKALON middleware services to realise their internal functionality and includes fault tolerance (i.e. checkpointing, retry, replication) and online monitoring functionality which we described in [6].

In this paper we aim to devise methods that address and reduce the second type of performance overheads related to the distributed execution workflow applications. The contributions of this paper are both theoretical and experimental. We propose: (1) a distributed service-oriented enactment engine with a master-slave architecture for decentralised coordination of scientific workflows; (2) an algorithm for partitioning a scheduled workflow for distributed coordination among several slave enactment engine services; (3) two phase control and data flow optimisation process for transforming and simplifying the workflow structure to further reduce its execution cost in geographically distributed Grid environments; (4) a new mechanism and algorithm that significantly simplifies the definition of workflows and reduces the cost of communication for compute intensive scientific workflows with large numbers of small sized data dependencies.

The next section overviews the architectural design of the Enactment Engine. Section III presents a formal approach to workflow partitioning, followed by several techniques for optimising workflow executions in Section IV. Section V evaluates our methods for a real-world material science application executed in a real Grid environment. Section VI discusses the related work. Section VII concludes the paper.

II. Architecture

We designed the Enactment Engine using a distributed service-oriented architecture organised in a master-slave communication model which includes three types of services (see Figure 1). (i) One *master engine* receives the XML representation of the workflow and interacts with the Scheduler

1-4244-0344-8/06/$20.00 ©2006 IEEE

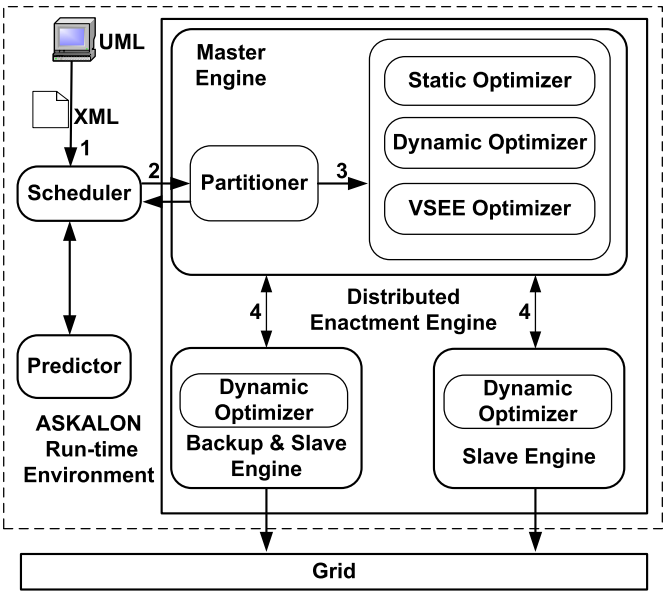


Fig. 1. Enactment Engine architecture.

for appropriate mapping onto the available Grid resources. The master engine monitors the execution of the entire workflow and the state of the slave engines. (ii) Several *slave engines* (usually one for each Grid site) monitor the execution of individual workflow partitions and report to the master whenever individual activities change their state or when the partitions produce some intermediate output data relevant to other partitions or to the overall execution. (iii) If the master engine crashes, a random backup engine (chosen by the master beforehand) becomes the master and immediately selects another backup slave randomly. Such a distributed architecture increases the fault tolerance of the engine and offers improved scalability through decentralised orchestration of large numbers of activities characteristic to scientific workflows. To support run-time optimisation of workflow executions in Grid environments, we enhanced the Enactment Engine with the following two additional components.

Workflow Partitioner: (see Section III) resides within the master engine and distributes the workflow into smaller partitions that can be executed more efficiently and with smaller overheads by individual slave engines, usually one for each Grid site. We define a *Grid site* as a collection of compute units (usually a parallel computer) accessible through access policies established by local administration authorities, usually through a local job queuing system (e.g. see Table III in Section V).

Workflow Optimiser: (see Section IV) performs transformations within workflow partitions, such as grouping of activities to composite activities to reduce the (remote job submission or file transfer) Grid service latencies and optimising the control and data flow for different types of workflow applications. The workflow optimiser has three components: (i) *Static Optimiser* simplifies the workflow structure within the master engine before the workflow execution; (ii) *Dynamic Optimiser* provides support for workflow run-time steering within the slave engine to cope with situations when the execution no longer follows the optimised plan computed by the Scheduler. Additionally, it also handles the case of special workflows whose structure is statically unknown or may change during the execution; (iii) *Virtual Single Execution Environment Optimiser* within the master engine for optimising special case of scientific workflows characterised by a large set of small data dependencies.

In our approach, the run-time optimisation of workflow executions is performed in a four phase procedure, as shown in Figure 1. In the first step, the XML-based workflow specification is delivered to the Scheduler for appropriate mapping onto the Grid resources. Once the concrete workflow schedule has been received, the master engine starts partitioning the workflow, followed by the static optimisation which transforms and simplifies the workflow for a light-weight execution control with reduced Grid service access latencies and data transfer communication overheads. After all these optimisations have been performed, the master engine sends each partition to a slave engine for execution. During run-time, the workflow execution is dynamically improved by the dynamic optimiser.

III. Workflow Partitioning

The basis for distributed execution of workflows in our approach is the workflow partitioning which needs to be performed such that the communication between the master and the slave engines that coordinate the individual partitions is minimised.

To determine the number of partitions of a set of n numbers is a classical problem of combinatorial mathematics called the n-th Bell number, which is an NP-hard problem. Some related partitioning algorithms are proposed to solve this problem, although their algorithms have different goals [1], [5].

Definition: Let $W = (AS, CFD, DFD)$ denote a workflow application, where $AS = \{A_1, \ldots, A_n\}$ is the set of activities, $CFD = \{(A_{source} \delta^c A_{sink}) \mid A_{source}, A_{sink} \in AS\}$ is the set of control flow dependencies, and $DFD = \left\{(A_{source} \delta^d A_{sink}, Data) \mid A_{source}, A_{sink} \in AS\right\}$ is the set of data flow dependencies, where $Data$ denotes the I/O data (i.e. parameters or files) to be transferred between the A_{source} and A_{sink} activities. We define a *workflow partition* as the largest sub-workflow with the following properties: (1) all activities are scheduled on the same Grid site; (2) there must be no control flow and data flow dependencies to / from activities that have predecessors / successors within the partition.

Goal: The goal of the partitioning algorithm presented in this section is to generate a partitioned workflow $W_P = (PS_P, CFD_P, DFD_P)$ from a workflow $W = (AS, CFD, DFD)$, where $PS_P = \{P_1, \ldots, P_n\}$ is the set of partitions $\wedge \bigcap_{i=1}^{n} P_i = \phi \wedge \bigcup_{i=1}^{n} P_i = AS$, and n is minimum.

We base our partitioning algorithm on graph transformation theory [2] 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 (W, R) denote a workflow transformation system, where R denotes the set of graph transformation rules. We approach the workflow partitioning problem using a four step transformation sequence:

$$\left(W \overset{R_{CF}}{\Longrightarrow} W_{CF},\ W \overset{R_{DF}}{\Longrightarrow} W_{DF}\right) \overset{R_{M1}}{\Longrightarrow} W' \overset{R_{M2}}{\Longrightarrow} W_P,$$

where $W_{CF} = (PS_{CF}, CFD_{CF}, DFD_{CF})$, $W_{DF} = (PS_{DF}, CFD_{DF}, DFD_{DF})$, $W' = (PS', CFD', DFD')$, and W_P are partition sets generated using different transformation rules that preserve the control and data flow dependencies of the original workflow W.

Step 1: $W \overset{R_{CF}}{\Longrightarrow} W_{CF}$*.:* Partition the original workflow according to three control flow dependency rules R_{CF}:

1) Every activity of the workflow must belong to exactly one partition: $\forall\ A \in AS,\ \exists\ P \in PS_{CF} \wedge A \in P \wedge A \notin P' \ \wedge\ \forall\ P' \in PS_{CF} \setminus P$;
2) Every partition is one composite or atomic activity. Currently we perform this step by using information provided by the user in the XML-based workflow representation and mapping one composite activity (e.g. parallel activity consisting of a set of independent atomic activities) to one partition;
3) No control flow dependencies between intermediate activities in different partitions are allowed: $\forall\ A_1 \in P_1 \in PS_{CF} \ \wedge\ (pred(A_1) \in P_1 \ \vee\ succ(A_1) \in P_1) \ \wedge\ (\nexists\ (A_1\ \delta^c\ A_2) \in CFD_{CF} \ \wedge\ \nexists\ (A_2\ \delta^c\ A_1) \in CFD_{CF},\ \forall\ A_2 \in P_2 \in PS_{CF})$, 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 processor number 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 Figure 2(a) we partition all atomic activities of the composite activities `IF`, `While`, and `Sequence` into one partition, respectively, which produces the following control flow partitioning:
$PS_{CF} = \{\{A1\}, \{A2\}, \{A3, \ldots, A6\}, \{A7, \ldots, A10\}, \{A11\}, \{A12, A13\}\}$.

Step 2: $W \overset{R_{DF}}{\Longrightarrow} W_{DF}$*.:* Partition the original workflow according to three data flow dependency rules R_{DF}:

1) Each activity of the workflow must belong to exactly one partition: $\forall\ A \in AS,\ \exists\ P \in PS_{DF} \ \wedge\ A \in P \ \wedge\ A \notin P',\ \forall\ P' \in PS_{DF} \setminus P$;
2) Eliminate the data dependencies between activities scheduled on the same Grid site: $DFD_{DF} = DFD \setminus (A_1 \delta^d A_2, Data),\ \forall\ A_1, A_2 \in AS \ \wedge\ schedule(A_1) = schedule(A_2)$;
3) Activities scheduled on the same Grid site belong to the same partition: $\forall\ A_1 \in P \in W_{DF} \ \wedge\ \forall\ A_2 \in P \ \wedge\ schedule(A_1) = schedule(A_2)$.

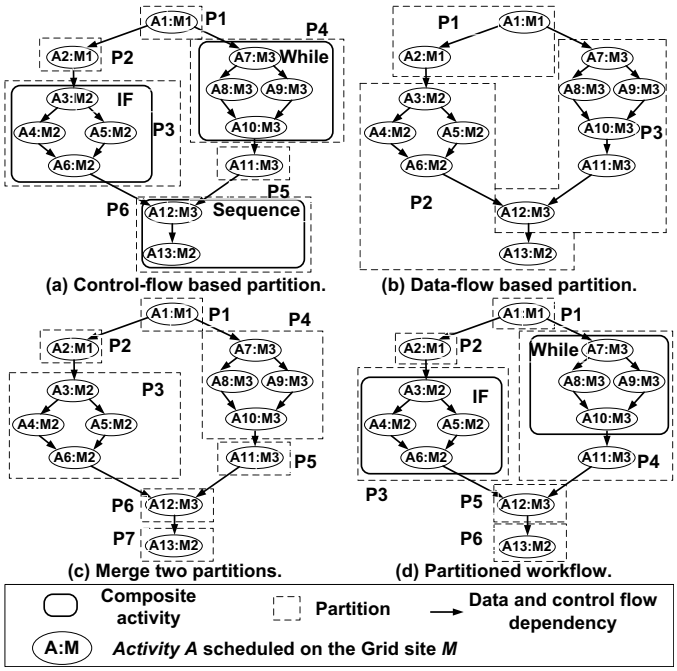


Fig. 2. Workflow partitioning example.

Figure 2(b) displays the result of the data flow partitioning according to the schedule of the activities: $PS_{DF} = \{\{A1, A2\}, \{A3, \ldots, A6, A13\}, \{A7, \ldots, A11, A12\}\}$.

Step 3: $(W_{CF}, W_{DF}) \overset{R_{M1}}{\Longrightarrow} W'$*.:* Merge the two sets PS_{CF} and PS_{DF} of control flow and data flow based partitions computed in the previous two steps into one partition set, as follows: $W' = \bigcup_{\substack{\forall PS_i \in PS_{CF} \\ \forall PS_j \in PS_{DF}}} \{PS_i \cap PS_j\}$, while preserving the control and data flow dependencies and the partitioning goals formally described in the beginning. For our example in Figure 2(c) we obtain:
$PS' = \{\{A1\}, \{A2\}, \{A3, \ldots, A6\}, \{A7, \ldots, A10\}, \{A11\}, \{A12\}, \{A13\}\}$.

Step 4: $W' \overset{R_{M2}}{\Longrightarrow} W_P$*.:* 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):

$$PS_P = \bigcup_{\forall\ P_i \neq P_j \in W'} \{\{P_i \cup P_j\} \setminus \{P_i\} \setminus \{P_j\} \mid \forall\ A_1 \in P_i \ \wedge\ \forall\ A_2 \in P_j \ \wedge\ \nexists\ (A_1 \delta^d A_2) \in DFD \ \wedge\ (P_i \delta^c P_j) \in CFD'\};$$

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 any-

more and the largest partitions are achieved:

$$PS_P = \bigcup_{\forall\ P_i \neq P_j \in W'} \{\{P_i \cup P_j\} \setminus \{P_i\} \setminus \{P_j\} \mid \neg(P_i \delta^d P_j \in DFD') \ \wedge \ (P_i \delta^c P_j \in CFD') \wedge ((\nexists\ P_x \neq P_j \in W' \mid (P_i \delta^c P_x \in CFD')) \wedge (\nexists\ P_x \neq P_i \in W' \mid (P_x \delta^c P_j \in CFD')))\}.$$

Therefore, $PS_P = \{\{A1\}, \{A2\}, \{A3, \ldots, A6\}, \{A7, \ldots, A11\}, \{A12\}, \{A13\}\}$.

The partitioning of a workflow helps the slave engines execute the workflow partitions independently with little asynchronous communication among themselves. The partitioning of the workflow also contributes to the reduction of the latency and coordination overheads of large numbers of activities characteristic to our scientific workflows.

IV. Workflow Optimisation

The workflow optimiser focuses on further simplifying the sub-workflows produced in the partitioning phase to achieve an easier and faster execution on the Grid. There are two phases in the optimisation process: static optimisation performed by the master engine before sending the partitions to the slave engines, and dynamic optimisation performed by the slave engines during the execution of a workflow.

A. Static Optimisation

According to our previous experiences [6], one remote job submission to a Grid site with a batch queuing system has about $10 - 20$ seconds of overhead mainly due to mutual authentication latency and polling for status change. This overhead may be significantly larger if the access to Grid sites is performed through batch queuing systems and becomes critical for large workflows comprising hundreds to thousands of activities like in the case of our real-world workflows. Similarly, the (GridFTP) file transfer latency is of about one second which is rather critical for the large numbers of small files that are produced by our applications.

The objective of the static optimisation is simplify and reduce the workflow structure and size by merging atomic activities and data dependencies that reduces the Grid latencies and other sources of execution overheads.

1) Static Control Flow Optimisation: This optimisation step aims to wrap the atomic activities defined by the user into composite activities that can be executed as one single remote job submission on a Grid site, which reduces the overall latency and also decreases the complexity of large workflows. The static workflow optimiser receives a workflow partition P and performs a transformation that produces a new partition CP that merges the activities linked through control flow dependencies but with no run-time data dependencies (i.e. since they are scheduled on the same Grid site) into composite activities are executed as an atomic unit of work (i.e. remote GRAM job submission): $CP = \{CP_1, \ldots, CP_n\}$, where $CP_i = \{A\} \ \vee \ (\forall\ A_1 \in CP_i,\ \exists\ A_2 \in CP_i \ \wedge \ ((A_1 \delta^c A_2) \in CFD_P) \ \vee \ (A_2 \delta^c A_1) \in CFD_P) \ \wedge \ \nexists\ A_3 \in CP_i \ \wedge \ ((A_1 \delta^d A_3) \in DFD_P \vee (A_3 \delta^d A_1) \in DFD_P)\},\ \forall\ i \in [1, n]$.

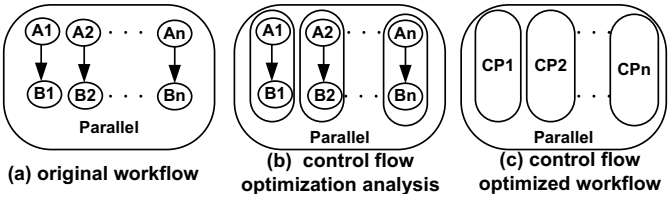


Fig. 3. Static control flow optimisation.

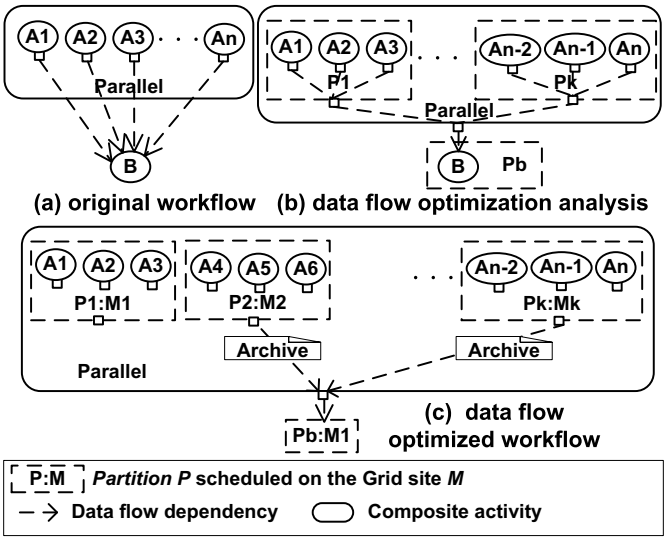


Fig. 4. Static data flow optimisation.

Figure 3(a) illustrates one typical static control flow optimisation in a workflow consisting of activities $A_1, \ldots, A_n$ and $B_1, \ldots, B_n$, where A_i and B_i are linked through a direct control flow dependency and have been scheduled to the same Grid site, which means that any eventual data dependency has been eliminated in the second step of the partitioning algorithm. Figure 3(b) displays the analysis of the control flow optimisation, which groups activities A_i and B_i in one single composite activity that simplifies the workflow and, therefore, reduces the job submission latencies to half in this particular example (see Figure 3(c)).

2) Static Data Flow Optimisation: After the optimisation of the control flow, the static data flow optimiser reorganises first the input and output ports of the partitions and composite activities. Thereafter, it analyses the data dependencies between all activities, groups them according to all dependencies involving the same source and destination Grid sites, and generates a file transfer activity of a single compressed archive whenever the source and destination sites are different: $DFD_P = \bigcup_{\forall P_1, P_2 \in PS_P} \{(P_1, P_2, Archive)\}$, where $Archive = \bigcup_{\forall (A_1, A2, Data) \in DFD \wedge A_1 \in P_1 \wedge A_2 \in P_2} \{Data\}$ is a compressed archive of all data dependencies between partitions P_1 and P_2 (typically instantiated during execution by files).

Figure 4(a) presents a typical example in which activity B collects data from a large number of parallel activities $A_1, \ldots, A_n$. First of all, the data flow analysis packs the data output ports of all activities belonging to the same partition (i.e. scheduled on the same site – see Figure 4(b)). Afterwards, one single (GridFTP-based) file transfer activity is generated between the partitions that are scheduled on different sites which reduces the number of file transfers from n to one in this example (see Figure 4(c)).

B. Dynamic Optimisation

There may occur many external factors during the execution of large workflows in dynamic Grid environments that no longer follows the plan computed by the Scheduler. Such unpredictable factors may include unexpected queuing times, external load on processors (e.g. on Grid sites that also serve as student workstation labs in our real Grid environment – see Section V), unpredictable availability of processors on workstation networks (e.g. if a student shuts down a workstation or reboots it to Windows mode), jobs belonging to other users on parallel machines, congested networks, or simply inaccurate prediction information. In addition, we often encountered in our real Grid environment sites that offer a limited capacity for certain resources, for example small number of I/O nodes that only allow a limited number of concurrent file transfers, otherwise generate a denial of service attack. The dynamic optimiser handles the run-time steering of the workflow execution and aims to minimise the losses due to such unpredictable factors that violate the optimised static mapping computed by the Scheduler.

The scientific workflows that we use to validate our work are typically characterised by a large number of independent activities to be executed in parallel, which we model in XML as a composite activity using a *parallel loop* syntax. Moreover, such workflows can have a dynamic structure which is known only at run-time, for example the number of activities to be executed in parallel may depend on the value taken by some output port of a predecessor activity (see Section V for a real world example). The dynamic optimiser handles this situation by building the new shape of the workflow as soon as it is known and sending it back to the Scheduler for a new optimised mapping onto the Grid.

In addition, executing such large numbers of parallel activities in dynamic Grid environments often produces a load imbalance (for example due to some unexpected external jobs submitted to the queue) that leaves some of the Grid sites idle, while others are overloaded with activities waiting in the queue. To handle this situation, the Enactment Engine regularly checks the load of available Grid sites based on the number of activities queued and, if an uneven distribution is detected (using execution time information provided by the Performance Prediction service), it selects some of the queued activities for migration and replicates them to the less loaded sites (e.g. with free processors). Additionally, the engine must also replicate the necessary input files as part of a data flow optimisation process.

C. Virtual Single Execution Environment

Certain scientific workflow applications are characterised by a large (hundreds to thousands) number of activities with complex data dependencies instantiated by a large number (hundreds to thousands) of small files (several kilobytes each). In such cases, the overhead of communication is dominated by latencies for sending individual small files where the effective data transfer is small. This effect can be easily grasped through a commonly used analytical formula that models the predicted communication time T_{comm} for sending n files of aggregated size $size$: $T_{comm} = n * Latency + \frac{size}{Bandwidth}$, where the latency for mutual authentication to the GridFTP server (orders of seconds) clearly dominates the communication time if n is large.

To handle this case, we propose a data flow optimisation called *Virtual Single Execution Environment* (VSEE) which replaces the data dependencies between activities with the full I/O *data environment*, defined for a partition P as follows:

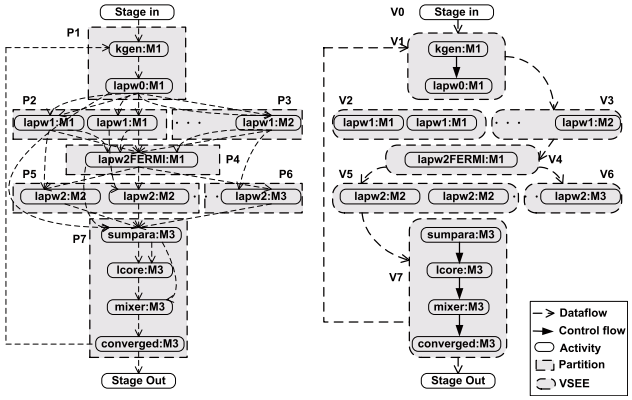
$$V_P = \bigcup_{\forall (P',P,Data) \in DFD_P} V_{P'} \bigcup_{\forall (P,P'',Data) \in DFD_P} \{Data\}.$$

Clearly, the following property holds: $\exists\ (P', P, Data) \in DFD_P \iff V_{P'} \subset V_P$.

Another benefit of using VSEE is the fact that specifying lots of data input and output ports for activities are painful and error prone. With this technique, users can assume that activities have one single aggregated data dependency to their predecessors, which eliminates the need to specify all fine grained logical data ports explicitly. This shields the user from the complexity of the workflow definition and it gives scientists a more friendly interface to the Grid. The VSEE mechanism can also reduce the overhead of activity migration defined in the dynamic optimisation process as we will show in Section V.

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 the complex data dependencies between activities to one environment dependency between partitions. At run-time, the engine packs the data environment which can be transferred by one single file transfer. VSEE, therefore, noticeably reduces the latency and the number of intermediate data transfers for compute intensive Grid applications that have large amounts of small sized data dependencies.

Figure 5 illustrates one real world workflow (that we also use for the experiments in Section V) to be executed on three Grid sites $\{M_1, M_2, M_3\}$. First of all, the workflow is split into seven partitions $PS_P = \{P_1, \ldots, P_7\}$ (see Figure 5(a)) based on the algorithm presented in Section III. Thereafter, the data flow between partitions is optimised according to the VSEE-based relationships depicted in Table I. For example, transferring data between partitions only according to the data flow dependencies requires P_6 receive the data from $V_{in} \cup V_1 \cup V_2 \cup V_3 \cup V_4 = V_4$, since $V_{in} \subset V_1 \subset V_2 \subset V_3 \subset V_4$. Table II displays the final result of this VSEE data flow optimisation process automatically performed by the Enactment Engine.



(a) Partitioned workflow and original data flow.

(b) VSEE optimised data flow.

Fig. 5. VSEE example.

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	–	$\subset$	$\subset$	$\subset$	$\subset$	$\subset$	$\subset$	
V_2		–		$\subset$	$\subset$	$\subset$		
V_3			–	$\subset$	$\subset$	$\subset$		
V_4				–	$\subset$	$\subset$	$\subset$	
V_5					–		$\subset$	
V_6						–	$\subset$	
V_7	$\subset$						–	$\supset$

TABLE I

VSEE RELATIONSHIPS.

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	✓							✓

TABLE II

MINIMUM VSEE TRANSFER SET.

For certain compute intensive applications characterised by large numbers of small data dependencies, the VSEE mechanism can drastically decrease the number of file transfers up to several orders of magnitude, while increasing the data size by several factors. Considering fixed latencies of about one second for every individual file transfer and the improved bandwidth utilisation for transferring large data archives, this mechanism can significantly improve the communication time in scientific workflows as we will illustrate in Section V.

V. EXPERIMENTS

In this section, we evaluate our proposed methods using a real world scientific workflow application executed in a national Grid infrastructure.

WIEN2k (see `http://www.wien2k.at/`) is a program package for performing electronic structure calculations of solids using density functional theory based on the full-potential (linearized) augmented plane-wave ((L)APW) and local orbital (lo) method. We have ported the application onto the Grid by splitting the monolithic code into several course-grain activities coordinated in a workflow, as already illustrated in Figure 5(a). The `lapw1` and `lapw2` activities can be solved in parallel by a fixed number of so-called *k-points*. A final activity `converged` applied on several output files tests whether the problem convergence criterion is fulfilled. The number of recursive loops is statically unknown.

One peculiarity of this workflow is that the number of parallel `lapw1` and `lapw2` activities is unknown until the first activity `lapw0` completes its execution. Since this number is statically unknown, the Enactment Engine assumes one serial activity in each case and performs no partitioning since the workflow only has a total of nine serialised activities that the Scheduler maps onto the same Grid site.

After `lapw0` completes, the dynamic optimiser reads an `lapw0` output port that indicates the number of subsequent parallel k-points, builds the complete shape of the workflow, and sends the full workflow back to the Scheduler for optimised mapping onto the Grid. We use a variation of the Heterogeneous Earliest Finish Time algorithm to map the workflow activities onto the Grid sites, which proved to be the most efficient heuristic for this application in previous experiments [6] conducted in a static Grid environment (i.e. no external load) using accurate prediction information obtained from previous training runs. After receiving the full workflow schedule, the Enactment Engine can start partitioning the workflow.

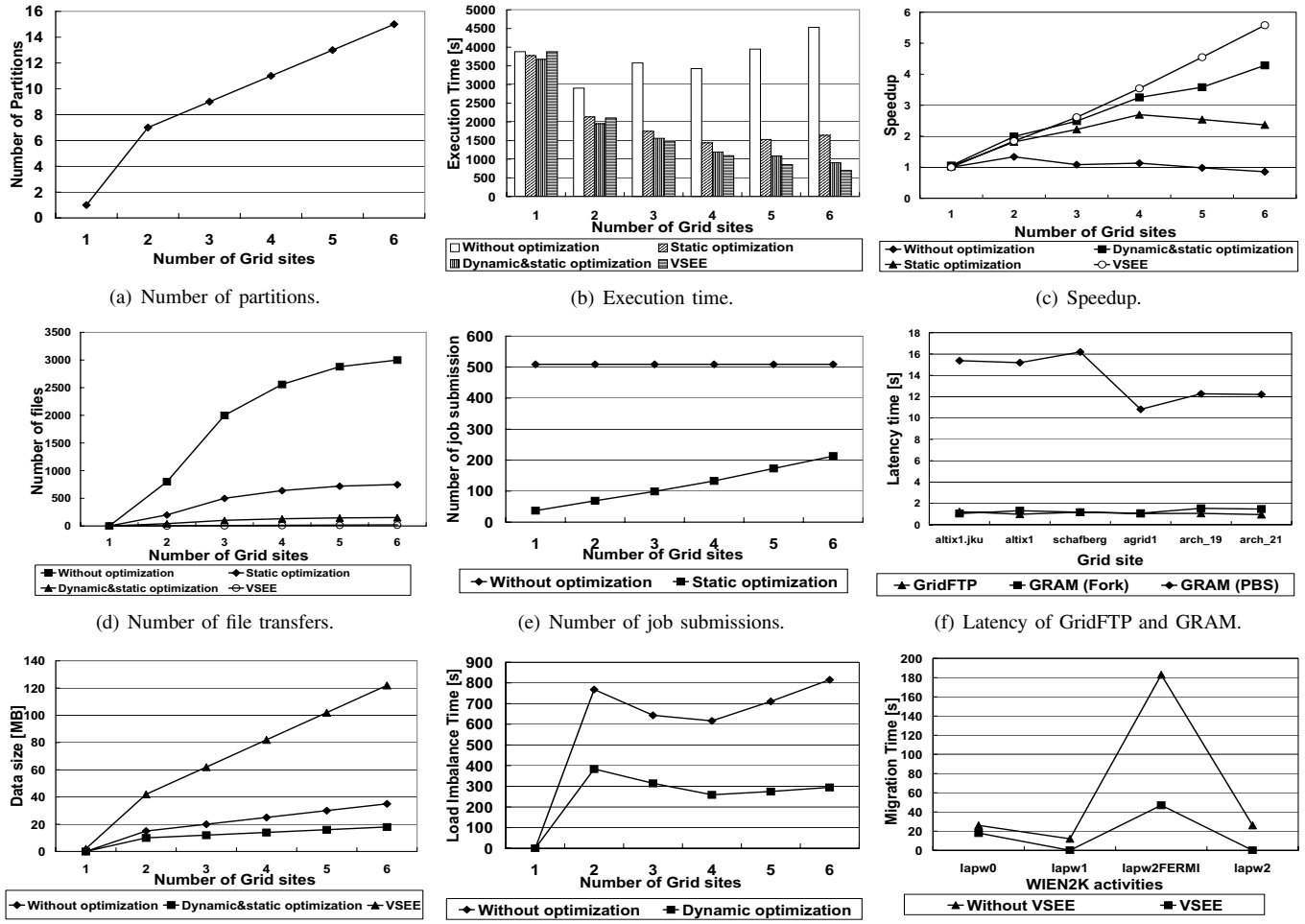
We chose for our experiments a WIEN2k problem size that produces 250 parallel k-points at run-time, which means a total of over 500 workflow activities. We executed this application on a subset testbed of a national Grid infrastructure [4] consisting of a set of parallel computers and workstation networks accessible through the Globus toolkit and local job queuing systems as separate Grid sites. We first executed the workflow application on the fastest site available (i.e. altix1.jku in Linz) that gives the 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, as indicated by the rank column in Table III, and observed the benefits or losses obtained by executing the same problem size in a larger Grid environment.

First, Figure 6(a) presents the number of WIEN2k partitions

Rank	*Site*	*#*	*CPU, GHz*	*Manager*	*Location*
1	altix1.jku	16	Itanium 2, 1.6	Fork	Linz
2	altix1.uibk	16	Itanium 2, 1.6	Fork	Innsbruck
3	schafberg	16	Itanium 2, 1.6	Fork	Salzburg
4	agrid1	16	Pentium 4, 1.8	PBS	Innsbruck
5	arch_19	20	Pentium 4, 1.8	PBS	Innsbruck
6	arch_21	20	Pentium 4, 1.8	PBS	Innsbruck

TABLE III

THE AUSTRIAN GRID TESTBED.



(a) Number of partitions.

(b) Execution time.

(c) Speedup.

(d) Number of file transfers.

(e) Number of job submissions.

(f) Latency of GridFTP and GRAM.

(g) Size of transferred files.

(h) Overhead of load imbalance.

(i) Overhead of activity migration.

Fig. 6. WIEN2k experimental results.

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 with the number of sites.

Figure 6(b) shows the execution times for running the same WIEN2k problem on different Grid size configurations ranging from one to six aggregated sites. Similarly, Figure 6(c) displays the Grid speedup computed as the ratio between execution time on multiple distributed sites and the execution time on the *fastest* local site available (altix1.jku in Linz). First of all, without any optimisation the performance and the speedup decrease with the increase in the number of Grid sites used for scheduling and running the workflow. With static and dynamic optimisation, the WIEN2k execution time improves because of the simplified data flow and balanced execution of the `lapw1` and `lapw2` parallel activities. We exhibit a slow down from five to six Grid sites using static optimisation because of the increased communication time across six distributed sites.

Figures 6(d) and 6(e) 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 6(f) displays the average GridFTP and GRAM latencies (i.e. measured for each job from the submission time until it becomes active) experienced in our runs, which ranges from one to 18 seconds when a queuing system is used underneath. Figure 6(g) shows that the 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 that drastically reduces the latencies (i.e. mutual authentication to the GridFTP service). The improvement of dynamic optimisation is due to several external jobs that we submitted to the fastest Grid site which caused several `lapw1` and `lapw2` activities wait in the queue. The consequence is an increased load imbalance in the execution of the `lapw1` and `lapw2` parallel activities, which is reduced to half by dynamic optimisation as shown in Figure 6(h).

Figure 6(i) compares the data transfer overheads of the activity migration upon dynamic control flow optimisation with and without the VSEE mechanism. One important aspect is that the data transfer overhead upon migrating a `lapw1`

and `lapw2` activity is zero when using the VSEE mechanism. The reason is that the sequential activities `lapw0` and `lapw2_fermi` replicate all their output to the sites where the following `lapw1` and `lapw2` parallel 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.

VI. Related work

The idea of distributed execution was pioneered by the INCAS [1] prototype which partitioned the workflow into independent subsets that are distributed to the nodes where execution may take place. Other research in performance-oriented distributed computing focused on partitioning algorithms for system-level load balancing or task scheduling [3] that aims to maximise the overall throughput or average response time of the system. The idea, however, was not yet evolved to the Grid computing field.

The Pegasus system [5] uses DAGMan as enactment engine enhanced with data derivation techniques that simplifies the workflow at run-time based on data availability. Pegasus provides a workflow partitioning approach which concentrates on different phases of the execution before the scheduling takes place. Our research, in contrast, focuses on how to partition the workflow after the scheduling phase and coordinate its execution in a decentralised environment, as opposed to the centralised approach of DAGMan.

Kennedy et al. [8] compared several task-based and workflow-based approaches to resource allocation for workflow application based on simulation rather than real executions as in our case. They conclude that workflow-based approaches perform better for data intensive cases while task-based approaches are suited for compute intensive workflows. In this paper we present a new mechanism called VSEE suited for compute intensive workflows with large amounts of small data dependencies. Kennedy et al. also study the impact of uncertainty to the overall workflow schedule, but do not propose run-time optimisation techniques as done in this paper.

Taylor et al. [7] use the Grid Application Toolkit interface to access Grid services through JXTA and Web services. Triana has a distributed engine to distribute sections of a workflow to remote machines. The user has to understand the Grid and the workflow execution, while our approach partitions and distributes the workflow automatically.

Taverna [9] in ^{my}Grid follows a service-oriented Grid architecture which use Freefluo enactor as the execution engine. Taverna focuses on data integration for bio-informatics, fault tolerance, and user-friendly interfaces for biologists. The Enactment Engine and ASKALON target more generic scientific workflows and are not designed for specific domains.

VII. Conclusions

Executing workflow applications in dynamic Grid environments needs run-time optimisation and steering techniques for achieving good performance. In this paper we described the distributed Enactment Engine of the ASKALON application development and computing environment. We proposed a partitioning algorithm for distributing a workflow application that offers improved fault tolerance and lower coordination overheads through a distributed master-slave service-oriented architecture. The engine improves the execution of large-scale workflows on multiple geographically distributed Grid sites through two run-time optimisation phases. The static optimisation simplifies the workflow structure before the execution by archiving and compressing multiple files and merging multiple atomic activities scheduled on the same site, which significantly reduces the huge service latencies exhibited in Grid environments (e.g. authentication latencies to GRAM and GridFTP services). The dynamic optimisation handles peculiar scientific workflows that dynamically change their graph-based structure at run-time and eliminates eventual load imbalance by using idle and redundant resources during executions that do not follow the original plan computed by the Scheduler. In addition, we proposed a new optimisation approach called VSEE that replaces large numbers of data dependencies between individual activities with the entire I/O data environment that significantly reduces data transfer complexity of compute intensive workflow applications. We have validated our methods for a real world material science workflow application consisting of over 500 activities executed in a national Grid testbed consisting of 104 processors geographically distributed on six sites.

Acknowledgement

This research has been supported by the Austrian Science Fund through the SFBF1104 project Aurora and by the European Union through the IST-2002- 511385 project K-Wf Grid.

References

[1] D. Barbara, S. Mehrotra, and M. Rusinkiewicz. Incas: A computation model for dynamic workflows in autonomous distributed environments. Technical report, May 1994.

[2] L. Baresi and R. Heckel. Tutorial introduction to graph transformation: A software engineering perspective. In *Proceedings of the First International Conference on Graph Transformation (ICGT 2002).*, page 402C429. Springer-Verlag, 2002.

[3] S. Chandra and M. Parashar. Towards autonomic application-sensitive partitioning for samr applications. *Journal of Parallel and Distributed Computing*, 65(4):519–531, 2005.

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

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

[6] T. Fahringer, R. Prodan, R. Duan, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H.-L. Truong, A. Villazon, and M. Wieczorek. ASKALON: A Grid Application Development and Computing Environment. In *6th International Workshop on Grid Computing (Grid 2005)*, Seattle, USA, November 2005. IEEE Computer Society Press.

[7] I.Taylor, M.Shields, I.Wang, and R.Rana. Triana applications within Grid computing and peer to peer environments. *Journal of Grid Computing*, 1(2):199–217, 2003.

[8] K.Kennedy, Jim Blythe, Sonal Jain, Ewa Deelman, Yolanda Gil, Karan Vahi, and Anirban Mandal. Task scheduling strategies for workflow-based applications in grids. In *Proceedings of IEEE International Symposium on Cluster Computing and the Grid 2005 (CCGrid 2005)*, Cardiff, UK, May 2005. IEEE Computer Society Press.

[9] 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):3045–3054, 2004.