On Building Parallel & Grid Applications: Component Technology and Distributed Services

Dennis Gannon, Sriram Krishnan, Liang Fang, Gopi Kandaswamy,
Yogesh Simmhan, Aleksander Slominski

Department of Computer Science, Indiana University
Bloomington, IN
gannon@cs.indiana.edu

Abstract

Software Component Frameworks are well known in the commercial business application world and now this technology is being explored with great interest as a way to build large-scale scientific application on parallel computers. In the case of Grid systems, the current architectural model is based on the emerging web services framework. In this paper we describe progress that has been made on the Common Component Architecture model (CCA) and discuss its success and limitations when applied to problems in Grid computing.

Our primary conclusion is that a component model fits very well with a services-oriented Grid, but the model of composition must allow for a very dynamic (both in space and it time) control of composition. We note that this adds a new dimension to conventional service workflow and it extends the "Inversion of Control" aspects of must component systems.

1. Introduction

A **Software Component Framework** provides a way to build applications by composing them from predefined and tested units of code called "components". The need for component frameworks is driven by the complexity of the programming tasks we see today. It is no longer conceivable that a single programmer can build a large application by writing every line of code. Professional programmers work by composing elements from large collections of libraries made available from third parties or the vast open-source community. In the area of scientific applications the multidisciplinary nature of our work has driven us to work in large teams of specialists. Though slower to change than the rest of the software world, scientific community has reached the software complexity threshold where a profound change in programming practice is required to build the next generation of high performance, multidisciplinary applications.

In 1995 an effort began within a small group of collaborators to define a component architecture for scientific applications that ran on massively parallel systems. This effort gained support from the Office of Science of the U.S. Department of Energy and it has grown to be a consortium of researchers from about 15 universities and laboratories. The resulting specification, called the Common Component Architecture (CCA) [7,8,10] has been implemented in several different frameworks [14,15,17,21,31].

At the same time that CCA has been developed and tested for applications on high performance parallel computers, another group has been looking at applications that span multiple resources in "Grid" environments [26]. These Grid applications have several additional dimensions of complexity because they are more heterogeneous, dynamic and distributed in both space and time. In addition, factors such as security are often as critical as performance. The emerging Grid architecture is based on web services and, as we shall show, this provides an interesting foundation for a Grid CCA.

In the paragraphs that follow we will describe some of the formal properties of component architectures that make them useful and some of the progress made on CCA. We will then describe the difficult problems that must be solved to make the architecture work in the Grid environment and discuss several solutions to these problems.

2. Software Component Frameworks

What sets component architectures apart from standard class or subroutine libraries are a few basic properties.

1. A **component** is a service that is defined by a set of interfaces that it implements. A user of a component invokes the functions defined by the services interfaces and the appropriate actions are taken by the component implementation. In most cases this service is considered to be stateless, but in some cases the component interacts with a resource, such as a database which has state.
2. Each **component framework** provides a component container or context in which components are instantiated and composed. In some systems the framework often also provides a set of standard services (implicit components) that can be used by the instantiated components.

A critical feature of component frameworks is that applications can be built by composing components and, because the components are designed to follow a specific set of behavior rules, the composed application works as expected. For example, an important feature of component frameworks that differs from many standard programming models, is the use of a design pattern called **Inversion of Control (IOC)**. The basic principle of IOC has two parts. First, application control is never distributed over or vested in more than one driver component of the application, and, in some systems, the control flow of the application lies completely within the framework "container". In its purest form IOC also implies that a component instance has a lifecycle and environment that is completely managed by the framework. Everything the component needs is supplied by the component. One aspect of this idea, as argued by Fouler [5], involves **Dependency Injection**, which is the concept that applications invoke services but the instantiation of the component that implements this service is determined by the framework at runtime. In other words, the dependency of one component instance upon another is injected into the system at the latest possible time.

Another type of behavior rule that many component systems enforce is a standard way for a framework to learn about a component at runtime. This type of component introspection is what allows a framework to discover that a component actually implements an interface required by an application.

The earliest component frameworks with many of this properties included Microsoft COM, the Java bean model and the CORBA Component Model [6]. Szyperski [12] provides a description of many other component system properties and features. More recently the complexity of the Enterprise Java Bean framework has spawned other frameworks like Spring [24] to simplify its programming model. Pico [23] and the Apache Avalon [25], which is a server side framework for Apache, are also important component frameworks.

In the case of scientific applications, early component architectures include the Model and CODE frameworks [1,2]. In visualization applications the most important examples are the AVS system [36] and the SciRun [22] framework, which now also implements the CCA model described in greater detail below. Webflow [26] was an early component model for distributed systems for scientific applications, and, more recently, the Discover project [3,4] considers the problem in the context of Grid systems.

3. CCA

The Common Component Architecture is defined by a set of framework services and the way components are defined. Each component communicates with other components by a system of "ports". Ports are defined by a type system, which is expressed in "Scientific Interface Definition Language" (SIDL). SIDL provides simple way to describe a method interface in terms of the data \types common in scientific computing. (The associated tools that are part of the Babel Toolkit [13] allow powerful language interoperability between Fortran, C++, and Python.) There are two types of CCA ports:

1. **Provides Ports** are the services offered by the component. Each provides port implements an interface defined in IDL.
2. **Uses Ports** are component features that implement the use of a service provided by another component. They bound to the "stubs" that a component "uses" to invoke the services of another port "provided" by another component. Uses ports are also defined by IDL.

As illustrated in Figure 1 below, a uses port on one component can be connected to the provides port of another component as long as they implement the same SIDL interface. The connection is made by the framework operation "connect" at runtime. When a

component wants to use a service connected to one of its "uses" ports, it executes a framework "getPort" operation. This provides an implementation of the port or blocks until one is available. When the component is finished it issues a "releasePort" operation.

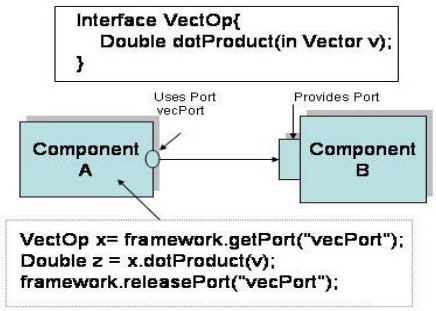

Figure 1. Component A has a uses port called vecPort which it uses to compute the dot product of a vector with itself. The port is defined by the SIDL interface VectOp. Component B has a provides port that implements the VectOp interface and the vecPort uses port from component A is connected to the provides port from B.

A consequence of this get/release semantics is that component connections may be changed at runtime, which is especially useful in dynamic environments.

.

4. The Grid Service Architecture

We can define a Grid to be a distributed collection of resources and services, which operate together as a single system. The services include:

- Data Services – archives of file objects, databases, streaming data sources such as those generated from on-line instruments, directories and indexes.
- Security Services – the mechanisms used by a service to authenticate users and user agents and to manage their authorization to use other services.
- Compute Services – the mechanisms that enable a user Grid application instance to schedule units of work on the computing hardware that underlies the grid.
- Messaging services – the mechanisms uses that allow an application component to subscribe to notification about events generated by other parts of the application or the Grid itself.

This is only a partial list. The precise Grid architecture of services is currently being defined in detail by the Global Grid Forum, Open Grid Service Architecture (OGSA) Working Group. Each service in this Grid architecture is implemented as a Web Service. In the first vision for OGSA [9], these web services all conformed to the Open Grid Service Infrastructure (OGSI) [20] specification, but that has now been replaced by the WS-Resources Framework proposal [35] being considered by OASIS. There are two important properties of OGSI/WS-RF that are critical for building a component architecture.

1. OGSI/WS-RF has a standard mechanism to record and report service metadata. This provides a uniform way to do runtime component introspection.
2. WS-RF contains a specification called WS-Notification that provides all the mechanisms need to implement a "publish-subscribe" messaging systems.

5. Building an Application Component Framework on top of OGSA.

XCAT is our CCA compliant Application Component Framework that is built on top of a web service foundation. XCAT provides ports are implemented as OGSI web services. (We are developing plans for a WS-RF based version.) XCAT uses ports can accept a connection to any properly typed XCAT provides port but also to any OGSI compliant web service [19,21]. Given this capability, what is involved in building distributed Grid applications using this system? What are the specific technical problems that are solved and what additional problems are encountered? We consider the following to be the most important issues.

1. How does one program a "Grid application"? Some components are linked uses-port-to-provides-port as described in the classical CCA model, but others consist of a set of components that are run at different time and their connection is more implicit and the logic is driven by sequences of events.
2. How does one use a Grid application? The applications often take the form of services that are access through a Grid portal. The user supplies parameters and the application is launched. How do we control multiple users trying to invoke the application at the same time?
3. How can traditional applications be included as components in this framework? For example, an HPF based Fortran application running on a parallel supercomputer?

4. How do we deal with very dynamically changing resource environments? In a conventional virtual memory operating system, applications migrate between physical memory and disk. Is there an analog for Grid applications that may run for days and need many resources to execute?
5. How do we handle user authentication and authorization for Grid applications? If I build a grid application that uses a variety of remote services that I am authorized to use, how can anybody else use this application unless they have the same privileges?

We address each of these questions below.

5.1 Programming Grid Applications: Control in a Distributed Framework.

An application component architecture for a Grid must assume that an application instance consists of coupled component instances that are physically distributed in both space and time. By this we mean the individual component instances may be running on different hosts and they communicate over the network. And, because of the nature of Grid workflow constraints they may be actually executing at very different times.

To understand this issue, consider the basic IOC model we assume for a component-based application. A framework or application controller component sequences the execution of component interactions. The CCA framework assumes that components interact through *uses* and *provides* ports. Because XCAT has ports that are Grid services, this means that connected components may reside on different hosts (see Figure 2) and the connection between the components can be implemented using standard web service messaging protocols such as SOAP.

In the case that the components are all running simultaneously and coupled via a system of uses-to-provides port connections, CCA application controllers can be written as a traditional C++ or Java application. Another approach that is very popular is to use a scripting language like Python.

An attractive alternative is to have a graphical representation for interconnected set of components. Users can freely add new components to the workspace (Figure 3.), connect them together, modify their properties and press a go button to have an application corresponding to this visual description executed. The visual description is based on a graph where nodes are component instances and edges are connection between component instances. Such graph describes all necessary information to proceed with execution.

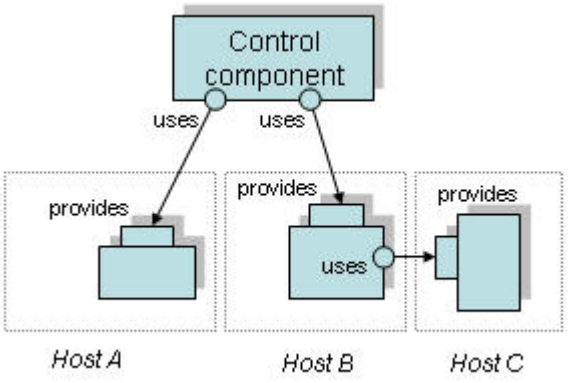

Figure 2. A Distributed Component Framework requires a remote message architecture to connect users to providers.

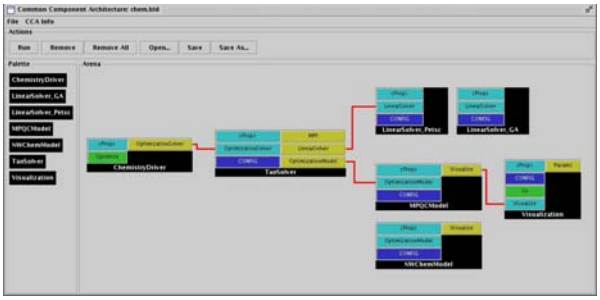

Figure 3. The Graphical User Interface for the Ccaffeine [17] CCA component application builder (from the CCA tutorial [18])

In many cases components may not actually be executing at the same time. For example, one application that we are working on involves severe storm modeling. In this case some components are data mining streams of data from sensors. When the mining components detect significant "bad weather" the application will invoke a number of simulation applications to predict the progress of the storm. As these applications complete, visualization processes turn the output into movies for the scientist to study. In this case we are now controlling Grid workflow (see Figure 3.). The important difference between the simultaneous execution model (composition in space) and the scheduling of tasks model (composition in time) lies in the way we manage communication. Composition in space typically follows an RPC model: the user invokes a service provided and waits for a response. In the case of composition in time communication is by message notification.

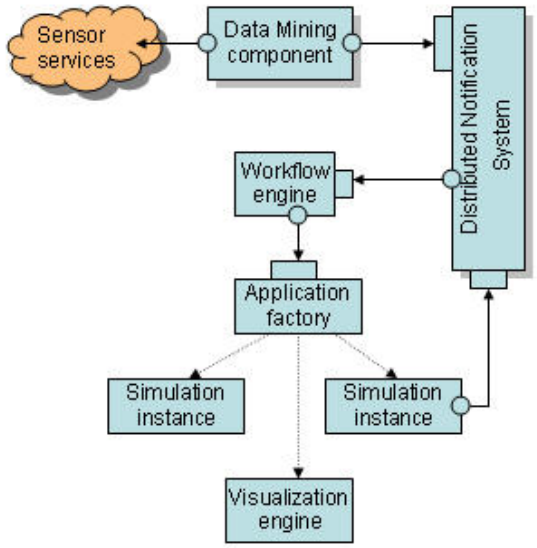

Figure 4. A workflow driven application. Data mining components poll sensor services and generate notification events when something interesting happens. This causes the workflow to be activated. The workflow script contacts the application factory to launch simulation components. Simulation events are relayed to the notification system, which notifies the workflow to instantiate the visualization components.

Fortunately, the web services model completely supports both communication models. A notification framework is a persistent service that provides ports for message delivery and subscription.

To "program" time-based composition we need to be able to describe the order in which components are created, used, and destroyed. Instead of reverting to scripts, it is worthwhile to consider more powerful graph languages that can be used to describe workflows. Workflows have rich body of research that is built on well understood concepts like Petri nets to formalize workflow description and to enable mathematical analysis. Moreover there are many existing workflow languages, products and tools that can be used. However they lack interoperability and rarely work well in Grid environments as most of them was created for business production workflows.

An ideal workflow language should have strong graph composition capabilities to make it possible to use it with visual workflow design tools. It should also have simple and easy to understand procedural constructs such as loops to avoid limitations of pure graph language. And it should work with Web services and Grid services.

We think that BPEL4WS [37] is a strong starting point for a Grid workflow composition language. Even though it was not designed for Grid environments, it is extensible enough to allow its use in OGSI and WSRF based Grids. We are currently investigating ways in which BPEL4WS can be adapted to work in WSRF Grids and in particular to compose XCAT components. We are considering ways BPEL4WS can be made more dynamic.

5.2 Using Grid applications: Portals and Factories

The use model for Grid applications is also different from the case where one scientist can simply give a code to another. Because the application is a scripted composition of services, there may not be much of a "code" to give away. And when there is code that can be moved, it is often extremely difficult to provide it in a way that is easy to use by somebody else. For Grid applications, complex dependencies on the users' environment on each host are usually the most difficult problem to solve.

A better approach is to provide the application as a service. NetSolve [27] and Ninf [28] were the first systems to exploit this concept. However, their approach only works if they services are stateless, and are only meant to provide certain functionality on the Grid. An approach we advocate for applications on the Grid that have state is the use of Application Factories [39]. In XCAT, Application Factories provide a set of pre-packaged applications consisting of a set of distributed components. Authorized users can connect to the factory service from a user portal and request an instantiation of an application. The factory service would then instantiate all the components that are part of the application, and connect them together to compose an instance of the distributed application. The user can then control the application using a reference to the Application Coordinator that is returned by the factory service after successful instantiation of the application.

5.3 Traditional Applications as Components

There have been several efforts at component-izing traditional scientific applications. CCA has been used in the context of parallel PDE solvers [16], earth systems modeling [30,33,34] and for other high performance scientific simulation codes [11,32]. However, this calls for rewriting legacy scientific codes as components. Another approach is to use the concept of application managers [38].

In the application manager approach, every scientific application is wrapped by a generic

application manager component, which is responsible for managing the execution of the application, monitoring its status, and staging its input and output data. The application managers can then be accessed via standard grid mechanisms.

An example of a chemical engineering application using the application manager approach is illustrated in Figure 5.

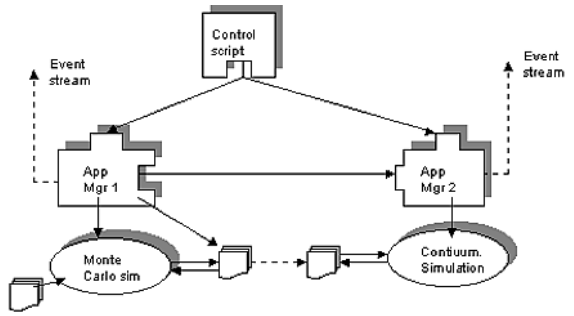

Figure 5. Two coupled Chemical Engineering simulation programs. Application Manger 1 signals Application Manger 2 when the Monte Carlo simulation completes a time step and the associated output state files have been migrated. Upon receipt of the message Application Manager 2 runs the continuum simulation. When this terminates, control is returned to Application Manager 1 for the next iteration

5.4 Dynamic Environments

The resources in a Grid-based environment can be highly dynamic. This is especially problematic when the applications are long running in nature. The applications on the Grid, hence, have to be able to adapt to the runtime environment, so that they can provide acceptable Quality of Service (QoS) to the user. The dynamic nature of the Grid causes several challenges – scheduling of the grid application such that applications get the resources they desire and the throughput of the system is maximized, providing fault tolerance for long running applications especially since resources can fail at any time, providing the ability for components to migrate to better resources as they become available. We address these issues in the following paragraphs.

Scheduling long running applications in a Grid system is difficult since it is difficult to predict the resource requirements and characteristics at job submission time. Projects such as Condor [29] provide scheduling services on the Grid. In addition, Condor also provides an ability to checkpoint an application so that (a) an application can start at the latest checkpoint if the resource on which it is executing crashes, and (b) the application can be migrated to another resource if either the resource is not capable of providing acceptable performance or if the resource decides to evict the application due to some policy violations.

However, the job of checkpointing and migration of an application is more complicated if the components those constitute an application are distributed over various resources (such as in our case). Checkpointing the components individually will not preserve consistency of global state, and distributed checkpointing algorithms have to be used for the same.

Currently, efforts are underway within the XCAT project to provide distributed checkpointing for fault tolerance of an application, and also to provide migration of individual components. The ability to migrate can be used in conjunction with traditional schedulers to provide better throughput in a dynamic environment.

5.5 Security

Security is an important issue in the Grid since the services and components are distributed all across the network. There are three primary issues to address:

1. Authentication – This is the step where the identity of the caller (i.e. the client who invokes the Grid service) is established
2. Authorization – This is the step where a decision is made whether a client does or does not have the right of invoking a service, once his/her identity is established.
3. Confidentiality – This is the step where all data that is sent between a caller and the callee is encrypted such that no person other than the intended receiver can make sense of it.

In the Grid world, authentication and encryption is generally addressed using X.509 certificates and public key cryptography. Authorization is generally provided by using Access Control and/or Revocation lists that provide or revoke privileges to users.

However, there are two popular approaches to security using the above technologies.

1. Transport Level – This approach advocates creation of a secure channel between the caller and callee prior to the invocation of a remote call (via the use of technologies such as SSL/TLS).
2. Message Level – This approach is advocated by the WS-Security [40] specification. WS-Security provides mechanisms to provide authentication, integrity and confidentiality of

a message, irrespective of the transport mechanism used.

We are currently analyzing the viability of each of the two approaches.

6. Conclusions

In this paper we have attempted to outline the primary design issues that are critical for building a distributed software component architecture for Grid applications. Unlike the non-distributed case, where software components typically all reside in the same address space and communicate by simple method or function invocation, the distributed case requires a very flexible communication model. The XCAT system we have built will allow components to be Grid web services which communicate by direct SOAP messages or indirectly through a notification service. This allows components in a application to be very dynamic at runtime. As services they are "virtually" always available, but exact host they are running on may not be known until they are actually invoked. And, they may move when resource constraints require them to do so. We have also observed that security is a critical issue that separates the Grid applications from more traditional scientific codes. Web service security protocols allow us to provide both authentication and authorization at the message level in order to make dealing with security easy for the application programmer.

7. References

[1] J. C. Browne, S. I. Hyder, J. Dongarra, K. Moore, P. Newton. Visual Programming and Debugging for Parallel Computing. *IEEE Parallel and Distributed Technology, Spring 1995, Volume 3, Number 1, 1995.*

[2] P. Newton and J.C. Browne. The CODE 2.0 Graphical Parallel Programming Language. *Proc. ACM Int. Conf. on Supercomputing, July 1992*.

[3] V. Bhat and M. Parashar. Discover Middleware Substrate for Integrating Services on the Grid. *Proceedings of the 10th International Conference on High Performance Computing (HiPC 2003), Lecture Notes in Computer Science*, Editors: T.M. Pinkston, V.K. Prasanna, Springer-Verlag, Hyderabad, India, Vol. 2913, pp 373 – 382, December 2003

[4] M. Agarwal and M. Parashar . Enabling Autonomic Compositions in Grid Environments.*Proceedings of the 4th International Workshop on Grid Computing (Grid 2003),* Phoenix, AZ, USA, IEEE Computer Society Press, pp 34 - 41, November 2003

[5] Martin Fowler. Inversion of Control Containers and the Dependency Injection pattern. January 2004, http://www.martinfowler.com/articles/injection.html

[6] CORBA Component Model. http://www.omg.org/ technology/documents/formal/components.htm.

[7] Rob Armstrong, Dennis Gannon, Al Geist, Katarzyna Keahey, Scott Kohn, Lois McInnes, Steve Parker, and Brent Smolinski. Towards a common component architecture for high performance scientific computing. In *Proceedings of the The Eighth IEEE International Symposium on High Performance Distributed Computing*, 1998.

[8] David Bernholdt, et al. A Component Architecture for High-Performance Scientific Computing. To appear.

[9] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. Grid Services for Distributed System Integration. *Computer 35(6),* 2002.

[10] CCA specification. http://cca-forum.org/specification.

[11] David E. Bernholdt, Robert C. Armstrong, and Benjamin A. Allan. Managing complexity in modern high end scientific computing through component-based software engineering. In *Proc. of HPCA Workshop on Productivity and Performance in High-End Computing (PPHEC 2004), Madrid, Spain*. IEEE Computer Society, 2004.

[12] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press, New York, New York 10036, 1999.

[13] Tammy Dahlgren, Tom Epperly, and Gary Kumfert. *Babel User's Guide*. CASC, Lawrence Livermore National Laboratory, version 0.8.4 edition, April 2003.

[14] Gary Kumfert. Understanding the CCA specification using Decaf. Technical Report UCRLMA-145991, Lawrence Livermore National Laboratory, 2003. http://www.llnl. gov/CASC/components/docs.html.

[15] David E. Bernholdt, Wael R. Elwasif, James A. Kohl, and Thomas G. W. Epperly. A component architecture for high-performance computing. In *Proceedings of the Workshop on Performance Optimization via High-Level Languages and Libraries (POHLL-02)*, 2002.

[16] B. Norris, S. Balay, S. Benson, L. Freitag, P. Hovland, L. McInnes, and B. Smith. Parallel components for PDEs and optimization: Some issues and experiences. *Parallel Computing*, 28 (12):1811–1831, 2002.

[17] Benjamin A. Allan, Robert C. Armstrong, Alicia P.Wolfe, Jaideep Ray, David E. Bernholdt, and James A. Kohl. The CCA core specification in a distributed memory SPMD framework. *Concurrency and Computation: Practice and Experience*, 14(5):1–23, 2002.

[18] CCA Tutorials. http://www.cca-forum.org/tutorials/

[19] Madhusudhan Govindaraju, Sriram Krishnan, Kenneth Chiu, Aleksander Slominski, Dennis Gannon, Randall Bramley. Merging the CCA Component Model with the OGSI Framework. In *3rd IEEE/ACM International Symposium on Cluster Computing and the Grid*, May 12-15, 2003.

[20] The Open Grid Services Infrastructure Working Group. http://www.gridforum.org/ogsi-wg, 2003.

[21] Sriram Krishnan and Dennis Gannon. XCAT3: A Framework for CCA Components as OGSA Services. In *Proceedings of HIPS 2004, 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, April, 2004.

[22] S.G. Parker and C.R. Johnson. SCIRun: A scientific programming environment for computational steering. In *Supercomputing '95*. IEEE Press, 1995.

[23] The Pico Framework, http://www.picocontainer.org
[24] The Spring Project. http://www.springframework.org
[25] The Avalon Project. http://avalon.apache.org/.
[26] Dimple Bhatia, Vanco Burzevski, Maja Camuseva, Geoffrey Fox, Wojtek Furmanski, Girish Premchandra WebFlow: A Visual Programming Paradigm for Web/Java Based Coarse Grain Distributed Computing (1997) .Concurrency - Practice and Experience.
[27] Henri Casanova and Jack Dongarra, NetSolve: a network server for solving computational science problems. Proceedings SC 96.
[28] Satoshi Matsuoka, et. al., Ninf: A Global Computing Infrastructure**,** http://ninf.apgrid.org/welcome.shtml
[29] Todd Tannenbaum, Derek Wright, Karen Miller, and Miron Livny. Condor - A Distributed Job Scheduler. *In Beowulf Cluster Computing with Linux*, The MIT Press, 2002.
[30] Earth System Modeling Framework (ESMF). http://sdcd.gsfc.nasa.gov/ESS/esmf_tasc.
[31] Felipe Bertrand and Randall Bramley. DCA: A distributed CCA framework based on MPI. In *Proceedings of the 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'03)*, Santa Fe, NM, April 2004. IEEE Press. 53
[32] S. Lefantzi, J. Ray, and H. N. Najm. Using the common component architecture to design high performance scientific simulation codes. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS 2003), 22-26 April 2003, Nice, France.* IEEE Computer Society, 2003.
[33] J. Walter Larson, Boyana Norris, Everest T. Ong, David E. Bernholdt, John B. Drake, Wael R. Elwasif, Michael W. Ham, Craig E. Rasmussen, Gary Kumfert, Daniel S. Katz, Shujia Zhou, Cecelia DeLuca, and Nancy S. Collins. Components, the common component architecture, and the climate/weather/ocean community. In *84th American Meteorological Society Annual Meeting*, Seattle, Washington, 11–15 January 2004. American Meteorological Society.
[34] S. Zhou, A. da Silva, B. Womack, and G. Higgins. Prototyping the ESMF using DOE's CCA. In *NASA Earth Science Technology Conference 2003*, College Park, MD, June 24–26 2003. http://esto.nasa.gov/conferences/estc2003/papers/A4P3(Zhou).pdf.
[35] WS-Resource Framework. http://www.globus.org/wsrf.
[36] Advanced Visual Systems (AVS). http://www.avs.com/
[37] Business Process Execution Language for Web Services Version 1.1. http://www-106.ibm.com/developerworks/library/ws-bpel/
[38] Sriram Krishnan, et al. The XCAT Science Portal. *In Proceedings of IEEE/ACM SuperComputing 2001*, Denver, CO. Nov 2001
[39] . Dennis Gannon, Rachana Ananthakrishnan, Sriram Krishnan, Madhusudhan Govindaraju, Lavanya Ramakrishnan, and Aleksander Slominski. Grid Web Services and Application Factories. *In Grid Computing: Making the Global Infrastructure a Reality, Chapter 9.* Nov 2002.
[40] Web Services Security Version 1.0. http://www-106.ibm.com/developerworks/webservices/library/ws-secure/