A framework for dynamic adaptation of parallel components

Jérémy Buisson, Françoise André, Jean-Louis Pazat

A framework for dynamic adaptation of parallel components

Jérémy Buisson[a], Françoise André[b], Jean-Louis Pazat[a]

[a]IRISA/INSA de Rennes, Campus universitaire de Beaulieu, avenue du Général Leclerc, 35042 Rennes CEDEX, France

[b]IRISA/Université de Rennes 1, Campus universitaire de Beaulieu, avenue du Général Leclerc, 35042 Rennes CEDEX, France

Abstract – The emergence of dynamic execution environments such as Grids forces applications to take dynamicity into account. Whereas sudden resource disappearance can be handled thanks to fault-tolerance techniques, these approaches are usually not well suited when resource disappearance is announced in advance. However, this case occurs in particular for resource preemption due to resource sharing or maintenance operations. Similarly, fault-tolerance techniques commonly do not take into account resource appearance. On the other side, dynamic adaptation covers techniques for handling changes in the execution environment. This article presents a framework intended to help developers in the task of designing dynamically adaptable (but not fault-tolerant) components. This article puts the emphasis on an experimental evaluation of the cost of using such a framework.

1. Introduction

The increase of resource consumption by applications is a fact. This is what leads to the introduction of notions such as meta- then grid-computing. Those approaches, that can be coarsely seen as resource pooling, permit to increase significantly the number of resources available to applications. However, increasing the number of resources lowers the mean time between failures. In addition, resource pooling requires users to share the resources; and it prevents them from controlling maintenance operations as they can with their own resources. This makes execution environments dynamic. Applications executed on such environments must take into account the dynamicity of the environment. Otherwise, they would not be able to perform well, and may even not be able to complete.

Dynamic adaptatability is one approach that can be used to tackle the problem of the dynamicity of execution environments. It consists in the ability of applications to modify themselves during their execution according to some observations. If the application observes its execution environment, it thus adapts itself according to its environment.

This article presents a framework for easing building dynamically adaptable components. Section 2 provides a description of dynamic adaptation. Section 3 presents the architectural view of an adaptable component. Section 4 focuses on the problem of coordinating the execution of the adaptation with the execution of the component itself. Section 5 describes the experimental results we obtained with our prototype framework. Section 6 compares our proposal to existing related works.

2. Dynamic adaptation

Dynamic adaptation is an event-based approach for dealing with dynamicity during the execution. When the component observes a change that is significant enough, it decides to react to this change. For example, when a component observes that new processors become available, it may increase its parallel degree by spawning new processes. At an abstract level, dynamic adaptation requires that the component is able to make observations, take a decision and execute a reaction previously

defined. What is exactly observed, how decision is made and which actions must be performed to execute the reaction are closely related to the component and to the goal given to dynamic adaptation. In the given example, the component observes processors because it is able to execute a parallel implementation of itself; the component decides to increase its parallel degree as it aims at executing as fast as possible; it does so by spawning new processes because it is its way of executing on a parallel environment. This shows that dynamic adaptation may not be done without the help of developers. Nevertheless, we can exhibit generic mechanisms and provide developers with a framework for both designing and programming dynamically adaptable components.

3. Framework for dynamic adaptation

The model for dynamic adaptability of software components we defined is divided into several functional "boxes" distributed in three levels as shown on figure 1.

At the functional level, the service provides an implementation of what the component is expected to do. If the component was not dynamically adaptable, it would contain only the service.

The component-independent level contains all mechanisms that can be defined independently of the content of the service functional box. The decider box is the start point of any adaptation. It decides whether the component should be adapted or not. To do so, it relies on incoming events and on some external probes. The connection to the external probes is modeled by the two ports exposed by the decider. The actual trigger of the decision-making process may either result from the reception of an event or be spontaneous. Once the decider has decided that the component should be adapted, it transmits a reaction to the planner. The reaction describes the kind of the adaptation that should be performed. Given this reaction, the planner establishes a plan for applying it. This plan is mostly a collection of action invocations connected by some control flow. This plan is given to the executor, which executes the invocations with respect to the provided control flow. To do so, it relies on coordinator which coordinates the invocations with the execution of the service. As section 4 details, several kinds of coordinators may be used.

The component-specific level is a placeholder for the developer to put specializations of the adaptation framework. The policy permits the developer to specialize the decider for the needs of its component. It describes how decisions can be made. The plan templates describe how the planner can build plans depending on the requested reaction and on the current execution environment. The actions are the elementary tasks that can be invoked from the plans. In order to simplify its task, the developer may be provided with a library of predefined actions. Endly, the relation between the service and the coordinator is weaved thanks to aspect-oriented techniques through the "adaptable"

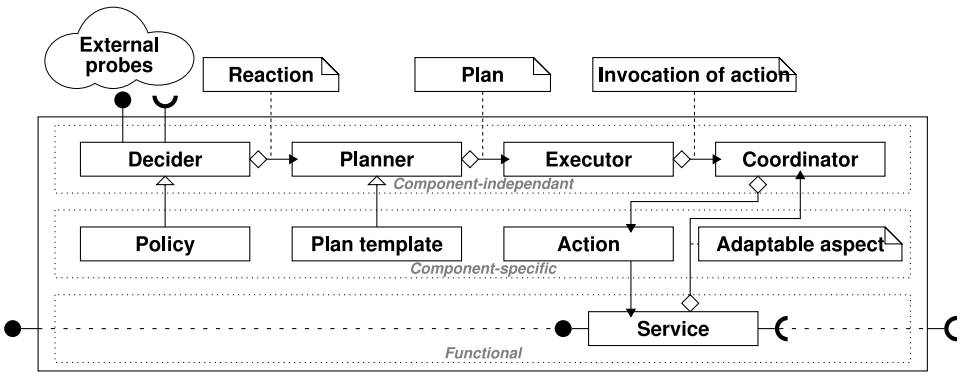

Figure 1. Architecture of an adaptable component

aspect. This aspect is parameterized specifically to the component.

An example of specialization of the framework is given in section 5.1 that describes the demonstrative component used for experiments.

4. Coordination of the invocations of the actions

As it has been previously described in section 3, the purpose of coordinators consists in coordinating the invocations of the actions requested by the plan with the execution of the service. Several coordination policy can be identified and classified according to several concerns:

- Parallel degree. The action may be invoked in a sequential way. When the service contains a parallel code, the action may also be invoked with the same parallel degree as the service functional box.
- Context of execution. The action may be invoked from within the context of the threads of the service. Alternatively, it may be invoked from the context of the processes of the service. It may also be invoked from processes distinct from those of the service, possibly hosted by other machines.
- Synchronization. The action may be invoked asynchronously with regard to the service. It may alternatively require the service to suspend its execution in a special state.

The main purpose of the synchronization concern consists in ensuring that the adaptation does not change (at least semantically) the results produced by the component. Indeed, some actions may not be allowed to be invoked from any state of the service. For example, a matrix redistribution might not be done while the matrix is being modified.

This is why the notion of "point" has been introduced. We call "points" the special states from which actions are allowed to be invoked. Points mostly consists in annotations in the source code of the service functional box; they are instantaneous statements placed by the developer at the locations at which he considers actions can be safely invoked. Given this context, solving the synchronization concern for a coordinator consists in choosing one point at which the action will be invoked.

When the service encapsulates a parallel code, the notion of "point" extends to the global dimension: a "global point" is a collection (one per thread of the service) of points. The choosability of a global point is restricted by a given consistency model. An example consistency model may intuitively assume that the parallel code consists in several parallel steps. Such a model would allow adaptation only between those steps. This model has been described in [2]; a proposal for implementing it has been presented in [3] that restricts the choice to points in the future of the execution path. To do so, it relies on annotations of the code to track the progress of the execution and on a representation of the control flow graph to predict future states. Those annotations are the result of weaving the "adaptable" aspect described in section 3; whereas the points, placed manually by the developer, are the parameters that specialize this aspect for the component.

5. Experiments

The experiments we have made are based upon the NAS Parallel Benchmark [10] 3.1 FFT code for MPI. This code computes the fast fourier transform of a $256 \times 256 \times 128$ matrix from within an iteration. For the purpose of the experiments, it has been slightly modified to use the framework. The modifications are exclusively annotations for indicating adaptation points and for tracking the progress of the execution. Those annotations consist in calls to some functions of the framework.

Experiments have been done using a cluster of dual 2.4 Xeon PC. Each PC hosts at most one process with exactly one thread of the service. For the communications, the service of the component uses LAM-MPI [4]; the framework uses OmniORB as an implementation of CORBA.

5.1. Specialization of the framework for the experiments

For the experiments, the FFT component has been made able to modify its parallel degree depending on the number of machines available in the cluster. The policy is given by figure 2. It states that when new machines become available, the component should spawn new processes; whereas when some machines are announced to disappear, it should stop the corresponding processes.

Algorithm $policy\,()$:
- upon $new\_machines\_appear\,(machines)$:
  $spawn\_process\,(machines)$
- upon $machines\_reclaimed\,(machines)$:
  $terminate\_process\,(machines)$

Figure 2. Policy for adaptable FFT

Figures 3 and 4 show the plan templates for both reactions. The prefix of actions in brackets gives the constraints for each of the coordination concerns listed in section 4.

In order to spawn new processes (reaction $spawn\_process$), the component have to be made available on the corresponding machines (action $deploy\_on$); then the processes must be created (action $spawn\_process\_on$); endly, the matrices have to be redistributed (action $redistribute\_matrices$).

Algorithm $spawn\_process\,(new\_machines)$:
[sequential, in distrinct process, asynchronously] $deploy\_on\,(new\_machines)$
[parallel, in service threads, same point] $spawn\_process\_on\,(new\_machines)$
[parallel, in service threads, same point] $redistribute\_matrices\,()$

Figure 3. Plan template for spawning processes

Similarly, to terminate processes (reaction $terminate\_process$), the component have firstly to redistribute its matrices (action $redistribute\_matrices$); then the processes executed by the reclaimed machines must terminate their execution (action $exit$); endly, everything that was previously installed specifically for those machines has to be cleaned up (action $cleanup$).

Actions are implementations of the invocations requested in plans. Those implementations are dependent on the component and its implementation. For example, the $deploy\_on$ action may transfer files and start required daemons; the $spawn\_process\_on$ action uses of the MPI_Comm_spawn function as the FFT component uses the MPI communication library.

5.2. Timeline of an adaptation

This experiment aims at showing the actions of an adaptation and their timing. This would permit to show how the different phases of dynamic adaptation relates one to each others. This experiment consists in one run of the demonstration component with one adaptation that increases the number of processes from two to four. Figure 5 shows the execution trace near this adaptation.

Algorithm $terminate\_process\,(reclaimed\_machines)$:
  [parallel, in service threads, same point] $redistribute\_matrices\,()$
  if $(local\_machine \in reclaimed\_machines)$ **then**
    [parallel, in service threads, asynchronously] $exit\,()$
    [parallel, in distrinct process, asynchronously] $cleanup\,()$
  end if

Figure 4. Plan template for terminating processes

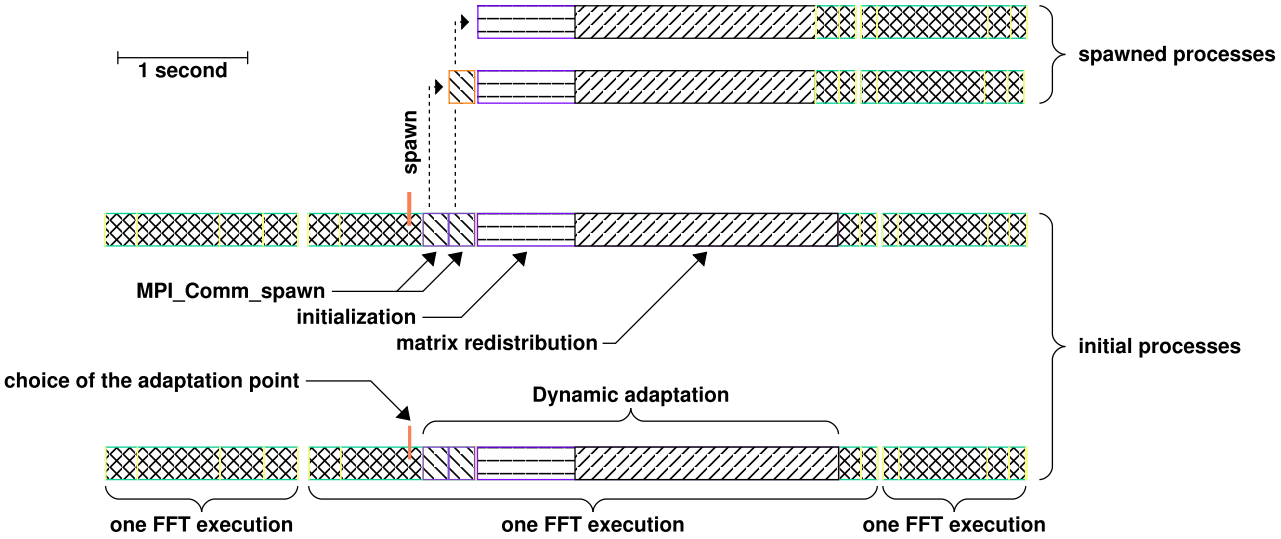

Figure 5. Execution trace of an adaptation that spawns processes

This trace shows that the choice of the adaptation point is done concurrently to the execution of the service. Then, the effective execution of the reaction is postponed to the chosen adaptation point, further in the future of the execution.

The plan begins by spawning new processes. Due to the MPI-2 specification, and in order to be able to stop each process independently of the others, each process has to be spawned individually. In order to simplify the manipulation of MPI communicators, spawned processes participate to the creation of the following processes. This is why one of the spawned processes has a call to MPI_Comm_spawn. Once processes have been spawned, some initialization is performed. This initialization action computes the values that depends on the set of processes, such as communicator objects. Then, matrices are redistributed among the new collection of processes. Endly, the execution of the FFT that was in progress resumes.

5.3. Overhead of the framework

In order to measure the overhead of the proposed framework, the demonstration component has been executed without any adaptation. This experiment permits to evaluate the overhead of the annotations required by the framework. For this experiment, the component executes 2000 iterations on a 16 machines cluster. Table 6 summerizes the execution time of each call to the framework in microseconds. The high maximum value for “function enter” appears to correspond to the first calls for each process. This can be explained by the absence of a complete warmup phase.

function	minimum	mean	maximum	calls per iteration
Adaptation point	14.	21.76	138.	6
Enter condition	7.	10.74	68.	3
Enter function	16.	19.63	510.	1
Enter loop	43.	45.94	58.	0
Fastforward	6.	19.09	104.	7
Iterate in loop	10.	14.05	132.	1
Leave condition	7.	17.34	180.	3
Leave function	8.	9.38	93.	1
Leave loop	9.	13.70	90.	1
Iteration body	777536.	852310.96	1560923.	

Figure 6. Execution times (in microseconds)

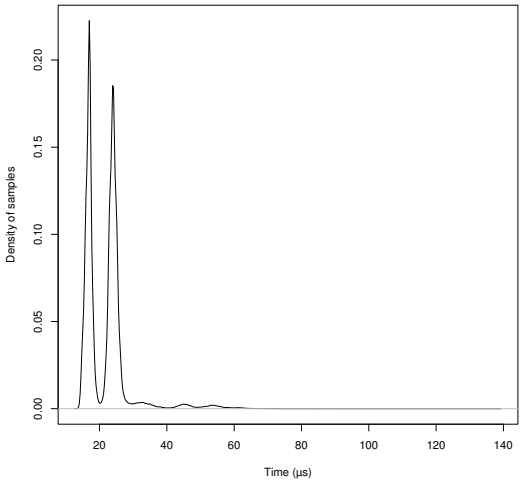

Figure 7. Distribution of measured execution times for the “adaptation point” function

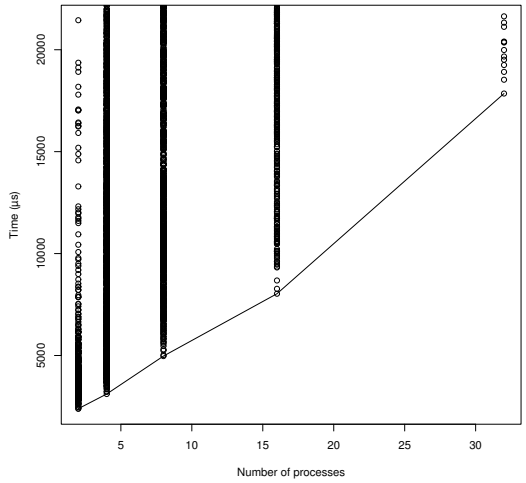

Figure 8. Scalability of the algorithm for choosing the adaptation point

Figure 7 shows the distribution of the measured times amongst the samples for the “adaptation point” function. This curves shows that two execution times have a high frequency: the lowest one corresponds to favorable cache situations; the highest one to defavorable cache situations. The same phenomenon appears with the other functions.

Given the number of calls per iteration for this sample component, the ratio of time lost because of the framework is under 0.05%. Given this result, it appears that the overhead of the framework can be considered as negligible for real world applications.

5.4. Scalability of the choice of the point

As our coordinator solves an agreement problem, the more processes are used, the more time it takes. In order to evaluate the scalability of the algorithm involved in our coordinator, the demonstration component has been executed and adapted with a parallel degree ranging from 2 to 32 processes. Figure 8 shows the time used for choosing the adaptation point at which the reaction is executed.

Care should be taken while interpreting the results. Indeed, the measured time depends not only on the number of processes, but also on the exact time at which the algorithm is triggered (and the

execution time of the service code between adaptation points). Whereas we want to evaluate the former, the latter can not be controlled and scrambles the measures. Computing the minimum time for each number of processes eliminates most of the noise caused by the variation of the trigger time if enough measures are done.

On the figure, dots represent measures; minima for each number of processes are connected by a line. This line appears to evolve almost linearly with regard to the number of processes. This result could be expected as the actual implementation relies on a ring communication scheme.

6. Related works

Several works have proposed architectures for dynamic adaptation such as [1,5,7,8,11,12]. Despite different architectures and approaches, those projects rely on concepts and functionnalities similar to the ones of our approach. Whereas many projects have studied dynamic adaptation in the context of mobile computing, only few are interested in this problem in the area of parallel computing. As the problem described in section 4 of coordinating the execution of the actions appears mainly in the context of parallel computing, many projects did not study it.

Whereas our approach focuses on building adaptable components by extending standard components, the ASSIST [1] approach for dynamic adaptation is based on high-level parallel language constructs. With this approach, the compiler itself is able to emit code for handling dynamicity. Whereas our approach gives full control of dynamic adaptation to the developer, the ASSIST approach permits some dynamic adaptation transparently to the developer. In addition, knowledge of the generated code can be used to specialize the runtime support for dynamic adaptation.

The PCL project [7] aims at easing the construction of adaptable distributed applications. It focuses on how the application can be modified for dynamic adaptation thanks to a runtime providing some reflexive programming support. In order to support reflection, PCL introduces the notion of "adapt sites", which are special nodes in the control flow graph that contain collections of unordered (and potentially concurrent) tasks. Intercession operators allow modifications of the collection of tasks associated to each adapt site. In addition, PCL defines a language for expressing when and how the application should be adapted thanks to "adapt methods". Whereas PCL mixes in a single function probes query, decision-making and planning of the adaptation, which may ease the global understanding of the adaptation, our framework separates these concerns in distinct components, which may simplify the design and reuse of more complex adaptation strategies. Endly, PCL defines a model for synchronizing the adaptation [6]. A comparison of the PCL model to ours is given in [3].

7. Future work

Although several projects address the problem of dynamic adaptation, only few of them provide developers with an abstract model of dynamic adaptation. Providing tools to design and reason about dynamically adaptable software is one of the upcoming challenges. A basis for a design methodology has been proposed in [9]. We are currently working with the team producing ASSIST [1], which includes facilities for dynamic adaptation, in order to propose a common abstract model that could be mapped to our frameworks. It could be expected from such a work to provide conceptual tools to ease the design of dynamic adaptation independently of the concrete platform.

Resource disappearance can be dealt with fault-tolerance mechanisms. However, fault-tolerance focuses on sudden resource disappearance, whereas maintenance operations for example could be announced in advance. Such announcements should be used to anticipate resource disappearance instead of blindly waiting for a fault to occur. Moreover, fault-tolerance approaches fail to make the

application benefit from appearing resources. On the other side, the event-based nature of dynamic adaptation makes it particularly suitable when changes are announced in advance. Fault-tolerance and our approach to dynamic adaptation are complementary in their way to address the dynamicity of the execution environment. Convergence of the two approaches within a single framework should be investigated. In particular, the synchronization concern of the coordinator functional box within dynamically adaptable components introduces the notion of "point". This notion can be compared to checkpoints that can be used to implement fault-tolerance. Although the two notions do not match exactly, they might rely on the same infrastructure, possibly leading to a unified framework.

References

[1] Marco Aldinucci, Sonia Campa, Massimo Coppola, Marco Danelutto, Domenico Laforenza, Diego Puppin, Luca Scarponi, Marco Vanneschi, and Corrado Zoccolo. Components for high performance grid programming in the grid.it project. In *Workshop on Component Models andd Systems for Grid Applications*, June 2004.

[2] Jérémy Buisson, Françoise André, and Jean-Louis Pazat. Dynamic adaptation for grid computing. In P.M.A. Sloot, A.G. Hoekstra, T. Priol, A. Reinefeld, and M. Bubak, editors, *Advances in Grid Computing - EGC 2005 (European Grid Conference, Amsterdam, The Netherlands, February 14-16, 2005, Revised Selected Papers)*, volume 3470 of *LNCS*, pages 538–547, Amsterdam, February 2005. Springer-Verlag.

[3] Jérémy Buisson, Françoise André, and Jean-Louis Pazat. Enforcing consistency during the adaptation of a parallel component. In *The 4th International Symposium on Parallel and Distributed Computing*, July 2005.

[4] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[5] Pierre-Charles David and Thomas Ledoux. Towards a framework for self-adaptive component-based applications. In *DAIS'03*, volume 2893 of *LNCS*. Springer-Verlag, November 2003.

[6] Brian Ensink and Vikram Adve. Coordinating adaptations in distributed systems. In *24th International Conference on Distributed Computing Systems*, pages 446–455, March 2004.

[7] Brian Ensink, Joel Stanley, and Vikram Adve. Program control language: a programming language for adaptive distributed applications. *Journal of Parallel and Distributed Computing*, 63(11):1082–1104, November 2003.

[8] John Keeney and Vinny Cahill. Chisel: a policy-driven, context-aware, dynamic adaptation framework. In *4th International Workshop on Policies for Distributed Systems and Networks (POLICY'03)*, pages 3–14. IEEE, 2003.

[9] Malcolm McIlhagga, Ann Light, and Ian Wakeman. Towards a design methodology for adaptive applications. In *Mobile Computing and Networking*, pages 133–144, May 1998.

[10] NAS. Parallel benchmark. http://www.nas.nasa.gov/Software/NPB/.

[11] Pierre-Guillaume Raverdy, Hubert Le Van Gong, and Rodger Lea. DART : a reflective middleware for adaptive applications. In *OOPSLA'98 Workshop #13 : Reflective programming in C++ and Java*, October 1998.

[12] Maria-Teresa Segarra and Françoise André. A framework for dynamic adaptation in wireless environments. In *Technology of Object-Oriented Languages and Systems (TOOLS 33)*, pages 336–347. IEEE, 2000.