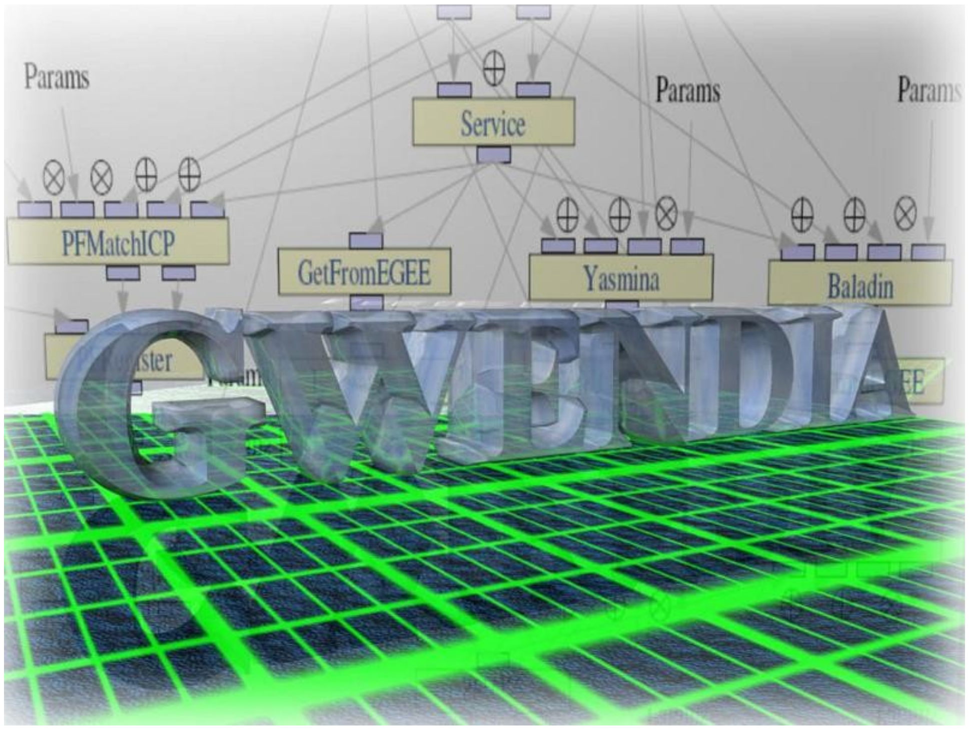
GWENDIA ANR-06-MDCA-009
Grid Workflow Efficient Enactment for Data Intensive Applications

L3.4

Data Management Techniques



Authors : Eddy Caron
Frederic Desprez
Benjamin Isnard
Johan Montagnat

Summary : This document presents different aspects of data management in the context of workflow enactment. It covers both the modelization aspect and the algorithms and techniques used by workflow enactors.

Document layout

1. Introduction

The representation of a data computing application as a workflow necessarily contains a description of the data provided as input and produced as output for each task in the workflow. This language must therefore provide a data model that represents efficiently all kinds of data a workflow task may use as input or output. The expressiveness of this data model is a key element of the workflow language as it gives exactly what is the level of detail available to the workflow enactment engine about the content and structure of the data processed. Using these details allows the engine to optimize the workflow execution using appropriate data management techniques.

Within the context of data-intensive scientific workflows and computational grids, these aspects of workflow language and workflow enactment engine are particularly important as they have a great impact on the performance of the application due to the large number of tasks. The execution of the workflow being distributed over a grid and parallelized, workflow data has to be transferred over the network and very often the same data has to be copied simultaneously on many hosts. If these transfers are not optimized by a data management system then they will be scheduled independently which will generally result in bottlenecks and high latency for workflow tasks.

In the first part of this document we will introduce the data model that is used in the GWENDIA workflow language. Then we will describe the different solutions we have proposed to the problem of data management when designing and implementing the grid workflow enactement engines DIET and MOTEUR, considering the grid execution environment constraints.

2. Workflow data model in the GWENDIA language

This part of the document describes the model used for workflow enactement in the GWENDIA language and introduces some aspects of data management involved in this process of worklow enactement.

2.1. Overview of the workflow model

The GWENDIA workflow language describes a graph of “workflow nodes” where the vertices can belong to one of the following categories:

- computing “activities”: a node describes a service that can be executed on a computing platform (e.g. a grid or cloud). The properties of this node include the data input and output ports with the type of data they receive or send.
- workflow control structures: a node controls the way other workflow nodes are used by the workflow engine. For example a conditional structure (if/then/else) can be used to trigger different activities depending on the value of an expression. These nodes also contain data input/output ports like activities.
- data “sources” and “sinks”: these nodes produce data items (respectively receive them). A data source can be implemented as a file or as a database query. A data sink

can be implemented as a terminal output or as a database insert. These nodes usually define one output port (respectively one input port).

The edges of the graph are oriented, one to one and represent data flows between workflow nodes. More precisely, they interconnects ports of the workflow nodes together (one edge or "link" connects an output port to an input port).

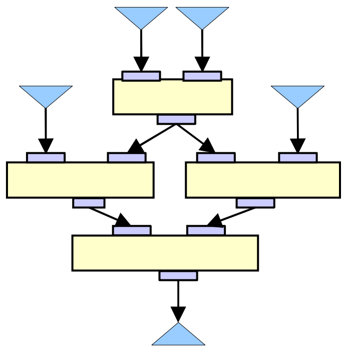


Figure 1. Workflow example

2.2. Role of the data model in workflow enactement

The GWENDIA workflow language adopts a "data-driven" approach which means that the data model determines the execution model of the workflow. This approach is different from control-driven workflow languages that use control structures that are independent from the data model to determine the execution schedule of the workflow.

This approach consists in using the characteristics of the data flows between workflow nodes to determine how the workflow is "instanciated" i.e. how the workflow engine generates a graph which has tasks as vertices and data dependencies (i.e. data "A" is produced by one task and used by another one) as edges. This graph can be generated dynamically and executed simultaneously, or the execution can be handled by a different workflow engine that takes this graph of tasks as input (generally called a "DAG" for "directed acyclic graph"). This engine manages task scheduling and execution.

2.3. Goals

One of the objectives of the data model used for the GWENDIA workflow language comes from the overall motivation of this language which is to simplify the process of creating a data-intensive workflow from the user point of view. This explains the choice for a simple structure for the data model and a relatively intuitive description of data flows within the workflow.

The second objective is to provide enough flexibility and dynamicity to data management in order to have an efficient way to execute the workflow. The main focus concerning data management is to allow data parallelism with an optimized replication of data.

2.4. Data model overview

To provide an expressive data model for data parallel applications, arrays are considered as first class structures in the GWENDIA language. Data items may be scalar values (with a given type τ) or arrays of values. An *Array* is an homogeneous and ordered collection of data items. Each data item is either a scalar of type τ or an *Array*. An array may include any level of nested array data. This structure is therefore defined by its scalar type τ and the *nesting level* of the Array that is the number of nesting levels the structure contains (for example an *Array* containing *Arrays* of atomic values is an *Array* of depth 2). By extension, a scalar value is a particular kind of array with *nesting level* 0.

The atomic type τ can be either a scalar type (integer, string, float, ...), a structured type (e.g. a matrix of scalars) or a file reference. A data of type τ is from the workflow engine perspective an indivisible entity which can be only initialized, copied and transferred.

Conversely, an *Array* is a dynamical structure that contains elements which can be addressed independently by the workflow engine, i.e. each element (either an atomic one or a nested one) can be individually initialized, copied and transferred.

Although the atomic type and the depth of an *Array* are part of the description and must therefore be defined in the workflow, the structure is flexible in size as the number of items at each level of the *Array* structure can vary dynamically during workflow execution or accross different executions of the same workflow. However when the number of items in an Array is known statically i.e. when it does not depend on workflow inputs, it may be useful to provide this information in the workflow to allow the workflow engine to optimize task scheduling.

2.5. Array manipulation during workflow execution

The GWENDIA workflow language uses the data model defined above as the basis to instanciate the workflow. The instanciation is the process used for generating actual service calls to the grid from the combination of the workflow description (written using the GWENDIA language) and an input data set. Each service call or “task” uses some input data and produces some output data which is described using the data model. The role of the workflow engine is to « glue » all the service calls together through re-use of output data as input for another service call.

The data model of the GWENDIA workflow language has been designed (cf §2.3) to allow data parallelism in the typical case of a data *A* which is an *Array* containing many data items that can be processed in parallel by different instances of the same service. Therefore the workflow engine is able to provide a reference to a given element of *A* as input for a service call. The objective is to avoid transferring the whole data *A* when only one element of the Array is to be used by the service.

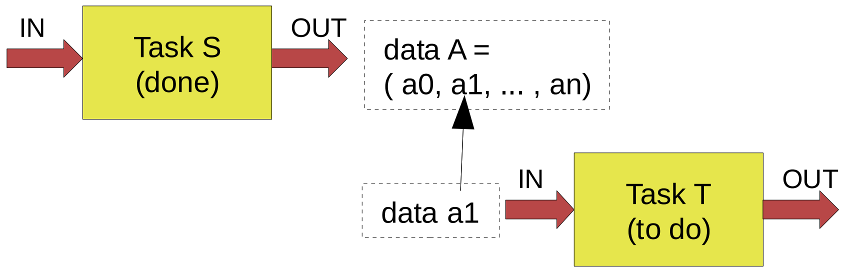


Figure 2. Access to one element of a data Array

Another typical case of data manipulation is the data merge. It is also related to data parallelism but is used when a single service call requires all the results of different instances of another service. Here the workflow engine is able to build a new Array containing all the results as its elements, and provide the reference of this Array as input to the service call. The objective is here to avoid transferring all the results to a central location before calling the service to avoid doubling the cost of communications and creating a network bottleneck.

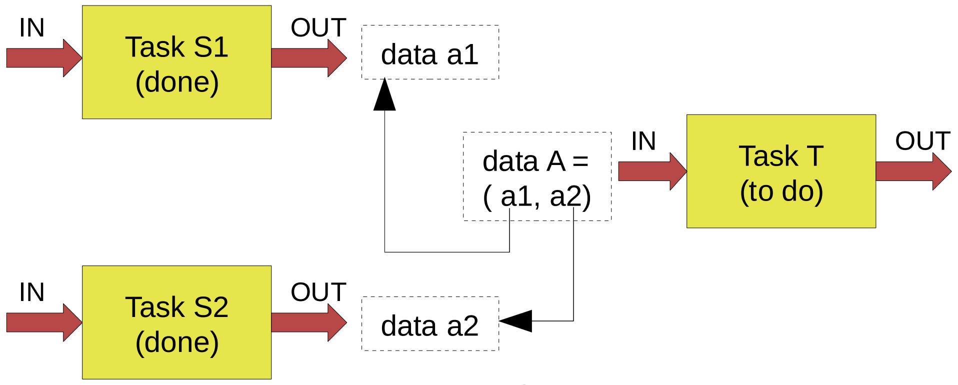


Figure 3. Merge of two data items in an Array

Due to conditional control structures (if/then/else) some data items contained in a given Array may be processed or not depending on their properties. This results in the creation of « void » data items that are elements of the Array produced as output of the workflow. These elements do not refer to actual data but they must be included as elements of the resulting Array because of the ordering and indexing of other elements. The GWENDIA workflow language provides constructs (« filter » and « merge » operators) used to facilitate the removal of these « void » elements from the result. This implies a complete re-indexing of the resulting Array or the combination of elements of two different Arrays.

2.6. Workflow engine architecture

The following proposed architecture for the workflow engine identifies the main components of the workflow engine and tries to group the main dependencies between these components.

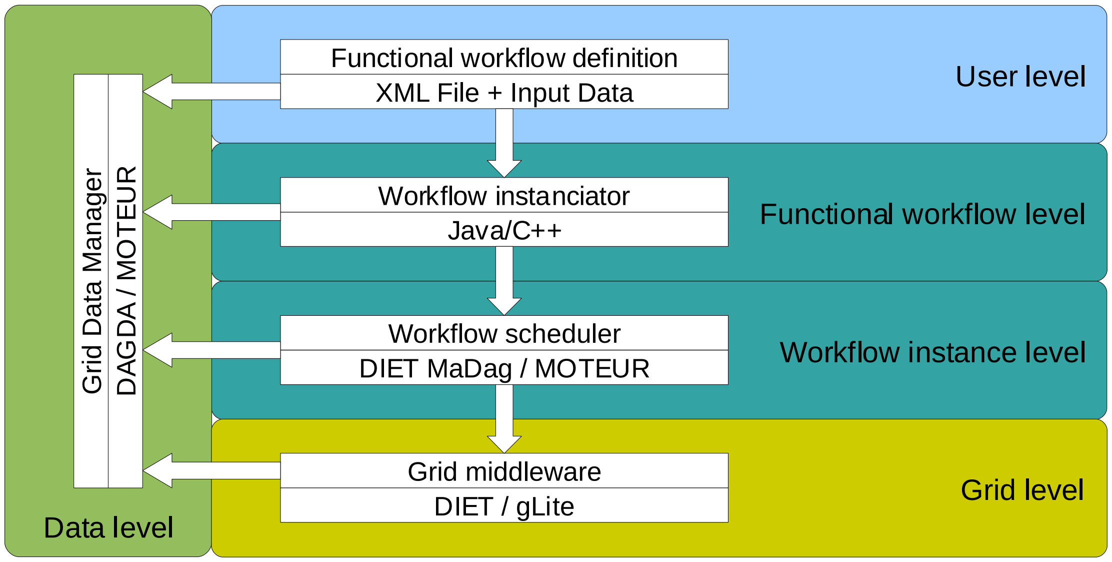


Figure 4. Architecture for GWENDIA language enactement

3. Optimization of data transfers by the workflow manager

For scalability reasons, the workflow manager should only manipulate a limited size data sets. Large experiments will cause large amounts of data items to be considered (possibly millions) and the manager should not saturate the memory of the engine running host nor become a bottleneck of the overall computation plan execution due to excessive data transfers centralized through a single host. For this purpose, our workflow engines are only manipulating limited size primitive data types (integers, floats, strings...) or references to files. File contents are never seen nor transferred through the hosting computer.

Files eventually need to be transferred towards the computing nodes though. The amount of data involved in application files transfer by far dominate the amount of data manipulated on the distributed platform during the workflow execution. Since data items and files might be shared between a various number of computing tasks generated by the workflow enactor, file transfer optimization is likely to have a significant impact on the overall workflow execution performance.

Data transfers and the opportunity for data transfer optimization and scheduling depend to a large extent on the underlying grid middleware capability. The DIET middleware for instance operates a grid of pre-deployed services, persistent on the host they were initially allocated on, which can implement data file persistency and peer-to-peer data transfers. This capability is exploited in the DIET DAGDA optimizer described below. Conversely, the gLite middleware operates a batch-oriented infrastructure on which anonymous computing resources are allocated on the fly, independently for each task, and freed as soon as the allocated computation completes. In this framework, the workflow manager has to ensure data persistence at the upper level. These two approaches and their connection to the workflow managers are discussed below.

3.1. Data management in the DIET grid middleware

DAGDA (Data Arrangement for the Grid and Distributed Applications [1]) is the data manager for the DIET middleware which allows data explicit or implicit replications and advanced data management on the grid.

DAGDA provides the following data management features:

- Distributed data persistency
- Container management
- Explicit or implicit data replications.
- File sharing between the nodes which can access to the same disk partition.
- High level configuration about the memory and disk space DIET should use for the data storage and transfers, with choice of a data replacement algorithm.

3.1.1. Distributed data persistency

The DAGDA system is a distributed data storage system on the grid. Its interface is accessible from any DIET agent and allows the agent to store or find data on the Grid without necessarily specifying where (i.e. on which host) the data should be stored or found. When a data is stored in DAGDA by a DIET agent (for example a SeD after execution of the service), DAGDA associates a unique ID to it. When another DIET agent requests the data using this ID, DAGDA performs the data search and manages the transfer by choosing the "best" data source according to statistics about previous transfers duration.

To transfer a data, DAGDA uses the *pull model*: data items are not sent from the source to the destination, but they are downloaded by the destination from the source. Figure 1 presents how DAGDA manages the data transfers for a standard DIET call:

1. The client performs a DIET service call.
2. DIET selects one or more SeDs to execute the service.
3. The client submits its request to the selected SeD, sending only the data descriptions.
4. The SeD downloads the new data from the client and the persistent ones from the nodes on which they are stored.
5. The SeD executes the service using the input data.
6. The SeD performs updates on the inout and out data and sends their descriptions to the client. The client then downloads the `volatile` and `persistent return` data.

At each step of the submission, the transfers are always started by the destination node.

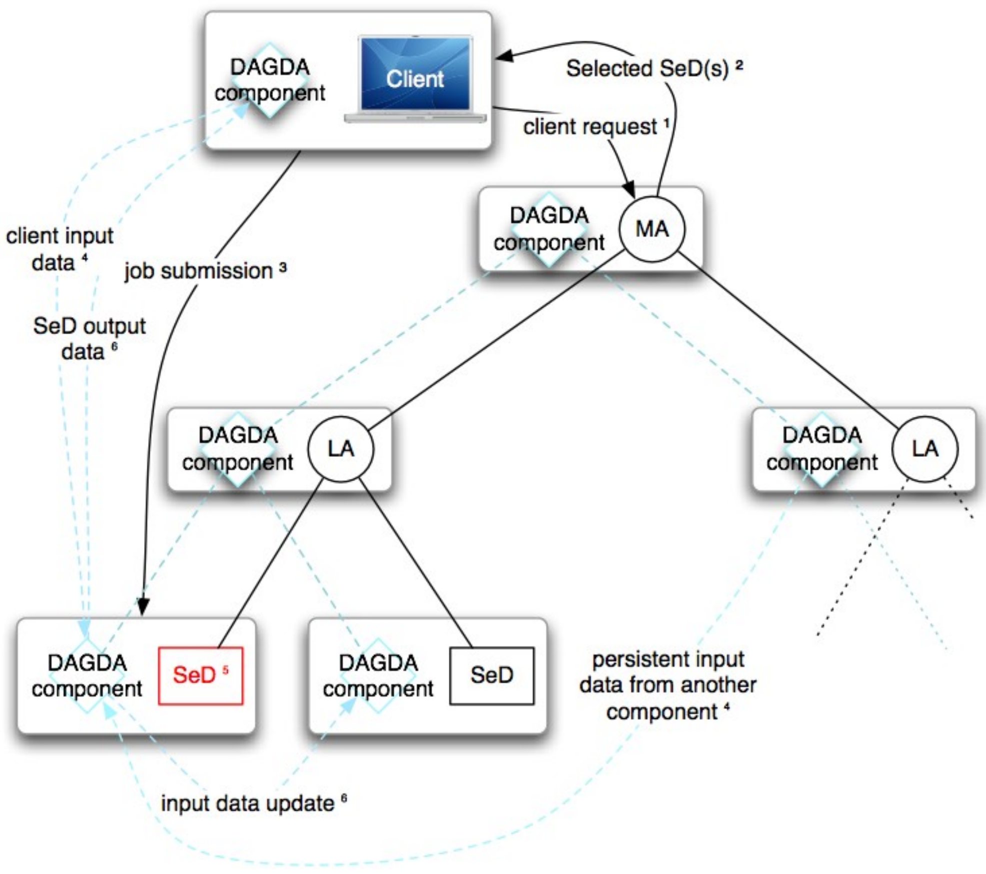


Figure 5.Figure : DAGDA transfer model

3.1.2. Container management

The DAGDA system provides a concept called a *container* used to group several data together as a single data. A container is a logical data that can contain an arbitrary number of data elements. Its size is dynamically adjusted and there are no constraints on the types of data that can be inserted as elements. This can be viewed as a dynamic array of pointers to data elements. Containers, as any other data items managed by DAGDA, can be transferred between any DAGDA agents. But the transfer of the container does not necessarily mean the transfer of all its elements. This is only when an element is used that the transfer happens.

Using this concept the workflow enactement engine can implement Arrays of arbitrary depth and manipulate elements within the Array, for example use references of elements of the Array to transfer them individually. A container produced by a given task executed on a node does not necessarily need to be transferred completely to other processing nodes; in the case of data parallelism, only one element of the container is transferred to each processing node, therefore it is essential that the workflow engine can provide a reference to that element only as the input of another task. The container is virtually splitted in different elements by the workflow engine.

Similarly when a task requires consolidation of results from many other tasks, usage of containers avoids data transfers through the client that would be required to build the input data. With containers the input data is virtually containing all the outputs to process but the data transfers occurs effectively only when required and between processing nodes only.

3.1.3. Data replication

When a data item produced by a task is used by many other tasks executed on different nodes, the data items is replicated on all these nodes therefore making it available from different sources. This behaviour is called “implicit replication”.

DAGDA also provides “explicit replication” through specific API calls available to all DIET components (client, agent or SeD). This can be used by the application developper to improve the replication policy by explicitely choosing specific nodes where the data will be available. For example if several grid sites are involved then a data that will be used by nodes on different sites can be pre-replicated on one agent per site (usually the DIET local agent).

3.1.4. File sharing

It is frequent that several nodes can access a shared disk partition (through a NFS server on a cluster for example). With DAGDA, a node can be configured to share the files that it manages with all its children nodes. A typical example of the usage of these features is for a service using a large file as read-only parameter executed on several SeD located on the same cluster. Figure 5 presents such a file sharing between several nodes:

- Two Local Agents and six SeDs can access to the same NFS partition (blue lines).
- The files registered on the first LA (red lines) are declared as shared with all of its children.
- If one of the SeDs or the second LA has to access to one of the files registered on the first LA, they can access it directly without data transfer through DIET.

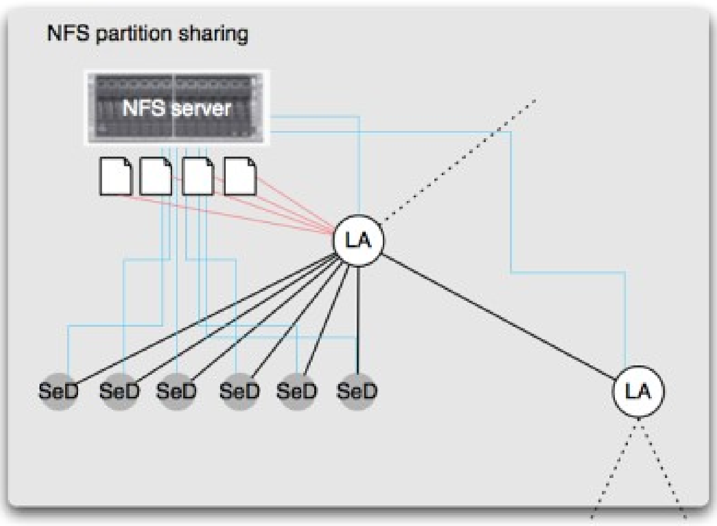


Figure 6. File sharing between an agent and its children

3.1.5. Physical resources usage configuration

The maximum amount of disk and memory space a host can use for computations submitted via DIET can be configured (as an option) for each DAGDA agent in order to avoid running out of memory or disk space during the workflow execution.

Using this feature implies that DAGDA will use a data replacement policy when there is a resource shortage on a DAGDA agent. Several policies are available and can be chosen in the configuration of the agent: either data that has not been used for the longest time, or data that has been stored the earliest (FIFO), will be removed first from the agent to free resources.

3.2. Data Management with the gLite grid middleware

The gLite middleware operates batch system principles: a computing node is allocated for the time of the computation and cleared after processing. As a consequence, files staged in by the computational process and file produced have no longer life time than the process itself. Any file that needs to be preserved after computation has to be transferred to a permanent storage resource by the computation process itself. To help in this process, the MOTEUR workflow manager interface to the gLite middleware uses a computing tasks wrapper that takes care of input files stage in and output files backups prior to and following on the wrapped code execution.

This environment does not allow for optimized data transfer strategies such as files persistence and peer-to-peer transfer between computing nodes. All files are transferred back and forth to storage servers which are visible throughout the grid infrastructure. The only optimization pattern that is supported by the gLite middleware is files replications: a logical instance of a file can be replicated to several places (several physical copies). By considering the physical location of a file, one might improve jobs execution performance. The data transfer cost for a job running on a given cluster will be lowered if it involves a storage resource hosted within this cluster. As part of the submission requirements, a job can be constrained to run on a given cluster. However, the middleware workload manager, using clusters load information to take jobs dispatching decision, does not include smart strategies appreciating data transfers cost with regards to computing resources availability cost. As a consequence, it is never guaranteed that the gain obtained by performing local transfers is not out weighted by the loss of using an overloaded computing site.

The global optimization problem (minimizing simultaneously expected computation times and data transfer) is very difficult and unresolved, especially on a large scale grid infrastructure such as EGEE where only a partial view of the infrastructure status can be determined. Therefore, we have been addressing the data transfer optimization problem at a higher level, within the workflow engine, were the knowledge of the application structure (and therefore the data transfers between consecutive tasks) can be exploited. Given that two consecutive tasks A and B in the workflow will cause intermediate data produced by A to be copied to a storage resources before being retrieved by B from the same resource, it can be interesting to chain the computations of A and B on the same computing resource. This will avoid back and forth data transfers and have the additional advantage of submitting a single (grouped) task A+B instead of submitting two tasks that will both be queued for a time in the system. As a matter of fact, for a sequential workflow, there is always a computing time benefit for grouping all sequential tasks.

The problem is more complex for non sequential workflow graphs. Grouping tasks of any graph may lead to improvement due to reduced file transfer but also losses due to prevention of parallelism. In [2] we have defined simple topological rules that may be used to decide whether the grouping of two connected tasks will lead to a loss of parallelism or not, by analyzing the workflow graph modified by the grouping transformation. The recursive application of these rules until the workflow graph is not transformed any more leads to a simplified graph for which some grouping may have been applied and there is a guarantee of not loosing any parallelism opportunity.

The strategy detailed in [2] only applies safe transformations, by static analysis on the workflow graph. Further grouping could be considered such as running bags of similar tasks (*i.e.* grouping data parallel task to some granularity level) or even grouping potentially independent tasks but for which the computing time is short enough for a grouping to be efficient as compared to a grid enactment due to the grid overhead that applies to each task. However, these strategies require execution and data transfer time estimates as run time as well as the analysis of the dynamic workflow execution directed acyclic graph. Such analysis involve more works that have not been addressed so far.

4. Conclusions

The GWENDIA language eases the description of large and compound data sets by using nested array structures. Computations applied to such structures can easily be described by applying principles of array programming to data-intensive workflows. Although the workflow managers develop are protected from the large amount of data distributed over the grid infrastructure during computations by manipulating references to data files, data transfers have to be optimized to ensure good performance of applications.

The data transfers scheduling is to a large extent dependent of the underlying middleware. The DIET middleware benefits from services pre-deployed and resilient on the computing hosts. Such services can store persistent data, enable peer-to-peer data transfers and, in some cases, provide shared files and replicates. The gLite middleware complies to less flexible batch processing rules. Data is cleared from the computing nodes as soon as the computational process finishes and it has to be backed up by the workflow manager for later use. In that environment, files cannot be persisted on the computing nodes. It is possible to minimize data transfers by grouping jobs at the workflow manager level though.

5. Bibliography

[1] E. Caron, F. Desprez and G. LeMahec, "DAGDA: Data Arrangement for the Grid and Distributed Applications", in IEEE Fourth International Conference on *eScience*, Indianapolis, USA, December 2008.

[2] T. Glatard, J. Montagnat, D. Emsellem, D. Lingrand. “A Service-Oriented Architecture enabling dynamic services grouping for optimizing distributed workflows execution”, in *Future Generation Computer Systems*, 24 (7), pages 720–730, Elsevier, July 2008.