SKELETON PARALLEL PROGRAMMING AND PARALLEL OBJECTS

Marcelo Pasin
CoreGRID fellow
on leave from Universidade Federal de Santa Maria
Santa Maria RS, Brasil
pasin@inf.ufsm.br

Pierre Kuonen
Haute Ecole Specialisée de Suisse Occidentale
École d'ingénieurs et d'architects de Fribourg
Fribourg, Suisse
pierre.kuonen@eif.ch

Marco Danelutto and Marco Aldinucci
Università di Pisa
Dipartimento d'Informatica
Pisa, Italia
marcod@di.unipi.it
aldinuc@di.unipi.it

Abstract This paper describes the ongoing work aimed at integrating the POP-C++ parallel object programming environment with the ASSIST component based parallel programming environment. Both these programming environments are shortly outlined, then several possibilities of integration are considered. For each one of these integration opportunities, the advantages and synergies that can be possibly achieved are outlined and discussed.

The text explains how GEA, the ASSIST deployer can be considered as the basis for the integration of such different systems. An architecture is proposed, extending the existing tools to work together. The current status of integration of the two environments is discussed, along with the expected results and fallouts on the two programming environments.

Keywords: Parallel, programming, grid, skeletons, object-oriented, deployment, execution.

1. Introduction

This is a prospective article on the integration of ASSIST and POP-C++ tools for parallel programming. POP-C++ is a C++ extension for parallel programming, offering parallel objects with asynchronous method calls. Section 2 describes POP-C++. ASSIST is a skeleton parallel programming system that ofers a structured framework for developing parallel applications starting from sequential components. ASSIST is described in Section 3 as well as some of its components, namely ADHOC and GEA.

This paper also describes some initial ideas of cooperative work on integrating parts of ASSIST and POP-C++, in order to obtain a broader and better range of parallel programming tools. It has been clearly identified that the distributed resource discovery and matching, as well as the distributed object deployment found in ASSIST could be used also by POP-C++. An architecture is devised in order to support the integration. An open question, and an interesting research problem, is whether POP-C++ could be used inside skeleton components for ASSIST. Section 4 is consacrated to these discussions.

2. Parallel Object-Oriented Programming

It is a very common sense in software engineering today that object-oriented programming and its abstractions improve software development. Besides that, the own nature of objects incorporate many possibilities of program parallelism. Several objects can act concurrently and independently from each other, and several operations in the same object can be concurrently carried out. For these reasons, a parallel object seems to be a very general and straightforward model to express concurrency, and thus to parallel programming.

POP stands for Parallel Object Programming, a programming model in which parallel objects are generalizations of traditional sequential objects. POP-C++ is an extension of C++ that implements the POP model, integrating distributed objects, several remote method invocations semantics, and resource requirements. The extension is kept as close as possible to C++ so that programmers can easily learn POP-C++ and existing C++ libraries can be parallelized with little effort. It results in an object-oriented system for developing high-performance computing applications for the Grid [13].

POP-C++ incorporates a runtime system in order to execute applications on different distributed computing tools [10][17]. This runtime system has a modular object-oriented service structure. Services are instantiated inside each application and can be combined to perform specific tasks using different lower level services (middleware, operating system). This design can be used to glue current and future distributed programming toolkits together to create a broader environment for executing high performance computing applications.

Parallel objects have all the properties of traditional objects, added to distributed resource-driven creation and asynchronous invocation. Each object creation has the ability to specify its requirements, making possible transparent optimized resource allocation. Each object is allocated in a separate address space, but references to an object are shareable, allowing for remote invocation. Shared objects with encapsulated data allow programmers to implement global data sharing in distributed environments. In order to share parallel objects, POP-C++ programs can arbitrarily pass their references from one place to another as arguments of method invocations. The runtime system is responsible for managing parallel object references.

Parallel objects support any mixture of synchronous, asynchronous, exclusive or concurrent method invocations. Without an invocation, a parallel object lies in an inactive state, only being activated a method invocation request. Syntactically, method invocations on POP objects are identical to those on traditional sequential objects. However, each method has its own invocation semantics, specified by the programmer. These semantics define different behaviours at both sides (caller and object) of a method call. Even though these semantics are important to define the POP model, they are irrelevant for the scope of this paper and will not be detailed here.

Prior to allocate a new POP object it is necessary to select an adequate placeholder. Similarly, when an object is no longer in use, it must be destroyed to release the resources it is occupying. POP-C++ provides (in its runtime system) automatic placeholder selection, object allocation, and object destruction. This automatic features result in a dynamic usage of computational resources and gives to the applications the ability to adapt to changes in both the environment and application behaviour.

Resource requirements can be expressed by the quality of service that components require from the environment. POP-C++ integrates the requirements into the code under the form of resource descriptions. Each parallel object constructor is associated with an **object description** that depicts the characteristics of the resources needed to create the object. Currently, resource requirements are expressed in terms of resource name, computing power, amount of memory, expected communication bandwidth and latency. Work is being done in order do broaden the expressiveness of the resource requirements.

The runtime system incorporates a server process called **job manager**, implementing services for object creation and for resource discovery. A simple distributed peer-to-peer resource discovery model is integrated, yet it does not scale well. Object creation is seen as a new process, which can be started with different management systems such as LSF [9], PBS [12] or even Globus [10].

3. Structured parallel programming with ASSIST

The development of efficient parallel programs is especially difficult with large-scale heterogeneous and distributed computing platforms as the Grid. Previous research on that subject exploited **skeletons** as a parallel coordination layer of functional modules, made of conventional sequential code [3]. This model allows to relieve the programmer from many concerns of classical, non structured parallel programming frameworks. With skeletons, mapping, scheduling, load balancing and data sharing, and maybe more, can be managed by either the compiler or the runtime system. In addition to that, using skeletons several optimizations can be efficently implemented, because the source code contains a description of the structure for the parallelism. That is much harder to do automatically when the parallelism pattern is unknown.

ASSIST is a parallel programming environment providing a skeleton based coordination language. It includes a skeleton compiler and runtime libraries. Parallel application are structured as generic graphs. The nodes are either parallel modules or sequential code. The edges are data streams. Sequential code can be written in C, C++ and Fortran, allowing to reuse existing code. The programmer can experiment different parallelisation strategies just changing a few lines of code and recompiling.

A **parallel module** is used to model the parallel activities of an ASSIST program. It can be specialized to behave as the most common parallelism patterns as farms, pipelines, or geometric and data parallel computations. Skeletons and coordination technology are exploited in such a way that parallel applications with complex parallelism patterns can be implemented without handling error prone details as process and communication setup, scheduling, mapping, etc.

The language allows to define, inside a parallel module, a set of virtual processors and to assign them tasks. The same task can be assigned to all virtual processors or to a certain group of them, or even to a single one. A parallel module can concurrently access state variables, and can interact with the external world using standard object access methods (like CORBA, for instance). A parallel module can handle as many input and output streams as needed. Non deterministic control is provided to accept inputs from different streams and explicit commands are provided to output items on the output streams.

Several optimizations are performed to efficiently execute ASSIST programs [15][1]. The environment was recently extended to support a component model (GRID.it) [2], that can interact with foreign component models, as CORBA CCM and Web Services. ASSIST components are supplied with autonomic managers [4] that adapt the execution to dynamic changes in the grid features (node or link faults, different load levels, etc.).

Along with binary executable files, the compiler generates an XML configuration file that represent the descriptor of the parallel application. GEA (see Section 3.1) is a deployer built to run the program based on the XML file. It takes care of all the activities needed to stage the code at remote nodes, to start auxiliary runtime processes, to run the application code and to gather the results back to the node where the program has been launched.

Grid applications often need access to fast, scalable and reliable data storage. ADHOC (Adaptive Distributed Herd of Object Caches) is a distributed persistent object repository toolkit [5], conceived in the context of the ASSIST project. ADHOC creates a single distributed data repository by the cooperation between multiple local memories. It separates management of computation and storage, supporting a broad class of parallel applications while achieving good performance. Clients access objects through proxies, that can implement protocols as complex as needed (e.g. distributed agreement). The toolkit enables object creation, set, get, removal and method call. The following section presents GEA in more detail.

3.1 Grid Application Deployment

ASSIST applications are deployed using GEA, the Grid Execution Agent. It is a parallel process launcher targeting distinct architectures, as clusters and the Grid. It has a modular design, intended for aggressive adaptation to different system architectures and to different application structures. GEA deploys applications and its infrastructure based on XML description files. It makes possible to configure and lauch processes in virtually any combination and order needed, adapting to different types of applications.

GEA has already been adapted for deployment on Globus grids and Unix computers supporting SSH access. Other different environments can be added without any modification in GEA's structure, because it is implemented using the Commodity Grid toolkit [16]. It currently supports the deployment of three different flavors of ASSIST applications, each one with a different process startup scheme. In the deployment of ASSIST applications, the compiler generates the necessary XML files, creating an automatic process to describe and launch applications. Besides the work described in this paper, the deployment of GridCCM components [8]is as well under way.

At the deployment of an application, after parsing the XML file that describe the resources needed, a suitable number of computing resources (nodes) are recruited to host the application processes. The application code is deployed to the selected remote nodes, by transferring the needed files to the appropriated places in the local filesystems. Data files and result files are transfered as well, respectively prior and after the execution of the application processes.

The necessary support processes to run the applications are also started at the necessary nodes.

The procedure for launching and connecting these processes with the application processes is automatized inside customized deployment modules. For example, ASSIST applications need processes to implement the data flow streams interconnecting their processes. ASSIST components need also supplementary processes for adaptation and dynamic connection. Other different launching patterns can be added with new modules, without any modification in GEA's structure.

4. Objects and skeletons getting along

Work is under progress within the CoreGRID network of excellence in order to establish a common programming model for the Grid. This model must implement a component system that keeps interoperability with the systems currently in use. ASSIST and POP-C++ have been designed and developed with different programming models in mind, but with a common goal: provide grid programmers with advanced tools suitable to develop efficient grid applications. They together represent two major and different parallel programming models (skeletons and distributed objects). Even if they may conduct the construction of the CoreGRID programming model to different directions, the set of issues addressed in both contexts has a large intersection. Compile or runtime enhancements made for any of them may be easily adapted to be used by other programming systems (possibly not only skeletal or object-oriented). Many infrastructural tools can be shared, as presented later in this text.

The possible relations between POP-C++ and ASSIST, one object-oriented and another based on skeletons are being studied inside CoreGRID. Work has been done to identify the possibilities to integrate both tools in such a way that effectively improve each one of them exploiting the original results already achieved in the other. Three possibilities that seem to provide suitable solutions have been studied:

1 Deploy POP-C++ objects using ASSIST deployment;

2 Adapt both to use the same type of shared memory;

3 Build ASSIST components of POP-C++ objects.

The first two cases actually improve the possibilities offered by POP-C++ by exploiting ASSIST technology. The third case improves the possibilities offered by ASSIST to assemble complex programs out of components written accordingly to different models. Currently such components can only be written using the ASSIST coordination language or inherited from CCM or Web Services. The following sections detail these three possibilities and discuss their relative advantages.

4.1 Same memory for ASSIST and POP-C++

POP-C++ implements asynchronous remote method invocations, using very basic system features, as TCP/IP sockets and POSIX threads. Instead of using those natively implemented parallel objects, POP-C++ could be adapted to use ADHOC objects. Calls to POP objects would be converted into calls to ADHOC objects. This would have the added advantage of being possible to somehow mix ADHOC applications and POP-C++ as they would share the same type of distributed object. This would as well add persistence to POP-C++ objects.

ADHOC objects are shared in a distributed system, as POP objects are. But they do not incorporate any concurrent semantics on the object side, neither their calls are asynchronous. In order to offer the same semantics, ADHOC objects (at both caller and callee sides) would have to be wrapped in jackets, which would implement the concurrent semantics using something like POSIX threads. This does not appear to be a good solution, neither about performance nor about elegance.

ADHOC has been implemented in C++. It should be relatively simple to extend its classes to be used inside a POP-C++ program, as it would with any other C++ class libraries. It means that it is already possible to use the current version of ADHOC to share data between POP-C++ and ASSIST applications. For all these reasons the idea of adopting ADHOC to implement regular POP-C++ objects has been precluded.

4.2 ASSIST components written in POP-C++

Currently, the ASSIST framework allows component programs to be developed with two type of components: **native** components and **wrapped** legacy components. Native components can either be sequential or parallel. They provide both a functional interface, exposing the computing capabilities of the component, and a non functional interface, exposing methods that can be used to control the component (e.g. to monitor its behaviour). They provide as well a **performance contract** that the component itself ensures by exploiting its internal autonomic control features implemented in the non functional code. Wrapped legacy components, on the other hand, are either CCM components or plain Web Services that can be automatically wrapped by the ASSIST framework tools to look like a native component.

The ASSIST framework can be extended in such a way that POP-C++ programs can also be wrapped to look like native components and therefore be used in plain native component programs. As the parallelism patterns allowed in native components are restricted to the ones provided by the ASSIST coordination language, POP-C++ components introduce in the ASSIST framework the possibility of having completely general parallel components. Of course, the efficiency of POP-C++ components would be completely in charge of POP-C++ compiler and its runtime environment.

Some interesting possibilities appear when exploring object oriented programming techniques to implement the non functional parts of the native component. In other words, one could try to fully exploit POP-C++ features to implement a customizable autonomic application manager providing the same non functional interface of native ASSIST components. These extensions, either in ASSIST or in POP-C++ can be subject to further research, especially in the context of CoreGRID, when its component model would be more clearly defined.

If eventually an ASSIST component should be written in POP-C++, it will be necessary to deploy and launch it. To launch an application, different types of components will have to be deployed. ASSIST has a deployer that is not capable of dealing with POP-C++ objects. One first step to enable their integration should be the construction of a common deployment tool, capable of executing both types of components.

4.3 Deploying ASSIST and POP-C++ alike

ASSIST provides a large set of tools, including infrastructure for launching processes, integrated with functions for matching needs to resouces capabilities. The POP-C++ runtime library could hook up with GEA, the ASSIST deployer, in different levels. The most straightforward is to replace the parts of the POP-C++ job manager related to object creation and resource discovery with calls to GEA.

As seen in Section 3.1, GEA was build to be extended. It is currently able to deploy ASSIST applications, each type of it being handled by a different deployer module. Adding support for POP-C++ processes, or objects, can be done by writing another such module. POP-C++ objects are executed by independent processes that depend on very little. Basically, the newly created process has to allocate the new object, use the network to connect with the creator, and wait for messages on the connection. The connection to establish is defined by arguments in the command line, which are passed by the caller (the creator of the new object). The POP-C++ deployer module is actually a simplified version of those used for ASSIST applications.

Process execution and resource selection in both ASSIST and POP-C++ happen in very different patterns. ASSIST relies on the structure of the application and is performance contract to specify the type of the resources needed to execute it. This allows for a resource allocation strategy based on graphs, specified ahead of the whole execution. Chosen a given set of resources, all processes are started. The adaptation follow certain rules and cannot happen without boundaries. POP-C++ on the other hand does not impose any program structure. A new resource must be located on-the-fly for every new object created. The characteristics of the resources are completely variable, and cannot be determined previous to the object creation.

It seems clear that a good starting point for integration of POP-C++ and ASSIST is the deployer, and some work has been done in that direction. The next section of this paper discusses the architecture of the extensions designed to support the deployment of POP objects with with GEA, the ASSIST deployer.

5. Architecture for a common deployer

The modular design of GEA allows for extensions. Nevertheless, it is written in Java. The runtime of POP-C++ was written in C++ and it must be able to reach code running in Java. Anticipating such uses, GEA was built to run as a server, exporting a TCP/IP interface. Client libraries to connect and send requests to it were written in both Java and C++. The runtime library of POP-C++ has then to be extended to include calls to GEA's client library.

In order to assess the implications of the integration proposed here, the object creation procedure inside the POP-C++ runtime library has to be seen more into detail. The steps are as following:

1 A proxy object is created inside the address space of the creator process, called **interface**.

2 The interface evaluates the object description (written in C++) and calls a resource discovery service to find a suitable resource.

3 The interface launches a remote process to host the new object in the given resource and waits.

4 The new process running remotely connects with the interface, receives the constructor arguments, creates the object in the local address space and tells the interface that the creation ended.

5 The interface returns the proxy object to the caller.

GEA can currently only be instructed to, at once, choose an adequate resource, then load and launch a process. An independant discovery service, as required by the POP-C++ interface, is not yet implemented in GEA. On the other hand, in can be used as it is just rewriting the calls in the POP-C++ object interface. The modifications are:

- The resource discovery service call has to be rewritten to just build an XML description of the resource based on the object description.

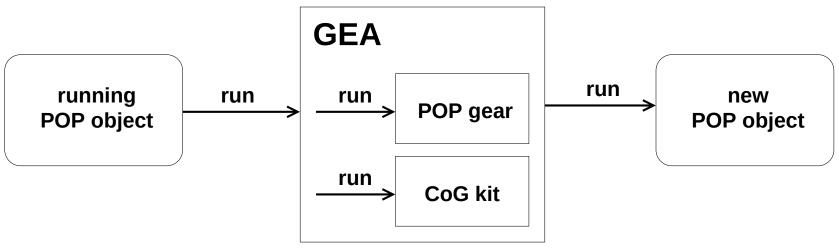
- The remote process launch should be rewritten to call the GEA C++ client library, passing the XML description formrly built.

Requests to launch processes have some restrictions on GEA. Its currently structured model matches the structured model of ASSIST. Nodes are divided into administrative domains, and each domain is managed by a single GEA server. The ASSIST model dictates a fixed structure, with parallel modules connected in a predefined way. All processes of parallel modules are assigned to resources when the execution starts. It is eventually possible to adjust on the number of processes inside of a running parallel module, but the new processes must be started in the same domain.

POP-C++ needs a completely dynamic model to run parallel objects. An object running in a domain must be able to start new objects in different domains. Even a sigle server for all domains is not a good idea, as it may become a bottleneck. In order to support multiple domains, GEA has to be extended to a more flexible model. GEA servers must forward execution calls between each other. Resource discovery for new processes must also take into account the resources in all domains (not only the local one). That is a second reason why the resource discovery and the process launch were left to be done together.

GEA is build to forward a call to create a process to the corresponding process type module, called **gear**. With POP-C++, the POP gear will be called by GEA for every process creation. The POP gear inspects all resources available and associates the process creation request with a suitable resource. The CoG kit will eventually be called to launch the process in the associated resource. This scenario is illustrated in Figure 1. A problem arises when no suitable resource is available in the local domain, as GEA does not share resource information with other servers.

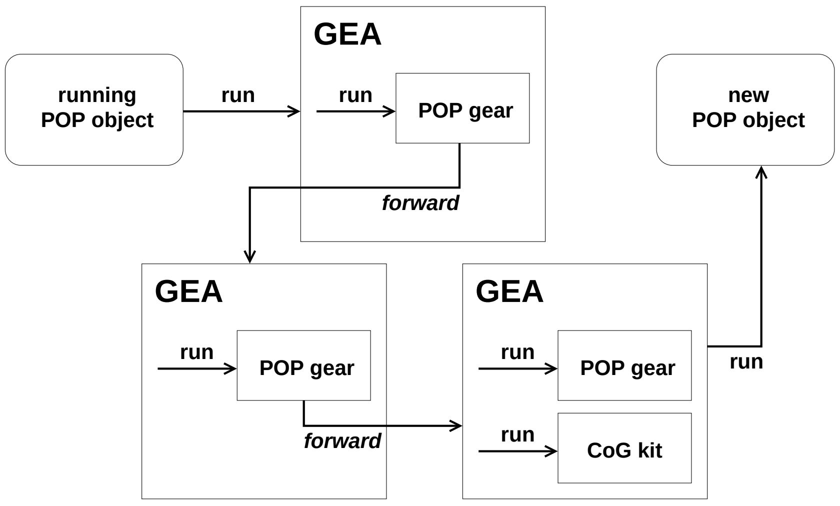
Figure 1. GEA with a cetralized POP-C++ gear



By keeping together the descriptions of the program and the resource, the mapping decision can be postponed to the last minute. The Figure 2 shows a scenario, where a POP gear does not find a suitable resource locally. A peer-to-peer network, established with GEA servers and their POP gears would forward the request until it is eventually satisfied, or a timeout is reached. A similar model was proposed as a Grid Information Service, using routing indexes to improve performance [14].

In the context of POP-C++ (and in other similar systems, as ProActive [7], for instance), the allocation is dynamic, with every new process created idependently of the others. Structured systems as ASSIST need to express application

Figure 2. GEA with a peer-to-peer POP-C++ gear



needs as a whole prior to the execution. Finding good mappings in a distributed algorithm is clearly an optimisation problem, that could eventually be solved with heuristics expoiting a certain degree of locality. Requirements and resource sets must be split into parts and mixed and matched in a distributed and incremental (partial) fashion [11].

In either contexts (static or dynamic), resources would better be described without a predefined structure. Descriptions could be of any type, not just amounts of memory, CPU or network capacity. Requirements sould be expressed as predicates that evaluate to a certain degree of satisfaction [6]. The languages needed to express requirements and resources, as well as efficient distributed resource matching algorithms are still interesting research problems.

6. Conclusion

The questions discussed in this paper entail a CoreGRID fellowship. All the possibilities described in the previous sections were considered, and the focus of interest was directed to the integration of GEA as the POP-C++ launcher and resource manager. This will impose modifications on POP-C++ runtime library and new funcionalities for GEA. Both systems are expected to improve thanks to this interaction, as POP-C++ will profit from better resource discovery and GEA will implement a less restricted model.

Further research on the matching model will lead to new approaches on expressing and matching application requirements and resource capabilities. This model should allow a distributed implementation that dynamically adapt the requirements as well as the resource availability, being able to express both ASSIST and POP-C++ requirements, and probably others.

A subsequent step can be a higher level of integration, using POP-C++ programs as ASSIST components. This could allow to exploit full object oriented parallel programming techniques in ASSIST programs on the Grid. The implications of POP-C++ parallel object oriented modules on the structured model of ASSIST are not fully identified, especially due to the dynamic aspects of the objects created. Supplementary study has to be done in order to devise its real advantages and consequences.

References

[1] M. Aldinucci, S. Campa, P. Ciullo, M. Coppola, S. Magini, P. Pesciullesi, L. Potiti, R. Ravazzoloand M. Torquati, M. Vanneschi, and C. Zoccolo. The Implementation of ASSIST, an Environment for Parallel and Distributed Programming. In *Proc. of EuroPar2003*, number 2790 in "Lecture Notes in Computer Science". Springer, 2003.

[2] M. Aldinucci, S. Campa, M. Coppola, M. Danelutto, D. Laforenza, D. Puppin, L. Scarponi, M. Vanneschi, and C. Zoccolo. Components for High-Performance Grid Programming in GRID.it. In *Component modes and systems for Grid applications*, CoreGRID. Springer, 2005.

[3] M. Aldinucci, M. Danelutto, and P. Teti. An advanced environment supporting structured parallel programming in Java. *Future Generation Computer Systems*, 19(5):611–626, 2003. Elsevier Science.

[4] M. Aldinucci, A. Petrocelli, E. Pistoletti, M. Torquati, M. Vanneschi, L. Veraldi, and C. Zoccolo. Dynamic reconfiguration of grid-aware applications in ASSIST. In *11th Intl Euro-Par 2005: Parallel and Distributed Computing*, number 3149 in "Lecture Notes in Computer Science". Springer Verlag, 2004.

[5] M. Aldinucci and M. Torquati. Accelerating apache farms through ad-HOC distributed scalable object repository. In M. Danelutto, M. Vanneschi, and D. Laforenza, editors, *10th Intl Euro-Par 2004: Parallel and Distributed Computing*, volume 3149 of *"Lecture Notes in Computer Science"*, pages 596–605, Pisa, Italy, August 2004. "Springer".

[6] S. Andreozzi, P. Ciancarini, D. Montesi, and R. Moretti. Towards a metamodeling based method for representing and selecting grid services. In Mario Jeckle, Ryszard Kowalczyk, and Peter Braun II, editors, *GSEM*, volume 3270 of *Lecture Notes in Computer Science*, pages 78–93. Springer, 2004.

[7] F. Baude, D. Caromel, L. Mestre, F. Huet, and J. Vayssière. Interactive and descriptor-based deployment of object-oriented grid applications. In *Proceedings of the 11th IEEE Intl Symposium on High Performance Distributed Computing*, pages 93–102, Edinburgh, Scotland, July 2002. IEEE Computer Society.

[8] Massimo Coppola, Marco Danelutto, Sébastien Lacour, Christian Pérez, Thierry Priol, Nicola Tonellotto, and Corrado Zoccolo. Towards a common deployment model for grid systems. In Sergei Gorlatch and Marco Danelutto, editors, *CoreGRID Workshop on Integrated research in Grid Computing*, pages 31–40, Pisa, Italy, November 2005. CoreGRID.

[9] Platform Computing Corporation. *Running Jobs with Platform LSF*, 2003.

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

[11] Felix Heine, Matthias Hovestadt, and Odej Kao. Towards ontology-driven p2p grid resource discovery. In Rajkumar Buyya, editor, *GRID*, pages 76–83. IEEE Computer Society, 2004.

[12] R. Henderson and D. Tweten. Portable batch system: External reference specification. Technical report, NASA, Ames Research Center, 1996.

[13] T.-A. Nguyen and P. Kuonen. ParoC++: A requirement-driven parallel object-oriented programming language. In *Eighth Intl Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'03), April 22-22, 2003, Nice, France*, pages 25–33. IEEE Computer Society, 2003.

[14] Diego Puppin, Stefano Moncelli, Ranieri Baraglia, Nicola Tonellotto, and Fabrizio Silvestri. A grid information service based on peer-to-peer. In *Lecture Notes in Computer Science 2648, Proceeeding of Euro-Par*, pages 454–464, 2005.

[15] M. Vanneschi. The Programming Model of ASSIST, an Environment for Parallel and Distributed Portable Applications . *Parallel Computing*, 12, December 2002.

[16] Gregor von Laszewski, Ian Foster, and Jarek Gawor. CoG kits: a bridge between commodity distributed computing and high-performance grids. In *Proceedings of the ACM Java Grande Conference*, pages 97–106, June 2000.

[17] T. Ylonen. SSH - secure login connections over the internet. In *Proceedings of the 6th Security Symposium*, page 37, Berkeley, 1996. USENIX Association.