A Multi-level Scheduler for the Grid Computing YML Framework

Sébastien Noël[1], Olivier Delannoy[3], Nahid Emad[3], Pierre Manneback[1], and Serge Petiton[2]

[1] Members of CoreGrid Institute on Resource Management and Scheduling
Faculté Polytechnique de Mons and CETIC, Mons, Belgium
{Pierre.Manneback,Sebastien.Noel}@fpms.ac.be
[2] Member of CoreGrid Institute on System Architecture
INRIA-Futurs, LIFL, USTL, Villeneuve d'Ascq, France
Serge.Petiton@inria.fr
[3] PRiSM - Laboratoire d'informatique - UVSQ, Versailles, France
{Nahid.Emad,Olivier.Delannoy}@prism.uvsq.fr

Abstract. This paper presents the integration of a multi-level scheduler in the YML architecture. It demonstrates the advantages of this architecture based on a component model and why it is well suited to develop parallel applications for Grids. Then, the multi-level scheduler under development for this framework is presented.[1]

Keywords: Grid Computing, YML, Scheduling, Resource Management, Workflow.

1 Introduction

High Performance Computing has emerged as a common need in many current applications. In order to solve such applications, Grid computing infrastructures have been developed to allow a high number of heterogeneous resources from different Virtual Organizations (VO) to be shared across a common network. Each cluster in each VO has its own management system. For example, availability of resources, access policies, Local Resource Manager (LRM), usage cost, etc. are usually different from site to site. Therefore common tools have to be provided to deal with resource heterogeneity and to facilitate the interconnection between them. Moreover, resource states are highly dynamic and volatile and increase the difficulty of managing a Grid infrastructure which is accessed concurrently by multiple users.

The development of Grid applications requires thorough knowledge of internal mechanisms and generally involves a preliminary step of identifying parallelizable parts of the application. This identification step leads to the creation of components, which are unitary tasks computed by one node of the Grid. An

[1] This research work is carried out under the FP6 Network Of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).

application is divided into components initialized with different input parameters and launched taking into account precedence constraints. Such workflow application is usually represented as a Direct Acyclic Graph and requires a high level of control of the Grid infrastructure.

In this paper, we study the use of a framework called YML for developing HPC applications on Grids and propose a multi-level scheduling architecture for it. The paper is organized as follows: in section 2, we succinctly present the YML framework and the associated workflow language YvetteML. Section 3 is devoted to the description of a general architecture of a multi-level scheduler for YML. Finally, section 4 presents some conclusions and perspectives

2 YML Framework

YML is a framework providing tools for parallelizing applications and has been developped at PRiSM laboratories in collaboration with Inria-Futurs/LIFL [3]. It focuses on two major aspects: the development of parallel applications and their execution in a Grid environment. YML makes this development independent of the Grid middlewares used underneath and hides the differences between them.

In the YML context, an application is divided into different computing sections, each of them containing some tasks sequentially or concurrently executed. A task, called a component, is a piece of work that can be mapped to one node in a parallel environment. It has some input and output parameters and is generally reusable in different parts of the application as well as in different applications. YML provides a special type of component, called *graph component*, which consists in the description of a subgraph. As we will see in 3.2, this kind of component will be exploited for the distribution of an application.

YML divides the development of a parallel application into three major steps:

1. *Definition of new components.* This definition consists of an abstract description and implementation component description, which are both presented in the next section.
2. *Description of the parallel application.* This description is independent of any underlying middleware and makes use of the components as functional units. It specifies the parallel and sequential parts of the application using the YvetteML graph description language and provides notifications to synchronise the execution of dependent components. This description is directly deduced from the graph representation of the application. More information on YvetteML is provided in subsection 2.3.
3. *Compilation of the application.* This step analyses and transforms the application graph into a list of parallel tasks taking into account the precedence constraints.

These three steps are all middleware independent and ensure that no Grid relevant knowledge is necessary to develop parallel applications.

After the compilation of the application, the execution can be started using a Workflow Scheduler which will interact with the underlying middleware. This

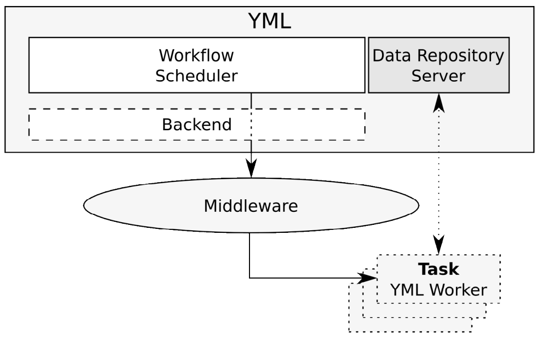

Fig. 1. YML Workflow Scheduler interaction with the middleware

interaction, represented in figure 1, requires the use of a specialized backend dedicated to the corresponding middleware. The execution of the application is directed by the Workflow Scheduler, which will submit tasks to the middleware through the dedicated backend. Each task is launched by a YML worker which will contact the Data Repository Server to obtain component binary and input parameters to start the computation.

2.1 YML Advantages

The comparison of YML with other workflow compatible frameworks like Unicore[5] or DAGMan[1] for Condor[6] points up several advantages.

YML helps the developer in the whole process of parallelizing applications. It starts at the early stage of component creation and goes through to the execution of strongly constrained workflow applications on a Grid. Moreover, YML allows the user to test and validate those applications on his own computer thanks to a special backend, which relies on the multithreading capabilities of the underlying operating system.

As we will see in the next subsection, component creation in YML is relatively simple. Existing code can be reused by importing libraries as new components without any adaptation. Those components are called by the application when computational tasks have to be started. Moreover, the notions of abstract and implementation descriptions of components add three interesting features to the Grid scheduler that could be used in the framework:

– data migration at the start and at the end of the application can easily be quantified from the abstract definition;
– the data used by a component is clearly defined in the abstract and implementation definitions; therefore this can be used in a checkpointing process to move a component from one node to another;
– computation time of a component can be evaluated from the implementation definition.

The use of Data Repository Servers hides the data migrations from the developer and ensures that the necessary data are always available to all application components.

The next subsections will present an example of how to use the YML framework to create a squared-matrix product application.

2.2 Component Creation

A component has to be defined and registered in the YML catalog in order to be used in a parallel application. This will be illustrated by a short example: a matrix multiplication component. The component creation can be done in three steps.

Definition of custom datatypes: New datatypes can be defined in new classes or in existing libraries (the YML compiler allows to include libraries, thus improving reusability of code). Functions for serializing and deserializing data have to be defined: the prototypes and their corresponding definitions (which are not represented here) are required for I/O operations made by the final component. Primitive datatypes such as integer, real and strings are already provided by the YML framework.

```
#ifndef MATRIX_HH
#define MATRIX_HH 1
#include <matrix.h>

typedef math::matrix<int> Matrix;

template <> bool param_import(Matrix& param,
          char* filename);
template <> bool param_export(const Matrix& param,
          char* filename);

#endif
```

This new datatype is called *Matrix* and makes use of the Matrix TCL Lite library [2] which does not require any modification.

Abstract definition: This definition includes a name for the component, a short description and a list of input and output parameters. This list specifies a name and a type for each parameter.

```
<?xml version="1.0" ?>
<yml-query login="userName" password="pass">
  <component name="MatrixProduct" type="abstract"
  description="Product of two matrices">

    <param name="result" type="Matrix"  mode="out"/>
    <param name="mat1"   type="Matrix"  mode="in" />
    <param name="mat2"   type="Matrix"  mode="in" />

  </component>
</yml-query>
```

This abstract definition is included in an XML request providing username and password for authentication purposes. The *Matrix* type is a custom datatype and has been defined at the previous step. The names of the three parameters match the names of the variables in the implementation part.

Implementation: It is based on the abstract definition. The output will be automatically sent to the Data Repository Server and made available for other components. This implementation is currently done using C/C++ but other programming languages can be added into all backends.

```
<?xml version="1.0" ?>
   <yml-query login="userName" password="pass">
      <component name="MatrixProduct_Impl" type="impl"
      abstract="MatrixProduct"
      description="Product of two matrices">
         <globals>
            <![CDATA[

               #include <matrix.h>

            ]]>
         </globals>
         <source lang="CXX">
            <![CDATA[

               result = mat1 * mat2;

            ]]>
         </source>
      </component>
   </yml-query>
```

This simple example demonstrates how easily components are created with YML. After the creation of the components, the graph description language YvetteML can be used to describe the application. Application creation with YvetteML is presented in the next subsection.

2.3 Application Creation with YvetteML

YvetteML provides different features for creating applications. These features are described in an illustrative example in figure 2, i.e. a parallel squared-matrix product. This application makes use of:

– *Component calls.* Their role is to submit a new task to the Local Resource Manager (LRM) providing the name of the component defined earlier and the different input parameters (lines 15, 16, 27 and 35 of figure 2).

```
 1  <?xml version="1.0"?>
 2  <yml-query login="userName" password="pass">
 3
 4    <application>
 5      <source>
 6  size    := 4;
 7  div     := 2;
 8  url1    := "http://www.prism.uvsq.fr/cni/yml/matrix1.csv";
 9  url2    := "http://www.prism.uvsq.fr/cni/yml/matrix2.csv";
10
11  par
12    par (i:= 1; div)         # i = index of the row
13        (j:= 1; div)         # j = index of the column
14    do
15      compute MatrixLoad(mat[1][i][j],url1,size,div,i,j);
16      compute MatrixLoad(mat[2][i][j],url2,size,div,i,j);
17      notify(evtMatrixLoaded[1][i][j]);
18      notify(evtMatrixLoaded[2][i][j]);
19    enddo
20  //
21    par (i:= 1; div)
22        (j:= 1; div)
23        (k:= 1; div)
24    do
25      wait(evtMatrixLoaded[1][i][k]);
26      wait(evtMatrixLoaded[2][k][j]);
27      compute MatrixProduct(result[i][j][k],mat[1][i][k],mat[2][k][j]);
28    enddo
29  endpar
30
31  seq (i:= 1; div)
32      (j:= 1; div)
33      (k:= 1; div)
34  do
35    compute MatrixComp(final,i,j,size,div,result[i][j][k]);
36  enddo
37
38      </source>
39    </application>
40  </yml-query>
```

Fig. 2. Squared-Matrix Product Application using YvetteML

– *Parallel sections.* They are used to explicitly define sections which will be executed in parallel (lines 11, 20 and 29 of figure 2) or to execute a parallel loop with iterators (lines 12 and 21 of figure 2).
– *Sequential loops.* They are loops with iterators, which are executed sequentially (line 31 of figure 2).
– *Conditional statements.* They can be used to test the value of iterators.

– *Event notifications.* They are used to synchronize the different parts of the execution when a precedence constraint has to be respected (lines 17, 18, 25 and 26 of figure 2). For instance in line 17, a new event called *evtMatrixLoad* is defined with an index ([1][i][j]) equal to that of the matrix that has just been loaded by the *MatrixLoad* component. After this notification, the corresponding wait call (in line 25) will stop blocking the execution of the iteration in the parallel loop.

The application described in figure 2 is presented for illustrative purpose. It makes use of three components: *MatrixLoad* (which loads part of a file into a *Matrix* datatype), *MatrixProduct* (which computes the product of two matrices) and *MatrixComp* (which composes the result *Matrix* by aggregating all submatrices).

This section has briefly presented the YML framework. More details can be found in [4]. Next section will describe the architecture of a scheduling model that we are considering.

3 A Multi-level Scheduling Model in YML

We describe in this section a multi-level scheduling model based on the YML framework. This model has multiple objectives:

1. to schedule a set of YML components with input data and precedence constraints issued from one or more users;
2. to provide computing resources for these components in a multi-middleware environment;
3. to offer users a guarantee in terms of completion time of the application;
4. to dynamically reorganise the schedule if unexpected events occur.

The following subsections develop different aspects of the model and present a case study.

3.1 An Economic Model

The context we focus on is characterized by the following points:

– the objective of the Grid is High Performance Computing;
– the applications are mostly compute-intensive rather than data-intensive;
– the resources are owned by different providers and part of different VOs;
– the number of resource providers ranges from several dozen to several hundred;
– the architecture is not centralized.

Each cluster:

– is composed of homogeneous resources;
– has a single access point;
– has a previously negotiated access policy to one or more other sites;
– is managed by an LRM (which may be different from one cluster to another).

The model we propose is based on an economic approach of resources and defines different entities which will interact within the Grid infrastructure. An entity can be a resource provider or a consumer, or both. Consumers require resources owned by different providers and available on the Grid. When a provider receives a request from a consumer, he will answer by proposing a set of suitable schedules and associated cost for parts of the application depending on access policy of the consumer and availability of local resources. He can possibly subcontract parts or the whole application to other resource providers without mentioning anything to the consumer.

This model can be used in different scenarios: either cooperation or competition between sites in the Grid infrastructure. Moreover, a hierarchy with different layers of scheduling instances, as presented in [9], can be built.

Technically, the main idea is to provide a YML server for each LRM. This YML server has 3 main purposes:

– to communicate with other YML servers and therefore, connect the different clusters in a common Grid;
– to interact with the underlying LRM using a specialized backend;
– to provide the features missing in the LRM.

The following subsection presents a typical scenario with this economic model.

3.2 Scheduling Scenario

A typical scheduling scenario is as follows:

1. the user/consumer submits his application to the local YML server;
2. the YML server analyses the application and decides whether it can provide the resources or not;
3. the YML server may forward the whole or parts of the request to other resource providers;
4. suitable schedules are sent back in return of each request;
5. the local YML server gathers the information and proposes differents prices to the user/consumer.

Steps 2, 3 and 4 are executed consecutively each time a YML server receives a scheduling request. The different sequences of the above scenario are explained in more detail in the next subsections.

Submission of the application. As described in section 2, the user makes use of YvetteML to describe a parallel application. Within the submission request, the user provides a completion time for the whole application or for some parts of it depending on the requirements. The local YML server handles the user requests in compliance with its local access policy. When the policy forbids access or the user has no authorization, the request fails and the computation is stopped. Otherwise, the scheduling process goes on to the next step.

Analysis of the application graph. Taking into account the amount and types of local resources on the one hand, and the current resource reservations on the other hand, the YML server will attempt to find suitable schedules for the whole or parts of the application. It will try to schedule successively:

1. the whole application;
2. parallel sections;
3. graph components;
4. tasks in the parallel sections.

If local resources are able to compute the whole application and meet the user's constraints, the scheduling process is either stopped or forwarded to other YML servers in the Grid. In the first case, reservation is made on local resources and the computation is started. In the latter case or in case the local infrastructure cannot provide sufficient resources for the whole application, the scheduling continues with step 3.

Forwarding of the request. The local server can decide whether it forwards the whole request or only parts of it (this decision can be made in compliance with the access policy). In the latter case, the request is split into different subrequests and is sent to other sites. To forward a request in the Grid infrastructure, the local server will interact with other resource providers with whom an access policy has been negotiated.

Return of suitable schedules. Each server will aggregate the suitable local schedules as well as schedules from other resource providers. Then, a reply is sent to the user.

3.3 Access Policy

When an instance wants to join the Grid infrastructure (to provide or to use resources), it has to negotiate access policies with one or more other scheduling instances. We propose an access policy divided into two sections, each containing static or dynamic information. This can be used to get a highly customizable contract between both scheduling instances. The static information will be used first to filter the list of resource providers without any interaction. The resulting list will be filtered again by querying each resource provider.

A non-exhaustive list of possible parameters that can be set in an access policy may include:

– time intervals;
– cost per node per time unit;
– constant/variable cost;
– application size;
– number of nodes;
– nodes description;
– number of providers;

- resource reservation;
- forwarding policy of the requests;
- failure compensation;
- access priority;
- etc.

Some or all of these parameters have to be set so as to define an access policy which can then be used in the resource discovery process. As presented in [7], this process essentially involves two filtering steps: an authorization filtering and a minimal requirement filtering.

The application requirements are defined by the YML Compiler, which analyzes the YvetteML code of the application provided by the user; this information is used for the minimal requirement filtering.

Figure 3 presents the two-step reduction of the set of suitable resources. First, a static filtering is applied to obtain a reduced list B. Then, if the list B is not empty, requests are sent to the resource providers to obtain dynamic information which will be used as a second filter to get a resource list C. This two-step filtering aims to reduce the number of requests exchanged between the scheduling instances.

Each resource provider in list C will be queried for possible schedules of the application. This process is illustrated in the next subsection.

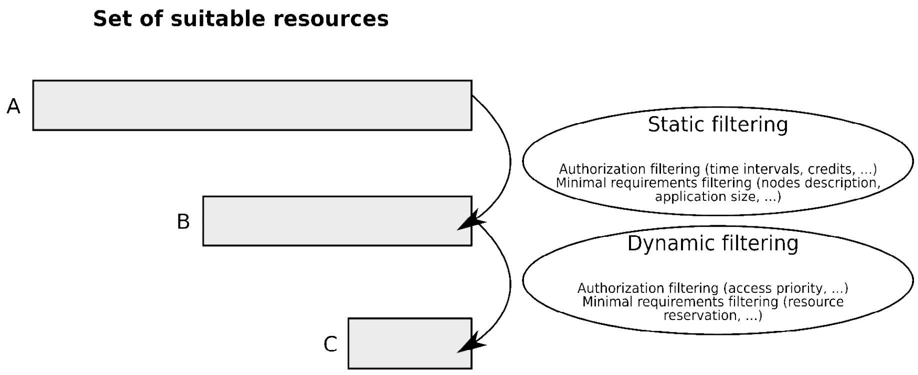
Set of suitable resources
A
B
C
Static filtering
Authorization filtering (time intervals, credits, ...)
Minimal requirements filtering (nodes description, application size, ...)
Dynamic filtering
Authorization filtering (access priority, ...)
Minimal requirements filtering (resource reservation, ...)

Fig. 3. Resource discovery process: static and dynamic filtering

3.4 Case Study

To describe the scheduling model, we will focus on an example Grid, presented in figure 4: the Grid infrastructure contains 5 clusters from different Virtual Organizations. An arrow from a server to another means that the first has an access policy to contact the latter. For instance, *YML server 1* has two resource providers, namely servers *2* and *4*; the rest of the Grid (servers *3*, *5* and *6*) is not visible to server *1*. When a server receives a request, it can handle the entire request or ask other resource providers. An access policy has previously been

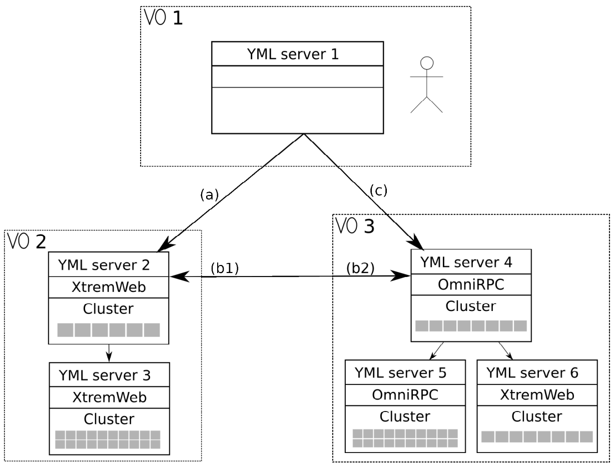

Fig. 4. Example of Grid infrastructure with different Virtual Organizations

negotiated and can be different for each client of a single site. Therefore, a direct request to a server may be less interesting than going through an intermediary. For instance, in figure 4, policy (c) could be more expensive than (a)+(b2); in this case, the client located in VO1 can ask for resources from server *2* which will negotiate resources with server *4* in VO3. The negotiation between *2* and *4* is not visible to the first client. As in VO1, a *YML server* can have no local resources; therefore, it acts only as a client and will contact other sites to get computational resources.

An example application is represented in figure 5 by a graph showing the interdependence between the tasks.

The start of the application is represented at the top of the figure and the end at the bottom. Large dashed squares represent parallel sections of the application, described by the user in the YvetteML code. Each task (which is a component with input parameters) is represented by a plain arrow. Notification arrows (dotted) are used to synchronise tasks and introduce precedence constraints into the application. The example presented in figure 5 has two parallel sections; the first is a preprocessing stage needed to start the computation process of the second one. For instance, the preprocessing tasks can be an initialization of the data. The duration is 3 for a preprocessing task and 10 for a computing task.

The scheduling process or mapping the application in figure 5 on the Grid presented in figure 4 will be simplified to help comprehension.

We suppose that:

- *YML servers 3* and *4* are unavailable for computation;
- 3 nodes of *YML server 2* are unavailable;
- the dialog between *YML servers 4*, *5* and *6* is not described;
- single task allocation is not presented but is effective in our model;

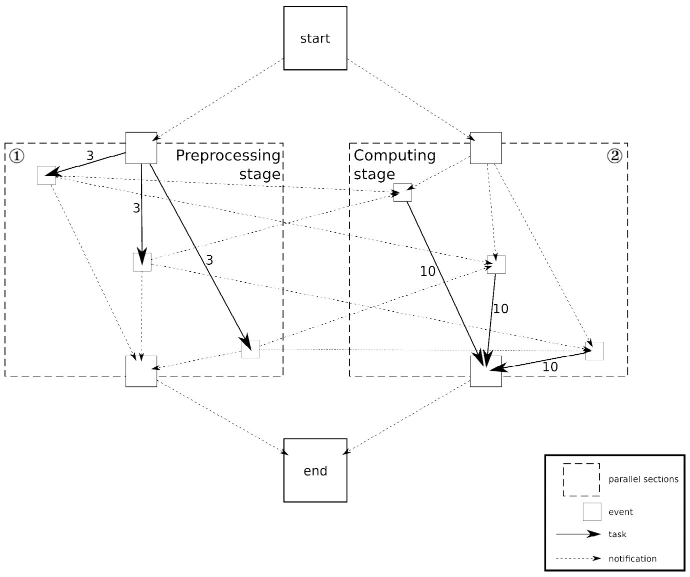

Fig. 5. Example of application graph

– the parameters of access policy (c) are such that no schedules will be proposed.

The response to each request is presented below from a YML server to another. Symbols ① and ② refer to the parallel sections in figure 5.

1. *Response from server 4 to server 1.* Access policy (c) is such that no schedules will be returned to YML server 1.
2. *Response from server 4 to server 2*
   Table 1 presents 3 different sets of schedules. The whole application can be coallocated to the resources of server *4* (or those of subcontractors which is not indicated to server *2*) but this coallocation cannot start before time 45. This means that many reservations have already been made or that the access policy cannot provide enough resources before this time. Other propositions consist in scheduling a parallel section (① or ②), which can be started earlier (on time 3).
   We suppose that the cost per node per time unit equals 2 in access policy (b2) (which is static information). The cost for the different schedules can

Table 1. Set of schedules proposed by server *4* to server *2*

application part	starting time	cost
① and ②	$[45,\infty]$	(3*3+3*10)*2=78
①	[3,17]	3*3*2=18
②	[3,10]	3*10*2=60

Table 2. Set of schedules proposed by server *2* to server *1*

application part	starting time	cost
① and ②	[45,∞]	$(3*3+3*10)*(2+1)=117$
①	[1,4]	$3*3*1=9$
①	[3,17]	$3*3*(2+1)=27$
②	[3,10]	$3*10*(2+1)=90$

be evaluated: 3 nodes x 3 time units x 2 for parallel section ① and 3 nodes x 10 time units x 2 for parallel section ②.

YML server 2 will aggregate those prices with its local suitable schedules.

3. *Response from server 2 to server 1*
   We suppose that the cost per node per time unit is not fixed in the access policy and is therefore a dynamic information. This means that server *2* is allowed to ask a different price at each request, depending on local considerations.

   The coallocation of the whole application can only be done by server *4* (or subcontractors): this is not indicated to server *1* which will see server *2* as only resource provider. Server *2* will increase the cost of the resources by 1 to take into account bandwidth use to access server *4*.

   A new schedule for parallel section ① is proposed by server *2*, which aims to enhance the use of local resources by applying an attractive cost of 1 per node per time unit.

   These schedules are received by *YML server 1* which will choose some of them and start resource reservation by requesting server *2*.

3.5 Features for the Scheduling Model

As presented in [8], a Grid scheduling architecture should provide different important features. These features are discussed in this subsection using the economic model described in 3.1.

The resource discovery process is not a major feature in our context. Each LRM is responsible for managing resource status and for providing suitable schedules depending on the resource availability.

In the same way, the status monitoring is not centralized and is only accessible by the local YML server, which will ask the LRM to provide the necessary information. This information can be accessed differently according to the installed LRM.

The reservation of resources is not supported by all LRMs and can therefore be managed by the YML server if necessary. In such Grid, the resource administrator has to ensure that YML is the only way of submitting tasks to the local resources.

The accounting and billing features will be managed at the YML level.

4 Conclusions and Perspectives

In this paper, we have presented the YML Grid computing framework which can be used to develop parallel applications and execute them in a Grid environment. We have also described a multi-level scheduling model which can be used to build cooperative or competitive Grids using a customized access policy between scheduling instances of the Grid. This scheduling model is currently being integrated into the YML framework and will provide multi-middleware capabilities.

We aim to validate this scheduling model by testing it on the YML framework. This testing phase will open up new perspectives and show what will be needed to be adapted in the current model.

References

1. Directed acyclic graph manager website. http://www.cs.wisc.edu/condor/dagman.
2. Techsoft - matrix TCL website. http://www.techsoftpl.com/matrix.
3. YML website. http://www.prism.uvsq.fr/cni/yml.
4. O. Delannoy, N. Emad, and S. Petiton. Workflow global computing with yml. To appear in Proceedings of GRID2006, Barcelona.
5. D. W. Erwin and D. F. Snelling. Unicore: A grid computing environment. *Lecture Notes in Computer Science*, 2150:825–834, 2001.
6. S. Santhanam, P. Elango, A. Arpaci-Dusseau, and M. Livny. Deploying virtual machines as sandboxes for the grid. In *Second Workshop on Real, Large Distributed Systems (WORLDS 2005)*, San Francisco, CA, December 2005.
7. J. M. Schopf. Ten actions when grid scheduling. *Grid Resource Management*, pages 15–23, 2004.
8. U. Schwiegelshohn and R. Yahyapour. Attributes for communication between grid scheduling instances. *Grid Resource Management*, pages 41–52, 2004.
9. N. Tonellotto, P. Wieder, and R. Yahyapour. A proposal for a generic grid scheduling architecture. volume TR-0015 of *CoreGRID Technical Report*, November 2005.