Skeleton Parallel Programming and Parallel Objects

Marcelo Pasin[1], Pierre Kuonen[2], Marco Danelutto[3], and Marco Aldinucci[4]

[1] CoreGRID fellow, on leave from Universidade Federal de Santa Maria, Brazil
[2] Haute Ecole Specialisée de Suisse Occidentale (HES-SO/EIA-FR), Fribourg, Switzerland
[3] Departimento d'Informatica, Univesità di Pisa, Italy
[4] Istituto di Scienza e Tecnologia dell'Informazione (CNR/ISTI), Pisa, Italy

Abstract. We describe here 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, first. 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. Eventually, the current status of integration of the two environments is discussed, along with the expected results and fallouts on the two programming environments.

1 Introduction

This is a prospective paper 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 and some of its components, namely GEA and ADHOC, are described in sections 3.1 and 3.2 respectively.

This paper 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 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 improves 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 for 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. All services are instantiated inside each application and can be combined to perform specific tasks using different system services. 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 cration has the ability to specify its requirements, making possible transparent optimized resource alocation. Each object is allocated on 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++ can arbitrarily pass them 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 of the parallel object, called the interface and the object-side semantics.

The interface semantics affect the caller of a method invocation, which can be either synchronous or asynchronous. With **synchronous** invocation, the caller blocks until the execution of the requested method on the object side is finished. This corresponds to the traditional (remote) method invocations. **Asynchronous** invocations, on the contrary, return immediately after sending the request to the remote object. Asynchronous invocation is important to exploit the parallelism because it enables to overlap computations and communications. No computing result is available when the invocation returns to the caller, so, under the current model, it cannot produce results.

The object-side semantics rule the execution of methods inside each object. A method can be of one of three types: concurrent, sequential, or mutex. Invocations to **concurrent** methods are executed concurrently if no mutex invocation is currently running. The concurrent invocation is important to achieve the parallelism inside each parallel object and to improve overlapping between computation and communication.

Using **sequential** invocation, methods are executed in mutual exclusion, following the requests' arrival order. Several simultaneous sequential methods invocations are served sequentially (see Fig. 1). Concurrent methods that have been previously started

can still continue their normal execution. This guarantees the serializable consistency of all sequential invocations in the same object.

Invocations to **mutex** methods are executed in complete exclusion with all other methods of the same object. A request is executed only if no other invocation are running. Otherwise, the current method will be blocked until all invocations (including concurrent ones) are terminated (see Fig. 1). Mutex invocations are important to synchronize concurrencies and to assure the correctness of shared data state inside the parallel object.

Fig. 1. Exampe of different invocation requests

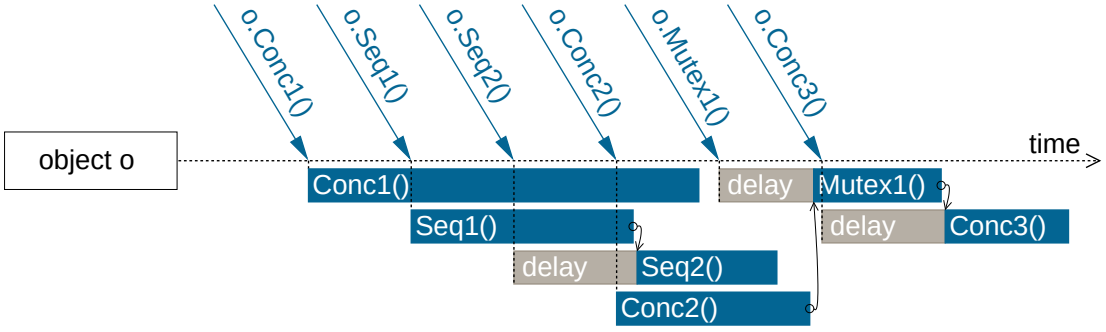


Figure 1 illustrates different invocation semantics. Sequential invocation `Seq1()` is served immediately, running concurrently with `Conc1()`. Although the sequential invocation `Seq2()` arrives before the concurrent invocation `Conc2()`, it is delayed due to the current execution of `Seq1()` (no order between concurent and sequential invocations). When the mutex invocation `Mutex1()` arrives, it has to wait for other running methods to finish. During this waiting, it also blocks other invocation requests arriving later, as `Conc3()`, until the mutex invocation request completes its execution.

Prior to allocate a new 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 the 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 parallel objects under the form of resource descriptions. Each parallel object constuctor is associated with an **object description** that depicts the characteristics of the resources needed to create the object. The resource requirements in object descriptions are expressed in terms of resource (host) name, computing power, amount of memory, expected communication bandwidth and latency.

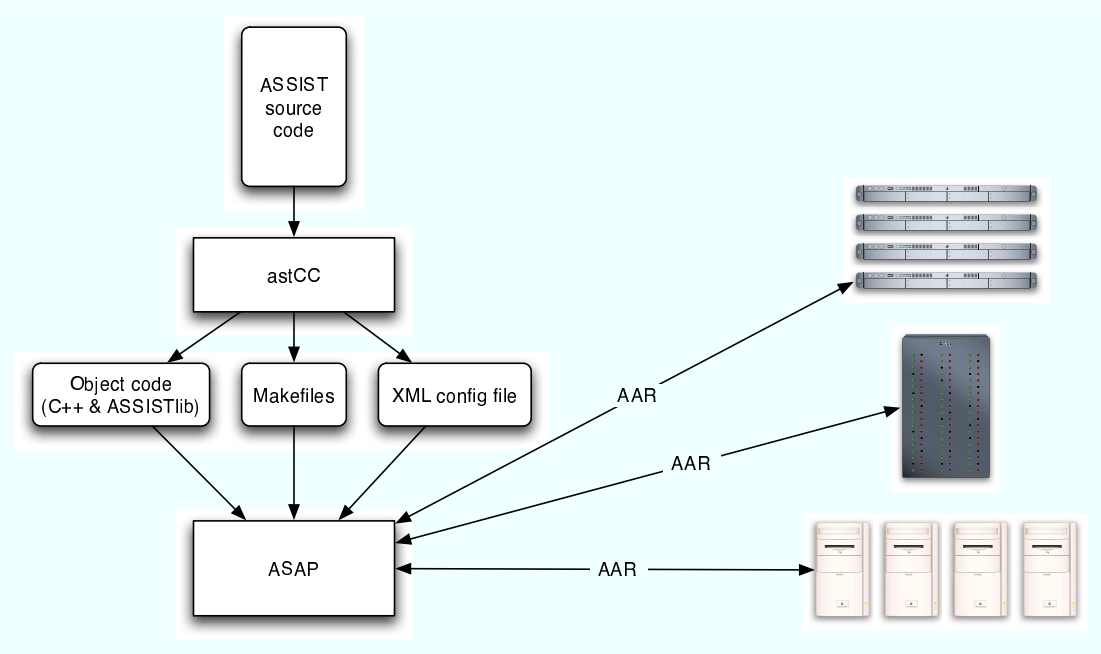
3 Structured parallel programming with ASSIST

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 programmer from many concerns of classical, non structured parallel programming frameworks. As an example, scheduling, mapping, load balancing and data sharing are all managed by either the compile tools or the runtime systems of structured parallel programming frameworks. In addition to that, due to the exposition by the programmer in the program source code of the structure of parallelism exploitation, several optimizations can be efficently implemented at either compiler or runtime level. That is not applicable in case the parallellism exploitation pattern is not available or it has to be mined from source code.

ASSIST is a parallel programming environment providing a skeleton based coordination language. A compiler and a set of runtime tools allow ASSIST programs to be run on clusters, networks of workstations and grids. Several optimizations are performed that allow to achieve high efficiency in the execution of ASSIST programs [15,1]. Its programming environment was recently extended to support GRID.it components [2]. They can as well be used to interact with non GRID.it components, in particular with CORBA components and with Web Services. ASSIST GRID.it components are supplied with autonomic managers [4] that adapt the component execution to dynamic changes in the grid features (node or link faults, different load levels, etc.).

Fig. 2. ASSIST structure



The structure of the ASSIST programming environment is outlined in Fig. 2. Source code is processed by the ASSIST compiler, producing C++ code, makefiles to be used to produce the actual object code for several different architectures, and an XML configuration file that represent the descriptor of the parallel application. To run the program, this XML file is processed by the GEA tool (see section 3.1), taking care of all the activities needed to stage the code at remote nodes, starting auxiliary runtime processes, starting application code and gathering results back to the node where the program has been launched. Some parts of the system processes launched with the application code of an ASSIST program are related to ADHOC ASSIST subsystem. ADHOC is basi-

cally a shared data resource that is used to support both data repository and stream communication.

3.1 Grid Application Deployment

The GEA ASSIST tool is a parallel process launcher targeting two distinct architectures: Globus grids and POSIX/TCP workstation networks and clusters supporting SSH access. GEA takes as an input an XML file generated by the ASSIST compiler out of the ASSIST source code and an AAR file (Assist ARchive file), hosting the code and the libraries needed to deploy the ASSIST program on a remote node.

The XML file is parsed by GEA, then a suitable number of computing resources (nodes) are recruited to host the application processes. In case of Globus, resource recruitement is performed interacting with standard MDS services. In case of POSIX/TCP SSH architectures, POSIX commands are used in conjunction with SSH. The application code is deployed to the selected remote nodes, by transferring to them the proper AAR files, then the archive files are uncompressed and unpacked. The object code and libraries are then trasferred to the proper places in the local filesystems.

The necessary support processes to run the applications are also started at the selected nodes. In particular, the HOC processes used to implement the data flow streams interconnecting ASSIST processes are started in this step. Eventually, the processes derived from and implementing the user code are run. They perform user defined code upon the data received from the HOC implemented data flow streams, which eventually deliver results again on the HOC channels.

All these steps can be performed exploiting two different kinds of technologies: Globus and SSH. With **Globus** (toolkit 2.4, currently moving to toolkit 4), the resource lookup is performed exploiting MDS facilities, data and code (AAR) staging is performed via GlobusFTP and processes are run remotely exploiting Globus remote commanding facilities. **SSH** is a standard mechanism to run remote commands and to transfer files, natively available by classical POSIX operating systems and supported, non natively, also by Windows. Code and data staging is performed using `scp`, remote processes are started via `ssh` and resources are looked for by inspecting a file or by a special lookup process testing access to the machines on the local network.

The whole process not only supports the user code launch, but also the management of all the runtime processes needed to monitor ASSIST program performance and possibly to force the program to terminate, or even to adapt (e.g. varying its parallelism degree) to changements in the grid architecture features and/or in the perfromance contracts issued by the users.

ASSIST GEA is currently being engineered by separating the code performing actions from the code planning the application deployment. The main GEA code implements a **plugin manager** built on top of the COG toolkit [16]. The plugin manager basically is able to load and run a module configured according to the XML file tags. The plugin, in turn, is able to perform all the actions needed to deploy and run a code developed with a particular environment. As an example, the ASSIST plugin works as described above, by first stagin and running the ADHOC code, than staging and running the ASSIST user code. A CORBA/CCM [11] plugin first sets up the CORBA framework and then launches the CCM code wrapped in the ASSIST program.

3.2 Distributed Data Collections

To profit from the large processing potential of the grid, applications cannot assume the platform to be neither homogeneous, secure, reliable nor centrally managed. Also, these applications should be fed with large distributed collections of data.

ADHOC (Adaptive Distributed Herd of Object Caches), is a distributed object repository [5]. It has been conceived in the context of the ASSIST project, as a distributed persistent virtual storage. It provides the application designers with a toolkit to solve data storage problems in a grid framework. In particular, it provides building blocks to set up client-server and service-oriented infrastructures which can cope with the grid characteristics. Its underlying design principle consists in decoupling the management of computation and storage in distributed applications.

Parallel grid applications often need processing large amounts of data. Data storages for such applications are required to be fast, dynamically scalable and enough reliable to survive to some failures. Decoupling helps in providing a broad class of parallel applications with these features while achieving good performances. ADHOC creates a local virtual memory associated with every processing element. A common distributed data repository is provided by the cooperation between multiple local memories.

ADHOC implements an external repository for arbitrary length objects. Clients may access objects through different protocols, implemented within proxy libraries. Proxies may act as a simple adaptors, or exploit complex behaviors, even cooperating with other proxies (e.g. distributed agreement). An object cannot be spread across different nodes, but it can be replicated. Objects can be grouped in ordered collections of objects, which can be spread across different nodes.

Objects and collections are identified by keys. The actual data location is found at execution time through a distributed hash table. ADHOC API enables to get, put and remove objects, and it provides remote execution of objects methods. This operation is initially meant as mechanism to extend server core functionalities for specific needs, as for example lock and unlock the object for consistency management.

4 Exploiting POP-C++ and ASSIST synergies

POP-C++ and ASSIST have been designed and developed with different programming models in mind, but with a common goal: provide grid programmers with advanced tools suitable to be used to develop efficient grid applications. Some of the problems addressed and (partially) solved in the two contexts are therefore common problems. In particular, the way active entities (objects in POP-C++ and processes in ASSIST) are deployed to the grid processing nodes, the kind of support needed to efficiently share data and the way parallelism can be exploited in a single grid program component are all subject of design and implementation efforts in both these environments.

In this section, we want to address the synergies that can be exploited among POP-C++ and ASSIST. We want to consider the possibilities of integrating the POP-C++ and the ASSIST environments and, in particular, the integration possibilities that effectively improve one of the two environments exploiting the original results already achieved in the other environment. Three kind of possibilities have been explored, that seem to provide suitable improvements in either the ASSIST or the POP-C++ environments:

1. to exploit the ASSIST GEA deployment tool to deploy and manage POP-C++ programs
2. to exploit ASSIST ADHOC shared memory support to implement shared state in POP-C++ programs
3. to use POP-C++ to implement GRID.it components in the ASSIST framework

The former two cases actually improve the possibilities offered by POP-C++ by exploiting ASSIST technology. The latter 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 advatages/disadvantages.

4.1 Exploiting ASSIST GEA in POP-C++

POP-C++ comprises a runtime library that implements some services for launching remote processes and for resource discovery. Launching remote processes is provided by a **job manager**, which has two main functionalities: launching the parallel object and managing the resources. It allows to submit jobs with different management systems such as LSF [9], PBS [12] or even Globus [10]. It does not provide authentication services and relies on the security infrastructure of the management system used.

A distributed resource discovery is integrated in the POP-C++ runtime system. It differs from the centralized approach such as in Globus, NetSolve [8] or Condor [14]. Information about the POP-C++ resources is fully distributed and accessed on demand, configuring an adaptive peer-to-peer model. Though, this model has shown some scalability problems and it is a good candidate for a replacement.

GEA provides a comprehensive infrastuture for launching processes, integrated with functions for matching needs to resouces capabilities. The integration of POP-C++ with GEA could be done in different levels. The most straightforward would be to replace the parts of the job manager related to object loading and running and the resource discovery with calls to GEA, which would perform all launching and all resource management. In any case, POP object files would have to be packed into ASSIST application packages, which is the file format understood by GEA.

In order to assess the implications of the integration proposed here, the object creation procedure inside the job manager has to be seen more into detail. Initially, a proxy object is created, called interface. The interface evaluates the object description and calls the resource discovery service to find a suitable resource. The interface then launches an object server in the given resource. The object server now running in the resource takes care of all other tasks, as downloading and starting the executable code, setting the connection with the interface, receiving the constructor arguments and signalling the interface about the end of the creation.

The discovery service as required by the interface is not yet implemented in GEA. If implemented, GEA, should return an access point to the resource found. As GEA can be instructed to load and launch a program in a specified resource, the interface algorithm could stay as it is. On the other hand, instead of adding a discovery call to GEA, the interface algorithm could be changed. It could directly ask GEA to launch the

new object using a resource description. This is also present in GEA, but only could be used with some modification.

Requests to launch processes have some restrictions on GEA. Its currently structured model matches the structured model of ASSIST. Nodes are divided into domains. The ASSIST model dictates a fixed structure for parallel programs, which are formed by parallel modules, that are connected in a predefined way. Modules are divided into processes, which are assigned to resources when the execution starts. All resources assigned to a single parallel module must belong to the same domain. 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. In order to support that, GEA has to be extended to a more flexible model. This can be done by making a process launching interface accessible from inside a domain. Also, resource discovery for new processes must take into account the resources in all domains (not only the local one). This functionalities can be added to GEA either as plugins or as a separate process, as is the case of the ADHOC server.

In most grid systems, node allocation is based on some sort of application requirements and on resource capabilities. In the context of POP-C++ (an in other similar systems, as ProActive [7], for instance), the allocation must be done dynamically. This is clearly an optimisation problem, that could eventually be solved with distributed heuristics expoiting a certain degree of locality. In order to do that, requirements and resource sets must be split into parts and mixed and matched in a distributed and incremental (partial) fashion. Requirements sould be expressed as predicates that evaluate to a certain degree of satisfaction [6]. Resources should be described without a predefined structure (descriptions could be of any type, not just memory, CPU and network). The languages needed to express requirements and resources, as well as good distributed resource matching algorithms are interesting research problems.

4.2 Data sharing in POP-C++ through ADHOC

POP-C++ implements asynchronous remote method invocations, using very basic system features, as TCP/IP sockets and POSIX threads. Instead of using those 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 (in the caller and in the callee side) 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.

4.3 Parallel POP-C++ components in the ASSIST framework

Currently, the ASSIST framework allows component programs to be developed with two type of components: **native** GRID.it components and **wrapped** legacy components. GRID.it 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 takes 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 GRID.it native component.

The ASSIST framework can be extended in such a way that POP-C++ programs can also be wrapped to look like GRID.it components and therefore be used in plain GRID.it component programs. As the parallelism exploitation patterns allowed in native GRID.it 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 the POP-C++ components is completely in charge of the POP-C++ compiler/runtime environment. Some interesting possibilities also come in this case from the exploitation of object oriented programming techniques to implement the non functional part of the GRID.it component. In other words, trying to exploit full POP-C++ features to implement a customizable autonomic application manager providing the same non functional interface provided by ASSIST/GRID.it components.

5 Conclusion

The questions discussed in this paper entail a CoreGRID fellowship. All the possibilities described in the previous sections are currently being considered. The main focus of interest is clearly 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. A running prototype is expected for the end of the year.

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 dinamically 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 GRID.it wrapped legacy components. This could allow to exploit full object oriented parallel programming techniques in ASSIST programs on grids. 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. H. Casanova and J. Dongarra. NetSolve: A network-enabled server for solving computational science problems. *The Intl Journal of Supercomputer Applications and High Performance Computing*, 11(3):212–223, Fall 1997.
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. Object Management Group. *CORBA Components*, 2002.
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. R. Raman, M. Livny, and M.H. Solomon. Resource management through multilateral matchmaking. In *HPDC*, pages 290–291, 2000.
15. M. Vanneschi. The Programming Model of ASSIST, an Environment for Parallel and Distributed Portable Applications . *Parallel Computing*, 12, December 2002.
16. G. von Laszewski, B. Alunkal, K. Amin, J. Gawor, M. Hategan, and S. Nijsure. The Java CoG Kit User Manual. MCS Technical Memorandum ANL/MCS-TM-259, Argonne National Laboratory, March 14 2003.
17. T. Ylonen. SSH - secure login connections over the internet. In *Proceedings of the 6th Security Symposium*, page 37, Berkeley, 1996. USENIX Association.