John von Neumann Institute for Computing

Building Interoperable Grid-aware ASSIST Applications via Web Services

M. Aldinucci, M. Danelutto, A. Paternesi, R. Ravazzolo, M. Vanneschi

published in

Parallel Computing:
Current & Future Issues of High-End Computing,
Proceedings of the International Conference ParCo 2005,
G.R. Joubert, W.E. Nagel, F.J. Peters, O. Plata, P. Tirado, E. Zapata (Editors),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **33**, ISBN 3-00-017352-8, pp. 145-152, 2006.

© 2006 by John von Neumann Institute for Computing
Permission to make digital or hard copies of portions of this work for personal or classroom use is granted provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise requires prior specific permission by the publisher mentioned above.

http://www.fz-juelich.de/nic-series/volume33

Building Interoperable Grid-aware ASSIST Applications via Web Services*

M. Aldinucci[a], M. Danelutto[b], A. Paternesi[b], R. Ravazzolo[b], M. Vanneschi[b]

[a]Inst. of Information Science and Technologies (ISTI) – CNR, Via Moruzzi 1, I-56124 Pisa, Italy

[b]Department of Computer Science, University of Pisa, Largo B. Pontecorvo 3, I-56127 Pisa, Italy

Abstract: The ASSIST environment provides a high-level programming toolkit for the grid. ASSIST applications are described by means of a coordination language, which can express arbitrary graphs of modules. These modules (or a graph of them) may be enclosed in components specifically designed for the grid (GRID.it components). In this paper we describe how ASSIST modules can be wired through standard Web Services, and how GRID.it components may be made available as standard Web Services.
Keywords: Grid, Web Services, components, ASSIST, GRID.it

1. Introduction

The idea of "software as a service" has recently gained more and more importance because of standardization of the way in which software may be delivered as a service over the network. Once software applications are available as a service, a composite service can be created by somehow connecting services one another under a centralized or distributed control. As an example, a single client may access a set of services programmatically using some scripting language realizing a composite service. This composite service expresses a business process capturing a particular intra or inter enterprise workflow. BPEL (Business Process Execution Language) is one popular scripting language enabling the composition of services [12].

Grid technologies have also been aligned with Web services technologies to capitalize on desirable Web services properties, such as service description and discovery, automatic generation of client and server code from service descriptions, binding of service descriptions to interoperable network protocols, compatibility with emerging higher-level open standards, services and tools, as well as broad commercial support. The Open Grid Services Architecture (OGSA) is a paradigmatic example [10]: the recent version of OGSA/Globus implements the emerging standard Web Services Resource Framework (WSRF) as specified by OASIS [7]. It allows interconnecting state full resources via Web services and includes APIs for the management of transient application life cycles and event notifications. Indeed, using Globus directly is difficult because the process requires manually writing and arranging multiple XML-configuration files. These files must cover explicit declaration of all resources, the services used to connect to them, their interfaces and the corresponding bindings to the employed protocol, since Globus applications should be accessible in a platform-and language-independent manner [9].

In addition, in order to build efficient grid-aware applications, programmers must design highly concurrent algorithms that can execute on large-scale platforms. They must then implement these algorithms correctly and efficiently. Therefore, we envision a layered, high-level programming model for the grid. In such software architecture, the bottom tiers should cope with key grid requirements for protocols and services (connectivity protocols concerned with communication and authentication, resource protocols concerned with negotiating access to individual resources) and collective

*This work has been supported by the Italian MIUR FIRB *GRID.it* project No. RBNE01KNFP.

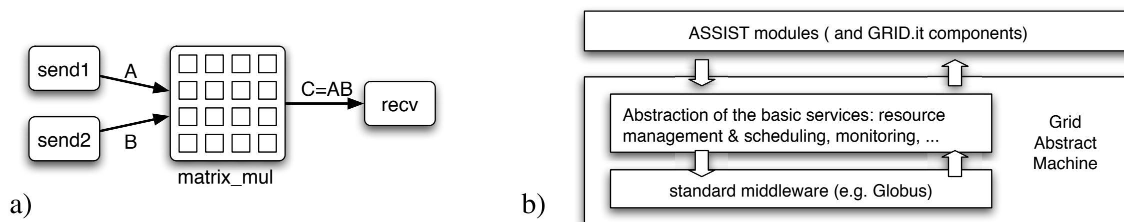

Figure 1. a) A simple ASSIST application (parallel matrix multiplication). b) ASSIST architecture.

protocols and services (concerned with the coordinated use of multiple resources) [11]. Concerning the top tiers, we envision a grid-aware application as the composition of a number of coarse grained, cooperating components within a high-level programming model, which is characterized by a high-level view of compositionality, interoperability, reuse, performance and application adaptivity. Applications are expressed entirely on top of this level. This vision is currently pursued by several research initiatives and programming environments, among the others, within the ASSIST (GRID.it project) [18] and GrADS [16] projects. The underlying idea of these programming environments is moving most of the grid specific efforts needed while developing high-performance grid applications from programmers to grid tools and run-time systems. This leaves programmers the responsibility of organizing the application specific code and the programming tools (i.e. the compiling tools and/or the run-time system) the responsibility of properly interacting with the grid.

In this work, we discuss two distinct but related kind of tools: those making available a whole ASSIST program or an ASSIST parallel module as a standard Web Service, and those supporting access to standard Web Services from within standard ASSIST code. Overall, these tools guarantee interoperability between the ASSIST and the Web Service worlds.

2. The ASSIST coordination language

ASSIST applications are described by means of a coordination language, which can express arbitrary graphs of modules, interconnected by typed streams of data. Each stream realizes a one-way asynchronous channel between two sets of endpoint modules: sources and sinks. Data items injected from sources are broadcast to all sinks. All data items injected into a stream should match stream type. A simple application implementing parallel matrix multiplication is shown in Fig. 1 a).

Modules can be either sequential or parallel. A sequential module wraps a sequential function. A parallel module *(parmod)* can be used to describe the parallel execution of a number of sequential functions that are activated and run as *Virtual Processes* (VPs) on items arriving from input streams. The VPs may synchronize with the others through barriers. The sequential functions can be programmed by using a standard sequential language (C, C++, Fortran).

A *parmod* may behave in a data-parallel (e.g. SPMD/for-all/apply-to-all) or task-parallel (e.g. farm) way and it may exploit a distributed shared state, which survives to VPs lifespan. A module can nondeterministically accept from one or more input streams a number of input items, which may be decomposed in parts and used as function parameters to instantiate VPs according to the input and distribution rules specified in the parmod. The VPs may send items or parts of items onto the output streams, and these are gathered according to the output rules. The simple application in Fig. 1 a) includes three sequential modules (send1, send2, and recv) and one parmod (matrix_mul), which take two matrixes and give their product. The matrix_mul parmod is declared as follows:

`matrix_mul` is declared as follows:

```
1  parmod matrix_mul (input_stream long M1[N][N], long M2[N][N]
2                     output_stream long M3[N][N]) {
3    topology array [i:N][j:N] Pv;
4    attribute long A[N][N] scatter A[*ia][*ja] onto Pv[ia][ja];
5    attribute long B[N][N] scatter B[*ib][*jb] onto Pv[ib][jb];
6    stream long ris;
7    do input_section {
8      guard1: on , , M1 && M2 {
9        distribution M1[*i0][*j0] scatter to A[i0][j0];
10       distribution M2[*i1][*j1] scatter to B[i1][j1];
11     }   } while (true)
12   virtual_processes {
13     elab1 (in guard1 out ris) {
14       VP i, j {          f_mul (in A[i][], B[][j] output_stream ris);          }}}
15   output_section {
16     collects ris from ALL Pv[i][j] {
17       int elem; int Matrix_ris_[N][N];
18       AST_FOR_EACH(elem) {
19         Matrix_ris_[i][j]=elem;
20       }
21       assist_out(M3, Matrix_ris_);
22     }<>; } }
23 proc f_mul(in long A[N], long B[N] output_stream long Res)
24 $c++{ register long r=0;
25   for (register int k=0; k<N; ++k)
26     r += A[k]*B[k];
27   assist_out(Res,r); }c++$
```

The parmod exhibits a matrix of NxN VPs (SPMD, line 3). Once the two input matrixes are received (line 8), they are both scattered to the VPs which store them in the distributed shared matrixes A and B (lines 9–10) that has been previously declared (lines 4–5). Then, all elements of the result matrix C are computed in parallel (lines 12–14). Once all VPs completed the operation, a result matrix is collected from the distributed matrix C and sent into the output stream (lines 15–22). The code of sequential modules is not shown for the sake of brevity.

It is worth pointing out that the ASSIST programmer is not requested to code any low level detail of the application, especially those typical, cumbersome details of grid programming such as concurrency activities set up and mapping (firewalls, multi-tier networks with private address ranges), communication and synchronization protocols, dynamic QoS control under critical conditions (faulty or overloaded platforms and network links). The ASSIST compiler and its run-time support provide these features. More details on these features of ASSIST environment can be found in [4,3,2,5].

3. Module assembly via Web Services

The ASSIST compiler translates each module into a network of processes. Sequential modules are translated into sequential processes, while parmods are translated into a parametric (w.r.t. the parallelism degree) network of processes. Communications within a parmod are implemented by using ASSIST native communication libraries (relying on either plain TCP/IP or Globus); communication parameters and patterns are established and optimized at compile time. Communication among parmods (i.e. streams, depicted as arrows in Fig. 1) a) may be configured according to several standard protocols, among the others the native ASSIST streamlib (TCP/IP), and HTTP/SOAP. In the latter case the compiler automatically produces the code needed to attach a client/server pair to each stream ends. In particular:

- an XML file for each module describing the module interface, i.e. module name, input and output stream types.
- a gateway process for each input stream of each module. The gateway implements a gSOAP-based Web Service exporting a single asynchronous method [14]. The method invocation is meant to carry a stream item as method parameter. A suitable WSDL is generated.

- an *invoke* method for each output stream of each module. The invoke function is triggered by any output request on the output stream. Since the WSDL of the target Web Service will be known only at run-time, the implementation of the invoke is generated on the fly at run-time (after the wiring), and dynamically linked to the code.

- a *configure* method related to each invoke. It enables to overwrite the IP address of the Web Server the invoke connects to. This enable the independent deployment and run of modules and module run-time mobility (for fault tolerance or performance reasons).

In addition, the application graph is translated in an assembly of services described in a XML file. This information is used at launch time to wire modules one another. As example the following file describes the application in Fig. 1 a):

```
1  <ComponentConfiguration>
2   <Assembly>
3    <ComponentSection>
4     <Component name="send1"  com="ws" kind="xml" file="./xmls/send1.xml"> </Component>
5     <Component name="send2"  com="ws" kind="xml" file="./xmls/send2.xml"> </Component>
6     <Component name="matrix_mul"  com="ws" kind="xml" file="./xmls/matrix_mul.xml"> </Component>
7     <Component name="recv"  com="ws" kind="xml" file="./xmls/rec.xml"> </Component>
8    </ComponentSection>
9    <ConnectionSection>
10    <Connection>
11     <Output component="send1" interface="Matrix1"/>
12     <Input component="matrix_mul" interface="Matrix1"/>
13    </Connection>
14    <Connection>
15     <Output component="send2" interface="Matrix2"/>
16     <Input component="matrix_mul" interface="Matrix2"/>
17    </Connection>
18    <Connection>
19     <Output component="matrix_mul" interface="Matrix_ris"/>
20     <Input component="recv" interface="Matrix"/>
21    </Connection>
22   </ConnectionSection>
23  </Assembly>
24 </ComponentConfiguration>
```

The file does not directly contain any mapping of deployment information. The ASSIST launcher exploits heuristics to decide the mapping of modules onto platforms at launch time taking in account the kind of available platforms, user hints, application structure and grid current status [1,8]. Then, it relies on middleware (e.g. Globus) mechanisms to deploy and run each module and to wire them one another (see Fig. 1 b).

3.1. Web Services payback and overhead

The ASSIST programming environment, extended as described in the previous section, enables the programmer to describe an application as a composition of services. From an abstract viewpoint, such composite services exhibit the same functionality of the original ASSIST application. As matter of a fact, the only difference consists in protocol used to carry data from one module to another, which relies on HTTP/SOAP in the Web Service implementation. From a practical viewpoint, this may considerably ease application deployment since Web Services are usually allowed to cross site boundaries without any specific intervention on firewall configuration, and they are platform neutral. On the other hand, the overhead due to message marshalling and HTTP parsing may range from high to average depending on the specific marshalling technique.

As an example, Fig. 2 reports the comparison of the execution times achieved when running the same application (sketched in Fig. 2 left) using plain ASSIST (plain TCP/IP communications) or exploiting Web Services to run the *workers*, that is, the processes actually computing results out of (independent) input data items. The application processed 1K tasks (flowing from Dist to Coll, see Fig. 2), taking each ~ 65 milliseconds to be computed and requiring to serialize/marshall some

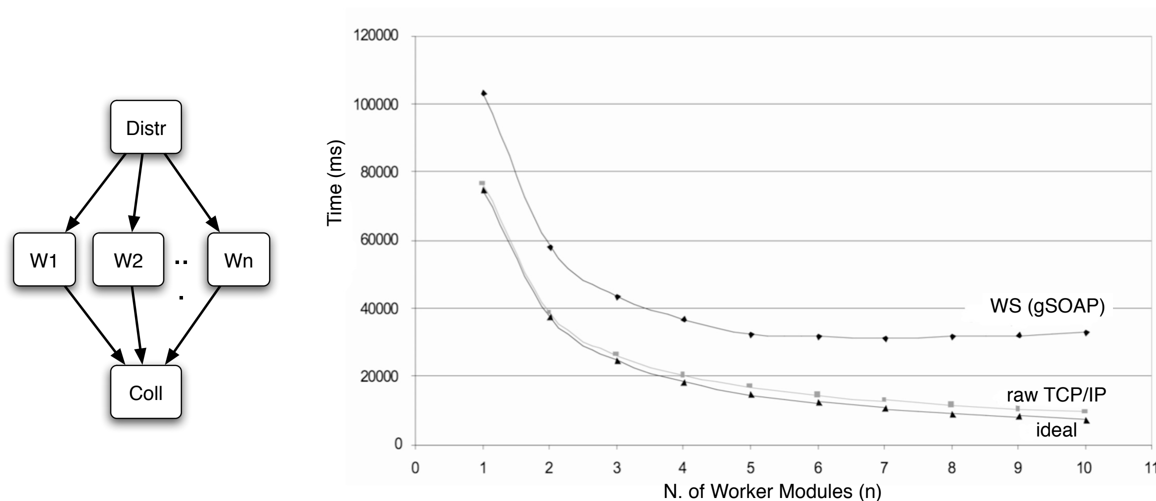

Figure 2. Plain TCP/IP and HTTP/SOAP protocols compared on a master-worker application.

Kbytes of data. The performance/scalability of the Web Service implementation is not extremely far from the one achieved using plain TCP/IP.

Differently from BPEL, the orchestration of services in an ASSIST application is fully decentralized. Since one-way streams connect ASSIST modules, all modules implement asynchronous Web Services. This solution may appear a bit unnatural when dealing with Web Services, as Web Services commonly exploit synchronous RPCs. However, this is also the underlying idea of the most promising proposals for efficient decentralized execution of BPEL-like workflows [17]. In particular the performance edge with respect to synchronous solution is likely to become a key factor for grid-aware high-performance applications. On the other hand, full interoperability with external, legacy Web Servers is very attractive. ASSIST modules support both one-way and RPC style HTTP/SOAP interaction as both client and server. Any ASSIST module (or a graph of them) can be wrapped and packaged into a so-called GRID.it component. As we shall see in the next section, a GRID.it component may inter-operate with other components both by means of one-way and RPC (use/provide) ports.

3.2. Component based grid programming: ASSIST and GRID.it

Some interesting, component based programming models have been proposed to be used in the grid context. In particular, the CORBA Component Model (CCM [13]) and the Common Component Architecture (CCA) component model [6] have been widely discussed in the grid context. Other models, coming from different experiences, such as JavaBeans, Web Services and Microsoft .NET are currently being considered in the field of grid programming although neither the web services model nor .NET can be properly called component models. In the context of the GRID.it project[2], our group introduced a fairly new component based programming model. Components can be either parallel or sequential. Legacy CCM components and WWW Web Services are assumed to be usable as sequential GRID.it components via proper wrapping. GRID.it components interact using three basic mechanisms, two inherited from previous component models and one which is brand new:

- classical use/provide ports are basically used to implement RPC-like component interaction.

[2]GRID.it is a three-year project, ending in 2005 that involves major Italian universities and research institutions [15]

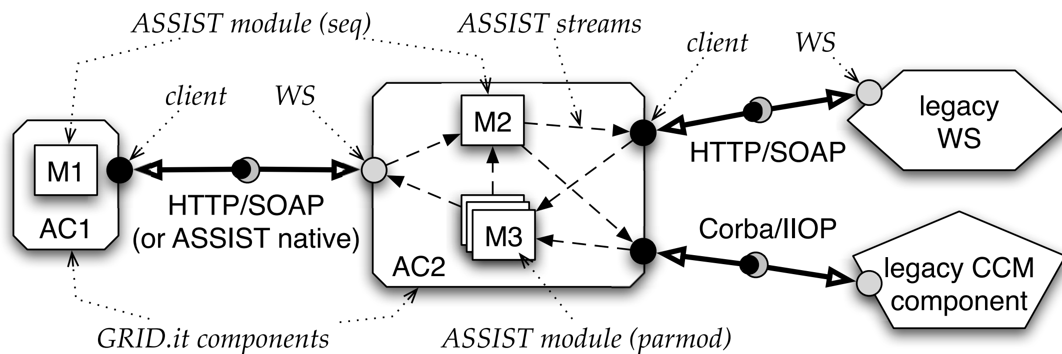

Figure 3. A grid application as an assembly of GRID.it components, CCM components, and legacy Web Server.

- events inherited from CCM, are basically used for component synchronization.
- data flow streams, a new mechanism that it is used to implement efficient, one-way data flow communication between components, are used to provide a way to transfer typed data items from one component (exporting a stream source interface) to another one (exporting the stream sink interface).

Even though data flow streams can be easily implemented in terms of either use/provide ports or events, they have been explicitly included in the set of primitive mechanism to enforce the concept that they provide optimized, high performance inter-component communication mechanisms. All these mechanisms are used to implement two distinct component interfaces:

- the functional interface exposing the component functional behavior to the other components. Using the mechanisms implemented in this interface a component can use the services provided by another component to actually compute a result
- the non-functional interface providing mechanisms that can be used to control the component behavior, that is, its execution features as well as its interaction with the underlying grid target architecture (not discussed in this paper, see [2,5]).

An ASSIST module (or a graph of modules) can be declared as a GRID.it component [2,3]. The ASSIST compiler automatically performs the wrapping of an ASSIST application graph into a GRID.it component membrane. As sketched in Fig. 3, a GRID.it component (e.g. AC1, AC2) may wrap any graph of ASSIST modules and it is characterized by provide (grey circles) and use ports (black circles). They may behave both in one-way/event and RPC-like manner, and may support several protocols, such ASSIST native, HTTP/SOAP , and CORBA/IIOP. These protocols enable the interoperability with other standard component models and paradigms.

3.3. GRID.it components and Web Services

All kind of GRID.it ports can be connected through HTTP/SOAP channels. Events and stream ports can be easily realized as described in section 3 since they are basically one-way channels. As RPC ports concern:

- a provide port behaves essentially as a (single method) stateful Web Service. When declaring a GRID.it component it is possible to bind a provide port to an ASSIST input stream and to

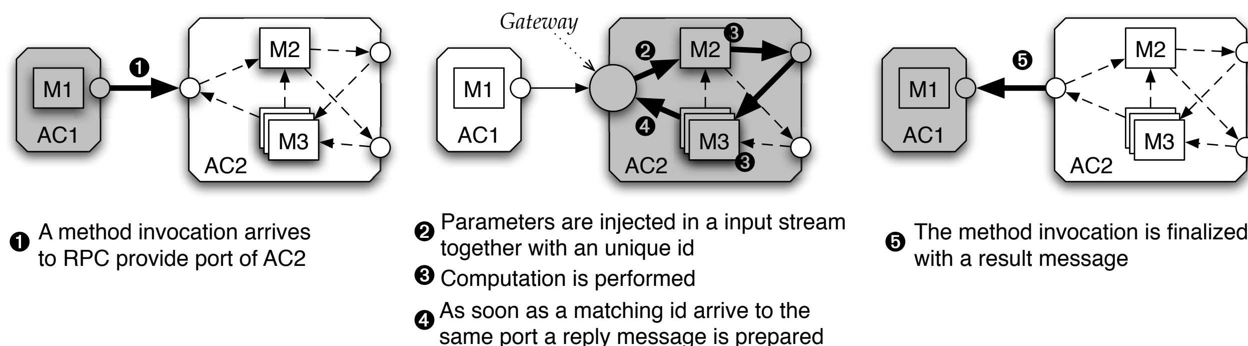

Figure 4. Evolution of a method call via Web Service in a GRID.it component.

an output stream within the component. An automatically generated gateway process, which is attached to the port and run the Web Server, collects method invocation from the provide port and injects the method parameter into the input stream bind to it. This data is properly colored with a unique identifier, which is carried along the entire ASSIST graph within the component. When the gateway collects a data item from the output stream matching the color of some ongoing method invocation in the port, a SOAP envelope is prepared and sent as service result. Coloring mechanism enable to distinguish possibly concurrent service requests on the same port. The succession of events is sketched in Fig 4.

- use ports are used as clients for Web Services. In this case is not possible to automatically generate all the code needed to invoke an external, possibly legacy Web Service because it depends on the service the port will be wired to (number and name of the methods, address, etc.). The GRID.it framework helps the programmer providing him with a proxy library whose entries are the stub methods for the remote Web Service. The library is generated starting from the target Web Service WSDL. The programmers have the responsibility to fulfill the stubs with the code needed to invoke the Web Service methods, and to place the calls to the stub methods within the sequential code of the ASSIST modules within the GRID.it components.

This solution has been already implemented and fully tested. It provides a bridge between ASSIST applications and GRID.it components and furtherly increases the interoperability opportunities deriving from the adoption of Web Service based mechanisms in the ASSIST/GRID.it grid programming environments.

4. Conclusions

The ASSIST environment provides a high-level programming alternative to the classic grid programming figure assuming that applications are built on top of grid middleware directly using/invoking the middleware functionalities at the user code level. As shown in previous works [4,3,2,5], ASSIST and GRID.it components can cope with many of the key issues of the grid programming while avoiding application programmers to entangle with the related cumbersome details such as software deployment. We shown that ASSIST modules can be composed as services with decentralized orchestration by using standard toolkits of Web Services development, and that all the code needed to turn ASSIST module interaction into Web Services can be automatically generated with no additional effort for the programmer. Also, we shown that ASSIST modules, wrapped as GRID.it components, can expose their functionalities as Web Services with any programmer intervention.

References

[1] M. Aldinucci and A. Benoit. Automatic mapping of ASSIST applications using process algebra. In *Proc. of HLPP2005: Intl. Workshop on High-Level Parallel Programming*, Warwick University, Coventry, UK, July 2005.

[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 V. Getov and T. Kielmann, editors, *Proc. of the Workshop on Component Models and Systems for Grid Applications*, CoreGRID series. Springer Verlag, January 2005.

[3] M. Aldinucci, M. Coppola, S. Campa, M. Danelutto, M. Vanneschi, and C. Zoccolo. Structured implementation of component based grid programming environments. In V. Getov, D. Laforenza, and A. Reinefeld, editors, *Future Generation Grids*, CoreGRID series. Springer Verlag, November 2005.

[4] M. Aldinucci, M. Coppola, M. Danelutto, M. Vanneschi, and C. Zoccolo. ASSIST as a research framework for high-performance Grid programming environments. In J. C. Cunha and O. F. Rana, editors, *Grid Computing: Software environments and Tools*. Springer Verlag, January 2006.

[5] 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*, volume 3648 of *LNCS*, pages 771–781, Lisboa, Portugal, August 2005. Springer Verlag.

[6] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Toward a common component architecture for high performance scientific computing. In *Proc. of the 8th Intl. Symposium on High Performance Distributed Computing (HPDC'99)*, 1999.

[7] OASIS Technical Committee. WSRF: The Web Service Resource Framework Globus web site. http://www.oasis-open.org/committees/wsrf.

[8] M. Danelutto, M. Vanneschi, C. Zoccolo, N. Tonellotto, S. Orlando, R. Baraglia, T. Fagni, D. Laforenza, and A. Paccosi. HPC application execution on grid. In *Dagstuhl Seminar Future Generation Grid 2004*, CoreGRID series. Springer-Verlag, 2005. To appear.

[9] J. Dünnweber, S. Gorlatch, M. Aldinucci, S. Campa, and M. Danelutto. Behavior customization of parallel components application programming. Technical Report TR-0002, Institute on Programming Model, CoreGRID - Network of Excellence, April 2005.

[10] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke. The physiology of the grid. http://www.globus.org/research/papers/, June 2002.

[11] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid: Enabling scalable virtual organization. *The Intl. Journal of High Performance Computing Applications*, 15(3):200–222, Fall 2001.

[12] S. Graham, S. Simeonov, T. Boubez, G. Daniels, D. Davis, Y. Nakamura, and R. Neyama. *Building Web Services with Java: Making Sense of XML, SOAP, WSDL, and UDDI.* SAMS, 2001.

[13] Object Management Group. Corba component model version 3.0 specification. http://www.omg.org, September 2002.

[14] gSOAP C++ Web Services home page. http://www.cs.fsu.edu/∼engelen/soap.html, 2003.

[15] The GRID.it home page. http://www.grid.it.

[16] K. Kennedy, M. Mazina, J. Mellor-Crummey, K. Cooper, L. Torczon, F. Berman, A. Chien, H. Dail, O. Sievert, D. Angulo, I. Foster, D. Gannon, L. Johnsson, C. Kesselman, R. Aydt, D. Reed, J. Dongarra, S. Vadhiyar, and R. Wolski. Toward a framework for preparing and executing adaptive Grid programs. In *Proc. of NSF Next Generation Systems Program Workshop (IPDPS 2002)*, 2002.

[17] M. G. Nanda, S. Chandra, and V. Sarkar. Decentralizing execution of composite web services. In *Proc. of Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Vancouver, B.C., Canada, October 2004. ACM.

[18] M. Vanneschi. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Computing*, 28(12):1709–1732, December 2002.