Workflow Global Computing with YML

Olivier Delannoy #1, Nahid Emad #1, Serge Petiton *2

#1 *PRiSM Laboratory, University of Versailles,*
45 avenue des Etats Unis, Versailles, France
{Olivier.Delannoy,Nahid.Emad}@prism.uvsq.fr

*2 *INRIA and LIFL Laboratoy, University of Lille,*
Cité Scientifique, 59655 Villeneuve d'Ascq, France
Serge.Petiton@inria.fr

Abstract— **In this paper we propose a framework dedicated to the development and the execution of parallel applications over large scale global computing platforms. A workflow programming environment will be introduced, based on a new workflow language *YvetteML* and a Human-GRID middleware interface called YML. This language allows description of different kind of components to be allocated to GRID resources. Depending of the different targeted resources, the components may be associated to computation, data migration or other resource controls. YML is designed to have several back-ends for different middleware, as a well-designed front end is developed independently of any dedicated middleware. In order to make the framework immediately useful, YML comes with pre-configured interfaces to some numerical routines and a numerical library for iterative linear algebra methods. We will present experimentations done on some large scale platforms using a peer to peer middleware with a numerical application case study.**

I. Introduction

An important number of GRID and peer to peer middleware are now mature enough for use in production. The number of applications relying on a GRID as run-time environment increases. However the lack of standard in GRID programming interface forces the application development to be tightly coupled with the chosen middleware. The selection of a run-time support and the development costs associated with the release of an application have a crucial role in the choice of the programming environment. The GRID complexity requires the use of high level tools hiding parts of the process to the users.

YML provides a framework dedicated to the development and the execution of parallel applications over large scale middleware [1]. YML includes a workflow language named *YvetteML* used in the description of applications and their executions. YML furnishes a compiler and a just-in-time scheduler for *YvetteML* . It allows to manage the execution of the application over the underlying parallel architecture which can be a peer to peer or a GRID middleware. The specificity of each middleware is hidden to the user through YML. This framework provides workflow engine capabilities on top of a global computing platform. YML is designed to act transparently for complex applications using numerous communications, code coupling, etc on dynamic platforms.

YML is implemented using a component-oriented approach to help the integration of existing numerical library. In order to illustrate the use of YML we adapt an existing object oriented library for the computation of linear algebra problem. Linear Algebra Kernel (LAKe) is a class library developed using an object oriented approach in order to enable very good code reuse for both sequential and parallel applications. LAKe design focuses on the development of iterative methods and does not provide simple mechanism to support hybrid methods. The latter consist in coupling several numerical methods (called co-methods) in order to decrease the number of iterations required to find the results. This kind of methods is well adapted to global computing environment. This is because they are fault tolerant, coarse grain parallelism, have asynchronous communications and benefit from hardware heterogeneity.

Following this introduction, the second section discusses the motivations for proposing YML and make a quick overview of other related projects. The third section presents YML and discuss its internal leaving for the fourth section the discussion on how YML interact with middleware and makes use of them. The YML default back-end enables the use of the XtremWeb middleware. The interaction between YML and XtremWeb is discussed in this section. The fifth section shows an example of application from linear algebra. In order to illustrate the use of YML for development and execution, this application is detailed in the section. The last section concludes the paper and presents our future work.

II. Motivations

High performance computers are mostly used for scientific computing. Numerical application requires a lot of computing power and takes time even on high performance computers. Numerous simulation problems issued from domain such as weather prediction, climate study, and aircraft design are regularly translated to linear algebra problems. Such methods are complex and require a good knowledge of the numerical aspects, the computing methods and the distribution techniques. For these applications, the design and development of high level tools which hides the complexity of the global computing middleware become necessary.

The programming of high performance computers has been a difficult task for years. At the beginning, most of the scientific computing was rested on vendor solution specific to each architecture of high performance computers. The definition of two standard named MPI and OpenMP changes a lot the application development and eases the migration

from one architecture to another. The growing of clusters and the success of distributed memory systems are mostly due to the MPI standard and its adoption by a large community of users. Several approaches were lengthily used in distributed environments but not in the context of high performance computers such as Remote Procedure Call (RPC) mechanism used in *Network File System*.

The generalization of the distributed computation systems leads to the global computing environments. A such environment is a distributed architecture with a lot of changes occurring asynchronously on the resources composing the GRID. A change can be a disappearance or an appearance of resources where a resource can be data storage or cpu time but is not limited to this two. The programming of such dynamic architectures does not fit well with API like MPI. The loss of computing resources during the execution of an MPI application is not supported this API until recent version while it is a standard event in global computing environment. Many projects study several approaches for creating MPI runtime for global computing middleware[2][3][4]. This solution is interesting for the end-user because it enables the reuse of existing application directly on this new kind of architectures.

The global computing environments rest a lot on Internet technologies. The common use of Internet follows the client-server model. It is well adapted to the RPC programming model which enforces the master-worker application organization. Most global computing environments define an RPC like programming interface. The notion of web services is an adaptation of the RPC model which standardizes the interaction between communicating systems, the discovery of distant services. The Globus Toolkit[5] is one of the most complex GRID solutions. It promotes a services oriented GRID architecture relying on web services technology for discovery, interaction and exploitation of remote services. Globus defines a set of building blocks needed in all GRID middleware and let the authority using the GRID registers new services dedicated to a particular community. The execution of complex application requiring multiple services collaboration is difficult to express. The first works in this direction by two early projects, WebFlow[6] and Common Component Architecture (CCA) have oriented the research activity toward the workflow frameworks.

The workflow frameworks are a more and more attractive topic in the domain of GRID middleware. The number of the projects involved in the definition of a language for expressing workflow on GRID leads to several models. The main models used to define workflow are detailed in [7]. Each model has interesting properties depending on the application domain. Directed Acyclic Graph (DAG) are used in many projects including DagMan[8], UNICORE[9] and GridAnt[10]. This model is efficient to describe static workflow. DAG misses constructions such as iteration or loop but the definition of a workflow using DAG is straightforward. The model of the YML workflow framework, proposed in this paper, is based on directed general graph (DGG). However directed graph express by YML can contain loops, iterations and branching. Other workflow model includes systems build around Petri-Net[11]. The workflow description in Petri-Net model is efficient and also associated to a graphical modeling tools such as GME.

The YML framework and its workflow engine distinguishes itself from the other available solutions in several aspects. Indeed, with YML the applications are interoperable between middleware supported by the framework. Most workflow framework do not integrate the component creation. YML integrates a component generator used to produce the binary components to be executed remotely. The end-user can implement components using standard languages such as C or C++. YML framework integrates with existing middleware thanks to a highly modular design. Using YML, users are able to create and validate applications even if he/she has no access to the GRID. A back-end called process rest on the multi-tasking capability of modern operating system to simulate the behavior of YML on a single computer.

III. YML FRAMEWORK

YML is a framework dedicated to the creation and the execution of parallel applications on grids and peer to peer middleware. YML rests on a dedicated programming language designed to help users to exploit large scale middleware. This workflow language is named *YvetteML* . It enables the description of complex parallel applications independently of the execution platform. An application written in *YvetteML* language can be executed on several middleware without changes. The language permits the description of the graph of an application. The nodes of the graph correspond to computation while edges correspond to dependencies or communications. The graph once compiled in an internal representation is sent to a workflow engine specialized in executing such graphs. The *YvetteML* language allows a description of different kind of components to be allocated to GRID resources. It integrates the ability to describe components on the one hand and application graphs on the other hand. Both aspects are encapsulated in XML document for homogeneity. *YvetteML* which is a graph description language provides a way to specify the communication between components during the execution of the application. The graph can contain parallel and sequential sections and standard construction of most languages including branching, exceptions and loops. The graph language explicites the dependencies between the components during the execution. Theses dependencies rests on the notion of events. The *YvetteML* language is not a graphical language like YAWL[12][11], Triana[13] and Kepler[14]. They are oriented-business workflow languages which rest on typical workflow patterns. While YML describes the scientific application graphs which are regularly using three or more dimensions which make them difficult to render and edit.

The rest of the section describes the *YvetteML* workflow language. The two aspects of the language are presented. The graph or coordination language is described just after the presentation of the component model. The section continues

with a presentation of the YML framework design and the role of its components.

A. The YvetteML Language

The *YvetteML* workflow language consists in two major aspects: a component model on the one hand and a graph description language on the other hand. Both aspects interact in order to create an application with strict separation of the computation and the control or communication of the application.

YvetteML rests on a light component model with the requirements listed bellow. The component model must hide the communication aspect and all stuff related to data serialization and transmission. The components have to be easy to specialize in order to benefit from the capabilities specific to a middleware. The model used by *YvetteML* makes use of stateless components. A component does not store any state. It means several executions of a component with the same parameters must produce the same result set[1]. A component represents a chunk of computation requiring no communication with the rest of the application. This component model is similar to condor job definition used in DagMan[15]. All components execution consists in three steps: the data acquisition, the computation and finally the data exportation. The components in *YvetteML* are described using XML and are of three kinds :

Abstract component: An abstract component is a middleware independent description of a computation. The abstract component is used in the definition of graphs and during the code generation step. An abstract component is similar to a declaration in the C language. It defines the communication channels with other components. Each channel corresponds to a data in input or in output of the component and is typed. Abstract components behave like an Interface Definition Language (IDL). It defines the communication interface between the component and the rest of the world.

Implementation component: An implementation component specializes an abstract component and provides an implementation for such component. There can be several implementations for the same abstract component. Implementation components are similar to the use of a function defined in another language like Fortran in a C program. Implementation components are only used during the execution of an application. Computation is described using a common language such as C/C++ but the software needed for handling communication channel is generated for each implementation components during their generation. The component generation will be discussed in detail while presenting the code generation process.

Graph component: It is a special kind of implementation component. It encompasses a graph expressed in *YvetteML* instead of a description of the computation. Such components are middleware independent. It is allowed to define graphs as well as implementation for the same abstract component. Graph components are similar to function definitions in the C language. The language itself does not make any assumption on the graph expansion mechanism. It's up to the implementation of the scheduler and compiler to make decision on the way graph components are handled. The current implementation of YML does not yet support graph components but the language definition already integrates those components.

[1]This is only true if component implementations do not rely on random number generator and similar functionality.

An example of component definition will be discussed in section V. The component model of *YvetteML* distinguishes itself from other component-oriented workflow like AGWL[16] or GFDL[17] by integrating the concrete component implementation in the model in order to provide component generator.

The second aspect of *YvetteML* is the control of components. This aspect is not expressed using XML because of the verbosity XML notation which is not user friendly. The control consists of the graph of the application and is described using a coordination language which is the input of a workflow engine in charge of the execution of the application. The graph coordination language provides standard construction for expressing a parallel application. The most important construction of the language is the component call. A component call corresponds to the execution of a component on the middleware. Graphs in *YvetteML* do not depend on the underlying middleware. One can only make call to abstract components. The compilation step does not need any information related to the implementation or graph effectively used by the workflow engine. Listing below illustrates the *YvetteML* language. It shows a reduction operation on a collection of objects. This example illustrates most of the constructions of *YvetteML* . It assumes that two components exist. The first component is named create, it generates an object (a number, a string, etc). The second component is the reduction operator and is named reduction. Component creation is illustrated in section V. The goal of the reduction is to store the result in a `data[1]`. This result can then be used in the rest of the application. In order to construct the result we apply a binary operator reduction by simulating a tree. In a binary tree the children of a node i are indexed $2i$ and $2i+1$. The initial data are stored as children of the tree and the second parallel section does the reduction and the computation of all *inner nodes* of the tree. In order to synchronized the reduction operation we explicitly use events manipulator **wait** and **notify**. Using two parallel sections, the reduction can start even if all the objects are not yet created. Finally, in the listing below comments start with a # and terminate at the end of the line.

```
# Const definitions
objectCount := 16;
intNode := objectCount − 1;
#data is a collection of objects
#evtData is a collection of YML events
#the first item is indexed objectCount.
par
  # First parallel section : Creation
```

```
  par (i := 1 ; objectCount) do
    # call create component and
    # store the result in data
    compute create(data[intNode + i]);
    # dependency explicit management
    notify(evtData[intNode + i]);
  enddo
// # Second parallel section: Reduction
  par (i := 1 ; intNode) do
    # wait for explicit synchronization
    # in order to create node i we need
    # to wait for nodes 2*i and 2*i+1
    wait(evtData[2*i] and evtData[2*i+1]);
    # compute the reduction
    compute reduction(data[i], data[2*i],
                     data[2*i+1]);
    # tell node i has been created.
    notify(evtData[i]);
  enddo
endpar
```

B. YML design

The workflow approach is based on a program that manages the execution of the computation involved in an application. The YML framework is based onto this approach. It provides the end-user with a set of tools to develop and execute applications over large scale architectures. It defines an abstraction and hides the specificities of each middleware. The user describes the computation of its application using the component description aspect and uses a graph language to describe the communication of its application. An application developed using the *YvetteML* language should be ready-to-use on multiple middleware. The YML framework strictly separates middleware specific information from the application description. The same compiled application can be executed on multiple middleware. Graphs described by the *YvetteML* language are fully expanded during the compilation process: loops are unrolled, condition evaluated, unvisited branches spread out of the graph and constants are propagated.

The YML framework is separated in two parts. The user view is middleware independent and contains the main services of the YML framework. These services consist in a compiler for the *YvetteML* language, a just-in-time scheduler and a development directory. The middleware independent part is associated to a back-end for middleware dependent services. Figure 1 describes the overall organization of the YML framework. This figure highlights the different parts of the YML framework and shows the middleware specific services known as the back-end. The client positions itself as a standard client for the YML framework. The test-bed platform relies on the XtremWeb[18] middleware.

The list bellow describes the role of each element composing the YML framework:

Compiler: It translates applications described using the *YvetteML* language to a set of components calls. Its component call contains two kind of information. The precondition of the execution is a boolean expression which determines whether the component can be executed or not. The post condition is a list of boolean flags used to describe the state of the application. The preconditions are evaluated based on the events.

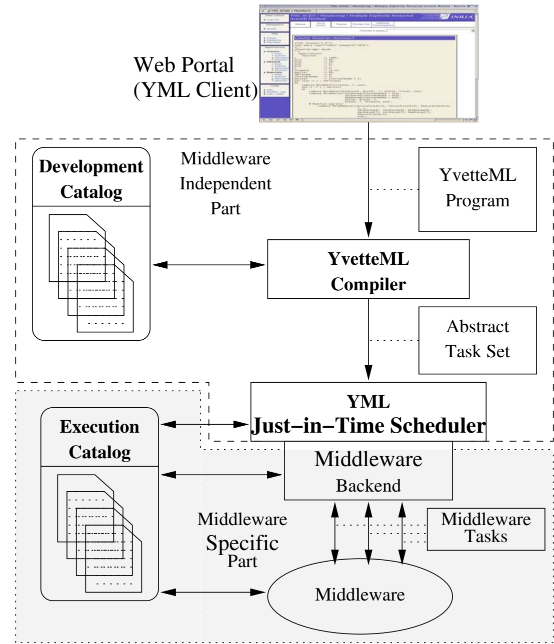

Fig. 1. The YML framework organization

Scheduler: It manages application executions and acts as a client for underlying middleware accurately requiring computing resources. During the application execution the scheduler detects task ready for execution solving dependencies at run-time. Each scheduling iteration may or not generate a set of parallel tasks which are translated in computing requests to middleware through dedicated back-ends.

Development Catalogs: The YML framework stores components in Catalogs. The *Development Catalog* stores information used only during the development stage. The middleware independent part relies only on this catalog. The *Development Catalog* stores components information and data type information used to validate the *YvetteML* input program. This catalog is not required for executing an application.

Data Repository: The YML framework implies a lot of data exchange through the network. The Data Repository server acts as a resource provider and delivers data to each components on demands. Note that the data repository does not appear in figure 1.

Back-end: All middleware specific services are encapsulated in a back-end. A back-end generally consists in a YML worker which constitutes the component execution supporting layer to be executed on remote peer and a client for the middleware itself. It is easy to add support for new middleware in YML. Back-Ends can rely on several services provided in YML such as the Data Repository client library and the component generator. YML comes with a default back-end for the XtremWeb GRID middleware.

C. *The Workflow engine*

The just-in-time scheduler role consists in executing the application on a middleware. The scheduler resolves dependencies between components execution. It relies on the notion of events. Each component execution composing an application has a precondition and a post condition. The precondition is a logical expression based on events. The default state of all events composing an application is not notified. All events constitute the state of a running application. Once an event occurred its state is toggled to notify. Once notified an event can not change its state again during the rest of the execution. The post condition associated to a component execution contains a list of events to be notified once the execution is terminated. The graph exploration or application execution relies on a table containing all events used to describe the graph.

IV. Middleware Integration

YML integrates supports for middleware through the use of an adaptation layer. This layer hides the specificities of middleware defining a set of features required by YML. This adaptation layer is loaded at runtime into the different program composing YML. Programs using the layer are the component generator and the scheduler. All aspects specific to a middleware are packed within what we called a back-end. YML acts like a client of middleware services requiring no modification to support YML.

YML requires only a few features common to all middleware. YML expect from the middleware to be able to execute a remote job. The most important requirement is that each job transmitted to the middleware has to finish its execution. It means that the middleware must to be able to run a component even if some fault occurs. Some middleware do not provide fault tolerance but the back-end for those middleware can define a simple restart on failure fault tolerance mechanisms.

YML makes use of middleware to find and provide computation resources for application execution. The current approach used by existing back-end runs a YML worker for each task composing the application. The worker is in charge of interacting with YML servers to run the task. The worker downloads the computation data from remote data repositories as well as the binary application generated using the component generator and runs it. The middleware registers only the worker which hides all the interactions with YML. This approach has been chosen in order to ease the installation of YML components in the middleware. NetSolve requires integration of components within each server during the installation process. OmniRPC requires the components to be generated locally on each remote host. Using one worker for a whole middleware eases the integration process. It also helps to provide a secured environment for both the application execution and the remote host safety [8]. The worker approach is not required by YML but it is used in its existing and ongoing back-ends.

A back-end defines the interface between YML and the middleware and consists mainly of the following operations:

Task Launcher acts as a client of the underlying middleware. It prepares the computation and launches asynchronously the execution of a component.
Task Retreiver monitors the execution of components on the middleware.
Component Generator is part of the back-end which is dedicated to the component generation. The generation consists in creating the source code of the component in one of the language supported by the framework. This source code is then compiled and translate to a binary application ready for execution on the middleware.
Execution Catalog is the catalog which stores the information related to the underlying middleware.

A back-end mostly consists in writing a client for the targeted middleware. The rest of the section presents two existing back-end.

A. *Process back-end*

The aim of this back-end is to provide the user with a way to test and validate its application in a well known environment : her/his local computer. It rests on the capability of modern operating system to run several process at the same time. This behavior reflects perfectly the way YML works on standard middleware. Being able to test and validate her/his application without any change is really interesting for end-users.

B. *XtremWeb back-end*

XtremWeb is a desktop grid middleware. The targeted networks consist of several independent distant sites creating a unique pool of resources. XtremWeb makes use of remote peers idle time for executing job requests. The architecture defines three roles: dispatcher, clients, workers. The dispatcher manages the whole system organization. It acts like an authoritative entity for the whole platforms and ensures communication between workers and clients. All communication in XtremWeb goes through the dispatcher but are always initiated from workers and clients similarly to standard services on the Internet. This policy is required to bypass security mechanisms such as firewall present on each site involved in the middleware. The problems caused by the presence of firewall prevents any synchronous messaging between peers and even direct communication between peers. XtremWeb allows clients, workers and dispatcher to leave the system at any time. Fault management is one of the priority of the middleware. The system status is maintained even if the dispatcher faults. In this case all workers and clients wait until the dispatcher is available again. In this middleware the client cannot control which peer runs a job.

Figure 2 details the various steps needed to run a job on XtremWeb. We previously explained that all communications go through the scheduler and that XtremWeb does not allow direct communication between workers and also between workers and clients. The execution of a job requires at least 6 steps. The client asks for the execution of a job (1). The active dispatcher stores the query and wait for a job request

from any available workers (2). The dispatcher transmits the job information to the worker (3) which computes the results and sends them back to the dispatcher (4). Asynchronously, the client makes regular check for the job completion (5) until the results are available on the dispatcher (6).

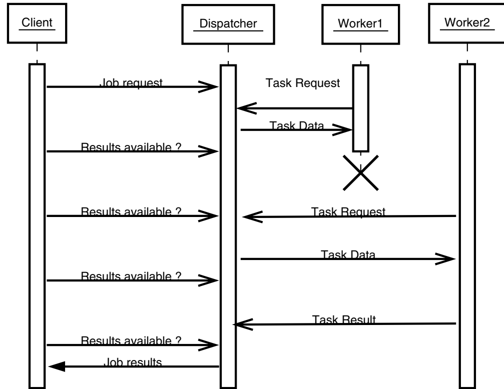

Fig. 2. XtremWeb Middleware: Job execution

In the context of YML this architecture implies high latency and no direct communication between peers involved in a computation. Compared to a GRID middleware based on the Globus Toolkit, the controls usable by clients are really limited. The few mechanisms available were used to define the minimum requirement of an YML back-end. However this subset eases the definition of new back-ends. The communication involved in the execution of a component has to transit through a third party data repository which acts as a communication channel between participants. The scheduler of XtremWeb present in the dispatcher runs jobs using a FIFO scheduling policy. The first submitted job is the first to be executed. XtremWeb is historically the default backend for YML which was motivated by providing workflow capabilities on top of XtremWeb.

V. Case study

In order to illustrate the *YvetteML* language and the various components of YML, this section develops an example based on a numerical method and the integration within YML of an object-oriented numerical libraries dedicated to the linear algebra iterative methods.

A. Application

Numerical methods are classified in two categories: direct methods on the one hand and iterative methods on the other hand. An iterative method computes a succession of approximations of the solution until the accuracy matches the desired one. An iteration is composed of two steps: (a) the computation of the solution of the current iteration and (b) the creation of the initial data if another iteration is needed. In direct methods instead, the number of operations required to construct the solution depends only of the size of the problem.

The eigenvalue problem[19] consists in finding the eigenvalues λ and the eigenvectors x of a matrix A which are the solutions of:

$$Ax = \lambda x \tag{1}$$

Generally, only a few eigenelements are demanding. When dealing with large sparse problems, the standard methods (QR) for solving the eigenproblems are not efficient enough. They must be coupled with a projection method which translates a huge n-size problem to a smaller one with a small matrix $H(m \times m)$, solves the problem with this small matrix and transforms the results back in the initial space. The projection converts the problem from the initial space to a Krylov subspace of base $B(n \times m)$ with similar eigenvalues. The projection acts like a compression algorithm with data loss. The problem reduction makes it possible to solve it using standard methods but with a lot of information loss. That means, in general, the accuracy of the approximated solution is not the desired one. Most matrices requires numerous iterations to converge. Each iteration concludes with the appliance of a restarting strategy. The efficiency of an iterative method highly depends on the restarting strategy which constructs the initial data of the next iteration. The layout of such methods for solving the eigenproblem is :

1) Create the initial vector
2) Iterate until maximum number of iterations is reached:
   a) Projection in the krylov subspace
   b) Resolution in the subspace
   c) if restarting go to (2.a) else go to (3)
3) Pack results

The first iteration begins with the creation of an arbitrary selected initial vector. One can use columns from the identity matrix, or any arbitrary chosen vector for the initial process. Each iteration will produce refinement of this initial vector in order to create the correct subspace with the expected eigenelements contained. This vector is transmitted to the projection which creates the subspace and produces a matrix. The QR process is then applied to the newly created matrix in order to find solutions in the subspace. The results obtained in the subspace are translated to the initial problem and checked for accuracy. If the accuracy of the solution matches the desired one the method is stopped and the eigenvalues and eigenvectors are packed for the user. Otherwise a restarting strategy is applied to the inaccuracy solution allowing to construct a new initial vector for the next iteration.

The explicitly restarted Arnoldi method (ERAM)[20] is an example of such methods which uses an explicit restarting strategy. It means that the restarting strategy creates the next initial data by using explicitly the eigenelements produced during the previous execution. The restarting strategy makes a linear combination of the computed eigenvectors in order to create a new initial vector. The multiply explicitly restarted Arnoldi method (MERAM) is an hybrid method made of several instances of ERAM, also called co-methods, using distinct initial parameter sets. The restarting strategy of an ERAM composing a MERAM takes into account the results produced by the other ERAM. Figure 3 presents an execution of a MERAM with three ERAM processes. Each process

creates different subspaces and the time needed for finding solution in a small subspace is smaller than the time needed for a larger subspace. The creation of the subspaces is also quicker when dealing with small subspaces. Each process is executed asynchronously and exchanges its results with other processes after each of its iterations. It takes into account results obtained by other processes.

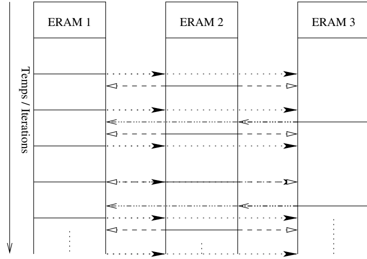

Fig. 3. MERAM execution with three ERAM processes

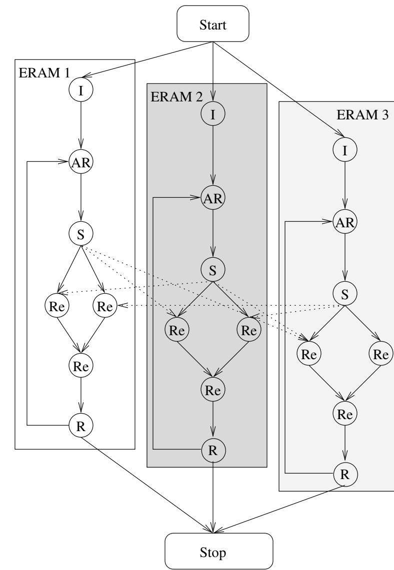

Fig. 4. MERAM Graph using *YvetteML*

B. LAKe integration

Linear Algebra Kernel (LAKe)[21] enables the definition of iterative methods for solving linear algebra methods applied to sparse non-Hermitian matrices. LAKe is an object oriented library which enables very good code re-use between sequential and parallel application. The design of LAKe splits the high level application code from the data representation and distribution. The parallelism is obtained by changing the type of data distribution handled by the program. The rest of the application is unchanged. Those mechanisms are designed around the concept of services which provides modularity in the design of applications. The iterative methods in LAKe are defined as a collection of services. However, LAKe design does not allow the definition of iterative hybrid methods. Indeed, the latter presents a supplementary level of parallelism between the co-methods participating in the hybrid algorithm. Consequently, the sequential and parallel code reuse of a co-method *at the same time* becomes necessary.

The *YvetteML* language splits the description of an application in components on the one hand and in a graph of component calls on the other hand. The MERAM hybrid method makes use of LAKe to define its implementation components while the control flow of the method is defined using the *YvetteML* workflow language. The graph of the method is presented in figure 4. The MERAM method requires the definition of five components. Component named *I* corresponds to the creation of the initial vector for the projection method. *AR* corresponds to the projection method based on Arnoldi. *S* is used to solve the problem in the subspace which consists in finding approximations of the eigenelements. *Re* is a component used to do a reduction. It takes eigenelements from two methods and outputs the best elements from each methods. Using a binary reduction the *Re* components extract the best restarting values from each ERAM process. Finally *R* denotes the restarting strategy. The *YvetteML* implementation component corresponding to the initial vector creation is presented bellow. XML is used for the YML related stuff while the implementation code is in C++. The communication are handled automatically by analyzing the definition of the abstract component `MeramStart`.

```
<component name="MeramStart" type="abstract">
  <param type="Matrix" mode="out" name="v"/>
  <param type="integer" mode="in" name="n"/>
  <param type="integer" mode="in" name="id"/>
</component>

<component name="MeramStart_impl" type="impl"
   abstract="MeramStart" >
  <globals><![CDATA[
#include <Matrix.hh> //LAKe matrix type
]]></globals>
  <source lang="C++" libs="LAKe"><![CDATA[
v.create(n , 1); // Define a vector of 0
v(id, 1) = 1.0f; // set to 1 one value.
]]>
  </source>
</component>
```

An overview of the *YvetteML* graph language follows. It shows the beginning of MERAM and the definition of the loop composing the ERAM process. In this partial graph we do not show the reduction and the restarting of the method. The construction of the language displayed here includes the parallel iterations, component calls, sequential loops.

```
par (id := 1 ; nbProcess)
do # id is the eram process identifier
  compute MeramStart(I[id], n, id);
  seq (i := 1 ; maxIter)
  do
    compute MeramArnoldi(H[id], B[id],
      n, m[id], I[id], id);
    compute MeramSolver(Val[intNodes+id],
      Vec[intNodes + id],
      Res[intNodes + id],
```

```
      H[id], B[id], n, m[id], r, tol, id);
    # Reduction
    # restart
  enddo
enddo
```

VI. Conclusion and Future Works

We presented YML, a workflow framwork for global computing environment. YML is build upon a dedicated workflow language called *YvetteML* and specially defined for the YML environment. The language splits the description of an application in two aspects; a component definition language on the one hand and a graph description language used to link components altogether in a complex workflow on the other hand. *YvetteML* uses XML for the definition of components but the workflow graph is not done using pure XML.

YML does not rely on a specific middleware. One of the goals of its design is to abstract any middleware and to provide all required tools to describe applications independently of the underlaying run-time environment. It requires the definition of a new workflow language based on a component model. This model enforces the separation between middleware specific information and generic information by using two catalogs. The development catalog stores and publishes all informations common to all middleware consisting mainly in abstract and graph component descriptions. An execution catalog is associated to all run-time environment and stores all component implementations available in the run-time. The middleware specific aspects are managed within back-ends. Each back-end deals with one middleware hiding its specific programming job scheduling mechanism to the end-user. Being independent from any middleware forced us to define the smallest set of requirement regarding the underlying platform services. By using YML tools, the end-user is free to change the run-time environment without changing its application.

Several aspects in YML need enhancement and experiments. The *YvetteML* compiler fully expands the graph before execution. This can be interesting in order to validate and optimize the graph. The use of the graph components is a path toward a dynamic graph expansion that can appear at boundaries of sub graph. It can also be used to distribute the workflow engine and remove bottlenecks of the centralized architecture of the workflow engine.

In workflow framework scheduling the application is a critical aspect. YML uses a trivial scheduling policy not taking into account the informations issued from the middleware. While it is already possible to plug new scheduling policies in the workflow engine an enriched scheduling infrastructure must be defined in order to experiment on this critical topic.

References

[1] *YML Project Page*, http://yml.prism.uvsq.fr.

[2] G. Fagg and J. Dongarra, "Ft-mpi: Fault tolerant mpi, supporting dynamic applications in a dynamic world," in *Lecture Notes in Computer Science: Proceedings of EuroPVM-MPI 2000*, S. Verlag, Ed., vol. 1908, 2000, pp. 346–353.

[3] N. Karonis, B. Toonen, and I. Foster, "Mpich-g2: A grid-enabled implementation of the message passing interface," *Journal of Parallel and Distributed Computing*, vol. 63, pp. 551–563, 2003.

[4] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov, "Mpich-v: toward a scalable fault tolerant mpi for volatile nodes," in *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*. Los Alamos, CA, USA: IEEE Computer Society Press, 2002, pp. 1–18.

[5] I. Foster, "Globus toolkit version 4: Software for service-oriented systems," in *IFIP International Conference on Network and Parallel Computing*. Springer-Verlag, 2005, pp. 2–13.

[6] D. Bathia, V.Burzevski, M. Camuseva, G. Fox, W. Furmasnski, and G. Premchandran, "Webflow - a visual programming paradigm for web/java coarse grain distributed computing," *Concurrency: Practice and Experience*, vol. 9, no. 6, pp. 555–577, 1997.

[7] J. Yu and R. Buyya, "A taxonomy of workflow management systems for grid computing," Grid Computing and Distributed Systems (GRIDS) Laboratory, University of Melbourne, Australia, Tech. Rep., 2005.

[8] 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.

[9] D. W. Erwin and D. F. Snelling, "Unicore: A grid computing environment," *Lecture Notes in Computer Science*, vol. 2150, pp. 825–834, 2001.

[10] K. Amin, M. Hategan, G. von Laszewski, N. J. Zaluzec, S. Hampton, and A. Rossi, "GridAnt: A Client-Controllable Grid Workflow System," in *37th Hawai'i International Conference on System Science*, Island of Hawaii, Big Island, 5-8 Jan. 2004. [Online]. Available: http://www.mcs.anl.gov/ gregor/papers/vonLaszewski–gridant-hics.pdf

[11] W. V. der Aalst, L. Aldred, M. Dumas, and A. ter, "Design and implementation of the yawl system," 2004. [Online]. Available: citeseer.ist.psu.edu/vanderaalst04design.html

[12] W. van der Aalst and A. Hofstede, "Yawl: Yet another workflow language," 2002. [Online]. Available: citeseer.ist.psu.edu/vanderaalst03yawl.html

[13] I. Taylor, M. Shields, I. Wang, and A. Harrison, "Visual Grid Workflow in Triana," *Journal of Grid Computing*, vol. 3, no. 3-4, pp. 153–169, September 2005.

[14] B. Ludscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger-Frank, M. Jones, E. Lee, J. Tao, and Y. Zhao, "Scientific workflow management and the kepler system," *Concurrency and Computation: Practice & Experience, Special Issue on Scientific Workflows*, 2005.

[15] T. Tannenbaum, D. Wright, K. Miller, and M. Livny, "Condor – a distributed job scheduler," in *Beowulf Cluster Computing with Linux*, T. Sterling, Ed. MIT Press, October 2001.

[16] M. Wieczorek, R. Prodan, and T. Fahringer, "Scheduling of Scientific Workflows in the ASKALON Grid Environment," *ACM SIGMOD Record*, vol. 35, no. 3, 2005, http://dps.uibk.ac.at/ marek/publications/acm-sigmod-2005.pdf.

[17] Z. Guan, F. Hernandez, P. Bangalore, J. Gray, A. Skjellum, V. Velusamy, and Y. Liu, "Grid-flow: A grid-enabled scientific workflow system with a petri net-based interface," *accepted for publication to the Grid Workflow Special Issue of Concurrency and Computation: Practice and Experience*, 2006.

[18] 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," *Future Generation Computer Science*, vol. 21, no. 3, pp. 417–437, March 2005.

[19] Y. Saad, *Numerical Methods for Large Eigenvalue Problems*. New York: Halstead Press, 1992.

[20] N. Emad, S. Petiton, and G. Edjlali, "Multiple explicitly restarted arnoldi method for solving large eigenproblems," *SIAM Journal on Scientific Computing*, vol. 27, no. 1, pp. 253–277, september 2005.

[21] E. Noulard and N. Emad, "A key for reusable parallel linear algebra software," *Parallel Computing Journal, Elsevier Science*, vol. 27, no. 10, pp. 1299–1319, 2001.