YML : un workflow scientifique pour le calcul haute performance

par Olivier Delannoy

Thèse présentée à

l'Université de Versailles Saint-Quentin
pour obtenir le titre de
Docteur en informatique

YML: A Scientific Workflow for High Performance Computing

by Olivier Delannoy

A dissertation submitted in
partial fulfillment of the
requirements for the Degree of

Doctor of philosophy
in Computer Science

at
the University of Versailles Saint-Quentin

Contents

List of Figures

Listings

Chapter 1

Introduction

1.1 Context and motivations

The simulation of real phenomena such as weather forecasting, natural disaster risks reduction, civil and military engineering as well as nanotechnologies, often leads to the solving of linear algebra problems. The size of these problems increases with the precision of simulations. For such simulations, the use of modern numerical methods decreases the convergence time of traditional solvers. They are well adapted to large scale distributed memory systems.

Nowadays many of these problems cannot be solved in an acceptable amount of time on workstation and desktop computers. A common way to improve performance is to use parallelism and high performance computers. Using parallel applications on adapted hardware can decrease significantly the time needed to get the computed solutions. Scientific applications can also require really huge internal and external storage capacity. Both are limited on commodity hardware.

In order to increase the performance and to be able to solve larger size problems, scientists jointly use several parallel systems with new numerical methods and new programming techniques.

Thus, numerical libraries design and implementation have evolved toward parallel versions. The first libraries available were dedicated to basic facilities or building blocks for applications on a single processor. Those libraries still constitute the foundations of higher level libraries used within sequential and parallel applications. Their design methodology tends to use more and more modularity by introducing reusable software components and object oriented conception.

Nowadays high performance computers gather thousands of processors with hierarchical memory models. Large scale distributed systems are the generalized form of distributed systems. Systems move toward heterogeneous, volatile, geographically distributed ones. The efficient exploitation of such systems, the conception/adaptation of appropriate methods, the programming paradigms as well as the performance prediction are still challenging.

1.1.1 High performance computer systems evolution

Supercomputers were initially driven by a small number of specific processors. The evolution in the number of processors leads to two approaches. Vector based multi-processors tend to use only a few high performance processors. This approach is balanced by massively parallel systems which rely on a high number of micro-processors to achieve performance.

Parallel architectures can be classified depending on the memory layout. There are two classical layouts. Shared memory architectures provide one memory area common to all processing units while distributed memory ones associate local memory to each processor. Based on these two approaches, several mixed solutions have been used in high performance computers. On top of these memory layouts, two main parallel programming models have been used: shared address space and message passing. Each of them can be mapped onto most parallel systems.

The shared memory layout is limited in scalability but programming parallel applications for such systems is a lot easier using OpenMP [138] like API multi-platform programming tools. Distributed memory architectures can scale fairly well. Programming for such environments is most of the time done using a message passing API such as PVM [53] or MPI [75]. MPI has become a *de facto* standard for high performance computer systems.

High performance computers were designed to have specific hardware with especially designed networking solutions. The performance improvement of commodity processors and their wide availability have limited the benefits of using dedicated ones mainly due to their much higher costs. High performance computer systems tend to use commodity hardware coupled with high performance networking solution such as Myrinet or InfiniBand. Such distributed architectures are known as clusters or network of workstations. Clusters have been widely accepted and installed in many computing centers. Programming applications for clusters is most of the time done using MPI. Many clusters include shared memory mechanisms at the workstation level. This leads to hybrid programming models with applications mixing OpenMP and MPI.

Clusters of clusters are a typical example of hierarchical memory layout. The time needed to exchange information between two processors vary significantly depending on whether the two processors are located in the same cluster or in two different clusters. Each cluster is homogeneous and communication time between all processors of the same cluster is about the same. Clusters are interconnected using special nodes called gateways. Gateways are connected together to compose the cluster of clusters. This network topology makes the programming of applications using MPI more difficult. The application distribution must take into account the network topology and assign special roles to gateway nodes.

Numerical libraries were evolved conjointly with high performance computers, this for their adaptation to new parallel and distributed systems. The evolution of libraries mostly concerns the modularity and composition capabilities offered to the user.

1.1.2 Numerical libraries evolution

Numerical applications have been and still continue to be developed and tuned to get the best performance of the various available hardware. Processor vendors are providing optimized numeric kernel libraries such as BLAS [113, 28] (Basic Linear Algebra Subprograms) for their processors (Intel MKL [182], AMD ACML [181], etc). These libraries are used as building blocks for most linear algebra applications. They are composed of a set of independent routines. The services provided by these libraries are simple and consist of operations between vectors and matrices. These libraries are written using C or Fortran. Linear algebra solvers are based on these libraries.

LINPACK (Linear system solver PACKage) [190] and EISPACK (EIgen Solver PACKage) [185] were among the first available libraries to solve linear algebra problems. They have been designed on top of BLAS using the same imperative approach. The two libraries have been superseded by LAPACK (Linear Algebra PACKage) [15] which provides services for both linear systems and eigenproblem solvers. ARPACK (ARnoldi PACKage) [115] provides solvers for larger eigenproblems. It also includes optimization based on the numerical properties of the used matrices.

Following the evolution of distributed memory architectures, parallel versions of these libraries have been developed. Projects like ScaLAPACK (Scalable Linear Algebra Package) [163, 54], PLAPACK (Parallel Linear Algebra Package) [90] and P_ARPACK (Parallel Arnoldi Package) [125] have been implemented on top of MPI and other message passing libraries. The approach used in these libraries consists in building the parallel version by using distributed versions of the basic operations used in LAPACK such as the BLAS. Nevertheless, these libraries allow neither data type abstraction nor code reuse between the parallel and sequential versions of the applications. That means the subroutines of the solvers are not able to adapt their behaviors according to the data types. Those subroutines must be defined once for use in sequential and once again for use in parallel.

The component approach used in libraries such as PETSc (Portable, Extensible Toolkit for Scientific Computation) [20, 21, 19] drastically increased the modularity, interoperability and reusability of high level components within the libraries as well as in the user applications. It increases the ease of use, code reuse, and maintainability of libraries. The object oriented approach, the introduction of generic programming and the use of MPI in LAKe (Linear Algebra Kernel) [135] allowed reusability in sequential and parallel versions of methods. Numerical methods are defined once as a sequential application and self adapt to the data type used. The data type incorporates, or not, support for execution in parallel environments. LAKe defines a solver framework which can directly be used in applications. This approach is also proposed in SLEPc [94, 95, 96, 97].

LAKe goes further in that direction by allowing the code to be the same between the sequential and parallel versions of a method. The method is implemented once and used either in sequential or in parallel. This is made possible by a separation between the computation and the data type representations. Data types are provided for use either in a parallel context or in a sequential one. The maintainability of the library is simplified using this approach.

Extended LAKe [63] adds to LAKe support for asynchronous linear algebra solvers.

Such solvers consist in using several numerical methods (also called co-methods) which collaborate in order to speed up the convergence of a global method. This collaboration is generally done by asynchronous communications between the co-methods. Often, co-methods are different instances of the same method. As a consequence, in the context of heterogeneous computing, one can be led to use, for a given application, parallel and sequential versions of a co-method *at the same time*.

However, the scalability of the reusability offered by Extended Lake is limited. This is because of the limitation of MPI at the time of the design of Extended LAKe. Mapping the application to the available processors is delicate in hierarchical network topology present in large scale distributed systems. Moreover, in the context of these systems, the use of libraries is difficult. They lack high level interface to help non expert end-users to exploit them efficiently.

1.1.3 Toward large scale distributed systems

Lately, the size of high performance computers has increased a lot and thus their price also has. The high performance computers are massively parallel. They are so expensive that alternative solutions are intensively investigated not as a replacement but as a complementary computing infrastructure. In the mean time, Internet speed increased a lot. Home computers are connected to the Internet using xDSL technology. Peer to peer file sharing solutions such as Bittorent [142], Gnutella [145] and Freenet [39] demonstrate the capability of Internet to efficiently broadcast information to computers all across the world. SETI@Home [12] and distributed.Net projects successfully aggregate computing resources to solve scientific problems by using volunteer computers connected to Internet, usually called peers. Those systems rely on inactivity periods of remote computers to exploit their resources. This approach has been generalized by projects such as XtremWeb [33] and BOINC [13]. The latter do not fix the applications supported by the environment. They are used to do embarrassingly parallel applications which decompose the amount of work in independent work units. A centralized master or a fixed group of masters dispatch independent work units to the workers in order to be executed on volunteered peers.

Based on this success, grid and peer to peer middleware have evolved and matured. The main idea of the grid is to aggregate resources of any kind into a consistent system reachable from anywhere. grid systems initial goal and direction were to use computing resources in a way similar to the electric power grid where one can get power by just plugging a device. Where the power comes from does not matter to the user. There is no need to know who provides the computing resources nor where the resources are located. Resources available in a grid are provided by institutions who agreed to share computing devices according to a contract. The programming tools available on grid are variants of message passing libraries, remote procedure call libraries and, more recently, workflow environments.

Peer to peer systems use hardware provided by volunteers. Peers connected to the system are heterogeneous in hardware and software. They are insecure and highly volatile and there is no trust between participants. Peer to peer systems classification is based on the way the management of the middleware is done. A centralized management area is used in systems such as XtremWeb, BOINC and Bittorent. The managing peers in such systems are trusted and stable. The second type of peer to peer systems does not require any management dedicated resources. In such systems all peers are or can have several roles at the same time. Common roles include : client, server and manager. Projects like JXTA [189] and Gnutella follow this model. The main programming model available in peer to peer environments is RPC. The discovery of remote services often depends on ads publication and search systems. A peer publishes its capabilities to a system of ads. When a consumer wants a resource, it looks for corresponding ads in the ads look-up service and contacts the remote peer.

1.1.4 Motivations

Compared to standard high performance computers, large scale distributed architectures programming is difficult due to several new constraints. Computing resources are volatile. They can appear and disappear at any time during the execution of an application. Resources are heterogeneous. They can have different computing capacities and software environments. Lastly, the networking capabilities can change significantly from one resource to another.

Despite the increasing complexity of large scale distributed systems such as grid and peer to peer, classical parallel programming models apply. Indeed the same paradigms (data parallelism, message passing, remote invocation, virtual shared memory) are used with the new constraints of the volatility and heterogeneity of resources. Middleware tends to provide a virtual execution environment. It relies on several software layers to simulate operations originally provided by the hardware in high performance computers. This approach is appealing to the end user because it allows him to use its existing application or its programming skills directly. This is especially true for standards like MPI as discussed in [104] for the Globus Toolkit and in MPICH-CM [151] for peer to peer systems.

Large scale distributed architectures still lack standardization. Environments are not yet interoperable and applications developed for one middleware are most often difficult to adapt to other systems. The Global Grid Forum which evolved in the Open GRID Forum [191] is working on defining standard for grid middleware. The standardization process extends works made by the W3C [196] in the context of Internet and web services technologies. One of the important standardization process in regards of the application development is the GRID-RPC [188] specification which is not finished yet. Independence of the application in regards of the underlying execution environment is especially important due to the number of different middleware deployed over the world. Being able to use resources from several virtual organizations transparently is also appealing.

Even without widely accepted standards, large scale distributed systems all provide a mechanism to request the execution of a work unit on a distant resource. This can be assimilated to remote execution mechanism (RPC). This facility is enough for applications relying on embarrassing parallelism. They are based on a collection of independent work units to be scheduled on available resources. The scope of applications which can be made in parallel by using this approach is limited. Many parallel applications have dependencies among work units.

In order to enable a wider scope of applications, and to be interoperable between several middleware, we have to rely on RPC mechanisms. They allow remote execution of independent work units. On top of RPC, we must be able to manage dependencies. Doing this task manually for each application is fastiduous and impossible for non expert end-users. The management of dependencies has to be automated and delegated to a specialized tool which controls the execution of the application.

To cope with the increasing complexity of scientific applications which incorporate more and more often several domains of research, the programming environments must provide assistants to drive the creation process of applications and exploitation of their results by users. Applications are more and more often composed of software issued from multiple scientific communities. In order to create an application as a collection of domain specific existing software, developers need to specify interface at the boundary of each domain. The programming environment must provide a strategy to isolate application domains and enforce the separation between the implementation and the interface of each service.

The problems mentioned above can be addressed using a scientific workflow. It is composed of a set of tools to support creation and execution of e-Science applications. The latter are defined as a set of work units which have to be executed in a particular order to obtain the results. Work units are linked to each other using data migration operations. A wortkflow enactment software manages the ordering of work units and data migration The user is solely responsible of defining work units and the pre-ordering of them.

It is necessary and interesting to dispose of a workflow environment which enables the realization of an application for large scale distributed systems. This workflow must also integrate tools to assist during the pre-processing, the processing and the post-processing of applications. The pre-processing consists in preparing the execution of the application such as data decomposition, pre-conditioning, data source selection, etc. The processing corresponds to the execution of the application and the integration of services. The post-processing includes everything related to the interpretation of the results and their visualization. Such workflow environment should allow users to follow the execution of its application. End-users of this kind of systems could be interested in live collaboration with other users around the world.

The main objective of this thesis is to bring a response to these issues by proposing a scientific workflow for large scale distributed systems. As we will see in the next section our workflow is based onto an as general as possible architecture model and covers a wide spectrum of the process of programming, execution and visualization of the results of complex applications.

1.2 A workflow for large scale distributed system: YML

In order to answer our requirements described in the previous section, we have designed and realized a workflow, named YML. We propose a solution which can be adapted to several middleware as well as many application domains and software environments. We adopted a component oriented approach. A component is constituted of a public interface and an implementation which is hidden to the rest of the application. Components interact with each other using their public interface only. Using this approach component implementation can be changed easily to adapt the model or its realization to new contexts.

The benefits of a component oriented design are numerous. The component approach enforces the definition of communication interface between entities. The model can also be specified at various levels of abstraction. Most high level components could also be defined as a collection of components themselves. The advantages of a component oriented design are even more significant when we consider the realization. The implementation of a component can be changed without consequence on the rest of the system. It allows anyone to adapt YML to multiple middleware, to use external libraries in our workflows, to experiment multiple scheduling strategies and many other aspects of the workflow.

1.2.1 Workflow and architecture modeling

We first propose a modeling of a scientific workflow as well as a modeling of targeted architecture. We define and implement then our worflow according to those models.

1.2.1.1 Workflow model

We propose a workflow model which consists in three main layers. The front-ends layer manages interaction with end-users. The back-ends layer brings interoperability between middleware. Lastly, the kernel layer is located in between and provides a set of modules which support the execution of workflow processes.

The front-ends layer contains end-users tools to interact with the workflow. Those tools fulfill the needs of both workflow process definition and their execution. They consist in graphical user interface, web portals and command line tools.

At the opposite direction of the end-users lies middleware. The back-ends layer sits on top of middleware and is responsible of the interaction with them. The main goal of this layer is to ensure interoperability of scientific workflow processes in regards of middleware. That permits the same process to be executed indifferently on various middleware.

On top of the back-ends layer, the kernel layer provides a collection of services used all along the lifetime of a workflow process. It also provides mechanisms to incorporate external services within processes. Examples of external services are libraries, databases access, etc. The kernel layer is thus divided in two aspects: external services on the one hand and workflow process management on the other hand. This last integrates the process definition language, and modules enabling compilation and scheduling of workflow processes.

The organization of our workflow model is described in figure 1.1.

1.2.1.2 Architecture model

The numerous existing global computing middleware differ in several aspects. However, it is possible to define a model to which almost any middleware can be mapped. This model, the one we based our solution on, defines the concept of peers. Peers correspond to all participants in the system. They provide resources of two kinds: processor time and

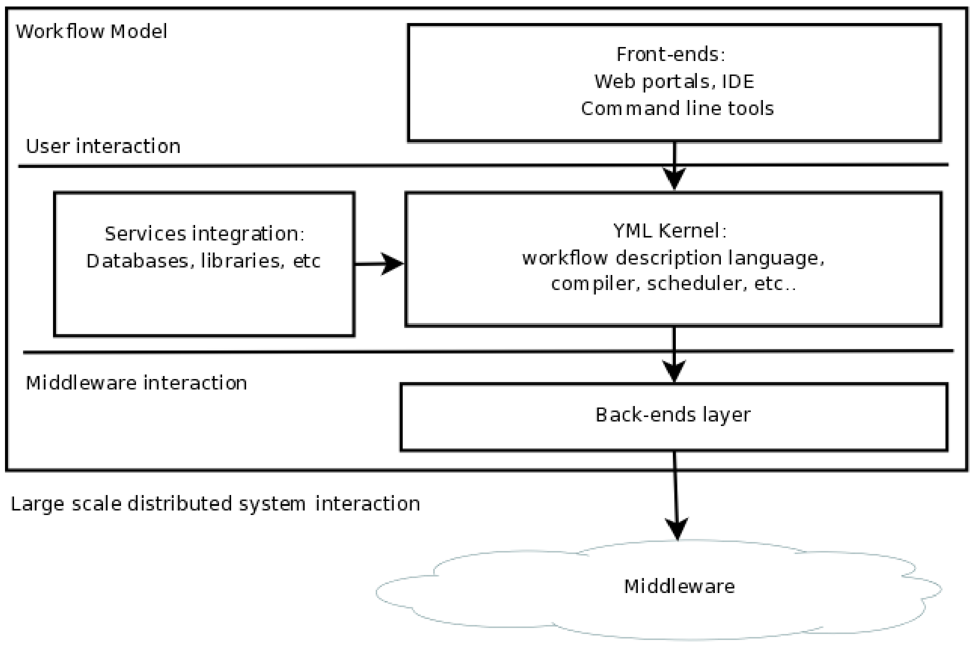


Figure 1.1: Workflow model overview

data storage. Our model also defines the behavior of peers relative to the communications and the faults.

Communications: The model assumes that direct communication between peers is not possible. Indeed, Internet is a network of networks. Each network can apply its own security policy which relies a lot on the principle of isolation using firewalls. The aim of a firewall is to protect a set of computers of the same administrative group from outside, for example a corporation. Firewalls allow outgoing communication and prevent incoming connexion requests. For a distributed system they introduce a lot of complexities.

In order to bypass firewalls and permit distant peer groups to participate in the same system, a solution consists in making use of a third party peer that can communicate with all distant sites and acts like a proxy or a mediator between the different groups. Consequently, our constraints assumption on not availability of direct communication could be raised for some peers. Indeed, it is possible to define a set of peers accessible from all peers participating in the system. Those peers are used to exchange data between distant peers. They provide virtual communication channels. We assume that indirect communication is always possible. The peers providing virtual communication channels are named data warehouses. This is because data warehouses are regular peers accessible from any other peer involved in the system. A peer expecting data will ask the data warehouse for a resource and waits for the other peer to store this data in it.

Faults: In large scale distributed architectures, fault tolerance policy is a critical aspect. Ideally, the model should deal with faults by allowing any peer to fault at anytime. However, in our model, peers are organized in two groups depending on whether they are allowed to fault (unstable) or not (stable). Stable peers are dedicated to the system management and they are not allowed to fault. Unstable peers are used for computing and allow faults at anytime. The model assumes that the workflow framework itself as well as data warehouses are assigned to stable peers during the execution of an application.

Figures 1.2 presents an overview of the architecture model. The solid arrows correspond to direct communications. Direct communication between two peers are often blocked by firewalls. Dashed arrows represent indirect communications.

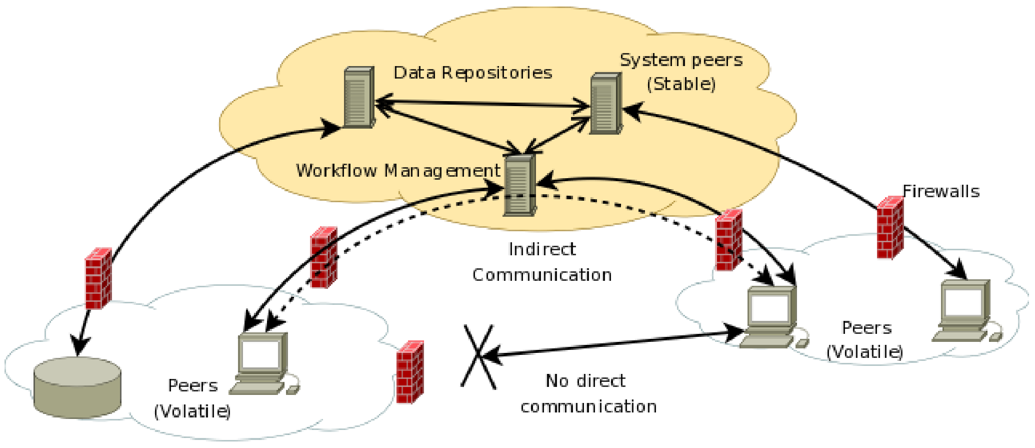


Figure 1.2: Architecture model overview

1.2.2 YML workflow realization

We have designed and implemented a scientific workflow, named YML, according to the models established in the last section. Consequently, YML is defined as a collection of collaborating components. The implementation of YML has been done using C/C++ as a primary language. It also relies on XML for most of its data files.

In order to present YML realization, we first illustrate the interaction with users by describing front-ends. We then discuss the interaction with middleware through back-ends. Lastly, the YML kernel and the integration of external services will be presented.

1.2.2.1 Front-ends layer

The front-end layer provides the user with several complementary tools to work with YML. According to our model, a front-end is a software allowing users to access YML kernel components. We identify two categories of user for YML. The applications developers on the one hand need tools to help in the creation of applications and the integration of third party services as YML components. On the other hand, some users are only interested in running applications, monitoring their execution and finally analyze and visualize the results obtained during and at the end of the execution of an application. Both categories of users have distinct needs and goals. YML proposes two graphical user interfaces (GUI) to support the two categories of users.

The *integrated development environment* (IDE) proposed by YML helps at the creation of an application. It is composed of a set of wizards. Wizards assist the user during the creation of workflow processes as well as the integration of services. The IDE would be even more useful if extended with a visual editor to define workflow processes based for example on UML diagrams [141, 69, 56].

The *web portal* is used once an application has been defined. YML application can be integrated within a *light* web portal. The portal provides a platform which incorporates all important steps of the exploitation of an application. It supports the operations related to the pre-processing, the execution monitoring based on log analysis, the post-processing and the visualization of results. The portal provides a common basis for all applications allowing submission and monitoring. The monitoring is based on a set of views or reports displaying statistics collected in the application logs. A set of views already exists, collecting information such as peers contribution, application statistics, application progress, and a few others. Visualization of results is supported through user defined views. This web portal has been used to present YML and hybrid linear algebra methods during the Super Computing(SC) conferences in 2005, 2006 and 2007.

The *IDE* and *web portal* rely on a set of command line tools. They support individual operations such as service generation and registration, workflow compilation and scheduling. The command line tools are based upon components defined in the kernel layer.

1.2.2.2 Back-ends layer

In accordance with our model, a back-end is the interface between workflow and middleware. A back-end is a component which grants access to middleware services such as the submission of work units and gathering of information about available peers.

The back-end manager aggregates several back-ends. It enables an application to be executed jointly on several middleware. This component can also be shared between applications. The back-end manager is still in an early state of research and needs further investigations. However it opens research topics including quality of services, load balancing and allocation based on sub-workflow definitions.

Back-ends are most often used during the execution of workflow on large scale distributed systems. They can also be considered as tools to assist users during the conception and development of applications. YML provides dedicated back-ends to assist developers to create workflow processes. They simulate workflow executions using degraded execution on a single peer, allowing the output of metrics, etc.

In its current version, YML supports two middleware: XtremWeb and OmniRPC. The back-ends layer in general and particularly that of OmniRPC are realized in collaboration with Laurent Choy[1]. The corresponding back-ends allow the execution of YML applications on a global peer to peer computing system and a grid environment. YML also provides two back-ends to assist the user during the creation of applications. Other back-ends are currently being added by researchers. Toussaint Guglielmi, from the university of Lille(France) contributes to create a back-end for Condor. Pierre Mannenbach

[1]INRIA Grand Large/LIFL, University of Lille

and Sebastien Noel from PolyTech Mons (Belgium) are currently providing a back-end for the Distributed Resource Management Application API (DRMAA) [162]. This back-end allows YML to be executed on top of middleware such as Sun Grid Engine [195], Condor [116], PBS/Torque, GridWay [100], EGEE [25, 175] and UNICORE [65].

1.2.2.3 Kernel layer

According to our model, the kernel layer interacts with all other layers of YML. Components of this kernel are oriented toward workflow management. They make use of components defined in the back-ends and services integration layers. This last is discussed in the next subsection. This kernel provides a compiler and scheduler components for workflow processes.

As we have shown on figure 1.1, the kernel layer defines a workflow description language. This language associates standard control flow patterns of imperative programming language with mechanisms to support asynchronous execution and synchronizations of concurrent control flows or threads.

The compiler component translates the workflow process definition into an oriented graph. Each node of the graph corresponds to the execution of a service on a peer. Edges represent dependencies between two service executions. Each node of the graph is translated into a rule composed of a pre-condition, a service invocation and a post-condition. The compiled workflow process is interoperable between several middleware.

The scheduler component is responsible for the execution of an application. The enactment of a workflow process consists in coordinating or orchestrating the execution of asynchronous services. The scheduler also controls data migrations. It depends on an expert system which manages all rules generated during the compilation of a process. Each time a service execution finished successfully, its post-condition is applied to the expert system. This activates new rules and the submission of new remote computations.

1.2.2.4 Services integration

The last layer of YML deals with the integration of external services. In YML, a service denotes a computation (or a database access, etc.) on a remote peer. An application is composed of a set of services collaborating with each other to produce a workflow process. Like any other aspects of YML, its services are represented by components. They are defined using XML documents.

In order to manipulate and discover services, YML defines two catalog components which interact with the kernel layer. The integration of service also provides a *services generation component*, a *services registration component* and the two catalogs mentioned before.

YML supports three kinds of services: abstract, concrete and graph services. An abstract service defines a public interface. This interface contains information such as the number of parameters required to execute a service, their type and the communication patterns.

Abstract service definition is used to generate a skeleton of a concrete service. The latter needs to be completed by calls to third party libraries to produce a concrete service.

Several concrete services can be generated from a single abstract service. YML provides another family of services to realize a public interface. It is possible to use graph services. They consist in sub-workflows which can be instantiated to compose larger workflows.

The *services generation component* is used to create skeletons for creating concrete services. In the current version of YML, one can write concrete services using C/C++. The component can be extended to support new programming languages.

The *services registration component* is used to store services in the catalogs mentioned before. Abstract and graph services are stored in the development catalog while concrete services are stored within the execution catalog. The two catalogs are independent from each other and are used respectively by the compiler and scheduler components.

In order to illustrate the integration of services and to implement hybrid methods for eigenproblems, we integrated the LAKe object oriented library with YML. The component oriented design of YML makes the *easy integration* of other libraries such as LAPACK possible.

1.3 Validation and applications

To validate a programming environment such as YML , the following two questions have to be answered.

1. Feasibility: Is it possible to define and exploit e-Science applications in a reasonable amount of time using the language, abstraction, tools provided by the environment?

2. What is the performance of the targeted applications defined using the programming environment? In the context of YML, the performance must be evaluated for several representative large scale distributed systems.

1.3.1 Feasibility

Despite the initial orientation of YML toward numerical applications, it does not contain any domain specific constructions. The scope of applications which can be implemented using YML is wide. In order to highlight families of applications well suited to YML, we choose and show - in the next section - the efficient exploitation of some representative examples from three different domains. These applications are from numerical simulations, as well as sorting and image synthesis.

One of the initial goal of YML is the transparent use of middleware. By making use of YML, the user does not need to know all the details related to the use of one or several middleware. The maintenance of applications is simplified as well as their portability. Each time the middleware API evolves, all applications need to be updated. The maintenance operation is required for all applications. However, if one uses YML, only the back-end needs to be updated. The maintenance of applications related to the middleware is significantly reduced. Application are also defined once for all supported middleware; the user does not have to maintain several versions of its applications if he/she intents to use them over several middleware. Lastly, end-users can easily choose the best suited middleware for a particular application depending on constraints such as performance, quality of services, budget and some others.

YML is based on a component oriented approach. This design introduces a lot of modularity at several levels. New back-ends can be added to support new middleware. New scheduling strategy components can be attached. This aspect is introduced in [134] in the context of quality of services. The proposal of some components containing scheduling strategies based on the definition of the parameters which impact on the execution time of applications is currently worked on by Mohamed Jemni and Nabil Rejeb from the university of Tunis(Tunisia). Services integration consists in adding new components to YML which can be used during the conception and the execution of workflow processes. Scientific libraries, could hence *easily* be integrated as a collection of components.

1.3.2 Applications and performance evaluations

Performance evaluation of applications which use YML is difficult due to the numerous factors that might impact on the execution time. In order to present the performances, we rely on several metrics and correlate them with the complexity of applications in term of operation, space and communication. Performance evaluation of a system like YML - which relies on maximizing usage of resources by techniques of cycle stealing - should not be based solely on time. However, it is out of the scope of this dissertation to define a better performance evaluation strategy.

We evaluate the performance of YML through four applications from three domains. The first one is a sorting algorithm which can be applied to arbitrarily large data sets. The second application implements post-processing filtering of images created using a distributed ray-tracing application. The last two applications are related to linear algebra methods.

The first application is a naive sorting algorithm defined to work on memory limited peers. The application first generates a large collection of random numbers in chunks. These chunks are then manipulated by a merging operation which takes into account at most two chunks. We present our results on data sets composed of a millions of elements.

The second application presents the use of YML in the context of image synthesis. The application generates a wall of images representing a 3 dimensional scene. A distributed ray-tracer is used to create the wall of images as a collection of square windows. The application then allows the execution of post-processing operations such as filtering on the resulting image. Examples of filters include gaussian-blur and edges detection.

The last two applications are used in the solving of linear algebra problems. Linear algebra methods and especially iterative methods for the solving of the Eigenproblem make an intensive use of the matrix vector product. Examples of such methods include: PRR, Arnoldi projection and the power methods. The matrix vector product is also used many times during the product of two matrices. Other implementations of this applications, are studied by Zaher Mahjoub and Olfa Hamdi from the University of Tunis El Manar. They are evaluating the performance of the matrix-vector product with different sparse matrix storage schemes [61] on top of YML and XtremWeb-CH [2, 3].

The community studying linear algebra methods for Eigenproblems is more and more interested in hybrid methods. These methods benefit from heterogeneity. They also rely on asynchronous coupling of several co-methods which collaborate to decrease the time needed to compute the solution. Thus they are good candidates for large scale distributed systems. We implemented the multiply explicitly restarted Arnoldi method [62] (MERAM) which uses as its co-methods several instances of ERAMs. We present our results and validate them by showing that they are similar to the one obtained on other meta computing systems such as Netsolve [16, 64]. Meanwhile, we present results for larger problems.

1.4 Outline of the document

This thesis is organised as the following:

Chapter 2 presents the state of the art of high performance computing and large scale distributed systems. We present the evolution of systems, models, and tools oriented toward high performance computing. We point out how this evolution leads to the large scale distributed systems and the emergence of workflow in the context of e-Science applications.

Chapters 3 to 6 concern the description of YML. Chapter 3 defines YML and gives the outline of the presentation related to the modeling and realization of YML. Chapter 4 focuses on the front-ends layer of the framework and its interactions with end-users. Chapter 5 presents one of the most important aspect of YML. It discusses all aspects related to the interaction between YML and middleware. Chapter 6 discusses the kernel layer and the integration of services. It first presents integration of external services within YML and the coupling between YML and LAKe used to create lowly coupled methods such as hybrid ones. It also depicts the workflow management which consists in the workflow process definition language, and the compiler and scheduler associated to that language.

Chapters 7 to 9 concern the validation of YML. Chapter 7 focuses on the study of the feasibility, an *a priori* evaluation of YML. Chapter 8 defines the four representative applications used during our experiments and proposes a study of their complexity in terms of operations, space and communications. It also presents the workflow processes used for each application. Lastly, chapter 9 presents experimental results obtained on the Grid'5000 testbed and a real platform composed of nodes located in Lille (France) and Tsukuba (Japan).

Finally, chapter 10 presents our conclusions and perspectives of research.

1.5 Notations

Here is a list of notations used in the rest of the document.

A	A square matrix.
n	The order of the matrix A.
nnz	The number of non zero elements of a sparse Matrix A.
r	The number of eigenelements expected.
λ	An eigen value.
u	An eigen vector.
m	The size of the krylov subspace.
tol	The accuracy of the solution computing by the iterative method.
Res	The restarting strategy used in MERAM
Red	The reduction strategy used to select the eigenelements used during the restarting
$Init$	The initialization strategy used in MERAM
bs	The size of block for data composed of collection of blocks
bc	The number of blocks composing a set.

Chapter 2

The state of the art

2.1 Introduction

A parallel computer is *a collection of processing elements that communicate and cooperate to solve large problems fast* [8]. This definition can be applied to most nowadays computers, desktops with multiple cores or hyper threaded processors, workstations with several processors, and high performance systems dedicated to scientific problems. The need for computing resources increases jointly with the order of scientific problems. Results obtained open new research areas which lead to more complex simulation models. The more accurate the results become the more experiments and simulations are needed. The amount of computation required increases faster than the computing capacities.

To support the demand of computation capacity, high performance systems evolve at a fearsome speed. They will soon reach the capacity of executing a peta floating point operations per second. In a little more than ten years, high performance computers will have multiplied their capacities a thousand times. This performance is possible thanks to the evolution of processors, networks, operating systems and programming environments.

In order to study the evolution of parallel computers, it is not enough to focus on the hardware. A high performance computer is always associated to multiple software layers involved in the management of the global architecture. Figure 2.1 is a simple representation of the main layers of a system which relates to application development activities. In this chapter, we focus on a presentation of the evolution of the hardware, parallel programming models and their realization for high performance computers.

In the mean time, the networking infrastructure supporting Internet evolved too in order to support massive and numerous data transfers, multimedia applications, games and more generally interactive contents. The number of computers connected to Internet has increased significantly too. The amount of potential resources connected to Internet is large enough to justify research for a *new* kind of high performance systems based on computers connected to each other using wide area networks(WAN). They are known as meta-computing systems. Several approaches exist in order to aggregate computing resources distributed in multiple locations. These approaches are known as grid, global computing and peer to peer systems. In the mean time, Internet evolved in a set of services which collaborate to provide more complex treatment to end users.

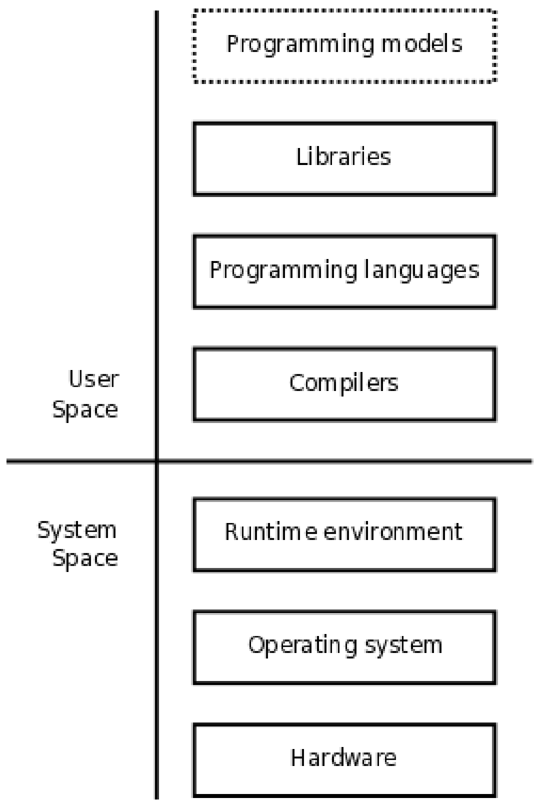


Figure 2.1: General overview of a computer system

Recently, the evolution of processors took a new direction. Indeed, the increase of the frequency is no more a possible solution to gain performance. However, the Moore law [130], which specifies that the number of transistors composing a processor doubles every eighteen months, still applies. Processor architects introduced the concept of cores. A processor is now composed of several cores following the MIMD model. Processors composed of multiple cores are already available on high performance systems, workstations, desktop computers and laptops.

The number of threads or control flows which will be executed concurrently on next generation high performance systems will be of a few millions. The issue related to the handling of large scale distributed systems are the same in the context of meta computing systems and future high performance systems. They will need to provide mechanism to handle heterogeneity, fault tolerance, scheduling, etc.

In this chapter, we first present the evolution of high performance systems by means of architecture changes, parallel programming models and tools to realize them. Section 3 is dedicated to large scale distributed systems and the evolution of the programming environment for these systems. Lastly, the section 4 discusses solutions based on workflow and service orchestration which have taken a significant role in scientific collaboration in multi-domain science.

2.2 High performance systems evolution

2.2.1 Classification and evolution of hardware architectures

The content of this subsection is inspired by [168] and [42]. A computer is the realization of an execution model. Execution models can be classified according to the taxonomy of Flynn [74]. This classification is based on the way of manipulating instructions and data streams and comprises four main architectural classes.

SISD machines : These are the conventional systems that contain one CPU and hence can accommodate one instruction stream that is executed serially.

SIMD machines : Single Instruction Multiple Data systems often have a large number of processing units, ranging from 1,024 to 16,384 that all may execute the same instruction on different data in lock-step. So, a single instruction manipulates many data items in parallel. No such machines are being manufactured anymore. Nevertheless, the concept is still interesting and it may be expected that this type of system will come up again or at least as a co-processor in large, heterogeneous HPC systems. Another subclass of the SIMD systems are the vectorprocessors. Vectorprocessors act on arrays of similar data rather than on single data items using specially structured CPUs. So, vector processors execute on their data in an almost parallel way but only when executing in vector mode.

MISD machines : Theoretically in Multiple Instruction Single Data machines multiple instructions should act on a single stream of data. As yet no practical machine in this class has been constructed nor are such systems easy to conceive.

MIMD machines : Multiple Instructions Multiple Data machines execute several instruction streams in parallel on different data. MIMD systems may run many subtasks in parallel in order to shorten the time-to-solution for the main task to be executed. There is a large variety of MIMD systems and especially in this class the Flynn taxonomy proves to be not fully adequate for the classification of systems. A wide variety of systems that behave very differently fall in this class.

In the remainder of this section we will mainly focus on MIMD machines. The Flynn taxonomy is not enough to classify architecture for high performance systems. Almost all nowadays systems fall in the MIMD class of machines. However, the classification of Flynn can be refined for MIMD systems based on the memory model used.

2.2.1.1 Shared memory systems

Shared memory systems have multiple CPUs all of which share the same address space. This means that the knowledge of where data is stored is of no concern to the user as there is only one memory accessed by all CPUs on an equal basis.

In shared memory systems, the communication between processors occurred through memory accesses. The memory controler is often more complex than in other systems. It is responsible for retrieving memory area accessed for reading and writing and to maintain the consistency of local caches. Shared memory systems used to depend on a single memory area which was accessible from all processors of the systems. In this kind of architecture, a bus or a network connects processors to memory. This approach quickly showed its limit with regard to the number of processors which can be integrated. Indeed, the number of processors is limited either by the number of concurrent access or by the bandwidth. The scalability of high performance systems such as the IBM SP series or the SGI Origine 2000, uses a different approach to allow scalability. Non uniform memory access (NUMA) architectures were introduced to bring more scalability to systems providing shared memory. NUMA systems distribute the memory to each processor. In such systems, the network of interconnexion is still used for all memory operations (read/write). However, the round trip time to retrieve a memory area is not fixed and varies depending on the distance between the two processors involved.

2.2.1.2 Distributed memory systems

In distributed memory systems each CPU has its own associated memory. The CPUs are connected by some network and may exchange data between their respective memories when required. In contrast to shared memory machines the user must be aware of the location of the data in the local memories and will have to move or distribute these data explicitly when needed. Distributed-memory MIMD systems exhibit a large variety in the topology of their connecting network. The details of this topology are largely hidden from the user which is quite helpful with respect to portability of applications.

The main difference between a NUMA shared and a distributed memory systems lies in the integration with the network of interconnection. In shared memory systems, the network interacts with the memory controller. In distributed systems, the network of interconnection is connected to processors instead. This approach is often preferred over the integration with the memory controller. Example of architectures which are based on this memory model include the CRAY T3E, Fujistsu AP3000 and networks of workstations.

The aspect which evolves the most in these kind of systems is the network of interconnection. Many topology have been evaluated. This kind of architecture is really similar to clusters where nodes are workstations built upon widely available processors and networking solutions. A cluster is constituted by a collection of nodes connected to each other by a network of interconnection. In clusters each node provides persistent storage, one or several CPUs, local memory and runs its own operating system. In order to improve performance, enhanced networking solution can be used instead of Ethernet (Myrinet, InfiniBand, etc).

2.2.1.3 Convergence and future systems

The convergence of high performance systems concerns CPUs, memory architecture as well as future systems. Systems used to have specific processors designed for high performance. The cost of such processor is so important that most systems now use regular processors available on the market. Vector processors have almost disappeared from high performance systems and have been replaced with scalar processors. High performance systems also converge on the aspect of the network of interconnection. Indeed, a large number of the fastest computers in the world are based on a network of interconnection which usez the same technology as Internet.

The distinction between shared and distributed memory architectures also tends to disappear. Shared memory architectures are most of the time emulated on top of distributed memory systems[106, 57, 105, 22]. Moreover, the growing number of high performance clusters which are typical distributed memory architectures tends to one class of architecture. Other class are emulated by the runtime environment.

Future systems also converge. The number of processors of next generation architecture is growing. The introduction of multiple cores in CPUs will speed up this tendency and the number of concurrent threads executed on the next generation of systems will be of the order of the million. In the mean time, systems tend to introduce extension mechanisms to support coupling of dedicated processors into the system. Several projects couple regular processors with FPGA, Cell or GPU. Systems are becoming heterogeneous. Thus, programming models and environments will soon need to follow this evolution to benefit from asynchronous execution and heterogeneity.

In conclusion, high performance systems evolved toward a unique architecture model based on distributed memory. Recent evolution tends to integrate more and more CPUs which contain multiple cores. They also introduce mechanisms to couple regular CPUs with dedicated ones. High performance systems are more and more modular. The basis of the high performance systems is a network on top of which nodes are interconnected. Nodes can be tuned to best fit the needs of applications. An example of such system is RoadRunner[112] which integrates IBM Cell processors [154, 58]. The heterogeneity and the introduction of specialized processor within the system is a way to increase the efficiency of architectures by means of power consumptions. Bleeding edge systems try to increase the efficiency in term of watt per flops. By using specialized processor like GPU or CELL the efficiency can be improved for many classes of applications.

2.2.2 Parallel programming models

A parallel programming model must allow a programmer to express:

- Concurrency: several activities happen at the same time.
- Synchronization: the coordination of several activities.
- Distribution: data assignation to concurrent activities.
- Communications: data exchange between concurrent activities.

These four aspects characterize the parallelism used in an application. They are linked to each other and constitute all together a parallel programming model. In this section we present the two parallel programming models: the data parallel and the task parallel model.

2.2.2.1 Data parallelism

Data parallelism consists in defining a single control flow which is common to multiple data. In this model the programmer is only able to explicit the data distribution. Other aspects of the parallelism are expressed implicitly. It corresponds to the SIMD execution model. The same instructions are executed synchronously by all processing elements. Data parallel languages allow the definition of a virtual topology of processing elements. This topology is mapped on the read processing units available on the system. Processing elements can communicate only with nearest neighbors. The underlying topology was most often a two dimensional grid. This topology is well adapted to operations on regular data sets such as matrices and images. Connexion Machines, like the CM-5, were especially design to support data parallel algorithms and applications. Vectorization is a adaptation of the data parallel model. Nowadays vector instructions are available in most general purpose processors.

A data parallel application first defines a virtual topology of processors such as a 2D grid. This topology is then mapped on real processors during the execution of the program. The program defines a single flow of execution which is executed by all virtual processors concurrently. Instruction of the program consists in traditional instructions and in communication operations with neighbors processors in the virtual topology. Many linear algebra applications can be efficiently implemented using this model [60]. A more detailed description of concepts, tools and languages associated with this model can be found in [140].

2.2.2.2 Task parallelism

Task parallelism consists in defining an application as a collection of control flow units. Control flow units collaborate using either explicit or implicit communications/synchronizations. In this model, each control flow unit manages a private memory area. It collaborates with other control flow units to create the global action of the program. Many realizations of this model exist. At the opposite of the data parallel model, the task parallel model allows the programmer to explicitly control every aspects of the parallelism. She/he is responsible for explicitly managing the concurrency, synchronization, distribution and the communications. Using the task parallelism model it is possible to emulate data parallelism. This approach is named SPMD for Single Program Multiple Data.

2.2.3 Parallel and distributed programming tools

In the remainder of this section we present some programming environments used on high performance systems. Those tools allow applications to exploit one or both of the parallelism introduced before.

2.2.3.1 Message passing tools

Parallel Virtual Machine [53, 194] (PVM) is a software system that enables a collection of heterogeneous computers to be used as a coherent and flexible concurrent computational resource. The individual computers may be interconnected by a variety of networks. PVM runtime executes on each machine in a user-configurable pool. It exposes to the application a unified computational environment for concurrent applications. The user defines programs using language C, C++ or Fortran which incorporate calls to the PVM library routines for facilities such as process initiation, message emission and reception, synchronization mechanism using barriers and rendezvous. The PVM system transparently handles message routing, data conversion required within an heterogeneous network environment.

On top of the PVM package, the Heterogeneous Network Computing Environment (HeNCE) [24] simplifies the task of writing, compiling, running, debugging and analyzing programs on a heterogenous network. HeNCE provides a graphical application editor relying on the definition of the graph of the application. The HeNCE runtime manages the execution of the application by solving dependencies between the nodes of the graph.
Message Passing Interface [75, 76] (MPI) is a standard that has been defined in the first half of the 90's with the simple goal of defining a standard for writing message-passing programs. This standard involved 60 peoples from numerous system manufacturers and academics. The standard takes the form of an API specification which provides a reliable communication interface, similar to available systems such as PVM, Express [41, 109], Parmacs [30]. The API provides point-to-point communication, collective operation, binding for C and Fortran, and easy definition of SPMD applications. List of nodes involved in the execution of an MPI application are assigned linear processor identifier which are used to identify sender and receiver in communication operations.

In order to support global communications, processors can be organized in groups of processors called communicators. Communicators are used to specify subsets of processors involved in global communication operations. A distinction is made between communication intra-communicator and extra-communicator.

The MPI standard as been widely accepted among the scientific community thanks to its large availability on high performance computers and a predicable performance. Like PVM, MPI relies on explicit communication between processors. The processors involved in the execution of an application are organized in communicators or group of processes. Each communication can be either point-to-point or collective with additional logic such as reduction operation.

2.2.3.2 Tools for (distributed) shared memory

Tools to program shared memory architectures mainly rely on the notion of threads. Threads usage can be made transparent to the user using dedicated language or extension to traditional language. The most well known is probably OpenMP.
OpenMP
OpenMP [138, 137, 44] takes the form of pragma directives introduced in the application source code. These pragmas allow the specification of parallel section, data partitioning and synchronization barriers. The execution of the program can be controled using environment variables. OpenMP has been left aside by hardware vendors of high performance computers due to the success of MPI (see 2.2.3.1). With the current limit reached by processor designers, the clock frequency growth, multiple core processors have appeared

and are now commonly available in most computers. These include desktop computers as well as high performance computers. Shared memory systems are now almost everywhere and standard compilers widely available added supports for OpenMP extensions to C and Fortran.

Tuple-space model of Linda

Tuple-space provides the user with an associative shared memory. This model and its most well known implementation Linda [51, 34] rests on the notion of tuple-space. The memory of the application is composed of a set of tuples. A tuple consists in the association of a memory area and a name. In order to manipulate a memory area, one needs to retrieve its tuple. The notion of tuple allows the definition of additional semantics on memory operation. Thanks to this additional level of semantic, it is possible to use a modified shared memory programming model more efficiently over distributed memory hardware.

2.2.3.3 Tools for remote execution

Environments based on remote service execution are based on the client server communication pattern. A server provides a set of routines which can be used from local or distant clients. Clients provide the input argument which are sent along with the service name to the server. Most of the time, the service request (the name of service associated with the input parameters) generates an answer which contains output parameters. Semantically the execution of a remote service is similar to a function call. The only difference consists in the code that must be generated at the location of the function call by the compiler.

Remote Procedure Calls RPC have been used widely in distributed systems. They have been used to implement file systems such as NFS or large scale distributed middleware such as grid (discussed in section 2.3.2.2. The development of RPC applications can be speed up a lot by the use of stub generators based on the definition of service interfaces using interface definition languages (IDL). The SUN RPC is a widely used implementation which provides tools to generate most of the code involved in the definition of the remote services as well as the encapsulation of the operations needed to execute remote services on the client side. The RPC approach is the foundation of all web oriented technologies. Several projects have extended the model proposed by Sun RPC with emphasis either on interoperability (DCE-RPC [144]), performance (DFN-RPC [143]), or fault tolerance with (RPC-V [33]).

In the context of high performance computing, asynchronous RPC have been widely used in master worker applications [152] and task farming environments. These applications and supporting infrastructures are detailed in section 2.3.2.2.

JAVA Remote Method Invocation The JAVA language defines a mechanism to access remote objects which follow the remote execution model. In JAVA it is possible to execute a function on an object executed by another process once it has been published in a registry. The RMI registry binds object names to distant objects. The method invocation on the client side is similar to the behavior of the standard RPC implementation. The use of threads allows the execution of asynchronous requests. The overhead of RMI is important as shown in [170]. Several projects provide alternate implementations of the Sun RMI [121, 18] with enhanced performance.

CORBA [153] is a middleware which has been designed by the Object Management

Group [183]. It provides access to remote object to applications. CORBA objects define a set of services described using an IDL. IDLs define the list of features provided by an object on the virtual network managed by CORBA. CORBA provides mechanism to discover remote objects, explore the set of methods provided by a remote object and invoke distant methods. Persistence, object creation, and many other services are available for application developers. Many technologies have been designed based on a subset of CORBA such as COM and DCOP.
Web services [196] is a specification defined by the W3C. It is an intent to standardize the communication between resources available on the Internet. Web services rely on XML and on the HTTP protocol. Remote invocation is managed by technologies such as XML-RPC[197] or SOAP which relies on an XML encryption of the request and the results of the service invocation. The web services specification is defined alongside a set of others. They define, for example, how to locate web services (discovery), learn the interface of a web service (introspection or reflection).

2.3 Toward large scale architectures

With the increasing number of processors and number of cores per processor, current high performance computers tend to grow toward large scale architectures based on hundreds of thousands cores. Clusters of clusters are used to solve applications which do not fit on a single cluster. It is also common to reuse old clusters which have been replaced with more powerful ones. The introduction of clusters of clusters also has an impact on the network topology. Network topology is more and more hierarchical with each cluster isolated in its own network and connected to other clusters through gateways.

In the mean time, meta computing systems based on heterogeneous computers connected using WAN become more and more mature. Internet based computing solutions incorporate millions of heterogeneous processing units in order to achieve numerous computation jobs. These architectures do not intent to replace existing high performance computers, they come in as a complementary solution. The development of Internet base computing solutions has been made possible thanks to three evolutions:

1. Networking infrastructure evolved toward high availability, performance and efficiency of world wide and local networks. In the mean time, the number of hosts connected to the Internet is in excess of five hundred millions [40].

2. Network communication protocol and data exchange formats have been standardized thanks to the use of XML base data format.

3. Finally, the high number of underused computing resources and data storage capabilities lead to the development of solutions to access and use these available resources.

Meta-computing environments used in e-Science projects focus on providing mechanisms to access heterogeneous resources remotely. A non exhaustive list of heterogeneous resources includes CPU cycles, persistent storage, databases, software and experimentation devices such as telescopes, imaging devices as well as visualization tools. The goal of a meta computing environment is to allow any user connected to the system to access and exploit any kind of resources available in the system.

Providing uniform access to many different kinds of devices requires the resolution of issues related to security, heterogeneity, fault tolerance and resources discovery. The security is probably one of the main matter of meta computing environment. The meta-computing environment creates a private community upon an open network of resources. Resources involved in a meta-computing environment are shared between local users, if it applies, and remote users. A taxonomy of meta-computing environment can rely on the following criteria:

- Trust in users and computing resources: How far can other users of the system trust a computing resources ?
- Layout of the management area: Is the system managed by a well identified set of servers ?

Based on those two criteria, the security mechanism that must be provided varies significantly.

Build upon security mechanism, several software layers are added to the meta-computing infrastructure in order to provide transparent access to remote resources and efficient exploitation of computing devices. The progress made in the area of Web services and grid services (OGSA) [84, 83] enabled the definition of a virtual organization or a grid as a collection of collaborating services. Each computing device involved in the virtual organization provides a set of services (either static or dynamic) to other computing devices. All mechanisms available to the end user of the infrastructure are made of a collection of services. A small list of the main services composing meta computing environments is presented below:

Authentification services are responsible for the identification of the user. Once signed in, the user can access services available on the network. The authentification service can also be used by computing devices to retrieve credentials associated to a particular user.

Information services are centralized or distributed services used to store and query information about computing devices which are connected or have been connected to the meta-computing environment. They provide information upon hardware, operating systems, softwares available and network bandwidth.

Allocation services manage user requests for resources. They transmit a request to a set of computing devices and reserve these devices for a dedicated amount of time like batch schedulers do on high performance computers and clusters. This service needs to cross information obtained by the authentification services and information services.

Communication services are used to create communication pipes between computing devices connected to a meta-computing environment. These pipes are used when two devices have to exchange information. It is similar to what is provided by

TCP/IP for the Internet. Several protocols might be available depending of the kind of communications. This layer builds a virtual network private to the meta-computing system. It is used for all communications between peers.

Parallel API services allow the use of programming environments on top of the meta-computing infrastructure. The meta-computing infrastructure defines a virtual architecture. Programming environments provide a runtime environment to end-users to exploit the virtual architecture. We will discuss in more details the programming tools available later in this section.

Three families of meta-computing infrastructure exist: grid, global computing and peer to peer environments. The next subsection provides a brief overview of these families and try to highlight the differences between those architectures. A detailed taxonomy of these systems can be found in [11].

2.3.1 Meta computing infrastructures

2.3.1.1 Grid computing environments

The grid systems reflect the goal of the I-Way project [80]. It aims at demonstrating the feasibility of connecting 17 institutions providing heterogeneous computing resources using 1Gbit/s dedicated network for high performance applications. Nowadays grid projects are based on this work. The definition of the grid as been given in the book *The grid Blueprint for a new computing infrastructure* [82] and refined in [77, 78]. An overview of grid computing environments can be found in [85].

In grid environments peers are provided by identified institutions which agreed to share their capabilities with other participants of the virtual organization (VO). A peer can provide access to resources such as CPU time, persistent storage, databases, applications, instruments, results computed by other applications, etc. The organization of a grid middleware is presented in figure 2.2. Grid middleware defines an overlay on top of a network which is most often a WAN. In order to guarantee the security of the grid, middleware often defined a virtual private network (VPN) which allows to bypass firewalls, and to isolate any communication of the grid from the public network by means of encryption.

In a grid environment, all resources are administered by one or several institutions. The level of trust which can be given to a resource is high. Indeed, resources are uniquely identified using certificates. Certificates assume the role of identity papers and are use to track the activity of a resource or of users. Any abuse can be tracked using the user certificate and thus, resources can be considered trusted. In a grid environment, grid management services, users and resources are trusted.

Many grid are already deployed. Those grid are supported by a middleware. The most representative examples of middleware are Legion [91, 92, 32], UNICORE [65, 43] and the Globus Toolkit [79, 81].

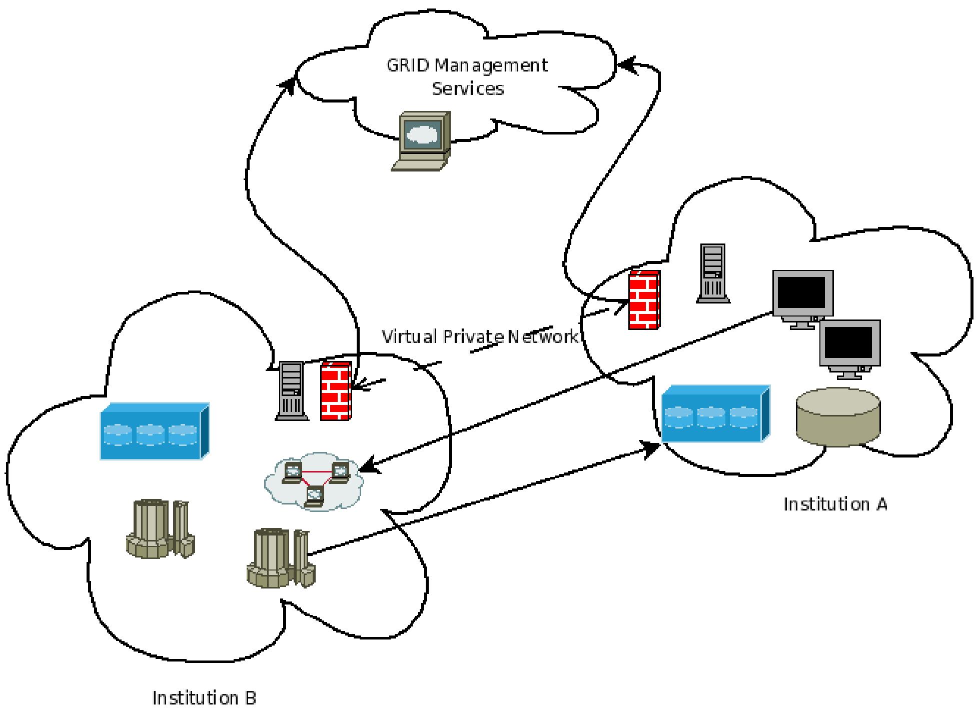


Figure 2.2: A grid infrastructure organization

2.3.1.2 Global computing environments

Web computing projects such as Jet [139], Charlotte [23], Javelin [38], Bayanihan [148], SuperWeb [6], ParaWeb [29] and PopCorn [133] appeared with the JAVA language and web applets. Volunteers connect their browser on the project page, an applet is loaded which contacts the web server to retrieve tasks to be executed on the remote hosts. This approach has multiple interesting properties. The host is protected by strong security policies guaranteed by the applet mechanism of JAVA. Applets cannot access files on the volunteer peer, they cannot connect to distant servers except the one which provides the applet and finally the host is unchanged once the navigator leaves the web page embedding the applet.

Those benefits are also the main constrains of the previous systems. They don't allow the use of available storage resources, and the storage of intermediate computation. But the main constrains of applet base solutions is the limiting policy regarding remote host connections. Web servers are the limiting bottleneck of the above systems. Based on the principle of these projects several meta-computing solutions have been proposed.

Global computing environment relies on a centralized set of servers which distribute independent tasks to remote peers. Peers are heterogeneous hardware, mostly desktop computers and workstations, provided by volunteers. Peers are used during their idle time and the system relies on cycle stealing policies. This kind of environment is an extension of Internet computing solutions which removes some of the limitations of system based on Java applets. Figure 2.3 presents the organization of a global computing system.

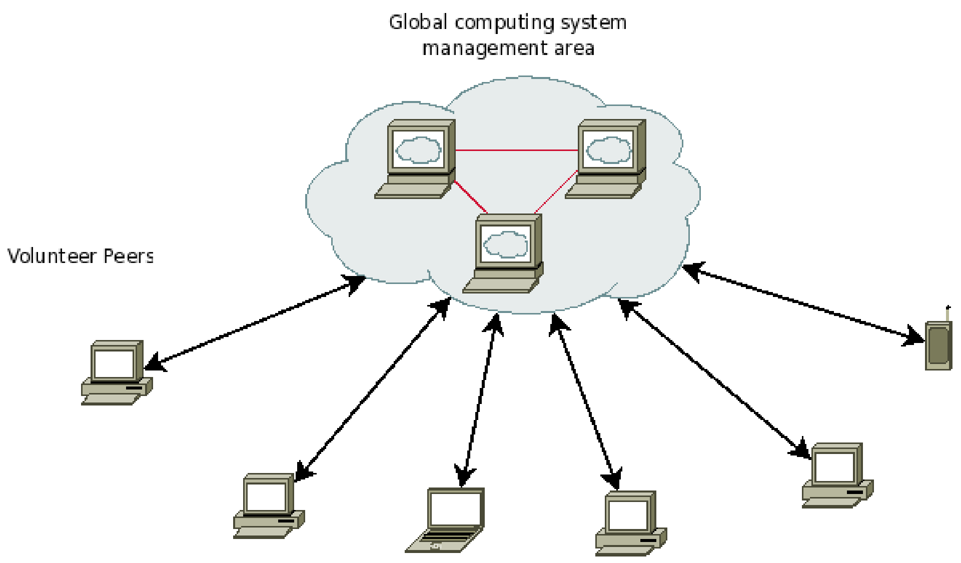


Figure 2.3: Global computing infrastructure organization

In Global computing environment, computing power is provided by volunteer peers. Only a set of resources are managed by an identified institution to provide the management of the global computing environment. Peers on the other hand cannot be considered safe. And thus can have malicious behavior. Currently used solutions to solve this kind of issue include: redundant execution, and code instrumentation. The latter consists in integrating into the results some hidden information which can be used to demonstrate that the results have been obtained using the registered application.

The Seti@Home project [12] is probably the most famous project of this category. XtremWeb [33] introduces support for multiple and user defined applications. It provides a set of servers which collaborate to dispatch work submitted by client and worker. XtremWeb is detailed in 5.3.1.1. BOINC [13] is the evolution of the Seti@Home project. Developed after XtremWeb, it supports several applications too. BOINC and Seti@Home emphasize on volunteers rewarding [14]. Many e-Science projects based on Monte-Carlo simulation now use BOINC.

2.3.1.3 Peer to peer environments

The last category of meta computing environment is peer to peer (P2P) systems. They are a generalization of global computing environment where each peer can play several roles. They can be at the same time client and server. They can also be involved in the management of the middleware itself. True peer to peer systems differ from meta-computing environments presented previously in regards of the management of the platform. In P2P systems, the management of the architecture is contributed either by all peers or by elected peers during the execution of the system. Each peer is at the same time producer, consumer and administrator of resources. In these systems, not only the computation peer (worker and client) can be malicious but also the management one. Representative middleware for peer to peer environments are JXTA [189, 89], Peer-to-Peer Simplified

[176] and DGET[99, 59, 17].

2.3.2 Programming tools for meta computing systems

Meta-computing environments are typical distributed memory systems. They rely solely on distributed computing architecture. The tools used to support this kind of systems have to handle new constraints such as heterogeneity and fault-tolerance either explicitly or implicitly. These systems are most of the time programmed using message passing libraries, or remote invocations. A more detailed presentation of the programming models and tools available in grids can be found in [114]. The remainder of this subsection describes several approaches used to program applications on large scale distributed systems.

2.3.2.1 Message passing

The MPI standard has been adapted to meta-computing environments. A globus version of MPI called MPICH-G2 [104] is available. This version solves the problem of heterogeneity and allows to mix versions of MPI in a single application. However, it does not provide any mechanism to manage faults. Fault-tolerant versions of MPI have been developed for meta-computing environments and large scale distributed systems in [146, 66]. Many specialized implementations of MPI solve some of the problems associated with WAN and heterogeneous environments. The MagPIe library [107] implements MPI's collective operations for wide area systems. PACX-MPI [87] enables collective operations using TCP and SSL. Stampi [102] has support for MPI-IO and MPI-2 dynamic process management. Lastly MPI_Connect [67] enables different MPI applications, eventually under different vendor MPI implementations, to communicate. MPICH-CM [151] is an adaptation MPICH which targets peer-to-peer systems.

OpenMPI [86] is a modern realization of MPI which sums the effort of various MPI implementations to provide a new environment for grids and large scale architectures. OpenMPI provides mechanisms to notify the user application that a fault happened on a particular resource. The resource is then blacklisted for the rest of the execution. The application is responsible for adapting its behavior to continue its execution.

2.3.2.2 Remote execution

RPC mechanisms are simpler than MPI. They are also more adapted to the service oriented architecture of nowadays middleware. Almost all meta computing environments provide their own mechanisms for remote invocation. Fault-tolerant versions of RPC library exist [52]. They hide the management of faults to applications. Remote invocation mechanisms are most often used following the master/worker model to do task-farming or to execute bag of tasks[36, 35].

Some tools provide more advance solutions built upon the remote invocation mechanism. They are known as problem solving environments[88, 98] (PSE). A PSE is a computer system that provides all the computational facilities needed to solve a target class of problems. These features include advanced solution methods, automatic and semi-automatic selection of solution methods, and ways to easily incorporate novel solution methods. Moreover, PSEs use the language of the target class of problems, so users can run them without specialized knowledge of the underlying computer hardware or software. By exploiting modern technologies such as interactive color graphics, powerful processors, and networks of specialized services, PSEs can track extended problem solving tasks and allow users to review them easily. Overall, they create a framework that is all things to all people: they solve simple or complex problems, support rapid prototyping or detailed analysis, and can be used in introductory education or at the frontiers of science.

NetSolve[16, 4, 1] is a client-server application that enables users to solve complex scientific problems remotely by providing access to hardware and software computational resources distributed across a network. NetSolve searches for resources on a network, chooses the best one available, and using retry for fault-tolerance solves a problem, and returns the answers to the user. Goals of the NetSolve project include ease-of-use, efficient use of resources, and the ability to integrate arbitrary software components as resources. Interfaces to Fortran, C, Matlab, and Java enable users to access and use NetSolve more easily. NetSolve can be used in a standalone environment or on top of Globus.

Other problem solving environments for grid include Ninf[150, 126] and DIET[10] which is a CORBA based realization of NetSolve. The latter evolved in a richer environment which incorporates for example a workflow enactment service.

2.3.2.3 Distributed Filtering

In data intensive application, which handle really huge amount of data, it is important to be able to extract critical information from the mass of available information. This extraction can be perform using a network of distributed filters [103]. Filters are programs that parse a data stream, analyze it and transform it into a more consistent form. Several filters can be chained to form a complex filtering operation. The filter model relies on the definition of a set of filter and the connection of this filters one after the other. Several filter can be executed in parallel to work on different data stream at the same time.

Data Cutter [26, 132, 111, 27] is a framework to manage the execution of filter based applications. This framework allows the definition of a topology of filtering operations connected by unidirectional streams. The topology definition is transmitted to a runtime manager which maps the filters and manage the communication between the many filter instances. Data Cutter is oriented toward really huge data filtering and knowledge extraction. The framework provides advanced indexing services. These services are responsible of extracting data from the storage system and send them into the filter network.

2.4 Scientific workflow

2.4.0.4 Workflow orchestration

A workflow is the description of a business process. It consists in a formal representation of the steps involved in the achievement of a task. A workflow is composed of a set of activities. An activity represents a step in the realization of the task. Activities are assigned to agents which can be either human or device. The workflow describes the matching between agents and activities. It also defines how activities are related to each other.

The concept of workflow has been adapted to computer systems in two steps. First the goal was to help managers to assign activities and control the evolution of the process leading to the completion of a task. The second step was to use workflow as a programming model to orchestrate the execution of a distributed application relying on agent or server available on the Internet. We are interested in the second category of workflow tools.

The web is now composed of an increasing number of services which can be combined in order to achieve a complex treatment. Typical examples of web services are Internet portals such as iGoogle which incorporates themes that adapt themselves based on weather forecasting or time services. This has been made possible thanks to the development of standards such as Web Services, XML for data representation and HTTP for communications. XML-RPC or Soap are used as a standard query mechanism for remote services. They follow the RPC approach and the client/server model.

We have seen before that grid middleware introduced in the previous sections rely heavily on the notion of services. Each role or mechanism in the system relies on a combination of services to achieve any single operation. If this orchestration is manageable for a simple operation it is difficult to orchestrate manually the execution of a complex scientific application requiring hundreds of service invocations. The complexity and the decomposition of roles which are good as a software development practice is a nightmare for application designers.

The adoption of workflow systems to manage the execution of coordinated services simplify the definition of complex applications based on services connected on the web. The workflow engine manages all operations required to execute the application :

- Selection of agents: the workflow enactment service is responsible of defining a mapping between agents and activities.
- Migration of the data between activities: the enactment service is responsible for migrating data during the execution of the workflow.
- Interaction with data sources: the enactment service is in charge of connecting data sources such as SQL database to activities.
- Retrieval of results: the enactment service must gather results and store them so that the user can retrieve them once the workflow execution is terminated.

2.4.1 Workflow for grid middleware

This subsection is strongly inspired by the taxonomy of workflow management systems for grid presented in [178].

2.4.1.1 Condor DAGMan

Condor [116, 155, 160] is a specialized resource management system (RMS). It relies on a heterogeneous batch system which dispatches jobs to computing resources. Condor-G allows to use Condor on top of the Globus toolkit. Condor proposes the Directed Acyclic Graph Manager (DAGMan) [184, 155]. It is a meta scheduler for Condor jobs. DAGMan allows the expression of dependencies. Each job is a node of the graph and the edges identify their dependencies. Each job can have any number of "parent" or "children" nodes. Childrens are only executed once their parents have completed their execution. DAGMan does not allow cycles in the graph to prevent dead-locks. Data migration are not automated by DAGMan and must be explicitly defined by the user. The mapping of the workflow to the computing resource is done during the execution of the workflow by the condor job scheduler.

2.4.1.2 Pegasus in GridPhyN

GridPhyN [193] is a data grid dedicated to physics experiments. It is based on the Globus toolkit. It defines Pegasus [45, 47, 46], a workflow manager. Pegasus performs a mapping from an abstract workflow to a set of available resources, and generates an executable workflow. Workflow are expressed using DAX (DAG XML description). Pegasus manages the execution of the workflow by means of datasets monitoring. The results of each activity of a workflow is a dataset. Pegasus progresses in the execution of the workflow each time a dataset is available. In order to map the abstract workflow to the grid resources, Pegasus makes use of several grid information services such as the *replica location service*, the *transformation catalog* and the *monitoring and discovery service*. The mapping of resources is assisted by articial intelligence techniques which generate a static planning for the execution of the workflow. Finally, the workflow is transformed into a DAGMan job.

2.4.1.3 Triana

Triana [157, 156, 158] is a visual workflow-oriented data analysis environment. It relies on GridLab Grid Application Toolkit interface (GAT) [7] which provides access to grid services through JXTA[159], web services[122, 123] and OGSA. Triana workflows are composed of units. Units can be assembled using drag-and-drop in a visual editor. Units are connected using logic units. Triana contains an integrated unit editor which allows at the same time the definition of computation unit and logic unit. It is also possible to group units together as a single unit for defining subworkflows. The mapping of the workflow to the resources is done at the startup of the workflow on the resources available. However, if a resource disappears, Triana requests a new resource from GAT and continues the execution.

2.4.1.4 ICENI

The Imperial College e-Science Network Infrastructure (ICENI) [128, 127] defines a component-based Grid middleware. ICENI allows the definition of abstract workflows which consist in a collection of components. Each ICENI component is described in terms of meaning, control flow and implementation. Components are supposed concurrent and are linked to each other through constraints mechanisms. This definition of the workflow is converted to a temporal view which consists in a graph where nodes represent computation and edge time dependencies. This model is similar to the one of Yet Another Workflow Language (YAWL) [164, 50, 165]. ICENI provides several scheduling strategies to map the abstract workflow onto grid resources. The scheduling strategy can make use of information gathered during the previous execution of components gathered in a performance service [127].

2.4.1.5 Taverna in ^{my}*Grid*

Taverna [136] is a workflow management system used to support experiments in biology. Taverna provides data models, enactor task extensions and graphical interfaces to control FreeFluo [186] the enactment engine. Data models consist in inputs, outputs processors, data flow and control flow. Through graphical interfaces, the user can control the mapping of the workflow to nodes of available resources or group of resources. Fault tolerance mechanisms allow the user to set some level of tolerance on the data model composing the workflow. The data model can be tagged with some level of requirement. For example, a critical data model that fails past several tries will abort the complete workflow while some other policy would allow to abort only a part of the workflow.

2.4.1.6 GridAnt

GridAnt [172] is a workflow system based on Ant[180]. It focuses on distributed process management rather than on the aggregation of services which is the concern of most other grid-enabled workflow frameworks. Ant provides a dependency management mechanism similar to Make and is thus limited to the execution of DAG. The language used to describe the workflow is based on XML and is similar to the description of a software built. Each operation is associated to a tag. In order to be executed, an operation needs to wait for the notification of all the tags on which it depends. Tags are implicit in GridAnt. GridAnt is part of the CoG toolkit[173] and is thus dedicated to Globus based grid middleware.

2.4.1.7 GridFlow

GridFlow [31, 93] is a grid workflow management system which is based on agent-based resource management. Rather than focusing on workflow specification and the communication protocol, GridFlow is more concerned about service level scheduling and workflow management. There are three layers of grid resource management within GridFlow: the grid resource, the local grid and the global grid. A grid resource is simply just a particular peer; local grid consists of multiple grid resources that belong to one organization; and a global grid consists of all local grids. Global grid also provides a portal for composing workflows.

Workflows are represented as a flow of several different activities, each activity representing a sub-workflow. Each sub-workflow is itself composed of flows of tighly coupled tasks to be executed on a local grid. The workflow is managed by a hierarchical scheduling system. In order to map a workflow to a collection of resources, GridFlow first simulates the execution of the workflow in order to generate a near optimal mapping. The latter is validated by the user before the real execution. The mapping strategy takes into account data-gathering during previous workflow executions.

2.4.1.8 Askalon

Askalon [68, 177, 55] defines two separate composition systems: AGWL (abstract Grid Workflow Language) [70, 71] an XML based language, and Teuta [141, 69] that supports the development of workflows using UML activity diagrams. AGWL provides a rich set of constructs to express sequence, parallelism, choice and iteration workflow stucture. In addition, programmers can specify high-level constraints and properties defined over functional and non-functional parameters for tasks and their dependencies which are used to optimize workflow at runtime.

The mapping of resources on the grid is based on a new hybrid approach which mixes static planification and dynamic adaptation of the planification depending on the status of the grid resources. The static planification depends on genetic algorithms. Askalon maps resources to grid resources using a *grid resource management service* based upon the Globus toolkit version 4. Like in GridFlow and ICENI, the planification of the workflow takes into account the data gathered during previous workflow executions.

2.4.1.9 Karajan

Karajan [171, 174] is derived from GridAnt and is part of CogKit. It extends the previous solution with workflow structures, scalability and error handling. It is easy to integrate in several Grid middleware. It has been integrated with Globus, Condor, SSH, and some data transfer techniques such as WebDAV and scp[2]. Karajan supports construction such as sequence, choices, loops of workflow structures. It relies on an XML workflow description language. A number of fault handling methods exists. Error handling allows users to integrate strategies for errors and exceptions into the workflow. Checkpointing enables users to store intermediate states of the workflow execution for later roll-backs when problems occur.

2.4.1.10 Kepler

Kepler[119, 9] is one of the most popular workflow systems. It is derived from the Ptolemy II system [117]. Kepler relies on an actor oriented feature. Independent actors communicate through well defined interfaces. An actor is an encapsulation of parameterized operations performed on input to produce output data. The execution model can be defined by users and is responsible for defining the order of execution of actors. This allows to change the workflow logic easily. Actors can be defined and deployed locally or can be discovered through a web services harvester. Kepler also defined a set of actors for interacting with grid middleware. It provides a visual editor to connect actors together. Kepler enables at the same time control flow and data flow construction.

[2]The secured shell file copy program provides a command similar to unix copy which allows to transfer encrypted files between two systems

2.5 Conclusion

We presented briefly the evolution of high performance computers and their programming. We highlighted that hardware architectures tend toward distributed memory architecture. Processors used in these systems are for most of them super scalar processors. Each processor is composed of more and more cores which can be either homogeneous or heterogeneous. The number of CPUs keeps increasing and is now of the order of a few thousands on the fastest high performance computers of the TOP 500. Those CPUs are connected to each other through a hierarchical interconnection network. Recently, dedicated processing units also called accelerators have been added to high performance systems to enable multiple programming paradigm and increase the efficiency of those systems. Examples of such processors are IBM CELLs and Graphic Processing Units (GPU). Accelerators are interesting because of the gain of efficiency by means of FLOPs per Watt. Indeed the power consumption of high performance systems is becoming a problem for nowadays architectures.

The motivations behind the important amount of existing large scale distributed systems are different. However, all achieve a common goal. They provide mechanisms to access distant resources in order to support computations. Meta-computing environments and especially grids are more and more present. The architecture of those systems is based on services which interact using mechanims based on remote service invocations. Despite a gain in maturity and a convergence in the organization of those systems, the APIs available to develop client applications are still lacking of standards. However, the lack of standards has a strong impact on the portability of applications built upon those systems.

We highlighted that high performance systems and meta-computing environments are starting to face the same kind of issues. Both systems tend toward heterogeneous processors. They will soon have millions of concurrent threads of executions. For this kind of systems, programming tools and environment need to be adapted.

In order to help the design of applications for this kind of systems, we modeled and implemented a scientific workflow named YML which allows interoperability of applications between middleware. Interoperability is achieved at two levels. Applications can be executed on several middleware without any modification and application execution can be spread across several middleware at the same time in order to enhance the coverage of available resources. The next four chapters are dedicated to the presentation of YML.

Chapter 3

A workflow for large scale distributed system: YML

3.1 Introduction

YML is a scientific workflow for large scale distributed systems. The aim of this chapter and of the following three is to describe this workflow environment. We modeled it using a component oriented approach. This in order to achieve simplicity, interoperability, modularity and integration.

At the beginning of YML, the intent was to provide a language to express more easily parallel applications on a global computing environment. This to enlarge the scope of applications which could be created for this kind of systems. The lack of standard API for programming meta-computing environments leads us to a need of separation between the representation of the application and the runtime environment. Indeed the evolution of those systems is fast but still in an early stage. Only recently, solutions like DRMAA appeared.

3.2 Workflow overview

The notion of workflow has been used for years in business management. A workflow is the description of a business process composed of activities which can be held either sequentially or concurrently. In a computing workflow, an activity can involve either a computing device or a human interaction (for example the validation of an intermediate result). In either case, an activity exposes an interface which always incorporates information about its inputs and its outputs, and sometimes parameters. An activity defines a collection of channels. Each channel represents a communication end-point used to exchange data between activities. The workflow process defines the order of activities. It also defines how activities exchange data by connecting their channels to each others.

The growing complexity of corporate promotes the use of workflow. Indeed, in order to decrease the costs, subcontracting is more and more used. However, for large companies which depend on many subcontractors, the coordination of projects is a difficult task. It involves the exchange of many documents between the various entities involved, validation of steps, step ordering, etc. This is a typical application of workflows. More and more companies start to use such tools to drive large projects and formalize intern and extern communication. This kind of usage of workflows is the first step toward the automatization of business process assisted by computers. This approach leads to the development of many workflow environments such as JBoss.

More recently, workflows have been used in the e-Science project to drive large experiments. Workflows were first introduced by the biology research community to drive experiments on grid middleware. The aim of those workflow environments is to allow non computer scientists to describe the process of converting row data to exploitable results. Since the introduction in the grid community of workflow environments, the amount of research held on that kind of technology is really important. The approach proposed by most workflows consists in allowing the coupling of existing software to constitute an application.

3.2.1 Specificity of YML

YML is not a tool designed solely for code coupling. It is a workflow dedicated to the creation of complete applications from the very beginning to the end. This includes the creation of the workflow as well as the integration of services. Our approach to workflow is slightly distinct from the traditional behavior. Indeed, most of them emphasize on the harvesting of services. In YML, we do not have any harvesting mechanism. We manage services by means of two catalogs which store services available. YML services are designed to work solely within YML workflow processes. Those services are created in the context of YML applications.

YML expresses an application using a workflow process definition language. This language allows the definition of any directed graph while many workflow environments rely on DAG. Unlike most existing workflows, the workflow process definition language is used to express the control flow of a YML application. We deduce the data flow. This approach is similar to regular programming language. Many workflows for grid environments do the opposite: they express the data flow, from which they deduce the control flow. Some workflow environments like Kepler, provide both explicit data flow and control flow. This behavior is adapted to grid middleware where resources are quite stable. On peer to peer and global computing systems however, we most often are not able to map the data flow onto a collection of peers. A workflow based solely on control flow enables the expression of asynchronism. Indeed, YML can express optional dependencies. This can be used to loosely couple several sub-workflows in order to make them collaborate asynchronously.

This workflow description and the graph it describes are not used directly. YML relies on an expert system to drive the execution of the workflow. An expert system, or a rule engine, is a tool which is used to demonstrate a fact. A fact can be a boolean attribute or a complex object. What is important for the expert system is the presence of the fact. The expert system behaves like a blackboard of facts which can be extended by applying transformation rules to it. A transformation rule is similar to the *if/then* construction of most languages. The if condition is a boolean expression based on the presence of facts. If the condition is true, the rule is executed which leads to two things: 1) the execution of side effects 2) the extension of the blackboard with new facts. YML converts all workflow processes into a collection of rules and facts.

YML provides an environment used in the conception of algorithms and their implementation for large scale distributed systems generally speaking. This is achieved by the notion of back-end. The latter assumes the same role as the operating system of a computer. It is the bridge between user applications and the hardware. YML back-ends are used to provide a normalized interface to middleware, the same way as standards like POSIX provide a common API on UNIX systems. The support for multiple middleware is - unlike most other workflow environments - not limited to a few systems and especially not limited to GRID environments. Indeed, most existing workflow environments support one or two middleware. The support for new middleware is not something exposed to end-users of the system and the dependencies in term of services are often important. YML, on the other hand, is designed to support multiple middleware from the early beginning. YML exposes this aspect to end-users. The aim is to allow other people to easily integrate YML within their middleware. In the mean time, this feature also allows reuse of existing back-ends to enable the execution of workflows, to make use of resources from multiple middleware at the same time.

3.3 Workflow model

A simplified model has been introduced in 1.2.1. It describes the modeling of the architecture as well as of the workflow environment. The architecture model highlights the notion of peers. Peers represent computing devices which agreed to be involved in a middleware. In order to provide a model which can be easily implemented by any middleware, our architecture model introduces as few constraints as possible. We based our model on typical global computing environments. Among the large scale distributed architecture, those are the ones which are offering the least capabilities to the end user. They are thus well adapted to define the minimum requierments for a model of architecture.

Our architecture model identifies a subset of the peers called management peers which are responsible for the management of the architecture. Management peers provide services for other regular peers. Each peer must be able to communicate with any management peer. However, two regular peers do not have to be able to communicate with each other. Management peers are supposed to be available at least during the execution of a workflow process.

Figure 3.1 presents the complete model of workflow we defined to answer our motivations. It highlights all the components of the workflow. The figure shows the layered approach used to compose the workflow environment.

A middleware is a software layer which is used to manage a collection of peers which agreed to collaborate in a large scale distributed system. In order to model the complete environment we need to expose a few services which are provided by the middleware. All middleware expose a feature which allows the submission of a task or a job to be executed by a peer. This is achieved by a service called the *resource scheduler*. The resource scheduler is responsible for matching job request to peer the most efficiently. The complexity of this service varies significantly from one middleware to another. The

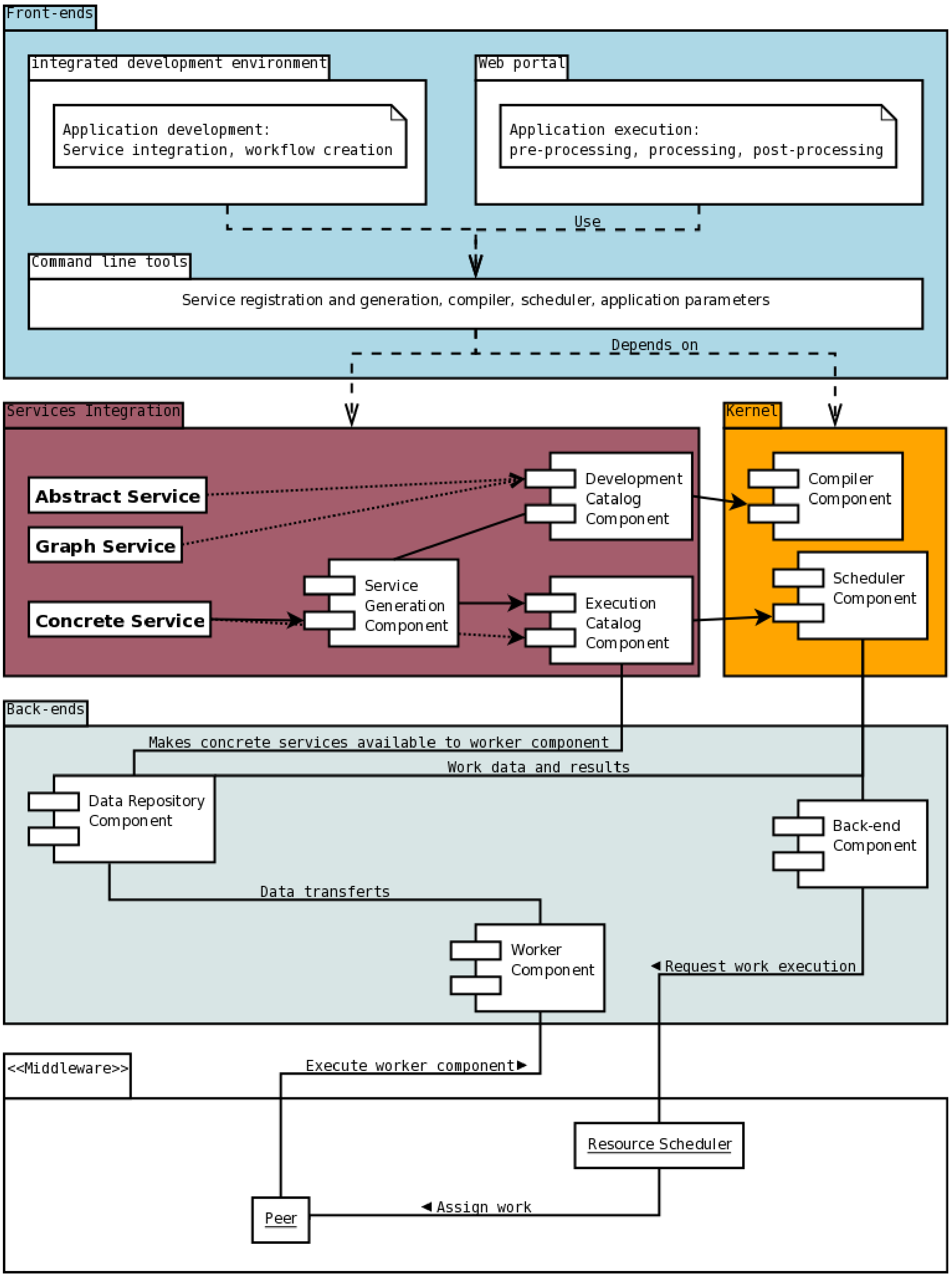

Front-ends
integrated development environment
Application development:
Service integration, workflow creation
Web portal
Application execution:
pre-processing, processing, post-processing
Use
Command line tools
Service registration and generation, compiler, scheduler, application parameters
Depends on
Services Integration
Abstract Service
Graph Service
Concrete Service
Development Catalog Component
Service Generation Component
Execution Catalog Component
Kernel
Compiler Component
Scheduler Component
Back-ends
Makes concrete services available to worker component
Work data and results
Data Repository Component
Back-end Component
Data transferts
Worker Component
◄ Request work execution
<<Middleware>>
Execute worker component ►
Resource Scheduler
Peer
◄ Assign work

Figure 3.1: Scientific workflow modeling

middleware coupled with the architecture constitutes a runtime environment.

If we compare middleware to a high performance computer, the middleware is similar to the operating system. Its goal is simply to provide an API to access hardware. Middleware offer a little more services than just the operating system. Most of them are also not interoperable. In order to achieve interoperability between the workflow environment and several middleware, we propose a component oriented modeling of a workflow system. Our workflow is modeled as a collection of interacting components. Components are interacting with each other through a well defined public interface.

The interaction with middleware is isolated in the back-end layer. This layer provides a uniform runtime environment to the kernel layer of YML. The layer is responsible for the mapping between abstract workflow (middleware independent) and concrete workflow, which is a mapping of the abstract workflow to available peers. The back-end layer is composed of three components: the back-end itself, the worker and the data repository.

The back-end component is responsible for the interaction with the resource scheduler service of the middleware. It translates the abstract work description composing a workflow into requests to the resource scheduler of a middleware. Each middleware has its own back-end component. The back-end component acts as a client of the middleware. The abstract work request which is processed by the back-end component leads to the execution of a worker component on a peer selected by the middleware resource scheduler. The worker component analyzes the work description and is responsible for its execution. It means the acquisition of the concrete service used to do the computation and the data required for the computation. Then, the worker executes the concrete service corresponding to the work. And finally, the worker publishes the result of the execution.

Each time a data transfer is required, the worker as well as the kernel layer interact with the data repository component. This component is responsible for the acquisition as well as the publishing of data in the workflow environment. The model thus defines two operations:

- *put*: this operation is used to publish data within a data repository or a network of repositories. Publishing data is similar to a write operation on a memory area.
- *get*: this operation is used to acquire data stored in a data repository. Acquiring data is similar to a read operation on a memory area.

Data is identified by a name and corresponds to a collection of arbitrary bytes. It is similar to the tuple space of linda. Indeed the data repository component can be used to simulate associative shared memory. One data is managed by a single data repository at a time.

Upon the back-end layer, the kernel layer is responsible for the management of workflow processes. A workflow process is composed of a collection of services which interact with each other by exchanging data. The services are created and managed in the service integration layer. Our workflow model depends on two catalogs to store services which can be used in a workflow. In our model services are regular components. We decided to name them services to highlight the difference between a component that is part of a workflow and a component defined in the workflow model. Our model defined three families of services:

- *Abstract services* define the public interface of the service. They are used to validate the workflow process. They describe how a service interacts with other services. Abstract services are defined independently of the underlying middleware. They are used solely during the development of an application and a workflow environment.
- *Graph services* realize an abstract service. They are used to define sub-workflows which can be used as regular services in the definition of the workflow. They provide a component which can be used during the execution of the workflow. They contain a workflow defined using the workflow process definition language provided. Those services do not depend on the middleware used.
- *Concrete services* are another mechanism to realize an abstract service. They are used during the execution of a workflow process and are effectively used during the execution of the workflow on a middleware. They correspond to services acquired and executed by the back-end layer worker component. They make use of a native language such as C/C++ to describe the computation effectively executed. A concrete workflow only contains execution of concrete services.

The interaction between the service integration layer and the other layers of YML is managed by two catalogs. The development catalog stores all information which is not specific to middleware. This information is mostly used during the development of workflow based applications. This catalog is used by the service generation component as well as the compiler component of the kernel layer. Concrete services on the other hand are stored within the execution catalog. This catalog contains all of the information needed for the execution of a workflow process. It includes a list of all concrete services already registered as well as a a mapping with abstract services. The selection of the concrete services to use is left until the time of the execution of the service. This in order to select the best concrete services at the time of the execution. This selection might contain criteria depending of the data size, complexity of the algorithm used, etc. The execution catalog is mainly used to store information used during the execution of a workflow process by the scheduler.

Concrete services are generated based on two information: the abstract services implemented and the computation in a traditional language. A component generator is used to mix the two information in order to create a usable concrete service. The abstract service behaves in a way similar to an IDL for the service generator component. Indeed, the abstract service is analysed to create a concrete service skeleton filled with the computation described in a traditional language such as C/C++. The component generator is then able to generate one or several binary applications used on peers. The component generator is also responsible for the registration of concrete services into the execution catalog.

The kernel layer contains two main components which aggregate catalogs, back-end component and data repository component. They are a workflow process compiler and scheduler. A workflow is described through a workflow process definition language. This language defines a process which can be represented by a generalized directed graph. This graph is processed by a workflow compiler and translated into a set of rules. Those rules are used by an expert system to manage the execution of a workflow process. The compiler generates those rules based on the information available in the development catalog. The result of the compilation of a workflow is a workflow representation which is independent of any middleware.

The workflow scheduler component is responsible for executing workflow processes. The scheduler does two main operations. It first schedules the execution of the workflow by solving the dependencies among its activities. The state of the application is represented by events. An event has only two states: It either happens or it does not happen yet. Events are injected into the expert system as facts. Rules represent computation on the middleware. Once a fact or a boolean combinaison of fact is validated, the corresponding rule is applied to the systems. This leads to the submission of a work to the back-end in the first place. It also leads to the introduction of new events in the system. The processus loops until the stop event is generated or all activities of the workflow have been processed.

Finally the front-end layer is responsible for the interaction between end-users and the workflow environment. It exposes several tools to interact with the workflow environment either to create new workflows or to execute and monitor them.

3.4 Outline of the description of YML

The model of workflow presented above is realized by YML. The description of YML follows the organization in layers of the model. We first describe the interactions between YML and its environment. We identify two kinds of interactions. We discuss the interaction between end-users and YML in the chapter 4. It focuses on the description of the three front-ends available. In chapter 5, we describe the interactions between YML and middleware. Back-ends assume an essential role. They are responsible for the interactions with the middleware and they do the mapping between a workflow process description and middleware peers. Finally, chapter 6 presents the integration of external services such as a numerical library within YML, the presentation of the workflow process definition language and the management of workflow processes.

Chapter 4

Front-ends: interactions with users

4.1 Introduction

In the previous chapter, we presented a model of workflow. The realization of YML is mapped directly to that model. In this chapter, we present the front-ends layer. It defines the mechanisms to interact with YML, available to end-users. The latters can interact with YML using one of the three front-ends available.

The integrated development environment is dedicated to the definition of workflow processes and services. This tool provides a convenient environment based on wizards to create and edit the XML documents composing a YML application.

The web portal is used to control and administer the execution of YML applications. Once a workflow application has been registered into the portal, users can schedule and execute applications from this web based environment. Emphasis is given on the monitoring and the result analysis of workflow applications. The portal can be extended to support application specific visualization.

The last front-end provided is composed of command line tools built upon the components defining the kernel layers. Those tools are not interactive and are targeted to expert users of YML. They are used by both the IDE and the web portal underneath it.

4.2 Integrated development environment

A YML application consists in a set of definitions. A definition corresponds either to a service or to a workflow process. A detailed presentation of the YML workflow process definition language and of services creation is given in chapter 6. Each definition is represented as an XML document. Most XML documents of a YML application incorporate sections of text which correspond to programs in another language. For example, concrete service definitions embedded C++ fragments. Graph service definitions and workflow process contain sections written using the workflow process definition language.

The IDE is focused on the development of applications. It targets developers of workflow processes as well as creators of services. It proposes a GUI dedicated to the edition of documents composing an application. The GUI allows the edition of multiple documents organized in projects. The IDE is built upon the model-view-controller design pattern.

It allows the edition or visualization of documents using one of the available views. The same document can be viewed and edited from any views. In order to simplify the creation of new documents composing a YML application, the user can be assisted by a wizard.

While the IDE is focused on the creation of applications, the second front-end is dedicated to applications' submission and monitoring.

4.3 YML web portal

The study of linear algebra methods such as MERAM involves many parameters. It also requires a lot of experimentation because of the lack of applicable methods to find the suitable parameter to use for a particular problem. It is thus interesting to have a tool which allows to easily test parameters and do the analysis of the results automatically. YML assumes the execution of the application. However the analysis itself and the production of plots are not incorporated within the workflow because those depend on the analysis of the logs. In the mean time, we were interested in a solution which requires as little knowledge of the runtime environment as possible on the one hand and in the mean time which eases the collaborative works. The YML web portal proposes a solution to these needs. The application domain which motivates this portal is linear algebra methods. However the solution proposed is not limited to them.

Generic portal solutions such as XCAT[110], Discover[124] or GridPort[161] target only GRID middleware based on Globus Toolkit. Such environments suffer from the same weakness as the Globus Toolkit: its complexity. However, those solutions are interesting because they have become services of the GRID itself. This kind of portal cannot be used in the context of YML. Indeed, YML hides the underlying middleware to the user. A web portal for YML, in order to be usable on top of any supported middleware, has to rely solely on the components exposed by the integration service and kernel layer.

YML provides a second front-end which covers all actions related to the execution of a YML application. It proposes a light client graphical interface for handling the preprocessing, the processing and the post-processing of YML applications. The web portal guides the end-user during the planification of an execution using wizards or step-by-step assistants. It monitors the execution of the application in real-time and proposes mechanisms to extend the monitoring by enabling visualization.

The portal consists in a list of registered applications. Once registered, the user can manage the execution of application, create a new execution or monitor current and past executions. The portal defines a set of hooks which can be used by an application to customize the behavior of the portal. The registration process allows to associate an application with several hooks. Hooks are used to allow an application to modify the behavior of the portal to best suit the application caracteristics.

The portal provides facilities to support all the steps of the exploitation of applications: the pre-processing, the processing and the post-processing. The pre-processing consists in applying a first pass on the data of the application to prepare it for the execution. It can for example consist in a sparse matrix conversion from one storage format to another, the fragmentation of a matrix in fragments or more generally, the validation of parameters. All applications registered provide at least one pre-processing form. The latter allows to create a new execution of the active application. The form is automatically generated based on the list of parameters defined by the workflow applications. This is the basic form of the pre-processing stage always available. Hooks are available to register the execution of third party applications before the start of the execution of the workflow.

The processing of the workflow itself is handled by the YML scheduler component of the kernel layer. It is not an interactive process. Thus the processing stage consists mainly in providing information on the progress of the execution in real time. The processing and the post-processing are tightly related to each other in the portal. Indeed, the monitoring and the visualization are based on the same construction: the notion of view.

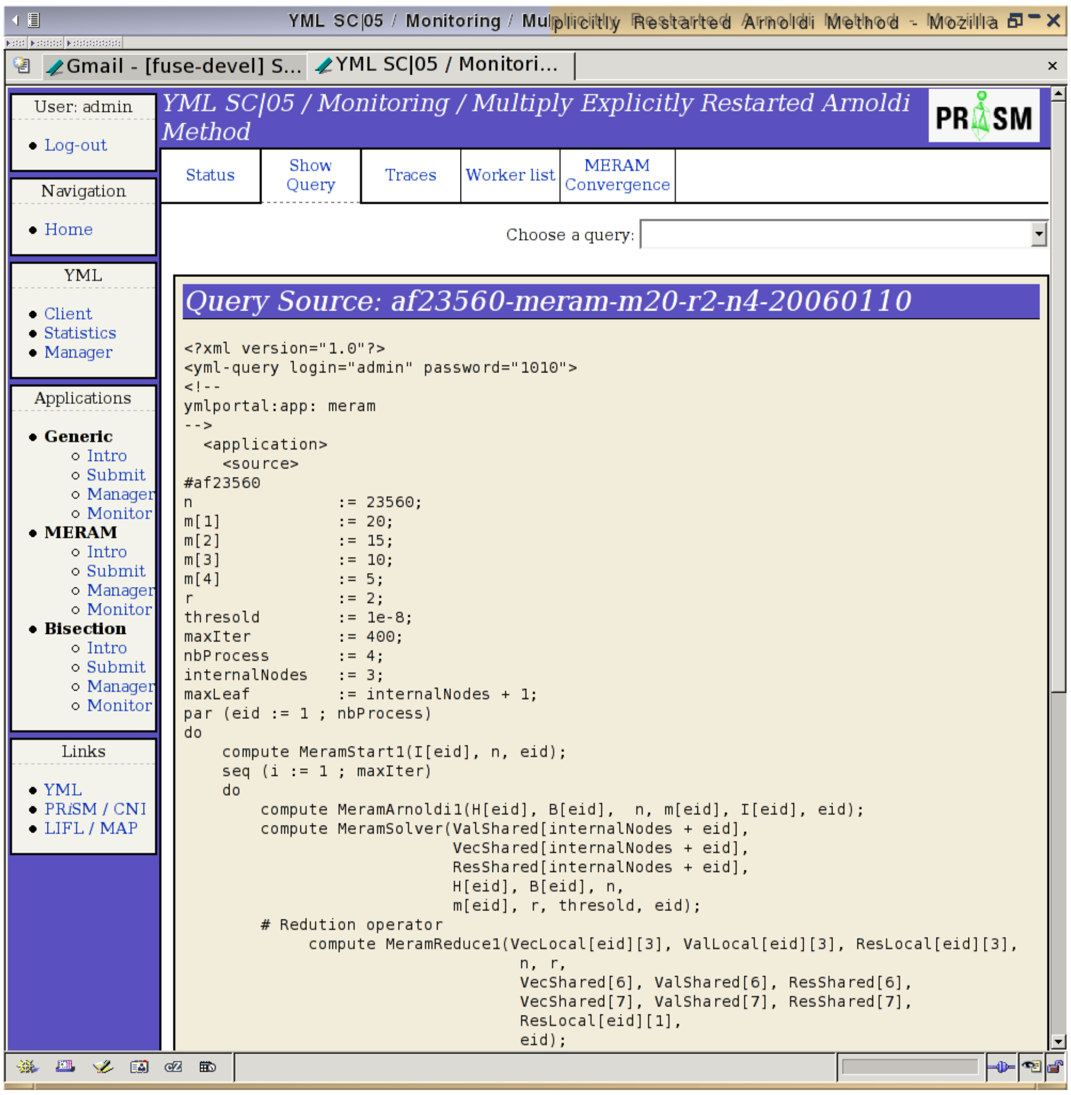
4.3.1 Monitoring and visualization views

The most interesting aspect of the application portal consists in the monitoring features. The execution of an application can be monitored in real-time and organized in views. Each view is dynamically created based on the analysis of the logs associated with a query. Views are either default views available for all applications or specific ones. They are based on the analysis of various information sources generated by the scheduler component. Sources include query states and query logs generated during the execution of the application. The logs of the query contain information issued from various sensors gathering information on the remote peer and embedded within concrete services. A YML component generator automatically integrates default sensors which are used to define generic views. Each application can add its own sensors during the creation of YML components. Figure 4.1 presents a standard view. It displays the YML program associated with the currently selected query. One can see the three applications registered on the left and the YML client on top. On the right of the screen, the top of the content of the page lists all the views available for an application. This example application is MERAM. It is a linear algebra method and will be described in chapter 8. The figure shows that a MERAM application defines five views. It makes use of four standard views: *Status* which displays statistics on the application execution (number of tasks waiting, executing, etc), *Query* showing the YML workflow description of the application (this is the view presented on figure 4.1), *Log* displaying the logs of the application execution and finally *Workers* listing the contributions of remote peers. The last view is specific to MERAM and is discussed in the next section.

4.3.2 Applications integrations

The integration of an application in the web portal consists in defining an application description script and dropping it in a special folder of the web portal. The portal detects the new application script and displays a set of entries in the application list.

Each application register provides four pages. The first page contains the description of the application, a list of the available views. The second page is used to schedule a new execution. It proposes a wizard which guides the user in setting up a new application. The third page allows to monitor current and past executions of the application. Finally the fourth page is the application execution monitoring.



Query Source: af23560-meram-m20-r2-n4-20060110

```
<?xml version="1.0"?>
<yml-query login="admin" password="1010">
<!--
ymlportal:app: meram
-->
  <application>
    <source>
#af23560
n                := 23560;
m[1]             := 20;
m[2]             := 15;
m[3]             := 10;
m[4]             := 5;
r                := 2;
thresold         := 1e-8;
maxIter          := 400;
nbProcess        := 4;
internalNodes    := 3;
maxLeaf          := internalNodes + 1;
par (eid := 1 ; nbProcess)
do
    compute MeramStart1(I[eid], n, eid);
    seq (i := 1 ; maxIter)
    do
        compute MeramArnoldi1(H[eid], B[eid],  n, m[eid], I[eid], eid);
        compute MeramSolver(ValShared[internalNodes + eid],
                            VecShared[internalNodes + eid],
                            ResShared[internalNodes + eid],
                            H[eid], B[eid], n,
                            m[eid], r, thresold, eid);
        # Redution operator
             compute MeramReduce1(VecLocal[eid][3], ValLocal[eid][3], ResLocal[eid][3],
                                  n, r,
                                  VecShared[6], ValShared[6], ResShared[6],
                                  VecShared[7], ValShared[7], ResShared[7],
                                  ResLocal[eid][1],
                                  eid);
```

Figure 4.1: YML web portal monitor: the query view (generic)

An application description script contains several sections. The first section includes a description of the application and of the various parameters involved. The second section describes the wizard used during the application submission. A wizard is composed of successive pages. Each page corresponds to a set of parameters. Parameters defined at step n can be used in step $n + 1$. The wizard can contain logic associated with next and previous hooks of each page specified. This can be used to integrate pre-processing operations of the data. The last section of the application description file lists the views that are available for the current application.

The creation of application specific views is the most difficult aspect of the integration of an application into the portal. Sensors are parsed and organized automatically by the web portal before the view creation is executed. Sensors data are transmitted to the view which needs to filter them and map data to the generate a visual representation.

In the context of MERAM, we added one specific view. It creates a visual representation of the evolution of the various ERAM processes composing a MERAM. Figure 4.2 shows the execution of a MERAM application consisting of four ERAM processes. This view relies on the definition of a sensor that exports the accuracy of intermediate results created at the end of each iteration. Each process is executed asynchronously from the others and benefits from their intermediate results. The study of the restarting strategies requires such visualization techniques.

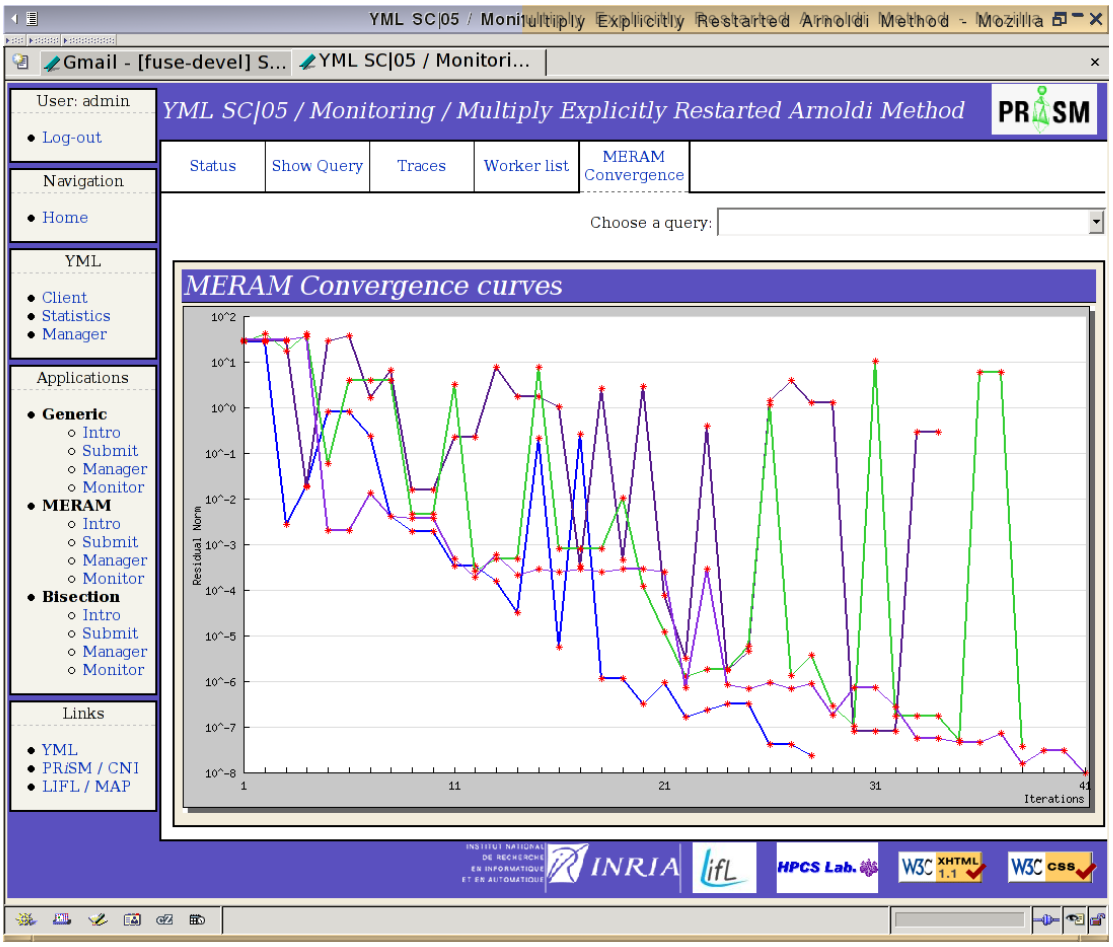


Figure 4.2: MERAM Convergence: an application specific view example

The view model based on sensors eases the separation of the logic and the representation of the data. It is interesting to enforce this separation in order to enable re-use of the logic and the quick definition of various representations showing different aspects of an execution.

4.4 Command line tools

The third front-end is a collection of command line tools which wrap components of the kernel layer into standalone applications. The exposed YML components are the service generator, the workflow compiler and scheduler, a tool to prepare the parameter of a workflow execution, and the data repository component of the back-end layer. Command line tools are not interactive, they are console software easily integrated within scripts. They support the two graphical front-ends presented previously.

An ongoing work called the manager component aims at providing web services which wrap all the command line tools presented in this section. The goal of the manager is to act as a broker for remote clients of YML.

4.5 Conclusion

YML defines three front-ends. An IDE is available for applications developers. It provides multiple ways, called views, to edit the document needed to create YML applications. The IDE provides wizards to drive the creation of services as well as workflows. Once an application has been defined, a second front-end can be used to exploit YML application. It consists in a web portal. It covers the pre-processing, processing and post-processing of applications. The web portal is based on the notion of views and sensor. Each application published on the portal comes with a set of predefined views which are used to monitor the execution of an application. Applications can also register new dedicated views to do post-processing operation or visualization. We described some specific views used to visualize in real time the evolution of the convergence of MERAM as well as global effect on wall of images. The IDE as well as the web portal both rely on a set of command line tools which grant direct access to the component of the kernel and service integration layer.

The web portal described in [48] has been used to present YML on the INRIA booth at SuperComputing conference (SC) in 2005, 2006 and 2007.

Chapter 5

Back-ends: interactions with middleware

5.1 Introduction

In the previous chapter, we presented interactions between the user and YML through its front-ends. This chapter focuses on interactions with middleware.

The component oriented model on which is based the realization of YML defines a back-end layer. This layer is responsible for the interaction between YML and middleware. Its goal is to allow YML to execute a service on a peer. In order to accomplish this, the back-end layer defines three components: the back-end, the worker and the data repository. These components interact with the middleware to provide a consistent execution environment to YML services.

In its current version, YML realization defines two back-ends. One allows the execution of workflow processes on top of XtremWeb [33], a global computing peer to peer system. The other one provides support for the OmniRPC [149] middleware which targets cluster, cluster of clusters and grid middleware. The OmniRPC back-end has been created by Laurent Choy[3]. Through his contribution, he has been involved in most aspect of the back-end layer and contributed to the data repository and worker components which will be presented later in this chapter.

In this chapter, we present the back-end layer of YML. We first discuss the model on which are based all back-ends. We detail the realization of the components of the back-end layer and the integration of XtremWeb and OmniRPC. We then present an extension to the back-end layer which allows the execution of a workflow process to spread over several middleware. Lastly, we present an alternate use of back-ends to simulate and assist the users in the creation of YML workflow processes.

[3]MAP Team, LIFL, University of Lille

5.2 Back-end model

According to our model, back-ends are used to connect YML with middleware. This layer consists in several components which interact to provide a consistent environment to execute services involved in a workflow process. The back-end layer consists in three components named back-end, worker and data repository.

The back-end component is a client of middleware services. It permits YML to request the execution of a work on a peer. It also notifies of work terminations. The application executed on peers is the YML worker component.

The worker component is a service container. It defines a consistent environment for the execution of a service. Among the tasks assumed by the worker, the most significant is data management. The worker is responsible of importing data for the service and exporting its results. In order to do that, the worker component interacts with data repositories.

Data repository components are responsible for all data exchanges between two peers and with the other components. A data repository component provides two services to its local or distant clients: the publication of a resource and its retrieval.

Figure 5.1 represents the interactions with middleware, between the components of the back-end layer and with the other layers of YML. In the remainder of this section we put at the end of some sentence a number inbetween parents. These numbers corresponds to the step of the execution of a work submitted by the kernel layer. If two number are identical it means actions are potentially concurrents. In this example, we present the execution of a single service part of a workflow.

5.2.1 Back-end component

A back-end component hides the specificity of some services provided by middleware behind a common, and minimalist public interface. In 1.1.4, we emphasize on the only widely available facility common to most middleware. It consists in the remote execution of an application. This capability is widely available. Nevertheless the submission mechanism varies significantly from one middleware to another. The back-end component hides the differences between middleware and provides a unified set of features to the other layers of YML, and especially the scheduler of the kernel layer (1a).

The back-end component allows the submission of asynchronous invocation of applications on peers. The corresponding service of a middleware has been generalized in chapter 3 and is named the *resource scheduler*. The role of the latter is to match a received request with an available peer or a set of peers. The selected peer must also match the requirements of the request. The resource scheduler selects arbitrary a peer and assigns it to the execution of the request (3). The content of the request varies significantly from one middleware to another. The back-end component translates YML service execution into requests understandable by the resource scheduler of the middleware currently used (2).

The back-end also polls the middleware in order to get termination notification of active requests (9). A back-end must execute several requests asynchronously. Regularly, YML contacts the back-end component to retrieve the status of submitted requests (10).

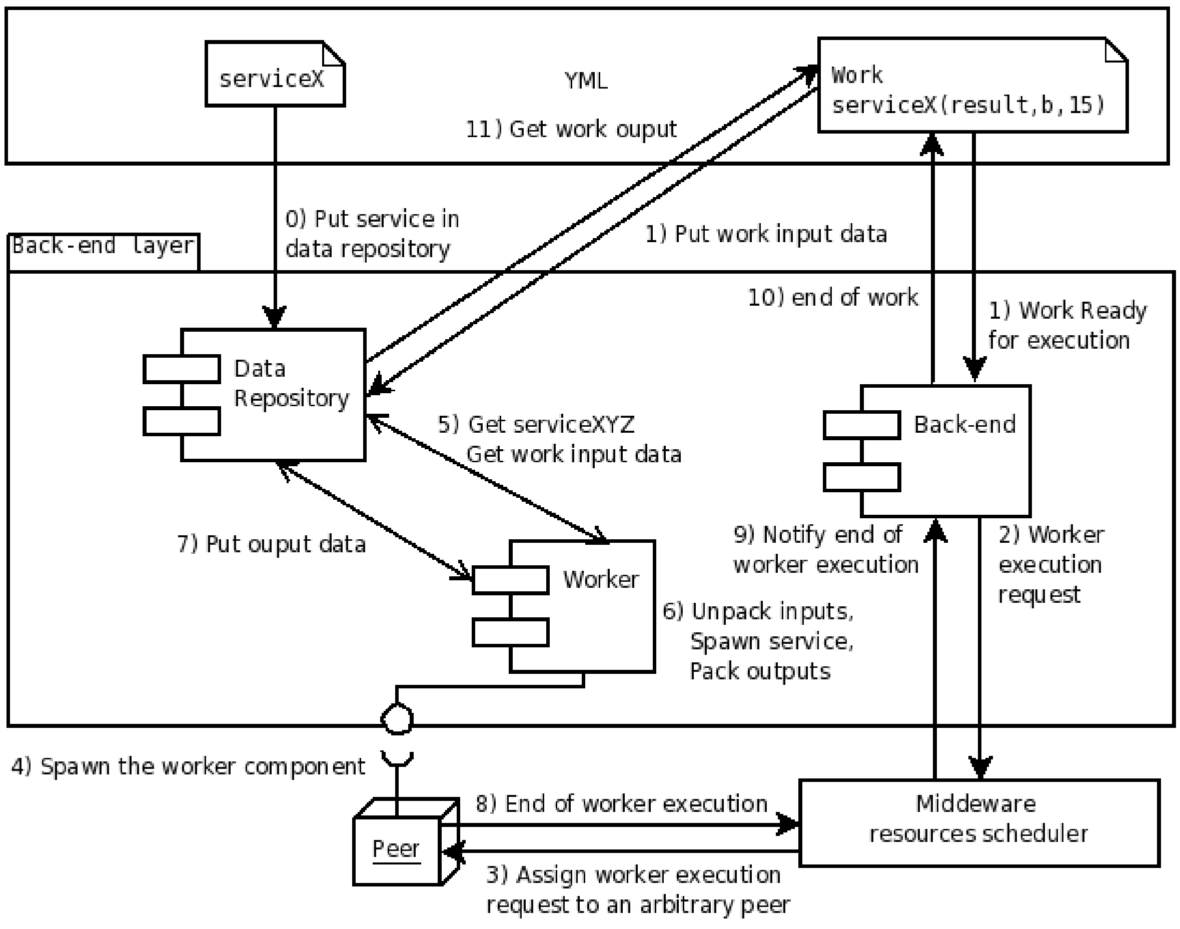

serviceX
YML
Work
serviceX(result,b,15)
11) Get work ouput
0) Put service in data repository
1) Put work input data
Back-end layer
10) end of work
1) Work Ready for execution
Data Repository
5) Get serviceXYZ Get work input data
Back-end
7) Put ouput data
Worker
9) Notify end of worker execution
2) Worker execution request
6) Unpack inputs, Spawn service, Pack outputs
4) Spawn the worker component
8) End of worker execution
Middeware resources scheduler
Peer
3) Assign worker execution request to an arbitrary peer

Figure 5.1: Back-end model

A back-end component maintains a list of active requests and a list of finished requests. The polling of the middleware allows the back-end to move requests from the active queue to the queue of finished requests.

In YML, a request consists in the execution of the worker component on a peer. Thus, the back-end allows YML to ask the middleware to execute many instances of the YML worker and to be notified when one terminates. The back-end component is a client of middleware services.

5.2.2 Worker component

The worker is a component responsible for the execution of a service. Services represent computations needed for the realization of a workflow process. In order to allow the execution of a service by a middleware, we introduce the notion of worker. It is a component which acts as a container for the execution of a service. The execution of a service (6) can be decomposed in the following steps:

1. retrieval of the work description,

2. retrieval of the concrete service binary (5),
3. retrieval of the input data (5),
4. execution of the concrete service binary on the input data (6),
5. publishing of the output data (7),
6. cleanup

When the worker component is started on a peer, it first analyzes a work description. This work description contains all the information needed to execute the service. It consists in a list of resources to retrieve from the data repository component, the parameter of the service as well as the destination of the results. The resources retrieved are the service and its input (5). The service is then executed. It generates a set of results as well as some meta-information such as the status reported by the service, the list of results produced and a trace of the execution of a service. These information are published to the data repository component together with the results of the service (7). The worker finally cleans up the peer and finishes its execution (8).

We introduce first the worker component to limit the number of applications which needed to be registered in the middleware. The benefits of this approach are multiple:

- The addition of new services is simplified. Services are registered once to YML.
- A service can be changed more easily. It does not need to be registered to the middleware every time it is changed.
- Services are instantiated on demands. One service can be executed numerous time concurrently.
- Middleware are not aware of the exact service used and there is no way to differenciate a workflow process from another.

The worker provides a consistent environment from one middleware to another. Through the use of the worker, all services can be shared between supported middleware. They do not include any mechanism to handle the specificity of middleware. The aim of the worker component is also to factorize all operations common to all services. Services are limited to the minimum, that is the computation.

In the context of large scale distributed systems, peers are shared between many users. As a consequence, each execution of an application on a peer must revert all changes made to the peer at the end of its execution. This aspect is most of the time handled by middleware. However, it is not always the case. Thus, the worker must enforce peers to be left unaffected after the execution of a service.

5.2.3 Data repository component

The data repository component is used each time a data transfer is needed between YML and a worker. The data repository component provides an exchange area between workers and is meant to support indirect communication mechanisms introduced in the architecture model. The model requires the component to support two operations: *put(resource)* (0,1,7) and *get(resource)* (5,11). In the context of the data repository, a resource is a name associated to a memory area containing arbitrary data.

5.3 XtremWeb and OmniRPC back-ends

5.3.1 XtremWeb back-end

5.3.1.1 XtremWeb overview

XtremWeb [33, 72, 198, 73] 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 work requests. The system defines three roles: dispatcher, clients and 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 communications in XtremWeb go 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.

Clients and workers are the two roles which are commonly assumed by peers. A client submits a work to the dispatcher. A work consists in an application and a set of data. The work is stored on the dispatcher and the client received a work identifier. The work identifier is used to check the status of the work and retrieve its results on completion. Workers assume the execution of work. They iterate through the following actions: contact the dispatcher to require a work, download the work, execute it and transmit the results back to the dispatcher.

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 work.

Figure 5.2 details the various steps needed to run a work on XtremWeb. We previously explained that all communications go through the dispatcher and that XtremWeb does not allow direct communication between workers and clients. The execution of a work requires at least 6 steps. The client submit a work (1). The dispatcher stores the work and wait for a work request from a worker (2). The dispatcher transmits the work 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 work completion (5) until the results are available on the dispatcher (6).

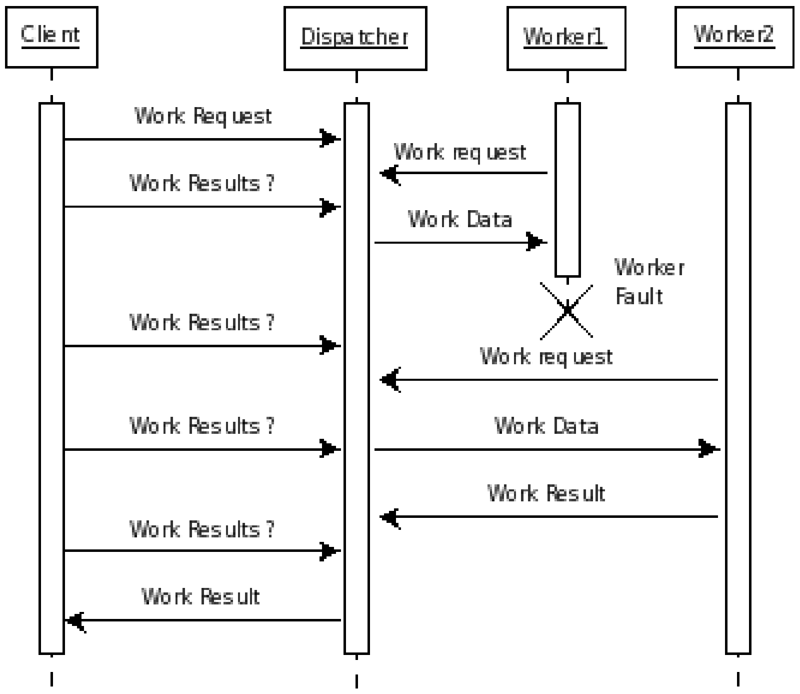


Figure 5.2: XtremWeb Middleware: work execution

5.3.1.2 XtremWeb back-end

The back-end behaves like a client of the XtremWeb systems. XtremWeb client interact with the dispatcher using a protocol similar in to the XML-RPC specification [197]. The XtremWeb dispatcher understand a slightly not compliant version of XML-RPC. The back-end use this protocol to communicate with the dispatcher. Any communication with the dispatcher is done using its own TCP connexion. The actions supported by the XtremWeb dispatcher are really basics. It is possible to submit a single work, to retrieve its status and its results. However you cannot check for the status of a set of works in a single action. It means that each time you poll XtremWeb to check if one of the work submitted by the back-end is finished, you have to check the status of each running work individually. The time needed to retrieve the status of all work currently ruining is significant. The back-end also suffers from the lack of notification systems. The back-end must poll regularly the dispatcher for finished works. XtremWeb back-end is not oriented toward performance. This back-end introduce a lot of latency and it can takes several seconds from the time of a work submission to the start of the work on a peer.

Nevertheless, XtremWeb is interesting as a back-end for the following reasons: the scalability of the platform is important. XtremWeb is able to execute millions of work in a week on a real platform composed of desktop computers and workstation of a university campus. XtremWeb is also representative of the most difficult class of middleware targeted by YML. It is dedicated to the execution of embarrassing parallelism based application such as Monte-Carlo simulations [118]. It is also useful to define the minimum set of requirement which is expected of a middleware.

5.3.2 OmniRPC back-end

5.3.2.1 OmniRPC overview

OmniRPC [149, 120, 131, 192] is a middleware which targets master worker applications for clusters, cluster of clusters and grid(globus, PBS, Sun Grid Engine) environments. OmniRPC proposes a thread safe API for the execution of synchronous and asynchronous RPC and the associated runtime.

RPC frameworks all follow the same mechanism. An interface definition language (IDL) is used to generate most of the code of the service. In OmniRPC an IDL defines the concept of module. This last can contains several definition (or service) which incorporate both the interface and its implementation in C or Fortran. IDLs are processed using a generator which creates the remote executable.

Remote executable (REX) are programs which provides one or several methods to be invoked by clients. OmniRPC does not manage the deployment of the remote executable which is left to the user. As OmniRPC targets clusters, cluster of clusters and grids middleware, the deployment of remote executable binary is often simplified by distributed file systems.

OmniRPC supports applications written in the master/worker[152] model. A client contains the control flow of the application and delegates time consuming operation to distant peers. The communication between the client and the remote executable depends on the invocation method specified for the peer executing the application. The communication mechanism used might depend on an agent responsible for the invocation of the remote executable and the communication with clients. The agent can also be responsible of managing a subset of workers.

Figure 5.3 shows the execution of a work using OmniRPC and agents. The difference with XtremWeb regarding communication mechanism is significant. In OmniRPC communication are initiated from both master and workers. The agent is used to relay and multiplex communication between master and workers.

5.3.2.2 OmniRPC back-end

The OmniRPC back-end allows YML to make use of OmniRPC in its version 1.0. The back-end also supports version 2.0 of OmniRPC but it does not take advantage of it yet. OmniRPC back-end relies on the worker and data repository components. The worker component is register and deployed as an OmniRPC remote executable. As we have seen previously, the worker interpret the work description, acquire the service, its data, and control the service execution. The worker and data repository components were first introduced to overcome the limitation we faced in OmniRPC in its version 1.0. The two components are now used by all of our back-ends.

The recently available version 2.0 of OmniRPC introduces many appealing features, such as file parameters, integration with data persistence mechanism and OmniStorage[5] for optimizing communications.

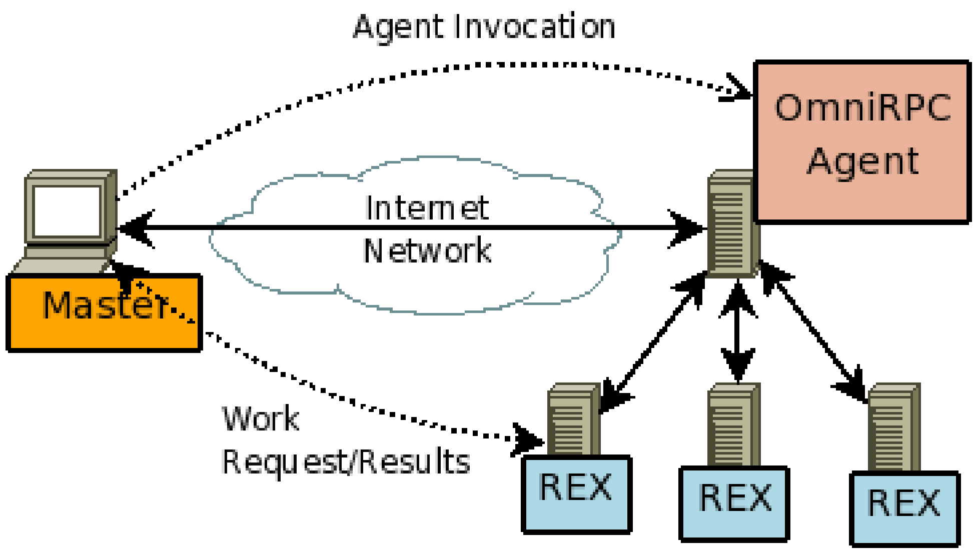


Figure 5.3: OmniRPC middleware: overview

5.4 Globalization of computing resources through dynamic middleware selection

A back-end allows YML to exploit one middleware. Middleware interoperability is achieved by the model presented so far. However, the model does not allow to exploit multiple middleware at the same time. This section presents an extension to the back-end layer which allows YML to exploit several middleware at the same time.

5.4.1 Motivations

Computing tools available on high performance systems try to benefit from the most larger scope of architectures. Successful solutions are available on multiple architectures to provide a consistent environment to end users. The evolution toward cluster of clusters and grids leads to heterogeneous architecture. In a near future, the evolution of multi-core processors or the integration of specialized processing unit will lead to heterogeneity at the level of processors. Many application can benefits from this heterogeneity.

Similarly middleware evolved in numerous directions. They tend to provide solutions adequate to specific usage such as massive data processing, independent low communication processing, etc. This heterogeneity is interesting to exploit this heterogeneity during the allocation of resources to works of a workflow process.

The aim of the globalization of computing resource is to provide an extension to the model of back-end presented before. This extension allow YML to use several middleware to execute YML applications. In the meantime the solution allow several workflow to be executed concurrently on the same collection of middleware. This solution provides global management of computing resources and dynamic selection of the middleware used to execute works of a workflow processes.

5.4.2 Extensions to the model

In order to globalize computing resources, we extend the model presented in section 5.2. The extension consists in three components. A *multi-back-end back-end* component, similar to the one used for XtremWeb or OmniRPC, connects to a *back-end manager* component. The *back-end manager* is associated to one or several middleware using the third component named the *back-end connector*. This last is used to bind regular back-ends such as the one for XtremWeb and OmniRPC with the *back-end manager* through a network connection. Figure 5.4 shows the components introduced to allow the globalization of resources. On the left, we represented the initial model, and the right side of the figure presents the component of the extension.

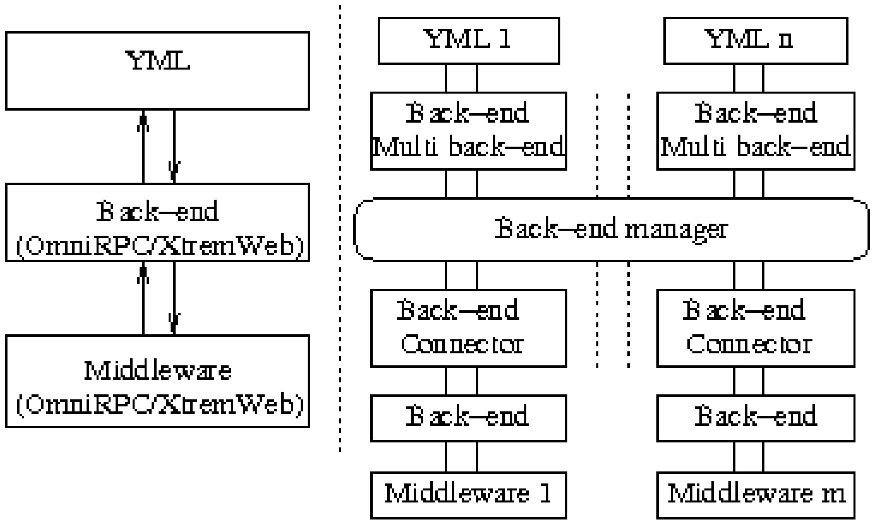


Figure 5.4: Globalization of computing resources using back-end managers

5.4.2.1 Back-end component

YML interacts with the middleware through a back-end[4]. This is unchanged by this extension. However, the back-end component used does not communicate directly with middleware. Instead, it establish a connection with the back-end manager. This back-end is connected with the back-end manager via a network. This is the first step toward a distribution of YML. It also allows several workflow processes to be executed concurrently by the same back-end manager.

5.4.2.2 Back-end connector component

On the opposite, regular back-end components such as the one for OmniRPC and XtremWeb are used to interact with middleware. This is unchanged. However, they do not integrate any mechanism to communicate with the back-end manager. We introduce a component

[4]called multi back-end

named the back-end connector. This connector is used to relay information between the back-end manager and the back-end component.

5.4.2.3 Back-end manager component

The back-end manager is the main component of this extension. It behaves as a proxy between workflow process schedulers, presented in the next chapter, and middleware. The back-end manager interacts with two kind of components presented previously. The back-end manager act as a proxy between a set of YML workflow processes being executed and a set of middleware providing computing resources.

The back-end manager defines the notion of clients. Clients are of two kind: workflow processes (or applications) and middleware. The two group of clients interact with each other through a component internal to the back-end manager called a dispatcher. Its role is to dynamically assign work to middlewares. In order to do that it can makes use of information such as history of previous works execution, middleware statistics and load.

The dispatcher component is defined in order to allow future research in scheduling and allocation strategies multi-middleware. Many strategy can be defined which can enforce some of the following criteria: fault-tolerance, quality of services, computing capacities, software availability, load-balancing, etc. These topics need a lot of experiments and open numerous researches related to scheduling strategies in the context of multiple middleware.

5.4.3 Conclusions and research perspectives

Due to the component oriented design methodology used in YML, we defined an extension of the back-end layer which reuse existing components of the back-end layer, and requires no change to the other layers of YML. This extension allows a globalization of computing resources. It means that several workflow processes can use concurrently multiple middleware to support their executions.

The design presented previously allows transparently the creation of hierarchies of back-end managers. It also allows to create pool of back-end which interact with a single middleware without any modification to the current realization. This aspect of YML is highly experimental. However, the perspective of research in scheduling and load balancing are numerous.

5.5 Execution, simulation and debugging of applications

YML is a framework to develop and execute applications for large scale distributed systems. The back-end layer hides the heterogeneity of middleware and computing resources. However, the development of a complex workflow is still a difficult process. Specific back-end components can be used to define helper tools. They assists end users during the testing and validation of workflow processes. YML defines two additional back-ends used solely during the development of the workflow process.

5.5.1 Process back-end

The process back-end is used to execute the application on a single host without using a middleware at all. It relies on the capabilities of multitasking of modern operating system to execute a workflow process. The behavior is identical to the execution on a distributed system. This back-end is not a simulator because the execution effectively produce the result similarly to an execution using a standard back-end such as OmniRPC or XtremWeb. The process back-end integrates a worker component. Service involved in the execution of a workflow are used the same in the context of the process, XtremWeb and OmniRPC back-end.

5.5.2 Dot output

A workflow execution can be represented using an acyclic directed graph which start from the start node and reach the The dot back-end does not execute anything as do the process back-end. It creates a graph representation which can be processed by the Graphviz tool in order to display visually the dependencies between tasks composing the application. It provides a visual representation of the execution of an application where all tasks have an identical execution time.

5.6 Conclusion

Back-ends assume a critical role, they are the interface between middleware and the rest of the workflow environment. To highlight the importance of the back-end layer, we make an analogy between a parallel computer and a middleware. Indeed, if one can only access to the hardware of a high performance computer, the creation of application is really difficult. An high performace system is not only naked hardware, it is also a collection of software layers. Each layer increases the abstraction level in regards of the hardware. This is similar to a middleware. The aim of a middleware is to provide an access to distributed hardware resources. A middleware does not have to do anything else. Its role is to provide a collection of API to allow the end-user to exploit distant resources. Those API coupled with middleware resources provide the same level of abstraction as operating system and hardware of high performance systems.

The heterogeneity of peers involved in a middleware and the lack of standards in the API makes the development of application really complex. This motivates the definition of an homogeneous runtime environment on top of them. This is the aim of the back-end layer. The back-end layer increase the level of abstraction by simplifying the usage of the middleware and providing a single mechanism to access them. It allows to create a runtime environment composed of resource from multiple middleware. This is similar to some extent to the use of several high performance system coupled using solution such as PVM or MPI_Connect.

YML in its current version support two middleware: XtremWeb, a global computing environment and OmniRPC, a GRID middleware. Through the mechanism of back-end and the introduction of an additional layer, we presented the notion of back-end manager, which can be use to combine several back-end to support the execution of an application. Whether one or several middleware are use is hidden to the other layer of YML.

Ongoing works on the back-end layer are related to performance imporvement. At the time being, the overhead introduces by the back-end layer is high and can be significantly reduce. Works in this area includes cache mechanism applied to the concrete service as well as to the input of each service execution. This last aspect is the most difficult to achieve in the current implementation. Indeed, to makes the communication easier, we are currently pack the results on persistent storage and we send them afterwards. A better approach would be to create the pack directly while sending the data in order to avoid this uneeded copy on persistent storage. The use of packing mechanism for collection of files is also incompatible with cache mechanism. In order to allow caching of service execution data, we need to first remove the packing of data and make each resource independent. This require to extend the data repository to support new operations. In the mean time, the collection of information about peer is really basic at the time being, a more detail collection of information which could include, memory, CPU information or storage available could be really interesting in the context of allocation of service execution to peers available.

Chapter 6

Services integration and workflow management

6.1 Introduction

In the previous three chapters, we presented the modeling of a workflow environment called YML, the interaction between end-users and YML and the interaction between YML and the middleware. This chapter discusses the kernel layer of YML. The kernel is responsible for the management of services, the compilation and the execution of workflow processes.

In YML, a workflow is defined as a collection of independent, reusable services. Services are coupled with each other by using a workflow process definition. The workflow process definition is a description of the control flow of the application.

The remainder of this chapter is organized in five sections. The section 6.2 presents the notion of services and the mechanisms available to integrate them within YML. The section 6.3.1 describes the workflow processes description language used. Section 6.3.2 presents the theoretical model used internally by YML at the time of the workflow execution. Section 6.3 presents the compiler and the scheduler of workflow processes. Finally, we conclude the presentation of YML.

6.2 Services integration

A workflow process describes the interaction between services involved in its execution. The scheduling of a process consists in accurately instantiating services on peers. In this context, accurate means that services must be instantiated with the appropriate input data when all services on which it depends have finished their execution. The instantiation of a service is handled by the worker component described in 5.2.2.

YML describes its services as XML documents. Three kinds of services exist. They are needed to allow the separation between a middleware independent application representation and the execution of the concrete workflow on a chosen middleware.

6.2.1 Abstract, graph, concrete and built-in services

YML is modeled as a collection of independent components which interact with each other by using well defined public and abstract interfaces. Services also follow this rule. However, the principle of separation between the interface and the realization of a component is enforced by the need of two definitions for a working service. A first definition is used to specify the public interface. This interface is named *Abstract service*.

An abstract service focuses on the definition of the inputs and outputs of a service. A component can define three kinds of communication channels. A channel can be one sided and support either input (noted *in*) or output (noted *out*). It can also be two sided (noted *inout*) and support at the same time input and output. Channels are strongly typed. Only two channels of the same type can be connected to each other. Abstract services are similar to a declaration in the C language. It is used by the compiler of workflow processes for validation purpose.

An abstract service definition is an XML document. Listing 6.1 is an example of such a definition for a really simple component which applies a binary operator to two real numbers. More complex examples will be presented in chapter 8. In this example, we define three channels of type `real` using either inputs or output and one channel of type `string` to specify the operator expected in input. The abstract service is named `binaryOp`.

```xml
<?xml version="1.0"?>

<component type="abstract" name="binaryOp"
  description="Apply a binary operator to the two operands op1/op2." >
  <params>
    <param name="result" mode="out" type="real" />
    <param name="op1" mode="in" type="real" />
    <param name="op2" mode="in" type="real" />
    <param name="operatorName" mode="in" type="string" />
  </params>
</component>
```

Listing 6.1: Example of abstract service

The definition of an abstract service does not incorporate any information on how the service is realized or what it effectively does. This is just the public interface of the service. There are two ways to realize or implement a service in YML. One can create either a *concrete service* or a *graph service*.

A concrete service defines a binary application which is going to be executed on the underlying middleware. For the same abstract service, multiple concrete services can be registered. Concrete services are also refered as implementation services because they incorporate code in one of the language supported by the YML component generator services. Concrete services are similar to the definition of a function in a third party library. The function can be a function in C or in another language interoperable with C such as Fortran. At the time of this writing, it is possible to create components using the

C or the C++ language. Concrete components can depend on external libraries. We will present this aspect later in this section.

```
<?xml version="1.0"?>

<component type="implementation" name="binaryOp_impl"
  abstract="binaryOp">
  <impl lang="CXX">
    <header>
// this is used to place C++ code at the beginning of the generated
// service
    </header>
    <source>
// Here we write a C++ program which can use the name of
// channel previously defined in the abstract component.

// The three channels are required, here we enforce this
if (!op1_param.exists() || !op2_param.exists() ||
     !operatorName.exists())
     error(); // One of the required channel is missing

if (operatorName == "-") result = op1 - op2;
else if (operatorName == "+") result = op1 + op2;
else if (operatorName == "*") result = op1 * op2;
//... more binary operator
else error(); // OperatorName does not match any supported one.
   </source>
   <footer>
// This is used to place C++ code at the end of the generated code
   </footer>
  </impl>
</component>
```

Listing 6.2: Example of concrete service

A concrete service definition is an XML document too. Listing 6.2 relies on the example of the binary operator abstract service. This example presents a realization of the abstract service using the C++ language. Concrete services are tagged with the type of implementation. The implementation refers to an abstract service name, in our example *binaryOp*. This information is used to match a concrete service with its abstract service. It is also used by the service generator component (discussed next) to create the skeleton of the service. The user is only responsible for providing the logic of the service. All other aspects are handle by the service generator discussed in the next subsection. For this example we do not require any include or any function, thus we only need to complete the *source* section of the component definition. For compactness, we could have omitted the header and footer section completely. In the source section, we first check the existence of the input parameter. Indeed, YML allows missing channels. If the channel name does not exist, it does not prevent the execution of a service. This feature is used for asynchronous coupling of iterative methods discussed in chapter 8. The C++ generator for YML defines a channel information object called `<channel>_param` which can be used to check various channel states. If one of the required channels does not exist, the service stops, returning an error status. Otherwise we analyze the value of the channel *operatorName* and apply the corresponding C++ operator to the two operands and assign the result to the channel result.

The second mechanism to realize an abstract service is to create a graph service. A graph service allows to define sub-workflows that can be used in other workflows. These components are defined using the workflow process definition language presented later in this chapter. The language does not enforce any mechanism to do the expansion of the sub-workflow during the execution. It is for example possible to do parameter substitution to incorporate the content of the graph service directly within the initial workflow. In this case the integration of the sub-workflow occurs during the compilation of the workflow. On the other hand if the sub-workflow is delegated to a new instance of the workflow scheduler then, the sub-workflow expansion happens at the time of the execution and thus is a more dynamic mechanism. The creation of such services relies on the definition of an XML document like the above. The differences lie in the `type` attributes which take the value `graph`, and the `impl` XML tag and its contents are replaced by a `graph` tag. The contents of this tag are discussed in section 6.3.1.

Finally, the last kind of services are built-ins. They are similar to concrete services in many aspects. However, they are part of the scheduler. Built-in services are executed by the scheduler to modify the data of the application. They are executed directly on the data store of the application. On the opposite, concrete services are executed on peers provided by a middleware. Existing built-ins include deletion, rename and copy of data manipulated by the applications. Other built-ins include operations on data types directly supported by YML. Operations such as number addition and multiplication are also available.

YML distinguishes itself from many other workflows in the way it integrates external software. YML uses a workflow not as a tool to do code coupling but as a programming model. Thus, it does not provide mechanisms to harvest existing services as it is the case in projects such as Askalon, GridFlow or Kepler. In that respect, YML is more similar to the approach used in GridAnt. In order to be able to list existing services, YML relies on two catalogs.

6.2.2 Development and execution catalogs

The communication between the services integration module and the workflow compiler and scheduler is handled by two catalogs. They are used to store information about services. The development catalog contains all the information needed during the compilation of workflow processes. The execution catalog is only used during the execution by the YML scheduler. Both catalogs are realized as components. The implementation of the two catalogs can be changed easily.

The development catalog stores abstract and graph service definitions alongside with a list of existing channels or parameter types supported. These information are independent of the middleware used. They are used mostly by the YML workflow compiler to produce a middleware independent compiled workflow. The realization of the component can be change at runtime to better integrate with middleware catalog services if needed. This catalog plays the role of the harvester services of other workflow environments. It is also interesting to note that by simply changing the realization of catalog, it is possible to introduce harvesting technics within YML.

During the execution of a workflow, the compiled workflow is analyzed and converted into a concrete workflow. Every abstract services are substituted to corresponding concrete services stored within the execution catalog. The execution catalog stores information about the list of concrete services available. This catalog is used only during the execution of a workflow process by the scheduler. As we have seen before an abstract service can have several implementations. The selection of the most adapted concrete component available can be made using several criteria such as the complexity of the algorithm used, the complexity in memory or storage. However, this kind of metrics is not reliable. With the introduction of the back-end manager, the selection of the most adapted concrete service is even more complex. This extension would benefit greatly from the availability of more information about the previous execution of services. Indeed, the selection of the most suited concrete services can be improved significantly with information on past execution of services and introduction of machine learning strategies. This is on the perspective of research which can be followed to enhance the scheduling mechanism of YML. For the same reason as of the development catalog, the execution one is defined as a component for which the implementation can be changed to integrate best within the middleware or the user environment.

6.2.3 services generation and registration

In order to be usable by the kernel layer components, a service must be registered within one of the two catalogs. The registration of abstract and graph services is straightforward and handled directly by the development catalog component. The registration of concrete services however is more complex due to an additional step of generation.

The service generation component converts a concrete service into a binary component. This component can be instantiated by the worker defined in the back-end layer. The tasks of the generation of a component are tightly associated with the behavior of the worker. As we have seen in chapter 5, the worker provides a consistent environment for the execution of services on several middleware. The worker is responsible for the following tasks:

1. acquiring the work data including the service itself,
2. executing the service,
3. transmitting the results back to YML,
4. cleaning up the peer.

The worker behaves as a container for the execution of services. Because all the communications with YML are handled by the worker, the service only needs to communicate with the latter. In an heterogeneous context, relying on mechanisms such as network communication or pipe is dangerous and introduces dependencies with the underlying architecture. The only way to communicate in a portable way with another program is to use files.

6.2.3.1 Peer heterogeneity

We don't emphasize on the heterogeneity of peers involved in the middleware so far. We identify two levels of heterogeneity. The first level concerns the architecture and especially the order of bytes used to represent multiple bytes data such as integer or floating point values. The second level of heterogeneity concerns the operating system. The latter is more difficult to solve than the first one. However, we do not consider this as a research topic. Many solutions exists to solve this. Java is one of the most well known solution. However it is not suited in our context as it does only work well for applications written completely in Java. This goes against one of our motivations, as we wanted to be able to integrate existing libraries within the services part of workflow processes. One of the other possible solutions is the one proposed by XtremWeb. It consists in providing several binary executables of the worker and of each service to match the heterogeneity of runtime environments. This can be made transparent to the end-users by setting up a cross-compilation environment alongside of YML. It is also needed to extend the service generator component to generate multiple concrete services for each runtime-environment. We do not consider this aspect challenging and assume that this is only subject to implementation details.

However, in order to allow the above modification of YML at a later time, we represent each service by a binary executable. The communication between workers and services relies on files. This is hidden to the user and can be changed afterward if a more efficient solution is chosen. There is potentially another significant advantage of this approach. A service does not have to use any operating system specific mechanism to communicate. It eases significantly the cross-compilation of services. Indeed services can depend solely on standard language features, services can thus be made portable at the application level.

6.2.3.2 Implementation language

In the example of a concrete service we presented earlier in this chapter, we used C/C++ to realize our components. This is currently the only language supported in YML. However, each language is supported by two components.

A source code generator component is used to produce a first view of a concrete service. This view is a program source code in the language of implementation of the service. It is then processed by a binary compiler component which converts the source code into a static binary application. The selection of the two components above is made based on the definition of the attribute `lang` in the concrete service definition. The creation and registration of support for new languages consist in providing a new pair of components, one for the source code generator and one for the binary compiler.

6.2.4 External library integration

The integration of an external library consists in providing some services which allow to use some of the features proposed by the library. This process is not automatic and cannot be. The services in YML are stateless, they cannot share information between multiple instances. Each service derived from a library must be made standalone.

We identify two concurrent activities to register partially or completely a library as a set of services. The first step consists in defining data types. Data types are one way to extend YML. They are used to define channels between services. YML defines in standard only a few types (integer, real, string and raw[5]). The integration of a data type is dependent on the language support in the generator. However, once made available it is transparent to the user of services. The definition of new types for YML can be summarized as the definition of two functions, procedures or methods depending on the language terminology. One is used during the import of input data. The second one is used during the exportation of the output data. In order to allow the integration of any kind of library, YML does not require a type to be built upon the basic types provided by the system unlike CORBA.

Once types are defined, library features must be translated into abstract and concrete services. Services often consist in a few calls to the library function. Most of the logic being handled within the library. If we consider the example of library like Blas, The best approach to integrate such a library consists in first definining new data types for matrices and vectors of real and complex. Once type are defined, it is possible to provide a simple wrapper which consists in a single call to all the procedure composing the library or a subset. This approach has been used to a subset of Blas and LAPACK. As examples of external library usage, the applications presented in chapter 8 are using libraries such as zlib, libpng, BLAS, LAPACK, rayshade and LAKe. Aside of that, the integration of BLAS and LAPACK as a mathermatical toolbox for YML is an going work.

6.2.4.1 Integration of the LAKe library

LAKe is an object oriented library written in C++. It defines a framework to implement iterative solvers. The LAKe design is organized in two parts as shown on figure 6.1. The left section contains numerical algorithms and services. The right section focuses on data management classes. They are used to represent both sequential and parallel data type used by LAKe numerical part. Using the object oriented approach and template based generic programming provided by C++, LAKe allows the numerical part to be common to both sequential and parallel versions of the application. The numerical part of the library is identical in the case of a sequential data set or distributed data set. The parallel version of LAKe makes use MPI for the communication between processors. However the use of MPI is completely transparent to the user. He/she can switch from a sequential solver to a parallel one by changing the type of the matrix representing the data. A `Matrix` data type is used for sequential version and `DMatrix` is used for parallel version of algorithms. LAKe achieves code reuse between sequential and parallel versions thanks to a strict separation between the numerical side and the data management aspects of applications.

[5]This is used to represent an arbitrary number of unformed data.

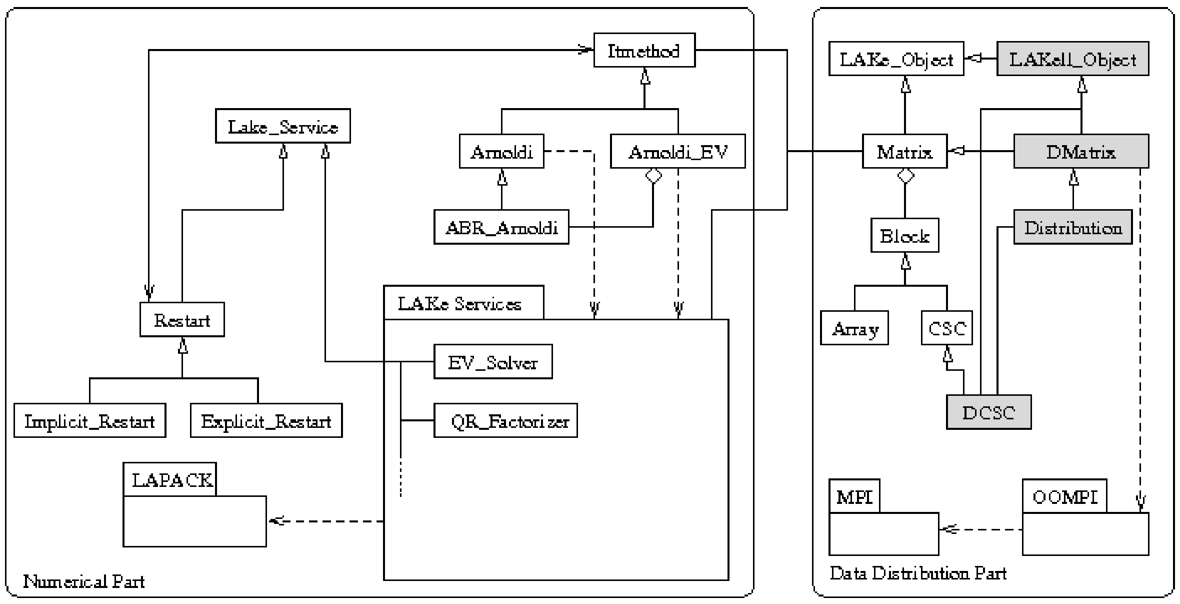


Figure 6.1: LAKe: Design and Architecture

The integration of a library such as LAKe is a straightforward process because of the object oriented approach used. Indeed, the decomposition in service is almost directly given by figure 6.1. The first step in integrating LAKe consists in defining data types. LAKe manipulates three kind of objects that must be exchange between services: scalars, dense matrices and sparse matrices. Scalars can be either an integer or a real. Both of this type are supported by YML natively. They are part of the language. Dense matrices are used in LAKe to represent a matrix or a collection of vectors. There is no explicit notion of vector in LAKe. The meaning of the matrix is defined using a Partition. A Matrix object can be decompose in subobject using a partition. Partition are also used to validate the validity of operation such as matrices products or matrix vector products.

LAKe makes intensive use of a design pattern called *service*. This pattern defined a mechanism which allows to loosely connect a service with its operands (in the context of LAKe, those are mainly matrices). A LAKe service behaves similarly to the notion of service of YML. Indeed, LAKe service expose a feature for the rest of the application to use. The execution of a service requires first the connection of its operand. Once all operand are connected the execution of the service itself is made possible. Before releasing the service, one need to first disconnect the operand. This pattern is similar to the execution of a service by the worker of YML. The main difference between an application making use of LAKe and the integration of LAKe as a collection of YML services is the lack of persistent state between two service execution. LAKe service are all translated directly into YML service. However, LAKe is more than just a collection of services. It also provides a framework to build iterative methods upon. Each iterative method is a construction of a set of service. It defines the control flow of a method. Those are not translated to abstract/concrete services. Indeed it would not allow any parallelism. Instead, they are translated to graph service or directly as an application workflow. This is the case of Arnoldi_EV in the diagram presented above. An exception is the Arnoldi class which is implemented as an iterative method in LAKe. The sequential Arnoldi is implemented as a pair of abstract/concrete services. The parallel version is however realized as a graph service or directly within the application workflow. The use of LAKe integrated with YML is illustrated in chapter 8.

6.3 Workflow management

The kernel layer is responsible for the management of workflow processes. It has to analyse workflow, convert them into a form suited for their execution, and finally execute them on one or several middleware using the back-end layer. The kernel layer links all layers of YML together and manages the interaction in the whole environment. It acts as a coordinator of the other components of YML. It is mainly composed of two components: a workflow compiler and a scheduler. It also contains the workflow process definition language, and the internal representation of an application.

Workflow processes are defined using a graph description language specifically designed for YML. This language is used to define graph services as well as applications. It is processed by a compiler in order to create an alternate representation of an application. The latter is then processed by a scheduler responsible for the execution of the workflow on the underlying runtime environment exposed by the back-end layer.

6.3.1 Workflow process definition language

In the previous section, we presented the integration of services within the workflow environment. This constitutes the first step in the creation of workflow applications. Indeed, services are used as building blocks for workflow processes. The workflow process definition language of YML is used to describe interactions between services. It is employed in the definition of applications and graph services.

YML workflow process definition language proposes a small and compact syntax similar to pascal and C. The language allows the definition of a graph where nodes are services execution and edges are dependencies. The graph is built based on a control flow analysis of the program. In order to describe the graph, YML proposes several constructions such as service calls, sequential and parallel iterations, parallel sections and events management.

This example shows a way to express a reduction global operation using YML. A second way will be presented while discussing the notion of collections. Examples of reduction operations are the sum of the elements, the extraction of the maximum or minimum, and any other operations which produce one result from a collection of values. Listing 6.3 is an excerpt of a workflow process definition. It shows a reduction operation on a collection of objects. This example illustrates most of the constructions of YML. It assumes that two services exist. The first service is named `create`, it generates an object (a number, a string, etc). The second service is the `reduction` operator and is named reduction. 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 is 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 synchronize the reduction operation, we explicitly use events manipulators **wait** and **notify**. Using two parallel sections, the reduction can start even if all the objects are not created yet. Finally, in the listing below, comments start with a # and terminate at the end of the line.

```
# Const definitions
objectCount := 16; # variable definition (assignment)
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 service 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
```

Listing 6.3: Example of the workflow process definition language

The execution of services is identified by the keyword `compute`. It is followed by the name of the abstract service to execute. The concrete service is selected automatically during the execution of the workflow. It is followed by a list of parameters which denote either data or constant value. It is also possible to give additional information to the scheduler on the nature of the service. Currently, two classes of services are supported in the language: `migrate` and `compute`. The first is used to highlight that the service execution is to be considered as data manipulation or transfer operation with little computation. The second is just the opposite and is used for any kind of computation which requires a large amount of calculus. This information is dedicated to the scheduler which can implement different policies according to the nature of a service.

The concurrency of two services of a workflow is always introduced by the keyword `par`. The first form is `par block1 // block2 endpar`. The second form is equivalent to the first but is used to generate a collection of blocks which depends on one or several iterators. The end of the parallel section is synchronization barrier. In this regard, the parallel construction of YML is similar to OpenMP. A future extension to the language will consist in adding parallel block without synchronization at the end of the execution of parallel constructions. YML also provides iterator construction.

The YML language depends on the notion of events. Each dependency is expressed implicitly or explicitly by using events. Events are similar to boolean values. They are either set or unset. Events are identified by name and can be organized in arrays. However, explicit events are only used as parameters of *notify* and *wait*. The former is used to set an event. Once an event has occurred, it is set for the rest of the lifetime of the application. The latter is used to wait for an event. It is a synchronization mechanism. Integration of exceptions to the language is easy. It can be achieved without any modification of the scheduler component relying solely on the existing internal representation of a workflow, as will be discussed later in this section.

6.3.1.1 Collections

YML supports the notion of collections. A collection is a group of data manipulated by services as a whole. The language collections are introduced using square brackets. For example, `data[1]` denotes the first element of a one dimension collection. `data` denotes the collection as a whole. Index used to denote elements of a collection are integer values. The index can be either a variable defined in the workflow process or numerical expressions. YML supports multi dimensional collections. There is no limitation on the number of dimensions used. Collections are always sparse. The elements are only present when assigned first. Each element of a collection represents data which can be used as a channel for a service. Collections can only be populated through service calls.

Services handle channels/parameters transparently for the user which just has to focus on the computation itself. This approach is efficient for most situations. However, it is restrictive with regard to two aspects. The life time of a channel is not managed by the user. All channels exist from the beginning of the service to its end. A channel can only handle one data. Collections solve both problems. Indeed, collections allow end-users to control the life time of a data. A service creator is explicitly responsible for managing the loading and the unloading of data. This enables services to make use of out of core technics.

The second advantage of collections is the ability to manipulate many different data as a single channel. This allows a service to generate a variadic number of data. It is not possible using regular channels. Indeed, a regular channel corresponds only to a single data. On the other hand, being able to manipulate multiple data through a single channel comes with a cost. All data composing a collection is transfered for each service execution using it. This is however perfectly well suited for reduction, fragmentation or defragmentation of data such as matrices. A collection allows to pass multiple parameters grouped in an array together in order to manipulate them as a whole. Using collections, it is possible to create services which allow global operations such as *all gather* or *reduction* to be present in traditional message passing solutions.

6.3.2 Application storage and internal workflow representation

The workflow process is converted by the workflow compiler component into an application file. This application file corresponds to an execution oriented representation of the workflow. This representation is well adapted to the scheduling of workflow processes.

Many theorical models exist to support the execution of workflows. Among them we can cite projects like tuple space [179] or network of petri [167]. YML relies on a traditional Artificial Intelligence technique known as "expert system". Expert systems, also known as rules engines, are often used in automatic demonstration systems. An expert system relies on two concepts: the notion of facts and the notion of rules. A fact is similar to a boolean value: happened or not happened yet. Facts are injected in the expert systems either by the end user or following the execution of a rule. A rule is composed of a condition and an action. The condition is a boolean expression based on facts. If the condition is evaluated as true, then the action is executed. Actions are most often composed of two steps. A first step is the execution of a side effect, such as the scheduling of a work composing a workflow process and the introduction of new facts in the expert system. These new facts are used to allow new rules to be executed. The goal of the expert system is to demonstrate a particular fact, called *stop* starting from the fact *start*. A rule can generate multiple facts. Each time new facts are introduced, some new rules are potentially executed. In the context of YML, each work composing a workflow leads to the definition of a rule. This rule contains a condition which is a boolean expression based on events. Events are translated into facts of the expert system. The execution of the workflow is equivalent to demonstrating the event *stop*. The sole difference with regular expert systems is the rule which can be used at most once in the execution of an application.

An application file is an archive composed of several resources which represent many aspects of a workflow application. Among them, the archive contains the table of rules and the table of facts which contains the information needed by the expert system. The application file has the following properties:

- middleware independent: the application file does not contain any information specific to one middleware. The application file can be shared between installations of YML as long as all sites use the same version of YML.
- multiple execution: the same application file can be used for executing several applications using different or identical input data.
- self contained: the scheduler does not need to access any information stored in the development catalog. It only depends on the execution catalog to be able to execute an application. The application file also contains information to be used by the other components of YML such as the web portal.

6.3.3 Workflow management components

YML defines three components in the kernel layer. The two main ones are the workflow compiler which converts a workflow process into an application file and a workflow scheduler which executes an application file. The last component is used to manage application input and output.

6.3.3.1 Workflow compiler

The workflow compiler analyses, validates, and translates a workflow process into an application file. The latter is used by the compiler for the execution. Most workflows do not depend on a compilation stage. They act more like interpreters. The workflow is processed, analysed, and executed at the same time. This approach is interesting and has been used in an early version of YML presented in [49]. However, in the current version of YML, the analysis and the execution are two distinct operations which do interact. The aim of this separate stage is to decrease the amount of work needed during the execution of a workflow and to provide a more modular decomposition of YML.

The compiler is composed of a set of transformation stages which lead to the creation of an application file. The compiler expands the graph described by the workflow process. The application file is the result of the packing of a collection of tables produced during the compilation process. Among those tables, the most important ones are the tasks table and the events table.

The current workflow compiler is restricted to the generation of static workflows. Indeed, it generates the whole graph during the compilation process. The graph is thus expanding during the execution. The main advantage of this approach is the complete knowledge of the graph. This can be used for advance workflow analysis and pre-scheduling strategy or mapping of the workflow graph onto peers of the runtime environment. However, the drawback of this approach is that the graph is fixed during the execution of the application. Also, for really large graphs, the compilation time can become large.

6.3.3.2 Application parameter management

The execution of a workflow can depend on input data and produce a set of results. In order for the scheduler to use them, YML uses archives of data for both input and output. The application data file contains some meta information which lists the input and output parameters. The parameter management component is dedicated to the handling of archives used in many places within YML. The parameter management component is especially designed to manipulate archives which are used as input of an application and as output. It allows the user to pass and retrieve data at the beginning and at the end of the execution of a workflow. The component relies on meta information stored in the application file to drive the user in the registration of its input parameters and the extraction of the results.

6.3.3.3 Scheduler

The role of the just-in-time scheduler consists in executing the application on the underlying runtime environment defined by the back-end layer. The scheduler takes an application file together with an input data set and executes the workflow until completion or until the execution triggers an error. The input data set is created using the parameters manager presented above. Once the execution has been initialized, the expert system controlling the execution of the workflow is initialized with the *start* event or fact. Once the system has started, the scheduler applies rules and creates a list of works *ready* to be sent to the back-end component. Once submitted to the underlying middleware, the work status is changed to *running*. This status lasts until the back-end has been notified of the termination of the work which is then in status *finishing* or *error* if its execution has encountered a problem. Once the scheduler acquires knowledge of this, it takes into account the data generated by the work and publishes new events to the expert system in order to activate new works. The work status then becomes *finished*. The execution of the workflow ends when one of conditions below is met: a component failed, a service terminated on an error or has thrown an exception and the event *stop* is generated or there is no more rule waiting to be activated.

The scheduler is executing two main operations sequentially. First, it checks for work ready for execution. This is done each time a new event/fact is introduced in the expert system. This leads to the submission of jobs to the back-end. The second operation is the monitoring of the work currently being executed. Once works have started their execution, the scheduler of the application regularly checks if new works have entered the *finished* state.

The scheduler interacts with the back-end layer through the data repository and back-end components. Every time a work status changed to *ready*, the scheduler prepares the execution of a service by creating a script describing the work. This script is processed by the worker component of the back-end layer. In the mean time, the work channels are stored in an archive in order to be easily exchanged between the data repository and the worker.

As we have seen in the previous section, an application file contains an abstract workflow in an alternate representation composed of facts and rules. This representation does not contain any middleware specific information. As we have seen before, the compiler does produce an application file which relies solely on the information contained in the development catalog. Concrete services are not available. The scheduler is also responsible for the matching between abstract services used in application file and concrete services during the execution of the workflow. This selection is at the time being straighforward. However, it is already possible to implement intelligent selection strategies for the selection of the concrete services used.

6.4 Conclusion

YML is built around the service integrations layer and the kernel layer. The service integration layer defines the mechanisms needed to register services to be used within workflow processes. This is possible thanks to the use of two independent catalogs which store information used on the one hand for the development of workflow processes and on the other hand for their execution. These catalogs are the interface between the service integration layer and the kernel layer.

Services represent computation on middleware peers. YML defines several classes of services known as *abstract*, *graph*, *concrete* and *built-ins*. The first is used to describe how services interact with each other. The last ones are used to describe service actions themselves. In order to compose scientific workflows related to numerical applications, YML integrates the LAKe library which is used for the creation of iterative and hybrid linear algebra methods for the solving of large problems. This integration is the first step toward the integration of general purpose libraries such as BLAS and LAPACK.

Services are associated with each other in workflow processes. The workflow process definition language of YML is used to describe the control flow of the application. Unlike many data flow oriented workflow environments, YML focuses on control flow and deduces automatically control flow. The language allows the definition of general oriented graphs. The synchronization of work between activities is done either implicitly following the workflow process language or explicitly by using the notion of event. This notion is critical in YML and has directly influenced the workflow scheduler. Indeed, the scheduler component uses an artificial intelligence technique known as expert systems to drive the execution of the workflow.

The compilation of the workflow process is done by a component which statically expands all the nodes of the graph before the execution of the worflow. The scheduler use this internmediate form coupled with an expert system engine in order to resolve dependencies among services executions and manage the execution of the entire process. The scheduler depends on the back-end layer for the interaction with the underlying middleware.

The kernel and service integration layers could be improved in several ways. Currently, the service generator component available is difficult to master because the end-users or the service creator have no mechanism to preview the generated C/C++ code. The generator can be used to generate a skeleton of services which is to be completed by end-users. Future work could extend the currently available integration for a few number of significant libraries such as BLAS and LAPACK.

The workflow management area could be enhanced by changing the internal representation of a workflow process to an hybrid model which would allow at the same tinme explicit event synchronization and runtime generation of the workflow nodes. This would enable the application graph to be generated and modified during the execution of the workflow process application.

Appendix A contains more information on the creation of workflow processes.

Chapter 7

YML a priori evaluation

7.1 Introduction

The evaluation of an environment like YML is difficult. The definition of a methodology to evaluate programming environments or programming languages is lacking. In order to compare workflow languages, an evaluation strategy has been suggested in [166, 147, 108]. The suggested methodology consists in evaluating the patterns which can be expressed using a workflow language. These patterns are used to evaluate the capability of expressing standard control flow and data flow constructions. The approach is targeted for visual workflow environments and does not apply to other ones.

The evaluation of YML begins with an analysis of how YML provides solutions to the motivations discussed in 1.1.4. We first present aspects related to runtime environments (middleware and peers) and then discuss the workflow environment built on top of the back-end layer. This constitutes an a priori evaluation as well as a conclusion to the presentation of YML.

7.2 *A priori* evaluation

In 1.1.4, we highlighted several constraints related to the development of applications for use on large scale distributed systems and especially meta-computing environments. The aim of YML is to ease the development of parallel applications on this kind of system. To achieve this goal YML must solve issues associated to peers, middleware, programming language, and supporting tools.

7.2.1 Runtime environments

Large scale distributed systems introduce new issues compared to high performance systems. A high performance system with numerous processors distinguishes itself from a meta-computing environment at several levels. These distinctions are mainly based on the processing element. In a high performance system, the processing element is constituted of one or several cores associated with some memory and a networking interface. However, in a large scale distributed system, the processing element is most of the time a complete computer with its owner, hardware, operating system and runtime environment. It can work on the owner applications and at the same time contribute to one or several middleware. These differences have a strong impact on a programming environment oriented toward the use of large scale distributed systems. Indeed the programming environment should provide mechanisms to deal with peer heterogeneity, peer volatility and security for both the peer and the application.

7.2.1.1 Security

The security of both the peer and the application is one of the biggest issues. Thanks to techniques such as sandbox and virtualization it is possible to enforce the security of the peer. However, no practical method exists to certify that a result has been produced by a specific application. This is a difficult problem in the context of malicious peers. This kind of problem happened in the project seti@home in the past. In order to ensure that the results obtained are genuine, a possible solution consists in executing several times the same computation on different peers of the system. The results are then compared and statistical methods are used to detect malicious peers.

From the very beginning, YML had never provided any mechanism to ensure security. We considered security as one of the duties of the middleware. Indeed, YML expects from the middleware the ability to execute an application. It means the YML worker component is an application deployed and managed by the middleware and thus it is the role of the middleware to ensure security of both parties. However, many middleware do not enforce security at all. Thus, the YML worker ensures at least correct cleanup of the files created during the execution of the services. In the future, the worker could become a sandbox for the execution of services and ensure at least the security of the peer.

7.2.1.2 Heterogeneity

The peers of a large scale distributed system are most of the time heterogeneous. This aspect is difficult to handle completely in a satisfying way. Indeed, there are several problems to solve. The first constraint is introduced by the re-usability of libraries, and especially numerical libraries. For the latter, the use of interpreted languages such as JAVA, Ruby or Python is just impossible. Indeed, numerical libraries are written using languages such as C or Fortran for the sake of efficiency. In order to use them, applications need to be written either in a language which allows the usage of those libraries or provides a binding (or glue). In both cases it means the execution of native, compiled code on the peer. Services must be able to make use of routines written in C or Fortran. Thus services are dependent on the operating system as well as the hardware. Assuming services are written using a compiled language which leads to a binary application directly usable by peers, we have no choice but to provide several versions of the service depending on the runtime environment of the peer. This can be achieved by several mechanisms. The first solution consists in defining a set of peers which are responsible for generating versions of the services. Each version matches one supported runtime environment. The second solution relies on the use of a cross-compilation environment. The component generator creates services which are runtime independent at the source level. This allows the component generator to create multiple binary applications from a single service definition.

The first solution is much more representative of the idea of distributed middleware. However, having peers used to generate applications for a group of peers is really difficult to use in practice due to the lack of a common set of libraries on the many available runtime environments. It also introduces more issues related to the certification of the binary application. This approach would require a mechanism proving that the binary application is genuine. The second approach is less difficult in practice and is often used to generate applications for high performance computers. In this context, a dedicated computer is commonly used on high performance systems to prevent the consumption of CPU cycles just for the compilation of applications. In the context of YML, the second solution is the one which is most easily realized, as it only needs works to settle the cross-compiling environment for the various operating systems/hardware of the targeted peers. Lastly, a novel approach in that regard consists in using virtualization/emulation in order to deploy an homogeneous environment. This approach is promising for future peer to peer and grid environments. It also enhances significantly the control given to the owner of a resource as well as the security of the peer and the application.

7.2.1.3 Volatility

Peers in large scale distributed systems are volatile. It means that peers can enter or leave the middleware at anytime. In YML, we consider only the failure of the peers involved in the computation of workflow services. As the behavior of middleware differs significantly with regard to fault tolerance, YML workflow contains many restarting points. Indeed, each service can be executed over and over until its execution is successful. The YML kernel and its scheduler component assume that the service execution is completed when it receives a notification. The fault tolerance mechanism can then be implemented either in the middleware if present or at the the back-end level by simply rescheduling the job to a new peer. However this solution does not work perfectly for all environments. Indeed a system like OmniRPC does not support the loss of a peer involved in the system.

The introduction of the back-end manager creates another level of fault tolerance to the system. Indeed, it allows each back-end to be disconnected completely from the system preventing permanently or temporally YML to access a complete middleware. Pending service executions are dispatched to the remaining back-ends if any or queued until new back-ends contact the back-end manager.

Security, Heterogeneity and volatility constitute the three aspects that must be discussed concerning peers involved in the execution of YML workflows. The next issues concern middleware. Although middleware are converging, they are still far from being interoperable. Most of them are designed to solve the problem of a particular community of users. These differences lead to many solutions dedicated to a particular usage pattern. Most of the issues related to the use of large scale distributed middleware are associated to this problem.

7.2.1.4 Multi-middleware

YML is a client of middleware. In that respect, it does not require any modification of middleware. It is thus simple to add new middleware. The support for a new middleware only requires the creation of a worker and the ability to request the execution of the YML worker on a peer. This is the case of most known middleware. With the introduction of libraries such as DRMAA, the writing of new back-ends is simplified significantly and grants access to many environments at once. One aspect of the integration that could be improved is the data repository component which is discussed later in this section.

7.2.1.5 Middleware interoperability

YML provides interoperability among middleware at several levels. First the services composing a workflow are shared among middleware. The YML worker allows this by providing a consistent environment for service execution on all supported middleware. Secondly, the application once compiled can be executed multiple times on different middleware just by changing the configuration of YML. Finally, the execution of a workflow process can use peers provided by multiple middleware at the same time.

7.2.1.6 Data exchanges

The data repository component is used for all communication in YML. It is used to communicate between the YML scheduler executing an application and the peer of the middleware. At the time being, we assume that at least one peer can be contacted by any other peers of the system. This peer is used to host the data repository component used in all data exchanges between the YML scheduler and the service executed on remote peers. Currently, this part of the system is not distributed. It is thus at the same time a bottleneck and a limitation to the integration within targeted systems. Indeed, middleware can define a private network which is used to exchange data between peers. However, peers might only be able to communicate with each other through mechanisms provided by the middleware. With such middleware, YML is currently not able to do any communication. A solution to this problem is to use a mechanism similar to the back-end manager for data repositories. This aspect is required to improve the performance of YML and the performance related to data management. This will also allow us to implement techniques such as data replication or peer to peer data transfers.

The middleware integration and interoperability are two solutions we adopted to ease significantly the use of several large scale distributed systems. YML enables the definition of an application to be made middleware independent not only through back-end components, but also thanks to a workflow description language used to define applications.

7.2.2 Language and toolset

In order to ease the exploitation of large scale distributed systems for end-users, a uniform runtime environment is not enough. In the remainder of this section, we discuss the aspects related to the YML language used to describe applications as well as the tools to support end-users. All these aspects are related to the higher layers of YML.

YML is organized around a workflow process definition language. It is used to express the control flow of the application. The use of a workflow oriented environment introduces several constraints to the end-users. First, the user must define its application as a collection of independent services. Independent services are connected to each other using a workflow process. Workflow processes can then be combined in order to create more complex applications.

7.2.2.1 Services

Nowadays practices in software development and engineering promote software architecture which leads to a decomposition of an application into independent components. Moreover, they promote a separation between the component interface and its realization in order to decrease the coupling between two components of an application on the first hand and to allow an application to change the behavior of one or several of its components independently of their implementation. This methodology has been used during the design of YML. The workflow environment also enforces this programming practice during the development of end-users applications.

The notion of abstract/concrete services highlights that an abstract service with a given interface can lead to several concrete implementations. The execution catalog of YML provides a mechanism to select at runtime the best available realization of an abstract service. We did not highlight this aspect in the previous chapters because we still need to develop some advance policy to match an abstract service to concrete services. Nevertheless, it is already possible to adapt the execution catalog component implementation provided with YML to take into account several selection criteria such as peer architectures[6], data size, complexity of the implementation, patterns availability, etc. The introduction of new realization of a service does not lead to a redefinition of the application workflow process or of other components.

The separation between abstract/concrete services allows services to be written in different programming languages. The language used to implement services is important. Indeed, one of the goals of YML is to allow the reuse of existing libraries and especially numerical ones. These libraries are mostly written using an imperative language such as C or Fortran. They need to be compiled into an architecture dependent representation called a binary application. This is one of the major problems of these languages in the context of heterogeneous environments. This has been discussed in the previous section. It introduces a complexity in the environment in order to handle all systems. The writing of services is also more difficult because of the differences which exist between systems. However, it is also an interesting property of large scale distributed systems. Indeed, the heterogeneity of floating point computing units often benefits to numerical applications and especially hybrid methods.

The notion of abstract/concrete services eases significantly the definition of concrete services. Indeed, abstract service definitions are used to generate a skeleton of the concrete service. The user does not have to manage the serialization of the data and more generally all input/output operations are hidden to the user. The communication between services

[6]Example of criteria: availability of graphic processing unit, amount of memory, number of processors

involved in a workflow process is transparent to the user.

7.2.2.2 Processes description language

In 2.2.2, we have presented the criteria used to describe the parallel programming models. We discussed both the data parallel model and the task parallel model. These models can be compared based on whether they implicitly or explicitly allow the user to express concurrency, synchronization, data distribution and communication. YML manages data distribution and communication for the user. Concurrency and synchronization must be expressed explicitly. The first two criteria are strongly coupled with the control flow while the last two are mostly associated to data flow of an application. YML detects the data flow based on the analysis of the control flow and thus hides this aspect to the end-user.

Many workflow process definition languages exist. Most often, they make an extensive use of XML. The advantage of this language is that it simplifies the communication between applications. However, it is really verbose and is not adapted to the creation of large documents manually. YML chooses a more compact syntax similar to C or pascal in order to allow the creation of workflow processes directly.

This language is dedicated to regular applications which make heavy use of multi-dimensional arrays. This is well adapted for numerical applications which often rely on the manipulation of large matrices. Large matrices are often decomposed in blocks. The decomposition of a matrix in blocks determines the granularity of the application. The workflow process of the applications as we will see in the next chapter only depends on the decomposition which can be part of the workflow process or executed before as a pre-processing step. The workflow process adapts itself when the decomposition of the matrix changes. Moreover, the workflow process can be shared between applications whose control flows are identical. It promotes generic programming. It behaves similarly to an application template. The concrete services used determine what is really computed by the workflow process.

In order to efficiently exploit large scale distributed systems, two aspects are really important. The environment/language should provide mechanisms to allow out of core computations on the one hand and data persistence on the other hand. The first aspect is fully supported in YML and is transparent to the user. However, data persistence is not yet possible because of the requirements we specified on middleware. Indeed, in order to be able to realize efficiently data persistence, we need to express affinity between work and data. However, in many middleware we cannot specify which is the resource to use to execute a work. Thus data persistence is meaningless. Despite this constraint, on going work to increase the performance of YML consists in adding some caching mechanism to the worker. This will be used to provide an automatic data persistence mechanism.

7.2.2.3 Extensibility and future evolution

YML was specified with numerical application as the main application domain. However, it is not limited to this domain. The language can be extended by integrating existing libraries, which often leads to the definition of user defined data types. It is also possible to register new functions to create the workflow processes. These functions are used during the compilation of a workflow process and are organized in packages.

The integration of existing libraries often requires the definition of user data types. For example, the LAKe library introduces two abstract data types for representing dense and sparse matrices. These data types consist mainly in wrappers around the classes provided by the LAKe library. The code generator component requires that all type definitions provide two functions used for the conversion from and to a byte stream. The integration of LAKe also introduces a set of abstract and concrete services which can be used in many workflow processes.

The image application presented in 8.2.2 used images stored in png. These images can be read and written using the open source library libpng.The aim of this library is not image processing. It focuses on both the reading and writing of images stored according to the png specification. In order to allow the creation of image processing applications, we defined a data type for image manipulation. An image basically consists in a two dimensional array of pixels. The libpng library is used for the conversion of image data into a stream of bytes. However, the libpng library does not lead to the definition of any abstract or concrete services. These two examples illustrate the extension of YML by the integration of existing libraries. It leads to the definition of new data types which can be used in the definition of workflow processes and services. It also often leads to the definition of a collection of abstract, graph and concrete services.

The last mechanism used to adapt YML to new application domains is function package. A function package contains a set of functions which can be used by the YML compiler during the compilation of the workflow process definition. These functions are grouped in packages which are loaded dynamically when needed. These functions can then be used to compute indices or constant values involved in the service calls. YML defines two default function packages which contain basic operators supported by the language and some mathematical functions such as log, power or square.

From the beginning of the project, we designed YML with the integration of existing software in mind. We were also interested in being able to extend YML with new data types. In order to do this we decided not to depend on a mechanism based on XML such as the one used in XML-RPC, or web services. XML base communication was not chosen because of the verbosity of the language. However, it might be used in future and this area needs to be investigated. A more realistic approach consists in providing an XML description of the data format which can be used by code generators to interpret the content of a file. Jointly to this approach, data will probably be extended with meta data in the future in order to be able to use data properties in the workflow process definition as well as the dynamic selection of concrete services during the execution of workflow processes.

In order to enlarge the scope of applications, the YML workflow process definition language can be extended without modification of the other components of YML. A possible extension consists in creating other parallel loops which do not generate any implicit synchronization at the end of their execution. Another possible evolution is the introduction of exception mechanisms. The processing of exceptions can be translated into rules as well. However, in order to support exceptions in a way similar to other languages, we need to slightly adapt the scheduler. The exception mechanism implies a rollback mechanism which is not yet supported by the scheduler component. The services are already able to generate exceptions. They are treated mostly like errors by the scheduler.

7.3 Conclusion

YML provides a solution to most of the problems presented in the motivations. In order to ease the development of applications for our targeted runtime environment, we designed a workflow environment. This environment hides the complexity of writing a parallel application by two main mechanisms. Firstly, YML provides an abstraction of the runtime environment which allows a YML workflow process to be deployed and executed on different middleware. It is also possible to spread the execution of one workflow process on multiple middleware at the same time. Secondly, YML defines an application as a collection of abstract/concrete services. These services interactions are defined by a simple language designed to express parallelism and synchronization. Many aspects are hidden to the end-user and handled automatically by the workflow management software. The ease of development is possible thanks to an increased level of abstraction.

Being a high level approach to the development of parallel and distributed applications, YML hides aspects of a parallel application such as volatility, heterogeneity, data management and communication. Heterogeneity needs further improvements in order to become one of the main assets of YML. Indeed, in a large scale distributed environment, heterogeneity is at the same time a complexity for the user and a chance. Environments which are able to benefit from the diversity of resources are still not well spread and do not solve all the problems related to the development of workflow process services. The separation between abstract and concrete services is a first step. In order to improve the capabilities of YML to deal with heterogeneity of peers, the next step is the dynamic selection of concrete services based on peers characteristics.

Chapter 8

Applications

8.1 Introduction

In order to evaluate practically YML, we present and discuss four applications issued from three domains. The first application sorts large data sets divided into chunks of data. One of the characteristics of the method used is the limited memory usage. Indeed, the application can sort arbitrary amounts of data by increasing the number of blocks. In order to work in a distributed environment, we needed an algorithm to manipulate a fixed quantity of memory easily evaluated before starting the application. Algorithms such as the quick sort for example do not respect this property. We also wanted an algorithm which uses resources in the most uniform way possible. The amount of parallel jobs, number of peers, memory, and network usage is constant during the sorting stage.

Then we selected the synthesis of images from a three dimensional scene description. This application produces large images in a distributed fashion and applies post-processing filtering operations on the results. The application is built upon a ray-tracer, named RayShade, used to create a wall of images. Ray tracing techniques most often rely on Monte Carlo sampling of all pixels composing the resulting image. All samples are based on the casting of rays which correspond to the path followed by the light. The ray tracer is coupled with a filtering operation which can be used to transform the wall of images using effects such as gaussian-blur or edge detection.

The initial goal of YML is to ease the creation of numerical applications and especially linear algebra ones on large scale distributed systems. The last applications presented are used in the resolution of linear algebra problems such as linear systems and eigenproblems. We first present the matrix vector product. This operation is used in many numerical applications and thus is particularly significant in our context. It is also the basis of most projection methods used to create a Krylov subspace.

Lastly we discuss an hybrid method called MERAM which solves eigenproblems for large sparse non-Hermitian matrices. This application is often used in applications including numerical simulations, Google search engine, and many other fields. MERAM is constituted of several asynchronous co-methods called ERAM which exchange intermediate results to decrease the number of iterations required to find a few number of eigenelements.

All applications are presented following the same outline. Each application description begins with a short description. It is followed by a list of the parameters and services. The workflow process is then described either as a source code or using a visual representation of it.

8.2 Non numerical applications

8.2.1 Distributed sort algorithm

8.2.1.1 Overview

This application presents a distributed sorting of a large dataset. The dataset is composed of integers stored in blocks of a fixed size. The application relies mostly on an operation we called *merge*. This operation takes two blocks of a_i and a_j and creates two new blocks a'_i and a'_j of the same sizes. The elements of block a'_i are all smaller than elements of block a'_j. All outputs blocks are also sorted. The sort algorithm is described by algorithm 1.

Algorithme 1 : Distributed Sort

```
INPUTS: $\{a_1, a_2, \dots, a_n\}$ $N$ unsorted blocks of $K$ integer values.
OUTPUTS: $\{a_1, a_2, \dots, a_n\}$ $N$ sorted blocks of $K$ integer values.
for $i$ in 1, $N$ in parallel do
    $DSortBlock$(INPUTS: $a_i$ OUTPUTS: $a_i$)
for $iter$ in 1, $N-1$ do
    $inc \leftarrow 1 + iter \bmod 2$
    for i in 0, $N/2-1$ in parallel do
        $block1 \leftarrow 1 + (inc + 2 * i) \bmod N$
        $block2 \leftarrow 1 + (inc + 2 * i + 1) \bmod N$
        $min \leftarrow min(block1, block2)$
        $max \leftarrow max(block1, block2)$
        $DSortMerge$(INPUTS: $a_{min}$, $a_{max}$ OUTPUTS: $a_{min}$, $a_{max}$)
```

This algorithm is not optimal nor is it the most efficient however it is interesting in the context of large scale distributed systems. Indeed, the number of operations of the merging and the memory consumption is identical for any couple of blocks. It is easy to determine a block size which is adequate for the peers composing the runtime environment. This is not the case with more efficient algorithms such as quick sort. Moreover, the number of parallel tasks is also identical during all steps of the sort. This algorithm is thus interesting for the environment we trigger.

8.2.1.2 Parameters

The application defines the two parameters:

- **blockSize** defines the number of integers contained in a each block.
- **blockCount** defines the number of blocks of **blockSize** integers. The size of the collection to sort is obtained by multiplying **blockSize** by **blockCount**. This parameter also determines the number of parallel operations which compose its iteration of the algorithm.

8.2.1.3 Services

The application defines three services:

- **Generator**: Create a block of the collection composed of random integers.
- **Sort**: Sort the elements contained in a single block. This is only needed after the generation of a block because the merging operation expects input blocks to be sorted.
- **Merge**: Merge two blocks as described before. The two blocks are merged so that the smallest elements are in the first resulting block and the largest elements fit in the second resulting block. The number remains sorted in both blocks.

8.2.1.4 Workflow process

Figure 8.1 is a visual representation of the workflow of the application using $N = 4$ blocks. The corresponding workflow process definition is presented in listing 8.1. The workflow process contains two different parallel sections. The first section consists in generating the collection of integers. The second parallel section realizes the sorting algorithm.

```
<?xml version="1.0" ?>
<application name="DSORT" >
    <description>
        This graph is used to show the sorting process at runtime and
            keep an history of blocks.
        It takes a huge space on disc for large block. Use it with
            care

    </description>
<params>
    <param name="blockSize" type="integer" mode="in" description="Set
        ␣the␣size␣pf␣blocks" />
</params>
<graph>
# Change this var to a power of two
# It corresponds to the number of blocks of data to handle
# The graph will be made of blockCount - 1 parallel front of
# n / 2 block operations
blockCount := 32;
par
  par (i := 1; blockCount)
  do
    compute DSortGenerator(block[i][0], blockSize, 256.0);
    compute DSortSort(block[i][0]);
    notify(evtBlock[i][0]);
  enddo
//
  par (i := 1; blockCount - 1)
  do
    inc := 1 + (i % 2);
    par (j := 0 ; blockCount / 2 - 1)
    do
      block1 := ( inc + 2*j ) % blockCount;
      block2 := ( inc + 2*j + 1 ) % blockCount;
      if (block1 gt block2)
      then
        finalBlock1 := 1 + block2;
        finalBlock2 := 1 + block1;
      else
        finalBlock1 := 1 + block1;
        finalBlock2 := 1 + block2;
      endif
      wait(evtBlock[finalBlock1][i - 1] and evtBlock[finalBlock2][i -
          1]);
      compute DSortMerge(block[finalBlock1][i],
                         block[finalBlock2][i],
                         block[finalBlock1][i - 1],
                         block[finalBlock2][i - 1]);
      notify(evtBlock[finalBlock1][i], evtBlock[finalBlock2][i]);
    enddo
  enddo
endpar
</graph>
</application>
```

Listing 8.1: Distributed sort workflow process definition

Additional resource used in this application is presented in appendix A.1.1.

8.2.2 Image synthesis and post-filtering

8.2.2.1 Overview

Rendering is the process of generating an image from a model, by means of a computer program. The model is a description of three dimensional objects often named the scene. The scene contains geometry, viewpoint, texture, lighting, and shading information. For movie animations, several images (frames) must be rendered, and stitched together in a program capable of making an animation of this sort. On the inside, a renderer is based on a selective mixture of disciplines related to light physics, visual perception and

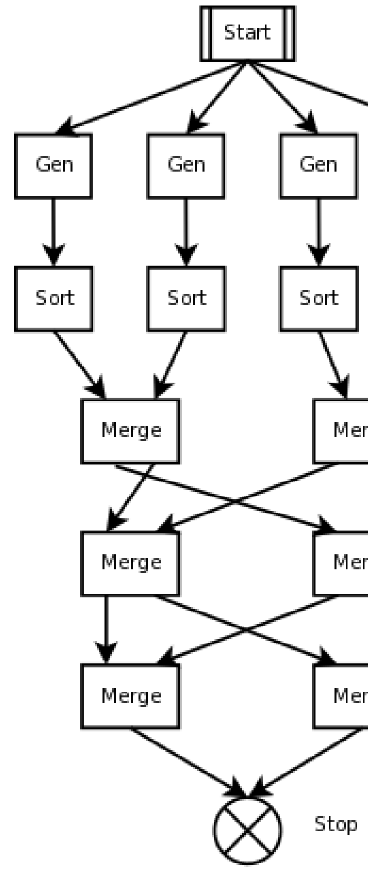


Figure 8.1: Distributed Sort: Application graph

mathematics.

The rendering of a scene can be achieved by several techniques such as scanline rendering and rasterisation, ray casting, radiosity and ray tracing. Ray-tracing is an extension of both scanline and ray casting. It is almost always a Monte Carlo technique, based on averaging a number of randomly generated samples from a model. In this case, the samples are imaginary rays of light intersecting the viewpoint from the objects in the scene. It is primarily beneficial where complex and accurate rendering of shadows, refraction or reflection are issues.

In order to produce a two dimensional image from a scene, the rendering consists in computing the color of the light which hits the camera. For each pixel in the resulting image, a ray-tracer tries to compute as accurately as possible the color of the light. This process is time consuming and can be really complex as it consists in casting many rays for each pixel. This is also called physical based rendering. Each pixel can be processed independently, which leads to trivial parallelism. Indeed, it is possible to distribute the rendering of a scene to several computers each working on distinct parts of the image. Most ray-tracers are able to render only a selected window of the complete image and many distributed rendering solutions rely on this. The distributed generation of an image or of the frame composing an animation has been used as a benchmark for several global computing or task farming solutions.

A post-processing stage applies a transformation to the pixels composing an image. Indeed an image is often represented as a two dimensional array of pixels. Each pixel is represented as a set of components which represent the amount of red, green, blue composing the color. We define a post-processing stage as an operation which applies to the totality of an image generated in a distributed way. However, we apply the post-processing without aggregating all windows composing an image. Examples of post-processing operations are regular image transformations such as re-sampling, blur, edge detection or image information extraction.

The post-processing of an image is an operation which transforms an image according to a filter. This can be used to enhance the quality of an image or to apply effects such as a wave, blur, edge detection effects. This kind of effect can be obtained by applying a filter to each pixel of an image. The general form of the execution of a filter on an image is described in algorithm 2.

Algorithme 2 : Image filtering algorithm

```
INPUTS: $I$ an image of size $w \times h$ pixels, $filter$ the filtering function.
OUTPUTS: $I'$ the resulting image of size $w$,$h$.
for $i \in 1, w$ in parallel do
    for $j \in 1, h$ in parallel do
        $I'[i][j] \leftarrow filter(I, i, j)$;
```

The definition of $filter$ varies depending of the filter used. The function $filter$ is applied to each pixel of the source image and generates one pixel of the resulting image. As a consequence, each pixel can be processed independently. However, most filters require the knowledge of the surrounding pixels. Several filters can be represented as a square matrix of coefficient of size $2n + 1$. The coefficient represents the contribution of the pixel associated to the coefficient to the new pixel value. For example, a gaussian blur effect is obtained by computing the average of the surrounding pixel values multiplied by the coefficient in the matrix. Figure 8.2 contains an example of a matrix 5×5 for a gaussian blur filter.

1	2	4	2	1
2	4	8	4	2
4	8	16	8	4
2	4	8	4	2
1	2	4	2	1

Figure 8.2: Gaussian blur filter

Our application consists in the distributed synthesis of an image and the execution of a sequence of filters. This without manipulating the whole image during the execution of the workflow. We currently use a gaussian blur filter for each step of the post-processing.

This application makes use of the rayshade library. This library which provides the ray-tracing facility is out of the scope of our traditional studies or area of expertise and we do not have a deep knowledge of its behavior. In that respect, the ray-tracer is considered as a black box. This is an example of integration of an external software for which we have no knowledge at all. This is different from the other three applications which can be considered from our point of view as white box integration.

8.2.2.2 Parameters

The application defines the following parameters:

- **Scene**: The most important parameter of the application is the scene. It is an archive which contains one or several files to be processed by the ray tracing service.
- **blockWidth/blockHeight**: They define the size of a block or window of the final image. Each block is generated by one instance of the ray-tracing component during the generation stage.
- **blockCount**: This parameter defines the number of blocks composing the final image. The complete image size is $blockCount \times blockWidth, blockCount \times blockHeight$ pixels.
- **filterStage**: It is used to specify the number of filtering stages to apply to the image. For our experiments, we simulate a pipeline of filtering operations.
- **filters**: One or several square matrices to apply at different stages of the post-filtering. A filter is represented by a square matrix which corresponds to coefficients to apply to color components of the image.

8.2.2.3 Services

The application defines two services:

- **rayshade**: ray-tracer service used to produce a two dimensional image. It takes a scene as input as well as the expected image size and the window to render and produce a two dimensional array corresponding to the window stored using the PNG image format.
- **filter**: apply a filter to a block and generate the transformed block. In order to apply a global transformation the filter also needs the neighborhood of the block being processed.

8.2.2.4 Workflow process

The workflow process is like previously composed of two stages. The first stage represented in the first and second parallel sections is responsible for the generation of the image based on the scene description. The image is composed of a square two-dimensional array of images also called windows. Each window is stored as a valid PNG image. During the generation process we also notify explicitly events for an additional border around the generated image. This is used to simplify the post-processing description which is given in the last parallel section. The post-processing is composed of *filterIteration* operations executed sequentially. However the parallelism expressed by the workflow allows a post-processing operation to start before the end of the previous one. This is possible due to the use of explicit synchronization mechanism based on the use of `notify` and `wait`.

```
<?xml version="1.0"?>
<application name="Image">
    <description>
        Image filtering application
    </description>
    <params>
        <param name="result"    type="PNGImage"   mode="out"
            collection="yes"/>
        <param name="scene"     type="ZIPArchive" mode="in"
            collection="no" />
        <param name="blockSize" type="integer"    mode="in"
            collection="no" />
        <param name="filter"    type="RealMatrix" mode="in"
            collection="no" />
    </params>
    <graph>
#########################################################################
# Change this to increase / decrease the number of blocks of the
# image
blockCount := 10;
# Number of filtering iterations
filterIteration := 4;
#########################################################################
# No change needed starting from here                                   #
#########################################################################
par
  # Prepare events for missing block at the border
  # It is better to generate all those events once for all
  par
    (k := 0 ; filterIteration)
  do
    # Generate border blocks events
    par
        par (i := 0 ; blockCount + 1)
        do
            # Border lines
            par
                notify(evtBlock[i][0][k]);
            //
                notify(evtBlock[i][blockCount + 1][k]);
            endpar
        enddo
    //
        par (j := 1 ; blockCount)
        do
            # Border Columns
            par
                notify(evtBlock[0][j][k]);
            //
                notify(evtBlock[blockCount + 1][j][k]);
            endpar
         enddo
    endpar
  enddo
//
  # Generate the initial image
  par (i:= 1; blockCount) (j:= 1; blockCount)
  do
    compute ImageRayshade(result[i][j][0], scene,
                          blockCount, blockCount,
                          blockSize, blockSize,
                          i - 1, j - 1);
      notify(evtBlock[i][j][0]);
  enddo
//
  # Start filtering stage
  par (k := 1; filterIteration)
      (i := 1; blockCount) (j:= 1; blockCount)
  do
    # wait for blocks around of (i:j)
    wait(evtBlock[i - 1][j - 1][k - 1]);
    wait(evtBlock[i - 1][j    ][k - 1]);
    wait(evtBlock[i - 1][j + 1][k - 1]);
    wait(evtBlock[i    ][j - 1][k - 1]);
    wait(evtBlock[i    ][j    ][k - 1]);
    wait(evtBlock[i    ][j + 1][k - 1]);
    wait(evtBlock[i + 1][j - 1][k - 1]);
    wait(evtBlock[i + 1][j    ][k - 1]);
    wait(evtBlock[i + 1][j + 1][k - 1]);
    compute ImageFilter(
        result[i    ][j    ][k    ], result[i    ][j    ][k - 1],
        result[i - 1][j    ][k - 1], result[i - 1][j - 1][k - 1],
        result[i    ][j - 1][k - 1], result[i + 1][j - 1][k - 1],
        result[i + 1][j    ][k - 1], result[i + 1][j + 1][k - 1],
        result[i    ][j + 1][k - 1], result[i - 1][j + 1][k - 1],
        filter);
    notify(evtBlock[i][j][k]);
  enddo
```

```
endpar
    </graph>
</application>
```

Listing 8.2: Image processing workflow process

Additional resources of this application are presented in appendix A.1.2.

8.3 Numerical applications

Linear algebra methods are used in many applicative contexts. Linear algebra applications often lead to the solving of either a linear system or an eigenvalues/vectors problem:

- solving of a linear system $Ax = b$ where b and x are two vectors of size n and A is a matrix of order $n \times n$.

$$\begin{array}{rcll} A & \in & \mathbb{R}^{n\times n} & (\text{or } \mathbb{C}^{n\times n}) \\ x & \in & \mathbb{R}^{n} & (\text{or } \mathbb{C}^{n}) \\ b & \in & \mathbb{R}^{n} & (\text{or } \mathbb{C}^{n}) \end{array}$$

- finding the eigenvalues(λ) and eigenvectors(u) of a matrix A which are solution of $Au = \lambda u$.

$$\begin{array}{rcll} A & \in & \mathbb{R}^{n\times n} & (\text{or } \mathbb{C}^{n\times n}) \\ u & \in & \mathbb{R}^{n} & (\text{or } \mathbb{C}^{n}) \\ \lambda & \in & \mathbb{R} & (\text{or } \mathbb{C}) \end{array}$$

We distinguish two kinds of resolution methods for these problems. Direct methods compute the solution in a fixed number of operations. However, iterative methods compute the solution in a potentially infinite number of operations. For each class of problem, direct and indirect methods exist. Both kinds have benefits and drawbacks depending on the characteristics of the problem to solve and of the runtime environment.

The matrix A which is the definition of the problem for the two major linear algebra applications can be stored using a large variety of representations. The simplest representation is dense matrices. In this storage scheme, all elements of the matrix are explicitly represented. This storage is not efficient for problems where the number of null elements is important. Many compressed schemes exist for representing matrices composed of a large number of null elements. Matrices compressed by removing null elements are known as sparse matrices.

Direct methods are most suited for linear algebra problems represented using dense matrices. They often transform matrices they operate on. Examples of those methods are Gauss-Jordan and block Gauss-Jordan[129] methods used to solve linear systems. They compute the solution by applying Gauss pivoting operations on the matrices A and A^{-1} which are built during the resolution. The effect of the Gauss elimination is the creation of new not null elements in A. This algorithm is thus inadequate for sparse matrices and really large problems. An evaluation of the block Gauss-Jordan method using YML is given in [101].

Iterative methods are on the opposite well adapted for computation on sparse matrices. They do not transform the initial matrix representing the problem. In order to produce a result, iterative methods are stopped once the solution computed meets a user defined accuracy criterion or when a limit on the number of operations executed is reached. Figure 8.3 represents the control flow of all iterative methods.

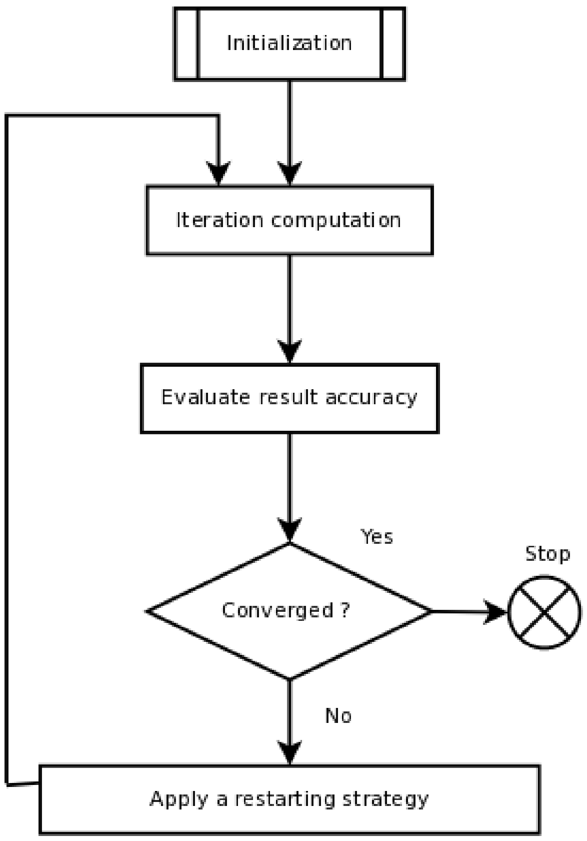


Figure 8.3: Iterative method control flow

Each iterative method can be decomposed in four computing steps. The *initialization* creates the initial guess of the methods. Its goal is to create a data set I_0 used to feed the first iteration of the method. The *Iteration computation* processes the data provided by the initialization or the last iteration and creates the solution R_i of the current iteration. The third step *evaluates the accuracy* of R_i. This information is compared to the user defined tolerance. If the accuracy is greater than the tolerance then the method needs to be restarted by applying a *restarting strategy*. This step creates I_i which is going to be used as the initial guess for the next iteration.

The eigenvalue problem is an old and well known problem. As it is used in numerous scientific and industrial applications, it is studied a lot [169]. Most applications are not interested in computing all the solutions to this problem. Only the eigenvalues of greater or smaller modules are generally of interest. The rest of this section introduces some of the methods used to extract these eigenvalues.

One of the most commonly used methods is QR. This method computes the solution of all the eigenvalues and eigenvectors of a matrix $A \in \mathbb{R}^{n \times n}$. The complexity of this method is $O(n^3)$ if the matrix is symmetric or $O(n^4)$ for asymmetric matrices. It is not adapted to large problems. Projection methods such as Arnoldi are used when the size of A increases.

A projection method translates a big problem into a small one which contains the eigenvalues of A. The projected problem contains the solution of the initial one. The projection method constructs two information: a basis of the subspace and a Hessemberg matrix. A derivative of the power method generates a set of vectors which together define the basis of a Krylov subspace. Krylov suggested the use of the vectors $x, Ax, A^2x, \ldots, A^nx$ generated by the power method as the foundation of all Krylov subspaces. The Lanczos and Arnoldi projection enhance the stability of the Krylov basis by ensuring orthogonality of the basis during its construction. Lanczos projection is used for symmetric matrices while Arnoldi projection is used for non-Hermitian ones.

The aim of an hybrid method is to speed up the convergence time of an iterative method. The idea is to couple synchronously or asynchronously several co-methods in order to decrease the number of iterations required to compute the solution. A special form of hybrid methods is called multi methods. Figure 8.4 represents an hybrid method. The figure highlights the collaboration pattern used between two methods. Other collaboration patterns exist when co-methods are used to speed-up only parts of the iterative methods. Most of the time co-methods exchange their results at the end of each iteration and this information is taken into account during the execution of the restarting strategy.

A multi method is a hybrid method composed of several instances of the same iterative method coupled to work on the same problem. Each instance is given a different set of initial parameters. At the end of each iteration, each iterative process exchanges its results with the other co-methods. The restarting strategy uses all the information available at that time to compute the initial guess for the next iteration.

Hybrid methods and especially multi methods are interesting in the context of distributed environment. One of the main interests of these methods is asynchronicity. Indeed, if the results of the other methods are not available, it is still possible to continue the computation. At the same time asynchronus methods are also interesting because they provide built-in fault tolerance. As long as one of the co-methods is still running, the application will generate a result. Lastly, these methods can improve the convergence speed thanks to the heterogeneity of processors. Different architectures or processors can lead to different round errors due to differences in floating point operations hardware. Each method can use the result obtained from the others to decrease the effect of round errors.

In the remainder of this section we present two applications implemented for YML: The matrix vector product is first presented and then we present a realization of the MERAM multi-method for the solving of eigenproblems.

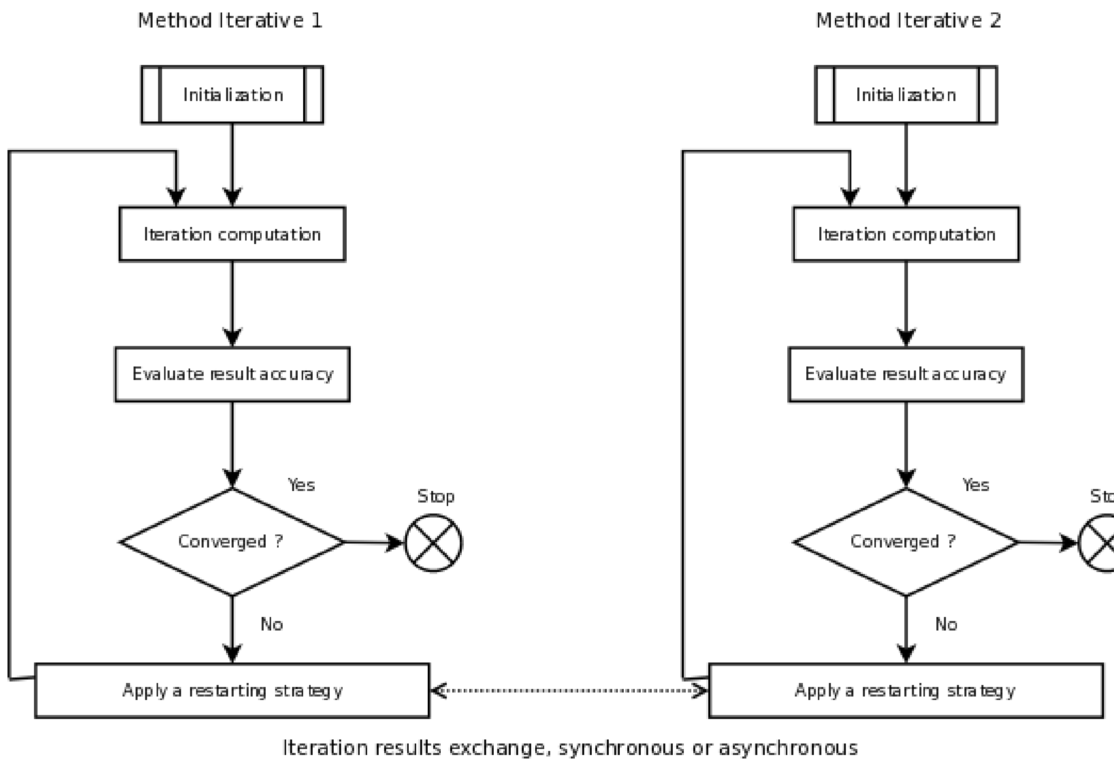


Figure 8.4: Hybrid method control flow

8.3.1 Matrix vector product

8.3.1.1 Overview

The matrix vector product is a basic operation in all linear algebra applications. More specifically, this application computes $y = Ax + x$. It is used almost everywhere and especially during the projection stage of iterative methods and the construction of Krylov subspaces. The study of this simple operation is the first step toward a distributed Arnoldi projection and a scalable eigensolver. A first evaluation of this method with YML is given in [49]. This evaluation describes the first prototype of YML and the matrix vector product of dense matrices executed on top of the XtremWeb middleware.

This application triggers really large matrices of a billion-order. In this context, it is not possible to delegate the work to a single processor and the matrix must be divided in blocks. Many decompositions exist and the best one depends on the matrix itself. However, in order to be able to execute this operation on any matrix, we select a decomposition which does not take into account the specificity of the matrix. We choose the same decomposition as for dense matrix vector product presented in [49]. The decomposition consists in splitting the whole matrix in $blockCount \times blockCount$ square blocks of size $blockSize$. Each block is a matrix itself. The input vector x and result vector y involved in the product are decomposed in $blockCount$ blocks of $blockSize$ elements.

This application depends on the LAKe library and relies mainly on the matrix multiplication operator service of LAKe. This class is especially designed to provide matrix vector product. It is designed to be specialized for sparse matrices and exposes a read-only interface for the matrix operand.

In order to study the matrix vector product on a problem of an arbitrary size, we emulate a distributed matrix generator by replicating a single matrix block. We also generate an arbitrary vector x. However, the parameter chosen for the vector and the matrices has no impact on the performance and involves the same complexities in operations and memory consumption as a real tuple of matrix A and vector x.

8.3.1.2 Parameters

The matrix vector product application defines the following parameters:

- A represents the matrix. The matrix is decomposed in blocks. A block of the matrix is noted $A[i][j]$ where i is the row index and j is the column index. Each block of the matrix is of size $blockSize \times blockSize$. The matrix A is composed of $blockCount \times blockCount$ blocks.

- x represents the input vector. It is decomposed in $blockCount$ blocks of size $blockSize$. A block is noted $x[i]$.

8.3.1.3 Services

The application depends on the following services which are part of the integration of the LAKe library with YML.

- **MatrixSet**: This service is used to create an arbitrary dense matrix or dense vector. It is used for the creation of the input vector x.

- **Product**: This service does the operation $y = y + Ax$. This operation is described in the section below.

8.3.1.4 Workflow process

The workflow process of the matrix vector product application consists in two stages. The first stage generates an arbitrary vector x and initializes the vector y with a copy of x. This is done in parallel for each block of both vectors (1). The second stage is composed of a set of **product** services (2, 3, 4). The rows of the matrix are processed in parallel. However, within each row, blocks are processed sequentially, in order to prevent a reduction operation at the end of the computation of each block. A reduction would require a single service to gather the results of all rows and require a huge storage capacity on the peer responsible for the reduction of a row. The sequential processing of a row reduces one step at a time. Figure 8.5 describes the matrix vector product application.

Additional resources for this application are described in appendix A.2.1 and A.3.

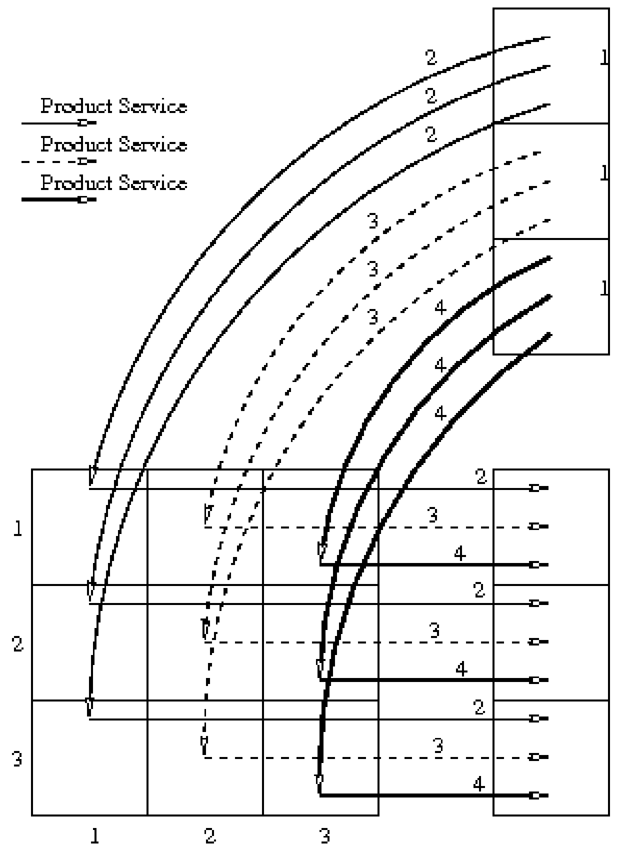


Figure 8.5: Matrix vector product: decomposition

8.3.2 Multiply explicitly restarted Arnoldi method

8.3.2.1 Overview

The explicitly restarted Arnoldi method (ERAM) is an iterative method which computes a few eigenvalues and eigenvectors of large sparse non-Hermitian matrices. It allows to compute an approximation of the solution of the large matrix A in a very much smaller m-dimensional subspace called Krylov subspace. If the accuracy of this approximation is not satisfactory, the process restarts using another Krylov subspace obtained with the information provided by the previous one. ERAM makes use of the Arnoldi projection to create a Krylov subspace K of base B. This projection is the generalization of the Lanczos method dedicated to the symetric case. A solver for the eigenproblem based on the Lanczos method has been adapted to YML in [37].

The eigenvalues and eigenvectors are then approximated by those of a matrix H representing A in the subspace K. The eigenelements of H can be computed using a classical QR solver. Algorithm 3 presents the ERAM method. I is the restarting vector initialized with an arbitrary vector. This last can be composed of randomly computed values, a vector of the identity matrix, a vector composed of 1, etc. *residuals* is an array of real values consisting in the residual norms associated to approximated eigenelements. If any value in that array is greater than the *tol*, the method needs to be restarted for another

Algorithme 3 : Explicitly Restarted Arnoldi Method

```
Data : $A$: The matrix of order $n$
Data : $m$: Subspace size
Data : $tol$: The tolerance expected by the user
Data : $maxIter$: The maximum number of iteration
Result : $V$: A set of $r$ eigenvectors
Result : $v$: A set of $r$ eigenvalues
begin
    $I \longleftarrow$ CreateInitialVector($n$);
    for $it \leftarrow 1$ to $maxIter$ do
        $I \longleftarrow$ Normalize($I$);
        $H$, $B \longleftarrow$ ArnoldiProjection($A$, $I$, $m$);
        $V$, $v$, $residuals \longleftarrow$ QRSolver($H$, $B$, $r$);
        if $residuals < tol$ then return $V$, $v$;
        else $I \longleftarrow$ ExplicitRestart($V$, $v$);
    end
end
```

iteration. The explicit restart sub program does a linear combination of the approximated eigenvectors, or Ritz vectors, in order to produce the initial vector of the next iteration.

The multiply explicitly restarted Arnoldi method (MERAM) is based on k instances of ERAM. The instances of ERAM work on the same problem (A, r, tol) but they are initialized with different subspace size $m_i, i \in [1, k]$. They can also use different initial vector generators and restarting strategies. The results R_i of the current iteration are produced after the execution of the QR solver sub program. They are sent to the other co-methods ERAM(j) with $j \neq i$ and $j \in [1, k]$. In the mean time ERAM(i) receives the results from the other ERAM(j). The best results are selected based on their accuracy. This step is later called *reduction*. They are then sent to the restarting strategy for each ERAM independently.

8.3.2.2 Parameters

In this section we present the parameters of the MERAM application defined by YML:

- A is the matrix of the problem to solve. The matrix is of size $n \times n$ and contains nnz non null elements. It is stored using a sparse matrix format called CSC also known as RUA. For our experiments we select matrices from the families AF, BFW, PDE and RAN of the MatrixMarket online collection.

- r represents the number of eigenelements which are expected by the user. The method focuses on finding the r eigenelements of greater modules. When applied to non Hermitian matrices, eigenelements are always complex values.

- tol is the convergence criterion used to stop the method. The application stops once the residuals which represent the estimated errors of the solution are smaller than tol.

- $m[i]$ represents the size of the subspace for the projections of the ERAM process i. $m[i]$ is a value in the range $r+1$ to n. The smaller it is the faster the iteration results are obtained. However if m is too small, the subspace might not contain the eigenelements and the method will not converge. This parameter has a critical impact on the number of iterations needed to converge. The best value for this parameter can not be calculated mathematically without solving the problem itself. We are thus limited to an arbitrary choice.

- $I[i]$ is the initial guess used at application startup for the ERAM process i. $I[i]$ is a vector used to initiate the projection stage. Several methods exist to initiate the projection such as arbitrary vector, unit vector, and identity matrix column vectors.

- *Reduction* is the strategy used to select the best result to be used to restart the method. It is a reduction operator applied to all available results at a time. We have used two strategies: the first strategy makes a selection among the best results available taking each eigenelement independently, while the second strategy groups all results of a method together and selects the best group of eigenelements.

- *Restart* is the strategy used to create a new initial guess for the next iteration. All the strategies we tested rely on a linear combination of the ritz vector and eigenvalues obtained during the last iteration and selected by the reduction strategy. Several criteria are available. We mostly experimented the simplest strategy, consisting in combining all vectors using the same contribution.

8.3.2.3 Services

The implementation of this application relies on the LAKe library for all its services. The applications define the following services:

- **Start**: Create an initial vector $I[i]$ used for the first iteration of each ERAM process. The vector is also normalized.

- **Arnoldi**: Compute the Arnoldi projection. This service takes as inputs the matrix A, the size of a subspace $m[i]$, and an initial vector $I[i]$. It computes the basis of the Krilov subspace $B[i]$, a dense matrix of size n, m and an Hessemberg matrix of size $m+1, m$ which represents the subspace itself. This projection is currently handled by a single service on a single peer. In order to allow our application to solve larger problems, we need to distribute this service in the future.

- **Solver**: Solve the eigenproblem in the subspace. It takes the basis and the Hessemberg matrices produced by the **Arnoldi** service and computes n eigenelements. If the accuracy of the solution matches the tolerance value *tol* given by the user, the **Solver** service generates an exception which stops the execution of the workflow.

- **Frobenius**: This service computes the frobenius norm of the matrix A this value is used during the evaluation of the accuracy of the solution obtained by the current iteration by the **Solver** service.

- **Reduction**: This service is involved during the restarting of each ERAM process. The restarting of the method is composed of two operations. The first operation consists in selecting the best restarting elements among the information available. This is the goal of this service. Several strategies to select the information used to select the restarting elements are used. It is this service which is responsible for the interaction of an ERAM process with the other co-method of the application.

- **Restart**: This service is involved in the second step of the restarting of an ERAM process. It constructs a new initial guess to be used in the next iteration of the process. It is basically a linear combination based of the Ritz vector produced by the **Solver** service. Several strategies can be chosen for the coefficient used for the linear combination.

8.3.2.4 Workflow

Implementing MERAM for YML scientific workflow rests on a set of services implemented by using the libraries such as LAKe. Figure 8.6 presents the graph of the application for four processes. The graph depends on two parameters: the maximum number of iterations and the number of ERAM processes. Data flow is represented using dot lines. Control flow uses solid lines. Services are boxed with solid lines. Each ERAM process is executed concurrently with the others. Data communication is managed by the workflow environment. The application stops as soon as the execution reaches the *stop* node of the graph. This happens when all methods have finished their execution or when a solution is found by an instance of the QR Solver service.

An important aspect of the solution relies on a shared storage. At the end of the QR solver service, results are sent to a storage service which gathers results from all methods. This shared storage is split in k independent slots which can be write and read concurrently. Each ERAM process is the owner and sole writer of its slot. There is no need for synchronization mechanism. The shared storage always contains the latest produced values. The reduction is executed locally by all ERAM processes. Such shared storage can be implemented in several ways in a workflow environment. Common solutions include state-full services such as an FTP server, distributed data repository services or simply the application storage managed by the workflow scheduling tool to store application run-time data.

The Arnoldi service can be implemented using three policies. It can be implemented as a sequential service and as a parallel service using LAKe or as a parallel sub-graph. The current implementation uses the sequential service approach due to limitation on the middleware we are currently using for our experiments. The first two policies are provided by LAKe. The instantiation of the parallel version depends solely on the resource allocated to the service. The third approach allows to handle larger problems on commodity hardware.

The implementation described above makes only a few assumptions on the underlying run-time environment. This approach rests on a strict separation between data flow, control flow and computation code. This approach is due to the use of workflow environments to express arbitrary levels of parallelism. The solution itself can be used in libraries

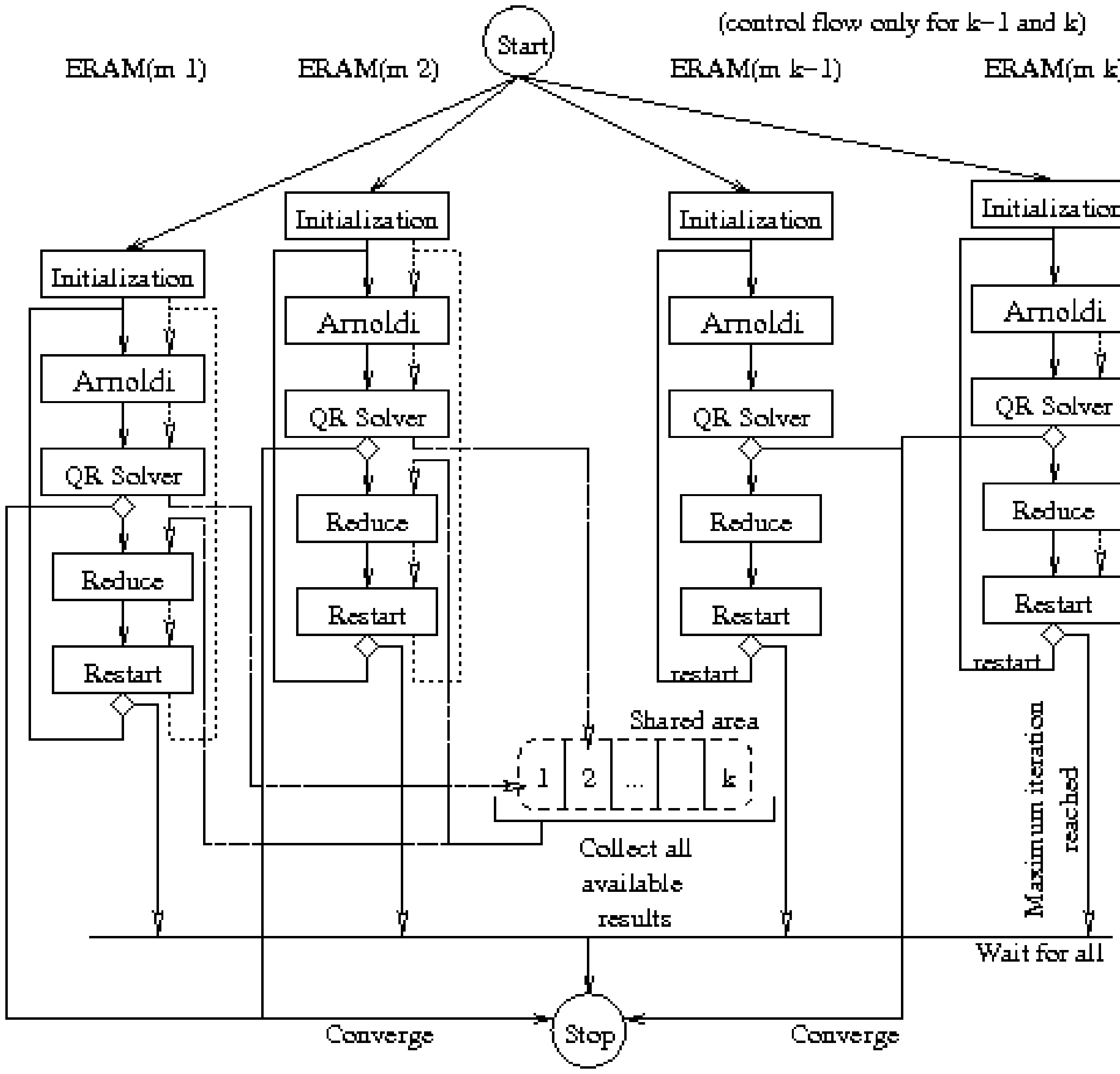


Figure 8.6: MERAM application graph

such as LAKe or extended LAKe by defining three kinds of elements: data, services and method framework.

Additional resources for this application are described in appendix A.2.2.

8.4 Conclusion

We presented four applications which demonstrate the genericity of the solution proposed in YML. The applications chosen are representative of their domain and highlight the contribution of a workflow environment to large scale distributed systems. We presented two non numerical applications: a distributed sort and an image rendering and processing application. The last two applications are numerical, the first one being representative of linear algebra problems and more specifically of the solving of eigenproblems.

We described numerical methods for the solving of linear algebra problems and introduced hybrid and multi methods for the solving of eigenvalues problems. The last application we introduced is MERAM. This application is a multi-method for the solving of large sparse distributed methods for non-Hermitian problems. We extended this method by proposing a realization on YML which allows to increase the number of collaborating methods in comparaison with previous solutions proposed in [63].

Finally, we highlighted a few applications which have been implemented using YML by other researchers from the university of Lille, FRANCE and the university of Tunis, TUNISIA:

- Block Gauss-Jordan linear system solver.
- Bisection based eigensolver for the symmetric case.
- Lanczos based eigensolver for the symmetric case.
- Several versions of the matrix vector product.

Chapter 9

YML performance evaluation

9.1 Introduction

The evaluation of performance of environments such as YML is difficult. Indeed, YML is layered on top of one or several middleware. These are responsible for the management of a dynamic platform composed of heterogeneous resources. Each peer can join and leave the system at any time. Moreover, computing resources are shared between many users and are not necessarly dedicated to the execution of computation associated with the middleware and thus YML applications. The impact of other applications (local or managed by the middleware) is almost impossible to measure. Finally, resources are exploited during their inactivity period which is also difficult to take into account in the evaluation process. The definition of a methodology for performance evaluation in such an environment is out of the scope of this thesis.

In this chapter, we present an early evaluation of YML. We illustrate the performance obtained with YML using both OmniRPC and XtremWeb on two experimental platforms. We evaluate the performance of YML through the applications presented in the previous chapter with an emphasis on the distributed sort and MERAM.

The first chapter describes the experimental platforms used. We then evaluates YML performance. Finally, we present detailed results for MERAM applications. The conclusion highlights the aspect that can be improved in YML in order to increase significantly the performance of YML applications.

9.2 Platforms

We executed YML on two runtime environments. The first platform is dedicated to the scientific community. It is a scientific tool for the evaluation of large scale experiments in distributed computing. The second platform is a *real-life* platform composed of nodes located in France and Japan and shared with end-users. Unlike the first platform, it is a production platform not dedicated to research.

9.2.1 GRID 5000 platform

The main experimental platform is GRID 5000 [187]. It is a research tool dedicated to experimentation on large scale distributed systems. It provides scientists with a platform which can be tuned and configured to match the experiment needs in domains such as operating system, networking, grid middleware, and applications for large scale architectures. It is driven through a reservation policy where each user is granted a specified number of processors for a specific time slot. The user can then deploy its own environment: operating system, distributed middleware, software and libraries to support the execution of scientific experiments.

It is composed of nodes located in various universities in France. Each site is connected to the others using a 1 Gbit/s wan provided by Renater. The platform evolves with time and it is not as simple to describe as a high performance computer. During our experiments, we mostly make use of resources located in Orsay (GDX cluster) which are composed of single or dual processor computers (AMD Opteron family, from 2.0 and 2.6 GHz). Each processor has a single core and the main memory available on each computer is at least 2GB. We also gather resources from other clusters when the number of resources available at Orsay are lower than the maximum concurrency level of our applications.

We focused our experiments on GRID 5000 to the OmniRPC middleware. We used ssh to connect the OmniRPC client to its agent. We used OmniRPC using the direct mode for all experiments which harnessed less than 90 nodes and the cluster-mode otherwise. In cluster mode, we used one agent for about fifty computing resources.

9.2.2 Eudil/Tsukuba platform

Unlike GRID 5000, the second platform is not a dedicated tool for the experimentation. On the opposite, this platform is a *real-life* environment where regular users use resources during the day. We make use of the resources available using a cycle stealing policy. It means that computing resources are used by the middleware when they are idle. Several criteria can be part of determining whether a host is idle or not. In our deployed grid we measure idleness based on the cpu usage and assume that resources can contribute to the middleware whenever the cpu usage is less than 20%. The platform is really dynamic and the amount of nodes involved in our experiments varies from one experiment to another. It can also change during the experiment.

The nodes composing the middleware are located both at the university of Tsukuba in Japan and at the university of Lille in France (more specifically at the Eudil engineering school). The total number of harnessed hosts is about a hundred hosts. The platform contains more heterogeneous resources in terms of processor model and operating systems. The resources are based on single and dual processors nodes based on intel celeron and pentium 4 processors as well as AMD processors. Some nodes also integrate dual core processors. The amount of memory available on a host varies from 512MB up to 2GB. The network and available storage vary significantly from one host to another.

On this platform, we deployed both XtremWeb and OmniRPC middleware in order to be able to compare the performance of the two middleware. We also deployed the YML worker with and without support for caching of the services binaries.

9.2.3 YML deployment

YML is component oriented and thus several configurations can be used to deploy YML during an experiment. In order to describe our experimental platform, we also need to present the layout use for the component involved in the execution of YML.

In order to support the execution of an application, YML mainly uses the following components: the data repository server and the scheduler. We adopted the simplest layout: The scheduler and the data repository server are both located on the same node which is also used to interact with the middleware.

When using OmniRPC, there is no real server for the middleware. The YML scheduler acts as the master of the system. However, XtremWeb is architectured around a server. This server is responsible for the management of the platform. Due to deployment issues, we were not able to experiment using XtremWeb and yml components on distinct nodes. Despite the important latency of a system such as XtremWeb, we don't think that the impact of interacting with a distant XtremWeb server will introduce that much latency. We will detail this aspect when we present the performance of YML.

9.3 YML evaluation

The evaluation of YML begins with the performance of some applications. We do not take into account the specificity of applications. We analyse several workflow processes based on a set of common matrices and highlight the behavior of YML. The table 9.1 presents a few results gathered from various applications. It shows that YML is able to process large data sets in a reasonable time. The Image application highlights that YML is not efficient when facing applications with really large concurrency. Among the experimentations presented below the maximum number of tasks executed per minute is 80 which represents a transfer rate of 320Mbytes per minute on Grid'5000. On the EUDIL platform the number of tasks executed per minute is around 30 and corresponds to a transfer rate of 60Mbytes per minute.[7]

App.	Platform	Tasks	Comms.	Concur.	Exec. Time	Workers
Dsort	EUDIL(OmniRPC)	560	832Mb	32	1h 55m 30s	32
DSort	EUDIL(OmniRPC)[8]	1913	2.8Gb	128	47m 46s	89
Image	Grid'5000	40000	1.6Gb	40000	7h 40m 14s	129
Image	Grid'5000	240	960Mb	240	3m 38s	116

Table 9.1: Performance evaluation of YML

9.4 Applications evaluation

We first focus on the performance of YML regardless of the application. In this section we are going further by focusing on one application at a time. Our analysis takes into

[7]This section is going to be extended in the final version of the document.

account information specific to the studied application.

9.4.1 Distributed sort

The distributed sort workflow process depends on two parameters. The number of blocks representing the number of tasks (service execution) and the concurrency. The second parameter defines the size of blocks. It thus determines the amount of data transfered for each communication. We present results obtained on Grid'5000 as well as the EUDIL platform using both OmniRPC and XtremWeb.

The table [9] presents the results of the experiments on the Grid'5000 platform using the OmniRPC back-ends. The worker does not provide any caching mechanism in those experiments. The most significant result in this table is the last column. It clearly highlights the effect of the size of the blocks on the performance. It also shows that the performance of applications: Those built upon YML are more efficient when using small blocks of data rather than huge ones. This is due to the data repository component which currently acts as a bottleneck for the whole system.

	Size of blocks			
blocks/tasks	1000	10000	100000	1000000
16/152	35s	36s	52s	53s
32/560	57s	56s	1m 1s	3m 02s
64/2144	4m 38s	4m 41s	5m 38s	15m 57s
128/8384	50m 38s	52m 10s	54m 19s	N/A
256/33152	11h 53m 14s	N/A	11h 48m 1s	14h 29m 38s

Table 9.2: Execution times of the distributed sort on Grid'5000 (OmniRPC)

The table 9.3 presents the performance obtained using the EUDIL platform with both OmniRPC and XtremWeb back-ends. The worker enables the caching of the computing services on the peers involved in the execution of the application. As shown in this table, YML application can be executed on different middleware. The exact same services are used in both cases. We highlight the overhead related to the deployment of computing services for each service and show the efficiency of the use of a cache for computing services. Use of caching on the worker is one of the most promising ways to improve the performance of YML applications. We also highlight the overhead due to the submission mechanism of XtremWeb and its client/dispatcher/worker architecture. Despite the fact that using XtremWeb for managing a platform such as Eudil is easier and more flexible than using OmniRPC, this is the initial targeted platform for YML and it constitutes the most significant proof of concept for our environment.

[9] Missing results will be added for the final version.

	Size of blocks			
blocks/tasks	1000	10000	100000	1000000
OmniRPC without cache on the worker				
32/560	8m 48s	9m 46s	20m 4s	1h 55m 30s
64/2144	34m 16s	N/A	47m 10s	N/A
128/8384	50m 38s	52m 10s	54m 19s	N/A
OmniRPC with cache on the worker				
32/560	40s	N/A	11m 23s	1h 44m 54s
64/2144	1m	5m 8s	47m 55s	15m 57s
128/8384	2m 33s	20m 12s	54m 19s	N/A
XtremWeb with cache on the worker				
32/560	39m 18s	N/A	39m 35s	2h 9m 19s
64/2144	1h 57m 46s	2h 3m 43s	2h 5m 12s	4h 45m 31s

Table 9.3: Execution times of the distributed sort on the EUDIL Platform (OmniRPC and XtremWeb)

Matrix name	**Size**	**NNZ**	**Froebenius norm**
ran12000	12000	4800000	10^{+3}
af23560	23560	484256	10^{+4}
pde490000	490000	2447200	10^{+3}
pde1000000	1000000	4996000	10^{+3}

Table 9.4: List of matrices used for experiments

9.4.2 Multiple explicitly restarted Arnoldi method

Asynchronous hybrid methods for linear algebra applications are one of the goals of YML. Indeed, it provides a rich application context which stresses the system and highlights many problems such an environment must solve. Nevertheless, they are potentially really interesting for large scale middleware as they gain from being used in an heterogeneous environment. They are also fully asynchronous and fault tolerant. Their main drawbacks are due to the iterative structure of the method. Indeed each iteration requires a set of results produced in the previous iterations and many components of the method are not fit for parallelization. In that respect the matrix vector product is a first step toward a parallel version of MERAM using YML.

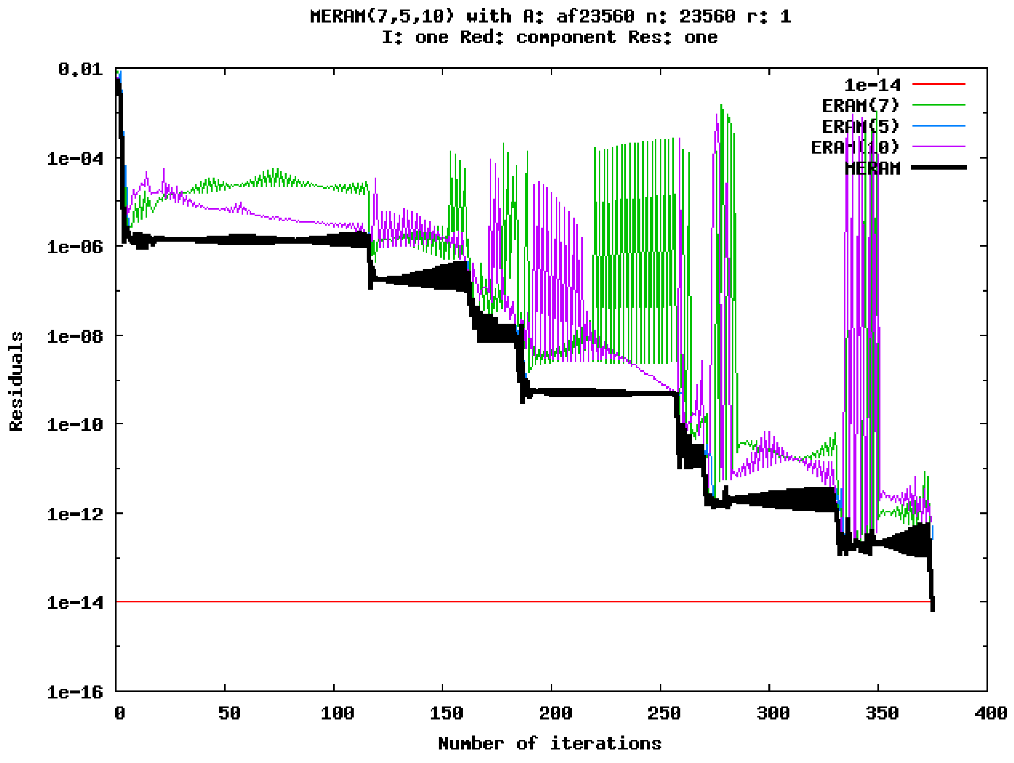
For our experiments, we selected a set of matrices from the MatrixMarket collection. The used matrices are summarized in the table 9.4. *NNZ* corresponds to the number of non zero elements of the matrix, we also added the Froebenius norm which impacts on the convergence criteria.

The experiments described in the remainder of this section have been executed using YML on Grid'5000. Due to the limited concurrency of the method, we make use of computing resources of the Grid Explorer cluster of Grid'5000 located in Orsay. We demonstrate the validity of the approach (a) by presenting the feasibility, (b) by decreasing the number of iterations needed to converge when the number of co-method increases and (c) by showing the scalability of the solution in regards of the matrix sizes and of the number of co-methods used for eigenproblems.

The MERAM method defines a set of parameters, the most significant ones are A the matrix used, n the size of the matrix, r the number of eigenelements expected, $m_1, .., m_p$ the size of the subspaces used for each co-method also noted MERAM $(m_1, ..., m_p)$. *tol* denotes the tolerance expected for the results. Lastly, MERAM requires the selection of the strategy to create the initial vector(I), the selection of the vectors to form the restarting vector(Red) and that one defining the restarting strategy(Res). The figure 9.1 illustrates two experiments with the af23560 matrix. On the top, the method converged in 375 iterations. On the bottom, the method doesn't converge. It is stopped after five hundred iterations. The only difference between the two experiments is the number of wanted eigenelements. The bold curve corresponds to MERAM. An horizontal line denotes the tolerance or the error allowed on the results. The vertical axis represents the estimated error of the solution obtained at each iteration. The horizontal axis represents the number of iterations.

In figure 9.2, we present two executions of MERAM for the matrix pde490000. The two executions differ in the number of involved co-methods. In the execution on the left, MERAM(10,30,50) requires 98 iterations to converge while MERAM(10, 20, 30, 50) requires 91 iterations. In other words, the increase in the number of co-methods decreases the iteration count of the hybrid method. We notice that by making use of YML/LAKe we are able to overcome the limitation in the number of co-methods composing a hybrid method.

One of the motivations of YML/LAKe is the scalability issue in regard to the number of co-methods composing a hybrid one and the size of the problems to be solved. Figure 9.3 illustrates progresses made in that regard. Using YML/LAKe we have been able to solve eigenproblems with one million-order matrices. Our approach is based onto the fragmentation in blocks of the matrix of the problem and its distribution during the projection step of the iterative method. The second scalability issue relates to the number of co-methods used to solve an eigenproblem. Extended LAKe allows to test the hybrid methods composed by only a small number of co-methods (up to 3). Using YML/LAKe we have been able to test effortlessly with ten co-methods and it is possible to increase this number. This is also illustrated in figure 9.4. In that set of charts we present the execution of MERAM on the ran12000 matrix. The two upper charts highlight the number of iterations needed to converge when using 6 co-methods while the two lower ones describe the result when using 9 co-methods. The horizontal axis of the left charts represent the number of iterations and the vertical axis represent the estimated error. The horizontal axis of the right charts represent the time and the vertical axis represent the time used per iteration. As we can see the number of iterations needed to converge is slightly smaller when using nine co-methods. However, the time needed to compute the solution is bigger due to the increased amount of data transfers.



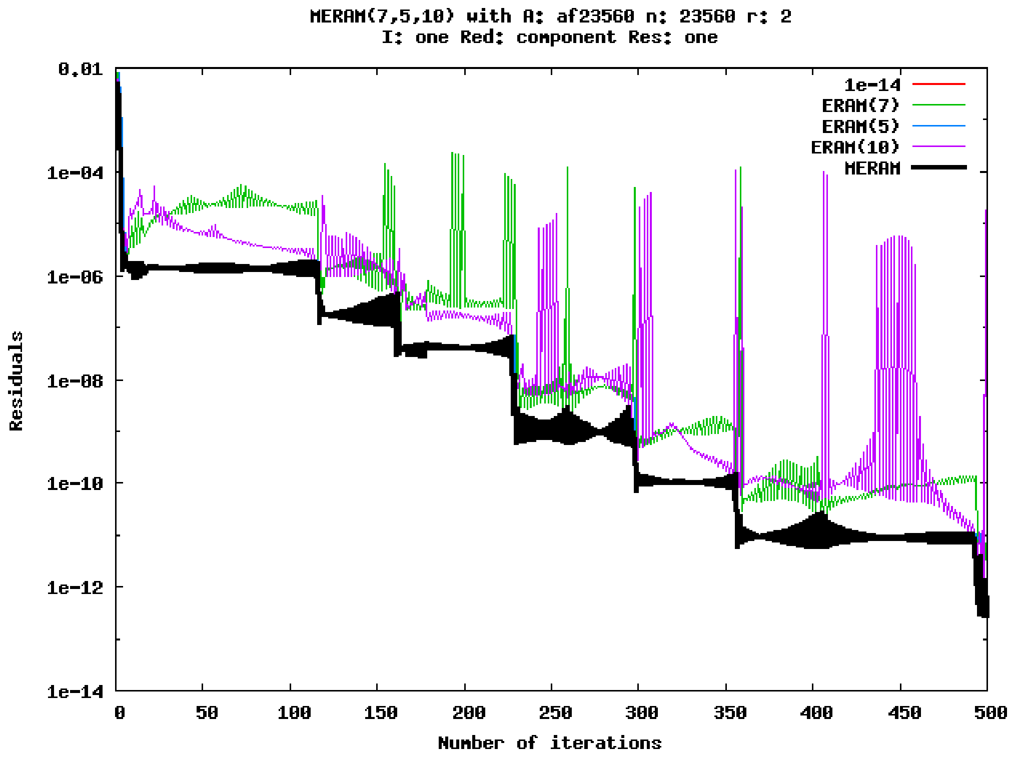


Figure 9.1: Convergence of MERAM(7,5,10) for the matrix af23560

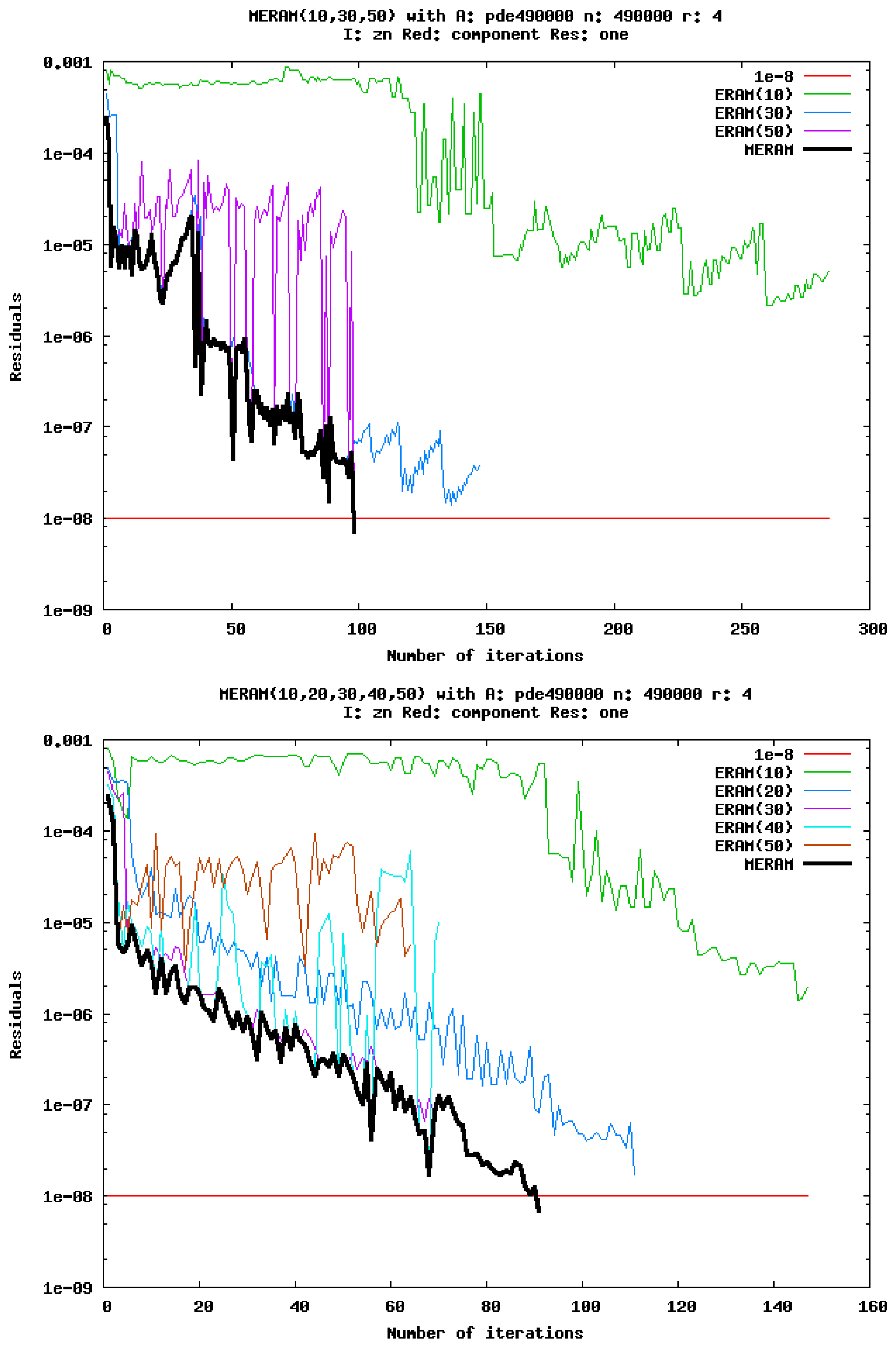
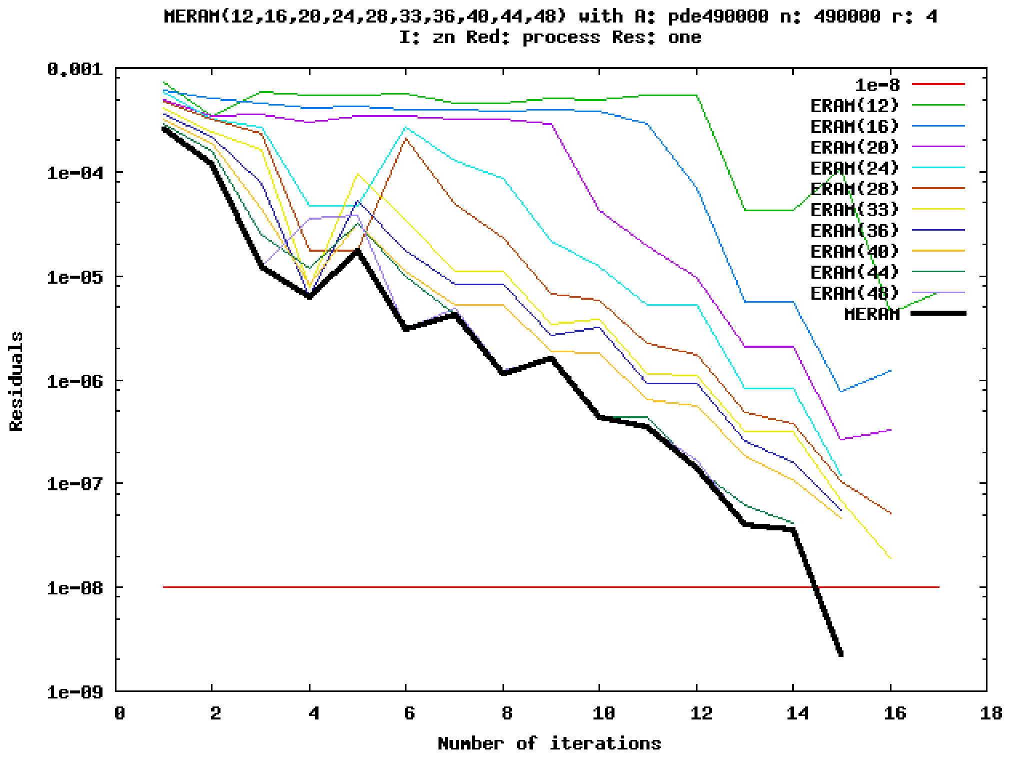


Figure 9.2: Convergence of MERAM for matrix 490000 with different numbers of co-methods



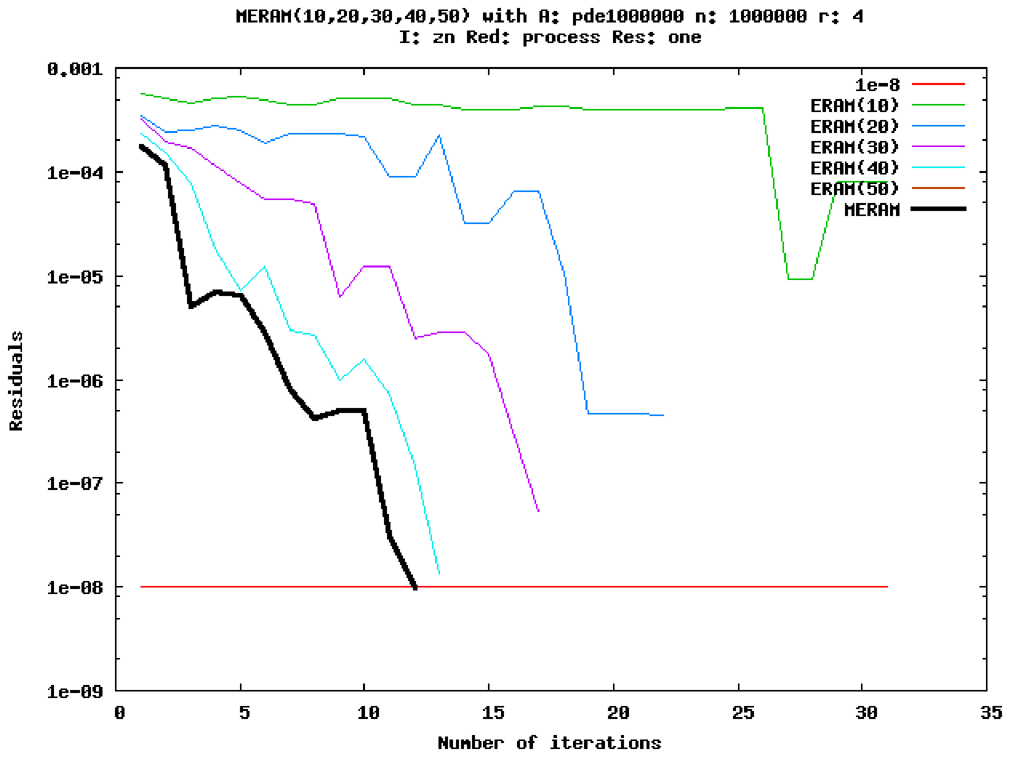


Figure 9.3: Scalability of the solution: number of co-methods/size of A

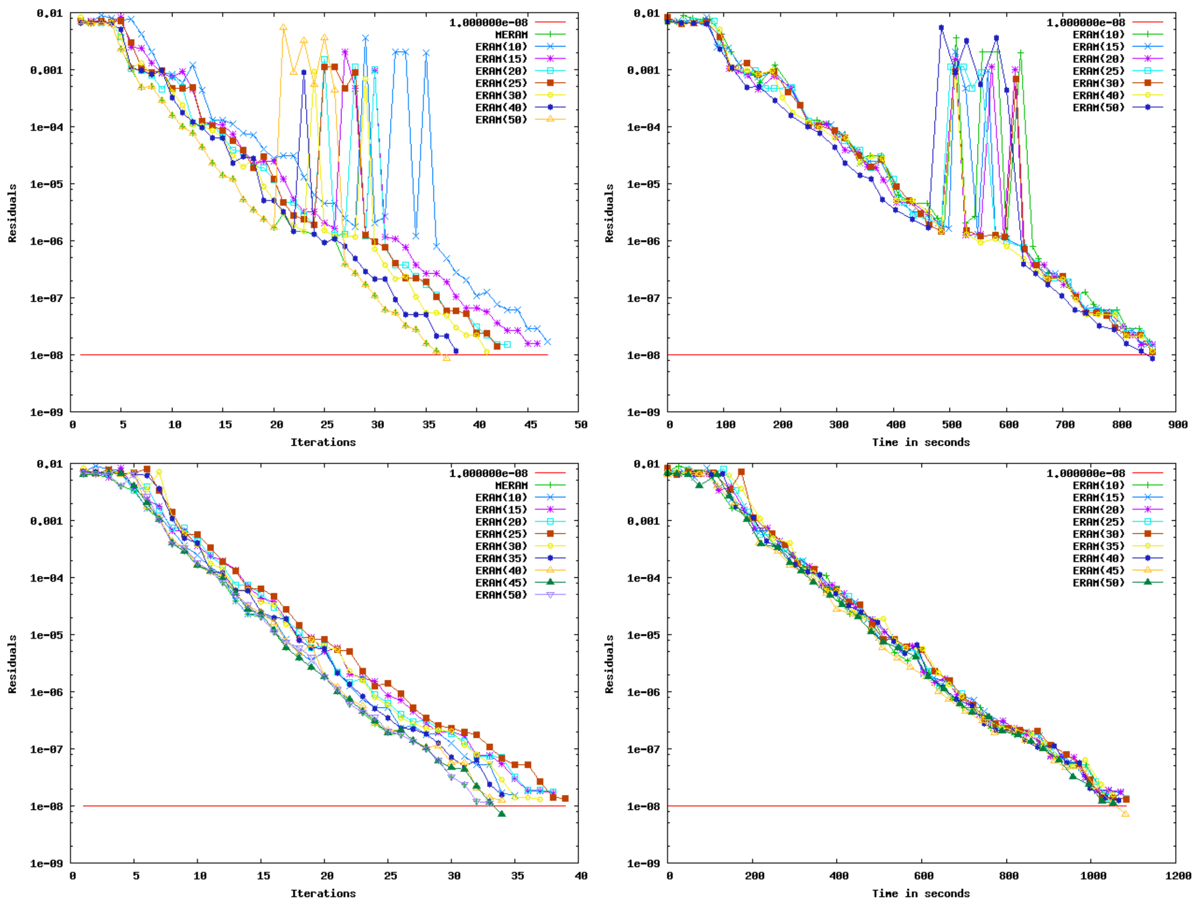


Figure 9.4: Scalabitlity of the solution: number of co-methods on ran12000

9.5 Conclustion

The experiments presented in this chapter demonstrated that YML is able to help during the definition of applications for large scale distributed systems. The experiments confirmed the interest of the approach used in YML and how YML can hide the complexity related to the exploitation of such platforms in the real life.

The experiments presented in this chapter highlighted several aspects that could evolve in order to improve the performance of the system. The results in this chapter show the feasibility of the approach. Indeed, among the middleware we have chosen, at least one is dedicated to the exploitation of idle resources connected through Internet. Thus, it is important to point out that this kind of computer usage is *free* and can just be considered as extra resources. It is then realistic to exploit them even not as efficiently as they could be in our applications.

A high level approach such as YML naturally comes with a cost. The current version of YML introduces an important overhead compared to direct use of middleware. We decided not to measure and discuss this overhead in our performance evaluation. Several evaluations of the overhead introduced by YML has been given in [37, 101]. They make use of the OmniRPC back-end. They are accurate because they measure the performance of two different applications that solve the same problem. Indeed, they both make use of the master peer for computing simple computations. Nevertheless, the overhead presented in these documents shows that YML should provide support for local execution of components using built-in components as described in 6.2 and that improvement is possible in the execution of services.

YML is an important tool for any application which can be detached from the underlying runtime environment. If performance becomes a significant issue, there is plenty of room for improvements in the following areas of YML:

1. Data management policies: YML lacks a decent data repository. It is currently the bottleneck of the system and could be improved significantly by introducing caching, replication, and distribution system based on a solution like Bittorent for example.

2. Planning: Due to its independance with middleware, YML does not integrate any execution planning tool. It would be really interesting to allow back-ends to analyse the application graph in order to constitute execution plans. This would allow optimistic strategies for data placement and direct communication between peers involved in the system.

3. Worker: The YML worker provides a convenient glue between middleware resources and YML services. It makes an intensive use of archives to represent data associated to the execution of a service. A more efficient solution would consist in using versioning on the application data store in order to allow workers to retrieve data one at a time. This would decrease significantly IO on the one hand and also allow automatic caching by the worker.

Chapter 10

Conclusion

The development of parallel applications for distributed systems is difficult. Indeed, the complexity of runtime environments keeps increasing while the tools to develop applications are mostly identical. In that respect, two families of runtime environments are currently evolving in similar ways. Nowadays high performance systems and large scale distributed systems are composed of an increasing number of heterogeneous and volatile computing resources. End-users must develop applications which take into account the dynamicity of these systems, but they have to deal with the lack of standardized programming interfaces for grid, global computing and peer to peer systems.

In this thesis, we focused mainly on the modeling and realization of a programming environment which provides solutions to the aforementioned problems. We first defined a model of workflow environment. It is built upon the notion of components. A component is responsible for one well specified role. Each component is defined in terms of an abstract interface and one or several concrete realizations. The model identifies three logical layers responsible for the interaction with end-users, the interaction with middleware and finally the management of workflow processes. We then presented YML, a realization of this model. YML is a scientific workflow environment dedicated to the creation and exploitation of large scale distributed systems.

A front-end layer provides three ways to interact with YML components. An IDE manages the creation of workflow processes. It provides wizards to assist the user. A web portal is also provided in order to exploit a YML application. It provides mechanisms to configure or pre-process an execution, monitor the execution of a workflow and analyse intermediate and final results. Both tools rely on a set of command line tools which are clients for the component provided by the other layers of YML.

The interaction with middleware is managed by the back-end layer. This layer solves issues introduced by large scale distributed systems. It defines a set of components to interact with supported middleware. YML provides concrete back-ends for executing workflow on top of the OmniRPC and XtremWeb middleware. It also provides two back-ends used to assist the user during the creation of workflow applications. Finally it allows the execution of a YML workflow to be executed on resources provided by multiple middleware at the same time. YML federates resources provided by several middleware in order to support the execution of applications.

A kernel layer connects the other layers together. It is responsible for the workflow management and the services integration. The kernel layer is composed of a collection of components dedicated to the manipulation of workflow process definitions. In YML, the process definition includes the services used during the computation as well as the description of the coordination between services. YML workflow process definition is textual and relies on XML and a dedicated language.

The creation of services involved in the execution process is part of YML. The integration of services relies on a service generator component. It interacts with two catalogs used to store the services according to whether they are abstract or concrete. These catalogs are then used to gather information about existing services during the compilation and the execution (respectively) of workflows.

YML separates workflow compilation and scheduling stages. This approach allows the decomposition of a workflow environment into independent components, removes the cost of workflow analysis from the execution, and also enables planning strategy and a more detailed analysis of the control flow thanks to the knowledge of the complete set of tasks composing a workflow.

In order to demonstrate the approach, we presented four applications which have been used to highlight its genericity as well as the performance achieved by the environment. The first two applications are not related to numerical problem solving. They consist in a distributed sort application and an image synthesis and post-processing application, both working on data set of arbitrary size. The last two applications are directly related to the solving of linear algebra problems. We presented a scalable version of a matrix vector product. This is a critical application used many times during the solving of linear problems and especially during the projection step used in many iterative methods for the solving of linear systems and eigenproblems. Our last application, called MERAM, is an implementation of an hybrid method for the solving of eigenproblems. This method uses multiple instances of the Arnoldi method collaborating asynchronously.

The performance evaluation allowed us to point out several optimizations which are needed in order to significantly decrease the execution time of YML workflow processes. Indeed, YML is the work of a small number of people. It is still a relatively new software with a few users. In order to attract more users, several aspects need enhancements. YML suffers from a significant overhead due to the current realization of the worker component. Transparently for applications defined using YML, we can improve significantly the performance of the system by adding a cache mechanism to the worker. Alongside with this first improvement, the latency of the system can be decreased by removing the creation of packs or archives of data before network communications. Those are not needed. They can be generated on the fly while sending the results back to YML. This last approach could greatly improve the performance of the system at a small cost.

Despite the increasing performance of networks, compression of communications is still useful. Indeed the compression can happen once at the time of the creation of the results of a service. YML does not inspect the content of files. It is then unneeded to uncompress this file on the server side. This approach has the benefits of increasing the ratio computation/communication transparently.

Currently the YML worker is identical from one middleware to another. However, some middleware and especially GRID middleware could use a worker able to execute several concrete services without being stopped. Indeed this would allow direct communication between peers as well as data persistency which is not supported by the requirements of the architecture model used. This approach is unfortunately not really possible for peer to peer middleware because of the volatility of peers involved.

Thanks to the contribution of new back-end components for Condor and DRMAA, the integration of middleware could be extended to rely on more middleware features if available. For example, scheduling strategies are useful once it is possible to assign resources explicitly. Scheduling strategies can then decide to allocate multiple works to the same computing device and thus benefit from data persistence mechanisms.

We think that despite its youth, YML can be used as a support for future research in topics such as quality of services and grid economy, scheduling, data management and applications for large scale distributed systems.

With the introduction of multiple core processors, existing software need to evoluate. The first step of this evolution is the optimization of basic linear algebra libraries such as BLAS and LAPACK. However, the optimization of these libraries is difficult due to the lack of adapted programming language, tools, and operating systems. Indeed, currently existing tools are not adapted anymore: They don't benefit at all from the really low communication time needed to share data between processor cores.

In order to efficiently use these processors as well as high performance systems, new programming models need to be investigated. Solutions derivated from the UPC language represent a first step in this direction. We also think that a workflow extension to regular languages would be worth considering and well adapted to the efficiency of communication between the cores composing a processor. It is especially interesting as the granularity of the workflow can be mapped to the available core without modifying the logic of the application.

Bibliography

[1] J. Dongarra A. YarKhan and K. Seymour. Gridsolve: The evolution of network enabled solver. In *Grid-Based Problem Solving Environments: IFIP TC2/WG 2.5 Working Conference on Grid-Based Problem Solving Environments*, pages 215–226, Prescott, AZ, July 2006.

[2] N. Abdennadher and R. Boesch. A large scale distributed system for high performance needs. In *Proceedings of HP-ASIA*, Biejing, China, December 2005.

[3] N. Abdennadher and R. Boesch. Towards a peer-to-peer platform for high performance computing. In *Proceedings of the Second International Conference, (GPC2007), Advances in Grid and Pervasive Computing*, Lecture Notes in Computer Science, pages 412–423, Paris, France, May 2007. Springer-Verlag.

[4] S. Agrawal, J. Dongarra, K. Seymour, and S. Vadhiyar. Netsolve: Past, present, and future - a look at a grid enabled server. In *Grid Computing: Making the Global Infrastructure a Reality.* Wiley Publishing, 2003.

[5] Y. Aida, Y. Nakajima, M. Sato, T Sakurai, D. Takahashi, and T. Boku. Performance improvement by data management layer in a grid rpc system. *Lecture Notes in Computer Science*, 3947:324–335, May 2006.

[6] A. D. Alexandrov, M. Ibel, K. E. Schauser, and C. J. Scheiman. Superweb: Towards a global web-based parallel computing infrastructure. In *11th IEEE International Parallel Processing Symposium*, April 1997.

[7] Gabrielle Allen, Kelly Davis, Konstantinos N. Dolkas, Nikolaos D. Doulamis, Tom Goodale, Thilo Kielmann, André Merzky, Jarek Nabrzyski, Juliusz Pukacki, Thomas Radke, Michael Russell, Ed Seidel, John Shalf, and Ian Taylor. Enabling Applications on the Grid: A GridLab Overview. *International Journal of High Performance Computing Applications: Special Issue on Grid Computing: Infrastructure and Applications*, 17(4):449–466, November 2003.

[8] G. S. Almasi and A. Gottlieb. Highly parallel computing, 1989.

[9] I. Altintas, A. Birnbaum, K. Baldridge, W. Sudholt, M. Miller, C. Amoreira, Y. Potier, and B. Ludaesher. A framework for the design and reuse of grid workflows. In *Internaltional Workshop on Scientific Applications on Grid Computing (SAG0́4)*, volume 3458 of *Lecture Notes in Computer Science.* Springer, 2005.

[10] A. Amar, R. Bolze, A. Bouteiller, A. Chis, Y. Caniou, E. Caron, P. K. Chouhan, G. Le Mahec, H. Dail, B. Depardon, F. Desprez, J.S. Gay, and A. Su. Diet: New developments and recent results. In *Euro-Par 2006: Parallel Processing*, volume 4375 of *Lecture Notes in Computer Science*, pages 150–170. Springer Berlin / Heidelberg, June 2007.

[11] K. Amin, G. von Laszewski, and A. R. Mikler. Grid Computing for the Masses: An Overview. In *Grid and Cooperative Computing (GCC2003)*, pages 464–473, Shanghai, China, December 2003.

[12] D. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. Seti@home: an experiment in public-resource computing. *Commun. ACM*, 45(11):56–61, 2002.

[13] D. P. Anderson. Boinc: A system for public-resource computing and storage. In *GRID '04: Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing (GRID'04)*, pages 4–10, Washington, DC, USA, 2004. IEEE Computer Society.

[14] D. P. Anderson. Public computing: Reconnecting people to science. In *In proceedings of the Conference on Shared Knowledge and the Web*, Madrid, Spain, March 2004.

[15] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[16] D. Arnold, S. Agrawal, S. Blackford, J. Dongarra, M. Miller, K. Seymour, K. Sagi, Z. Shi, and S. Vadhiyar. Users' Guide to NetSolve V1.4.1. Innovative Computing Dept. Technical Report ICL-UT-02-05, University of Tennessee, Knoxville, TN, June 2002.

[17] T.N. Ellahi B. Hudzia, L. McDermott and T. Kechadi. Entity based peer to peer in data grid environments. In *17th IMACS World Congress, Paris, France*, 2005.

[18] H. E. Bal, A. Plaat, T. Kielmann, J. Maassen, R. van Nieuwpoort, and R. Veldema. Parallel computing on wide-area clusters : the albatross project. In *In proceedings of Extreme Linux Workshop*, pages 20–24, California, USA, 1999.

[19] S. Balay, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, Matthew G. Knepley, L. Curfman McInnes, B. F. Smith, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 2.1.5, Argonne National Laboratory, 2004.

[20] S. Balay, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. Curfman McInnes, B. F. Smith, and H. Zhang. PETSc Web page, 2001. http://www.mcs.anl.gov/petsc.

[21] S. Balay, W. D. Gropp, L. Curfman McInnes, and B. F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[22] A. Barak and O. La'adan. The mosix multicomputer operating system for high performance cluster computing. *Journal of Future Generation Computer Systems*, 13(4/5):361–372, March 1998.

[23] A. Baratloo, M. Karaul, Z. Kedem, and P. Wyckoff. Charlotte: Metacomputing on the web. In *In Procedding of the 9th Conference on Parallel and Distributed Computing Systems*, 1996.

[24] A. Beguelin, J. J. Dongarra, A. Geist, and R. Manchek. Tools for heterogeneous network computing. pages 854–861, 1993.

[25] Rüdiger Berlich, Marcel Kunze, and Kilian Schwarz. Grid computing in europe: From research to deployment. volume 44 of *CRPITV*, 2005.

[26] M. Beynon, R. A. Ferreira, T. M. Kurc, A. Sussman, and J. H. Saltz. Datacutter: Middleware for filtering very large scientific datasets on archival storage systems. *Eighth NASA Goddard Conference on Mass Storage Systems and Technologies/-Seventeenth IEEE Symposium on Mass Storage Systems*, pages 119–133, December 2000.

[27] M. D. Beynon, T. Kurc, U. Catalyurek, C. Chang, A. Sussman, and J. H. Saltz. Distributed processing of very large datasets with datacutter. *Parallel Computing*, 27(11):1457–1478, 2001.

[28] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An updated set of Basic Linear Algebra Subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28(2):135–151, June 2002.

[29] T. Brecht, H. Sandhu, M. Shan, and J. Talbot. Paraweb: Towards worldwide supercomputing. In *In proceedings of the Seventh ACM SIGOPS European Workshop on System Support for Worldwide Applications*, 1996.

[30] R. Calkin and R. J. Enbody. Portable programming with the pharmacs message-passing library. *Parallel Computing.*

[31] J. Cao, S. A. Jarvis, S. Saini, and G. R. Nudd. Gridflow: Workflow management for grid computing. *ccgrid*, 00:198, 2003.

[32] Capacity and Capability Computing in Legion. A. natrajan and m. humphrey and a. grimshaw. In *International Conference on Computational Science*, May 2001.

[33] 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*, 21(3):417–437, March 2005.

[34] N. Carriero and D. Gelernter. Linda in context. *Communication of the ACM*, 32(4):444–458, April 1989.

[35] H. Casanova, M. Kim, J. S. Plank, and J. J. Dongarra. Adaptive scheduling for task farming with grid middleware. *The International Journal of High Performance Computing Applications*, 13(3):231–240, Fall 1999.

[36] H. Casanova, G. Obertelli, F. Berman, and R. Wolski. The apples parameter sweep template: user-level middleware for the grid. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 60, Washington, DC, USA, 2000. IEEE Computer Society.

[37] L. Choy. *Toward a workflow programming paradigm for linear algebra on power-aware Global Computing platform.* PhD thesis, Lille University, France, September 2007.

[38] B. Christiansen, P. Capepello, M. F. Ionescu, M. O. Neary, K. E. Schauser, and D. Wu. Javelin: Internet-based parallel computing using java. *Concurrency: Practice and Experience*, 9(11):1139–1160, 1997.

[39] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science*, 2009, 2001.

[40] Internet Software Consortium and Network Wizards. Internet domain survey, January 2007. http://www.isc.org/index.pl?/ops/ds/.

[41] Parasoft Corporation. Express user's guide, 1992.

[42] D.E. Culler, J.P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann Publishers Inc., August 1998.

[43] et al D. Erwin. Unicore plus final report - uniform interface to computing resources. Technical report, BMBF, December 2002.

[44] L. Dagmun and R. Nenon. OpenMP: An industry-standard api for shared-memory programming. *IEEE COmputational Science and Engineering*, 5(1):66–78, January 1998.

[45] E. Deelman, J. Blythe, Y. Gil, and C. Kesselman. *Grid Resource Management*, chapter Workflow Management in GriPhyN. Kluwer, 2003.

[46] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M.-H. Su, K. Vahi, and M. Livny. Pegasus : Mapping scientific workflows onto the grid. In *Across Grids Conference*, Nicosia, Cyprus, 2004.

[47] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, K. Blackburn, A. Lazzarini, A. Arbree, R. Cavanaugh, and S. Koranda. Mapping abstract complex workflows onto grid environments. *Journal of Grid Computing*, 1(1):25–39, 2003.

[48] O. Delannoy and N. Emad. YML/LAKe web portal: Web based research portal for linear algebra. In *HPC 2006: High Performance Computing Symposium*, Huntsville AL, USA, April 2006. Society for Modeling and Simulation International.

[49] O. Delannoy and S. Petiton. A peer to peer computing framework: Design and performance evaluation of YML. In *Third Internaltional Symposium on Parallel and Distributed Computing and Third Internaltional Workshop on Algorithms, Models, and Tools for Parallel Computing on Heterogeneous Networks*, pages 362–369. IEEE Computer Society, July 2004.

[50] W. Van der Aalst, L. Aldred, M. Dumas, and A. ter. Design and implementation of the yawl system, 2004.

[51] A. Deshpande and M. Schultz. Efficient parallel programming with linda. In *Supercomputing '92: Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pages 238–244, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[52] Samir Djilali. P2p-rpc: Programming scientific applications on peer-to-peer systems with remote procedure call. In *In proceedings of CHEP2003, conference for Coputing in High Energy and Nuclear Physics*, March 2003.

[53] J. J. Dongarra, G. A. Geist, R. Manchek, and V. S. Sunderam. Integrated PVM framework supports heterogeneous network computing. *Computers in physics*, 7(2):166–174, – 1993.

[54] J. J. Dongarra, S. Hammarling, and A. Petitet. Case studies on the development of ScaLAPACK and the NAG numerical PVM library. pages 236–248, 1997.

[55] R. Duan, R. Prodan, and T. Fahringer. DEE: A distributed fault tolerant workflow enactment engine for grid computing. In *Proceedings of the International Conference on High Performance Computing and Communications(HPCC 05)*, Lecture Notes in Computer Science, Sorrento, Italy, September 21-25 2005. Springer Verlag.

[56] M. Dumas and A. H.M ter Hofstede. Uml activity diagrams as a workflow specification language. In *In proceedings of the UML'2001 Conference*, 2001.

[57] S. Dwarkadas, P. Keleher, A.L. Cox, and W. Zwaenepoel. An evaluation of software distributed shared memory for next-generation processors and networks. In *Proceedings of the Twentieth Symposium on Computer Architecture*, pages 144–155, May 1993.

[58] A. E. Eichenberger, J. K. O'Brien, K. M. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. K. Gschwind, R. Archambault, Y. Gao, and R. Koo. A compiler enabling and exploiting

the cell broadband processor architecture. *IBM Systems Journal Special Issue on Online Game Technology*, 45(1), January 2006.

[59] T.N. Ellahi, B. Hudzia, L. McDermott, and M-T. Kechadi. Entity management and security in p2p grid framework. In *7th Int'l. Conference on Computing, Journal Research in Computing Science*, ISSN: 1665-9899, Mexico City, Mexico, May 8-10 2006.

[60] N. Emad. Mapping strategies in data parallel programming models; the projection methods. *Parallel and Distributed Computing Practices Journal*, 2:41–51, 2000.

[61] N. Emad, O.Hamdi, and Z. Mahjoub. On Sparse Matrix-Vector Product Optimization. Egypte, January 2005.

[62] N. Emad, S. Petiton, and G. Edjlali. Multiple explicitly restarted arnoldi method for solving large eigenproblems. *SIAM Journal on Scientific Computing*, 27(1):253–277, september 2005.

[63] N. Emad and A. Sedrakian. Toward the Reusability for Iterative Linear Algebra Software in Distributed Environment. *Parallel Computing*, 32(3):251–266, March 2006.

[64] N. Emad, S.-A. Shahzadeh-Fazeli, and J. Dongarra. An asynchronous algorithm on netsolve global computing system. *Future Generation Computing Systems(FGCS)*, 22(3):279–290, Febuary 2006.

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

[66] G. Fagg and J. J. Dongarra. Ft-mpi: Fault tolerant mpi, supporting dynamic applications in a dynamic world. In Springer Verlag, editor, *Lecture Notes in Computer Science: Proceedings of EuroPVM-MPI 2000*, volume 1908, pages 346–353, 2000.

[67] G. E. Fagg, K. S. London, and J. Dongarra. Mpi_connect managing heterogeneous mpi applications ineroperation and process control. In *Proceedings of the 5th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 93–96, London, UK, 1998. Springer-Verlag.

[68] T. Fahringer, A. Jugrav, S. Pllana, R. Prodan, C.S. Jr, and H. L. Truong. ASKALON: a tool set for cluster and grid computing. *Concurrency and Computation: Practice and Experience*, 17:143–169, 2005.

[69] T. Fahringer, S. Pllana, and J. Testori. Teuta: Tool support for performance modeling of distributed and parallel applications. In *International Conference on Computational Science, Tools for Program Development and Analysis in Computational Science*, Krakow, Poland, June 2004. Springer-Verlag.

[70] T. Fahringer, S. Pllana, and A. Villazon. AGWL: Abstract Grid Workflow Language. In *International Conference on Computational Science. Programming Paradigms for Grids and Metacomputing Systems.*, Krakow, Poland, June 2004. Springer-Verlag.

[71] T. Fahringer, J. Qin, and S. Hainzer. Specification of Grid Workflow Applications with AGWL: An Abstract Grid Workflow Language. In *Proceedings of IEEE International Symposium on Cluster Computing and the Grid 2005 (CCGrid 2005)*, Cardiff, UK, May 9-12 2005. IEEE Computer Society Press.

[72] G. Fedak, C. Germain, V. Néri, and F. Cappello. Xtremweb: A generic global computing system. In *In IEEE Int. Symp. on Cluster Computing and the Grid (CCGRID'2001)*, May 2001.

[73] Gilles Fedak. *XtremWeb: une plate-forme pour l'étude expérimentale du calcul global pair-à-pair.* PhD thesis, Université Paris XI, Laboratoire de Recherche en Informatique (LRI), June 2003.

[74] M.J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computing*, C-21:948–960, 1972.

[75] Message Passing Interface Forum. Mpi: A message-passing interface standard. Technical Report UT-CS-94-230, MPI Forum, 1994.

[76] Message Passing Interface Forum. Mpi-2: Extensions to the message-passing interface. Technical report, University of Tennessee, July 1997.

[77] I. Foster. The anatomy of the grid: Enableing scalable virtual organizations. *Internal Journal on Supercomputer Applications*, 2001.

[78] I. Foster. What is the grid? a three point checklist, July 2002.

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

[80] I. Foster, J. Geisler, W.Nickless, W. Smith, and S. Tuecke. Software infrastructure for the i-way high performance distributed computing experiment. In *In Proceedings of the 5th IEEE Symposium on High Performance Distributed Computing*, pages 562–571, 1997.

[81] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[82] I. Foster and C. Kesselman. *The grid: Blueprint for a future computing infrastructure.* Morgan Kaufmann Publishers, 1999.

[83] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integraton, June 2002. Open Grid Service Infrastructure WG.

[84] I. Foster, J. Kishimoto, A. Savva, D. Berry, A. Djaoui, A. Grimshaw, B. Horn, F. Maciel, F. Siebenlist, R. Subramaniam, J. Treadwell, and J. Von Reich. The open grid servoces architecture. Technical report, Global Grid Forum, January 2005.

[85] G. Fox, D. Gannon, and Mary Thomas. A summary of grid computing envrionments. *Concurrency and Computation: Practice and Experience*, 14(13-15):1035–1044, 2002.

[86] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[87] E. Gabriel, M. Resch, T. Beisel, and R. Keller. Distributed computing in a heterogeneous computing environment. In *PVM/MPI*, pages 180–187, 1998.

[88] E. Gallopoulos, E. Houstis, and J. R. Rice. Computer as thinker/doer: Problem-solving environments for computational science. *IEEE Computational Science & Engineering*, 1:11–21, 1994.

[89] L. Gong. Jxta: A network programming environment. *IEEE Internet Computing*, 5(3), 2001.

[90] B. Grayson and R. van de Geijn. A high performance parallel strassen implementation. *Parallel Processing Letters*, 6(1):3–12, 1996.

[91] A. GRimhaw and W. Wulf. The legion vision of a worldwide virtual computer. *Communications of the ACM*, 40, January 1997.

[92] Andrew S. Grimshaw, William A. Wulf, James C. French, Alfred C. Weaver, and Paul F. Reynolds Jr. Legion: The next logical step toward a nationwide virtual computer. Technical Report CS-94-21, University of Virginia, 8, 1994.

[93] 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.*, 2005.

[94] V. Hernandez, J. E. Roman, A. Tomas, and V. Vidal. SLEPc home page. http://www.grycap.upv.es/slepc, 2006.

[95] V. Hernandez, J. E. Roman, A. Tomas, and V. Vidal. SLEPc users manual. Technical Report DSIC-II/24/02 - Revision 2.3.2, D. Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, 2006.

[96] V. Hernandez, J. E. Roman, and V. Vidal. SLEPc: Scalable Library for Eigenvalue Problem Computations. *Lecture Notes in Computer Science*, 2565:377–391, 2003.

[97] V. Hernandez, J. E. Roman, and V. Vidal. SLEPc: A scalable and flexible toolkit for the solution of eigenvalue problems. *ACM Transactions on Mathematical Software*, 31(3):351–362, September 2005.

[98] E.N. Houstis, J.R. Rice, E. Gallopoulos, and R. Brambley. *Enabling Technologies for Computational Science: Feameworks, Middlewareand Environments.* Kluwer Academic Publishers, 2000.

[99] B. Hudzia, L. McDermott, T.N. Ellahi, and T. Kechadi. A java based architecture of p2p-grid middleware. In *The 2006 International Conference on Parallel and Distributed Processing Techniques and Applications*, 2006.

[100] E. Huedo, R. S. Montero, and I. M. Llorente. A framework for adaptive execution in grids. *Software: Practice and Experience*, 34(7):631–651, March 2004.

[101] M. Hugues. Algorithmique scientifique distribué à grande échelle et programmation yml sur cluster de clusters. Master's thesis, Lille University, France, 2007.

[102] T. Imamura and H. Takemiya Y. Tsujita, H. Koide. An architecture of stampi: Mpi library on a cluster of parallel computers. September 2000.

[103] R. K. Joshi. Distributed filter processes. *Concurrency and Computation: Practice and Experience*, 17:1497–1506, April 2005.

[104] N. Karonis, B. Toonen, and I. Foster. Mpich-g2: A grid-enabled implementation of the message passing interface. *Journal of Parallel and Distributed Computing*, 2003.

[105] P. Keleher, A.L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the Nineteenth Symposium on Computer Architecture*, pages 13–21, May 1992.

[106] P. Keleher, S. Dwarkadas, A.L. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the Winter 94 Usenix Conference*, pages 115–131, January 1994.

[107] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang. Magpie: Mpi's collective communication operations for clustered wide area systems. In *PPoPP '99: Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 131–140, New York, NY, USA, 1999. ACM.

[108] B. Kiepuszewski, A. Hofstede, and W. van der Aalst. Fundamentals of control flow in workflows, 2002.

[109] A Kolawa. The express programming environment. In *In proceedings of the workshop on heterogeneous Network-Based Concurrent computing*, October 1991.

[110] S. Krishnan, R. Bramley, D. Gannon, M. Govindaraju, R. Indurkar, A. Slominski, B. Temko, J. Alameda, R. Alkire, . Drews, and E. Webb. The xcat science portal. In *Supercomputing ´01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing*. ACM Press, 2001.

[111] I. S. Kumar, B. Rutt, T. M. Kurc, U. V. Catalyurek, S. Chow, S. Lamont, M. Martone, and J. H. Saltz. Large image correction and warping in a cluster environment. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing (SC2006)*, 2006.

[112] Los Alamos National Laboratory. Road-runner website, 2007. http://www.lanl.gov/orgs/hpc/roadrunner/index.shtml.

[113] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, September 1979.

[114] C. Lee and D. Talia. Grid programming models: Current tools, issues and directions. In *Grid Computing: Making the Global Infrastructure a Reality*. Wiley Publishing, 2003.

[115] R. Lehoucq, D. Sorensen, and C. Yang. Arpack users' guide: Solution of large scale eigenvalue problems with implicitly restarted arnoldi methods, 1997.

[116] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *In 8th international Conference of Distributed Computing Systems ICDCS)*, pages 104–111, Los Alamos, CA, USA, June 1988. IEEE CS Press.

[117] X. Liu, J. Liu, J. Eker, and E. A. Lee. Heterogeneous modeling and design of control systems. *Software Enabled Control: Information Technology for Dynamical Systems*, April 2003.

[118] O. Lodygensky, G. Fedak, V. Néri, A. Cordier, and F. Cappello. Auger & xtremweb : Monte carlos computation on a global computing platform. In *In proceedings of CHEP2003, conference for Coputing in High Energy and Nuclear Physics*, March 2003.

[119] B. Ludäscher, 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.

[120] D. Takahasi M. Sato, T. Boku. Omnirpc: a grid rpc system for parallel programming in cluster and grid environment. In *In proceedings of CCGrid2003*, pages 206–213, Tokyo, May 2003.

[121] J. Maassen, R. van Nieuwpoort, R. Veldema, H. E. Bal, T. Kielmann, C. J. H. Jacobs, and R. F. H. Hofman. Efficient java rmi for parallel programming. *Programming Languages and Systems*, 23(6):747–775, 2001.

[122] S. Majithia, M. S. Shields, I. J. Taylor, and I. Wang. Triana: A Graphical Web Service Composition and Execution Toolkit. In *Proceedings of the IEEE International Conference on Web Services (ICWS'04)*, pages 514–524. IEEE Computer Society, 2004.

[123] S. Majithia, I. Taylor, M. Shields, and I. Wang. Triana as a Graphical Web Services Composition Toolkit. In Simon J. Cox, editor, *Proceedings of UK e-Science All Hands Meeting*, pages 494–500. EPSRC, September 2003.

[124] V. Mann and M. Parashar. Engineering interoperable computational collaboratories on the grid. *Concurrency and Computation: Practice and Experience*, 2002.

[125] K. J. Maschhoff and D. C. Sorensen. P_ARPACK: An efficient portable large scale eigenvalue package for distributed memory parallel architectures. In *PARA*, pages 478–486, 1996.

[126] S. Matsuoka, M. Sato, H. Nakada, and S. Sekiguchi. Design issues of network enabled server systems for the grid. In *GRID '00: Proceedings of the First IEEE/ACM International Workshop on Grid Computing*, pages 4–17, London, UK, 2000. Springer-Verlag.

[127] S. McGough, L. Young, A. Afzal, S. Newhouse, and J. Darlingthon. Performance architecture within iceni. In *In UK e-Science All Hands Meeting*, pages 906–911, Nottingham, UK, September 2004. IOP Publishing Ltd.

[128] S. McGough, L. Young, A. Afzal, S. Newhouse, and J. Darlingthon. Workflow enactment in iceni. In *In UK e-Science All Hands Meeting*, pages 894–900, Nottingham, UK, September 2004. IOP Publishing Ltd.

[129] N. Melab, E.-G. Talbi, and S. Petiton. A Parallel Adaptative Gauss-Jordan Algorithm. *The Journal of Supercomputing*, 17:167–185, 2000.

[130] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[131] Y. Nakajima, M. Sato, T. Boku, D. Takahashi, and H. Goto. Performance evaluation of OmniRPC in a grid environment. In *In proceedings of SAINT2004, Workshop on High Performance Grid Computing and Networking*, pages 658–664, Jan 2004.

[132] S. Narayanan, T. M. Kurc, and J. H. Saltz U. V. Catalyurek. Database support for data-driven scientific applications in the grid. *Parallel Processing Letters*, pages 245–271, 2003.

[133] N. Nisan, S. London, and O. Regev admd N. Camiel. Globally distributed computation over the internet - the popcorn project. In *In Proceedings for the 18th International Conference on Distributed Computing Systems*, volume 6, 1998.

[134] S. Noel, O. Delannoy, N. Emad, P. Manneback, and S. Petiton. A multi-level scheduler for the grid computing yml framework. In *EuroPAR 2006 - CoreGRID Workshop*, Dresden, Germany, August 2006.

[135] E. Noulard and N. Emad. A key for reusable parallel linear algebra software. *Parallel Computing Journal, Elsevier Science*, 27(10):1299–1319, 2001.

[136] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li. Taverna: A tool for the composition and enactment of bioinformatics workflows. *Bioinformatics journal*, 17(20):3045–3054, 2004.

[137] OpenMP Architecture Review Board. *OpenMP Application Program Interface 2.5*, May 2005.

[138] OpenMP Architecture Review Board. OpenMP website, 2007. http://www.openmp.org/drupal/.

[139] H. Pedroso, L. M. Silva, and J.G. Silva. Web-based metacomputing with jet. In *In Proccedings of the ACM 1997 PPoPP Workshop on Java for Science and Engineering Computation*. ACM, June 1997.

[140] *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*, number 1132 in Lecture Note in Computer Science, Les Ménuires, France, June 1996. Springer Verlag.

[141] S. Pllana, T. Fahringer, J. Testori, S. Benkner, , and I. Brandic. Towards an uml based graphical representation of grid workflow applications. In *The 2nd European Across Grids Conference*, Cyprus, January 2004. Springer-Verlag.

[142] J. A. Pouwelse, P. Garbacki, D. H. J. Epema, and H. J. Sips. The Bittorrent P2P file-sharing system: Measurements and analysis. In *4th Int'l Workshop on Peer-to-Peer Systems (IPTPS)*, volume 3640. LNCS, Feb 2005.

[143] R. Rabenseifner. The dfn remote procedure call for parallel and distributed applications. In *In Kommunikation in Verteilten Systemen*, pages 415–429, 1995.

[144] R. Rabenseigner and A. Schuch. Comparison of dce rpc, dfn-rpc, ons and pvm. In *In DCE Workshop*, pages 39–46, 1993.

[145] M. Ripeanu. Peer-to-peer architecture case study: Gnutella network, 2001.

[146] A. Roy, I. Foster, W. Gropp, N. Karonis, V. Sander, and B. Toonen. Mpich-gq: Quality-of-service for message passing programs. November 2000.

[147] N. Russell, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst. Workflow data patterns. Qut technical report, Queensland University of Technology, Brisbane, 2004.

[148] L. F. G. Samenta, S. Hirano, and S. A. Ward. Towards bayanihan: building an extensible framework for volonteer computing using java. In *In Proceeding ACM Workshop on Java High Performance Network Computing*. ACM, 1998.

[149] 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.

[150] M. Sato, H. Nakada, S. Sekiguchi, S. Matsuoka, U. Nagashima, and H. Takagi. Ninf: A network based information library for global world-wide computing infrastructure. In *HPCN Europe*, pages 491–502, 1997.

[151] A. Selikhov, G. Bosilca, S. Germain, G. Fedak, and F. Capello. Mpich-cm: A communication library design for p2p mpi implementation. In *Proceedings of the 9-th EuroPVM/MPI conference*, Lecture Notes in Computer Science. Springer-Verlag, 2002.

[152] Gary Shao. *Adaptive Scheduling of Master/Worker Applications on Distributed Computational Resources.* PhD thesis, University of California, San Diego, 2001.

[153] J. Siegel. Corba 3.0.3. Technical report, OMG, 2004.

[154] J. Silberman, N. Aoki, D. Boerstler, J.L. Burns, Sang Dhong, A. Essbaum, U. Ghoshal, D. Heidel, P. Hofstee, Kyung Tek Lee, D. Meltzer, Hung Ngo, K. Nowka, S. Posluszny, O. Takahashi, I. Vo, and B. Zoric. A 1.0-ghz single-issue 64-bit powerpc integer processor. *IEEE Journal of Solid-State Circuits*, 33(11):1600–1608, November 1998.

[155] T. Tannenbaum, D. Wright, K. Miller, and M. Livny. Condor – a distributed job scheduler. In Thomas Sterling, editor, *Beowulf Cluster Computing with Linux.* MIT Press, October 2001.

[156] I. Taylor, M. Shields, and I. Wang. Resource Management for the Triana Peer-to-Peer Services. In Jarek Nabrzyski, Jennifer M. Schopf, and Jan Węglarz, editors, *Grid Resource Management*, pages 451–462. Kluwer Academic Publishers, 2004.

[157] I. Taylor, M. Shields, I. Wang, and A. Harrison. Visual Grid Workflow in Triana. *Journal of Grid Computing*, 3(3-4):153–169, September 2005.

[158] I. Taylor, M. Shields, I. Wang, and A. Harrison. The Triana Workflow Environment: Architecture and Applications. In I. Taylor, E. Deelman, D. Gannon, and M. Shields, editors, *Workflows for e-Science*, pages 320–339. Springer, New York, Secaucus, NJ, USA, 2007.

[159] I. Taylor, M. Shields, I. Wang, and O. Rana. Triana Applications within Grid Computing and Peer to Peer Environments. *Journal of Grid Computing*, 1(2):199–217, 2003.

[160] D. Thain, T. Tannenbaum, and M. Livny. Condor and the grid. In *Global Grid Computing: Making the Global Infrastructure a Reality*, NJ, USA, 2003. John Wiley & Sons.

[161] M. Thomas, J. Boisseau, S. Mock, M. Dahan, K. Mueller, and D. Sutton. The gridport toolkit: a system for building grid portals. In *Proceedings of the 10th IEEE Intl. Symp. on High Perf. Dist. Comp*, August 2001.

[162] P. Troger, H. Rajic, A. Haas, and P. Domagalski. Standardization of an api for distributed resource management systems. In *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2007)*, pages 619–626, Rio de Janeiro, Brazil, May 14-17 2007.

[163] Scalapack users' guide.

[164] W. van der Aalst and A. Hofstede. Yawl: Yet another workflow language, 2002.

[165] W.M.P. van der Aalst and A.H.M. ter Hofstede. Workflow patterns: On the expressive power of (petri-net-based) workflow languages.

[166] W.M.P van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(3):5–51, July 2003.

[167] W.M.P. van der Aalst, K.M. van Hee, and G.J. Houben. Modelling workflow management systems with high-level petri nets. In *Proceedings of the second Workshop on Computer-Supported Cooperative Work, Petri nets and related formalisms*, pages 31–50, 1994.

[168] A. J. van der Steen and J. J. Dongarra. Overview of recent supercomputers. Technical report, top500, 2007. http://www.top500.org/2007_overview_recent_supercomputers.

[169] H. A. Van der Vorst and G. H. Golub. 150 years old and still alive: Eigenproblems. 63:93–120, 1997.

[170] R. V. van Nieuwpoort, J. Maassen, R. Hofman, T. Kielmann, and H. E. Bal. Ibis: an efficient java based grid programming environment. In *In proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, pages 18–27. ACM Press, 2002.

[171] G. von Laszewski. Java cog kit: Workflow concepts for scientific experiments. Technical report, Argonne National Laboratory, Argonne, IL, USA, 2005.

[172] G. von Laszewski, K. Amin, M. Hategan, N. J. Zaluzec, S. Hampton, and A. Rossi. Gridant: A client-controllable grid workflow system. In *In proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS0́4)*, Big Island, Hawaii, Jannuary 2004.

[173] G. von Laszewski, I. Foster, J. Gawor, and P. Lane. A java commodity grid kit. *Concurrency and Computation: Practice and Experience*, 13(8-9):643–662, 2001.

[174] G. von Laszewski and M. Hategan. Java cog kit karajan/gridant workflow guide. Technical report, Argonne National Laboratory, Argonne, IL, USA, 2005.

[175] J. L. Vázquez-Poletti1, E. Huedo1, R. S. Montero1, and I. M. Llorente. *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, volume 4276 of *Lecture Notes in Computer Science*, pages 1143–1151. Springer Berlin / Heidelberg, November 2006.

[176] I. Wang. P2PS (Peer-to-Peer Simplified). In *Proceedings of 13th Annual Mardi Gras Conference - Frontiers of Grid Applications and Technologies*, pages 54–59. Louisiana State University, February 2005.

[177] M. Wieczorek, R. Prodan, and T. Fahringer. Scheduling of Scientific Workflows in the ASKALON Grid Environment. *ACM SIGMOD Record*, 35(3), 2005. http://dps.uibk.ac.at/ marek/publications/acm-sigmod-2005.pdf.

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

[179] Jia Yu and Rajkumar Buyya. A novel architecture for realizing grid workflow using tuple spaces. In *GRID '04: Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing*, pages 119–128, Washington, DC, USA, 2004. IEEE Computer Society.

[180] Apache ant website, 2007. http://ant.apache.org.

[181] AMD Core Math Library, 2007.

[182] Intel math kernel library, 2007. http://www.intel.com/cd/software/products/asmo-na/eng/307757.htm.

[183] *OMG Website*, Last visited 2006. http://www.omg.org.

[184] DAGMan application, December 2004. http://www.cs.wisc.edu/condor/manual/v6.4/.

[185] Eispack at netlib.org. http://www.netlib.org/eispack/.

[186] Freefluo website, 2007. http://freefluo.sourceforge.net.

[187] Grid 5000 web site, 2007. http://www.grid5000.fr/.

[188] Grid remote procedure call working group, 2007. https://forge.gridforum.org/sf/projects/gridrpc-wg.

[189] Jxta project, 2007. https://jxta.dev.java.net/.

[190] Linpack at netlib.org. http://www.netlib.org/linpack/.

[191] Open grid forum, Last visited 2007. http://www.ogf.org/.

[192] OmniRPC: Web site, 2007. http://www.omni.hpcc.jp/OmniRPC/.

[193] GryPhN project, 2004. http://www.griphyn.org.

[194] Pvm website, 2007. http://www.csm.ornl.gov/pvm/.

[195] Sun grid engine 6.1, 2007. http://www.sun.com/software/gridware/.

[196] Web services activity group, 2007. http://www.w3.org/2002/ws/.

[197] Xml-rpc specification, 2007. http://www.xmlrpc.com/spec.

[198] Xtremweb project, 2007. http://www.xtremweb.net/.

Appendix A

Introduction to the creation of workflow with YML

The aim of this appendix is to describe the creation of a workflow application with YML. It discusses the creation of the resources which compose a workflow. We present the definition of user data type, abstract and concrete services as well as the definition of a workflow process. All the resources described in this appendix are based on the application used in this thesis.

A.1 Adding user defined data type

YML can be extended through user defined data types. Currently, YML in its version 1.0.6 allows the definition of services using the C++ language. A user defined data type consists in a type and a pair of serialization methods. It is needed to add a type definition to the `yml` namespace. The type definition can be a simple type alias to an existing basic data type, a user defined class or an imported one from another namespace with the `using` keyword. Data types have to provide at least a public default constructor. The serialization of the data type to a byte stream is done using two methods named `param_import` and `param_export` which are specialized in any new data type.

Below, we present two examples taken from the distributed sort example and the image application.

A.1.1 Block of integer definition

In the distributed sort we manipulate blocks composed of collections of integers. We decided to represent a collection of integers using the `std::vector` template of the standard template library (STL) of the C++ language. The type definition of a block of integers is then really simple as the vector class is responsible for the allocation, deallocation and meets the requirements of YML.

```
#ifndef VECTOR_INTEGER_TYPE_HH
#define VECTOR_INTEGER_TYPE_HH 1
#include "integer.type.hh"
#include <cstdio>
#include <vector>
namespace yml
{
// Type definition based on the instantiation of an STL template.
typedef std::vector<integer> VectorInteger;
// Serialization method for input
template<>
bool param_import(VectorInteger& param, const char* filename) {
    FILE* data = fopen(filename, "rb");
    if (data) {
        int count;
        integer value;
        int status = fread(reinterpret_cast<char*>(&count), sizeof(
           count), 1, data);
        if (status == 1) {
            int i;
            for (i = 0 ; i < count ; ++i) {
                status = fread(reinterpret_cast<char*>(&value),
                   sizeof(value), 1, data);
                if (status == 1) {
                    param.push_back(value);
                } else {
                    fclose(data);
                    return false;
                }
            }
        } else {
            fclose(data);
            return false;
        }
        fclose(data);
        return true;
    }
    return false;
}

// Serialization function for output
template<>
bool param_export(const VectorInteger& param, const char* filename) {
    FILE* data = fopen(filename, "wb");
    if (data) {
        size_t count = param.size();
        int status;
        status = fwrite(reinterpret_cast<char*>(&count), sizeof(count), 1, data);
        if (status == 1) {
            size_t i;
            integer value;
            for(i = 0 ; i < count ; ++i) {
                value = param[i];
                status = fwrite(reinterpret_cast<char*>(&value), sizeof(value), 1, data);
                if (status != 1) {
                    fclose(data);
                    unlink(filename);
                    return false;
                }
            }
            fclose(data);
            return true;
        } else {
            fclose(data);
            unlink(filename);
            return false;
        }
        fclose(data);
        return true;
    }
    return false;
}

} // End of yml namespace
#endif
```

Listing A.1: Vector integer type definition

The serialization code in the previous listing depends on the C standard library for IO operations. YML provides an alternate interface which allows to use a more secure encoding solution based on a set of C++ abstractions. The listing below highlights the loading of a dense matrix based on the C++ abstraction for IO operations. These IO mechanisms are available to all user defined data type definitions. In this interface, a stream is decorated with a serializer to support typed binary read operation. In this short example we present only the IO abstraction for input. The same interface is available for output as well.

```
// Loading of the matrix from a file
inline bool RealMatrix::load(const char* filename) {
    // C++ Abstraction for IO operation
    FileStream in(filename, STREAM_READ_ONLY);
    if (in) {
        Serializer reader(in); // Binary typed IO operation
        reader.readUInt32(d_rows);
        reader.readUInt32(d_cols);
        d_values.resize(d_rows * d_cols);
        for(size_t i = 0 ; i < d_values.size() ; ++i)
            reader.readReal64(d_values[i]);
        if (!in) // Check that all operations are successfull
            return false;
    }
    return true;
}
// Integration with the YML service generator
template <>
bool param_import(RealMatrix& param, const char* filename) {
    return param.load(filename);
}
```

Listing A.2: C++ interface for IO operation

A.1.2 PNG Image type definition

In the image application, we exchange image from one service to another using a widely available format called PNG. It is also an interesting as it provides compression without loss of information. The listing below contains the definition of the data type. It's a header only definition which contains at the same time the definition of the `PNGImage` class as well as the operations used for the input and output of images in services. We remove the definition of the methods of the class `PNGImage` for clarity.

```
#ifndef PNGIMAGE_TYPE_HH
#define PNGIMAGE_TYPE_HH 1
#include <cstring>
#include <png.h>
namespace yml
{
  // Definition of the notion of Pixel here we use RGB images.
  struct Pixel
  {
    unsigned char red;
    unsigned char green;
    unsigned char blue;
  };
  // PNG Image class declaration
  class PNGImage
  {
  public:
    PNGImage ():d_width (0), d_height (0), d_pixels (0) {}
    PNGImage (size_t width, size_t height):d_width (width), d_height
        (height),
      d_pixels (new Pixel[d_width * d_height]) {}
    ~PNGImage () { delete[] d_pixels; }
    PNGImage(const PNGImage & source);
    PNGImage& operator= (const PNGImage & source);
    size_t width () const { return d_width; }
    size_t height () const { return d_height; }
    void setSize (size_t width, size_t height);
    void swap (PNGImage & image);
    void set(size_t x, size_t y,
             unsigned char r, unsigned char g, unsigned char b);
    const Pixel& operator   () (size_t x, size_t y) const
    { return d_pixels[x + d_width * y]; }
    Pixel & operator   () (size_t x, size_t y)
    { return d_pixels[x + d_width * y]; }
    const char *data() const
    { return reinterpret_cast<const char*>(d_pixels); }
    char *data ()
    { return reinterpret_cast < char *>(d_pixels); }
    size_t dataSize () const
    { return sizeof (Pixel) * d_width * d_height; }

    bool read (const char *path);
    bool write (const char *path) const;
  private:
    size_t d_width;
    size_t d_height;
    Pixel *d_pixels;
  };
  // Integration with the C++ services generator methods:
  // param_import is used for reading a png image
  template <>
  bool param_import(PNGImage& param, const char* filename)
  {
    return param.read(filename);
  }
  // param_export is used for writing a png image
  template <>
  bool param_export(const PNGImage& param, const char* filename)
  {
    return param.write(filename);
  }
}
#endif
```

Listing A.3: PNG Image type definition

A.2 Service definitions

In this section, we present the definitions of abstract and concrete services. In our examples, we always associate an abstract and a corresponding concrete service definition. A service requires the definition of two XML documents. An abstract service definition is used to specify how the service interacts with the outside world while a concrete service definition is used to describe the computation.

A.2.1 Matrix vector product service

The matrix vector product is an example extracted from the matrix vector application. It consists in the execution of the expression $y = y + Ax$ where y and x are vectors and A is a sparse matrix. The service relies on the `matmul_operator` class provided by the LAKe library. The listing below contains the definition of the abstract service:

```
<?xml version="1.0"?>
<component name="pmv_product" type="abstract">
    <description>
        Parallel matrix vector product y = y + Ax
    </description>
    <params>
        <param name="y"  type="LakeMatrix" mode="inout" />
        <param name="A"  type="LakeCSC"  mode="in" />
        <param name="x"  type="LakeMatrix" mode="in" />
    </params>
</component>
```

Listing A.4: Matrix vector product abstract service definition

This service defines three parameters. Vectors are represented using `LakeMatrix` while the sparse matrix is represented using `LakeCSC`. All these data type are user-defined and based on the LAKe library classes. Vectors are represented using a one column or one line matrix in LAKe. The mode is used to specify whether the parameter is in input, output or both for the service.

The concrete service depends on the LAKe and the zlib libraries. The implementation begins by checking the inputs of the services. In this service, both A and x are required while y is optional. The computation begins line 15. It begins with the creation of a temporary vector to hold the result of the product $y' = Ax$. A LakeCSC matrix provides the apply method which executes the matrix vector product. Finally, we sum the intermediate results. Thanks to the LAKe library the implementation of the service is short.

```
<?xml version="1.0"?>
<component type="impl" abstract="pmv_product" name="pmv_productImpl"
   >
    <impl lang="CXX" libs="lake␣zlib">
        <header />
        <source><![CDATA[

if (! x_param.exists() || ! A_param.exists())
    logError("parameter x or A are missing");

if (! y_param.exists())
{
    y.create(x.shape());
    set(y, 0.0);
}
out() << "Starting computation" << std::endl;
yml::LakeMatrix ytmp(A.nrow(), 1);
A.apply(x, ytmp);
out() << "Matrix operator applied" << std::endl;
// do a sum of x, y
out() << x.shape() << std::endl;
out() << ytmp.shape() << std::endl;
out() << y.shape() << std::endl;
for(int i = 1 ; i <= x.nrow() ; ++i)
    y(i, 1) += ytmp(i, 1) + x(i, 1);

out() << "y = y + Ax finished" << std::endl;
]]></source>
        <footer />
    </impl>
</component>
```

Listing A.5: Matrix vector product concrete service definition

A.2.2 MERAM reduction service

The reduction service is used to select the best set of vectors, generated by the other services. This service illustrates the manipulation of collections from a service written in C++. The reduction service is large in size thus we present only a subset of its implementation focusing on the interaction with collections. In the abstract service definition, a collection parameter is described in the same way as a regular parameter except that it has an extra attribute named 'collection' with the value `yes`. By default, this optional attribute is set to `no`.

```xml
<?xml version="1.0"?>
<yml-query login="admin" password="1010">
  <component name="MeramReduce" type="abstract" description="Reduce␣
      operator␣used␣in␣the␣selection␣of␣the␣restarting␣elements">
    <param name="values"    type="LakeMatrix"  mode="out" />
    <param name="vectors"   type="LakeMatrix"  mode="out" />
    <param name="residuals" type="LakeMatrix"  mode="out" />
    <param name="n"         type="integer"     mode="in"  />
    <param name="r"         type="integer"     mode="in"  />
    <param name="vals"      type="LakeMatrix"  mode="in"  collection=
       "yes" />
    <param name="vecs"      type="LakeMatrix"  mode="in"  collection=
       "yes" />
    <param name="res"       type="LakeMatrix"  mode="in"  collection=
       "yes" />
    <param name="mode"      type="string"      mode="in"  description
       ="Reduction␣mode:␣component,␣process" />
    <param name="eid"       type="integer"     mode="in"  />
  </component>
</yml-query>
```

Listing A.6: MERAM Reduction abstract service definition

The processing of a collection is described in the next listing. Collections are homogeneous: They cannot be composed of different types of resources. A collection is modeled using a tree. Each node can be associated to a resource except the root node. In order to access a particular resource one needs first to access the corresponding node and then can either get or set the resource. Each resource/node is identified uniquely using a sequence of integers which represents a path in the tree. If the layout of the collection is unknown, it is possible to iterate through the nodes of the collection using iterators similar to the standard template library iterators.

The first part of the listing consists in a set of type aliases used to manipulate the collection. In this service, we are interested in selecting the best restarting information from the multiple ERAM processes collaborating to find the solution of an eigenproblem. In order to select this information we first navigate through the three collections which represent the restarting information generated by each ERAM process e_i. In each collection, the element produced by e_i is located at the position i in the collection. Due to asynchronous algorithm, it is possible that the information from an ERAM process be only partially available. The second part of the code is used to collect the valid indices common to the three collections. The last excerpt of this listing uses `CollectionHelper` to navigate through a collection. This class provides a secure interface for the manipulation of a collection and allows the use of the range notation to access elements of the collection directly. The reduction operation is illustrated in this part. It works similarly to an accumulator or the selection of the maximum value in a sequence. `vectors`, `values` and `residuals` contains the temporary results. These results are updated each time the solution provided by an element of the collection is more interesting according to the reduction strategy used.

```
<?xml version="1.0"?>
<component name="MeramReduceImpl" type="impl"
    abstract="MeramReduce">
  <impl lang="CXX" libs="lake">
      <header><![CDATA[
#include <Lake.hh>
#include <Matrix.hh>
#include <vector>
// Definition of type alias for compactness
typedef yml::LakeMatrix Mat;
// A collection is similar to a tree.
// Each node is identified by a sequence of integers
// A vector of integers can be used to identify
// any internal node or leaf uniquely.
typedef std::vector<yml::CollectionIndex> Indices;
// A collection of lake matrices
typedef yml::Collection<yml::LakeMatrix> MatrixCollection;
// A node in a collection of lake matrix
typedef yml::CollectionNode<yml::LakeMatrix> MatrixCollectionNode;
// Resources are associated to nodes which are
//associated to existing data.
typedef yml::CollectionResource<yml::LakeMatrix>
   MatrixCollectionResource;
// Iterators are used to navigate through the nodes of a collection.
typedef yml::CollectionNodeIterator<yml::LakeMatrix>
   MatrixCollectionNodeIterator;
//------------------------ end of first part ------------------------
// First we navigate through the nodes of the collection in
// order to collect the list of nodes associated to a resource
  Indices ids;
  vecs.getRoot()->getIndices(ids);
  out() << "Non checked ids are: ";
  for (size_t i = 0 ; i < ids.size() ; ++i)
        out() << " " << ids[i];
  out() << std::endl;
  for(size_t i = 0 ; i < ids.size() ; ++i)
  {
    // Check consistency : Remove all entries
    // that are not present in all collections
    if (vals.getRoot()->getChild(ids[i]) == 0 ||
        res.getRoot()->getChild(ids[i]) == 0) {
      // We must delete i
      for (size_t j = i ; j < ids.size() - 1 ; ++j)
         ids[j] = ids[j+1];
      ids.pop_back();
      --i;
    }
  }
// ———————————————— end of second part ———————————————————
  // At this stage, everything has been filtered
  // We initialize the results with the value of the first
  // element of the three collections.
  yml::CollectionHelper<yml::LakeMatrix> helperVec(vecs);
  yml::CollectionHelper<yml::LakeMatrix> helperVal(vals);
  yml::CollectionHelper<yml::LakeMatrix> helperRes(res);
  if (! helperVec[ids.back()].get(vectors)) {
    logError("Unable to import data for eigenvectors");
    error();
  }
  if (! helperVal[ids.back()].get(values)) {
    logError("Unable to import data for eigenvalues");
    error();
  }
  if (! helperRes[ids.back()].get(residuals)) {
    logError("Unable to import data for residuals");
    error();
  }
  ids.pop_back();
  // Apply the selection strategy by iterating through
  // the elements of each collection.
  const Indices::const_iterator end = ids.end();
  for(Indices::const_iterator iter = ids.begin() ;
      iter != end ; ++iter) {
    Mat currentRes;
    Mat currentVal;
    Mat currentVec;
    if (helperVec[*iter].get(currentVec) &&
        helperVal[*iter].get(currentVal) &&
        helperRes[*iter].get(currentRes)) {
      // Strategy based on component wise selection
      for (yml::integer i = 1 ; i <= r ; ++i) {
        if (residuals(1, i) > currentRes(1, i)) {
          // Update the final results with the current values.
              residuals(1, i) = currentRes(1, i);
              values(i, 1) = currentVal(i, 1);
              values(i, 2) = currentVal(i, 2);
              for(yml::integer j = 1 ; j <= n ; ++j)
                vectors(j,i) = currentVec(j, i);
          }
      }
    }
  }
```

Listing A.7: MERAM Reduction concrete service definition

A.3 Workflow definitions

The last kind of resources which is needed to define an application using YML is the workflow process. Like the definition of services, a workflow process is described using an XML document. A workflow process is very similar to a service definition: The beginning consists in a set of parameters in the exact same way as for an abstract service. The rest of the definition contains the graph of tasks expressed using the YML graph description language. Several workflow processes have been presented in 8.

We conclude this appendix with a small excerpt issued from the matrix vector product application. It describes the parallel section corresponding to the product. It illustrates the most common construction of the language: the parallel and sequential loops, the synchronization events `wait` and `notify` and the execution of services introduced by the keyword `compute`. The workflow is described in 8.3.1.4. 1In the excerpt below, the A matrix is square and split in square blocks. The event `evtX` are notified in another section (not present in the listing) of the workflow of the application executed in parallel.

```
par (i := 1 ; blockCount) do
    wait(evtX[i]);
    seq (j := 1 ; blockCount) do
        wait(evtX[j]);
        # check this first
        compute pmv_product(y[i], A[i][j], x[j]);
    enddo
    notify(evtY[i]);
enddo
```

Listing A.8: Matrix vector product

Résumé

Les intergiciels à grande échelle proposent des solutions aux problèmes d'accès à distance, de sécurité, d'hétérogénéité et de tolérance aux fautes des participants au système. L'exploitation de ces intergiciels par des utilisateurs finaux reste très difficile pour des applications réelles. Les environnements de développements et d'exécution pour ces systèmes sont très rares voire inexistants.

Nous présentons YML, un environnement de calcul scientifique à base de workflow pour les systèmes distribués. L'objectif principal de YML est de rendre l'utilisation d'intergiciel transparente à l'utilisateur. Il s'agit d'une couche logicielle située entre l'intergiciel déployé et les applications. YML est composé de trois parties: les interfaces utilisateurs, le noyau et l'interface avec les intergiciels. Le noyau de YML comprend un langage de workflow utilisé pour décrire l'application ainsi qu'un compilateur et un ordonnanceur de tâches. Le noyau possède également un service d'intégration de systèmes permettant d'intégrer les bibliothèques scientifiques en tant que service de calcul.

Nous validons l'approche choisie et présentons une évaluation préliminaire de la performance d'YML par le biais de quatre applications issue de trois domaines distincts. Nous focalisons l'analyse sur deux cas d'études : une application de tri et une méthode d'algèbre linéaire de résolution des problèmes de valeurs propre appliquée aux matrices creuses non Hermitienne. Nous présentons une méthode hybride asynchrone de résolution appelée MERAM et analysons les résultats obtenus en utilisant la bibliothèque LAKe (Linear Algebra Kernel) intégrée au sein d'YML en tant que services.

Abstract

Distributed large-scale middleware aim at solving problems such as remote access, security, heterogeneity, and fault tolerance. Using these middleware with real life applications is still complex and not easy for end-users. The adapted programming and execution environments built upon middleware services still lack.

We present YML, a scientific workflow environment for the development and execution of parallel applications on distributed architectures. It aims at making the use of such architectures transparent and independent of middleware. It is a software layer located on top of middleware of a widely distributed system. It consists of three overlapping parts: the front-end as the user interface, the kernel and the back-end as the interface with middleware. The core of YML includes a workflow language used to express and run the application and provides a compiler and a scheduler for it. The kernel supports also a service integration system offering the possibility to integrate the scientific libraries as services.

We validate our approach and evaluate the performance of the implementation through four applications from three different domains. We emphasize our analysis on two applications: a distributed sort and a linear algebra solver for eigenproblems applied to non Hermitian sparse matrices. We present an asynchronous implementation of MERAM, a hybrid iterative method and discuss the results obtained using the Linear Algebra Kernel library as a service integrated in YML.