Flexible and efficient workflow deployement of data-intensive applications on grids with MOTEUR

Tristan Glatard[1][2]
Johan Montagnat[1]
Diane Lingrand[1]
Xavier Pennec[2]

Abstract

Workflows offer a powerful way to describe and deploy applications on grid infrastructures. Many workflow management systems have been proposed but there is still a lack of a system that would allow both a simple description of the dataflow of the application and an efficient execution on a grid platform. In this paper, we study the requirements of such a system, underlining the need for well-defined data composition strategies on the one hand and for a fully parallel execution on the other hand. As combining those features is not straight forward, we then propose algorithms to do so and we describe the design and implementation of MOTEUR, a workflow engine that fulfills those requirements. Performance results and overhead quantification are shown to evaluate MOTEUR with respect to existing comparable workflow systems on a production grid.

1 Introduction

As a consequence of the tremendous research effort carried out by the international community these last years and the emergence of standards, grid middlewares have reached a maturity level such that large grid infrastructures where deployed (EGEE[3], OSG[4], NAREGI[5]) and sustained computing production was demonstrated for the benefit of many industrial and scientific applications [Montagnat et al., 2004].

Considering the considerable amount of sequential, non grid-specific algorithms that have been produced for various data processing tasks, grid computing is very promising for performing complex computations involving many computation tasks (codes parallelism) and processing large amounts of data (data parallelism). Indeed, beyond specific parallel codes conceived for exploiting an internal parallelism, grids are adapted to the massive execution of different tasks or the re-execution of a sequential code on different data sets which are needed for many scientific applications. In both cases, temporal and data dependencies may limit the parallelism that can be achieved.

Yet, current middlewares expose rather low level interfaces to the application developers and enacting an application on a grid often requires a significant work involving computer and grid experts. Workflow systems [Yu and Buyya, 2005] offer a simple way to gridify an application by providing an explicit description of the dependencies between its components. They stands as an abstraction layer between the low level grid middlewares and the user's application and have been successfully used to gridify several

[1]FRENCH NATIONAL CENTER FOR SCIENTIFIC RESEARCH, I3S LABORATORY, RAINBOW TEAM

[2]FRENCH NATIONAL CENTER FOR COMPUTER SCIENCE AND AUTOMATIC RESEARCH (INRIA)

[3]Enabling Grids for E-SciencE EU IST project, http://www.eu-egee.org

[4]Open Science Grid, http://www.opensciencegrid.org

[5]Japan National Research Grid Initiative (NAREGI), http://www.naregi.org

applications. However, those systems still focus either on execution performance [Deelman et al., 2003] or on expressiveness and user-friendly description of the application [Oinn et al., 2004].

The goal of this paper is to design a workflow system combining an expressive and user-friendly workflow description with an efficient execution of data-intensive scientific applications. We study techniques to simplify the workflow description from a user point of view (section 2) and performance for the workflow execution (section 3) on a grid. Even if some of those methods have already emerged, combining them is not straightforward and has never been done before in classical workflow systems (section 4). We propose the design and implementation of MOTEUR, a novel workflow engine that combines expressive workflows description with efficient grid execution in section 5.

All along this paper, we distinguish two grid middleware approaches, that influence application development. To handle user processing requests, two main strategies have indeed been proposed and implemented in grid middlewares:

In the *task based* strategy, also referred to as *global computing*, users define computing tasks to be executed. Any executable code may be requested by specifying the executable code file, input data files, and command line parameters to invoke the execution. The task based strategy, implemented in GLOBUS [Foster, 2005], LCG2[6] or gLite[7] middlewares for instance, has already been used for decades in batch computing. It makes the use of non grid-specific code very simple, provided that the user has a knowledge of the syntax to invoke each task.

The *service based* strategy, also referred to as *meta computing*, consists in wrapping application codes into standard interfaces. Such services are seen as black boxes from the middleware for which only the invocation interface is known. Various interfaces such as Web Services [World Wide Web Consortium, 2001] or gridRPC [Nakada et al., 2005] have been standardized. The services paradigm has been widely adopted by middleware developers for the high level of flexibility that it offers (OGSA [Foster et al., 2002]). However, this approach is less common for application codes as it requires all codes to be instrumented with the common service interface. To ease the migration of existing codes, generic service wrappers have been proposed [Kacsuk et al., 2004, Glatard et al., 2006a].

Task based and service based grid computing strategies are very different. This difference is even more visible when constructing application workflows. Those strategies differ regarding the description as well as the execution of the application. In the two following sections, we will compare them and discuss their respective strengths and weaknesses.

[6]LCG middleware, http://lcg.web.cern.ch

[7]gLite middleware, http://www.gLite.org

2 Data-intensive description

In this section, we focus on the expressiveness and user-friendliness of the workflow description provided by the task and service based paradigms.

2.1 Workflow description

An application workflow can be represented through a directed graph of *processors* (graph nodes) representing processings and dependencies (graph arrows) constraining their order of invocation (see figure 1).

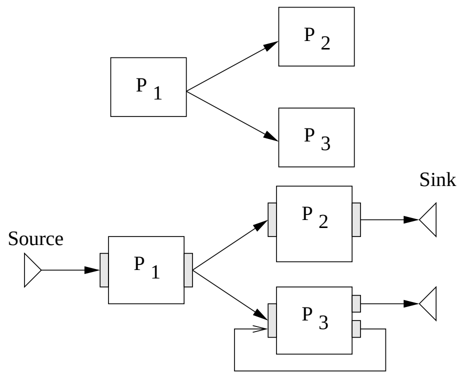

Fig. 1: Simple workflow example: task based (top) and service based (bottom)

In the task based approach, the description of a task, or computation job, encompasses both the processing (binary code and command line parameters) and the data (static declaration). Workflow processors directly represent computing tasks and the dependencies between them are precedence constraints. The user is responsible for providing the binary code to be executed and for writing down the precise invocation command line. All computations to be performed are statically described in the graph.

Conversely, in the service based approach, the application program is described separately from the data. The input data is treated as parameters (dynamic declaration), and the service appears to the end user as a black box hiding the code invocation. This difference in the handling of data (static or dynamic declaration) makes the application description far easier from a user point of view.

In a service based workflow, each processor represents an application component and arrows represent data dependencies. In addition to the processors and the data arrows, a service based workflow representation requires a number of input and output ports attached to each processor. Input ports are holding references to the data to be processed and output ports contain references to the data produced by a service. The oriented arrows connect output ports to input ports and represent data channels. Two special processor nodes are defined: *data sources* are processors without input ports (they produce data to feed the workflow) and *data sinks* are processors without output ports (they collect the data produced).

2.2 Dynamic data sets and looping

Task-based and service-based workflows differ in depth in their handling of data. The non-static nature of data description in the service-based approach enables dynamic extension of the data sets to be processed: a workflow can be defined and executed although the complete input data sets are not known in advance. It will be dynamically fed in as new data is being produced by sources. Indeed, it is common in scientific applications that data acquisition is an heavy-weight process and that data segments are being progressively produced. Some workflows may even act on the data production source itself, stopping data production once computations have shown that sufficient inputs are available to produce meaningful results. Finally, this dynamicity is required when the input data is the result of a data base query whose response size is not known in advance.

A significant difference between the task and service approaches coming from the ability of the latter one to deal with dynamic data sets is that there may exist loops in a service based workflow given that an input port can collect data from different sources as illustrated in the bottom of figure 1. This kind of workflow pattern is common for optimization algorithms: it corresponds to an optimization loop converging after a number of iterations determined at the execution time from a computed criterion. In this case, the output of processor P_1 would correspond to the initial value of this criterion. P_3 produces its result on one of its two output ports, whether the computation has to be iterated one more time or not. On the contrary, there cannot be a loop in the graph of a task based workflow as this would mean that a processor input is depending on one of its output. Executing an optimization loop would not be possible, as the number of iterations is dynamically determined and thus cannot be statically described.

2.3 Data-intensive applications

From a user point of view, the main difference between the task based and the service based approaches appears when considering the reexecution of the same application workflow over different input data sets, as it is commonly done for instance by so-called embarrassingly parallel applications. In a task based workflow, a computation task is defined by a single input data set and a single processing. Executing the same processing over two different data sets results in the description of two independent tasks. This approach enforces the replication of the execution graph for every input data to process.

To get closer to the service-based approach, a simple extension to the task based approach is to propose parametric data tasks descriptions where a generic task can be described for a set of input data, resulting in the execution of multiple jobs: one per input data. However, it is far from being commonly available in today's production middlewares and it is often treated at the application level. Moreover, parametric tasks cannot be used in a workflow where each task needs to be replicated for every input data set.

On the other hand, the service based approach easily accommodates with input data sets. Data sources are sequentially delivering input data but no additional complexity of the application description graph is needed. An example of the flexibility offered by the service-based approach, related to data sets processing, is the ability to define different data composition strategies over the input data of a service [Oinn et al., 2004]. When a service owns two input ports or more, a data composition strategy defines the composition rule for the data coming from all input ports pairwise. Those data composition strategies are studied in section 2.4.

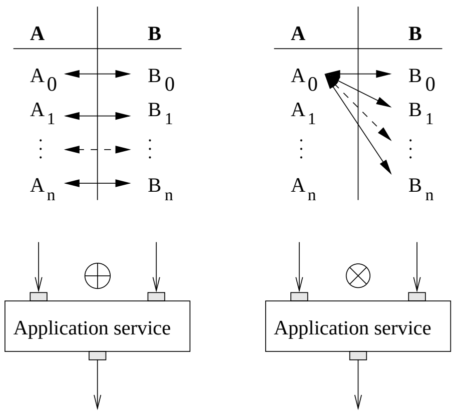

Fig. 2: Action of the *one-to-one* (left) and *all-to-all* (right) operators on the input data sets

2.4 Data composition strategies

Each service in a data-intensive workflow of services is receiving input data on its input ports. Depending on the desired service semantic, the user might envisage various input composition patterns between the different input ports.

2.4.1 Basic data composition patterns

Although not exhaustive, there are two main data composition patterns, very frequently encountered in scientific applications, that were first introduced in the Taverna workbench [Oinn et al., 2004]. They are illustrated in figure 2. Let $\mathbf{A} = \{\mathrm{A}_0, \mathrm{A}_1, \ldots, \mathrm{A}_n\}$ and $\mathbf{B} = \{\mathrm{B}_0, \mathrm{B}_1, \ldots, \mathrm{B}_m\}$ be two input data sets.

The *one-to-one* composition pattern (left of figure 2) is the most common. It consists in processing two input data sets pairwise in their order of arrival. This is the classical case where an algorithm needs to process every pair of input data independently. An example is a matrix addition operator: the sum of each pair of input matrices is computed and returned as a result. We will denote $\oplus$ the one-to-one composition operator. $\mathbf{A} \oplus \mathbf{B} = \{A_1 \oplus B_1, A_2 \oplus B_2, \ldots\}$ denotes the set of all outputs. For simplification, we will denote $A_1 \oplus B_1$ the result of processing the pair of input data (A_1, B_1) by some service. Usually, the two input data sets have the same size ($m = n$) when using the one-to-one operator, and the cardinality of the results set is $m = n$. If $m \neq n$, a semantics has to be defined.

The *all-to-all* composition pattern (right of figure 2) corresponds to the case where all inputs in one data set need to be processed with all inputs in the other data set. A common example is the case where all pieces of data in the first input set are to be processed with all parameter configurations defined in the second input set. We will denote $\otimes$ the all-to-all composition operator. The cardinality of $\mathbf{A} \otimes \mathbf{B} = \{A_1 \otimes B_1, A_1 \otimes B_2 \ldots A_1 \otimes B_m, A_2 \otimes B_1 \ldots A_2 \otimes B_m \ldots\ldots A_n \otimes B_1 \ldots A_n \otimes B_m\}$ is $m \times n$.

Note that other composition patterns with different semantics could be defined (*e.g. all-to-all-but-one* composition). However, they are more specific and consequently more rarely encountered. Combining the two operators introduced above enable very complex data composition patterns, as illustrated below.

2.4.2 Combining data composition patterns

As illustrated at the left of figure 3, the pairwise one-to-one and all-to-all operators can be combined to compose data patterns for services with an arbitrary number of input ports. In this case, the priority of these operators needs to be explicitly provided by the user. We are using parenthesis in our figures to display priorities explicitly. If the input data sets are

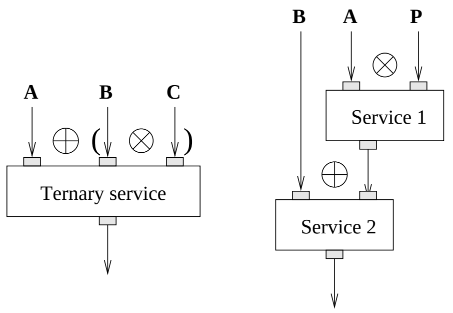

Fig. 3: Combining composition operators: multiple input service (left) and cascade of services (right)

$\mathbf{A} = \{A_0, A_1\}$, $\mathbf{B} = \{B_0, B_1\}$, and $\mathbf{C} = \{C_0, C_1, C_2\}$, the following data would be produced in this case:

$$\mathbf{A} \oplus (\mathbf{B} \otimes \mathbf{C}) = \left\{ \begin{array}{ll} A_0 \oplus (B_0 \otimes C_0), & A_1 \oplus (B_1 \otimes C_0), \\ A_0 \oplus (B_0 \otimes C_1), & A_1 \oplus (B_1 \otimes C_1), \\ A_0 \oplus (B_0 \otimes C_2), & A_1 \oplus (B_1 \otimes C_2) \end{array} \right\}$$

Successive services may also use various combinations of data composition operators as illustrated at the right of figure 3. The example given corresponds to a classical situation where an input data set $\mathbf{A} = \{A_0, A_1\}$, is processed by a first algorithm (using different parameter configurations $\mathbf{P} = \{P_0, P_1, P_2\}$), before being delivered to a second service for processing with a matching number of data $\mathbf{B} = \{B_0, B_1\}$. The output data set would be:

$$\mathbf{B} \oplus (\mathbf{A} \otimes \mathbf{P}) = \left\{ \begin{array}{ll} B_0 \oplus (A_0 \otimes P_0), & B_1 \oplus (A_1 \otimes P_0), \\ B_0 \oplus (A_0 \otimes P_1), & B_1 \oplus (A_1 \otimes P_1), \\ B_0 \oplus (A_0 \otimes P_2), & B_1 \oplus (A_1 \otimes P_2) \end{array} \right\} \quad (1)$$

As it can be seen, composition operators are a powerful tool for data-intensive application developers who can represent complex data flows in a very compact format. Although the one-to-one operator preserves the input data sets cardinality, the all-to-all operator may lead to drastic increases in the number of data to be processed. It makes the task replication problem associated to the task-based workflows combinatorial: an all-to-all composition produces an enormous amount of tasks and chaining all-to-all patterns just makes the application workflow representation intractable even for a limited number (tens) of input data. Despite the availability of graphical tools to design application workflows, dealing with many input data quickly becomes impossible for users.

3 Efficient workflow execution

Nowadays, executing data-intensive scientific applications on a single platform is not always possible, even if the platform is a parallel one. First, from a user point of view, it is frequent to compose applications using codes coming from different institutes, having various requirements, which imposes the use of different platforms. Second, from a performance point of view, it is crucial to choose the right grid platform for the right service: a trade-off has to be found between large scale multi-users grids, providing high throughput and latencies and local systems with low latency but also low throughput [Silberstein et al., 2006, Glatard et al., 2006b].

3.1 Single interface

The service based approach is able to transparently deal with multiple execution platforms. Each service is called as a black box without knowledge of the underlying execution platform. Several services may execute on different platforms transparently at the application level, which is convenient when dealing with legacy codes. In the task based approach, the workflow engine requires a specific submission interface for each infrastructure.

The service based approach is also well suited for chaining the execution of different algorithms assembled to build an application. Indeed, the interface to each application component is clearly defined and the middleware can invoke each of them through a single protocol, regardless of their implementation.

3.2 Resources allocations

The service based approach is making grid application description easier than the task based one as discussed above. It is thus highly convenient from the end user point of view. However, in this approach, the control of jobs submissions is delegated to external services, making the optimization of the workflow execution much more difficult, whereas many elaborated scheduling strategies are developed for task graphs. Many solutions have been proposed in the task-based paradigm to optimize the scheduling of an application in distributed environments [Casanova et al., 2000]. Concerning workflow-based applications, authors propose in [Chen and Yang, 2006] a framework to adapt fixed-time constraints to the highly variable case of grids. Works such as [Blythe et al., 2005, Malewicz et al., 2006] propose specific heuristics to optimize the resource allocation of a complete workflow. Even if it provides remarkable results, this kind of solutions is not directly applicable to the service-based approach: the services are black boxes isolating the workflow manager from the execution infrastructure. In this context, resources allocation is not under the direct control of the workflow scheduler. Hence, there is a strong need for precisely identifying performance optimization solutions that apply to service-based workflows.

Focusing on the service-based approach, nice developments such as the DIET middleware [Caron et al., 2002] and comparable approaches [Tanaka et al., 2003, Arnold et al., 2002] introduce specific strategies such as hierarchical scheduling. However, those works focus on middleware design and do not include any workflow management yet. As far as we know, such a deployment has only been done on experimental platforms [Cappello et al., 2005], and requires more investigation before being used on production infrastructures.

3.3 Exploiting parallelism

In the following sections, we study performance solutions that can be applied in service-based workflows. 3 kinds of parallelism have to be exploited to enable efficient execution.

3.3.1 Asynchronous services calls

To enable parallelism during the workflow execution, multiple application services have to be called concurrently. The calls made from the workflow enactor to these services need to be non-blocking for exploiting the potential parallelism. GridRPC services may be called asynchronously as defined in the standard [Nakada et al., 2005]. Web Services also theoretically enable asynchronous calls. However, the vast majority of existing web service implementations do not cover the whole standard and none of the major implementations [Van Engelen and Gallivan, 2002, Irani and Bashna, 2002] do provide any asynchronous service calls for now. As a consequence, asynchronous calls to web services need to be implemented at the workflow enactor level, by spawning independent system threads for each invocation.

3.3.2 Workflow parallelism

Given that asynchronous calls are possible, the first level of parallelism that can be exploited is the intrinsic workflow parallelism depending on the graph topology. For instance if we consider the simple example presented in figure 1, processors P_2 and P_3 may be executed in parallel. This optimization is trivial and implemented in all the workflow managers.

3.3.3 Data parallelism

When considering data-intensive applications, several input data sets are to be processed using a given workflow. Benefiting from the large number of resources available in a grid, workflow services can be instantiated as several computing tasks running on different hardware resources and processing different input data in parallel.

Data parallelism denotes that a service is able to process several data fragments simultaneously with a minimal performance loss. This capability involves the processing of independent data on different computing resources. Consider the simple workflow made of 3 services and represented on top of figure 1. Suppose that we want to execute this workflow on 3 independent input data sets D_0, D_1 and D_2. The data parallel execution diagram of this workflow is represented on the left of figure 4. On this kind of diagram, the abscissa axis represents time. When a data set D_i appears on a row corresponding to a processor P_j, it means that D_i is being processed by P_j at the current

P_3	X	D_0 D_1 D_2
P_2	X	D_0 D_1 D_2
P_1	D_0 D_1 D_2	X

P_3	X	D_0	D_1	D_2
P_2	X	D_0	D_1	D_2
P_1	D_0	D_1	D_2	X

Fig. 4: Data (left) and service (right) parallel execution diagrams of the workflow of figure 1

time. To facilitate legibility, we represented with the D_i notation the piece of data resulting from the processing of the initial input data set D_i all along the workflow. For example, in the diagram of figure 4, it is implicit that on the P_2 service row, D_0 actually denotes the data resulting from the processing of the input data set D_0 by P_1. Moreover, on those diagrams we made the assumption that the processing time of every data set by every service is constant, thus leading to cells of equal widths. Data parallelism occurs when different data sets appear on a single square of the diagram whereas intrinsic workflow parallelism occurs when the same data set appears many times on different cells of the same column. Crosses represent idle cycles.

3.3.4 Services parallelism

Input data sets are likely to be independent from each other, for instance when a single workflow is iterated in parallel on many input data sets.

Services parallelism denotes that the processing of two different data sets by two different services are totally independent. This pipelining model, very successfully exploited inside CPUs, can be adapted to sequential parts of service-based workflows. Consider again the simple workflow represented in left of figure 1, to be executed on the 3 independent input data sets D_0, D_1 and D_2. The right of figure 4 presents a service parallel execution diagram of this workflow. Service parallelism occurs when different data sets appear on different cells of the same column. We here supposed that a given service can only process a single data set at a given time (data parallelism is disabled). Here again, service parallelism is of major importance to optimize service-based workflows.

3.3.5 Data synchronization barriers

A particular kind of processors are algorithms that need to take into account the whole input data set in their processing rather than processing each input one by one. This is the case for many statistical operations computed on the data, such as the computation of a mean or a standard deviation over the produced results for instance. Such processors are referred to as *synchronization* processors as they represent real synchronization barriers, waiting for all input data to be processed before being executed. Data synchronization barriers are of course a limitation to services parallelism. In this case, this level of parallelism cannot be exploited because the input data sets are dependent from each other.

4 State of the art of grid-enabled scientific workflow systems

A detailed review of the workflow systems is available in [Yu and Buyya, 2005]. The main scientific service-based workflow managers are the Kepler system [Ludäscher et al., 2005], the Taverna workbench [Oinn et al., 2004] and the Triana workflow manager [Taylor et al., 2005].

Kepler targets many application areas from gene promoter identification to mineral classification. It can orchestrate standard Web-Services linked with both data and control dependencies and implements various execution strategies. Taverna, designed in the context of the myGrid e-Science UK project[8], was initially developed for the bioinformatics community and is able to enact Web-Services and other components such as Soaplab services [Senger et al., 2003] and Biomoby ones. It implements high level tools for the workflow description such as the Feta semantic discovery engine [Lord et al., 2005]. Tri-

[8]http://mygrid.org.uk

ana [Taylor et al., 2005], from the GridLab project[9], is decentralized and distributes several control units over different computing resources. It has been applied to various scientific fields, such as gravitational waves searching [Churches et al., 2003] and galaxy visualization [Taylor et al., 2003].

The emblematic task-based workflow manager is the Directed Acyclic Graph Manager (DAGMan)[10]. This system basically allows the description of precedence constraints between Condor jobs. The Pegasus [Deelman et al., 2003] system is built on top of it and implements scheduling optimizations. Close to DAGMan too, the P-GRADE portal [Kacsuk and Sipos, 2005] offers a user-friendly graphical web interface to it.

4.1 Data composition

As discussed in section 2, data composition is only available in service-based workflow systems.

Taverna [Oinn et al., 2004].
The one-to-one and the all-to-all data composition operators were first introduced and implemented in the Taverna workflow manager. They are part of the underlying Scufl workflow description language. In this context, they are known as the *dot product* and *cross product iteration strategies* respectively. The strategy of Taverna for dealing with input sets of different sizes in a one-to-one composition is to produce the $\min(m, n)$ first results only. However, the semantics adopted by Taverna when dealing with a composition of operators as illustrated in figure 3 is not fully satisfying as will be discussed in section 5.3.

Kepler [Ludäscher et al., 2005] and Triana [Taylor et al., 2005].
The Kepler and the Triana workflow managers only implement the one-to-one composition operator. This operator is implicit for all data composition inside the workflow and it cannot be explicitly specified by the user.

We could implement an all-to-all strategy in Kepler by defining specific actors but this is far from being straight forward. Kepler actors are blocking when reading on empty input ports. The case where two different input data sets have a different size (common in the all-to-all composition operator) is not really taken into account. Similar work can be achieved in Triana using the various *data stream* tools provided. However, in both cases, the all-to-all semantics is not handled at the level of the workflow engine. It needs to be implemented inside the application workflow.

4.2 Parallelism exploitation

In task-based workflow systems, the 3 kinds of parallelism described in the previous section are equivalent and resume to the workflow one. Indeed, in this approach, data as well as service parallelism are explicitly expanded in the workflow graph description.

Concerning the service-based approach, workflow parallelism is available in Taverna, Kepler and Triana. Data parallelism is present in Taverna, even if it is currently limited to 10 parallel threads. Service parallelism is not available in this system yet, although it is planned for the coming Taverna II.

Kepler implements the service parallelism within its PN director. In this execution framework, each processor (actor in the Kepler vocabulary) is executed on a dedicated thread. Strategies have been developed to cope with nested collections [McPhillips and Bowers, 2005] and to retrieve data provenance [Bowers et al., 2006] in service parallel workflows but data parallelism is not present.

It appears that no existing workflow system implements all the features that we described in sections 2 and 3 to build an expressive and efficient workflow manager, as we target to do so. Combining them is not a trivial issue: in particular, fully exploiting parallelism leads to a complete disordering of the data segments among the workflow, so that applying data composition operators requires some developments. In the next section, we thus propose a design and an implementation of MOTEUR, a prototype workflow engine that integrates those characteristics.

[9] http://www.gridlab.org

[10] Condor DAGMan, http://www.cs.wisc.edu/condor/dagman/

5 The MOTEUR workflow engine

None of the existing workflow manager implementations mentioned in section 4 does combine data composition strategies (simple and compact framework to describe scientific applications in a flexible service oriented approach) and fully parallel execution (efficient enactment of data-intensive applications on grids). Our hoMe-made OpTimisEd scUfl enactoR (MOTEUR) prototype was designed to take advantage of both, thus providing a flexible and expressive workflow description framework to the user and transparently exploiting parallel grid resources. It was implemented in Java and is available for downloading under CeCILL Public License (a GPL-compatible license) at http://www.i3s.unice.fr/~glatard. A basic portal is also available.

5.1 Standards involved

As it is the most elaborated existing solution in terms of data composition, we started from the Taverna workbench. We thus adopted the Simple Concept Unified Flow Language (Scufl) used by this system as the workflow description language. This language is convenient for describing data flows and widely used in the e-Science community. Apart from describing the data links between the services, the Scufl language also defines so-called coordination constraints. A coordination constraint is a control link which enforces an order of execution between two services even if there is no data dependency between them. We use coordination constraints to identify services that require data synchronization.

We developed a simple XML-based language to be able to describe input data sets. This language aims at providing a file format to save and store the input data set in order to be able to re-execute workflows on the same data set. It simply describes each item of the different inputs of the workflow.

MOTEUR is interfaced to both Web Services and GridRPC instrumented application codes.

5.2 Enactor implementation

The central enactor periodically queries each processor of the service graph to determine whether it is ready to be enacted. The corresponding node then computes the data sets resulting from the application of its data composition strategy on its ports containing the data segments coming from its predecessors. If a given data set has not been previously computed, a dedicated thread is then started for the computation. When the computation is finished, the service pushes the results to all of its successors ports.

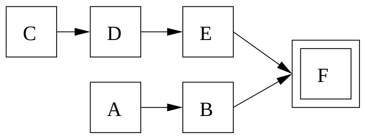

Fig. 5: Implementation of the synchronization barrier: F starts when A, B, C, D and E are inactive and have run at least once

Implementing synchronization barriers in a data and service parallel workflow requires an in-depth inspection of the service tree. Indeed, if we for example consider the workflow depicted on figure 5, where service F synchronizes the data produced by both B and E, it must be guaranteed that B and E have produced *all* their data segments before F starts. A necessary and sufficient condition to ensure this is that *all the ancestors of F are inactive and have processed a non-null number of data sets.* We used this condition to implement synchronization barriers.

Handling the data composition strategies presented in section 2.3 in a service and data parallel workflow is not straightforward. Indeed, the data composition result for a given service cannot be computed once for all the data sets. Indeed, due to service parallelism, the input data segments of a service are received one by one in input ports. The data composition thus has to be recomputed each time a new data segment is coming and the service has to record the data sets it has already computed. Moreover, data provenance has to be properly tracked in order to compute *one-to-one* composition operators. Indeed, due to data parallelism, a data is able to overtake another one during the processing and this disordering could lead to a causality problem, as we exemplified in [Glatard et al., 2005]. Besides, due to service parallelism, several data sets are processed concurrently and one cannot number all the produced data once computations completed. We have implemented a data provenance strategy to sort out the causality problems that may occur. Attached to each processed data segment is a history tree containing all the intermediate results computed to process it. This tree unambiguously identifies the data as detailed in the next subsection and in [Montagnat et al., 2006].

5.3 DATA COMPOSITION ALGORITHM

The semantics of combining data composition operators is not straight forward. Taverna provides the most advanced data composition techniques. Yet, we argue that the semantics adopted is not intuitive for the end user. Consider the simple example described in right of figure 3. The priority on the data composition is implicit in the workflow. There is no user control on it. In this case, Taverna will produce:

$$\mathbf{B} \oplus_{\text{Taverna}} (\mathbf{A} \otimes \mathbf{P}) = \{ \; B_0 \oplus (A_0 \otimes P_0), \quad B_1 \oplus (A_1 \otimes P_0) \; \} \tag{2}$$

More data will be produced at the output of the Service1 (namely, $A_0 \otimes P_1, A_1 \otimes P_1, A_0 \otimes P_2, A_1 \otimes P_2$) but the truncation semantics of the one-to-one operator will apply in the second service and only two output data will be produced.

This semantics differs from the one that we consider and that is illustrated in equation 1. Given that two correlated input data sets $\mathbf{A}$ and $\mathbf{B}$, with the same size, are provided, the user can expect that the data A_i will always be analyzed with the correlated data B_i, regardless of the algorithm parameters P_j considered: this is the case in equation 1 where A_i is always consistently combined with B_i.

In order to implement a clear and intuitive semantics for such data compositions, we propose a new algorithm. To formalize this approach, we need to take into account the complete data flows to be processed in the application workflow. Let us consider the very general case, common in scientific applications, where the user needs to independently process sets of input data $\mathbf{A}, \mathbf{B}, \ldots$ that are divided into *data groups*. A group is a set of input data tuples that defines a relation between data coming from different sets. For instance:

$$\mathrm{G} = \{(A_0, B_0, C_0), (A_1, B_1, C_1), (A_2, B_2, C_2)\}$$
$$\mathrm{H} = \{(A_4, B_0), (A_1, B_2), (A_2, B_5), (A_6, B_6)\}$$

are two groups establishing a relation between 3 data triplets and 4 data pairs respectively. The relations between input data depend on the application and can only be specified by the user. However, we will see that this definition can be explicit (as illustrated above) or implicit, just considering the workflow topology and the order in which input data segments are received by the workflow manager.

5.3.1 Data composition operator semantics

We consider that the one-to-one composition operator does only make sense when processing related data. Therefore, only data connected by a group should be considered for processing by any service. When considering a service directly connected to input data sets, determining relations between data is straight forward. However, when considering a complete application workflow such as the one illustrated in figure 7, other services need to determine which of their input data segments are correlated. The one-to-one composition operator does introduce the need for the algorithm described below.

Conversely, the all-to-all operator does not rely on any pre-determined relation between input data. Any number of inputs can be combined, with very different meaning (such as data to process and algorithm parameters). Each data received as input yields to one or more invocations of the service for processing.

5.3.2 Algorithm

The directed data graph is constructed from the roots (workflow inputs) to the leafs (workflow outputs) by applying the two following simple rules implementing the semantics of the one-to-one and the all-to-all operators respectively:

1. Two data segments are always combined in an all-to-all operation.
2. Two data (graph nodes) are combined in a one-to-one operation **if and only if** there exists a common ancestor to both data in the data graph.

To implement it, MOTEUR dynamically resolves the data combination problem by applying the following algorithm. We name *orphan* data, input data that have no group parent.

1. Build the directed graph of the data sets to be processed.
2. Add data groups to this graph.
3. Initialize the directed acyclic data graph:
   (a) Create root nodes for each group instance G_i and add a child node for each related data.
   (b) Create root nodes for each orphan data.
4. Start the execution of the workflow.
5. For each tuple of data to be processed:
   (a) Update the data graph by applying the two rules corresponding to the one-to-one and the all-to-all operators.
   (b) Loop until there are no more data available for processing in the workflow graph.

To implement this strategy, MOTEUR needs to keep representations of:

- the topology of the services workflow;
- the graph of data;
- and the list of input data that have been processed by each service.

Indeed, the data graph is dynamically updated during the execution. When a new data is produced, its combination with all previously produced data is studied. In particular in an all-to-all composition pattern, a new input data needs to be combined with all previously computed data. It potentially trigger several services invocation. The history of previous computations is thus needed to determine the exhaustive list of data to produce.

The graphs of data also ensures a full traceability of the data processed by the workflow manager: for each data node, the parents and children of the data can be determined. Besides, it provides a mean to unambiguously identify each data produced. This becomes mandatory when considering parallel execution of the workflow introduced in section 3.

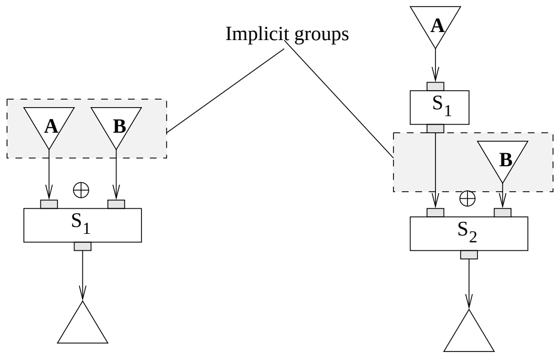

Fig. 6: Implicit groups definition.

5.3.3 Implicit combinations

The algorithm proposed aims at providing a strict semantics to the combination of data composition operators, while providing intuitive data manipulation for the users. Data groups have been introduced to clarify the semantics of the one-to-one operator. However, it is very common that users are writing workflows without explicitly specifying pairwise relations between the data. The order in which data segments are declared or send to the workflow inputs are rather used as an implicit relation.

To ease the workflow generation by the user, groups can be implicitly generated when they are not explicitly specified by the user. Figure 6 illustrates two different cases. On the left side, the reason for generating an implicit group is straight forward: two input data sets are being processed through a one-to-one service. But there may be more indirect cases such as the one illustrated on the right side of the figure. The systematic rule that can be applied is to create an implicit group for each *one-to-one* operator whose input data segments are orphans. For example, in the case illustrated in left of figure 6, the input datasets **A** and **B** are orphans and bound *one-to-one* by the S_1 service. An implicit group is therefore created between **A** and **B**. In the case illustrated in the right side of figure 6, the implicit group will be created between the two inputs of service S_2. There will therefore be an implicit grouping relation between each output of the first service $S_1(A_i)$ and B_i.

Workflow engines	Data composition		Parallelism		
	one-to-one	all-to-all	Workflow	Data	Service
Taverna	X	X	X	/	O
Kepler	X	O	X	O	X
Triana	X	O	X	O	X
MOTEUR	X	X	X	X	X

Table 1: Comparison of the main service-based workflow managers. X: present; /: limited; O: absent

The implicit groups are created statically by analyzing the workflow topology and the input data sets before starting the execution of the workflow.

5.4 MOTEUR EVALUATION

In table 1, the characteristics of the main service-based workflow managers are compared to MOTEUR, considering the data composition operators and the levels of parallelism implemented. It provides a qualitative evaluation of the prototype we developed. MOTEUR is the only workflow manager that implements the two basic data composition operators and the 3 levels of parallelism at the same time. The description of data-intensive workflows is made flexible due to the data composition operators and, on the other hand, the execution is made much more performant, as quantified in the following sub sections, that consider a real data-intensive application.

5.4.1 Evaluation on a real application

We evaluated MOTEUR on a real data-intensive application, which aims at assessing the accuracy of medical imaging algorithms. It is based on a statistical procedure which computes accurate results only if enough input data is available. This application is very scalable: the larger the input data sets, the more accurate it is. Hundreds of input images are typically needed. The workflow of this application is represented on figure 7. Each application service corresponds either to one of the algorithms to assess, or to related services for manipulating input/output data formats for these algorithms. The double squared last service of the workflow is the data synchronization barrier corresponding to the statistical assessment procedure.

The application makes an intensive usage of the data composition operators as illustrated in figure 7. Despite the complexity of the application (hundreds to thousands of tasks are needed for a full run), its description remains very compact. The data composition strategies allow the user to easily change those input data sets without modifying the workflow.

5.4.2 Performance results

We benchmarked this application on a Pentium IV, 2.4GHz, 512MB RAM running Linux 2.6.5. The execution of the workflow on a single input data set was 800 seconds. The application was tested on 126 input data segments in sources A and B. The sequential time on this data set would thus be 28 hours.

The platform used for this evaluation is the EGEE production grid with the LCG2 middleware. This infrastructure is characterized by its high throughput (3000 processors are available for our Virtual Organization) but also by its high latency (more than 5 minutes) due to its large scale and multi-users nature.

On this platform, we obtained a 13.2 speed-up on our application using MOTEUR. The existing workflow solution that provides the required level of flexibility to describe our application (*i.e* the Taverna workbench) does not implement service parallelism (see section 4). If we disable service parallelism in MOTEUR, the speed-up of the application sinks to 7. Moreover, Taverna is currently limited to 10 parallel threads only. MOTEUR thus provides a speed-up higher than 1.9 with respect to the existing comparable workflow engines. Going further in a quantitative evaluation of MOTEUR would require to quantify the impact of each parallelism level on the makespan of the application. On a production grid infrastructure such as EGEE, this problem is not straight forward because the impact of a given level of parallelism depends on the variability of the execution platform. We are investigating solutions to address this problem through a probabilistic approach but this goes beyond the scope of this paper.

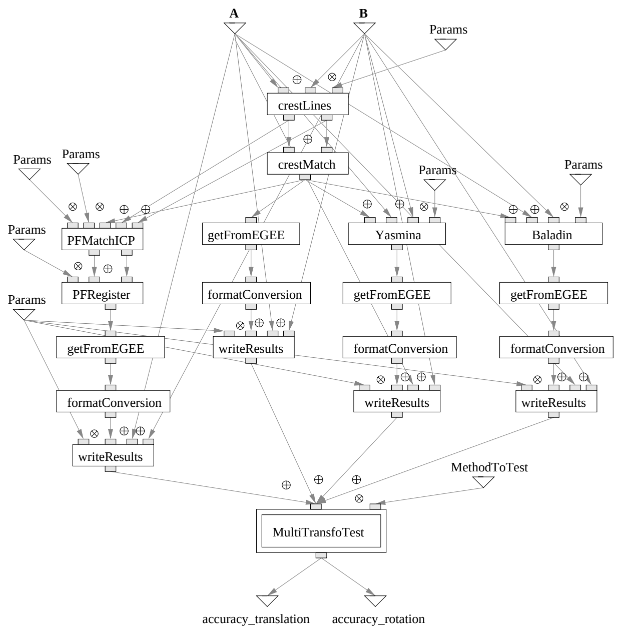

Fig. 7: Workflow of the application

5.4.3 MOTEUR overhead

Evaluating MOTEUR overhead on the execution is necessary to identify sources of performance drops. To quantify this overhead, we implemented a basic simulator that fakes the grid middleware and assigns fixed execution times to the jobs so that the middleware overhead is negligible.

Figure 8 displays MOTEUR overhead with respect to the number of input data segments on this application. For 12 data segments, MOTEUR introduces a latency of 15 seconds on the application. For each service invocation, a few seconds are lost due to submission and polling loops. Indeed, the LCG2 middleware does not provide any job status notification mechanism, and the application has to periodically poll it. A too short polling interval excessively stresses the middleware workload manager, hence intervals have to be at least a few seconds long. Those 15 seconds in the overhead could be reduced by optimizing looping delays but they are not of outmost importance in the whole execution time.

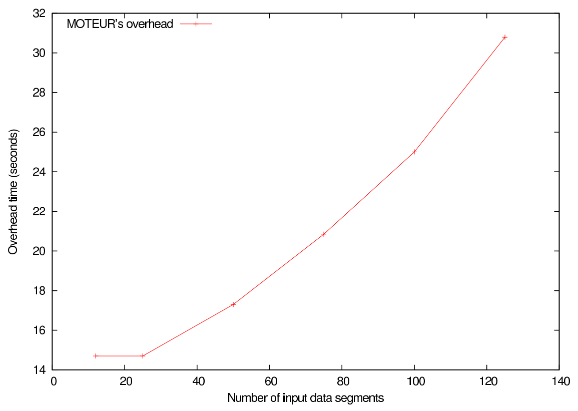

Fig. 8: MOTEUR overhead

MOTEUR overhead reaches 31 seconds when the number of input data segments is 126. If we linearly model the overhead w.r.t the number of input data segments, we obtain a penalty of 0.14 seconds per extra data segment. This overhead is negligible as compared to the application execution time, which is several hours long, and to the grid infrastructure access cost.

6 Conclusion

We presented a complete design of a workflow manager focusing on ease of use and performance. User-friendliness is achieved by the service based approach that particularly eases the iteration of a workflow on a whole data set by enabling data composition strategies. We studied the semantic of those strategies.

Performance is a difficult point in the service based paradigm and we described 3 levels of parallelism that have to be exploited in order to exploit the resources of grid infrastructures.

Because no existing workflow systems were providing the required features in data composition and parallelism, we implemented MOTEUR, a workflow engine targeting those application requirements. It includes new algorithms to handle data synchronization barriers and data composition strategies in a fully parallel environment.

We finally evaluated MOTEUR overhead on a medical imaging application run on the EGEE grid. Even when dealing with hundreds of data sets, MOTEUR overhead remains around 30 seconds, which is negligible for most of the data-intensive applications running on a production grid infrastructure. On this application, MOTEUR provides a 1.9 speed-up with respect to the workflow managers that offer the same level of flexibility in the workflow description.

Acknowledgment

This work is partially funded by the French research program "ACI-Masse de données", AGIR project. We are grateful to the EGEE European project for providing the grid infrastructure and assistance.

References

[Arnold et al., 2002] Arnold, D., Agrawal, S., Blackford, S., Dongarra, J., Miller, M., Seymour, K., Sagi, K., Shi, Z., and Vadhiyar, S. (2002). Users Guide to NetSolve V1.4.1. Technical Report ICL-UT-02-05, University of Tennessee, Knoxville.

[Blythe et al., 2005] Blythe, J., Jain, S., Deelman, E., Gil, Y., Vahi, K., Mandal, A., and Kennedy, K. (2005). Task Scheduling Strategies for Workflow-based Applications in Grids. In *CCGrid*, Cardiff, UK.

[Bowers et al., 2006] Bowers, S., McPhillips, T., Ludäscher, B., Cohen, S., and Davidson, S. (2006). A Model for User-Oriented Data Provenance in Pipelined Scientific Workflows. In *International Provenance and Annotation Workshop (IPAW)*, LNCS.

[Cappello et al., 2005] Cappello, F., Desprez, F., Dayde, M., Jeannot, E., Jegou, Y., Lanteri, S., Melab, N., Namyst, R., Vicat-Blanc Primet, P., Richard, O., Caron, E., Leduc, J., and Mornet, G. (2005). Grid'5000: A Large Scale, Reconfigurable, Controlable and Monitorable Grid Platform. In *6th IEEE/ACM International Workshop on Grid Computing (Grid'2005)*, Seattle, Washington, USA.

[Caron et al., 2002] Caron, E., Desprez, F., Lombard, F., Nicod, J.-M., Quinson, M., and Suter, F. (2002). A Scalable Approach to Network Enabled Servers. In *8th International EuroPar Conference*, volume 2400 of *LNCS*, pages 907–910, Paderborn, Germany. Springer-Verlag.

[Casanova et al., 2000] Casanova, H., Legrand, A., Zagorodnov, D., and Berman, F. (2000). Heuristics for Scheduling Parameter Sweep Applications in Grid Environments. In *9th Heterogeneous Computing Workshop (HCW)*, pages 349–363, Cancun.

[Chen and Yang, 2006] Chen, J. and Yang, Y. (2006). Multiple States based Temporal Consistency for Dynamic Verification of Fixed-time Constraints in Grid Workflow Systems. *Concurrency and Computation: Practice & Experience*.

[Churches et al., 2003] Churches, D., Sathyaprakash, B. S., Shields, M., Taylor, I., and Wand, I. (2003). A Parallel Implementation of the Inspiral Search Algorithm using Triana. In *Proceedings of the UK e-Science All Hands Meeting*, Nottingham, UK.

[Deelman et al., 2003] Deelman, E., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Vahi, K., Blackburn, K., Lazzarini, A., Arbree, A., Cavanaugh, R., and Koranda, S. (2003). Mapping Abstract Complex Workflows onto Grid Environments. *Journal of Grid Computing (JGC)*, 1(1):9–23.

[Foster, 2005] Foster, I. (2005). Globus Toolkit Version 4: Software for Service-Oriented Systems. In *International Conference on Network and Parallel Computing (IFIP)*, volume 3779, pages 2–13. Springer-Verlag LNCS.

[Foster et al., 2002] Foster, I., Kesselman, C., Nick, J., and Tuecke, S. (2002). The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Technical report, Open Grid Service Infrastructure WG, GGF.

[Glatard et al., 2006a] Glatard, T., Emsellem, D., and Montagnat, J. (2006a). Generic web service wrapper for efficient embedding of legacy codes in service-based workflows. In *Grid-Enabling Legacy Applications and Supporting End Users Workshop (GELA'06)*, Paris, France.

[Glatard et al., 2005] Glatard, T., Montagnat, J., and Pennec, X. (2005). Grid-enabled workflows for data intensive medical applications. In *18th IEEE International Symposium on Computer-Based Medical Systems (CBMS)*.

[Glatard et al., 2006b] Glatard, T., Montagnat, J., and Pennec, X. (2006b). An experimental comparison of Grid5000 clusters and the EGEE grid. In *Workshop on*

Experimental Grid testbeds for the assessment of large-scale distributed applications and tools (EXPGRID'06), Paris, France.

[Irani and Bashna, 2002] Irani, R. and Bashna, S. J. (2002). *AXIS: Next Generation Java SOAP*. Wrox Press.

[Kacsuk et al., 2004] Kacsuk, P., Goyeneche, A., Delaitre, T., Kiss, T., Farkas, Z., and Boczko, T. (2004). High-Level Grid Application Environment to Use Legacy Codes as OGSA Grid Services. In *Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing (GRID '04)*, pages 428–435, Washington, DC, USA. IEEE Computer Society.

[Kacsuk and Sipos, 2005] Kacsuk, P. and Sipos, G. (2005). Multi-Grid, Multi-User Workflows in the P-GRADE Grid Portal. *Journal of Grid Computing (JGC)*, 3(3-4):221 – 238.

[Lord et al., 2005] Lord, P., Alper, P., Wroe, C., and Goble, C. (2005). Feta: A light-weight architecture for user oriented semantic service discovery. In *European Semantic Web Conference*.

[Ludäscher et al., 2005] Ludäscher, B., Altintas, I., Berkley, C., Higgins, D., Jaeger, E., Jones, M., Lee, E. A., Tao, J., and Zhao, Y. (2005). Scientific Workflow Management and the Kepler System. *Concurrency and Computation: Practice & Experience*.

[Malewicz et al., 2006] Malewicz, G., Foster, I., Rosenberg, A. L., and Wilde, M. (2006). A tool for prioritizing DAGMan jobs and its evaluation. In *Proceedings of the 15th International Symposium on High Performance Distributed Computing (HPDC'06)*, pages 156–167, Paris, France.

[McPhillips and Bowers, 2005] McPhillips, T. and Bowers, S. (2005). An Approach for Pipelining Nested Collections in Scientific Workflows. *SIGMOD Record*, 35(3).

[Montagnat et al., 2004] Montagnat, J., Bellet, F., Benoit-Cattin, H., Breton, V., Brunie, L., Duque, H., Legré, Y., Magnin, I., Maigne, L., Miguet, S., Pierson, J.-M., Seitz, L., and Tweed, T. (2004). Medical images simulation, storage, and processing on the european datagrid testbed. *Journal of Grid Computing (JGC)*, 2(4):387–400.

[Montagnat et al., 2006] Montagnat, J., Glatard, T., and Lingrand, D. (2006). Data composition patterns in service-based workflows. In *Workshop on Workflows in Support of Large-Scale Science (WORKS'06)*, Paris, France.

[Nakada et al., 2005] Nakada, H., Matsuoka, S., Seymour, K., Dongarra, J., Lee, C., and Casanova, H. (2005). A GridRPC Model and API for End-User Applications. Technical report, Global Grid Forum (GGF).

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

[Senger et al., 2003] Senger, M., Rice, P., and Oinn, T. (2003). Soaplab - a unified Sesame door to analysis tool. In *UK e-Science All Hands Meeting*, pages 509–513, Nottingham.

[Silberstein et al., 2006] Silberstein, M., Geiger, D., Schuster, A., and Livny, M. (2006). Scheduling mixed workloads in multi-grids: the grid execution hierarchy. In *Proceedings of the 15th International Symposium on High Performance Distributed Computing (HPDC'06)*, pages 291–302, Paris, France.

[Tanaka et al., 2003] Tanaka, Y., Nakada, H., Sekiguchi, S., Suzumura, T., and Matsuoka, S. (2003). Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing. *Journal of Grid Computing (JGC)*, 1(1):41–51.

[Taylor et al., 2003] Taylor, I., Shields, M., Wand, I., and Philp, R. (2003). Grid Enabling Applications Using Triana. In *Workshop on Grid Applications and Programming Tools*. Held in Conjunction with GGF8.

[Taylor et al., 2005] Taylor, I., Wand, I., Shields, M., and Majithia, S. (2005). Distributed computing with Triana on the Grid. *Concurrency and Computation: Practice & Experience*, 17(1–18).

[Van Engelen and Gallivan, 2002] Van Engelen, R. A. and Gallivan, K. A. (2002). The gSOAP Toolkit for Web Services and Peer-to-Peer Computing Networks. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CC-GRID '02)*, page 128, Washington, DC, USA. IEEE Computer Society.

[World Wide Web Consortium, 2001] World Wide Web Consortium, W. (2001). Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl.

[Yu and Buyya, 2005] Yu, J. and Buyya, R. (2005). A taxonomy of scientific workflow systems for grid computing. *ACM SIGMOD records (SIGMOD)*, 34(3):44–49.