Dynamic Adaptation of Parallel Codes: Toward Self-Adaptable Components for the Grid

Françoise André, Jérémy Buisson, Jean-Louis Pazat

DYNAMIC ADAPTATION OF PARALLEL CODES: TOWARD SELF-ADAPTABLE COMPONENTS FOR THE GRID

Françoise André
IRISA/Université de Rennes 1
Rennes, France
Francoise.Andre@irisa.fr

Jérémy Buisson and Jean-Louis Pazat
IRISA/INSA
Rennes, France
Jeremy.Buisson@irisa.fr
Jean-Louis.Pazat@irisa.fr

Abstract One of the challenges that come from the emergence of Grid architectures is to invent new programming techniques for these new platforms. As we explain in this article, we think that the architecture of the applications should reflect both the parallel and the distributed aspects of Grid architectures. It results in applications built as assemblies of parallel components. Since Grid architectures are known to be highly dynamic, using resources efficiently on such architectures is a challenging problem. Software must be able to react dynamically to the changes of the underlying execution environment. In order to help developers to create software for the Grid, we are investigating a model for the adaptation of parallel components. This paper focuses on the adaptation mechanisms that are provided as a meta-level for components. We describe how a generic platform can help to develop efficient Grid software. First experimental results show the gain that can be expected from the use of such a platform.

Keywords: Grid computing, dynamic self-adaptation, parallelism, software component, reflexive programming.

1. Introduction

Research in Grid computing mainly focuses on the development of middleware and services allowing applications to use various distributed resources. Infrastructures and toolkits such as OGSA [3] and previously Globus [5] provide tools allowing naming and discovery of resources; they also provide the necessary services for applications to deal with the underlying heterogeneity of the Grid. Those projects provide the users with useful tools for deploying and running applications without explicitly dealing with the various batch queues, communication libraries and so on, installed on the local sites.

Although resource allocation and scheduling are taken into account, these tools give no help for applications to make efficient use of the available Grid resources at run time. Due to the dynamic nature of the Grid, it is also very hard to design an application that fits well with any configuration of the Grid. Moreover, constraints such as the number of available processors, their respective load level, available memory and network bandwidth are not static. The bandwidth between sites running some parts of the same application may vary during the execution time or some processors may be requested by other applications. For example, the CPU manager described in [8] may dynamically change the number of processors allocated to each application. We think that in many cases, application performance can be greatly improved when any part of an application can take into account varying resources eg. is able to adapt its behavior to “environmental changes”.

Because Grid applications are also quite complex, many approaches now rely on service-oriented technologies such as OGSA [3] or on component-based approaches [11] such as CCM [9]. This helps building reusable software. Within these programming techniques, one of the main issue is the so called “separation of concerns” paradigm. Entities implementing distinct functionalities should be located in different modules, objects, services or components. In the remaining of this paper, the term “component” stands for any of these kinds of pieces of application.

Our work focuses on the problem of adapting parallel codes encapsulated in components to varying constraints on resources. In this paper, we show how to combine parallel programming and adaptation techniques in a unified framework. As a first step, this paper focuses on the adaptation of the inner parallel code of a component. In section 1.2, we describe parallel objects and parallel components. Section 1.3 is devoted to the presentation of adaptation techniques in the context of scarce resources, putting the emphasis on the description of the ACEEL framework. Section 1.4 presents the application model we consider. Section 1.5 explains how we transpose adaptation models to the parallel case to build a unified framework. The results of our experiments are given in section 1.6. As a conclusion, we describe the main steps of our ongoing work.

2. Parallel distributed objects for the Grid

During the last decade, improvements have been made in terms of software reusability when object then component technologies appeared in many application fields other than high performance computing. Because Grid architectures are complex, heterogeneous and dynamic, they make the development of parallel applications more complex. Now, it becomes necessary for high performance applications to adopt technologies enforcing code reusability. Well-known component models such as CCM or Enterprise Java Beans should be a good basis but they were not designed with performance and parallelism in mind, so they have not been able to take into account High Performance Grid applications.

In order to improve these environments, several projects have studied how parallel code could be encapsulated within objects or components. Projects such as PARDIS [7] and PaCO++ [2] have focused on increasing performances of parallel distributed objects. They consider a parallel object as a set of identical sequential objects. The same definition also applies to GridCCM [10] within the component world.

Those projects allow to encapsulate SPMD code into so called "high performance CORBA objects/components". When a parallel object/component has to process a remote call, each process executes one part of the processing related to one part of the data set. The parallelism comes from the distribution of the parameters.

In order to get some performance out of high-speed networks, an enhanced request protocol has been defined among parallel objects/components: servers allow their clients to "see" their internal structure and distribution at run time. This allows parallel clients to send data directly from the source process to the target process: data do not need to be received/sent by a single master object. This multi-port communication mode allows to use the aggregated bandwidth, which can be higher than if only one centralized communication port was used.

These approaches have shown that it is now possible to use component-based techniques for programming high performance applications on the Grid without loosing performance. The next step is to be able to have components that are more flexible to allow the adaptation (not only the configuration) of parallel codes.

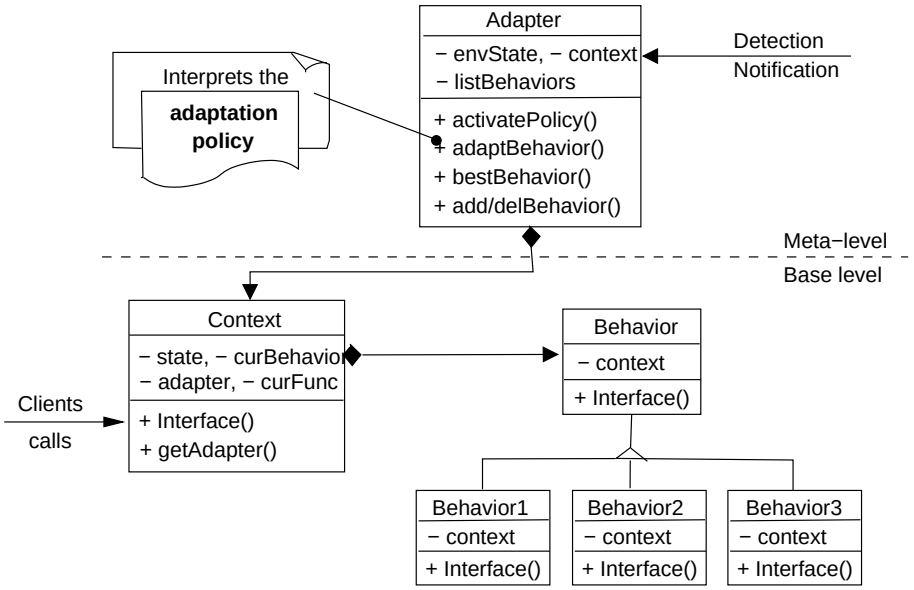
3. Dynamic adaptation of components

In the area of wireless computing and mobile environments where resources are a key issue, many techniques of dynamic adaptation have been developed: from the observation of the environment, codes can adapt their behavior to fit the resource constraints. This adaptation can be achieved in many different ways ranging from a simple modification of some parameters to the complete

exchange of the running code with a new one that is more suited to the environment.

The adaptation could be achieved by designing ad hoc applications that take into account the specificities of the targeted environment. For example, this was done for the Web applications access protocol on mobile networks by defining the WAP protocol [12]. A more general way to allow an application to evolve according to its environment is to provide mechanisms that permit dynamic self-adaptation by changing the behavior depending on the currently available resources. In many cases, this has been achieved by embedding the adaptation mechanism within the application code. For example, the AdOC compression algorithm [6] includes such a mechanism to change dynamically the compression level according to the available resources. However, it is desirable to separate the adaptation engine from the application code in order to make the code easier to maintain and to easily change or improve the adaptation policy. In this case, a framework that provides generic mechanisms for the adaptation process and for the definition of the adaptation rules is needed. This is the case for example of the ACEEL framework.

Figure 1. Architecture of an ACEEL component



ACEEL [1] is a generic framework for self-adaptable components that allows the developer to focus on the implementations of the functionalities of his component and on its adaptation *policy*: it separates the adaptability aspect from the functional part of the component, as shown by the architecture of an

ACEEL component on figure 1. Based on the *Strategy* design pattern [4], the component offers a set of possible implementations, called *behaviors*. At any time, only one behavior is active: the one that processes the incoming requests. The generic *adapter* meta-object decides which of the available behaviors the best to use according to the environment is. To help the adapter object to decide, the component developer provides a policy as a set of event-based rules: each rule is a triggering change of the environment associated with a reaction, which might be either the activation of another behavior or the tuning of some parameters. When a change in the characteristics or availability of a resource happens, the monitoring engine notifies the adapter of the components that depend on this resource for their adaptation policy. The *context* holds the state of the component. Separating the state from the implementation makes it easier to replace dynamically the implementation of the component.

4. A programming model for Grid applications

The applications we target in our project are "Grid applications"; they may be composed of several parallel codes, so in our model, an application is considered to be built as an assembly of components. Each component is deployed on a site that is a parallel machine such as a cluster. As a first step, we focus on the deployment and execution of one component and we do not investigate the relations between components. In our model, each component is both parallel and adaptable.

A component is parallel: this means that it is composed of a number of internal processes working together to execute a given service. These processes communicate between each other using a communication library such as PVM or MPI or through a distributed shared memory. Here, we do not require specifying how those processes are encapsulated within the component: this aspect relies on constraints of existing components platforms such as GridCCM.

A component is adaptable: the platform where components are deployed can monitor the resources of the deployment site and allow any component to react to any change in the state of the resources.

We think that it makes sense to allow one single component to adapt itself dynamically in most Grid environments for two main reasons. First, one characteristic of Grid architectures is that sites are administrated independently one from another and of the users of the Grid. It is thus possible that the site into which the component is deployed is modified while the component is running. Secondly, in a longer term, we can consider component migration as a special case of adaptation. Because the sites implied in the migration may have different characteristics, the adaptation of the component will be needed.

5. Adaptation of parallel components

A parallel self-adaptable component is a component that is composed of several processes working together and that is able to change its behavior according to the changes of the environment.

5.1 Structure of the components

The structure of parallel self-adaptable components includes an adaptation *policy*, a set of available implementations, called *behaviors*, and a set of *reaction steps*.

The purpose of the adaptation policy is to define when the adaptation mechanism should be triggered and what should be the associated reactions. It is mostly a set of event-based rules. Each rule associates a reaction to a specific event. Events represent any change in the state of the environment. For example, an adaptation policy can include the rule: "if the number of nodes is increased, spawn new processes and redistribute arrays". This rule shows both the trigger event and the associated reaction.

Behaviors are implementations of the component. Each behavior differs from the others in the way it uses resources and/or in the algorithm used. Each behavior of a component implements the whole interface of the component; they just use different ways. The active behavior is the one used to process the incoming requests. The expression "functional code" denotes any code that is productive and that resides in the behaviors.

Reaction steps are the means by which the component adapts itself. It can be for example the replacement of the active behavior, the tuning of some parameters, or the redistribution of arrays. These pieces of code are dynamically inserted in the execution flow when the component adapts itself. Reactions must ensure that they leave the component in such a consistent state that the execution can resume and lead to the same result than if no reaction has been executed.

Because reactions must enforce the consistency of the component, reaction steps cannot be inserted at any time in the execution flow. In order to specify the places at which the component is able to adapt itself safely, we define the notion of *adaptation point*.

5.2 Semantic of adaptation points

An adaptation point is an annotation in the code that indicates where the component can be safely modified. The developer indicates that the behavior is able to suspend in a consistent state and to resume from this state at an adaptation point, no matter which behaviors and reaction steps combination leads the component to that state.

The platform enforces the mutual exclusion between the functional code and reactions: it ensures that reactions might only be executed when the functional code is suspended at an adaptation point. Adaptation points are thus the moments at which reaction steps can be inserted in the execution flow.

Reaction steps must ensure that if the state of the component is consistent, it remains consistent after the execution of the whole reaction. The scope of this consistency includes both the variables of the component and the active behavior. This means that the functional code should not be able to determine whether a particular reaction step has been executed at a particular adaptation point. Adaptation points are almost invisible to the functional code.

An adaptation point is said *active* when a reaction is scheduled at that adaptation point. Otherwise, the adaptation point is said *inactive*.

5.3 Introducing global adaptation points

Because several processes may collaborate during a single reaction, they need to be synchronized and coordinated. As for global consistent states in distributed systems, some of the combinations of adaptation points do not represent valid states at which the component is able to adapt itself.

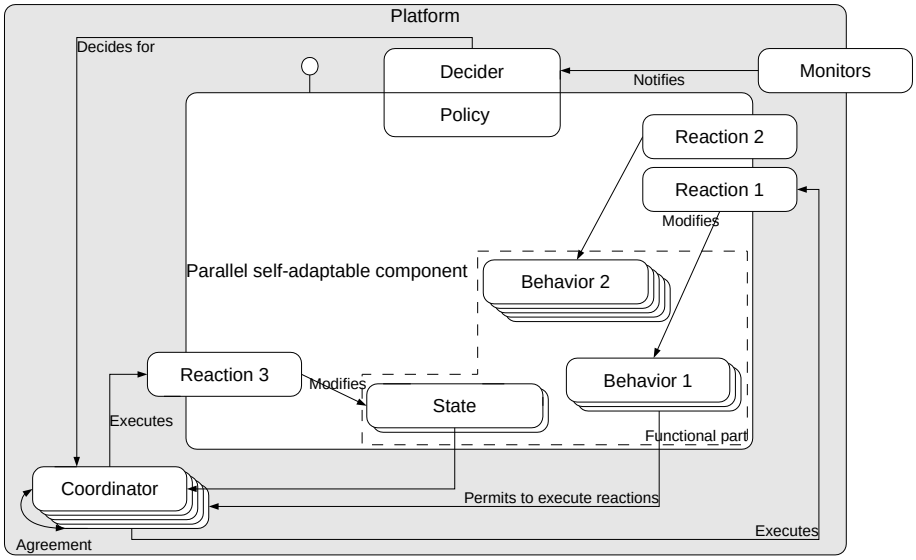
Adaptation points are local to each process; so are the annotated states. For that reason, adaptation points are not sufficient to specify states at which the whole behavior can be modified. This is why the developer has to give explicitly a compatibility relationship between the adaptation points of each process of the behavior in order to allow the platform to find consistent states. The platform enforces the fact that reactions can only be executed when all the processes are suspended on adaptation points that are compatible with each other. Those *global adaptation points* specify global states at which the component is able to adapt itself, global states at which the developer permits the adaptation mechanism to be executed. Our model only specifies the semantic of global adaptation points: the developer should place them to indicate the global states at which he thinks the adaptation can occur safely.

5.4 Building a host platform

We consider that modern programming techniques should conform to the "separation of concerns" paradigm. For this reason, we think that adaptability should be a service given by the platform that hosts the component is deployed. Figure 2 shows the overall architecture of a platform hosting a parallel self-adaptable component.

The platform mainly provides two kinds of objects: the *decider* and the *coordinators*. The decider is the object that makes the decisions: it decides when (events to watch) and how (reactions to execute) the component should adapt itself according to the adaptation policy. It bases its decisions on the reports

Figure 2. Overall architecture of a parallel self-adaptable component



given by the *monitors* that are interfaced with the platform. The monitors are daemons that track and report changes in the state of the environment. The *coordinators* execute the directives given by the decider: they serve as intermediaries between the code of the component and the platform. Their role is to synchronize the adaptation mechanism with the functional code and to coordinate the execution of the reactions.

When a monitor detects a change in the state of the resources it watches, it notifies the decider. The decider then interprets the given adaptation policy. During this step, the decider might query the monitors for a detailed report on the state of the resources. If it concludes that there is no need to adapt, it stops the adaptation process until the next notification from a monitor; otherwise, it broadcasts its decision to the coordinators, which decide collectively when the reaction is effectively executed. This is done by selecting a global adaptation point - and activating the corresponding adaptation point of each process. Once a process reaches an active adaptation point, its coordinator executes the reaction chosen by the decider.

6. Experimental results

In order to validate the feasibility of our approach, we have built a prototype platform that partially implements the model we have proposed.

6.1 Test platform

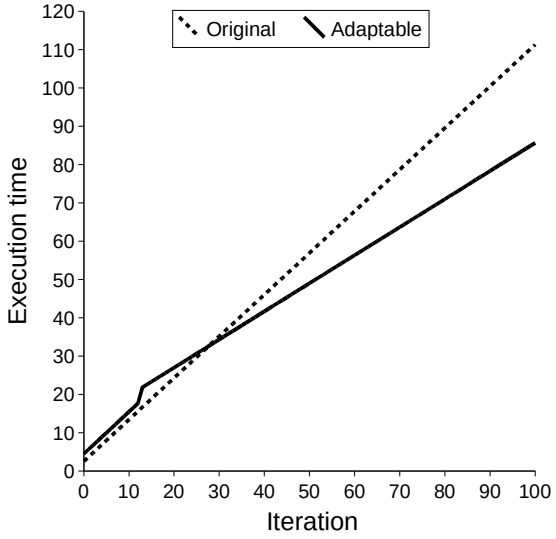
The test platform we developed implements a subset of the model. It concentrates on SPMD applications. The adaptation points we implemented allow to execute user-defined reaction.

Although the application we use in our tests seems simple, it is sufficient to show that our adaptation model works well with parallel applications without significant performance degradation. Our application is a generic vector iteration that distributes its vectors using a block scheme. It uses MPI for its communications. Its adaptation policy is to use as many nodes as the monitor reports: it spawns new processes when nodes are added to the system and terminates the processes that use nodes reclaimed by the system. Because the MPI implementation we use cannot dynamically spawn or terminate processes, the application starts with a fixed number of processes, but uses only the reported number. For this reason, we simulate the monitoring of the number of available processors; this also allows us to have full-control over the adaptation mechanism. We place one adaptation point between iterations.

6.2 Gain of the adaptation

In this experience, once the application has been started, the number of available nodes is increased from 4 to 6. Figure 3 shows the elapsed time at the end of the iterations. The execution of the reaction occurs between iterations 12

Figure 3. Execution time of an adaptable application



and 13; this appears as a break on the curve. This figure shows that several iterations are needed in order to balance the cost of the execution of the reaction. In the long term, the gain of the adaptable version over the original application is substantial. The amount of iterations needed before the adaptable version becomes effectively better depends on the reaction that has been executed and on the component itself.

6.3 Overhead of the adaptation platform

We compared the execution time of the original application and of its adaptable version in a totally static environment. The difference linearly depends on the number of encountered adaptation points. It shows the time needed to initialize the platform and the execution time lost in adaptation points. The execution time overheads are shown in table 1, depending on the number of encountered adaptation points. This measure is very noisy. The execution time lost in each adaptation point seems constant at about 0.2 milliseconds.

Table 1. Execution time for several iterations count

Adaptation points	*Overhead (ms)*
50	264.9
100	295.6
150	304.3
200	376.4
250	278.5

6.4 Ease of use

Our prototype assumes that the application is iterative and that adaptation points only reside between iterations. The reason is that it makes it easier to build the state of spawned processes. However, our model does not impose such a restriction.

One of the hardest questions to answer to when using the platform is the placement of adaptation points and their frequency. For our experiments, we arbitrarily place one adaptation point per iteration, but there was no a priori reason for doing so. Having many adaptation points makes the application more reactive to environmental changes at the cost of an increased overhead.

7. Conclusion

In this paper, we have shown that the idea of combining a dynamic adaptation framework with parallelism and distribution is a promising way for efficiently programming the Grid. We have built a prototype that extends the ACEEL adaptation engine to take into account the parallelism that can reside in components. This allowed us to experiment our ideas.

Our model complies with the separation of concerns paradigm, since it completely separates the adaptation mechanism from the functional code of the component. Moreover, it provides a basis for the dynamic adaptation of parallel code, whereas many projects have focused on their configuration at startup. This separation allowed us to experiment our idea without integrating a full-featured component platform. However, we expect that building application using a component infrastructure help doing the adaptation. Indeed, containers offered by the component infrastructures are a privileged place into which non-functional services (such as security or adaptation) should reside. Those containers also help doing the adaptation with their introspection ability.

In our ongoing works, we plan to define more formally the properties that the component is required to satisfy in order to be able to adapt itself. This includes the properties of global states where an adaptation can occur. The constraints on behavior replacement should also be investigated. The goal of those studies is to help the developer when establishing constraints on the adaptation. We expect this will allow to detect automatically valid adaptation points or at least to check that the points specified by the developer are correct.

Studying the relationship between fault tolerance systems that use checkpointing and adaptation in the context of Grid computing is an important perspective. Firstly, finding shared properties between checkpoints and adaptation points would be of great help in establishing properties and constraints on adaptation point placement. Secondly, fault tolerant systems suppose that the execution environment is dynamic since any fault results in changes in the environment. Keeping in mind fault-tolerance related works seems a natural approach since they share with adaptation the need to find “special points” at which the execution can resume. However, fault tolerance systems try to repair faults rolling-back the execution, whereas adaptation does its best until the code is able to react.

By the time, we have only studied the overall architecture for the adaptation of parallel codes. Our work is currently focusing on how to choose the global adaptation point at which the reaction steps should be executed. This work is going to lead to the definition of an algorithm to suspend the execution in a well-defined point without rollback.

We have no precise idea of the overhead required from the developer to make a component able to adapt itself using a generic framework. However,

several examples exist of parallel codes made dynamically adaptable in an ad-hoc way and we expect that having a generic framework simplifies the task of the developer. In order to have a reasonable opinion about this subject, we plan to study how several parallel and distributed codes can be made self-adaptable. In particular, we think of how parallel discrete event simulators can be modified to adapt itself to the execution environment. We expect from these experiments to get a measurement of the work such a goal needs.

References

[1] Djalel Chefrour and Françoise André. Développement d'applications en environnements mobiles à l'aide du modèle de composant adaptatif aceel. In *Langages et Modèles à Objets LMO'03. Actes publiés dans la Revue STI, série L'objet, volume 9*, Vannes, France, February 2003.

[2] Alexandre Denis, Christian Pérez, and Thierry Priol. Portable parallel CORBA objects: an approach to combine parallel and distributed programming for grid computing. In *Proc. of the 7th Intl. Euro-Par'01 Conference (EuroPar'01)*, pages 835–844, Manchester, UK, August 2001. Springer.

[3] Ian Foster, Carl Kesselman, Jeffrey Nick, and Steven Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration. In *Global Grid Forum*, June 2002.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: elements of reusable object-oriented software*. Addison Wesley, 1998.

[5] Globus toolkit. http://www.globus.org.

[6] Emmanuel Jeannot, Björn Knutsson, and Mats Björkman. Adaptive online data compression. In *11th IEEE International Symposium on High Performance Distributed Computing (HPDC-11 2002)*, pages 379–388. IEEE Computer Society, 2002.

[7] Katarzyna Keahey and Dennis Gannon. PARDIS: A parallel approach to CORBA. In *HPDC*, pages 31–39, 1997.

[8] Xavier Martorell, Julita Corbalán, Nacho Navarro, and Jesús Labarta. The NANOS resource management system. In *4th Operating System Design and Implementation (OSDI 2000)*, 2000. Poster session.

[9] OMG. Corba components, June 2002. Document formal/02-06-65.

[10] Christian Pérez, Thierry Priol, and André Ribes. A parallel corba component model for numerical code coupling. In Manish Parashar, editor, *Proc. 3rd International Workshop on Grid Computing*, number 2536 in Lect. Notes in Comp. Science, pages 88–99, Baltimore, Maryland, USA, November 2002. Springer-Verlag. Held in conjunction with SuperComputing 2002 (SC '02).

[11] C. Szyperski. *Component software: beyond object oriented programming*. Addison Wesley, 1998.

[12] Wireless Application Protocol 2.0: technical white paper. http://www.wapforum.org, January 2002.