Project no. FP6-004265

CoreGRID

European Research Network on Foundations, Software Infrastructures and Applications for large scale distributed, GRID and Peer-to-Peer Technologies

Network of Excellence

GRID-based Systems for solving complex problems

D.PM.06 Programming models for the single GCM component: a survey

Due date of deliverable: September, 30, 2006
Actual submission date: November 21, 2006

Start date of project: 1 September 2004 Duration: 48 months

Organisation name of lead contractor for this deliverable: HES-SO / EIA-FR

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)

Dissemination Level

PU Public PU

Keyword List: Parallel and distributed programming, Grid programming languages and tools, Grid Components.

Contributors: Pierre Kuonen (HES-SO/EIA-FR), Francoise Baude, Ludovic Henrio, Christian Perez, Matthieu Morel (INRIA), Marco Danelutto, Sonia Campa (UNIPI), Rob van Nieuwpoort (VUA), Jan Duennweber (U Münster), Martin Griebl (Uni-Passau), Peter Kilpatrick (Belfast U.), Rosa M. Badia (UPC), Philipp Wieder (fz-juelich).

Reviewers:

Approved by:

Version	Date	Authors	Sections Affected
1.1	2.11.06	P. Kuonen (HES-SO/EIA-FR) from reviewer comments	All

Executive Summary

The objective of this document is to identify and to characterize programming languages and tools suitable to implement single Grid components compliant with the GCM (Grid Component Model). The deliverable D.PM.02 - *Proposals for a Grid Component Model* – describes the Grid Component Model (GCM) proposed by the CoreGRID NoE. Indeed, one of the main objectives of the Institute on programming models is to deliver a definition of a component programming model to be used to design, implement and run high-performance, efficient Grid applications. In this context, the Fractal model has been chosen as the reference model and has been extended mainly to better address the following issues: virtual nodes, collective interfaces, dynamic controllers, and autonomic components. Main features of the GCM, as presented in detail in D.PM.02, are summarized in section 2 of this document.

In section 3 we propose a method for the characterization and the classification of programming models suitable for implementing a single Grid component compliant with GCM. This high-level characterization is based on the definition of a set of characteristics allowing comparison and identification of general classes of languages. Eight characteristics have been identified and for each characteristic a set of possible value has been defined.

The objective of section 4 is to present the main languages developed inside the CoreGRID NoE and suitable for implementing a single Grid component compliant with GCM. To ease reading and comparison of the selected languages, a standardized way for their presentation has been defined. For each language in addition to a short general presentation, information as availability, source of documentation or developing team are indicated.

In section 5 all the languages presented in section 4 plus some broadly known or used languages are characterized using a “table approach”. This table indicates for each language the corresponding values for each of the characteristics defined in section 3. This allows for a very synthetic view of the languages and considerably eases their comparison and classification.

Section 6 concludes this document and presents future envisioned activities of the task “Basic programming language” of the Institute on programming models.

TABLE OF CONTENT

1. Introduction

One of the main objectives of the Institute on programming models is to deliver a definition of a component programming model that can be usefully exploited to design, implement and run high-performance, efficient Grid applications. The same component model should also be exploited in the design of tools supporting Grid programming, such as in the development of PSEs[1] or in the development of tools supporting resource management or system architecture related activities. Deliverable D.PM.02 - *Proposals for a Grid Component Model* – describes the Grid Component Model (called GCM) proposed by the CoreGRID NoE. The proposed GCM is briefly summarized in section 2 of this document.

As it is mentioned in sub-section 1.2 - Challenges, requirements and Main Characteristics of the GCM - of document D.PM.02: "*a suitable programming model (that is user friendly and efficient) to program individual components is needed*" A single component's structure can vary from being a very simple sequential piece of code to a very complex parallel program designed to run on a specific target platform, e.g., a Grid middleware. As a consequence, a large range of different programming models to implement single components can be used to address the diversity of Grid components.

In section 3 of this document we identify, characterize and classify programming models suitable for implementing a single Grid component compliant with GCM. In section 4 we present a survey of some existing or under development programming languages and tools which can be used for the implementation of a single Grid component. In section 5 we classify, using a table, different programming languages and tools regarding models defined in section 3. Section 6 is the conclusion.

2. Definition of a component according to GCM

This section briefly summarizes the deliverable *D.PM.02 - Proposals for a Grid Component Model* which defines a Grid Component model (called GCM) proposed by the CoreGRID NoE.

The main features to be included in the GCM have been assessed in the Programming Model Institute. By defining the GCM, the Institute aims at the precise specification of an effective Grid Component Model.

The features are discussed taking Fractal as the reference model. The features are defined as extensions to the Fractal specification in order to better target the Grid infrastructure: mainly virtual nodes, collective interfaces, dynamic controllers, and autonomic components.

2.1 A Programming Model for the Grid

It is assumed that the component based programming model's main aim is to address the new characteristic challenges of Grid computing - heterogeneity and dynamicity - in terms of programmability, interoperability, code reuse and efficiency. Grid programmability, in particular, represents the biggest challenge. Grid programs cannot be constructed using traditional programming models and tools (such as those based on explicit message passing or on remote procedure call/web service abstraction, for instance), unless the programmer is prepared to pay a high price in terms of programming, program debugging and program tuning efforts.

New programming models are required that exploit a layered approach to Grid programming which will offer user friendly ways of developing efficient, high performance applications. This is particularly true in cases where the applications are complex and multidisciplinary. Within CoreGRID, the challenge is to design a component-based programming model that overcomes the major problems arising when programming Grids.

The challenge, per se, requires that a full set of sub-challenges be addressed:

- A suitable programming model (that is user friendly and efficient) to program individual components is needed.

[1] Problem Solving Environment

- Component definition, usage and composition must be organized according to standards that allow interoperability to be achieved.
- Component composition must be defined precisely in such a way that complex, multidisciplinary applications can be constructed by the composition of building block components, possibly obtained by suitably wrapping existing code. Component composition must support and, in addition, guarantee scalability.
- Semantics must be defined, precisely modelling both the single component semantics and the semantics of composition, in such a way that provably correct transformation/improvement techniques can be developed.
- Performance/cost models must be defined, to allow the development of tools for reasoning about components and component composition programs

All of these sub-challenges must be dealt with taking into account that improvements in hardware and software technology require new Grid systems to be transparent, easy to use and to program, person centric rather than middleware, software or system-centric, easy to configure and manage, scalable, and suitable to be used in pervasive and ubiquitous contexts.

The GCM is the component model adapted for the Grid that the Programming Model Institute proposes. It should possess some basic and essential characteristics.

- *Support for reflection*: Both introspection and intercession;
- *Hierarchical structure*: Facilitate the identification of architectural units, thereby facilitating the design of complex systems. Grid systems are usually composed of many different and heterogeneous software units. Each software unit may itself integrate other software units. For example a unit responsible for data mining may itself be constituted of several units running on a given cluster, and this data-mining unit may itself be part of a larger system, like a weather prediction system;
- *Model Extensibility*: The model should include placeholders to allow for new features to be added to components and the component model;
- *Language neutrality*: GCM should be specified in a programming language independent way;
- *Support for adaptivity*: providing automatic ways of adapting running GCM programs to the features required/provided by the target Grid and component architecture;
- *Interoperability*: GCM should allow components to export and import functionalities from other frameworks; for instance, one should be able to use a Web service or a GCM component from within the GCM as well as being able to export a Web Service port for any GCM component.;
- *Support of MxN communications*: GCM should support parallel-to-parallel component communications. It appears very important to achieve HPC and to efficiently embed legacy parallel codes such as those based on MPI.

Additionally the model should be *lightweight* in order to allow compact and portable implementations to be designed, and have a *well defined semantics*. Moreover, the GCM should take into account the necessity for its implementations to ensure high performance.

This component model must be suitable both for implementing Grid applications and Grid platforms themselves, with both of them benefiting from having the above features. For example, adaptivity is a key issue for programming Grid applications that can be deployed on heterogeneous environments, but this also means that Grid platforms will themselves be deployed and have to manage heterogeneous systems; consequently such platforms would necessitate an even stronger support for adaptivity than the applications themselves.

2.2 Synthesis of the GCM proposed in D.PM.02

The proposal for the definition of the GCM includes the following aspects:

- *GCM is based on Fractal component architecture*: highly extensible component model which enforces separation of concerns (*content* and *controller*), and separation between interfaces and implementation (*Server versus Client interface, Functional versus non-functional or Control interface*).

- *Abstract Component Model and Architecture*: abstract view of the Grid Component Model. The following architecture is proposed for concretely defining the GCM:
    1. Component Specification as an XML schema or DTD
    2. Run-Time API defined in several languages
    3. Packaging described as an XML schema
- *Communication Semantics Standards*: GCM components include various communication semantics; however *asynchronous method invocation* is the default case.
- *Deployment of components may rely on Virtual Nodes*: Virtual Nodes are used in the code or in the ADL and abstract away names and creation and connection protocols to physical resources, from which applications remain independent. Virtual Nodes are optional; they can be single or multiple in order to represent a single resource or a set of resources.
- *Multicast and Gathercast Interfaces*: Multicast and gathercast interfaces give the possibility to manage a group of interfaces as a single entity, and expose the collective nature of a given interface.
- *Component Controllers*: Possibility to consider a controller as a sub-component, which can then be added, plugged or unplugged dynamically within the component it controls. This approach gives a better adaptivity, both with respect to the platform, and with respect to the controlled components.
- *Autonomic Components*: The GCM defines four autonomic aspects, and it gives a precise interface for each of these four aspects. These interfaces are non-functional and may be exposed by each component: for self-configuration, self-optimization, self-healing or self-protection of the component. They correspond to Fractal controllers. Consequently, they may, as any other controller, be implemented using objects or sub-components (see Component Controllers above) for a better adaptivity.

3. Classification of programming models

Section 2 briefly summarizes the definition of the GCM proposed in D.PM.02. In this section we propose to identify, to characterize and to classify programming models and tools suitable for programming single Grid components. We will remain at a high level of abstraction by identifying a set of characteristics for which a set of possible values are proposed. The main purpose of this approach is to allow comparison and classification of existing programming languages and tools. As single GCM components can also be programmed using programming languages or tools compliant with GCM or any other component model, the proposed characteristics should allow distinction of component based models from non-component based models. Nevertheless, as this classification does not specifically target component-based models, proposed characteristics will not allow specification in detail of component models' characteristics.

In order to ease understanding we will, as far as possible, mention existing programming languages or tools that illustrate characteristics and values we are presenting. Nevertheless, a more complete survey of existing programming languages and tools suitable for implementing a single GCM component is presented in sections 4 and 5.

3.1 List of proposed characteristics and values

Bellow we list and explain the proposed characteristics allowing identification of classes of programming models for single GCM component. For each characteristic we indicate the possible values. In some case the mentioned characteristic is not applicable to the programming model or tool. In such cases the value *NA* (Non Applicable) will be given for this characteristic.

- The Language style (*style*) characterizes the main programming paradigm the programming language or tool is based on. Usually a programming language or tool which offers a programming paradigm also offers "lower level" paradigms. For example, object oriented languages usually also contained the notion of block. For the *style* characteristic only the highest offered paradigm will be indicated. The possible values for the style characteristic is listed below in level increasing order:
    1. Block oriented (*block*) such as Fortran, Pascal or C programming languages.
    2. Object Oriented (*object*) such as Java, C# or C++.

3. Pattern oriented (*pattern*) corresponds to cases in which the application is built by composing "black-box" modules or patterns of modules such as CO2P2S (Correct Object-Oriented Pattern-based Programming System).
4. Skeleton oriented (*skeleton*) such ASSIST.
5. Component oriented (*component*) which, additionally to services makes it explicit which are the services used by a given service.

- The control flow (*flow*) characterizes the fact that the programming paradigm offers single or multiple flow of control. Single flow of control corresponds to traditional sequential languages; all other values are multiple flows. Single flow of control is mutually exclusive with all other possible values for the *flow* characteristic. However several values can be attributed to this characteristic if the flow of control is multiple. Possible values for this characteristic are listed below:
  1. *Sequential* corresponds, as mentioned above, to sequential languages (C/C++, Fortran,...)
  2. *Concurrent* corresponds to "intra node multi-tasking" such as thread in Java
  3. *Implicit* parallelism
  4. *Synchronous* parallelism corresponds to block synchronous like paradigms (BSP) [1].
  5. *Parallel* corresponds to the SPMD programming style (data parallelism e.g. MPI).
  6. *Distributed* corresponds to the "task parallelism" paradigm such as POP-C++ or ProActive.
- *Communication* model characterizes the communication paradigm. This characteristic is not applicable (*NA*) to sequential (single flow of control) programming models. One or several of the listed values are applicable:
  1. Shared variables (*Shared*) with or without mutual exclusion mechanism
  2. Rendezvous (*Sync*) such as in ADA
  3. Message passing (*MP*) such as MPI or PVM
  4. Remote procedure call (*RPC*) such as in Corba, ProActive or POP-C++
  5. *Global* communication such as broadcast, multicast, gather, etc…
- *Execution* type characterizes the way processes are executed on the Grid nodes. The possible values are:
  1. *Compiled*
  2. *Interpreted*
  3. *Virtual* machine
  4. Just in time compiled (*JIT*)
- Application *deployment* characterizes the fact that the complexity of the Grid is exposed or not to the programmer.
  1. *Low level* : programmer is largely exposed to the complexity of the Grid.
  2. *High level* : the complexity of the Grid is largely hidden from programmers.
- Specific *supports* lists the specific support embedded in the programming language or tool. None or several of the listed values are applicable:
  1. *Fault tolerance*
  2. *HPC*
  3. *Security*
  4. *Portability*
- *Resource* management lists the specific embedded features allowing explicit management of resources from within the program. None or several of the listed values are applicable.
  1. Resources *discovery*
  2. Process *migration*
  3. *Remote* launch of a piece of code on a remote resource
  4. Resources and/or process *monitoring*
- *Standardization* characteristic indicates if programming language or tools is standardized (ISO, ANSI,…)
  1. *Yes*
  2. *No*

3. In *progress*

Using the above proposed list we can, by fixing values for some given characteristics[2], identify some general classes of programming languages or tools. Below are some examples of these classes.

- Standard "old fashioned" sequential programming languages such as Fortran, C or Pascal are characterized by:
    1. Style = block
    2. Flow = Sequential
    3. Communication = NA
    4. Deployment = Low level
    5. Resource = NA
- Commonly used or known HPC parallel tools or languages for HPC (MPI, HPF,…) are characterized by:
    1. Style = Block
    2. Flow = Parallel
    3. Communication = Global (at least)
    4. Deployment = Low level
    5. Resource = None
- Distributed object oriented tools such as Corba, ProActive or POP-C++ are characterized by:
    1. Style = Object
    2. Flow = distributed
    3. Communication = RPC (at least)
    4. Resource = Remote (at least)

In section 5 we present a table for the classification of the best known programming languages and tools and for the major programming languages and tools currently developed by research teams within the CoreGRID NoE.

4. Survey of some programming languages and tools

This section is dedicated to the presentation of programming languages or tools commonly used in the Grid community as well as major programming languages and tools developed or currently under development in the CoreGRID NoE. The objective is to give, for these tools, more information and references than the characteristics listed in the previous section. For all the programming languages and tools presented in this section, values for characteristics defined in section 3 are given in the table presented in section 5. To facilitate reading and comparison between the presented tools, the way in which the tools are presented has been standardized. Presentations are made using the following rubrics:

- ***Title:*** the name and acronym of the tool
- ***Origin***: where and when the tool was originally developed and optionally some information on the history of the tool.
- ***Availability:*** Indication on how the product is available (the type of license, is it a commercial product or an academic prototype, etc..).
- ***Main characteristics:*** Main technical characteristics of the tool.
- ***References***: some bibliographic or web references for the tool.
- ***Presentation***: a short free text to describe the tool.

POP-C++ : Parallel Object Programming with C++

Origin: PhD thesis of Tuan Anh Nguyen presented in 2003 at EPFL, Switzerland. POP-C++ is still maintained and developed at HES-SO/EIA-FR (University of Applied Sciences, Fribourg).

[2] Values of other characteristics can vary from one language or tool to another

Availability: Open source free tool. Latest version of the tool is available at www.eif.ch/gridgroup/popc or on the SourceForge web site

Main characteristics: Object oriented distributed programming language, based on original remote method invocation semantics.

References: http:// www.eif.ch/gridgroup/popc, [2]

Presentation:. The POP-C++ programming system has been built to provide Grid programming facilities which greatly ease the development and the deployment of parallel applications on the Grid. The model is based on the simple idea that objects are suitable structures to encapsulate and to distribute heterogeneous data and computing elements over the Grid. POP-C++ language is an extension of C++ that implements the POP model (POP stands for Parallel Object Programming) with the integration of resource requirements into distributed objects. POP-C++ programming model introduces a new type of object: *the parallel object*. Parallel objects coexist and cooperate with sequential objects during the application execution. Parallel objects in POP-C++ generalize the sequential object concept by keeping the advantages of object-orientation such as data encapsulation, inheritance and polymorphism and by adding new properties to the object such as:

- Distributed shareable objects
- Dynamic and transparent object allocation driven by the high-level requirement descriptions
- Various method invocation semantics

Although POP-C++ programming system focuses on an object-oriented programming model for the Grid it also includes a runtime system which is responsible for providing necessary services for running POP-C++ applications. The POP-C++ runtime system consists of three layers: the service layer, the essential service abstractions layer, and the programming layer. The service layer is built to interface with lower level grid middleware (e.g. Globus). The essential service abstractions layer provides an abstract interface for the programming layer. On top of the architecture is the programming layer, which provides necessary support for developing grid-enabled object-oriented applications. With POP-C++, programmers can guide the resource allocation for each object through the high-level resource descriptions. The object creation process, supported by the POP-C++ runtime system, is transparent to programmers. Both inter-object and intra-object parallelism are supported thanks to the various method invocation semantics.

ASSIST : A Software development System based on Integrated Skeleton Technology

Origin: The programming environment and the coordination language have been designed in a joint project ASI/CNR (Italian National Space Agency and Italian National Research Council), by people of the Dept. of Computer Science of Pisa. Version 1.2.1 is available (stable).

Availability: ASSIST has been developed inside an academic environment enforced by industrial interests (e.g. the Italian National Space Agency). Nowadays, it's copyrighted by a GNU licence, as well as the code automatically generated by the compiler.

Main characteristics: ASSIST is a programming environment including a skeleton based parallel programming (imperative) language (ASSISTcl, cl stands for coordination language) and a set of compiling tools and run time libraries.

References: http://www.di.unipi.it/groups/architetture/

Presentation: The framework provides functions for both asynchronous and synchronous parallelism (through event-driven and RPC-like calls), external objects linkage (e.g. CORBA objects) and it supports the reuse of C, C++ and FORTRAN existing code. Different parallelisation strategies can be experimented with by just changing a few lines of code and recompiling. The compilation tools and runtime support provided by the current distribution take care of process and communication setup, scheduling, mapping.

HOC-SA : Higher-Order Component Service Architecture

Origin: University of Muenster (Jan Duennweber, Sergei Gorlatch et al.), since 2004

Availability: academic, free

Main characteristics: supports Java, C, C++ scripting

References: http://pvs.uni-muenster.de/pvs/forschung/hoc/index-en.html, concept explained in Springer CoreGRID series [7], implementation described in [8]

Presentation: HOCs are designed with the aim to provide skeletons as Grid-enabled components. The HOC-SA is a runtime environment for HOCs. The current implementation runs on top of Globus and consists mainly of two elements: the Code Service and the Remote Code Loader. Via the Remote Code Loader, Web services and Web service consumers can upload and download pieces of executable code from the Code Service, which is a Grid repository (i.e., a database accessible via a Web service) for sharing code among distributed servers. Thus, the HOC-SA extends the parameter type range of Web services, as it allows defining Web services which take parameters carrying executable code. A HOC is a component (typically a skeleton implementation, which can be based on Java, C+MPI or any of the component architectures proposed by other CoreGRID partners), which is remotely accessible (via a Web Service) and which takes one or more code parameters.

LooPoPM : LOOp parallelization in the POlyhedron model as a Programming Model

Origin: The LooPo started in 1994 at the University of Passau. Project responsible: Martin Griebl.

Availability: academic, free for academic usage; parts under GNU public license. No official release, but any interested persons may directly contact loopo@fmi.uni-passau.de.

Main characteristics: source-to-source loop parallelizer. The purpose is to implement and to test advanced methods in automatic loop parallelization.

References: www.fmi.uni-passau.de/~loopo

Presentation: The input to LooPo is either an imperative loop program in (a subset of) C or FORTRAN, or a specification with affine recurrence equations. The output in the original project is HPF or C/MPI or C/OpenMP. We can generate synchronous as well as asynchronous parallelism. The next goal is load balancing in heterogeneous and dynamic environments. With a CoreGRID fellow, we also plan to automatically generate the application-specific code for existing parallel (higher-order) components, e.g., the task farming component of the University of Muenster. All GRID-specific activities (deployment, resource discovery,...) are then managed by the target component system – LooPo becomes an internal part of a generalized task farming component.

GAT : The Grid Application Toolkit

Origin: The GAT was developed within the European Union GridLab project. The project started in 2003, and ended in 2005. It was assessed as one of the best FP5 projects. The GAT development continues outside of GridLab. The Java version of the GAT is developed by the Vrije Universiteit Amsterdam, as a part of the Ibis project.

Availability: Academic, free for all usage, BSD-style license. Releases are made available regularly.

Main characteristics: high-level, middleware independent grid programming model.

References: www.gridlab.org

Presentation: GAT is a set of coordinated, generic and flexible APIs for accessing Grid services from e.g. generic application codes, portals, data managements systems. GAT is designed in a modular plug-and-play manner, such that tools developed anywhere can be plugged into GAT.

GAT lifts the burden of grid application programmers by providing them with a uniform interface to numerous types of grid middleware. As a result, grid application programmers need only learn a single API, that of GAT, to obtain access to the entire grid. The GAT is available in many programming languages: C, C++, Java, Perl, Python, etc.

Ibis

Origin: Ibis was developed at the Vrije Universiteit Amsterdam. The project started in 1998, and is still ongoing. Project responsible: Rob van Nieuwpoort (rob@cs.vu.nl).

Availability: Academic, free for all usage, BSD-style license. Releases are made available regularly.

Main characteristics: Supports low-level communication operations (point-to-point and broadcast) and several high-level programming models. Java-based.

References: www.cs.vu.nl/ibis

Presentation: The Ibis communication library is specifically designed for usage in a grid environment. Its run-everywhere property and support for fault-tolerance, malleability[3] and high-speed networks, make it an easy to use and reliable grid communication infrastructure.

Ibis also provides several higher level programming models, such as a highly efficient RMI implementation, a method-oriented group communication model (GMI), A divide-and-conquer framework (Satin), and a Java implementation of MPI (MPJ).

Satin

Origin: Satin was developed at the Vrije Universiteit Amsterdam. The project started in 1998, and is still ongoing. Project responsible: Rob van Nieuwpoort (rob@cs.vu.nl).

Availability: Academic, free for all usage, BSD-style license. Released as a part of the Ibis project.

Main characteristics: Supports master/worker and divide-and-conquer programming on the grid. Java-based.

References: www.cs.vu.nl/ibis

Presentation: Satin supports master/worker and divide-and-conquer programming on the grid. Satin also provides shared objects, and a mechanism for speculative parallelism. Satin is written in Java, so it runs on any grid site with a JVM. Satin features completely transparent fault-tolerance and a grid-aware load-balancing mechanism.

ProActive

Origin: ProActive was developed by the OASIS research team at the University of Nice-Sophia-Antipolis, CNRS I3S, INRIA. The project started in 1998, and is still ongoing. Project responsible: Denis.Caromel@inria.fr

Availability: Academic, free for all usage, LGPL license. Project hosted by the ObjectWeb consortium for Open Source Middleware.

Main characteristics: Supports distributed Java objects (possibly wrapping native codes such as e.g. C MPI ones) and component programming on any sort of support (single machine, multi-processors, clusters, LAN, grids)

References: proactive.objectweb.org

Presentation: ProActive is a distributed programming model and corresponding APIs, that is particularly relevant for wide-area distributed infrastructures programming such as computing grids. Following an active object pattern, it enables to address parallelism, distribution, concurrency in a uniform way. More specifically, ProActive programming relies on Asynchronous calls: typed messages (request and reply); Automatic future-based synchronizations : wait-by-necessity; Remote creation of remote objects; Distributed and non-functional exceptions handling; Transparent, dynamic code loading (up and down); Reuse: polymorphism between standard objects and remote objects; Group communications with dynamic group management grounding parallel or distributed programming patterns such as Object Oriented SPMD, Mobile agents. The model is thus general purpose, and as such can be used to provide higher-level APIs for structured

[3] We define malleability as: the ability to cope with dynamically changing number of processors within an application run.

parallelism (a.k.a..skeletons), expressing master/worker, divide-and-conquer or any sort of task parallelism based programs for the grid. ProActive is entirely written in Java and relies on Meta programming techniques for hiding distribution. The ProActive platform features are:

- completely transparent fault-tolerance: the set of active objects constituting the application is regularly check pointed, so that in case of failure, the application can be automatically restarted on non-faulty nodes;
- transparent secured communications: first security policies regarding authentication, non-repudiation, encryption of data between the various sites forming the grid, can be expressed in a declarative way outside the application; then, depending on the effective location of active objects on the grid, the required policy if any is transparently applied to all inter-sites communications.
- a portable way to transfer files and acquire computing resources and let them be accessible from within the running program through virtual nodes,
- a self-organized set of ProActive daemons acting as a peer-to-peer computing grid onto which any ProActive application can be deployed and run, while being automatically load-balanced (i.e. migration of active objects can be triggered in order to load balance the load of the JVMs they run onto).

PaCO++: Parallel CORBA Objects

Origin: PhD thesis of André Ribes presented in 2004 at Université of Rennes, France. PaCO++ is still maintained and co-developed by INRIA and EDF R&D (France).

Availability: LGPL for the runtime and GPL for the compiler. Latest version of the tool is available at http://paco.gforge.inria.fr/

Main characteristics: Parallel distributed object oriented based on parallel remote invocation semantics.

References: http://www.irisa.fr/paris/Paco++, [9]
http://www.irisa.fr/paris/Biblio/Papers/Perez/PerPriRib03IJHPCA.pdf

Presentation: PaCO++ applies the notion of parallel and distributed entities to CORBA objects: it provides parallel CORBA objects. A parallel CORBA object is a CORBA object whose execution model is parallel. It is accessible externally through a CORBA reference whose interpretation is identical to a standard CORBA object. The objectives of PaCO++ are to allow a simple and efficient embedding of a SPMD code into a parallel CORBA object and to allow parallel communication flows and data redistribution during an operation invocation on such a parallel CORBA object, that is to say to support MxN communications: from a CORBA object running an SPMD code with M processes, towards an other CORBA object running an other SPMD code with N processes .

PaCO++ extends the CORBA specifications but does not modify the model because it is defined as a portable extension to CORBA so that it can be implemented on top of any CORBA implementation. This choice stems also from the consideration that the parallelism of an object appears to be an implementation issue of the object. Thus, the OMG IDL is not required to be modified.

PaCO++ is made of two elements: a compiler and a runtime library. The compiler generates parallel CORBA stub and skeleton from an IDL file which describes the CORBA interface and from an XML file which describes the parallelism of the interface. The runtime deals with the parallelism of the parallel CORBA object. It is very portable thanks to the utilization of abstract APIs for communications, threads, redistribution and communication scheduling libraries.

GS: Superscalar

***Origin*:** The first prototype version was developed in 2002 by Raul Sirvent (CEPBA-UPC, Barcelona, Spain), as his master thesis project, using Condor (MW) as middleware. We switched to Globus Toolkit V2.4 at the end of the same year 2002, since Condor limited the possibility of

dynamic scheduling. The system runs now on top of GT2, GT4, Ninf-G and ssh/scp (this later version is tailored for clusters, specially for MareNostrum supercomputer). Besides, there is a new version under development, where the code has been componentized and re-structured and offers new functionalities. This second version has been extended to implement a programming model for multicore processors, especially Cell BE processors (Cell superscalar). Since year 2005 it is maintained and further developed at the Barcelona Supercomputing Center (since the research of CEPBA has been moved to this center).

Availability: Binaries are distributed under request with no economical charge. The possibility of offering as open source is currently being evaluated at BSC.

Main characteristics: Grid-unaware programming environment. The first version is based on code generation and a small user API, and offers two graphical tools for application automatic deployment and monitoring. The second version is based in a source to source compiler based on code annotations (similar to OpenMP, for example). In both cases, main features are performed by a run-time library.

References**:** http://www.bsc.es/grid/grid_superscalar, [3-6]

Presentation**:** Grid Superscalar is a Grid-unaware application framework focused to scientific applications. The definition of Grid-unaware applications in the framework of Grid superscalar are those applications where the Grid (resources, middleware) is transparent at the user level, although the application will be run on a computational Grid. The key for Grid superscalar applications is the identification of coarse grain functions or subroutines (in terms of CPU consumption) in the application. Once these functions or subroutines (called tasks in the Grid Superscalar framework) are identified, the Grid superscalar system is able to detect at runtime data dependencies and the inherent concurrency between different instances of the tasks. Therefore, a data-dependence task graph is dynamically built, and tasks are executed on different resources on the Grid. Whenever possible (because data-dependencies and available resources allow) different tasks are executed concurrently, increasing application performance. Other techniques, as file renaming, are applied to increase the parallelism of the task graph and therefore potentially improve the performance of the application.

Input applications languages currently supported are: C/C++, Perl, shell script and Java. Grid superscalar distribution is composed of a graphic interface for Grid configuration and automatic application deployment, a set of binaries for code generation and a runtime library. OpenMP

5. Table of classification

This section presents a table which indicates values of the characteristics defined in the section for programming languages and tools currently used in the Grid community as well as for all languages and tools presented in the section 4.

		Style	*Flow*	*Communication*	*Execution*	*Deployment*	*Support*	*Resource*	*Standardization*
1	**Fortran**	Block	Sequential	NA	Compiled	Low Level	HPC	NA	Yes
2	**C**	Block	Sequential	NA	Compiled	Low Level		NA	Yes
3	**C++**	Object	Sequential	NA	Compiled	Low Level		NA	Yes
4	**JAVA**	Object	Concurrent	Shared	Virtual, JIT	Low Level		NA	Yes

		Style	*Flow*	*Communication*	*Execution*	*Deployment*	*Support*	*Resource*	*Standardization*
5	**MPI**	Block	Parallel	MP, Global	Compiled	Low Level	HPC		Yes
6	**OpenMP**	Block (Fortran & C) Object (C++)	Distributed, Concurrent,	Shared	Compiled	Low Level	HPC	NA	Yes
7	**POP-C++**	Object	Distributed, Concurrent	RPC, Shared	Compiled	High Level	HPC	Discovery, Remote	No
8	**HOC-SA**	Component	Distributed	RPC, Global	Virtual	High Level	HPC, Portability	Discovery, Remote, Monitoring	No
9	**ASSIST**	Skeleton	Parallel	RPC, MP, Shared	Compiled	Low Level	HPC, Adaptivity	Remote	No
10	**ProActive**	Object	Distributed	RPC	Virtual	High Level	Fault tolerance, Security, Portability	Discovery, Migration, Remote	No
11	**GAT**	Object	Distributed	MP, Global	Compiled, JIT	High Level	HPC, Security, Portability	Discovery, Remote, Monitoring	Progress
12	**Ibis**	Object	Concurrent, Distributed, Parallel, Implicit	Shared, MP, RPC, Global	Compiled, JIT	High Level	Fault tolerance, HPC, Portability	Discovery, Remote, Monitoring, Migration	No
13	**Satin**	Object	Parallel, Distributed	Shared, Global	Compiled, JIT	High Level	HPC, Portability	NA	No
14	**Superscalar**	Object	Distributed	Implicit	Compiled	High-Level	Fault-tolerance, HPC	Remote launch, monitoring	No
15	**PaCO++**	Object	Distributed, Concurrent, Parallel	RPC	Compiled	Low Level	HPC, Portability	Remote	No
16	**LooPoPM**	Block	Parallel, Implicit	Shared, MP, Global	Compiled	High-Level	HPC, Portability		No

6. Conclusion

In the table presented in section 5, sixteen different languages and tools have been characterized. On these sixteen tools, ten (line 7 to 16) have been developed or are still under development in CoreGRID partner's institutions. All these tools provide support for parallel programming based, at least, on the distributed parallel programming paradigm (task parallelism). The object oriented programming style is the most common one and the very large majority of these ten tools offer High-Level application deployment feature. Unfortunately none of these ten tools are currently standardized. Only GAT, which is the result of another European project, has an on-going process for standardization.

The objective of the task 3.1 of the Programming Model Institute (WP3) of CoreGRID is to investigate languages and tools for programming single GCM components. In this deliverable we identified major tools and languages currently offered by CoreGRID partners to fulfill this task. Standardization of these tools is under the responsibility of the developing institutions and we do consider that the CoreGRID NoE does not have to choose one particular tool or language. Indeed experience has shown that programming languages cannot be imposed by any institution because different user communities choose different programming tools. This process of adopting a particular programming tool is a competing process and is largely driven by non-technical considerations. Therefore the next step of work for the task 3.1 will constitute in proposing, for each or at least for a large part of these ten tools guidelines and cases study demonstrating how, using the considered tool, single GCM components can be efficiently implemented.

7. References

[1] David B. Skillicorn, Jonathan M. D. Hill and W. F. McColl, “Questions and answers about BSP" Scientific Programming, 6(3): 249-274, Fall 1997

[2] “Programming the Grid with POP-C++”, T. A. Nguyen, P. Kuonen, Future Generation Computer Systems, Elsevier, 2006 (accepted, to be published).

[3] Rosa M. Badia, Jesús Labarta, Raül Sirvent, Josep M. Pérez, José M. Cela and Rogeli Grima, “Programming Grid Applications with GRID Superscalar”, Journal of Grid Computing, Volume 1, Issue 2, 2003.

[4] Vasilis Dialinos, Rosa M. Badia, Raül Sirvent, Josep M. Pérez and Jesús Labarta, “Implementing Phylogenetic Inference with Grid superscalar”, Cluster Computing and Grid 2005 (CCGRID 2005), Cardiff, UK, 2005.

[5] Raül Sirvent, Josep M. Pérez, Rosa M. Badia, Jesús Labarta, Automatic Grid workflow based on imperative programming languages, Concurrency and Computation: Practice and Experience, Volume 18, Issue 10, 2006. Pages: 1169-1186.

[6] Rosa M. Badia, Raul Sirvent, Jesus Labarta, Josep M. Perez, “Programming the GRID: An Imperative Language-based Approach”, book chapter in “Engineering The Grid: Status and Perspective”, American Scientific Publishers, 2006.

[7] Sergei Gorlatch and Jan Dünnweber. From Grid Middleware to Grid Applications: Bridging the Gap with HOCs. In *Future Generation Grids*. Springer Verlag, 2005.

[8] Jan Dünnweber and Sergei Gorlatch. HOC-SA: A grid service architecture for higher-order components. In *IEEE International Conference on Services Computing, Shanghai, China*, pages 288-294. IEEE Computer Society Press, September 2004.

[9] Pérez Christian, Priol Thierry and Ribes André, “A Parallel CORBA Component Model for Numerical Code Coupling”, The International Journal of High Performance Computing Applications (IJHPCA), SAGE Publications, Vol.17, Number 4, 2003, p.417-429.