MOTEUR: a data-intensive service-based workflow manager

Tristan Glatard, Johan Montagnat, Xavier Pennec, David Emsellem, Diane Lingrand

LABORATOIRE

I3S

INFORMATIQUE, SIGNAUX ET SYSTÈMES
DE SOPHIA ANTIPOLIS
UMR 6070

MOTEUR: A DATA-INTENSIVE SERVICE-BASED WORKFLOW MANAGER

Tristan Glatard, Johan Montagnat, Xavier Pennec, David Emsellem, Diane Lingrand

Projet RAINBOW

Rapport de recherche
ISRN I3S/RR 2006-07 FR

Mars 2006

LABORATOIRE I3S: Les Algorithmes / Euclide B – 2000 route des Lucioles – B.P. 121 –
06903 Sophia-Antipolis Cedex, France – Tél. (33) 492 942 701 – Télécopie : (33) 492 942 898
http://www.i3s.unice.fr/I3S/FR/

RÉSUMÉ :

MOTEUR est un gestionnaire de flots optimisé pour traiter efficacement des applications manipulant de grandes masses de données sur des infrastructures de grille. MOTEUR exploite plusieurs niveaux de parallélisme et groupe les tâches à réaliser pour réduire le temps d'exécution des applications. De plus, MOTEUR utilise un Web Service générique d'encapsulation pour faciliter la réutilisation de codes développés sans prise en compte des spécificités des grilles de calcul. Dans ce rapport, nous présentons MOTEUR et les stratégies d'optimisation mises en oeuvre. Nous montrons comment nous avons défini une sémantique précise décrivant des flots de données complexes dans un format très compact. L'Architecture Orientée Services de MOTEUR est détaillées et la flexibilité de l'approche adoptée est démontrée. Des résultats sont donnés sur une application réelle de traitement d'images médicales qui s'appuie sur plusieurs infrastructures de grilles.

MOTS CLÉS :

flots de contrôle et de données, parallélisme, composition de données, groupage de tâches, AOS

ABSTRACT:

MOTEUR is a service-based workflow manager designed to efficiently process data-intensive applications on grid infrastructures. It exploits several levels of parallelism and can group services to reduce the workflow execution time. In addition, MOTEUR uses a generic web service wrapper to ease the use of legacy or non-service aware codes. In this report, we present MOTEUR and the optimization strategies implemented. We show how it is defining a precise data flows semantics to express complex data-intensive applications in a compact framework. MOTEUR' Service-Oriented Architecture is detailed, demonstrating the flexibility of the approach adopted. Results are given on a real application to medical images processing using two different grid infrastructures.

KEY WORDS :

workflows, data flows, parallelism, data composition, jobs grouping, SOA

MOTEUR: a data-intensive service-based workflow manager

Tristan Glatard[1], Johan Montagnat[1], Xavier Pennec[2], David Emsellem[1], Diane Lingrand[1]
[1]RAINBOW, I3S, CNRS. {glatard,johan,emsellem,lingrand}@i3s.unice.fr
[2]INRIA, Epidaure, Xavier.Pennec@sophia.inria.fr

March 11, 2006

Abstract

MOTEUR is a service-based workflow manager designed to efficiently process data-intensive applications on grid infrastructures. It exploits several levels of parallelism and can group services to reduce the workflow execution time. In addition, MOTEUR uses a generic web service wrapper to ease the use of legacy or non-service aware codes.

In this report, we present MOTEUR and the optimization strategies implemented. We show how it is defining a precise data flows semantics to express complex data-intensive applications in a compact framework. MOTEUR' Service-Oriented Architecture is detailed, demonstrating the flexibility of the approach adopted. Results are given on a real application to medical images processing using two different grid infrastructures.

Keywords: workflows, data flows, parallelism, data composition, jobs grouping, SOA.

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 [13], OSG [37], NAREGI [34]) and sustained computing production was demonstrated for the benefit of many industrial and scientific applications [32]. 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 system experts.

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).
- 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 applications. In both cases, temporal and data dependencies may limit the parallelism that can be achieved.

Assembling basic processing components is a powerful mean to develop new scientific applications. The reusability of data processing software components considerably reduces applications development time. In the image processing community for instance, it is common to chain basic processing operators and image interpretation algorithms to set up a complete image analysis procedure. Workflow description languages are generic tools providing a high-level representation for describing complex application control flows and the dependencies between application components. Workflow execution engines provide the ability to chain the application components execution while respecting causality and inter-components dependencies expressed within this abstract representation. Interfacing workflow managers with a grid infrastructure enables the efficient exploitation of code parallelism embedded in application workflows. It is a mean to transparently provide parallelism, without requiring specific code instrumentation nor introducing much load on the application developers side.

In addition, in many scientific areas applications exhibit a massive data-parallelism aspect that should be exploited. Taking the medical image analysis area as an example, many procedures require the processing of full image database:

- atlases construction;
- statistical and epidemiological studies;
- assessing image processing algorithms;
- validating medical procedures;
- . . .

In such data-centric applications, the workflow manager should not only efficiently handle control flows but also data flows which might well dominate the execution time.

Another important aspect for easing scientific applications migration towards grid infrastructures is to embed legacy codes into workflows. Indeed, many scientific codes represent tremendous development efforts. It is often undesirable (to take the risk of breaking the accepted validity of the code), or even impossible (when sources are not available for instance), to make any change to these codes.

In this report, we are summarizing our research activity in the area of supporting scientific application workflows. We introduce MOTEUR, a workflow engine interfaced with grid infrastructures, specifically designed to handle data-intensive applications by transparently exploiting both application code and data parallelism. We show MOTEUR was built in a modern Service Oriented Architecture framework to offer a maximum of flexibility. We provide experimental results for validating our approach on a medical image registration application.

2 State of the art and definitions

Workflow managers can be classified into two main categories: *control-centric* and *data-centric*. The control-centric managers, such as BPEL [1], are more focused on the description of complex application flows. They provide an exhaustive list of control structures such as branching, conditions and loop operators. They can describe very complex control composition patterns and some of them are comparable to small programming languages, including a graphical interface for designing the workflow and an interpretor for its execution. Conversely, data-centric managers usually provide a more limited panel of control structures and rather focus on the execution of heavyweight algorithms designed to process large amounts of data. The complex application logic is supposed to be embedded inside the basic application components. Although there is *a priori* no much contradiction in implementing a workflow manager that is both control and data-centric, the optimization of different managers for different needs often leads to several implementations.

Control-centric managers are commonly implemented to fulfill the e-business community needs. In this area, applications are often not so compute nor data-intensive and can be described in a high level language suitable for non-experts. Conversely, in the scientific area complex application codes, both compute and data intensive, are frequently available. The workflow description languages are not so rich but the execution engines are better taking into account execution efficiency and data transfer issues [46]. In the remaining of this paper, we will consider scientific workflow managers only.

2.1 Task-based and service-based approaches

To handle user processing requests, two main strategies have been proposed and implemented in grid middlewares:

1. 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 [14], LCG2 [28] or gLite [20] 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 exact syntax to invoke each computing task.

2. 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 [45] or gridRPC [33] have been standardized. The services paradigm has been widely adopted by middleware developers for the high level of flexibility that it offers (OGSA [15]). However, this approach is less common for application code as it requires all codes to be instrumented with the common service interface.

The task-based approach has been used for grid and batch computing for a very long time. To invoke a task-based job, a user needs to precisely know the command-line format of the executable and the meaning of parameters. It is not always the case when the user is not one of the developers. Input and output data are transmitted through files which have to be explicitly specified in the task description. Invoking a new execution of a same code on different data segments requires the rewriting of a new task description.

Conversely, in the service-based approach the actual code invocation is delegated to the service which is responsible for the correct handling of the invocation parameters. The service is a black box from the user side and to some extent, it can deal with the correct parameterization of the code to be executed. Services better decouple the computation and data handling parts. A service dynamically receives inputs as parameters. The inputs are not limited to files but may also be values of given types (number, text, etc). This decoupling of processing and data is particularly important when considering the processing of complete data sets rather than single data segments. Indeed, grid infrastructures are particularly well suited for data-intensive applications that require repeated processings of different data.

The service-based approach is more dynamic and flexible but it is usually used for accessing remote resources which do not necessarily benefit from grid computing capabilities. This is acceptable for most middleware services that are located and executed on a single server but application services that may require compute-intensive code execution and that are invoked concurrently in the context of the target applications, can easily overwhelm the computing capabilities of a single host. To overcome these limitations some approaches have been explored, such as submission services replacing the straight task submission [18] or generic services for wrapping any legacy code with a standard interface [26].

2.2 task-based and service-based workflows

An application workflow can intuitively be represented through a directed graph of *processors* (graph nodes) representing computation jobs and data dependencies (graph arrows) constraining the order of invocation of processors (see left of figure 1).

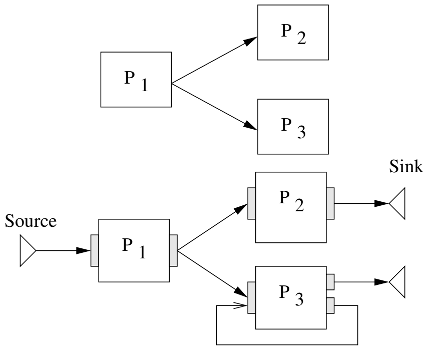


Figure 1: Simple workflow example. Task-based (top) and service-based (bottom).

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. 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 input data are treated as input parameters (dynamic declaration), and the service hides the code invocation. This difference in the handling of data (static or dynamic declaration) makes the application composition far easier from a user point of view, as detailed in section 2.3. The service-based approach is also naturally very 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.

In a service-based workflow, each processor is representing an application component, or service. 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. The oriented arrows are connecting output ports to input ports. Two special processor nodes are defined: *data sources* are processors without input ports (they are producing data to feed the workflow) and *data sinks* are processors without output ports (they are collecting data produced).

A significant difference between the service and task approaches of workflow composition 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 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. Conversely, there cannot be a loop in a workflow of tasks. If there were a loop, a data segment would depend on itself for its production. Hence, task-based workflows are always Directed and Acyclic Graphs (DAGs). Only in the case where the number of iterations is statically known, a loop may be expressed by unfolding it in the DAG. An emblematic task-based workflow manager is indeed called Directed Acyclic Graph Manager (DAGMan) [1]. Composing such optimization loop would not be possible, as the number of iterations is determined during the execution and thus cannot be statically described. Conversely, in a workflow of services, there may exist loops in the graph of services since it does not imply a circular dependency on the data. This enables the implementation of more complex control structures.

The service-based approach has been implemented in different workflow managers. The Kepler system [31] 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. The Taverna project [36], from the myGrid e-Science UK project[2] targets bioinformatics applications and is able to enact Web-Services and other components such as Soaplab services [40] and Biomoby ones. It implements high level tools for the workflow description such as the Feta semantic discovery engine [30]. Other workflow systems such as Triana [43], from the GridLab project[3], are decentralized and distribute several control units over different computing resources. This system implements both a parallel and a peer-to-peer distribution policies. It has been applied to various scientific fields, such as gravitational waves searching [11] and galaxy visualization [42].

2.3 Dynamic data sets

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 pro-

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

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

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

cessed: 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 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.

Most importantly, the dynamic extensibility of input data sets for each service in a workflow can also be used for defining different data composition strategies as introduced in section 3. The data composition patterns and their combinations offer a very powerful tool for describing complex data processing scenarios as needed in scientific applications. For the users, this means the ability to describe and schedule very complex processings in an elegant and compact framework.

2.4 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 has they represent real synchronization barriers, waiting for all input data to be processed before being executed.

2.5 Services flexibility

The service-based approach enables discovery mechanisms and dynamic invocation even for *a priori* unknown services. This provides a lot of flexibility both for the user (discovery of available data processing tools and their interface) and the middleware (automatic selection of services, alternatives services discovery, fault tolerance, etc).

In the service-based framework, the code reusability is also improved by the availability of a standard invocation interface. In particular, services are naturally well adapted to describe applications with a complex workflow, chaining different processings whose outputs are piped to the inputs of each other.

Another strength of the service-based approach is to easily deal with multiple execution platforms. Each service is called as a black box without knowledge of the underlying execution infrastructure. Several services may execute on different platforms transparently, which is convenient when dealing with legacy code, whereas in the task-based approach, a specific submission interface is needed for each infrastructure.

The flexibility and dynamic nature of services depicted above is usually very appreciated from the user point of view. Given that application services can be deployed at a very low development cost, there are number of advantages in favor of this approach.

2.6 Executing services

From middleware developers point of view, the execution of workflow of services is more difficult to optimize than the execution of workflows of tasks though. As mentioned above, the service is an intermediate layer between the user and the grid middleware. Thus, the user does not know nor see anything of the underlying infrastructure. Tuning of the jobs submission for a specific application is more difficult. In addition, data transfers can drastically impact some data-intensive application performances. Services are completely independent. Consequently, for chaining two different services P_0 and P_1, P_0's output data first needs to be returned to the user before being sent back as an input to P_1. *A priori*, this mechanism does not take advantage of grid data management systems. Therefore, some precautions need to be taken when considering service-based applications to ensure good application performances.

3 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.

3.1 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 [36]. 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. We will consider that only the $\min(m, n)$ first pieces of data are processed in this case.

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 will be illustrated below.

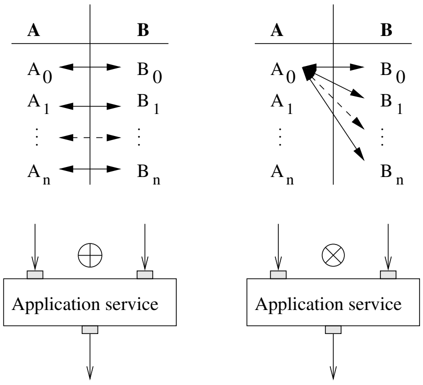


Figure 2: Action of the *one-to-one* (left) and *all-to-all* (right) operators on the input data sets

3.2 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 $\mathbf{A} = \{A_0, A_1\}$, $\mathbf{B} = \{B_0, B_1\}$, 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, say two pieces of data $\mathbf{A} = \{A_0, A_1\}$, is processed by a first algorithm (using different parameter configurations, say $\mathbf{P} = \{P_0, P_1, P_2\}$), before being delivered to a second service for processing with a matching number of data, say $\mathbf{B} = \{B_0, B_1\}$. The output data

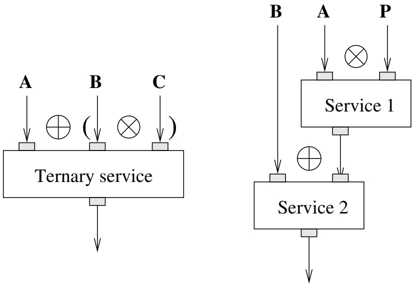


Figure 3: Combining composition operators: multiple input service (left) and cascade of services (right)

set would be:

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

As 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 leads to drastic increases in the number of data to be processed.

3.3 State of the art in data composition

Taverna [36]**.** 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 straight forward.

In the ternary service (left of figure 3), Taverna will produce the:

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

output set. Given that only two input data are available on the first service port, the $\min(m, n)$ truncation rule of the one-to-one (dot product) operator applies. Note that changing the priority of operators will produce a different output. Indeed,

$$(\mathbf{A} \oplus_{\text{Taverna}} \mathbf{B}) \otimes \mathbf{C} = \left\{ \begin{array}{l} \forall i, (A_0 \oplus B_0) \otimes C_i, \\ \forall i, (A_1 \oplus B_1) \otimes C_i \end{array} \right\}$$

Taverna proposes a graphical interface for allowing the user to define the desired priority on the data composition operators.

In the case of the example given in the 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:

$$\mathbf{B} \oplus_{\text{Taverna}} (\mathbf{A} \otimes \mathbf{P}) = \left\{ \begin{array}{ll} B_0 \oplus (A_0 \otimes P_0), & B_1 \oplus (A_1 \otimes P_0) \end{array} \right\} \tag{2}$$

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 segments will be produced. Note that this semantics differs from the one that we consider and that is illustrated in equation 1.

Kepler [31] **and Triana** [43]**.** 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.

MOTEUR. We designed the MOTEUR workflow engine so that it implements the semantics of the operators defined in section 4. MOTEUR recognizes both one-to-one and all-to-all operators (it does recognize Scufl workflows) but it uses the algorithm introduced in section 4 to define the combination semantics.

4 Data composition algorithm

As can be seen, even considering simple examples such as the ones shown in figure 3, the semantics of combining data composition operators is not straight forward. Different workflow engines have different capabilities and implement different combination strategies. Our goal is to define a clear and intuitive semantics for such combinations. We propose an algorithm to implement this data combination strategy.

Taverna provides the most advanced data composition techniques. Yet, we argue that the semantics described in equation 2 is not intuitive for the end user. 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. We therefore adopt the semantics proposed in equation 1 where A_i is consistently combined with B_i.

To formalize and generalize this approach, we need to consider the complete data flows to be processed in the application workflow. In the reminder of this paper, we will consider the very general case, common in scientific applications, where the user needs to independently process sets of input data $\mathbf{A}, \mathbf{B}, \mathbf{C}\ldots$ 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:

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

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 are delivered by the workflow data sources.

4.1 Data composition operator semantics

We consider that the one-to-one composition operator does only make sense when processing related data segments. 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 segments is straight forward. However, when considering a complete application workflow such as the one illustrated in figure 4, other services (*e.g.* S_4) 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.

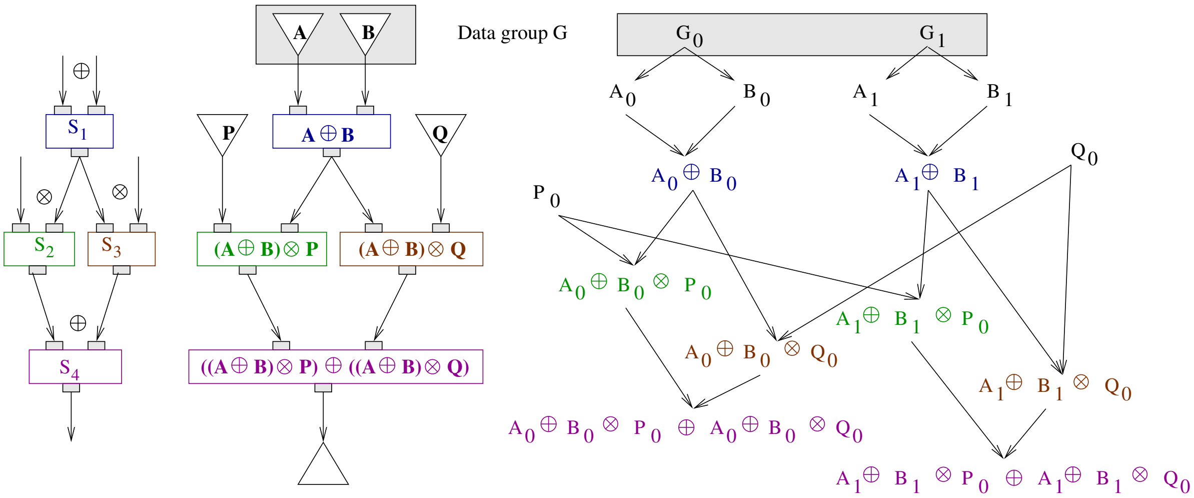
Conversely, the all-to-all operator does not rely on any pre-determined relation between input data segments. Any number of inputs can be combined, with very different meaning (such as data to process and algorithm parameters). Each data received as input yields to one or more invocations of the service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

4.2 Combination semantics

The left of figure 4 represents a sample workflow made of 4 application services and combining the one-to-one and the all-to-all composition operators. In the center of the figure is represented the directed graph of the data sets produced. Given 4 input data sets, $\mathbf{A}, \mathbf{B}, \mathbf{P}$ and $\mathbf{Q}$, the complete workflows produces

$$((\mathbf{A} \oplus \mathbf{B}) \otimes \mathbf{P}) \oplus ((\mathbf{A} \oplus \mathbf{B}) \otimes \mathbf{Q}).$$

as output of the S_4 service. Given the one-to-one operator semantics described above, the data set $\mathbf{A} \oplus \mathbf{B}$ produced by the first service will be non empty if and only if data in $\mathbf{A}$ and $\mathbf{B}$ are related through a group G that is represented at the top of the figure (A_i, the i$^{\text{th}}$ element of $\mathbf{A}$, is correlated with B_i, the i$^{\text{th}}$ element of $\mathbf{B}$).



$\oplus$
S_1
$\otimes$
$\otimes$
S_2
S_3
$\oplus$
S_4
A
B
P
A $\oplus$ B
Q
(A$\oplus$ B)$\otimes$ P
(A$\oplus$ B)$\otimes$ Q
((A$\oplus$ B)$\otimes$ P) $\oplus$ ((A$\oplus$ B)$\otimes$ Q)
Data group G
G_0
G_1
A_0
B_0
A_1
B_1
Q_0
$A_0 \oplus B_0$
$A_1 \oplus B_1$
P_0
$A_0 \oplus B_0 \otimes P_0$
$A_1 \oplus B_1 \otimes P_0$
$A_0 \oplus B_0 \otimes Q_0$
$A_1 \oplus B_1 \otimes Q_0$
$A_0 \oplus B_0 \otimes P_0 \oplus A_0 \oplus B_0 \otimes Q_0$
$A_1 \oplus B_1 \otimes P_0 \oplus A_1 \oplus B_1 \otimes Q_0$

Figure 4: Workflow example (left), associated data sets directed graph (center), and part of the associated directed acyclic data graph.

Considering the service S_4, it is not trivial to determine the content of the output dataset, resulting from a one-to-one composition of the two inputs $(\mathbf{A} \oplus \mathbf{B}) \otimes \mathbf{P}$ and $(\mathbf{A} \oplus \mathbf{B}) \otimes \mathbf{Q}$. Intuitively, two input data $(A_i \oplus B_i) \otimes P_k$ and $(A_j \oplus B_j) \otimes Q_l$ should be combined only if $i = j$. Indeed, combining A_i with B_i, or a subsequent processing of these data, does make sense given that the user established a relation between this input pair through the group G. Conversely, there is no relation between $A_i \oplus B_i$ on the one side and any P_k or Q_l that are combined in an all-to-all operation on the other side. Therefore, the processing of $((A_i \oplus B_i) \otimes P_k) \oplus ((A_i \oplus B_i) \otimes Q_l)$ does make sense for all k and all l.

To formalize this approach we need to consider the data production Directed Acyclic Graph that is represented in right of figure 4. This graph shows how all pieces of input are combined by the different processings. At the roots of the graph, the *input* data are parents of all *produced* data. The formal relation between each data pair (A_i, B_i) is represented through a group instantiation G_i, parent of both A_i and B_i. We will name *orphan* data, input data that have no group parent such as P_0 and Q_0. 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 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.

The interpretation of the first rule is straight forward. The second rule is illustrated by the full data graph displayed at the right of figure 4. For instance, the data $A_0 \oplus B_0$ is produced from A_0 and B_0 because there exists a common ancestor G_0 to both A_0 and B_0. Similarly, $((A_0 \oplus B_0) \otimes P_0) \oplus ((A_0 \oplus B_0) \otimes Q_0)$ is computed because $A_0 \oplus B_0$ is a common ancestor to $(A_0 \oplus B_0) \otimes P_0$ and $(A_0 \oplus B_0) \otimes Q_0$. There exists other common ancestors such as A_0, B_0, and G_0 but it is not needed to go back further in the data graph as soon as one of them has been found. Note that in a more complex workflow topologies, the common ancestor does not need to be an immediate parent. It can be easily demonstrated by recurrence that following this rule, two input data sets may be composed one-to-one if and only if there exists a grouping relation between them at the root of the data graph.

4.3 Algorithm and implementation

To implement the data composition operators semantic introduced above, MOTEUR dynamically resolves the data combination problem by applying the following algorithm:

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;
- 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 triggers 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 5.

4.4 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 are declared or send to the workflow inputs are rather used as an implicit relation.

To ease the workflow generation from the user point of view, groups can be implicitly generated when they are not explicitly specified by the user. Figure 5 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 are orphans. For example, in the case illustrated in left of figure 5, the input datasets **A** and **B** are orphans and bound *one-to-one* by the service S_1. An implicit group is therefore created between **A** and **B**. In the case illustrated in the right side of figure 5, 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.

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

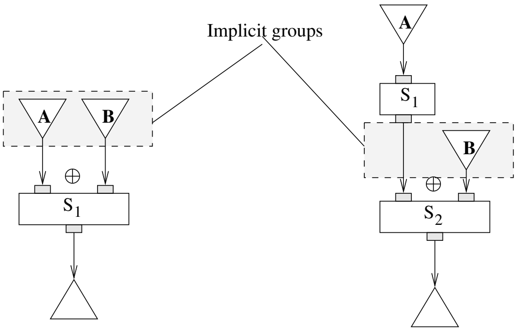


Figure 5: Implicit groups definition.

4.5 Coping with data fragments

So far, we have only considered the case were the number of outputs of a service matches the number of inputs. In some cases though, an application service will split input data in smaller fragments, either for dealing with smaller data sets (*e.g.* a 3D medical image is split in a stack of 2D slices) or because the service code function implies that it produces several outputs for each input. The workflow displayed in figure 6 illustrates such a situation. The service S_1 is splitting each input data (*e.g.* A_0) in several fragments (A_0^0, A_0^1 and A_0^2).

In the example given in figure 6, it is expected that service S_2 will receive the same number of data on both input ports (one-to-one composition operator). However, there is no way for the user to specify an explicit grouping between two data sets. Grouping the data sets **A** with **B** would only create a relation between A_0 and B_0. Therefore, the fragments A_0^0, A_0^1 and A_0^2, children of A_0, would all be related to B_0 and the service S_2 would produce

$$\mathbf{A} \oplus \mathbf{B} = \{A_0^0 \oplus B_0, A_0^1 \oplus B_0, A_0^2 \oplus B_0\}.$$

Instead, the implicit grouping strategy will group $S_1(A_0)$ outputs with **B**. Consequently, the grouping will result in the data graph shown in right of figure 6 and the output produced will be

$$\mathbf{A} \oplus \mathbf{B} = \{A_0^0 \oplus B_0, A_0^1 \oplus B_1, A_0^2 \oplus B_2\}$$

as expected. Note that the number of inputs to service S_2 needs to be consistent in this case.

5 Scheduling and executing workflows of services

The service-based approach is making services composition easier than the task-based approach as discussed in section 2. 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. The services are black boxes isolating the workflow manager from the execution infrastructure. In this context, most known optimization solutions do not hold.

5.1 Related work

Many solutions have indeed been proposed in the task-based paradigm to optimize the scheduling of an application in distributed environments [10]. Concerning workflow-based applications, previous works [4] 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. Indeed, in this latest approach, the workflow manager is not responsible for the task submission and thus cannot optimize the resource allocation.

Focusing on the service-based approach, nice developments such as the DIET middleware [9] and comparable approaches [41, 2] introduce specific strategies such as hierarchical scheduling. In [7] for instance, the authors describe a way to handle file persistence in distributed environments, which leads to strong performance improvements. However, those works focus on middleware design and do not include any workflow management yet. Moreover, those solutions require specific middleware components to be deployed on the target infrastructure. As far as we know, such a deployment has only been done on experimental platforms yet [6], and it is hardly possible for an application running on a production infrastructure.

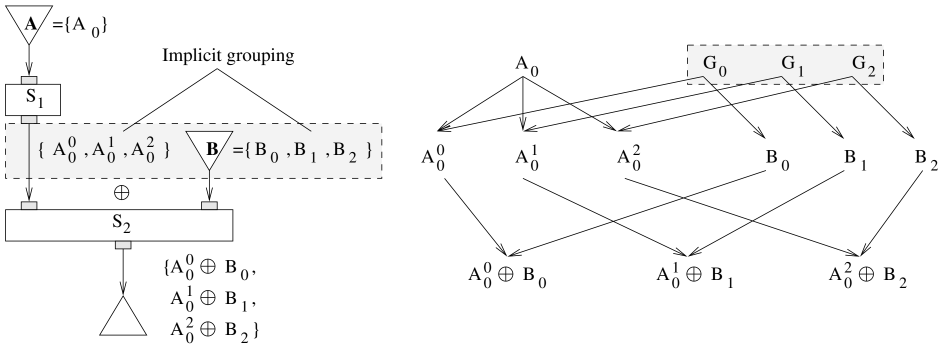


Figure 6: Implicit groups relating data fragments (A_0^0, A_0^1, A_0^2) and input **B**.

Hence, there is a strong need for precisely identifying generic optimization solutions that apply to service-based workflows. In the following sections, we are exploring different strategies for optimizing the workflow execution in a service-based approach, thus offering the flexibility of services and the efficiency of tasks. First of all, several level of parallelism can be exploited when considering the workflow execution for taking advantage of the grid computing capabilities. We describe them and then study their impact on the performances with respect to the characteristics of the considered application. Besides, we propose a solution for grouping sequential jobs in the workflow, thus allowing more elaborated optimization strategies in the service-based workflow area.

5.2 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 [33]. 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 [44, 23] 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 processor being executed.

5.3 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 cited above.

5.4 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* is achievable when a service is able to process several data sets simultaneously with a minimal performance loss. This capability involves the processing of independent data on different computing resources.

Enabling data parallelism implies, on the one hand, that the services are able to process many parallel connections and, on the other hand, that the workflow engine is able to submit several simultaneous queries to a service leading to the dynamic creation of several threads. Moreover, a data parallel workflow engine should implement a dedicated data management system. Indeed, in case of a data parallel execution, a data is able to overtake another one during the processing and this could lead to a causality problem, as we exemplified in [18]. To properly tackle this problem, data provenance has to be monitored during the data parallel execution. Detailed work on data provenance can be found in [47].

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 figure 7. 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 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 7, 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.

As demonstrated in the next sections, fully taking into account this level of parallelism is critical in service-based workflows, whereas it does not make any sense in task-based ones. Indeed, in this case it is covered by the workflow parallelism because each task is explicitly described in the workflow description.

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

Figure 7: Data parallel execution diagram of the workflow of figure 1

5.5 Services parallelism

Input data sets are likely to be independent from each other. This is for example the case when a single workflow is iterated in parallel on many input data sets. *Services parallelism* is achievable when 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 figure 1, to be executed on the 3 independent input data sets D_0, D_1 and D_2. Figure 8 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).

Data synchronization barriers, presented in section 2.4, 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.

Here again, we show in the next section that service parallelism is of major importance to optimize the execution of service-based workflows. In task-based workflow, this level of parallelism does not make any sense because it is included in the workflow parallelism.

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

Figure 8: Service parallel execution diagram of the workflow of figure 1

5.6 Theoretical performance analysis

The data and service parallelism described above are specific to the service-based workflow approach. To precisely quantify how they influence the application performances we model the workflow execution time for different configurations. We first present general results and then study particular cases, making assumptions on the type of application run.

5.6.1 Definitions and notations

- In the workflow, a *path* denotes a set of processors linking an input to an output. The *critical path* of the workflow denotes the longest path in terms of execution time.
- n_W denotes the number of services on the critical path of the workflow and n_D denotes the number of data sets to be executed by the workflow.
- i denotes the index of the i^{th} service of the critical path of the workflow ($i \in [0, n_W - 1]$). Similarly j denotes the index of the j^{th} data set to be executed by the workflow ($j \in [0, n_D - 1]$).
- $T_{i,j}$ denotes the duration in seconds of the treatment of the data set j by the service i. If the service submits jobs to a grid infrastructure, this duration includes the overhead introduced by the submission, scheduling and queuing times.
- $\sigma_{i,j}$ denotes the absolute time in seconds of the end of the treatment of the data set j by the service i. The execution of the workflow is assumed to begin at $t = 0$. Thus $\sigma_{0,0} = T_{0,0} > 0$.
- Σ denotes the total execution time of the workflow

$$\Sigma = \max_{j<n_D}(\sigma_{n_W-1,j}) \tag{3}$$

5.6.2 Hypotheses

The critical path is assumed not to depend on the data set. This hypothesis seems reasonable for most applications but may not hold in some cases as for example the one of workflows including algorithms containing optimization loops whose convergence time is likely to vary in a complex way with regards to the nature of the input data set.

Data parallelism is assumed not to be limited by infrastructure constraints. We justify this hypothesis considering that our target infrastructure is a grid, whose computing power is sufficient for our application.

In this section, workflows are assumed not to contain any synchronization processors. Workflows containing such synchronization barriers may be analyzed as two sub workflows respectively corresponding to the parts of the initial workflow preceding and succeeding the synchronization barrier.

5.6.3 Execution times modeling

Under those hypotheses, we can determine the expression of the total execution time of the workflow for different execution policies:

- Sequential case (without service nor data parallelism):

$$\Sigma = \sum_{i<n_W} \sum_{j<n_D} T_{i,j} \tag{4}$$

- Case DP: Data parallelism only

$$\Sigma_{DP} = \sum_{i<n_W} \max_{j<n_D} \{T_{i,j}\} \tag{5}$$

- Case SP: Service parallelism only

$$\begin{aligned}
\Sigma_{SP} &= T_{n_W-1,n_D-1} + m_{n_W-1,n_D-1} && (6)\\
&\text{with:} \quad \forall i \neq 0 \quad \text{and} \quad \forall j \neq 0, \\
m_{i,j} &= \max(T_{i-1,j} + m_{i-1,j}, T_{i,j-1} + m_{i,j-1}) \\
&\text{and:} \\
m_{0,j} &= \sum_{k<j} T_{0,k} \quad \text{and} \quad m_{i,0} = \sum_{k<i} T_{k,0}
\end{aligned}$$

- Case DSP: both Data and Service parallelism

$$\Sigma_{DSP} = \max_{j<n_D} \left\{ \sum_{i<n_W} T_{i,j} \right\} \qquad (7)$$

All the above expressions of the execution times can be demonstrated recursively [17].

5.6.4 Asymptotic speed-ups

To better understand the properties of each kind of parallelism, it is interesting to study the asymptotic speed-ups resulting from service and data parallelism in particular application cases.

Massively data-parallel workflows. Let us consider a massively (*embarrassingly*) data-parallel application (single processor P_0, very large number of input data). In this case, $n_W = 1$ and the execution time is:

$$\Sigma_{DP} = \Sigma_{DSP} = \max_{j<n_D}(T_{0,j}) \quad \ll \quad \Sigma = \Sigma_{SP} = \sum_{j<n_D} T_{0,j}$$

In this case, data parallelism leads to a significant speed-up. Service parallelism is useless but it does not lead to any overhead.

Non data intensive workflows. In such workflows, $n_D = 1$ and the execution time is:

$$\Sigma_{DSP} = \Sigma_{DP} = \Sigma_{SP} = \Sigma \quad = \quad \sum_{i<n_W} T_{i,0}$$

In this case, neither data nor service parallelism lead to any speed-up. Nevertheless, none of them does introduce any overhead.

Data intensive complex workflows. In this case, we will suppose that $n_W > 1$ and $n_D > 1$. In order to analyze the speed-ups introduced by service and data parallelism, we make the simplifying assumption of constant execution times: $T_{i,j} = T$. The workflow execution time then resumes to:

$$\begin{aligned} \Sigma &= n_D \times n_W \times T \\ \Sigma_{DP} = \Sigma_{DSP} &= n_W \times T \\ \Sigma_{SP} &= (n_D + n_W - 1) \times T \end{aligned}$$

If service parallelism is disabled, the speed-up introduced by data parallelism is:

$$S_{DP} \quad = \quad \frac{\Sigma}{\Sigma_{DP}} = n_D$$

If service parallelism is enabled, the speed-up introduced by data parallelism is:

$$S_{DSP} \quad = \quad \frac{\Sigma_{SP}}{\Sigma_{DSP}} = \frac{n_D + n_W - 1}{n_W}$$

If data parallelism is disabled, the speed-up induced by service parallelism is:

$$S_{SP} \quad = \quad \frac{\Sigma}{\Sigma_{SP}} = \frac{n_D \times n_W}{n_D + n_W - 1}$$

Theoretically, service parallelism does not lead to any speed-up if it is coupled with data parallelism: $S_{SDP} = \frac{\Sigma_{DP}}{\Sigma_{DSP}} = 1$. Thus, under those assumptions, service parallelism may not be of any use on fully distributed systems. However, section 9 will show that even in case of homogeneous input data sets, T is hardly constant in production systems due to the high variability of the overhead coming from submission, scheduling and queuing times on such large scale and multi-user platforms. The constant execution time hypothesis does not hold. This appears to be a significant difference between grid computing and traditional cluster computing. Figure 9 illustrates on a simple example why service parallelism do provide a speed-up even if data parallelism is enabled, if the assumption of constant execution times does not hold. The left diagram does not take into account service parallelism whereas the right one does. The processing time of the data set D_0 is twice as long as the other ones on service P_0 and the execution time of the data set D_1 is three times as long as the other ones on service P_1. It can for example occur if D_0 was submitted twice because an error occurred and if D_1 remained blocked on a waiting queue. In this case, service parallelism improves performances beyond data parallelism as it enables some computations overlap. It justifies the experimental observations done in section 9.

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

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

Figure 9: Workflow execution time without (left) and with (right) service parallelism when the execution time is not constant.

5.7 State of the art in service-based workflow managers

Workflow parallelism is usually implemented in existing workflow managers.

Taverna implements data parallelism (known as *multiple threads* in this context). However, data parallelism is limited to a fixed number of threads specified in the Scufl workflow description language. It cannot dynamically adapt to the size of data sets to be processed. Service parallelism is not supported yet but this feature has been proposed for the next major release of the engine (version 2).

Kepler implements services parallelism through the *Physical Network* (PN) director. There is no data parallelism in Kepler.

Triana does not implement service nor data parallelism.

MOTEUR was designed to optimize the performance of data-intensive applications on grids by implementing the three level of parallelism. To our knowledge, this is the first service-based workflow engine doing so.

6 Legacy code wrapping

To ease the embedding of legacy-codes in the service-based framework, an application-independent job submission service is required. In this section, we briefly review systems that are used to wrap legacy code into services to be embedded in service-based workflows.

The Java Native Interface (JNI) has been widely adopted for the wrapping of legacy codes into services. Wrappers have been developed to automate this process. In [22], an automatic JNI-based wrapper of C code into Java and the corresponding type mapper with Triana [43] is presented: JACAW generates all the necessary java and C files from a C header file and compiles them. A coupled tool, MEDLI, then maps the types of the obtained Java native method to Triana types, thus enabling the use of the legacy code into this workflow manager. Related to the ICENI workflow manager [16], the wrapper presented in [29] is based on code re-engineering. It identifies distinct components from a code analysis, wrap them using JNI and adds a specific CXML interface layer to be plugged into an ICENI workflow.

The WSPeer framework [21], interfaced with Triana, aims at easing the deployment of Web-Services by exposing many of them at a single endpoint. It differs from a container approach by giving to the application the control over service invocation. The Soaplab system [40] is especially dedicated to the wrapping of command-line tools into Web-Services. It has been widely used to integrate bioinformatics executables in workflows with Taverna [36]. It is able to deploy a Web-Service in a container, starting from the description of a command-line tool. This command-line description, referred to as the metadata of the analysis, is written for each application using the ACD text format file and then converted into a corresponding XML format. Among domain specific descriptions, the authors underline that such a command-line description format must include (i) the description of the executable, (ii) the names and types of the input data and parameters and (iii) the names and types of the resulting output data. As described latter, the format we used includes those features and adds new ones to cope with requirements of the execution of legacy code on grids.

The GEMLCA environment [12] addresses the problem of exposing legacy code command-line programs as Grid services. It is interfaced with the P-GRADE portal workflow manager [25]. The command-line tool is described with the LCID

(Legacy Code Interface Description) format which contains (i) a description of the executable, (ii) the name and binary file of the legacy code to execute and (iii) the name, nature (input or output), order, mandatory, file or command line, fixed and regular expressions to be used as input validation. A GEMLCA service depends on a set of target resources where the code is going to be executed. Architectures to provide resource brokering and service migration at execution time are presented in [27].

Apart from this latest early work, all of the reviewed existing wrappers are static: the legacy code wrapping is done offline, before the execution. This is hardly compatible with our approach, which aims at optimizing the whole application execution at runtime. We thus developed a specific grid submission Web-Service, which can wrap any executable at runtime, thus enabling the use of optimization strategies by the workflow manager.

The following section 6.1 introduces a generic application code wrapper compliant with the Web Services specification. It enables the execution of any legacy executable through a standard services interface. The subsequent section 7 proposes a code execution optimization strategy that can be implemented thanks to this generic wrapper. Finally, section 8 proposes a service oriented architecture of the system, based on a service factory.

6.1 Generic web service wrapper

We developed a specific grid submission Web Service. This service is generic in the sense that it is unique and it does not depend on the executable code to submit. It exposes a standard interface that can be used by any Web Service compliant client to invoke the execution. It completely hides the grid infrastructure from the end user as it takes care of the interaction with the grid middleware. This interface plays the same role as the ACD and LCID files quoted above, except that it is interpreted at the execution time.

To accommodate to any executable, the generic service is taking two different inputs: a descriptor of the legacy executable command line format, and the input parameters and data of this executable. The production of the legacy code descriptor is the only extra work required from the application developer. It is a simple XML file which describes the legacy executable location, command line parameters, input and output data.

6.2 Legacy code descriptor

The command line description has to be complete enough to allow dynamic composition of the command line from the list of parameters at the service invocation time and to access the executable and input data files. As a consequence, the executable descriptor contains:

1. The name and access method of the executable. In our current implementation, access methods can be a URL, a Grid File Name (GFN) or a local file name. The wrapper is responsible for fetching the data according to different access modes.
2. The access method and command-line option of the input data. As our approach is service-based, the actual name of the input data files is not mandatory in the description. Those values will be defined at the execution time. This feature differs from various job description languages used in the task-based middlewares. The command-line option allows the service to dynamically build the actual command-line at the execution time.
3. The command-line option of the input parameters: parameters are values of the command-line that are not files and therefore which do not have any access method.
4. The access method and command-line option of the output data. This information enables the service to register the output data in a suitable place after the execution. Here again, in a service-based approach, names of output data files cannot be statically determined because output file names are only generated at execution time.
5. The name and access method of the sandboxed files. Sandboxed files are external files such as dynamic libraries or scripts that may be needed for the execution although they do not appear

on the command-line.

6.3 Example

An example of a legacy code description file is presented in figure 10. It corresponds to the description of the service `crestLines` of the workflow depicted in figure 15. It describes the script `CrestLines.pl` which is available from the server `legacy.code.fr` and takes 3 input arguments: 2 files (options `-im1` and `-im2` of the command-line) that are already registered on the grid as GFNs at execution time and 1 parameter (option `-s` of the command-line). It produces 2 files that will be registered on the grid. It also requires 3 sandboxed files that are available from the server.

6.4 Discussion

This generic service highly simplifies application development because it is able to wrap any legacy code with a minimal effort. The application developer only needs to write the executable descriptor for her code to become service aware.

But its main advantage is in enabling the sequential services grouping optimization introduced in section 7. Indeed, as the workflow enactor has access to the executable descriptors, it is able to dynamically create a virtual service, composing the command lines of the codes to be invoked, and submitting a single job corresponding to this sequence of command lines invocation.

It is important to notice that our solution remains compatible with the services standards. The workflow can still be executed by other enactors, as we did not introduce any new invocation method. Those enactors will make standard service calls (e.g. SOAP ones) to our generic wrapping service. However, the optimization strategy described in the next section is only applicable to services including the descriptor presented in section 6.2. We call those services MOTEUR services.

```
<description>
  <executable name="CrestLines.pl">
    <access type="URL">
      <path value="http://legacy.code.fr"/>
    </access>
    <value value="CrestLines.pl"/>
    <input name="floating_image" option="-im1">
      <access type="GFN"/>
    </input>
    <input name="reference_image" option="-im2">
      <access type="GFN"/>
    </input>
    <input name="scale" option="-s"/>
    <output name="crest_reference" option="-c1">
      <access type="GFN"/>
    </output>
    <output name="crest_floating" option="-c2">
      <access type="GFN"/>
    </output>
    <sandbox name="convert8bits">
      <access type="URL">
        <path value="http://legacy.code.fr"/>
      </access>
      <value value="Convert8bits.pl"/>
    </sandbox>
    <sandbox name="copy">
      <access type="URL">
        <path value="http://legacy.code.fr"/>
      </access>
      <value value="copy"/>
    </sandbox>
    <sandbox name="cmatch">
      <access type="URL">
        <path value="http://legacy.code.fr"/>
      </access>
      <value value="cmatch"/>
    </sandbox>
  </executable>
</description>
```

Figure 10: Descriptor example

7 Services grouping optimization strategy

We propose a services grouping strategy to further optimize the execution time of a workflow. Services grouping consists in merging multiple jobs into a single one. It reduces the grid overhead induced by the submission, scheduling, queuing and data transfers times whereas it may also reduce the parallelism. In particular sequential processors grouping is interesting because those processors do not benefit from any parallelism. For example, considering the workflow of our application presented on figure 15 we can, for each data set, group the execution of the `crestLines` and the `crestMatch` jobs on the one hand and the

PFMatchICP and the PFRegister ones on the other hand.

Grouping jobs in the task-based approach is straightforward and it has already been proposed for optimization [4]. Conversely, jobs grouping in the service-based approach is usually not possible given that (i) the services composing the workflow are totally independent from each other (each service is providing a different data transfer and job submission procedure) and (ii) the grid infrastructure handling the jobs does not have any information concerning the workflow and the job dependencies. Consider the simple workflow represented on the left side of figure 11. On top, the services for P_1 and P_2 are invoked independently. Data transfers are handled by each service and the connection between the output of P_1 and the input of P_2 is handled at the workflow engine level. On the bottom, P_1 and P_2 are grouped in a virtual single service. This service is capable of invoking the code embedded in both services sequentially, thus resolving the data transfer and independent code invocation issues.

7.1 Grouping strategy

Services grouping can lead to significant speed-ups, especially on production grids, as it is demonstrated in the next section. However, it may also slow down the execution by limiting parallelism. We thus have to determine efficient strategies to group services.

In order to determine a grouping strategy that does not introduce any overhead, neither from the user point of view, nor from the infrastructure one, we impose the two following constraints: (i) the grouping strategy must not limit any kind of parallelism (user point of view) and (ii) during their execution, jobs cannot communicate with the workflow manager (infrastructure point of view). The second constraint prevents a job from holding a resource just waiting for one of its ancestor to complete. An implication of this constraint is that if services A and B are grouped together, the results produced by A will only be available once B will have completed.

A workflow may include both MOTEUR Web-Services (*i.e.* services that are able to be grouped) and classical ones, that could not be grouped. Assuming those two constraints, the following rule is sufficient to process all the possible groupings of two services of the workflow:

> Let A be a MOTEUR service of the workflow and $\{B_0,...B_n\}$ its children in the service graph. **If** there exists a MOTEUR child B_i which is an ancestor of every B_j $(i \neq j)$ and whose each ancestor C is an ancestor of A or A itself, **then** group A and B_i.

Indeed, every violation of this rule also violates one of our constraints as it can easily be shown. The grouping strategy tests this rule for each MOTEUR service A of the workflow. Groups of more than two services may be recursively composed from successive matches of the grouping rule.

For example, the workflow displayed in figure 12, extracted from our medical imaging application, is made of 4 MOTEUR services that can be grouped into a single one through 3 applications of the grouping rule. On this figure, notations nearby the services corresponds to the ones introduced in the grouping rule.

The first application case of the grouping rule is represented on the left of the figure. The tested MOTEUR service A is crestLines. A is connected to the workflow inputs and it has two children, B_0 and B_1. B_0 is a father of B_1 and it only has as single ancestor which is A. The rule thus matches: A and B_0 can be grouped. If there were a service C ancestor of B_0 but not of A as represented on the figure, the rules would be violated: A and C can be executed in parallel before starting B_0. Similarly, if there were a service D the rule would be violated as the workflow manager would need to communicate results during the execution of the grouped jobs.

In the second application case, in the middle of the figure, the tested service A is now crestMatch. A has only a single child: B_0. B_0 has two ancestors, A and C. The rule matches because C is an ancestor of A. A and B_0 can then be grouped.

For the last rule application case, on the right of figure 12, A is the PFMatch service. It has only one child, B_0, who only has a single ancestor, A. The rule matches and those services can thus be grouped.

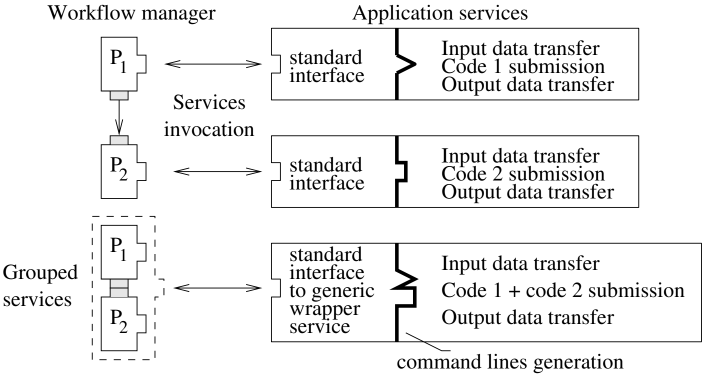


Figure 11: Classical services invocation (top) and services grouping (bottom).

When A is the `PFRegister` service, the grouping rule does not match because it does not have any child. Note that in this example, the recursive grouping strategy will lead to a single job call.

8 Dynamic generic service factory

The generic web service drastically simplifies the wrapping of legacy code into application services. However, it is mixing two different roles: (i) the legacy command line generation and (ii) the grid submission. Job submission is only dependent on the target grid and not on the application service itself. In a Service Oriented Architecture (SOA) it is preferable to split these two roles into two independent services for several reasons. First, the submission code does not need to be replicated in all application services. Second, the submission role can be transparently and dynamically changed (to submit to a different infrastructure) or updated (to adapt to middleware evolutions). In addition, an application wrapper factory service further facilitates the wrapping of legacy code services and their grouping. We thus introduce a complete SOA design based on three main services as illustrated in figure 14.

The (blue) MOTEUR web services represents legacy code wrapping services. They are assembling command lines and invoking the (red) submission service for handling code execution on the grid infrastructure. The code wrapper factory service is responsible for dynamically generating and deploying application services. The aim of this factory is to achieve two antagonist goals:

- To expose legacy codes as autonomous web services respecting the main principles of Service Oriented Architectures.
- To enable the grouping of two of these web services in as a unique one for optimizing the execution.

On one hand, the specific web service implementation details (*i.e.* the execution of legacy code on a grid infrastructure) are hidden to the consumer. On the other hand, when the consumer is a workflow manager which can group jobs, it needs to be aware of the real nature the web service (the encapsulation of a MOTEUR descriptor) so that it could merge them at runtime. We choose to use the WSDL XML Format extension mechanism which allows to insert user defined XML elements in the WSDL content itself. On figure 14, we represent the overall architecture and some usage scenario. First, the legacy code provider submits (**1.a**) a MOTEUR XML descriptor P1 to the MOTEUR factory. The factory, then dynamically deploy (**1.b**) a web service which wraps the submission of the legacy code to the grid via the generic service

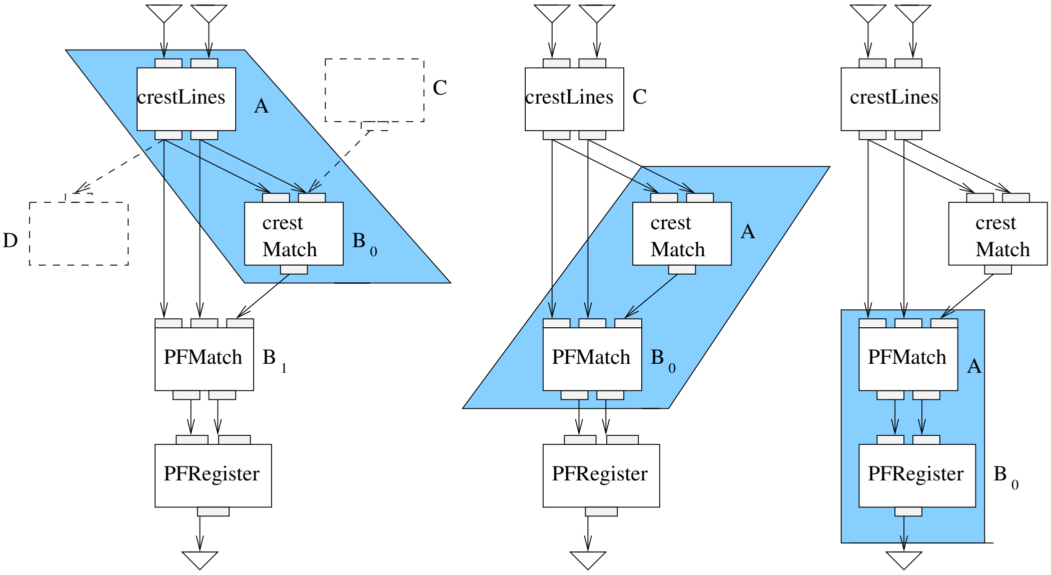


Figure 12: Services grouping examples

wrapper. Another provider do the same with the descriptor of P2 (**2.a**). The resulting web services expose their WSDL contracts to the external world with a specific extension associated with the WSDL operation. For instance, the WSDL contract resulting of the deployment of the `crestLines` legacy code described on figure 10 is printed on figure 13.

This WSDL document defines two types (`CrestLines-request` and `CrestLines- response`) corresponding to the descriptor inputs and outputs and a single `Execute` operation. Notice that in the binding section, the WSDL document contains an extra `MOTEUR-descriptor` tag pointing to the URL of the legacy code descriptor file (`location`) and a binding to the Execute operation (`soap:operation`).

Suppose now that the workflow manager identifies a services grouping optimization (*e.g.* P_1 and P_2) (**3.a** on figure 14). Because of its ability to discover the extended nature of these two services, the engine can retrieve the two corresponding MOTEUR descriptors. It can ask the factory to *combine* them (**3.b**) resulting in a single composite web service (**3.c**) which exposes an operation taking its inputs from P_1 (and P_2 inputs coming from other external services) and returning the outputs defined by P_2 (and P_1 outputs going to other external services). This composite web service is of the same type than any regular legacy code wrapping service. It is accessible through the same interface and it also delegates the grid submission to the generic submission web service by sending the composite MOTEUR descriptor and the input link of P_1 and P_2 in the workflow.

9 Experimental results

The goal of this section is to present experimental results that quantify the relevance of the optimizations described above on a real service-based data-intensive application workflow. We evaluate MOTEUR's performances on two different grid infrastructures.

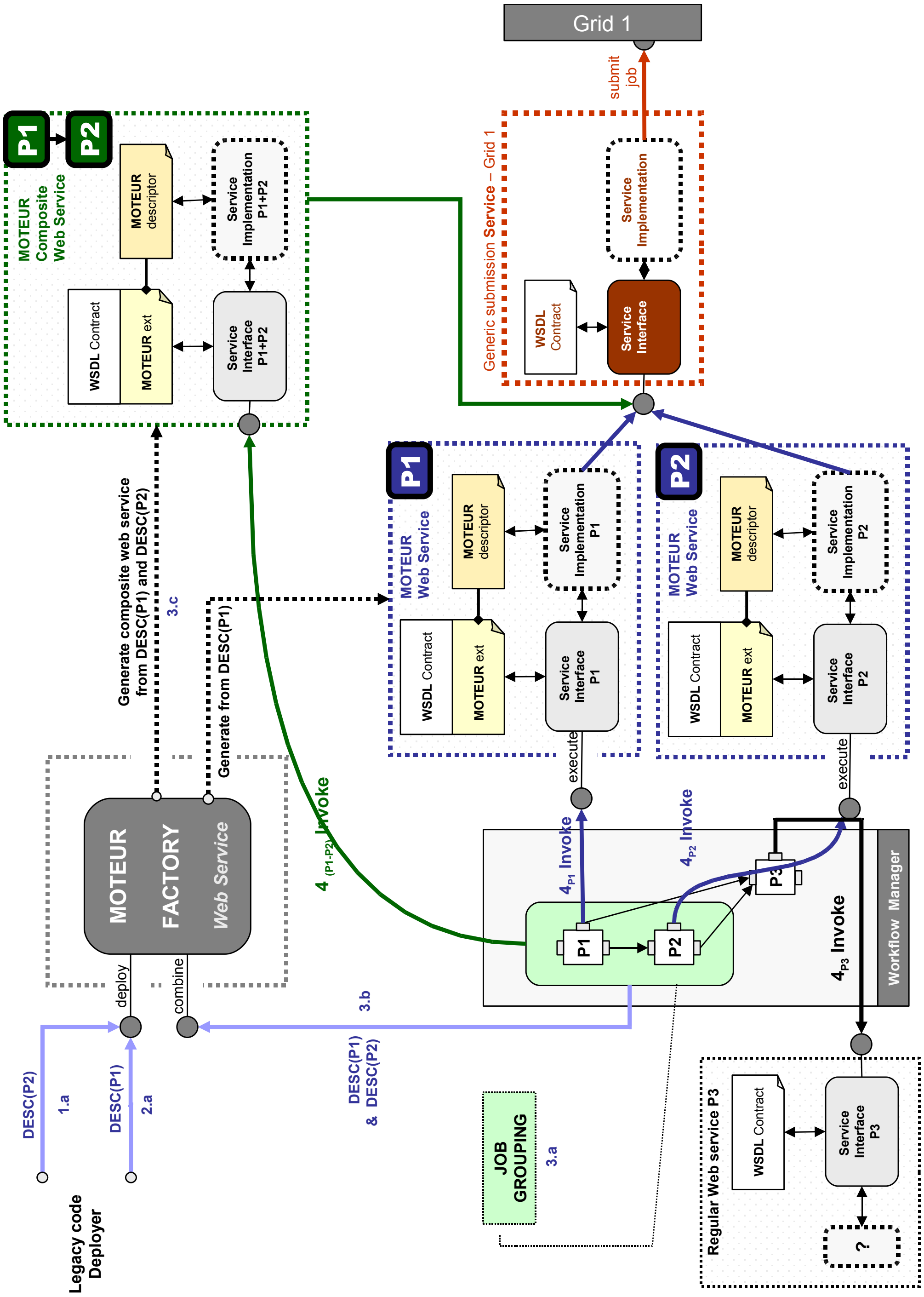


Figure 14: Services factory.

```
<?xml version="1.0" encoding="utf-8" ?>
<definitions ...>
 <types>
  <schema>
   <element name="CrestLines-request">
     <complexType>
      <sequence>
       <element name="floating_image"
                type="string"... />
       <element name="reference_image"
                type="string"... />
       <element name="scale" type="string"... />
      </sequence>
     </complexType>
    </element>
    <element name="CrestLines-response">
      <complexType>
        <sequence>
          <element name="crest_reference"
                   type="string"... />
          <element name="crest_floating"
                   type="string"... />
        </sequence>
      </complexType>
    </element>
  </schema>
 </types>
 <message name="ExecuteSoapIn">
  <part name="parameters"
        element="CrestLines.pl-request" />
 </message>
 <message name="ExecuteSoapOut">
  <part name="parameters"
        element="CrestLines.pl-response" />
 </message>
 <portType name="CrestLines.plSoap">
  <operation name="Execute">
   <input message="ExecuteSoapIn" />
   <output message="ExecuteSoapOut" />
  </operation>
 </portType>
 <binding ...>
  <soap:binding transport="http://..." />
   <operation name="Execute">
    <soap:operation soapAction="http://.../Execute"
          style="document" />
     <MOTEUR-descriptor xmlns="urn:...">
      <location>http://...</location>
     </MOTEUR-descriptor>
     ....
   </operation>
 </binding>
</definitions>
```

Figure 13: WSDL generated by the factory

9.1 MOTEUR implementation

We implemented a prototype of a workflow enactor taking into account the optimizations described in section 5: workflow, data and service parallelism and sequential processors grouping. Our hoMe-made OpTimisEd scUfl enactoR (MOTEUR) prototype was implemented in Java, in order to be platform independent. It is available under CeCILL Public License (a GPL-compatible open source license) at http://www.i3s.unice.fr/~glatard. To our knowledge, this is the only service-based workflow enactor providing all these levels of optimization.

The workflow description language adopted is the Simple Concept Unified Flow Language (Scufl) used by the Taverna workbench [36]. This language is well disseminated in the e-Science community. Apart from describing the data links between the services, the Scufl language allows to define 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 used those coordination constraints to identify services that require data synchronization.

We developed an XML-based language 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.

Handling the data composition patterns presented in section 3 in a service and data parallel workflow is not straightforward because produced data sets have to be uniquely identified. Indeed they are likely to be computed in a different order in every service, which could lead to wrong dot product computations. Moreover, 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 refering to all the intermediate results computed to process it. This tree unambiguously identifies the data.

Finally, MOTEUR is implementing an interface to both Web Services and GridRPC instrumented application code.

9.2 Bronze Standard application

We made experiments considering the *Bronze Standard*, an application that aims at assessing medical image registration algorithms. Medical image registration consists in searching a transformation (that is to say 6 parameters in the rigid case – 3 rotation angles and 3 translation parameters) between two images, so that the first one (the floating image) can superimpose on the second one (the reference image) in a common 3D frame. Medical image registration algorithms are a key component of medical image analysis procedures.

A difficult problem, as for many other medical image analysis procedures, is the assessment of these algorithms robustness, accuracy and precision [24]. Indeed, there is no well established *gold standard* to compare to the algorithm results. Different approaches have been proposed to solve this issue. It is possible to simulate artificial images from a controlled model and to experiment the algorithm on these synthetic images [3]. However, realistic images are difficult to produce and hardly perfect enough for fine assessment of the algorithms. Phantoms (manufactured objects with properties close to human tissues for the imaging modality studied) can also be used to acquire test images. However, it is also very difficult to manufacture realistic enough phantoms.

An alternative for assessing registration algorithms is a statistical approach called the *Bronze Standard* [35]. The goal is basically to compute the registration of a maximum of image pairs with a maximum number of registration algorithms so that we obtain a largely overestimated system to relate the geometry of all the images. It makes this application very compute and data-intensive.

Suppose that we have n images of the same organ of one patient and m registration algorithms. We have in fact only $n-1$ free transformations to estimate that relate all these images, say $\bar{T}_{i,i+1}$. The transformation between images i and j is obtained using a compositions such as $\bar{T}_{i,j} = \bar{T}_{i,i+1} \circ \bar{T}_{i+1,i+2} \circ \ldots \circ \bar{T}_{j-1,j}$ if $i < j$ (or the inverse of both terms if $j > i$). The free transformation parameters are computed by minimizing the prediction error on the observed registrations:

$$\min_{\bar{T}_{1,2},\bar{T}_{2,3},\ldots,\bar{T}_{n-1,n}} \sum_{i,j\in[1,n],k\in[1,m]} d\left(T^k_{i,j}, \bar{T}_{i,j}\right)^2 \qquad (8)$$

where $T^k_{i,j}$ is the transformation computed between image i and j by the k^{th} registration algorithm, and d is a distance function between transformations chosen as a robust variant of the left invariant distance on rigid transformation developed in [38]. The estimation $\bar{T}_{i,i+1}$ of the perfect registration $T_{i,i+1}$ is called bronze standard because the result converges toward $T_{i,i+1}$ as the number of methods m and the number of images n increase. Indeed, considering a given registration method, the variability due to the noise in the data decreases as the number of images n increases, and the registration computed converges toward the perfect registration up to the intrinsic bias (if there is any) introduced by the method. Now, using different registration procedures based on different methods, the intrinsic bias of each method also becomes a random variable, which is hopefully centered around zero and averaged out in the minimization procedure. The different bias of the methods are now integrated into the transformation variability. To fully reach this goal, it is important to use as many independent registration methods as possible.

In this process, we do not only estimate the optimal transformations, but also the rotational and translational variance of the "transformation measurements", which are propagated through the criterion to give an estimated of the variance of the optimal transformations. These variance should be considered as a fixed effect (i.e. these parameters are common to all patients for a given image registration problem, contrarily to the transformations) so that they can be computed more faithfully by multiplying the number of patients.

The workflow of the bronze standard application is represented on figure 15. In the following experiments, we are considering $m = 4$ different registration algorithms in our implementation of the bronze standard method: (1) *Baladin* and (2) *Yasmina* are intensity-based. The former uses a block matching strategy while the later optimizes a similarity measure on the complete images using the Powel algorithm. (3) *CrestMatch* is a prediction-verification method and (4) *PFRegister* is based on the ICP algorithm. Both CrestMatch and PFRegister register features (crest lines) extracted from the input images. These algorithms are further described in [35]. The two inputs *referenceImage* and *floatingImage* correspond to the image sets on which the evaluation is to be processed. The first registration algorithm is *crestMatch*. Its result is used to initialize the other registration algorithms which are *Baladin*, *Yasmina* and *PFMatchICP*/*PFRegister*. *crestLines* is a preprocessing step. Finally, the *MultiTransfoTest* service is responsible for the evaluation of the accuracy of the registration algorithms, leading to the outputs values of the workflow. This service evaluates the accuracy of a specified registration algorithm by comparing its results with means computed on all the others. Thus, the *MultiTransfoTest* service has to be synchronized: it must be enacted once every of its ancestor is inactive. This is why we figured it with a double square on figure 15.

We chose this particular application because it is a real example of data-intensive workflow in the medical imaging field. Moreover, it embeds a synchronization barrier and thus provides an interesting case of complex service-based workflow.

Input image pairs are taken from a database of injected T1 brain MRIs from the cancer treatment center "Centre Antoine Lacassagne" in Nice, France, courtesy of Dr Pierre-Yves Bondiau. All images are 256×256×60 and coded on 16 bits, thus leading to a 7.8 MB size per image (approximately 2.3 MB when compressed without loss).

9.3 Grid5000 and EGEE infrastructures

In order to evaluate the relevance of our prototype and to compare real executions to theoretically expected results, we made experiments on two different grid infrastructures: the EGEE production grid[4] and the Grid5000 experimental platform[5]. Grids are novel and complex systems that are difficult to optimally exploit from the end users point of view as their behavior is not very well known. The Grid5000 and the EGEE infrastructures for instance have different characteristics leading to different performances and behaviors under load. We first propose a modeling of these two infrastructure to better interpret the experimental results.

Grids overview. Table 1 summarizes the main features of both infrastructures, especially considering the *Workload Management System* (WMS) and the *Data Management System* (DMS), both strongly affecting applications.

Grid5000 is made of 9 clusters located in France, representing more than 2000 CPUs. These resources are shared by dozens of registered users. Inside each cluster, the OAR batch scheduler [5] is used as WMS. The inter-sites GridOAR scheduler is not yet available for users. Hundreds of GB of disk space are shared through NFS [39]. We mostly experimented the Grenoble cluster (12 bi-processor nodes) and the Sophia cluster (105 bi-processor nodes).

EGEE is made of more than 180 computing centers distributed all over Europe and beyond. Hundreds of users are using these resources in production mode (24/7 load of the infrastructure). A total of 18000 CPUs are available out of which 3000 CPUs are effectively accessible to our user community. The EGEE WMS is a two levels batch system: each computing center batch system is fed by higher level *Resource Brokers* (RBs) which receive and queue user computing requests before dispatching them to the available centers. A total amount of 5 PB of storage space is available through *Storage Elements* (SEs) on each site. Data transfer between SEs are handled by gridFTP.

Experimental setting. Figure 16 displays our experimental setting. The OAR batch scheduler can receive parallel requests. Our experiments have shown that above 80 parallel connections, OAR is overloaded. To perform load experiments we thus implemented a requests sequencer. The Grid5000 clusters front-end is shared among users. To avoid overloading it, we reported our heavy-weight application on

[4] Enabling Grids for E-sciencE, http://www.eu-egee.org/

[5] Grid5000, http://www.grid5000.org/

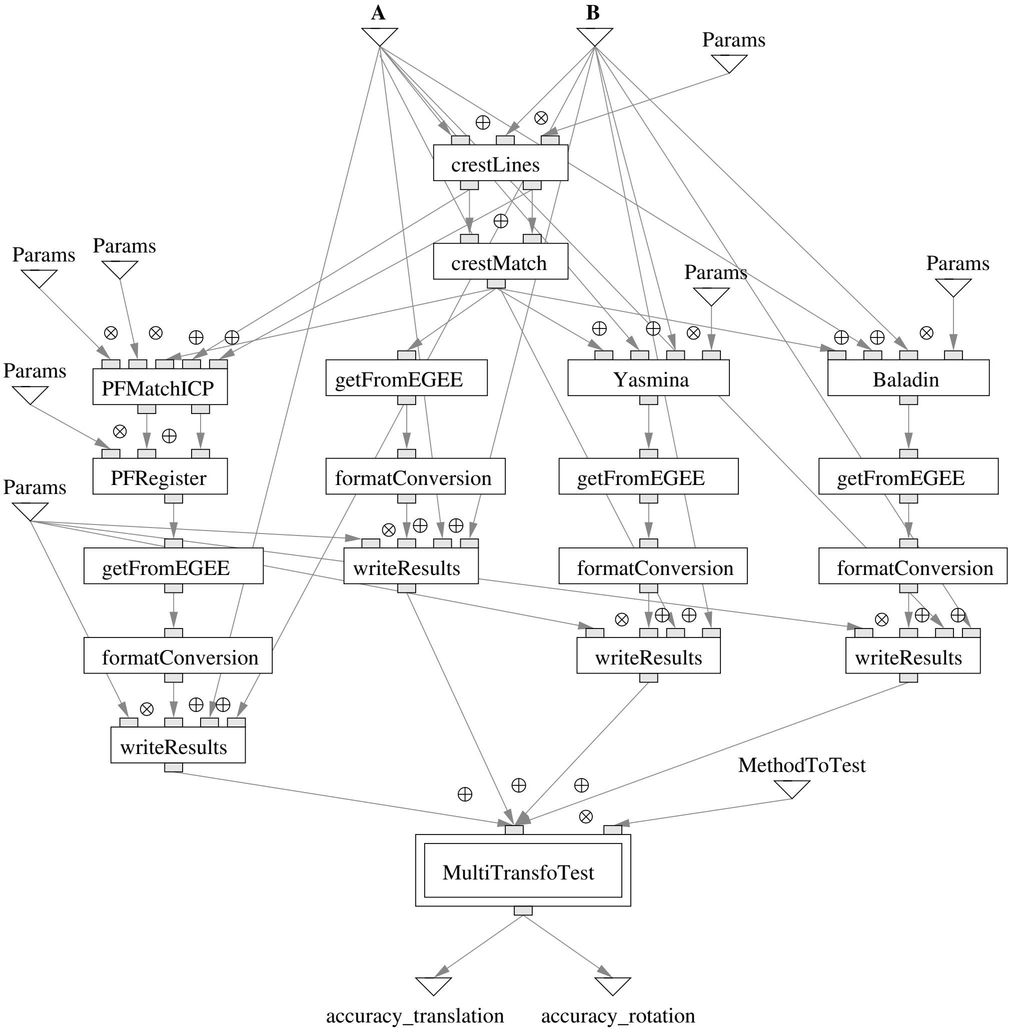


Figure 15: Workflow of the application

Infrastructure	EGEE - LCG2	Grid'5000
Workload Management System	RB	GridOAR
	PBS, BQS, . . .	OAR
CPUs	18,000	1,400
Data access	gridFTP	NFS
Storage resources	couple of PB	Hundreds of GB

Table 1: Overview of the systems

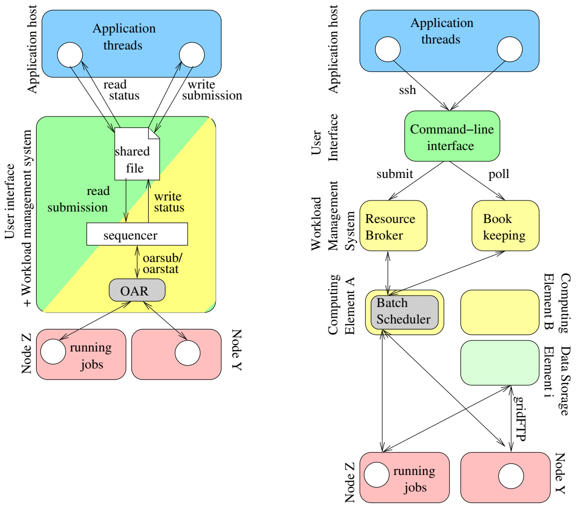


Figure 16: Grid5000 (left) and EGEE (right) system components

a dedicated node. On the EGEE grid, the application code is similarly executed on a dedicated *User Interface* host. Requests are sent and processed sequentially by the RB.

Given its scale and its usage in production mode, the EGEE infrastructure is more likely to be affected by variable load conditions, network interruptions, and temporary resources volatility. As a side effect, there are outliers: jobs that are lost or blocked for a considerable time before being processed. This problem is characteristic of grid infrastructures and cannot be ignored or a single job could stop a very complex computation. Timeouts have to be set up to deal with such outliers. Due to these outliers, we did not compute any means nor standard deviations in the analysis of the experimental results shown below. We used *medians* and *inter-quartile ranges* (IQR) which are less sensitive to outliers instead. The IQR is defined as the interval between the 25% and the 75% lowest values. It corresponds to the range of values measured, centered on the median, after excluding one quarter of low value outliers and one quarter of high value outliers.

9.4 Workload management modeling

While grid infrastructures provide a considerable amount of computing power, the overhead introduced by the WMS when managing large amounts of jobs may cause performances loss. We are studying this overhead by comparing the difference between jobs *execution time* (t_{exec}: the waiting time for the user) and their *running time* (t_{run}: the CPU time consumed). This overhead may be significant on large scale infrastructures, thus penalizing the execution of applications with a high turn-over of jobs to process.

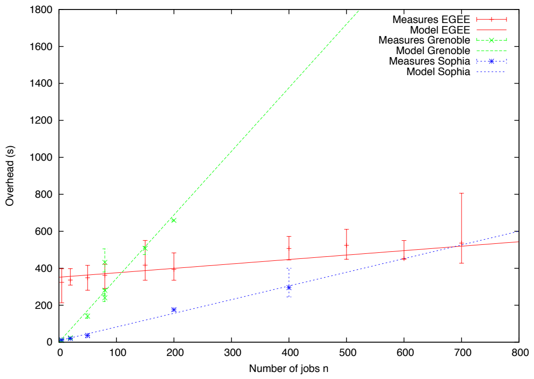


Figure 17: WMS overhead vs number of jobs

Experimental method. We progressively loaded the Grid5000 and the EGEE WMS by submitting an increasing number, n, of short jobs to the system. We resubmitted a new job each time a job completed, so that the total load introduced by the experiment was constant. We considered short (t_{run} =1 minute long) jobs for favoring a short turn-over of jobs and stressing conditions of the WMS. Experiments were run over 3 hours periods (a long enough period compared to the jobs duration to capture the system behavior over a statistically significant number of measurements).

Results. Figure 17 displays the median and the IQR of $t_{\text{over}} = t_{\text{exec}} - t_{\text{run}}$ for a growing number n of submitted jobs. This information measures the spread of the samples and gives an information about the variability of the system. For this experiment, $20,000$ jobs were submitted to the EGEE infrastructure, $32,000$ to the Sophia cluster and $28,000$ to the Grenoble one.

System	A (s/job)	B (s)
Grid5000 – Grenoble	3.44	0.48
Grid5000 – Sophia	0.74	8.25
EGEE	0.24	351.4

Table 2: WMS parameters

Modeling. As the measurements suggest an affine behavior of the median overheads, we fitted a linear model ($A.n + B$) to the experimental data by linear regression. The lines obtained are plotted on figure 17. The parameters of this model are shown in table 2, where the systems are sorted from the smallest one to the largest. These parameters can be used as metrics characterizing the variation of the median of the overhead with respect to the number of jobs for each system. The B parameter measures the *nominal overhead* of the system. It corresponds to the overhead introduced by the system without any load. A measures the *scalability* of the system with respect to the number of jobs. It represents the additional time generated by the submission of 1 extra job to the system.

Discussion. The nominal overheads, B, are growing with the size of the infrastructure. The EGEE system almost has a 6 minutes overhead due to the infrastructure load and the communication costs while the nominal overhead of the Grid5000 clusters is in the order of seconds. Conversely, the scalability of the systems (A metric) is growing with their size. The EGEE overhead due to the submission of a single extra job is 0.24 second while the Grenoble cluster requires an extra 3.44 seconds per job. On all the systems evaluated, submission is done from a single entry point (the user interface) to a central workload manager (OAR or RB host). There is here a bottleneck and serious performance drops can be forecast in the scheduling when the load reaches a critical point. Distributed WMS such as presented in [8] should thus be studied.

It also appears from the IQR bars displayed in figure 16 that the variability of the system response time for the Grid5000 clusters is increasing with the load. On the EGEE production infrastructure, the situation is quite different as the variability is higher, even when considering a low number of jobs due to the concurrent activity of other users. We proposed in [19] a probabilistic framework addressing the problem of large scale systems load.

9.5 Data management performances

Experimental setting. To compare the performances of the data management systems of EGEE and Grid5000 infrastructures, we submitted to the infrastructures a number of jobs doing nothing but transferring 7.8MB files on their execution resource. This corresponds to the size of medical images manipulated in the application of section 9.2. To limit the overhead due to concurrent job submissions, only a few of them (5) were submitted in parallel. Measures were done during 3 hours periods again. The median running time and the IQR are displayed in figure 18.

Results. Median performances of both data management systems are quite similar: the mean speedup of the Grenoble cluster data management system

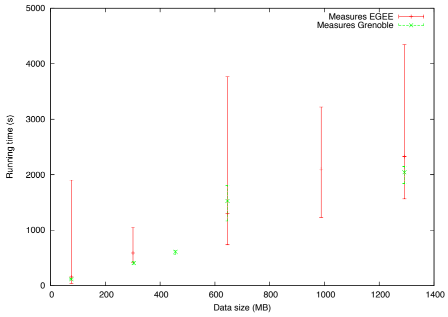


Figure 18: Median running times of data jobs

with respect to the EGEE one is 1.19. This result indicates a good level of performances for the EGEE data management system as this experiment implied inter-clusters transfers, whereas only intra-clusters transfers were performed on Grid5000. However, the variability of the data transfers time on the EGEE infrastructure is far more important than on the Grenoble one, which is not surprising given the scale of the infrastructure.

9.6 Experiments on the bronze standard application

We executed the bronze standard workflow on 3 different inputs data sets, with various sizes, corresponding to the registration of $n_D = 12$, 66 and 126 image pairs corresponding to images from 1, 7 and 25 patients respectively. Each of the input image pair was registered with the 4 algorithms ($n_W = 5$ in this workflow) and leads to 6 job submissions, thus producing a total number of 72, 396 and 756 job submissions respectively. We submitted each dataset 6 times with 6 different optimization configurations in order to identify the specific gain provided by each optimization.

Figure 19 compares the results obtained both on the EGEE infrastructure and the Sophia cluster of Grid5000. MOTEUR is run on the application host of figure 16. Table 3 displays the quantitative values measured on the EGEE infrastructure.

Configurations	**Computation time (s)**		
	12 images	26 images	126 images
NOP	32855	76354	133493
JG	22990	68427	125503
SP	18302	63360	120407
DP	17690	26437	34027
SP+DP	7825	12143	17823
SP+DP+JG	5524	9053	14547

Table 3: Execution time for each configuration

For a given configuration, the execution on the Sophia cluster of Grid5000 is always quicker than on the EGEE system, even for 126 input image pairs. However, we can notice on figure 19 that the graphical representations of the execution times with respect to the size of the input data set size for the EGEE infrastructure are almost straight lines. This could be expected as the infrastructure is large enough to support the increasing load.

The influence of data parallelism can be studied from configurations C_{SP} and $C_{\mathrm{SP+DP}}$. On the Sophia cluster, data parallelism respectively leads to a 6.04, 7.74 and 9.46 speed-ups for 12, 66 and 126 image pairs. On the EGEE infrastructure, corresponding speed-ups are 2.34, 5.22 and 6.76. On both systems, the speed-up introduced by data parallelism is growing with the number of input data sets, which is coherent with the results presented in section 9.4. The influence of service parallelism can be studied from configurations C_{DP} and $C_{\mathrm{SP+DP}}$. On the Sophia cluster, service parallelism respectively leads to a 0.86, 2.9 and 2.86 speed-up for 12, 66 and 126 input image pairs. On the EGEE infrastructure, corresponding speed-ups are 2.26, 2.17 and 1.90.

9.6.1 Discussion

To analyze performances, the first relevant metric from the user point of view is the speed-up, measured as the ratio of the execution time over the reference execution time. We also used the scalability and the nominal overhead metrics, as introduced in section 9.4, which allow a more precise interpretation

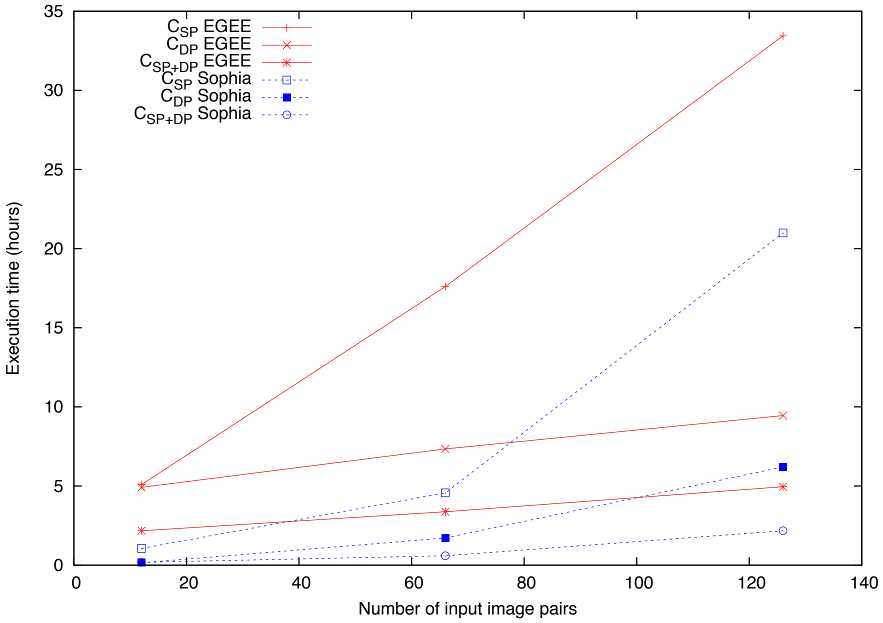


Figure 19: Comparison of EGEE and Grid5000 infrastructures on the bronze standard application

	nominal overhead (seconds)	scalability (s/data sets)
NOP	20784	884
JG	11093	900
SP	6382	897
DP	16328	143
SP+DP	6625	88
SP+DP+JG	4310	79

Table 4: Nominal overhead and scalability for each configuration

of experiments on grid infrastructures.

For each configuration, we reported the nominal overhead and the scalability parameters measured on the EGEE infrastructure in table 4. Those values were obtained by linear regressions on measurements displayed in table 3. We relate our experimental results to the theoretical ones that we presented in section 5.6. As our data set is quite homogeneous (all the images have the same size), we make the hypothesis of constant execution times and thus refer to the results presented in the last paragraph of section 5.6.4 and in particular to S_{DP}, S_{SP} and S_{SDP}.

9.6.2 Impact of the data and service parallelism

DP versus NOP. Given the data-intensive nature of the application, the first level of parallelism to enable to improve performances is data parallelism. In this case, the last paragraph of section 5.6.4 predicted a speed-up $S_D = n_D$. We obtain speed-ups of 1.86, 2.89 and 3.92 for $n_D = 12$, 66 and 126 image pairs respectively. This speed-up is effectively growing with the number of input images as predicted by the theory, although it is lower than expected. Indeed, this experiment shows that the system variability (on transfer and queuing time in particular) and the increasing load of the middleware services on a production infrastructure cannot be neglected.

To go further in the analysis, we can compute from table 4 that in this case, data parallelism leads to a scalability ratio of 6.18 and to a nominal overhead ratio of 1.27. Data parallelism thus mainly influences the scalability ratio. It is coherent as this metric is designed to evaluate the data scalability of the system. Although a higher scalability ratio could be expected on a dedicated system to some extent (until the number of dedicated resources is reached), we can see that in our experiment the grid infrastructure smoothly accepts the increasing load (no saturation effect). This is interesting for applications such as the Bronze Standard that needs the highest number of data to be processed as possible.

9.6.3 (DP + SP) versus DP.

One can notice is that service parallelism does introduce a significant speed-up even if data parallelism is enabled. Indeed, it leads to a speed-up of 2.26, 2.17 and 1.90 for $n_D = 12$, 66 and 126 image pairs respectively whereas the theory predicted a speed-up of $S_{SDP} = 1$. This result can be justified by noticing that the constant times hypothesis may not hold on such a production infrastructure, as already suggested in section 5.6.4. On a traditional cluster infrastructure, service parallelism would be of minor importance whereas it is a very important optimization on the production infrastructure we used.

Moreover, we can then notice that in case of data parallelism, service parallelism leads to a scalability ratio of 1.62 and to a nominal overhead ratio of 2.46. This is another argument which demonstrates that service parallelism is particularly important on production infrastructures. On traditional clusters indeed, nominal overhead values may be close to 0 and such systems would therefore be less impacted by a reduction of this metric.

9.7 Impact of the job grouping

JG vs NOP. The speed-up introduced by job grouping is 1.43, 1.12 and 1.06 for $n_D = 12$, 66 and 126 image pairs respectively. It leads to a scalability ratio of 0.98 and to a nominal overhead ratio of 1.87. Job grouping only influences the nominal overhead ratio. It is coherent because it has been designed to lower the system's overhead which is evaluated by the nominal overhead value.

(JG + SP + DP) vs (SP + DP). In addition to data and service parallelism, job grouping introduces a speed-up of 1.42, 1.34 and 1.23 for $n_D = 12$, 66 and 126 image pairs respectively. It leads to a scalability ratio of 1.11 and to a nominal overhead ratio of 1.54. Here again, job grouping mainly improves the nominal overhead ratio, which is coherent with the expected behavior.

We can thus conclude that job grouping effectively addresses the problem for which it as been designed as it leads to a significant reduction of the system's overhead.

9.8 Optimization perspectives

The nominal overhead and scalability parameters are able to quantify how an application could be improved, without any reference to the scale of the infrastructure. Indeed, an ideal system would have a null scalability ratio and a close to zero overhead.

The nominal overhead value of DP+SP+JG quantifies the potential overhead reduction that could be targeted. In the future, we plan to address this problem by grouping jobs of a single service, thus finding an trade-off between data parallelism and the system's overhead.

Besides, the scalability value of DP+SP+JG quantifies the potential data scaling improvement that could be targeted. On an ever-loaded production infrastructure, middleware services such as the user interface or the resource broker may be critical bottlenecks. The theoretical modeling does not take into account these limitations. A probabilistic modeling considering the variable nature of the grid infrastructure is probably an interesting future path to explore for further optimizing this value [19].

9.9 Experiments on service grouping

To quantify the speed-up introduced by services grouping on a real application workflow, we first made experiments on the bronze standard application. To show how services grouping is able to speed-up the execution on highly sequential applications, we also extracted a sub-workflow from our application, as shown in figure 15. It is made of 4 services that correspond to the `crestLines`, `crestMatch`, `PFMatchICP` and `PFRegister` ones in the application workflow. Our grouping rule groups those 4 services into a single one, as it has been detailed in the example of figure 12. It is important to notice that even if this sub-workflow is sequential, and thus does not benefit from workflow parallelism, its execution on a grid does make sense because of data and service parallelisms.

Results. Table 5 presents the speed-ups induced by our grouping strategy for a growing number of input image pairs. Experiments were lead on the grid5000 infrastructure for the two workflows described above. We can notice on those tables that services grouping does effectively provide a significant speed-up on the workflow execution. This speed-up is ranging from 1.23 to 2.91.

The speed-up values are greater on the sub-workflow than on the whole application one. Indeed, on the sub-workflow, 4 services are grouped into a single one, thus providing a 3 jobs submission saving for each input data set. On the whole application workflow, the grouping rule is applied only twice, leading to a 2 jobs saving for each input data set, as depicted on figure 20.

9.10 Multi-grids model

Grid5000 and EGEE exhibit different behaviors under load. It is therefore interesting to determine, given a number of jobs n to process, the optimal fractions $\delta \in [0, 1]$ and $1 - \delta$ of these jobs that should be submitted to each infrastructure to minimize the total execution time. Let $t^{(i)}_{\text{over}}(n)$ be the median overhead time introduced by system i when it deals with the submission of n concurrent jobs. The goal is to minimize the mean overhead time of the submitted jobs:

$$H(\delta) = \delta.t^{(1)}_{\text{over}}.(\delta.n) + (1 - \delta).t^{(2)}_{\text{over}}.((1 - \delta).n)$$

If we consider the linear model introduced in section 9.4, we get:

$$H(\delta) = \delta(A_1.\delta.n + B_1) + (1 - \delta)(A_2.(1 - \delta).n + B_2)$$

Number of input image pairs	Speed-up on the sub-workflow	Speed-up on the whole application
12	2.91	1.42
66	1.72	1.34
126	2.30	1.23

Table 5: Grouping strategy speed-ups

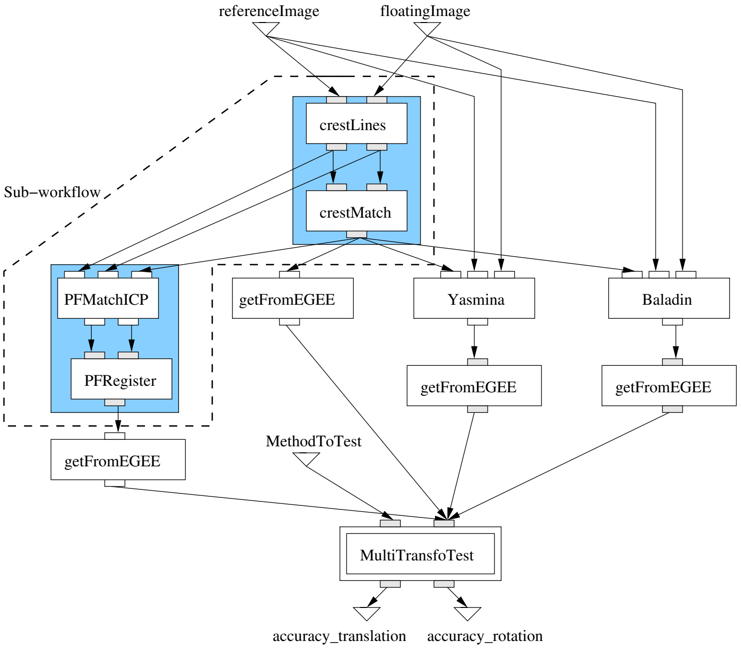


Figure 20: Workflow of the application. Services to be grouped are squared in blue.

Largest system	Smallest system	n_0	$n_{0.5}$	$\delta(\infty)$
EGEE	Sophia	232 jobs	686 jobs	76%
EGEE	Grenoble	51 jobs	110 jobs	93%
Sophia	Grenoble	1 job	3 jobs	82%

Table 6: Multi-grids model parameters

where, A_i and B_i are the model parameters of the i^{th} system. H has a unique minimum reached for the optimal proportion of jobs $\hat{\delta}$ to submit on the first system:

$$\hat{\delta}(n) = \frac{B_2 - B_1 + 2.A_2.n}{2.n.(A_2 + A_1)} \qquad (9)$$

We have to determine when $\hat{\delta}(n)$ is in [0,1]. Suppose that system 1 is larger than system 2. According to section 9.4, it implies that $B_1 > B_2$ (nominal overhead of the largest system is the highest one) and $A_1 < A_2$ (the scalability of the largest system is better). In this setting, it is straightforward to prove that $\hat{\delta}(n) < 1$, showing that the proportion of jobs to submit on the smallest infrastructure is never null (as long as it is not overwhelmed it is faster to exploit it). Moreover, $\hat{\delta}(n) > 0$ if and only if $n \geq n_0 = \frac{B_1 - B_2}{2.A_2}$. Below this threshold, the number of jobs is low enough for all of them to be submitted to the smallest, but fastest, infrastructure. Beyond n_0, the number of jobs to be submitted to the largest infrastructure increases. For $n_{0.5} = \frac{B_1 - B_2}{A_2 - A_1}$, both infrastructures are loaded with the same number of jobs. Beyond, the model enters a saturation phase, where $\hat{\delta}$ tends to its asymptotic value $\hat{\delta}(\infty) = \frac{A_2}{A_1 + A_2}$. This value is inferior to 1 and denotes the remaining proportion of jobs that would always be submitted to the largest platform, even if the number of concurrently submitted jobs becomes very high.

Table 6 displays the different thresholds for the EGEE and grid5000 infrastructures, considering the parameters from table 2. Note that n_0 is high compared to the number of CPUs available on the smaller infrastructures. The $n_{0.5}$ values lead to similar interpretations. It corresponds to the abscissa where the lines cross on figure 17. We thus can see that the EGEE infrastructure and the Sophia cluster lead to the same overhead if 686 jobs are submitted on each infrastructure. This number of jobs is 110 when comparing EGEE to the Grenoble cluster and 3 for the Sophia versus Grenoble comparison. When considering asymptotic behavior, the Sophia cluster should handle $\delta(\infty) = 82\%$ of jobs when used concurrently with the Grenoble cluster due to their difference in size (this result is close to the proportion of nodes on the Sophia cluster in the total number of nodes on the two systems: $\frac{105}{105+12} = 89.7\%$). When comparing EGEE to the Sophia cluster, $\delta(\infty) = 76\%$: it is never efficient to submit more than three quarter of the jobs on EGEE.

10 Conclusions

Grids have a very interesting potential for processing data intensive applications and composing new applications from services wrapping application code. The MOTEUR workflow manager was designed to efficiently exploiting such infrastructures while adopting a modern service-based architecture offering a maximum of flexibility to the users. In particular MOTEUR exploits workflow parallelism, service parallelism and data parallelism. To reduce the grid overhead, MOTEUR is also able to group service calls. A coherent Service-Oriented Architecture eases the implementation of this functionality. Legacy codes and non-service aware codes can easily be scheduled in MOTEUR workflows through the generic web service wrapper. Performances are shown on two different real grid infrastructures using a real application to medical images as a benchmark.

Acknowledgment

This work is partially funded by the French research program "ACI-Masse de données" (http://acimd.labri.fr/), AGIR project (http://www.aci-agir.org/). We are grateful to the EGEE European project and the Grid5000 French national project for providing the grid infrastructures and user assistance.

References

[1] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business Process Execution Language for Web Services. Technical Report version 1.1, http://www-128.ibm.com/developerworks/library/ws-bpel/, May 2003.

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

[3] Hugues Benoit-Cattin, Fabrice Bellet, Johan Montagnat, and Christophe Odet. Magnetic Resonance Imaging (MRI) Simulation on a Grid Computing Architecture. In *Biogrid'03, proceedings of the IEEE CCGrid03 (Biogrid'03)*, may 2003.

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

[5] Nicolas Capit, Georges Da Costa, Yiannis Georgiou, Guillaume Huard, and Cyrille Marti. A batch scheduler with high level components. In *Cluster computing and Grid 2005 (CCGrid'05)*, 2005.

[6] Franck Cappello, Frdric Desprez, Michel Dayde, Emmanuel Jeannot, Yvon Jegou, Stphane Lanteri, Nouredine Melab, Raymond Namyst, Pascale Vicat-Blanc Primet, Olivier Richard, Eddy Caron, Julien Leduc, and Guillaume Mornet. 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, nov 2005.

[7] Eddy Caron, Bruno Del-Fabbro, Frdric Desprez, Emmanuel Jeannot, and Jean-Marc Nicod. Managing Data Persistence in Network Enabled Servers. *Scientific Programming Journal*, 2005.

[8] Eddy Caron and Frdric Desprez. DIET: A Scalable Toolbox to Build Network Enabled Servers on the Grid. *International Journal of High Performance Computing Applications*, 2005.

[9] Eddy Caron, Frdric Desprez, Frdric Lombard, Jean-Marc Nicod, Martin Quinson, and Frdric Suter. A Scalable Approach to Network Enabled Servers. In *8th International EuroPar Conference*, volume 2400 of *LNCS*, pages 907–910, Paderborn, Germany, aug 2002. Springer-Verlag.

[10] Henri Casanova, Arnaud Legrand, Dmitrii Zagorodnov, and Francine Berman. Heuristics for Scheduling Parameter Sweep Applications in Grid Environments. In *9th Heterogeneous Computing Workshop (HCW)*, pages 349–363, Cancun, may 2000.

[11] David Churches, B. S. Sathyaprakash, Matthew Shields, Ian Taylor, and Ian Wand. A Parallel Implementation of the Inspiral Search Algorithm using Triana. In *Proceedings of the UK e-Science All Hands Meeting*, Nottingham, UK, sep 2003.

[12] Thierry Delaitre, Tams Kiss, Ariel Goyeneche, G Terstyanszky, S Winter, and Pter Kacsuk. GEMLCA: Running Legacy Code Applications as Grid Services. *Journal of Grid Computing (JGC)*, 3(1-2), 2005.

[13] European IST project of the FP6, Enabling Grids for E-sciencE, apr. 2004-mar. 2006. http://www.eu-egee.org/.

[14] Ian Foster. 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, 2005.

[15] Ian Foster, Carl Kesselman, J Nick, and S Tuecke. The Physiology of the Grid: An Open

Grid Services Architecture for Distributed Systems Integration. Technical report, Open Grid Service Infrastructure WG, GGF, jun 2002.

[16] N Furmento, A Mayer, S McGough, S Newhouse, T Field, and J Darlington. ICENI : Optimisation of component applications within a Grid environment. *Journal of Parallel Computing*, 28(12):1753–1772, 2002.

[17] Tristan Glatard, Johan Montagnat, and Xavier Pennec. An optimized workflow enactor for data-intensive grid applications. Technical Report I3S/RR-2005-32-, I3S, Sophia-Antipolis, oct 2005.

[18] Tristan Glatard, Johan Montagnat, and Xavier Pennec. Grid-enabled workflows for data intensive medical applications. In *18th IEEE International Symposium on Computer-Based Medical Systems (ISCBMS)*, jun 2005.

[19] Tristan Glatard, Johan Montagnat, and Xavier Pennec. Probabilistic and dynamic optimization of job partitioning on a grid infrastructure. In *14th euromicro conference on Parallel, Distributed and network-based Processing (PDP06)*, Montbliard-Sochaux, feb 2006.

[20] gLite middleware. http://www.gLite.org.

[21] Andrew Harrison and Ian Taylor. Dynamic Web Service Deployment Using WSPeer. In *Proceedings of 13th Annual Mardi Gras Conference - Frontiers of Grid Applications and Technologies*, pages 11–16, feb 2005.

[22] Yan Huang, Ian Taylor, David M. Walker, and Robert Davies. Wrapping Legacy Codes for Grid-Based Applications. In *17th International Parallel and Distributed Processing Symposium (IPDPS)*, page 139. IEEE Computer Society, 2003.

[23] Romin Irani and S Jeelani Bashna. *AXIS: Next Generation Java SOAP*. Wrox Press, may 2002.

[24] P. Jannin, J.M. Fitzpatrick, D.J. Hawkes, X. Pennec, R. Shahidi, and M.W. Vannier. Validation of medical image processing in image-guided therapy. *IEEE Trans. on Medical Imaging*, 21(12):1445–1449, December 2002.

[25] Pter Kacsuk, Gbor Dzsa, Jzsef Kovcs, Rbert Lovas, Norbert Podhorszki, Zoltn Balaton, and Gabor Gombs. P-GRADE: A Grid Programing Environment. *Journal of Grid Computing (JGC)*, 1(2):171–197, 2003.

[26] Pter Kacsuk, Ariel Goyeneche, Thierry Delaitre, Tams Kiss, Zoltn Farkas, and Tams Boczko. 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, 2004. IEEE Computer Society.

[27] G Kecskemeti, Y Zetuny, Tams Kiss, G Sipos, Pter Kacsuk, G Terstyanszky, and S Winter. Automatic deployment of Interoperable Legacy Code Services. In *UK e-Science All Hands Meeting*, Nottingham, UK, sep 2005.

[28] LCG2 middleware. http://lcg.web.cern.ch/LCG/activities/-middleware.html.

[29] Jianzhi Li, Zhuopeng Zhang, and Hongji Yang. A Grid Oriented Approach to Reusing Legacy Code in ICENI Framework. In *IEEE International Conference on Information Reuse and Integration (IRI'05)*, Las Vegas, USA, aug 2005.

[30] Phillip Lord, Pinar Alper, Chris Wroe, and Carole Goble. Feta: A light-weight architecture for user oriented semantic service discovery. In *European Semantic Web Conference*, 2005.

[31] Bertram Ludscher, Ikay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A. Lee, Jing Tao, and Yang Zhao. Scientific Workflow Management and the Kepler System. *Concurrency and Computation: Practice &amp; Experience*, 2005.

[32] Johan Montagnat, Fabrice Bellet, Hugues Benoit-Cattin, Vincent Breton, Lionel Brunie, Hector Duque, Yannick Legr, Isabelle Magnin, Lydia Maigne, Serge Miguet, Jean-Marc Pierson, Ludwig Seitz, and T Tweed. Medical images simulation, storage, and processing on the european datagrid testbed. *Journal of Grid Computing (JGC)*, 2(4):387–400, dec 2004.

[33] Hidemoto Nakada, Satoshi Matsuoka, K Seymour, J Dongarra, C Lee, and Henri Casanova. A GridRPC Model and API for End-User Applications. Technical report, Global Grid Forum (GGF), jul 2005.

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

[35] Stphane Nicolau, Xavier Pennec, Luc Soler, and Nicholas Ayache. Evaluation of a New 3D/2D Registration Criterion for Liver Radio-Frequencies Guided by Augmented Reality. In *International Symposium on Surgery Simulation and Soft Tissue Modeling (IS4TM'03)*, volume 2673 of *LNCS*, pages 270–283, Juan-les-Pins, 2003. INRIA Sophia Antipolis, Springer-Verlag.

[36] Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew R. Pocock, Anil Wipat, and Peter Li. Taverna: A tool for the composition and enactment of bioinformatics workflows. *Bioinformatics journal*, 17(20):3045–3054, 2004.

[37] Open Science Grid (OSG). http://www.opensciencegrid.org.

[38] Xavier Pennec, R. G. Guttman, and J.-P. Thirion. Feature-Based Registration of Medical Images: Estimation and Validation of the Pose Accuracy. In *Medical Image Computing and Computer-Assisted Intervention (MICCAI98)*, volume 1496 of *LNCS*, pages 1107–1114, Cambridge, USA, oct 1998. Springer.

[39] R Sandberg, D Goldberg, S Kleiman, D Walsh, and B Lyon. Design and Implementation of the Sun Network File System. In *USENIX Conference*, Berkeley, CA, 1985.

[40] Martin Senger, Peter Rice, and Tom Oinn. Soaplab - a unified Sesame door to analysis tool. In *UK e-Science All Hands Meeting*, pages 509–513, Nottingham, sep 2003.

[41] Yoshio Tanaka, Hidemoto Nakada, Satoshi Sekiguchi, Toyotaro Suzumura, and Satoshi Matsuoka. Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing. *Journal of Grid Computing (JGC)*, 1(1):41–51, 2003.

[42] Ian Taylor, Matthew Shields, Ian Wand, and Roger Philp. Grid Enabling Applications Using Triana. In *Workshop on Grid Applications and Programming Tools ()*. Held in Conjunction with GGF8, 2003.

[43] Ian Taylor, Ian Wand, Matthew Shields, and Shalil Majithia. Distributed computing with Triana on the Grid. *Concurrency and Computation: Practice &amp; Experience*, 17(1–18), 2005.

[44] Robert A. Van Engelen and Kyle A. Gallivan. 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 (CCGRID '02)*, page 128, Washington, DC, USA, 2002. IEEE Computer Society.

[45] (W3C) World Wide Web Consortium. Web Services Description Language (WSDL) 1.1, mar 2001. http://www.w3.org/TR/wsdl.

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

[47] Jun Zhao, Carole Goble, Robert Stevens, Dennis Quan, and Mark Greenwood. Using Semantic Web Technologies for Representing e-Science Provenance. In *Third International Semantic Web Conference (ISWC2004)*, pages 92–106, Hiroshima, nov 2004.