On Abstractions of Software Component Models for Scientific Applications*

Julien Bigot[1], Hinde Lilia Bouziane[1], Christian Pérez[1], and Thierry Priol[2]

[1] INRIA/LIP, ENS Lyon, 46 allée d'Italie, F-69364 Lyon Cedex, France
[2] INRIA/IRISA, Campus de Beaulieu, F-35042 Rennes Cedex, France
{Julien.Bigot,Hinde.Bouziane,Christian.Perez,
Thierry.Priol}@inria.fr

Abstract. In the seek of more computing power, two sources of complexity are to face. On one hand, it is possible to aggregate a large amount of computing power (and storage) at the price of very complex resources such as grids. On the other hand, such available computing power allows to imagine more complex applications such as code coupling applications to achieve more realistic simulations. Component models appear as a solid foundation to handle simultaneously both sources of complexity. However, component models need to provide adequate abstractions to offer a simple programming model while enabling high performance on any kind of resources. This paper reviews several abstractions dedicated for scientific applications: data sharing between components, master-worker relationships, parallel to parallel component communications and collective communications among components.

1 Introduction

In order to better simulate the reality, applications are always looking for more computing power – as well as storage space. A huge computing power is available but at the price of a huge complexity to harness it such as in grids, with recently multi-core multi-CPU nodes. Moreover, to make use of such a computing power, applications turn out to get more and more complex. For example, code coupling applications aim to compose several (complex) codes coming from independent teams.

Hence, there is a clear need of a programming model that can handle simultaneously both sources of complexity so as to hide the infrastructure complexity (heterogeneity, volatility, etc.) while providing a simple model to efficiently build applications from several pieces of code.

Software component technology appears to provide an interesting foundation to reach such a goal. Software components aim at handling code reuse [1] and distributed software component models such as CCM [2] or SCA [3] have already dealt with code and resource heterogeneity. Generic and hierarchical component models such as FRACTAL [4] provide a foundation where more specific component models such as GCM can be defined [5]. Dedicated high performance models have also been proposed such as CCA [6].

* This work was supported by the CoreGRID European Network of Excellence and by the French National Agency for Research project LEGO (ANR-05-CIGC-11).

E. César et al. (Eds.): Euro-Par 2008 Workshops, LNCS 5415, pp. 438–449, 2009.

However, a common limitation of these models is that their programming model is very close to their execution model. It is very annoying for application portability as the programmer has to know the infrastructure architecture when designing its application. Moreover, it is not any longer satisfactory as more and more resources are volatile. Therefore, several researches have been conducted to increase the abstraction level of component models. For example, behavioral skeletons have been recently added to GCM [7]. Such an improvement of the abstraction level of component models a) improves the productivity by avoiding duplicating the effort for recurring composition patterns; b) reduces the complexity by embedding expertise in a "simple" concept; and c) hides the implementation details and thus enables resource independent composition. Hence, the challenge is to simplify application development while keeping high performance.

This paper presents four abstractions – namely data sharing, master-worker paradigm, parallel component and collective communications. These abstractions take place at the different levels of a component model: the port level (data sharing and collective communication), the component level (parallel component), and the assembly level (master-worker paradigm and collective communications). Our aim is to show that it is possible to simplify application development while keeping high performance.

The remaining of this paper discusses the four abstractions. Section 2 deals with data sharing between components. The support of the master-worker paradigm in component models is studied in Section 3. Section 4 is devoted to parallel components and Section 5 focuses on collective communications between components. Section 6 concludes the paper.

2 Data Sharing

2.1 Presentation

The *shared memory* paradigm is an attractive programming paradigm that allows data to be shared by multiple concurrent entities. Its advantage relies on the ease of programming: multiple entities (threads, processes, etc) can concurrently read/write data in a global space without any need to explicitly handle data localization, transfer and persistence. This concept has been successfully applied in several contexts: (1) multithreading within the same process, (2) data segment sharing among multiple processes running on the same host, (3) global data sharing across a cluster of workstations through Distributed Shared Memory (DSM) systems, and (4) grid data sharing services such as JUXMEM [8] in grid environments. Such a service transparently manages data localization and persistence in a dynamic, large-scale, distributed environment. A common objective of underlying data sharing systems is to hide the complexity of data sharing management. They also deal with non-functional concerns related to the nature of targeted execution resources and processes placement. Resources volatility and performance are examples of such specificities.

Sharing data among multiple computation entities is appropriate for some applications where data structures are complex and access patterns are irregular. Such applications may be component based applications. Hence, it is interesting to bring the benefits of the shared memory paradigm into component models.

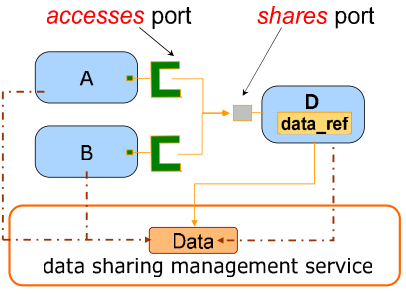

Fig. 1. Overview of data shared ports

```
interface AccessPort {
    float* get_pointer();
    long get_size();
    // Synchronization primitives
    void  acquire();
    void  acquire_read();
    void  release();
}
```

Fig. 2. Example of data port API accessing an array of floats

2.2 Limitations with Existing Component Models

In a classical way, interactions between components are done through well-defined ports. In fact, in existing component models, ports only enable explicit data transfer where the data is part of an exchanged message. Consequently, it is not easy to share data between components. When several components want to modify a same data, the functional code of a component should deal with data persistence, data consistency and fault tolerance issues. This therefore leads to an increased code complexity.

2.3 Data Shared Port Model

In [9], we proposed an additional family of ports named *data shared ports* (Figure 1). A data port logically attaches a shared data to a component. It can be of two kinds: a *shares* port to give an access to a shared data and an *accesses* port to enable a component to access a data exported through a *shares* port. These ports rely on a *transparent data access model*.

The proposed model provides a user view divided in two parts: an internal view that allows the implementation of a component to access a data and an external view that allows a component to share a data. In the internal user view, an interface named *AccessPort*, is implicitly associated to a data port. This interface is shown in Figure 2. It is available through *accesses* ports as well as *shares* ports. It allows a component with a *shares* port to also access the associated data. The *AccessPort* interface provides `get_pointer` and `get_size` operations to respectively retrieve a pointer to the shared data and its size. It also provides synchronization primitives, like `acquire` and `release`. The `acquire_read` primitive sets a lock in a read-only mode so that multiple readers can simultaneously access the shared data, whereas `acquire` sets a lock in an exclusive mode.

In the external view, a component which aims to access a data through an *accesses* port should have this port connected to a *shares* one. Such a connection implies passing the reference of the shared data from the shares component to the accesses component. It is assumed that the shared data is previously allocated and associated to the *shares* port. That is done through a dedicated interface provided only on the *shares* port side.

The global user view of data ports allows data to be shared between components without worrying about the mechanism used to share the data. Such a mechanism can

be a memory shared between components collocated within a same process, a shared memory segment for components in two different processes but deployed on the same host, a DSM for a cluster or a grid data-sharing service like JUXMEM for a grid. The choice of a particular mechanism is expected to be done by the execution/deployment framework once execution resources and the placement of components are known. This choice determines which implementation of data port interfaces is to be used. The ability to use several implementations does not require to modify the user code of components.

2.4 Implementation

The proposal was projected on CCM [9] and CCA [10]. The projection is based on extending the specification of these models with the possibility to define and use data ports. For the particular case of CCM, data ports are defined in an extended IDL3. We realized a prototype implementation based on classical CCM concepts, where the extended IDL3 is translatable to classical IDL3 definitions. To manage the shared data, different mechanisms were tested, like NFS and JUXMEM. This prototype is a proof of the concepts presented in Section 2.3 and of the facilities offered to the user. More details about the realized prototype as well as an application example using data ports can be found in [9].

3 Master-Worker Paradigm

3.1 Presentation

In the MASTER-WORKER programming paradigm, several instances of a same code (workers) have to be executed simultaneously with different parameter values sent by a master code. This paradigm is widely used in distributed and embarrassingly parallel applications like parametric applications. The relevance of this paradigm promoted numerous research activities to propose dedicated software environments. Examples are SETI@Home [11], XtremWeb [12] and BOINC [13] for Global Computing systems or DIET [14], NetSolve [15], Ninf-G [16] and Nimrod/G [17] for Network Enabled Server environments. A common objective of these environments is to lower the time taken by programmers to design, implement and deploy MASTER-WORKER applications. They offer transparent management of load balancing and dependability issues to enable efficient execution on a given computing infrastructure.

However, most of the proposed environments only focus on the MASTER-WORKER paradigm. Even if some of them provide another paradigm, like task farming in Ninf-G, they remain very specialized environments. Therefore, they are not sufficient to deal with multi-paradigm applications like code coupling applications.

3.2 Limitations with Existing Component Models

In opposition to MASTER-WORKER environments, existing component models require the designer to manage worker components (their number and load balancing.) Consequently, the code complexity is increased. This complexity is further increased since a designer has also to implement an adequate request transport policy. Such a policy

can be very complex and can depend on the underlying execution infrastructure. For instance, the policy is probably not the same when using PC clusters with thousands of processors or grids with heterogeneous processors and failures. Thus, code re-use is also limited.

As a result, existing component models do not offer a high level of abstraction to design those parts of an application that follow the MASTER-WORKER paradigm.

3.3 Collection Overview

In [18], we proposed to improve the support of the MASTER-WORKER paradigm in component models. For that, we proposed a high level MASTER-WORKER design model for which an overview is given in Figure 3. The proposal is based on the concept of *collection*. A collection is defined as a set of *exposed* ports, bound to some internal component type ports. A collection behaves like a component: it can be connected to other components and/or collections. However, such a composition is done in an *abstract* architecture description, which represents the user's view of the application. In this view, the number of component instances inside the collection as well as the mechanism to be used to distribute incoming requests are unknown. Ideally at deployment time – when resources are known – a collection is turned into a concrete assembly. Hence, at runtime, the collection is made of some internal component instances and of an instance of a *request transport pattern*. A *pattern* represents an implementation of an algorithm that specifies how to transfer requests from the *master* to *worker* components and how to schedule them. Its implementation is expected to be realized by experts and it may be based on software components. Figure 3 shows an example of a component based pattern (Round-Robin pattern) and a non component based one which reuses the already existing environment DIET. It also shows a collection instantiation with n workers and a Round-Robin pattern instance.

The interest of patterns is to enable the separation of request transport concerns. Several request transport algorithms, like Round-Robin, load balancing, request sequencing or others can be used within an application. Moreover, a pattern can be replaced by another without any other change in the application. The need of pattern replacement

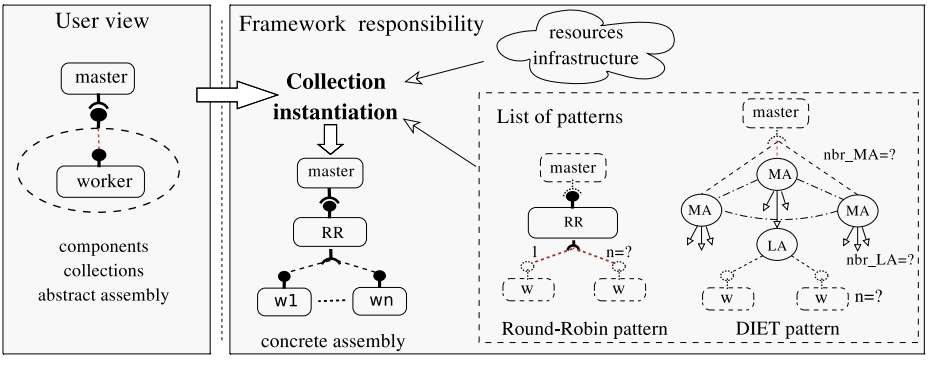

Fig. 3. Overview of the generic MASTER-WORKER model

appears when the number of requests or their loads lead to slow down the execution. In such a situation, the request transport algorithm and/or the number of workers may be a bottleneck. In [19] we studied how an adaptability framework can be integrated within the concept of collection to support dynamic modifications of collection's content. However, it is out of the scope of this paper.

3.4 Implementation

To illustrate the feasibility of the proposed generic model, we extended three specific component models: CCM [18], GCM/FRACTAL [18] and CCA [10]. Theses extensions provide specifications for collection and pattern concepts (description and usage). This section gives an overview of the CCM extension.

To describe a collection, the IDL3 of CCM was extended and a *Collection Description Language* (CDL) was defined. The extended IDL3 introduces a keyword `collection` to define a collection ports in a similar way as for components. The CDL allows the description of a collection content. To use a collection in an assembly, the CCM assembly language was extended with 14 new *XML* elements. This extension preserves the CCM composition principle.

To describe a pattern, many solutions are possible. We used for instance the *XSLT* language to both describe a pattern architecture and realize its introduction in a collection. As cited in Section 3.3, a pattern may be based on non component technology. In this case, its architecture introduces the concept of *adapter* components. An adapter component is a proxy component responsible to translate a master (resp. a MASTER-WORKER environment specific) request to the used MASTER-WORKER environment specific (resp. worker) request. Thus, the architecture of a pattern as well as a concrete assembly may be heterogeneous. To support such an assembly, we proposed an extension of the CCM assembly language. Such an assembly was realized for reusing DIET.

A set of experiments using the CCM extension was done to illustrate the benefits of the proposal. Results can be found in [20].

4 SPMD Parallel Components

4.1 Presentation

The *Single Program Multiple Data* (SPMD) paradigm is a paradigm where multiple instances of a single program are run in parallel, each one in its own process. The number of instances to use is a parameter of the application and these instances can communicate thanks to a well founded communication model based on message passing and collective communications.

This paradigm is well suited for scientific simulations where a meshing of the space to simulate is done. Each process works on a different set of meshes and the interaction at the boundaries are handled by message exchange between processes.

In order to couple SPMD codes it must be possible for each code to call methods implemented by others. The efficient implementation of such $M \times N$ method calls requires that no single process is in charge of the communications between the codes but rather that each process participate in the communication.

4.2 Limitations with Existing Component Models

Component models that do not natively support parallel components have no technical limitation that prevent component implementation to dynamically create new processes to implement parallel component. However, there is also no support for that. This means that the code handling the placement of new processes has to be replicated in every component. This also makes it difficult to obtain an optimal planning as each component instance does not have the informations about the placement of the processes of other component instances. Finally, this leads to the centralization of all interaction with a given component instance on the single process known by the component model which creates a bottleneck.

4.3 Parallel Component Overview

In [21], we proposed a model to allow the implementation and efficient coupling of parallel components. The implementation of a parallel component is done as a classical component implementation. The instantiation of such a component does however lead to the multiple instantiations of this implementation. The number of instances to create is a parameter set in the assembly.

Efficient $M \times N$ method calls are allowed by letting each process of the caller (resp. callee) provide (resp. receive) a part of each parameter and then receive (resp. provide) a part of the result. The ports used to couple the component are described as classical use/provide ports in what is called the user component interface description. The distribution of the data inside each component is a detail of the implementation and is therefore described in a side XML description that is part of the component implementation. The fact that ports of parallel component are described as classical use/provide ports also allows to handle the case where a non parallel component is connected to the port of a parallel component.

The implementation of the component is based on a component interface description that is slightly different from the user component interface: the internal component interface. In this interface, the parameters that have been described as distributed in the distribution description are replaced by their distributed equivalent data type (a matrix can for example be replaced by a vector for a (block(1),$*$) distribution).

4.4 Implementation

This model has been implemented as a CCM extension. The internal component interface is automatically generated together with a GridCCM glue layer and a manager by using the user component interface description and the XML data distribution description. The GridCCM glue code is then inserted in the user implemented component when it is compiled.

When the parallel component is instantiated, it is in fact the manager that is instantiated. This manager is then responsible for the multiple instantiations of the user implementation according to the parameters set in the assembly. Similarly, when a client gets a handle on a parallel component, it is in fact a handle on the manager. When two parallel components are connected, their manager transparently exchange information

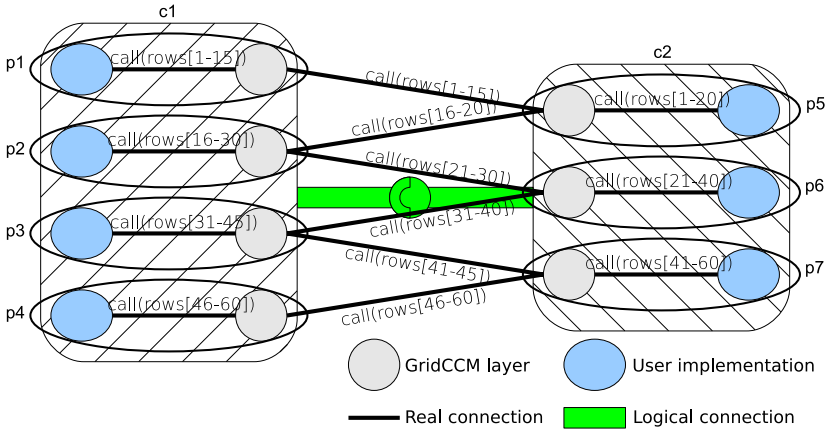

Fig. 4. Behavior of a method call in GridCCM

in order to connect the GridCCM layers of both components according to the redistribution that must be done.

When the user implementation calls a method on one of its use ports, the behavior is the one described in Figure 4. The call is intercepted by the GridCCM layer of the caller component that sends the data to the GridCCM layers of the callee component according to the data redistribution schema. Then this callee GridCCM layer calls the method on the callee user implementation. The same process is repeated for the return of the result of the method call.

5 Collective Communications

5.1 Presentation

As it has been said in the previous section, the most common interaction paradigm between processes of a SPMD parallel code is the use of message passing. Two kinds of operations exist: point to point operations that involve two processes (a sender and a receiver) and collective communication operations that involve a group of processes. As there is no assumption made on the number of processes with this latter kind of operations, it is well suited to SPMD codes where the number of processes is not known when the code is written.

The simplest collective operation is the *barrier* synchronization operation where each process is blocked in the call until all processes have called it. Other collective operations involve an exchange of data. For example in the *allgather* operation, each process receives a copy of the data of all processes in the group as shown in Figure 5 and in the *broadcast* operation, a specific process designed as the root sends its data to all the other processes in the group as shown in Figure 6.

Efficient implementation of these operations is greatly dependant on the resources on which the processes are executed and is still an active research domain. Effective implementation on distributed resources such as clusters requires communications to be done

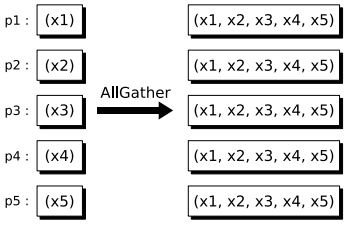

Fig. 5. The *allgather* collective operation

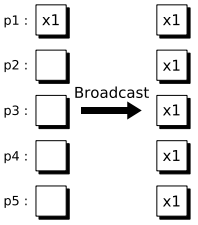

Fig. 6. The *broadcast* collective operation

in parallel as much as possible to take part of all communication links. There should therefore be no central process involved in all communications. In the case of grids, the different properties (namely bandwidth and latency) between intercluster links and the backbone link interconnecting clusters must also be taken into account.

5.2 Limitations with Existing Component Models

The next step after the introduction of SPMD parallel components as described in the previous section is to let the replicated entities in parallel components be implemented with components. In this case, communications between these instances has to be described by ports. While it is quite easy to reuse existing communication paradigms such as events or remote method call to provide a semantic similar to point to point message passing, there is no straightforward solution for the use of collective communication operations.

5.3 Collective Communications Model

In [22], we proposed a model to use collective communication operations between components. A component that use collective communication has to describe a use port with a dedicated `CollComm` interface. This interface is very similar to the MPI interface in order to ease the transition from MPI to component collective communication except for the groups that are not described by a parameter for each call. Instead, groups are described in the assembly, they are created by the instantiation of a `CollCommProvider` component that provides the `CollComm` interface. The component instances whose use ports are connected to the provide port of this instance are part of the group. A component instance can be part of more than one group if it declares several use ports of the `CollComm` interface as shown in Figure 7.

5.4 Implementation

Collective communications have been implemented as a CCM extension. As a classical centralized implementation of the `CollCommProvider` component would lead to bottlenecks, the component implementation model has been extended to allow efficient implementations. Some concepts have been added: *AnyToAny* connections and *replicating* implementations.

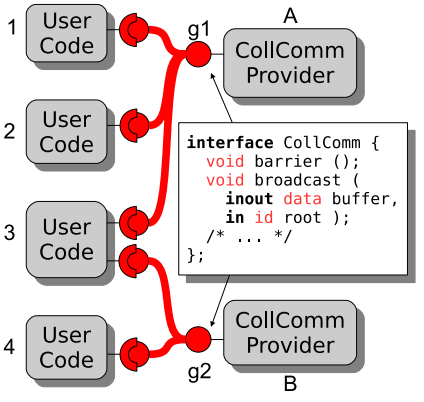

Fig. 7. Collective communication usage in the assembly

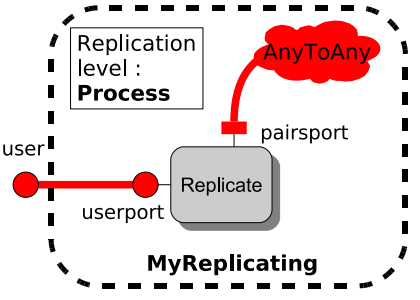

Fig. 8. A replicating component

A *AnyToAny* connection provides a semantic similar to use/provide connections but with any number of component participating. Each component has to provide the interface of the connection, in return it can call an operation of the interface on any component participating in the connection.

A *replicating* component implementation is defined by a *replicate* component, a *replication level* (process, node, cluster, grid ...), internal *AnyToAny connections* and a *mapping* of the ports of the component to ports of the replicate as shown in Figure 8. At runtime, the component is replaced by a set of instances of the replicate. The number of instances is determined by the replication level and the usage of the component. For example if the replication level is set to process and that components connected to a provides port of the component are spread amongst three different processes, then three instances of the replicate will be created, one on each process where the component is used. Internal *AnyToAny* connections are used to connect all the replicate instances and the connections to the ports of the component are replaced by connection to the mapped ports.

6 Conclusion

Software component appears to be a very promising concept to be a *programming* entity. It allows to improve productivity, to reduce complexity and to hide implementation details. This paper showed how four important abstractions for scientific applications can fit well into component models without impacting performance. These four abstractions impacted the various levels of a component model: data sharing can be provided through a new kind of port while parallel components require a new kind of component; the master-worker paradigm and the collective communications mainly involve the assembly level.

However, this paper does not claim to be exhaustive. In particular, this paper does not deal with abstractions that required to modify the kind of the assembly level such as workflow [23] or skeletons [24].

Future researches can be divided into two branches. First, most of the abstractions requires parameter selection to perfectly fit to the actual resources. Hence, strategies and algorithms are needed to achieve automatically selection. This is particular very important for volatile resources so as to have self-$*$ implementations of the abstractions. Second, the implementation of all these abstractions into an efficient and coherent runtime is challenging. Moreover, new abstractions may appear. Thus, we need a flexible mechanism. A promising technique is to use model transformation to transform programming abstractions to execution entities.

References

1. Szyperski, C., Gruntz, D., Murer, S.: Component Software - Beyond Object-Oriented Programming, 2nd edn. Addison-Wesley/ACM Press (2002)
2. OMG: CORBA component model, v4.0. Document formal/2006-04-01 (April 2006)
3. Beisiegel, M., Blohm, H., Booz, D., Edwards, M., Hurley, O., Ielceanu, S., Miller, A., Karmarkar, A., Malhotra, A., Marino, J., Nally, M., Newcomer, E., Patil, S., Pavlik, G., Raepple, M., Rowley, M., Tam, K., Vorthmann, S., Walker, P., Waterman, L.: SCA Service Component Architecture - Assembly Model Specification, version 1.0. Technical report, Open Service Oriented Architecture collaboration (OSOA) (March 2007)
4. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.B.: The fractal component model and its support in java. Software Practice and Experience, special issue on Experiences with Auto-adaptive and Reconfigurable Systems 36(11-12) (2006)
5. Institute, P.M.: Basic features of the grid component model. CoreGRID Delivrable D.PM.04, CoreGRID (March 2007)
6. Bernholdt, D.E., Allan, B.A., Armstrong, R., Bertrand, F., Chiu, K., Dahlgren, T.L., Damevski, K., Elwasif, W.R., Epperly, T.G.W., Govindaraju, M., Katz, D.S., Kohl, J.A., Krishnan, M., Kumfert, G., Larson, J.W., Lefantzi, S., Lewis, M.J., Malony, A.D., McInnes, L.C., Nieplocha, J., Norris, B., Parker, S.G., Ray, J., Shende, S., Windus, T.L., Zhou, S.: A component architecture for high-performance scientific computing. International Journal of High Performance Computing Applications 20(2), 163–202 (2006)
7. Aldinucci, M., Campa, S., Danelutto, M., Vanneschi, M., Dazzi, P., Laforenza, D., Tonellotto, N., Kilpatrick, P.: Behavioural Skeletons in GCM: Autonomic Management of Grid Components. In: Baz, D.E., Bourgeois, J., Spies, F. (eds.) Proc. of Intl. Euromicro PDP 2008: Parallel Distributed and Network-based Processing, Toulouse, France, pp. 54–63. IEEE, Los Alamitos (2008)
8. Antoniu, G., Bougé, L., Jan, M.: JuxMem: An adaptive supportive platform for data sharing on the grid. Scalable Computing: Practice and Experience 6(3), 45–55 (2005)
9. Antoniu, G., Bouziane, H., Breuil, L., Jan, M., Pérez, C.: Enabling transparent data sharing in component models. In: 6th IEEE International Symposium on Cluster Computing and the Grid (CCGRID), Singapore, May 2006, pp. 430–433 (2006)
10. Antoniu, G., Bouziane, H.L., Jan, M., Pérez, C., Priol, T.: Combining data sharing with the master-worker paradigm in the common component architecture. In: The 15th IEEE International Symposium on High Performance Distributed Computing (HPDC), Paris, France (June 2006)
11. Anderson, D., Bowyer, S., Cobb, J., Gebye, D., Sullivan, W., Werthimer, D.: A new major SETI project based on Project SERENDIP data and 100,000 personal computers. In: Conference Paper, Astronomical and Biochemical Origins and the Search for Life in the Universe, IAU Colloquium 161, p. 729. Bologna, Italy (1997)

12. Germain, C., Néri, V., Fedak, G., Cappello, F.: XtremWeb: building an experimental platform for Global Computing. In: Buyya, R., Baker, M. (eds.) GRID 2000. LNCS, vol. 1971, pp. 91–101. Springer, Heidelberg (2000)
13. Anderson, D.P.: Berkeley Open Infrastructure for Network Computing (2002), http://boinc.berkeley.edu/
14. Caron, E., Desprez, F., Lombard, F., Nicod, J., Quinson, M., Suter, F.: A Scalable Approach to Network Enabled Servers. In: Monien, B., Feldmann, R.L. (eds.) Euro-Par 2002. LNCS, vol. 2400, pp. 907–910. Springer, Heidelberg (2002)
15. Casanova, H., Dongarra, J.: NetSolve: A Network-Enabled Server for Solving Computational Science Problems. The International Journal of Supercomputer Applications and High Performance Computing 11(3), 212–223 (1997)
16. Tanaka, Y., Nakada, H., Sekiguchi, S., Suzumura, T., Matsuoka, S.: Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing. J. Grid Compututing 1(1), 41–51 (2003)
17. Buyya, R., Abramson, D., Giddy, J.: Nimrod/G: An Architecture for a Resource Management and Scheduling System in a Global Computational Grid. High-Performance Computing 01(1), 283 (2000)
18. Bouziane, H.L., Pérez, C., Priol, T.: Modeling and executing master-worker applications in component models. In: 11th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS), Rhodes Island, Greece (April 2006)
19. André, F., Bouziane, H.L., Buisson, J., Pazat, J.L., Pérez, C.: Towards dynamic adaptability support for the master-worker paradigm in component based applications. TR RT-0333, INRIA (April 2007)
20. Bouziane, H.: De lábstraction des modèles de composants logiciels pour la programmation dápplications scientifiques distribuées. Ph.D thesis, Université de Rennes 1, IRISA/INRIA, Rennes, France (February 2008)
21. Pérez, C., Priol, T., Ribes, A.: A parallel corba component model for numerical code coupling. In: Parashar, M. (ed.) Proc. 3rd International Workshop on Grid Computing. LNCS, vol. 17, pp. 88–99. Springer, Heidelberg (2000); Special issue Best Applications Papers from the 3rd Intl. Workshop on Grid Computing
22. Bigot, J., Pérez, C.: Enabling collective communications between components. In: CompFrame 2007: Proceedings of the 2007 symposium on Component and framework technology in high-performance and scientific computing, pp. 121–130. ACM Press, New York (2007)
23. Bouziane, H., Pérez, C., Priol, T.: A software component model with spatial and temporal compositions for grid infrastructures. In: Luque, E., Margalef, T., Benítez, D. (eds.) Euro-Par 2008. LNCS, vol. 5168, pp. 698–708. Springer, Heidelberg (2008)
24. Aldinucci, M., Bouziane, H., Danelutto, M., Pérez, C.: Towards software component assembly language enhanced with workflows and skeletons. In: Joint Workshop on Component-Based High Performance Computing and Component-Based Software Engineering and Software Architecture (CBHPC/COMPARCH 2008), October 14-17 (2008)