Workflow Level Parametric Study Support by MOTEUR and the P-GRADE Portal

Tristan Glatard, Gergely Sipos, Johan Montagnat, Zoltan Farkas, Péter Kacsuk

18

Workflow Level Parametric Study Support by MOTEUR and the P-GRADE Portal

Tristan Glatard, Gergely Sipos, Johan Montagnat, Zoltan Farkas, and Peter Kacsuk

18.1 Introduction

Many large-scale scientific applications require the processing of complete data sets made of individual data segments that can be manipulated independently following a single analysis procedure. Workflow managers have been designed for describing and controlling such complex application control flows. However, when considering very data-intensive applications, there is a large potential parallelism that should be properly exploited to ensure efficient processing. Distributed systems such as Grid infrastructures are promising for handling the load resulting from parallel data analysis and manipulation. Workflow managers can help in exploiting the infrastructure parallelism, given that they are able to handle the data flow resulting from the application's execution.

To handle users' processing requests, two main strategies have been proposed and implemented in Grid middleware: the *task-based* approach, where a computing task is formally described before being submitted; and the *service-based* approach, where a computation handled by an external service is invoked through a standard interface. Both approaches have led to the design of different workflow managers. They significantly differ

- in the way data flows are described and manipulated; and
- regarding the optimizations that can be achieved for executing the workflows.

In particular, in the context of scientific applications, it is often necessary to run experiments following a single workflow but considering different, and sometimes dynamic, input data sets. We will name as *parametric applications* such data-intensive scientific procedures to underline the variable nature of their data flows. Workflow managers are expected to offer both

- a high level of flexibility in order to enable *parametric studies* based on these applications; and
- a Grid interface and workflow optimization strategies in order to ensure efficient processing.

In Section 18.2, we introduce the task-based and the service-based approaches in more detail. We then study their differences in terms of managing the resulting data flows (Section 18.3) and computation flows (Section 18.4). In Section 18.7, we introduce P-GRADE portal, a generic interface to both approaches. P-GRADE portal is able to use both the task-based DAGMan and the service-based MOTEUR [209, 238] (hoMe-made OpTimisEd scUfl enactoR) workflow managers. It conciliates to both approaches as much as possible (Section 18.5), and it offers a single interface to describe a data-intensive workflow. The execution technique to be used can then be selected by the user.

18.2 Task-Based and Service-Based Workflows

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 [194], LCG2 [282], or gLite [211] middleware for instance, has already been used for decades in batch computing. It straightforwardly enables legacy code execution without requiring any modification, provided that the user knows the command line of the code to be launched. An emblematic workflow manager using the task-based framework is the directed acyclic graph manager (DAGMan [151]) from Condor (see Chapter 22) and other frameworks (e.g., VDS), are built on top of this (see Chapters 17 and 23 for instance).

The *service-based* strategy, also referred to as *meta computing*, consists of wrapping application codes into standard interfaces. Such services are seen as black boxes from the workflow manager, for which only the invocation interface is known. Various interfaces, such as Web services [467] (also see Chapter 12) or GridRPC [331], have been standardized. The services paradigm has been widely adopted by middleware developers for the high level of flexibility that it offers (e.g. in the Open Grid Service Architecture [196] and the WS-RF extension to Web services). However, this approach is less common for application code, as it requires all codes to be instrumented with the common service interface. Yet, the service-based approach has been adopted in well-known workflow managers such as the Kepler system [302], Taverna (see Chapter 19), Triana (see Chapter 20), and MOTEUR.

The main difference between the task-based and the service-based approaches is the way data sets to be processed are being handled. In the task-based approach, input data segments are specified with each task. This representation mixes data and processing descriptions. The dependency between two tasks is explicitly stated as a data dependency in these two task descriptions. This representation is static and convenient for optimizing the corresponding computations: The full oriented graph of tasks is known when the computations are scheduled, thus enabling many optimization opportuni-

ties for the workflow scheduler [109]. Conversely, the service-based approach decouples data and processing. Input data sets are dynamically specified at execution time as input parameters to the workflow manager. Each service is defined independently from the data sets to be processed, and it is only at the service invocation time that input data segments are sent to the service. This eases the reexecution of application workflows on different input data. In this framework, the dependencies between consequent services are logically defined at the level of the workflow manager. Each service is designed independently of the others.

18.3 Describing Parametric Application Workflows

18.3.1 Dynamic Data Sets

The nonstatic nature of data descriptions in the service-based approach enables dynamic extensions of the data sets to be processed: A workflow can be defined and executed even though the complete input data sets are not known in advance, perhaps because the data segments are being dynamically fed in as they are produced. Indeed, it is common in scientific applications that data acquisition is a heavyweight process and that data are progressively produced. Some workflows may even act on the data production source itself, stopping data production when sufficient inputs are available to produce meaningful results.

Due to the dynamic nature of data and data interdependencies, it is not always possible to define loops and therefore task-based workflows are typically represented using directed and acyclic graphs (DAGs). Only in the case where the number of iterations is statically known may a loop be expressed by unfolding it in the DAG. However, if the loop condition is dynamically determined (e.g. in optimization loops, which are very frequent in scientific applications), the task-based approach cannot be used. In a workflow of services, loops may exist since the circular dependence on the data segments is not explicitly stated in the graph of services. This enables the implementation of more complex control structures.

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 18.3.2. The data composition patterns and their combinations offer a very powerful tool for describing the complex data-processing scenarios needed in scientific applications. For the users, this means the ability to describe and schedule very complex processing in an elegant and compact framework.

18.3.2 Data Composition Patterns

A very important feature associated with the service-based approach for describing scientific applications is the ability to define different data composi-

tion strategies over the input data set of a service. When a service owns two or more input ports, a data composition strategy describes how the data segments received on the inputs are combined prior to service invocation. There are two main composition strategies illustrated in Figure 18.1.

Let us consider two input data sets, $\mathbf{A} = \{A_0, A_1, \ldots, A_n\}$ and $\mathbf{B} = \{B_0, B_1, \ldots, B_m\}$, as an example. The most common data composition pattern is a *one-to-one* association of the input data segments (A_0 is being processed with B_0, A_1 with $B_1, \ldots$) as illustrated in left of Figure 18.1. It results in the invocation of the service $\min(n, m)$ times (usually, $m = n$ in this context) and the production of as many results. Another common strategy is an *all-to-all* composition, illustrated on the right in Figure 18.1, where each data segment in the first set is processed with all data segments in the second set. It results in $m \times n$ service invocations. We will denote by $\mathbf{A} \oplus \mathbf{B}$ and $\mathbf{A} \otimes \mathbf{B}$ the one-to-one and the all-to-all compositions of data sets $\mathbf{A}$ and $\mathbf{B}$.

Many other strategies could be implemented, but these two are the most commonly encountered and are sufficient for implementing most applications. The consideration of binary composition strategies only is not a limitation, as several strategies may be used pairwise for describing the data composition pattern of a service with more than two inputs.

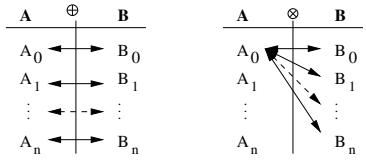

Fig. 18.1: *One-to-one* (left) and *all-to-all* (right) composition strategies.

18.3.3 Data Synchronization Barriers

Some special workflow services require the complete data set (not just one data segment) to perform their computation. This is the case for many statistical operations computed on the data sets, such as the calculation of a mean or a standard deviation over the produced results, for instance. Such services are introducing *data synchronization* in the workflow execution, as they represent real barriers, waiting for all input data to be processed before being executed. They can be easily integrated into workflows of services. The workflow manager will take care of invoking the service only once, as soon as all input data sets are available.

18.3.4 Generating Parametric Workflows

The expressiveness of the application description language has consequences for the kind of applications that can be described. Using composition strategies to design complex data interaction patterns is a very powerful tool for

data-intensive application developers. In the task-based framework, two input data segments, even when processed by the same algorithm, result in the definition of two independent tasks. This becomes very tedious and quickly even humanly intractable when considering the very large data sets to be processed (the all-to-all compositions may produce a considerable number of tasks). Additional higher-level tools are needed to automatically produce the huge resulting DAGs, such as the P-GRADE portal (see Section 18.7).

Workflows of services easily handle the description of input data sets independently from the workflow topology itself. Adding extra inputs or considering parametric inputs does not result in any additional complexity. For instance, the Scufl description language from the Taverna workbench (see Chapter 19) can define one-to-one and all-to-all compositions (known as *dot product* and *cross product* iteration strategies). The service-based approach offers the maximum flexibility when dealing with dynamically extensible data sets.

18.4 Efficient Execution of Data-Intensive Workflows

When considering Grid infrastructures with a large potential for parallelism and optimization in data-intensive applications, efficiency needs to be taken into account to avoid performance drops. Although very convenient for representing workflows independently from data sets to be processed, the service-based approach introduces an extra layer between the workflow manager and the execution infrastructure that hides one from the other [210]. The workflow manager does not directly control the execution of computing tasks to a target infrastructure but delegates this role to the services, which are seen as black boxes. The infrastructure used and the way processings are handled are fully dependent on the service implementation.

Many solutions have been proposed in the task-based paradigm to optimize the scheduling of an application in distributed environments [134]. Concerning workflow-based applications, previous works [109] propose specific heuristics to optimize the resource allocation of a complete workflow. Even if it provides remarkable results, this kind of solution 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 DIET middleware [132] and comparable approaches [88, 419] introduce specific strategies such as hierarchical scheduling. In [131], for instance, the authors describe a way to handle file persistence in distributed environments, which leads to strong performance improvements. However, these works focus on middleware design and do not include yat any workflow management. Moreover, those solutions require that specific middleware components be deployed

on the target infrastructure. Hence, there is a strong need for precisely identifying generic optimization solutions that apply to service-based workflows.

In the following sections, we explore different levels of parallelism that can be exploited for optimizing workflow execution in a service-based approach, thus offering the flexibility of services and the efficiency of tasks. We describe them and study their theoretical impact on performance with respect to the characteristics of the application considered.

18.4.1 Asynchronous Calls

To enable parallelism during the workflow execution, multiple application tasks or services have to be called concurrently. In the task-based approach, this means that the workflow manager should be able to concurrently submit jobs, as is commonly the case (e.g. in DAGMan). In workflows of services, this means that calls made from the workflow manager to the application services need to be non-blocking. GridRPC services may be called asynchronously, as defined in the standard [331]. 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 [249, 449] provide any asynchronous service calls for now. As a consequence, asynchronous calls to Web services need to be implemented at the workflow manager level by spawning independent system threads for each service being executed.

18.4.2 Workflow Parallelism

Given that asynchronous calls are possible, the first level of parallelism that can be exploited is the intrinsic workflow parallelism depending on the graph topology. For instance, if we consider the meteorological application workflow that is presented in Figure 18.2, services `cummu`, `visib`, and `satel` may be executed in parallel. This optimization is usually implemented in all workflow managers.

18.4.3 Data Parallelism

When considering data-intensive applications, several input data sets need to be processed independently using a given workflow. Benefiting from the large number of resources available in a Grid, the same workflow service can be instantiated multiple times on different hardware resources to concurrently process different data segments. 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 the case of a data parallel execution, a data segment is able to overtake another one during the processing, and this could lead to a causality problem. To properly tackle this problem, data provenance has to be monitored during the data parallel execution.

Consider the simple subworkflow made of three services and extracted from a meteorological application (Figure 18.2). Suppose that we want to execute this workflow on three independent input data sets D_0, D_1, and D_2. The data parallel execution diagram of this workflow is represented in Figure 18.3. In this kind of diagram, the abscissa axis represents time. When a data set D_i appears on a row corresponding to a service S_j, it means that D_i is being processed by S_j at the current time. To facilitate legibility, we represented with the D_i notation the data segment resulting from the processing of the initial input data set D_i all along the workflow. For example, it is implicit that on the S_2 service row, D_0 actually denotes the data segment resulting from the processing of the input data segment D_0 by S_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 width. 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.

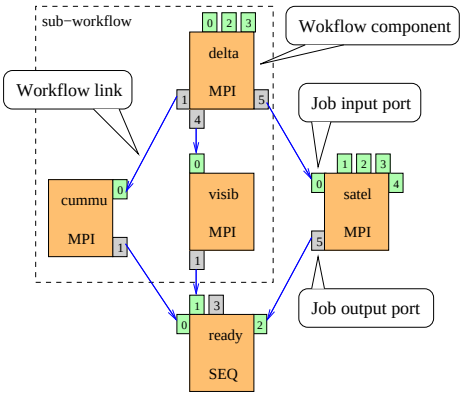

Fig. 18.2: MEANDER nowcast meteorology application workflow.

18.4.4 Service Parallelism

Input data sets are likely to be independent from each other. For example, this is the case when a single workflow is iterated in parallel on many input data sets. *Service parallelism* denotes that the processing of two different data sets by two different services is totally independent. This pipelining model, very successfully exploited inside CPUs, can be adapted to sequential parts of service-based workflows. Consider again the simple subworkflow represented in Figure 18.2, to be executed on the three independent input data sets D_0, D_1, and D_2. Figure 18.3 (right) 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 did not consider data parallelism in this example.

Here again, we show in the next section that service parallelism is of major importance to optimizing the execution of service-based workflows. In task-based workflows, this level of parallelism does not make any sense because it is included in the workflow parallelism. Data synchronization barriers, presented in Section 18.3.3, are of course a limitation to service parallelism. In this case, this level of parallelism cannot be exploited because the input data sets are dependent on each other.

S_1: delta	D_0 D_1 D_2	X
S_2: cummu	X	D_0 D_1 D_2
S_3: visib	X	D_0 D_1 D_2

S_1: delta	D_0	D_1	D_2	X
S_2: cummu	X	D_0	D_1	D_2
S_3: visib	X	D_0	D_1	D_2

Fig. 18.3: Data parallel (left) and service parallel (right) execution diagrams of the sub-workflow extracted from Figure 18.2.

18.4.5 Theoretical Performance Analysis

The data and service parallelisms described above are specific to the service-based workflow approach. To precisely quantify how they influence the application performance 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.

Definitions and Notations

In the workflow, a *path* denotes a set of services 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 ith service of the critical path of the workflow ($i \in [0, n_W - 1]$). Similarly, j denotes the index of the jth 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 $\sigma_{0,0} = T_{0,0} > 0$. $\Sigma = \max_{j<n_D}(\sigma_{n_W-1,j})$ denotes the total execution time of the workflow.

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 when workflows include algorithms that contain optimization loops whose convergence time is likely to vary in a complex way with respect to the nature of the input data set.

Data parallelism is assumed not to be limited by infrastructure constraints. We justify this hypothesis by 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 service. Workflows containing synchronization barriers may be analyzed as two subworkflows corresponding to the parts of the initial workflow preceding and succeeding the synchronization barrier.

Execution Time Modeling

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

$$\text{Sequential case (no parallelism) :} \quad \Sigma = \sum_{i<n_W} \sum_{j<n_D} T_{i,j},$$

$$\text{Case DP, data parallelism only :} \quad \Sigma_{DP} = \sum_{i<n_W} \max_{j<n_D} \{T_{i,j}\},$$

$$\text{Case SP, service parallelism only :} \quad \Sigma_{SP} = T_{n_W-1,n_D-1} + m_{n_W-1,n_D-1},$$

$$\text{with} \begin{cases} \forall i \neq 0, \forall j \neq 0, m_{i,j} = \max(T_{i-1,j} + m_{i-1,j}, T_{i,j-1} + m_{i,j-1}) \\ m_{0,j} = \sum_{k<j} T_{0,k} \text{ and } m_{i,0} = \sum_{k<i} T_{k,0}, \end{cases}$$

$$\text{Case DSP, data + service parallelism :} \quad \Sigma_{DSP} = \max_{j<n_D} \left\{ \sum_{i<n_W} T_{i,j} \right\}.$$

All the expressions of the execution time above can easily be shown recursively. Here is an example of such a proof for Σ_{SP}. We first can write that, for a service-parallel but not data-parallel execution:

$$\forall i \neq 0, \forall j \neq 0, \sigma_{i,j} = T_{i,j} + \max(\sigma_{i-1,j}, \sigma_{i,j-1}). \tag{18.1}$$

Indeed, without data parallelism, data sets are processed one by one and service i has to wait for data segment $j - 1$ to be processed by service i

before starting to process the data segment j. This expression is illustrated by the two configurations displayed in Figure 18.4. We, moreover, note that (i) service 0 is never idle until the last data set has been processed and (ii) D_0 is sequentially processed by all services. Thus

$$\sigma_{0,j} = \sum_{k \leq j} T_{0,k} \quad \text{and} \quad \sigma_{i,0} = \sum_{k \leq i} T_{k,0}. \tag{18.2}$$

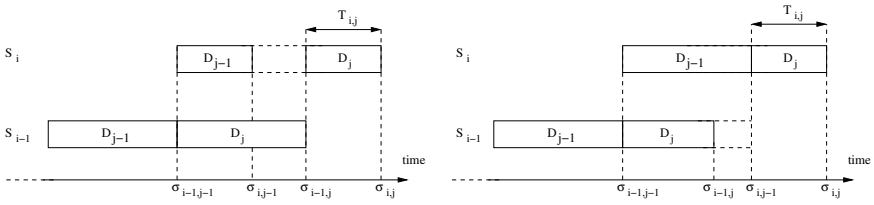

Fig. 18.4: Two different configurations for an execution with service parallelism but no data parallelism.

We can then use the following lemma, whose proof is deferred to the end of the section: $P(i,j) : \sigma_{i,j} = T_{i,j} + m_{i,j}$ with $\forall i \neq 0$ and $\forall j \neq 0, m_{i,j} = \max(T_{i-1,j} + m_{i-1,j}, T_{i,j-1} + m_{i,j-1})$, $m_{0,j} = \sum_{k<j} T_{0,k}$, and $m_{i,0} = \sum_{k<i} T_{k,0}$. Moreover, we can deduce from Equation 18.1 that for every nonnull integer j, $\sigma_{i,j} > \sigma_{i,j-1}$, which implies that $\Sigma_{SP} = \sigma_{n_W-1,n_D-1}$ (by definition of Σ).

Thus, according to the lemma, $\Sigma_{SP} = T_{n_W-1,n_D-1} + m_{n_W-1,n_D-1}$ with $\forall i \neq 0, \forall j \neq 0, m_{i,j} = \max(T_{i-1,j}+m_{i-1,j}, T_{i,j-1}+m_{i,j-1})$, $m_{0,j} = \sum_{k<j} T_{0,k}$, and $m_{i,0} = \sum_{k<i} T_{k,0}$.

The lemma can be shown via a *double recurrence*, first on i and then on j. Recursively, with respect to i:

- $i = 0$: According to Equation 18.2:

$$\forall j < n_D, \quad \sigma_{0,j} = \sum_{k \leq j} T_{0,k} = T_{0,j} + m_{0,j} \text{ with } m_{0,j} = \sum_{k<j} T_{0,k}.$$

  Thus, $\forall j < n_D$, P(0, j) is true.
- Suppose H_i: $\forall j < n_D$, P(i,j) true. We are going to show recursively with respect to j that H_{i+1} is true:
  - $j = 0$: According to Equation 18.2:

$$\sigma_{i+1,0} = \sum_{k \leq i+1} T_{k,0} = T_{i+1,0} + m_{i+1,0} \text{ with } m_{i+1,0} = \sum_{k<i+1} T_{k,0}.$$

  H_{i+1} is thus true for $j = 0$.

– Suppose K_j: H_{i+1} is true for j. We are going to show that K_{j+1} is true.
According to Equation 18.1, $\sigma_{i+1,j+1} = T_{i+1,j+1} + \max(\sigma_{i,j+1}, \sigma_{i+1,j})$.
Thus, according to K_j, $\sigma_{i+1,j+1} = T_{i+1,j+1} + \max(\sigma_{i,j+1}, T_{i+1,j} + m_{i+1,j})$ and according to H_i,

$$\begin{aligned}\sigma_{i+1,j+1} &= T_{i+1,j+1} + \max(T_{i,j+1} + m_{i,j+1}, T_{i+1,j} + m_{i+1,j}) \\ &= T_{i+1,j+1} + m_{i+1,j+1}\end{aligned}$$

with $m_{i+1,j+1} = \max(T_{i,j+1} + m_{i,j+1}, T_{i+1,j} + m_{i+1,j})$.

K_{j+1} is thus true. H_{i+1} is thus true. The lemma is thus true.

Asymptotic Speed-ups

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

Massively data-parallel workflows. Let us consider a massively (*embarrassingly*) data-parallel application (a single service S_0 and a very large number of input data). In this case, $n_W = 1$ and the execution time is

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

In this case, data parallelism leads to a significant speedup. 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 $\Sigma_{DSP} = \Sigma_{DP} = \Sigma_{SP} = \Sigma = \sum_{i<n_W} T_{i,0}$. In this case, neither data nor service parallelism lead to any speedup. Nevertheless, neither of them 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 speedups 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

$$\Sigma = n_D \times n_W \times T, \quad \Sigma_{DP} = \Sigma_{DSP} = n_W \times T, \quad \Sigma_{SP} = (n_D + n_W - 1) \times T.$$

The speedups associated to the different configurations are thus

$$S_{DP} = \frac{\Sigma}{\Sigma_{DP}} = n_D, S_{DSP} = \frac{\Sigma_{SP}}{\Sigma_{DSP}} = \frac{n_D + n_W - 1}{n_W}, S_{SP} = \frac{\Sigma}{\Sigma_{SP}} = \frac{n_D \times n_W}{n_D + n_W - 1}.$$

Service parallelism does not lead to any speedup 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, in practice, even in the case of homogeneous input data sets, T is hardly constant in production systems because of the high variability of the overhead due to submission, scheduling, and queuing times on such large-scale and multiuser platforms. The constant execution time hypothesis does not hold. Figure 18.5

illustrates in a simple example why service parallelism can provide a speedup 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 S_0, and the execution time of the data set D_1 is three times as long as the other ones on service S_1. This 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 performance beyond data parallelism, as it enables some computations to overlap.

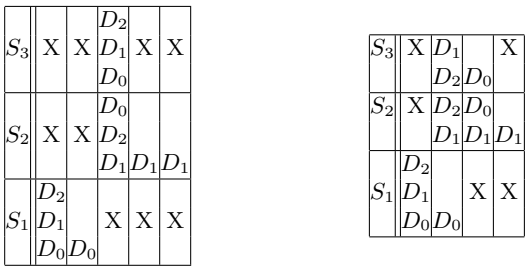

Fig. 18.5: Workflow execution time without (left) and with (right) service parallelism when the execution time is not constant.

18.4.6 Application-Level Parallelism

In addition, an application code may be instrumented to benefit from a parallel execution through a standard library (e.g. MPI). The exploitation of this fine-grain level of parallelism is very dependent on the application code and cannot be controlled at the workflow management level. However, the procedure for submitting parallel tasks is often specific in Grid middleware and the workflow manager needs to recognize the specific nature of such jobs to handle them properly. Usually, application-level parallelism can only be exploited intrasite for performance reasons (intersite communication being too slow), while the other levels of parallelism are coarse-grained and can be exploited intersite.

18.5 Exploiting Both Task- and Service-Based Approaches in Parametric Data-Intensive Applications

To execute parametric and data-intensive applications, two approaches are thus possible:

1. In the task-based approach, a high-level tool for transforming the parametric description of the application into a concrete execution DAG is needed prior to the execution of the workflow manager.

2. In the service-based approach, the separate description of the workflow topology and the input data sets is sufficient. However, the efficient execution relies on an optimized workflow manager capable of exploiting parallelism through parallel service calls.

In the task-based framework, it is not possible to express dynamically expandable data sets and loops. However, parallelism is explicitly stated in the application DAG and easy to exploit. The service-based approach offers more flexibility but requires an optimized application enactor, such as MOTEUR, to efficiently process the workflow, enabling all levels of parallelism described above. In the following sections, we introduce the P-GRADE portal and MOTEUR. P-GRADE conciliates both approaches by providing a unique GUI for describing the application workflow in a high-level framework. P-GRADE is interfaced with both DAGMan, for dealing with task-based workflows, and MOTEUR, for handling workflows of services.

18.6 MOTEUR Service-Based Workflow Enactor

MOTEUR [238] was designed with the idea that the service-based approach is making services and data composition easier from the application developer point of view. It is therefore more convenient, provided that it does not lead to performance losses. The MOTEUR (hoMe-made OpTimisEd scUfl enactoR) workflow manager was implemented to support workflow, data, and service parallelism, described in Section 18.4. Our prototype was implemented in Java in order to be platform independent. It is freely available under CeCILL public license (a GPL-compatible open source license).

The workflow description language adopted is the Simple Conceptual Unified Flow Language (Scufl) used by the Taverna engine (see Chapter 19). Apart from describing the data links between the services, the Scufl language allows one to define *coordination constraints* that are control links enforcing 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. The Scufl language also specifies the number of threads of a service (fixed number of parallel data). In the case of MOTEUR, this number is ignored, as it is dynamically determined during the execution, considering the number of input data segments available for processing. 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 composition strategies presented in Section 18.3 in a service and data parallel workflow is not straightforward because the data sets produced have to be uniquely identified. Indeed, they are likely to be computed in a different order in every service, which could lead to causality problems

and incorrect mapping of the input parameters in one-to-one composition patterns. Moreover, due to service parallelism, several data sets are processed concurrently and one cannot number all the produced data segments once computations are completed. We have implemented a data provenance strategy to sort out the causality problems that may occur. Attached to each processed data is a history tree keeping track of all the intermediate results computed to process it. This tree unambiguously identifies the data segment.

Finally, MOTEUR implements an interface to both Web services and GridRPC instrumented application code. To ease application code wrapping in services and job submissions on a Grid infrastructure, we provide a generic submission Web service. It encapsulates the user code and handles the interface with the Grid infrastructure. It has been interfaced with both the EGEE [180] production Grid infrastructure and the Grid5000 [332] experimental Grid infrastructure.

18.7 P-GRADE Portal

The goal of the P-GRADE portal is to provide a high-level user interface that hides the low-level details of the underlying Grid systems. Users can construct complex Grid applications as workflows without learning the specific Grid interface. Moreover, the P-GRADE portal plays the role of a bridge between different Grids, solving the interoperability problem at the portal level [260]. The components of a workflow can be executed on any Grid that is connected to the portal and for which the user owns an access certificate. P-GRADE portal 2.3 [376] serves as the production portal service for several different Grid systems: VOCE (Virtual Organization Central Europe of EGEE), HunGrid (Hungarian VO of EGEE), EGRID (Economics VO of EGEE), SEE-GRID (South Eastern European Grid), and UK NGS (National Grid Service). If a portal is configured to access all these Grids, then users can access any resource of these Grids from the same workflow.

The portal provides a graphical interface through which users can easily construct workflows based on the DAG concept. Nodes of the graph can be jobs or GEMLCA legacy code services [170]. Arcs among the nodes represent file transfers between the nodes. The workflow enactor of portal version 2.3 is based on DAGMan, which supports only the task-based strategy. Therefore, parametric applications cannot be defined. This portal version supports two levels of parallelism: application parallelism (Section 18.4.6), which is employed when a node of the workflow is an MPI job that is assigned to a multiprocessor Grid site; and workflow parallelism (Section 18.4.2). However, portal version 2.3 is not able to support data and service parallelisms described in Sections 18.4.3 and 18.4.4, respectively.

In order to support the service-based strategy, parametric study applications, and all kinds of parallelism, we extended the portal with two new features:

1. We have extended the workflow creation interface of the portal in order to enable the definition of parametric study applications.
2. We integrated the MOTEUR workflow enactor within the portal in order to support the service-based strategy and to exploit data parallelism and service parallelism.

This new portal version will support the development of DAGs consisting of normal and parametric jobs as well as Web services. It will also support the execution of components of such workflows in Globus-2, Globus-4, LCG-2, gLite, and Web services Grids. While the normal and parametric job components will be executed in Globus-based Grids using DAGMan, Web service invocations will be forwarded to the MOTEUR workflow enactor as illustrated in Figure 18.6.

The current section focuses on the parametric study extension of the portal and shows the workflow user interface that can support both the MOTEUR enactor described in Section 18.6 and the Condor DAGMan-based enactor.

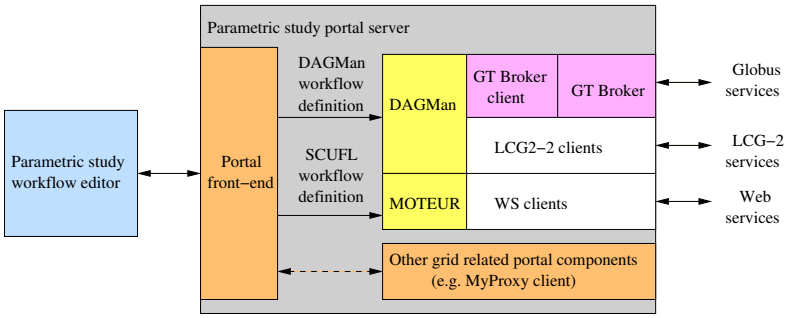

Fig. 18.6: Structure of the parametric study version of the P-GRADE portal.

18.7.1 Interface to Workflow Managers

In order to enable parametric studies, the P-GRADE portal includes the new concept of parametric value. It is based on multiple layers, from high-level graphical definition of the workflows to low-level workflow enactment, as illustrated in Figure 18.7. This architecture enables both the representation of parametric application workflows and the transformation of the abstract workflow into a graph of services or a DAG of tasks as required by the underlying workflow enactors.

At the top of the P-GRADE workflow definition process, *parameter spaces* are defined. Parameter spaces enable the description of parametric values. These parametric values are transformed into data segments corresponding to the *data streams* (application input data sets) that will be handled by the workflow manager. At this layer, there are two possibilities, depending on the user setting: either the input data sets and the services description are sent to

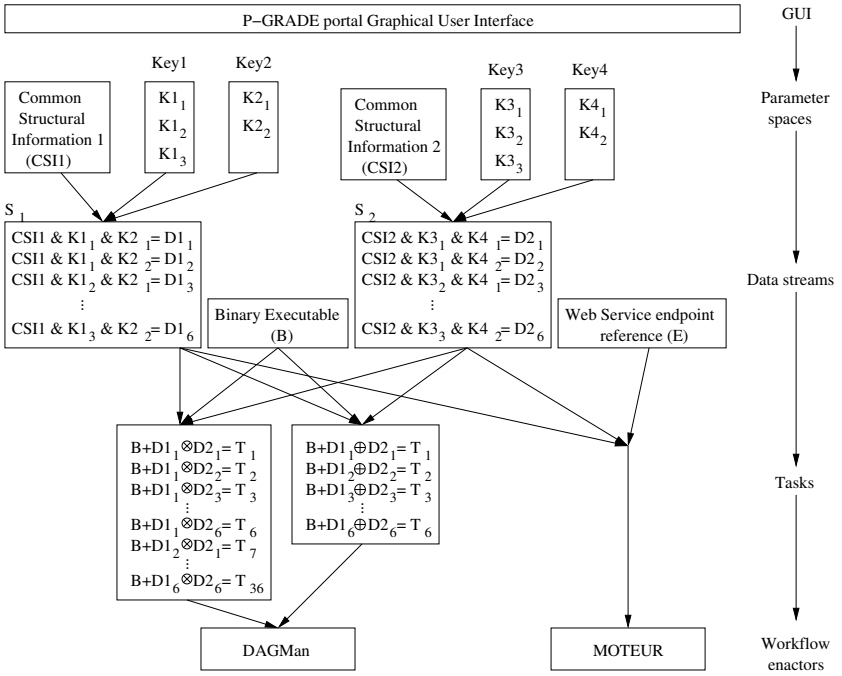

Fig. 18.7: The P-GRADE portal multilayer architecture.

MOTEUR for execution in the service-based framework, or data segments are composed with binary executables according to the data composition patterns to build tasks. The DAG of tasks can then be submitted to DAGMan for workflow enactment in the task-based framework.

The P-GRADE portal defines all elements required for defining such parametric application workflows. It proposes a rich and intuitive GUI for describing the workflow graph, defining parameter spaces, and composing data streams. During workflow execution, the P-GRADE portal handles the interface to the workflow manager, monitors the execution, and provides graphical feedback to the user. Examples of workflows described through the P-GRADE GUI are given in Figures 18.2 and 18.8.

18.7.2 DAGMan Workflow Elements

Figure 18.2 illustrates the workflow elements available in P-GRADE portal version 2.3 to define DAGMan workflows on a real application. They include the following elements:

- *Component.* All components are *normal jobs.* A normal job is a program that has one binary executable file and must be started in batch mode.

The normal job can be either a sequential or an MPI job. The binary executable of the program is taken from the portal user's machine.

- *Port*. Input and output ports can optionally be connected to jobs. *Normal input ports* represent one file to be used by the connected component as input. A *Normal output port* represents one file to be generated by the connected job during execution.
- *Link*. All links in a task-based workflow are *normal file channels*. They define a data channel between a normal output port and a normal input port that represents a transformation of an output file into an input file of a subsequent task.

Based on these elements, a user can create complex workflow applications that can exploit intrasite (MPI) and intersite (workflow) parallelism.

18.7.3 Parametric Workflow Elements

The parametric workflow elements are useful for representing parametric data-intensive applications. In the P-GRADE portal, the same elements are used for specifying parametric task-based or service-based workflows even though they can be executed in different ways. Figure 18.8 displays the new parametric elements.

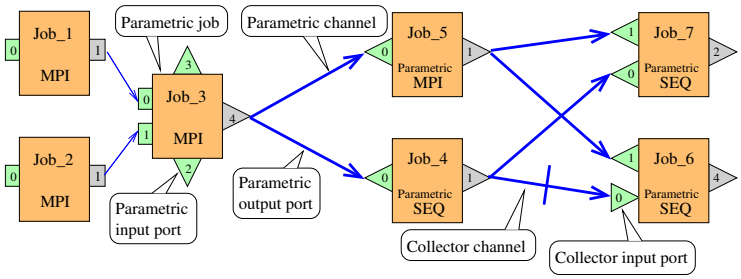

Fig. 18.8: Normal and parametric workflow elements in the P-GRADE portal GUI.

Although represented identically in the GUI, the parametric elements differ in their nature. In particular, parametric job inputs are files, represented through *ports*, while Web service inputs are strings (possibly identifying a file), represented through *fields*. The new workflow elements are:

- *Component*. *Parametric jobs* represent a program that has one binary executable file and must be started in batch mode on independent input file sets. *Parametric Web services* represent one operation of one Web service that must be invoked multiple times with independent input string sets.

Depending on the service implementation, it can submit jobs to a Grid infrastructure when serving the request. Graphically, parametric Web services are identified by the "WS" label, while parametric jobs are labeled "SEQ" or "MPI."

- *Port.* For parametric jobs, *parametric input ports* represent the simultaneously processable instances of a single *file* (files with the same structure but different contents), and *parametric output ports* represent the instances of a single output *file* generated by the instances of a parametric job component (files with the same structure but different contents). Similarly for parametric Web services, *parametric input fields* represent the simultaneously processable instances of an input string, and *parametric output fields* represent the instances of an output string generated by a Web service component.
- *Link. Parametric file* (resp. *parametric string*) *channels* define a data channel between a parametric output and a parametric input port (resp. field). These channels "fire" each time an output data segment becomes available.

In addition, *collector* ports and channels are introduced to represent data synchronization barriers (Section 18.3.3). *Collector input ports* (resp. *fields*) represent N files (resp. strings) with different structures and different contents, which are expected by the connected component as input. They can be connected to both parametric and nonparametric job components through *collector file* (resp. *string*) *channels*. These channels fire only when every output file is available.

Some constraints on the components apply in order to form a semantically correct parametric study workflow application. It makes sense for normal input ports to be connected to a parametric job (every instance of the job is using the same file), while it is not the case for normal output ports. Parametric input ports (resp. fields) can only be connected to parametric job (resp. Web service) components. Parametric jobs (resp. Web services) always have at least one input port (resp. field).

18.7.4 Parameter Spaces and Data Flows

The P-GRADE portal provides a flexible framework for defining variable values of parameters sent to parametric jobs and Web services. The property window of an input parametric port (on the left in Figure 18.9) enables the definition of *keys* (variable values) and *common structural information* (CSI) of the parameters (the common structure of all inputs). The user defines the CSIs for each parameter. A parameter may be n-dimensional, as it may depend on n different input keys $K_1, \ldots, K_n$. The parameter key definition window (on the right in Figure 18.9) enables the definition of a key value generation rule (types of values, values read from files or generated according to different rules, etc.).

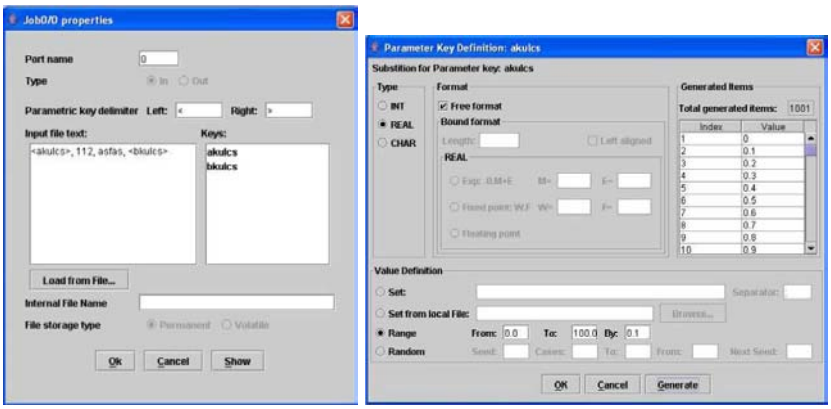

Fig. 18.9: Parameter space definition user interface.

The transformation between a parameter space definition and data streams (see Figure 18.7) is an automatic generation process where every occurrence of a key in the CSI is replaced with a parameter value, according to the algorithm presented in Figure 18.10. This algorithm produces an indexed (ordered) array of data segments `D`. It assumes a precedence order among the keys (primary, secondary...). This precedence order influences the indexing order of data segments. In the P-GRADE portal, the precedence order of keys is the key declaration order. For example, the CSI given in Figure 18.9 (`<akulcs>, 112, asfas, <bkulcs>`) contains two keys (`akulcs` and `bkulcs`). The algorithm will produce the data segments (`0, 112, asfas, 0`), (`0, 112, asfas, 0.1`)...

```
for i = 0 to (K1.length - 1)
  primaryKey = K1[i]
  for j = 0 to (K2.length - 1)
    secondaryKey = K2[j]
    D[i * K1.length + j] = replace(CSI, primaryKey, secondaryKey)
  end
end
```

Fig. 18.10: Parameter generation algorithm.

18.7.5 Workflow Execution

Workflow applications are taken as input sets of data segments ($S_i = Di_j$). In the case of the MOTEUR enactor, the definition of the input data sets

is sufficient to process the workflow. In the case of DAGMan, data streams still need to be composed according to the data composition operators (Section 18.3.2) to produce a list of tasks. The P-GRADE portal interface allows the definition of the one-to-one and the all-to-all data composition strategies on the parametric input data ports (or fields) pairwise. From this input, the data elements, and the job binary, the system generates several computational tasks for each parametric job component (see the tasks layer of Figure 18.7).

Each data segment generated has a unique index value within its set (these values are denoted by the lower indexes in Figure 18.7). The indexes are used by the workflow enactors during workflow execution to determine the order of elements for a one-to-one or all-to-all data composition. Since the computational tasks or the service invocation requests represented by a parametric component are independent from each other, their submission order is irrelevant. Even in the case of a known submission order, the completion time of a task or service is unpredictable. It is the responsibility of the workflow enactment system to keep track of the order of the execution results according to the workflow description.

18.8 Conclusions

Task-based and service-based approaches are two very common frameworks for handling scientific workflows. The service-based approach is very flexible, enabling the expression of complex data composition patterns and dealing with parametric data sets. The task-based approach is more static, but it eases the optimization of the workflow execution since the complete DAG of tasks is known prior to the application execution.

The MOTEUR service-based workflow manager was specifically designed to exploit all levels of parallelism that can be automatically handled by the workflow manager. Using a high-level tool such as the P-GRADE portal, it is possible to describe parametric workflows that will be instantiated either as workflows of services or DAGs of tasks. The P-GRADE portal conciliates the two approaches to some extent, as it automatically produces large DAGs corresponding to data-intensive parametric applications. Yet, the static nature of DAGs does not permit dynamic input data set management, contrary to workflows of services. The P-GRADE portal provides a unique interface for exploiting both approaches. It is relying on MOTEUR and the DAGMan workflow managers to deal with the low-level execution.

18.9 Acknowledgments

The work on MOTEUR is partially funded by the French research program “ACI-Masse de données” (`http://acimd.labri.fr/`), AGIR project (`http://www.aci-agir.org/`). The P-GRADE portal extension work is partially funded by the EU SEEGRID-2 and CoreGrid projects.