# 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 Programming Model*
INRIA-Futurs, LIFL, USTL, Villeneuve d'Ascq, France
Petiton@lifl.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 under development multi-level scheduler for this framework will be presented. [1]

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

## 1 Introduction

High Performance Computing emerged has a common need in a lot of current applications. As a new way to solve such applications, Grid computing infrastructures have been developed to share a high number of heterogeneous resources from different Virtual Organizations (VO) into 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. Common tools have to be provided to deal with resource heterogeneity and to ease the interconnection between all of them. Moreover, resource states are highly dynamic and volatile and increase the difficulty of managing a Grid infrastructure which is accessed by multiple users at the same time.

The development of parallel applications requires a high knowledge of internal mechanisms and generally needs a primary step of identifying parallelizable parts of the application. This identification step leads to the creation of components

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

which are unitary tasks computed by one node of the Grid. An application is divided into components initialized with different input parameters and launched with respect to precedence constraints. Such worfklow applications which are usually represented as a Direct Acyclic Graph need a high level of control in the Grid infrastructure to deal with the complex set of constraints.

As we will see in section 2, YML provides tools to define complex workflow applications independently of the underlying Grid infrastructure. Its definition needs the use of a specialized workflow language called YvetteML. Section 3 will present a functional model aiming at providing a multi-level scheduler in YML. Finally, some conclusions and perspectives will be sketched.

## 2 YML framework

YML is a framework providing tools to parallelize applications which 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 differences between them.

On the YML point of view, an application is divided into different computing sections, each of them containing some tasks executed sequentially or concurrently. 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 components, called graph component, that consists in the description of subgraph. As we will see in 3.2, this kind of components will be exploited for the distribution of the application.

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

1. A *definition of new components.* This definition consists in an Abstract and Implementation component description which are both presented in the next section.
2. A *description of the parallel application.* This description is independent of any underlying middleware and makes use of the components as functional units. It specifies parallel and sequential parts of the application using the YvetteML graph description language [5] 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. The *compilation of the application.* This step analyses and transforms the application graph into a list of parallel tasks with respect of the precedence constraints.

The three steps above are all middleware independent and ensure that no Grid relevant knowledge is needed to develop parallel applications.

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

This interaction, represented on figure 1, needs the use of a specialized backend dedicated to the corresponding middleware. The execution of the application is

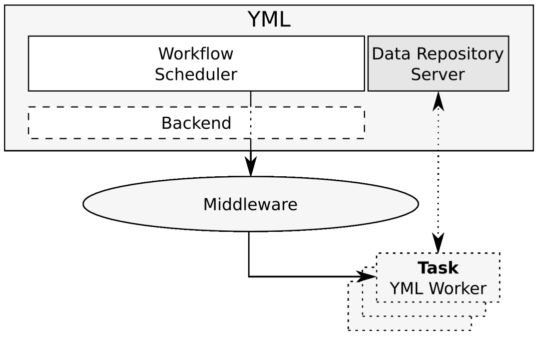

**Fig. 1.** YML Workflow Scheduler interaction with the middleware

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[6] or DAGMan[1] for Condor[7] points on several advantages.

YML helps the developer in the whole process of parallelizing applications. It starts at the early stage of components creation to the execution of hardly constrained workflow applications on a Grid. Moreover, YML allows the test and validation of those applications on the user computer using a special backend which relies on the multi threading capabilities of the underlying operating system.

As we will see in next subsection, YML eases components creation. Existing code can be reused by importing libraries as some 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 bring three interesting features for the Grid scheduler that can be included in the framework:

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

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

The next subsections will present an example usage of the YML framework for the creation of 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 (YML compiler enables inclusion of libraries improving reusability of code). Functions to serialize and deserialize data have to be defined: the two 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 do 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>
```

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

**Implementation** : based on an Abstract definition. The results data will be automatically sent to the Data Repository Server and available for other components. At now, this implementation is made using C/C++ but other programming languages can be easily added into each backend.

```
<?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 short example demonstrates the easiness of component construction with YML. After the creation of the needed components, YvetteML can be used to describe the application. The creation of the application and YvetteML features are presented in the next subsection.

### 2.3 Application creation with YvetteML

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

- *Component calls*. They 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).
- *Parallel sections*. They can be used to explicitly define sections which will be executed in parallel (lines 11, 20 and 29 of figure 2) or to execute a loop in parallel with iterators (lines 12 and 21 of figure 2).
- *Sequential loops*. They consist in loops with iterators sequentially executed (line 31 of figure 2).
- *Conditional statements*. They may 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 on line 17, a new event called evtMatrixLoad is defined with an index ([1][i][j]) equal to the one of the matrix that has just been loaded by the MatrixLoad component. After that notification, the corresponding wait call (on line 25) will stop blocking the execution of that 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 is described in the previous subsection and compute the product of two matrices) and *MatrixComp* (which composes the result *Matrix* by aggregating all sub-matrices).

This section has presented the current state of YML. The next section will explain the scheduling model we are implementing on YML to enable its use with multiple middlewares.

## 3 A multi-level scheduling model in YML

YML is a research project and therefore, different major features are still missing, e.g.:

- backends for more middlewares (XtremWeb[4] and OmniRPC[8] are currently supported);
- support for multi-middleware environment;
- service level agreement.

To provide these features, we propose a multi-level scheduling model we are currently implementing in YML. This model has multiple objectives:

```
<?xml version="1.0"?>
<yml-query login="userName" password="pass">

  <application>
    <source>
size    := 4;
div     := 2;
url1    := "http://www.prism.uvsq.fr/cni/yml/matrix1.csv";
url2    := "http://www.prism.uvsq.fr/cni/yml/matrix2.csv";

par
  par (i:= 1; size/div)         # i = index of the row
      (j:= 1; size/div)         # j = index of the column
  do
    compute MatrixLoad(mat[1][i][j],url1,size,div,i,j);
    compute MatrixLoad(mat[2][i][j],url2,size,div,i,j);
    notify(evtMatrixLoaded[1][i][j]);
    notify(evtMatrixLoaded[2][i][j]);
  enddo
//
  par (i:= 1; size/div)
      (j:= 1; size/div)
      (k:= 1; size/div)
  do
    wait(evtMatrixLoaded[1][i][k]);
    wait(evtMatrixLoaded[2][k][j]);
    compute MatrixProduct(result[i][j][k],mat[1][i][k],mat[2][k][j]);
  enddo
endpar

seq (i:= 1; size/div)
    (j:= 1; size/div)
    (k:= 1; size/div)
do
  compute MatrixComp(final,i,j,size,div,result[i][j][k]);
enddo

    </source>
  </application>
</yml-query>
```

**Fig. 2.** Squared-Matrix Product Application using YvetteML

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

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

## 3.1 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 is about tens to hundreds;
– the architecture is not centralized.

Each cluster:

– is composed of homogenous resources;
– has a single access point;
– has a previously negotiated policy to access one or more other sites;
– is managed by a 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 in the Grid infrastructure. An entity can be a resource provider or consumer or both. Consumers require resources of the Grid which are owned by providers. 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 regarding 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 a cooperation or a competition between sites in the Grid infrastructure. Moreover, a hierarchy with different layers of scheduling instances, as presented in [11], can be built.

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

– 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 needed features missing in the LRM.

The following subsection presents a typical usage scenario in this economic model.

## 3.2 Scheduling scenario

A typical scheduling scenario is:

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 needed resources or not;
3. it 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 bids to the user/consumer.

Steps 2, 3 and 4 are executed consecutively each time a YML server receives a scheduling request. The different parts of the above scenario are explained in more details 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 regarding his local access policy. When the policy forbids access or the user has no autorization, the request fails and the computation is stopped. Otherwize, the scheduling process goes on with the next step.

**Analyzis of the application graph.** Regarding the number and type of the local resources in one side, the current resource reservations in the other side, the YML server will attempt to find suitable schedules for the whole or parts of the applications. In order, it will try to schedule :

– the whole application;
– parallel sections;
– graph components;
– tasks in the parallel sections.

If local resources are able to compute the whole application respecting the user constraints, the scheduling process is either stopped, either forwarded to other YML server in the Grid. In the first case, reservation is made on local resources and the computation is started. In the other case or in the case local infrastructure can not provide sufficient resources for the whole application, scheduling continues with the next step.

**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 with regards to the access policy). In the last case, the request is splitted into different sub-requests and are sent to other sites. To forward a request in the Grid infrastructure, the local server will interact with other resource providers for which 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, the answer is sent to the author of the request.

## 3.3 Access policy

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

The non-exhaustive list of possible parameters that can be set in an access policy is:

- time intervals;
- cost per node per time unit;
- constant/variable cost;
- application size;
- number of nodes;
- nodes description;
- number of providers;
- resource reservation;
- forwarding of the requests;
- failure compensation;
- access priority;
- etc.

Some or all of these parameters have to be fixed in order to define an access policy which can therefore be used in the resource discovery process. As presented in [9], this process is mainly composed of 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 will be used for the minimal requirement filtering.

Figure 3 presents the reduction of the set of suitable resources which is done in two parts. At first, a static filtering is applied to obtain a reduced list B. Then, if this 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 filtering in two steps tries to reduce the amount of requests exchanged between the scheduling instances.

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

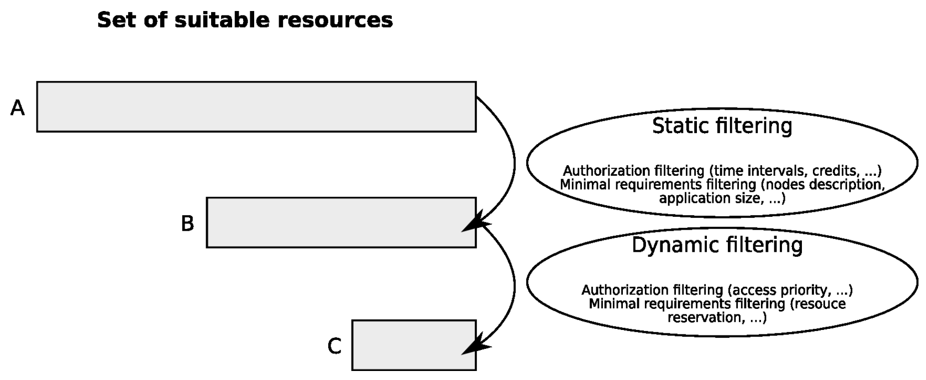
Set of suitable resources
A
B
C
Static filtering
Authorization filtering (time intervals, credits, ...)
Minimal requirements filtering (nodes description, application size, ...)
Dynamic filtering
Authorization filtering (access priority, ...)
Minimal requirements filtering (resouce reservation, ...)

**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 which is presented in figure 4: the Grid infrastructure contains 5 clusters from different Virtual Organizations. An arrow from a server to an other means that the first

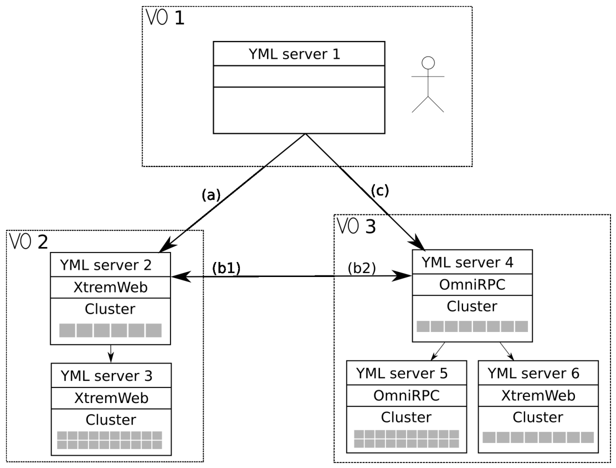

**Fig. 4.** Example of Grid infrastructure with different Virtual Organizations

has an access policy to contact the second. For instance, server YML 1 has two resource providers which are 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 all the request or ask other resource providers. An access policy is previously negotiated and may be different for each client of a same site. Therefore, a direct request to a server may be less interesting than passing by an intermediate. For instance, in the figure, (c) policy could be more expensive than (a)+(b); in this case, the client located in VO1 may ask resources to server 2 which will negotiate resources with server 4 in VO3. The negotiation between 2 and 4 will not be visible to the first client. As in the VO1, a YML server may have no local resources; therefore, it acts only as a client and will contact other sites to get computational resources.

We present an example application which, as shown on figure 5 can be represented by a graph specifying dependances between tasks.

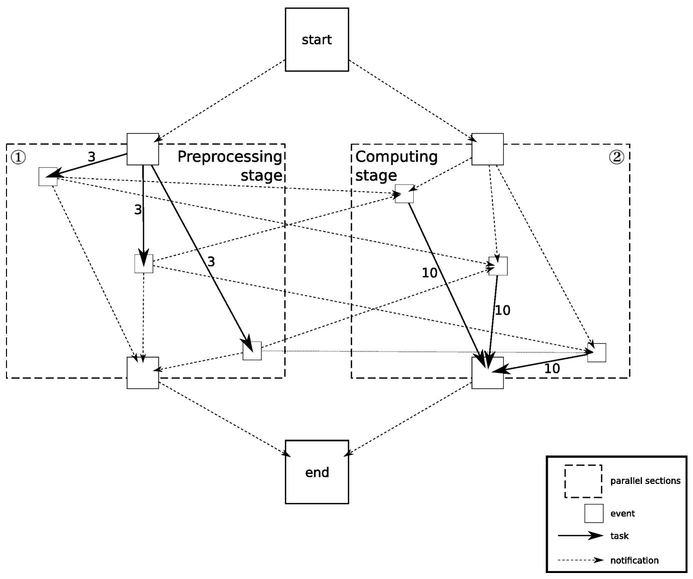

**Fig. 5.** Example of application graph

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 an oriented plain arc. Notification arrows are used to synchronise tasks and introduce precedence constraints in the application. The example application presented in figure 5 has two parallel sections; the first one consists in a preprocessing stage needed to start the computation process in the second parallel section. 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 to map the application of figure 5 on the Grid presented in figure 4 will be simplified to enhance comprehension.

We suppose that:

- YML servers 3 and 4 are unavailable for the computation;
- 3 nodes of YML server 2 are unavailable;

- dialog between YML servers 4, 5 and 6 will not be described;
- single task allocation will not be presented but is effective in our model;
- parameters of (c) access policy are such that no schedules will be proposed.

The response to each request is presented bellow 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.* The (c) access policy is such that no schedules will be returned to YML server 1.
2. *Response from server 4 to server 2.*

| application part | starting time | cost |
|---|---|---|
| ① and ② | [45,∞] | (3*3+3*10)*2=78 |
| ① | [3,17] | 3*3*2=18 |
| ② | [3,10] | 3*10*2=60 |

The table presented above propose 3 differents sets of schedules. The whole application can be coallocated on the resources of server 4 (or those of subcontractors which is not indicated to server 2) but this coallocation can not start before time 45. This means that a lot of reservations have already been made or that access policy can not provide enough resources before that time. Other propositions consist in scheduling a parallel section (① or ②), which can be started earlier (on time 3).

We suppose the usage cost per node per time unit equals 2 in access policy (b2) (which is a static information). The cost for the different schedules can be evaluated: 3 nodes multiplied by 3 time units multiplied by 2 for parallel section ① and 3 nodes multiplied by 10 time units multiplied by 2 for parallel section ②.

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

3. *Response from 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 |

We suppose that the usage cost per node per time unit is not fixed in the access policy and is therefore dynamic information. This means that server 2 is allowed to provide a different price at each request regarding 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 by 1 the usage cost of the resources to take into account bandwidth usage to access server 4.

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

Those 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 [10], a Grid scheduling architecture should provide different major features. Those features are discussed in this subsection regarding 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 provide suitable schedules regarding 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 LRM to get needed information. This information can be accessed differently according to the installed LRM.

The reservation of resources is not supported by all LRM and can therefore be managed by the YML server if needed. In such a 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 integrated in the YML framework and will provide multi-middlewares capabilities.

Our aim is to improve this scheduling model by testing it in the YML framework. This testing phase will bring new perspectives and will show needed adaptations of 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. F. Capello, S. Djilali, G. Fedak, T. Herault, F. Magniette, V. Neri, and O. Lodygensky. Computing on large scale distributed systems: Xtremweb architecture, programming models, security, tests and convergence with grid. *FGCS - Future Generation Computer Science*, 2004.
5. O. Delannoy, N. Emad, and S. Petiton. Workflow global computing with yml. Technical report, 2006.
6. D. W. Erwin and D. F. Snelling. Unicore: A grid computing environment. *Lecture Notes in Computer Science*, 2150:825–834, 2001.
7. 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.
8. M. Sato, M. Hirano, Y. Tanaka, and S. Sekiguchi. OmniRPC: A Grid RPC facility for cluster and global computing in OpenMP. *Lecture Notes in Computer Science*, 2104:130–136, 2001.

9. J. M. Schopf. Ten actions when grid scheduling. *Grid Resource Management*, pages 15–23, 2004.
10. U. Schwiegelshohn and R. Yahyapour. Attributes for communication between grid scheduling instances. *Grid Resource Management*, pages 41–52, 2004.
11. N. Tonellotto, P. Wieder, and R. Yahyapour. A proposal for a generic grid scheduling architecture. volume TR-0015 of *CoreGRID Technical Report*, November 2005.