Consistent Rollback Protocols for Autonomic ASSISTANT Applications

Carlo Bertolli[1], Gabriele Mencagli[2], and Marco Vanneschi[2]

[1] Department of Computing, Imperial College London 180 Queens Gate, London, SW7 2AZ, UK
c.bertolli@imperial.ac.uk

[2] Department of Computer Science, University of Pisa Largo B. Pontecorvo 3, I-56127, Pisa, Italy
{mencagli,vannesch}@di.unipi.it

Abstract. Nowadays, a central issue for applications executed on heterogeneous distributed platforms is represented by assuring that certain performance and reliability parameters are respected throughout the system execution. A typical solution is based on supporting application components with adaptation strategies, able to select at run-time the better component version to execute. It is worth noting that the efficacy of a reconfiguration may depend on the time spent in applying it: in fact, albeit a reconfiguration may lead to a better steady-state behavior, its application could induce a transient violation of a QoS constraint. In this paper we will show how consistent reconfiguration protocols can be derived for stream-based ASSISTANT applications, and we will characterize their costs in terms of proper performance models.

1 Introduction

Today distributed platforms include heterogeneous sets of parallel architectures, such as clusters (e.g. Roadrunner), large shared-memory platforms (e.g. SGI Altix) and smaller ones, as off-the-shelf multi-core components also integrated into mobile devices. Examples of applications that enjoy such heterogeneity are Emergency and Risk Management, Intelligent Transportation and Environmental Sustainability. Common features are the presence of computationally demanding components (e.g. emergency forecasting models), which are constrained by the necessity of providing results under a certain *Quality of Service* (QoS). To assure that the QoS is respected, applications must be *autonomic*, in the sense that their components must apply proper adaptation and fault-tolerance strategies.

A reconfiguration can dynamically modify some implementation aspects of a component, such as its parallelism degree. In some cases, when components are provided in multiple alternative versions, a reconfiguration can also dynamically select the best version to be executed. Multiple versions can be provided in order to exploit in the best way as possible different architectures on which the computation may be currently deployed and executed. For this reason programming

M. Alexander et al. (Eds.): Euro-Par 2011 Workshops, Part I, LNCS 7155, pp. 139–148, 2012.
© Springer-Verlag Berlin Heidelberg 2012

models for autonomic applications include a *functional logic*, which is in charge of performing the computation, and a *control logic* (or manager), aimed at assuring the required QoS levels in the face of time-varying execution conditions. In this paper we do not focus on the policy under which a reconfiguration is taken. Rather we are interested in how reconfigurations are implemented, and their impact on the application performance.

The complexity of a reconfiguration protocol depends on the way in which the application semantics (*computation consistency*) is preserved during the reconfiguration itself. Several research works, especially in the Grid computing area, have studied consistent reconfiguration protocols for general parallel applications (e.g. MPI computations), by applying coarse-grained protocols [1] involving the whole set of functional logic processes, and for specific programming models (e.g. Divide-and-Conquer applications), in which reconfigurations are applied at task granularity [2] level. In many cases we may experience a strong reconfiguration overhead, that could avoid the component to respect the required QoS. In other words, even if a reconfiguration leads to a better situation from a performance standpoint, the cost of a reconfiguration protocol should be taken into account if we want to develop effective adaptation models.

In this paper we show how to derive consistent reconfiguration protocols, focusing on the specific case of dynamic version selection. Our approach is characterized by performance models that can be used to dynamically predict the reconfiguration cost and its impact on the provided QoS. We show our contribution for the **ASSISTANT** programming model [3], which is our research framework for studying autonomic high-performance applications.

The paper is organized as follows: Section 2 introduces the main points of the ASSISTANT programming model. In Section 3 we describe a high-level modeling framework for deriving consistent reconfiguration protocols and we provide a specific technique based on a rollback approach. In Section 4 we assess the correctness of the reconfiguration model through experiments.

2 The ASSISTANT Programming Model

ASSISTANT is our framework for autonomic applications and it is based on structured parallel programming [4] to express alternative parallel versions of a same component. ASSISTANT allows programmers to define parallel applications as graphs of parallel modules (i.e *ParMod*), interconnected by means of streams, i.e. possibly unlimited sequences of typed elements. The ParMod semantics is characterized by two interacting logics:

- ***Functional Logic*** or Operating Part: it encapsulates multiple versions of the parallel module, each one with a different behavior according to several parameters (e.g. memory utilization and expected performance). Only one version at time is allowed to be active;
- ***Control Logic*** or Control Part: it implements the adaptation strategy by analyzing the current platform and application behavior and by issuing reconfiguration commands to the Operating Part.

We introduce a new construct, called *operation*, implementing the functional part of a version and the corresponding adaptation strategy applied when that version is executed. A ParMod includes multiple operations which totally describe its functional and control logics. For lack of space in this paper we focus on the interactions between the control and the functional logic, which follows the abstract scheme depicted in Figure 1. The computation performed by the functional logic can be reconfigured at implicitly identified reconfiguration points (e.g. between the reception of two subsequent tasks).

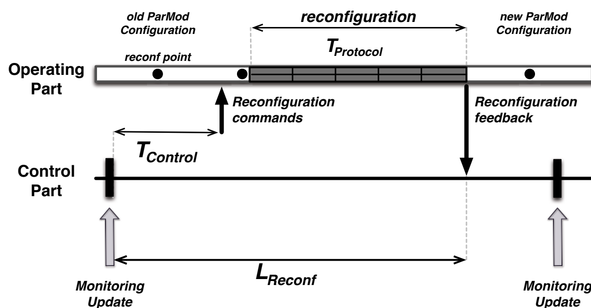


Fig. 1. Interaction scheme between functional and control logics

When a context update is received by the ParMod monitoring, the Control Part decides the set of reconfigurations by executing a control algorithm ($T_{Control}$). After that, a set of reconfiguration commands are sent to the Operating Part which applies them at the first reconfiguration point. For applying them, the Operating Part processes cooperatively execute a reconfiguration protocol, which induces a corresponding overhead (i.e. $T_{Protocol}$). We focus on two general structured parallelism models, namely task-farm and data-parallel. A ***task-farm*** computation is based on the replication of a given functionality F (e.g. a user-provided function) which is applied to a set of elements scheduled from an input stream. Each application of F to an input element gives place to a result, hence the set of results forms an output stream. From an implementation level an emitter process receives input elements and schedules them to workers; the collector receives results and delivers them to the output stream. A performance model can be formally derived: let us denote with T_E , T_W and T_C respectively the service time of the emitter, worker and collector. By considering them as successive stages of a pipeline system, we can calculate the inter-departure time of results from the collector as $T_{farm} = \max\{T_E, T_W/N, T_C\}$, where N is the parallelism degree (i.e. the number of workers).

A ***data-parallel*** computation in ASSISTANT is, like task-farm ones, stream-based, where each input task gives places to a composite state, which is scattered amongst a set of workers by a scatter process. The workers apply a same user-provided function (say G) sequentially on each element of their partition and for a fixed number of iterations (or steps) or until some convergence condition is

satisfied. At each step workers may be required to communicate between themselves, in this case the data-parallel program is characterized by some form of stencil. At the end of the computation the resulting composite state is gathered on a gather process and delivered to the output stream. A simple performance model for data-parallel programs considers the scatter, workers and gather processes as successive stages of a pipeline graph. The results inter-departure time is $T_{dp} = \max\{T_S, T_W, T_G\}$, where T_S , T_W and T_G are the service times of the three functionalities. In particular T_W is the worker execution time: this value depends on the number of steps performed and it accounts for the calculation time and the communication time at each step of the data-parallel computation.

Finally, the ParMod implementation also includes a set of manager processes that implement the Control Logic. Managers may be replicated on different architectures on which the computation can be executed to assure their reliability in face of failures. A replicated set of managers along with the related processes implementing the functional part of a version, can be dynamically deployed and started during a reconfiguration phase.

3 Consistent Reconfiguration Protocols

We suppose a model which is quite general for a broad class of parallel applications: tasks are independent (no internal state of the ParMod survives between different task executions) and idempotent (a task can be repeatedly calculated without getting the ParMod into an inconsistent state).

If we take a computation snapshot, there are: (i) a set of tasks T_{IN} which are still to be received and which are stored in the input streams; a set of tasks T_P currently in execution on the ParMod; and a set of task results T_{OUT} previously produced by the ParMod on the output streams. *The goal of a consistent reconfiguration protocol is to properly manage the set T_P of tasks.* We can formalize the concept of consistency according to two notions:

Definition 1 (Weak Consistency). *All input tasks should be processed by the ParMod and their results delivered to the intended consumers.*

Note that this definition does not admit to loose any result but it permits their replication.

Definition 2 (Strong Consistency). *All elements produced on the input stream are processed by the ParMod and their results delivered to the intended consumers at most one time.*

We introduce a formalization methodology which is based on a proper modeling tool enabling us to define protocols in terms of tasks and results.

3.1 Formalization

Our methodology is inspired by the ***Incomplete Structure*** model (shortly I-Structure), introduced with other purposes in data-flow programming models [5]

and previously used in [6] to model fault-tolerance protocols. An I-Structure is a possibly unlimited collection of typed elements, uniquely identified by sequence identifiers. There are two ways of accessing an I-Structure:

- we can read (**get**) the element stored at a given position. If it is empty, the operation blocks the caller until a value is produced on that position;
- we can write (**put**) a value to a given position. A write semantics assures the write-once property: i.e. it is not possible to perform a write more than once on the same position.

In Figure 2 is depicted how the I-Structure tool is used: each task input stream of a ParMod is mapped to a single I-Structure denoted with *IN*, and result output streams are mapped to a further I-Structure denoted with OUT.

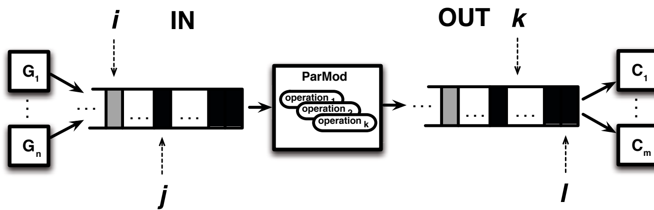


Fig. 2. I-Structure model applied to a ParMod

For a correct understanding of how I-Structures are used, note that there is not any notion of ordering between the IN and OUT elements: broadly, a task produced on an input stream has an index that may be different w.r.t the index of the corresponding result. For each I-Structure we can identify two indexes: one of the last consumed element (e.g. j and l in Figure 2) and the one of the last produced element (e.g. i and k). By using them we can precisely characterize the sets T_P , T_{IN} and T_{OUT}. For instance T_P is the set of elements with indexes on the IN I-structure from 0 to j to which we have to subtract all elements whose results have been produced to the output streams, i.e. results with indexes on the OUT I-Structure from 0 to k.

Although a formal description of proofs about reconfiguration protocols can be expressed through the I-Structure methodology, in this paper we are mainly interested in defining reconfiguration protocols at the level of implementation, by using the information derived from the abstract I-Structure model.

3.2 Implementation

At the implementation level interactions between ParMods are exploited by proper typed communication channels. In order to correctly implement the I-Structure model, we have to extend the basic ParMod implementation [3]: in fact the model requires that elements can be recovered at any time during the

computation, simply passing their unique identifier. In a classical channel implementation, when a message is received (i.e. extracted) from the channel buffer, its content can be overwritten by successive messages. To this end we have two main implementation choices:

- the channel run-time support can be equipped with message logging techniques [7]: i.e. when a message is stored in the channel buffer, it is also copied into an external memorization support that can not be overwritten;
- message recovery can be faced directly at the application level: i.e. we can require that application components can re-generate elements on-demand.

Even if the first approach may induce an additional communication overhead, it is completely transparent from the application viewpoint. Nevertheless in this paper we suppose to adopt the second approach: i.e. every ParMod is able to re-generate past tasks/results. Anyway the presented protocols are independent on the way in which stream elements are recovered.

Therefore we can map the two I-Structures IN and OUT to different communication channels Ch_{IN} and Ch_{OUT}. For the purpose of re-generation, input tasks are labeled with their input sequence identifiers, whereas the results include a reference to the sequence identifier of the corresponding input task. In this way we assure that all ParMods have a common view of task and result identifiers.

Finally, we define the notion of ***Vector Clock*** (VC). A VC models a correspondence between output stream identifiers and input stream identifiers. It is a set of pairs of the form $(1, k_1), (2, k_2), \ldots, (N, k_N)$, where the first element of each pair is an result identifier, and the second one is the corresponding task identifier. Another important notion for our purpose is the *maximum contiguous sequence identifier* (shortly MC), which is the maximum task identifier on the input stream such that all its predecessors are included in the vector clock.

3.3 Description of Reconfiguration Protocols

In this section we describe consistent reconfiguration protocols focusing on version switching reconfigurations, in which we stop executing a *source* operation and we start executing a *target* one. Two techniques can be identified:

- we can wait for the source operation to perform all tasks in T_P and then make the target operation start from the first task which was not consumed by the source operation. We denote this technique as ***rollforward protocols***;
- when the Control Part notifies an operation switching, the source operation can simply stop its execution. Then the control is passed immediately to the target operation that has to re-obtain the tasks in set T_P and re-start their execution. We denote this kind of approach as ***rollback protocols***.

In this paper we focus on a generic rollback protocol, and we show how to optimize it for data-parallel programs. In the description we assume that the target operation has been previously deployed and it is ready to start its execution. For a comprehensive analysis of rollforward protocols, interested readers can read [8].

In a rollback protocol, the target operation needs to obtain all tasks in T_P from the related generators. To this end the Control Part should provide to the target operation some kind of information. Depending on the data passed we obtain different protocols:

- the information passed is the MC value: in this case the target operation will request the generators to re-generate elements whose identifier starts from $MC + 1$ to the sequence identifier of the message on the top of the Ch_{IN} buffer queue. Note that this techniques may induce a duplication of results: therefore it implements the weak consistency definition;
- the source operation can pass the whole Vector Clock to the target one, that drives the re-generation of T_P by issuing to generator components only the missing identifiers. Note that this protocol avoid the re-execution of previously performed tasks, hence it implements the strong consistency definition.

The choice of applying this kind of protocol depends on the amount of work which can be lost. We can quantify it depending on the parallelization scheme adopted by the source operation. If the source operation implements a task-farm computation, we have to re-execute at most $N + 4$ tasks (i.e. one task for each of the N workers; two tasks on the emitter and the collector; two tasks on the input and the output channels). In addition, in the first version of the protocol, we have also to sum up all tasks whose results have been delivered in an unordered way to the output stream. On the other hand, if the source operation implements a data-parallel program, we have to re-execute at most 5 tasks: one on the scatter process; one currently executed by workers; one corresponding to the result processed by the gather process; one on the input channel and a result on the output channel. In this scheme the input and the output streams are ordered between themselves, therefore, in both the versions of the protocol there are no further tasks that must be executed.

Optimizations Based on Checkpointing Techniques. Let us suppose a special case in which source and target operations are data-parallel programs with the same characteristics (e.g. in terms of the stencil definition). In this case we can think to transfer the partially transformed state from the source to the target operation in order to reduce the rollback overhead (time spent in re-executing tasks belonging to the set T_P). Of course this optimization is applicable only if it does not compromise the computation consistency, i.e. depending on the specific properties of the two data-parallel versions.

If this approach is viable, a straightforward solution is to take a snapshot of (all the workers) the computation of the source operation and transfers it to the target operation. The snapshot includes also all messages currently stored in the channel buffers, as well as in-transit messages. As it can be noted, this snapshot-based solution is valid only under the hypothesis that the source and target operations have the same parallelism degree. Moreover the whole computation state formed by the local state of each worker may be inconsistent, due to the fact that workers may be computing at different stencil steps. The reason behind this is that we consider data-parallel programs not based on a step-synchronous

logic, but workers are allowed to proceed in their computation depending only on the availability of their input stencil information (i.e. their data dependencies).

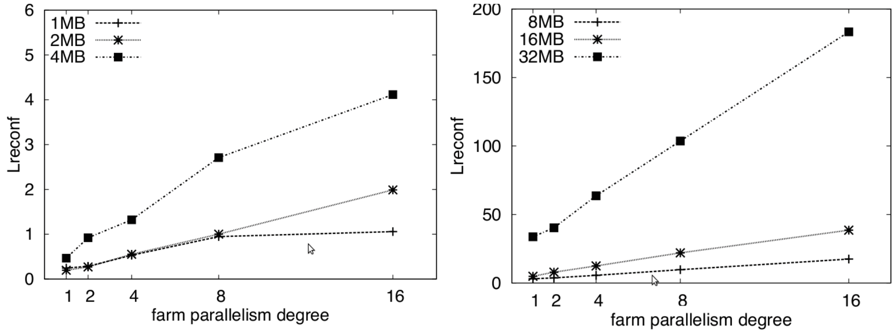
Therefore, the main point consists in how a consistent state is reached by workers. A solution to this problem is based on *checkpointing*: workers can perform periodic checkpointing activities of their local partitions at the very same frequency (e.g. every d steps). Note that checkpointing activities can be performed independently by workers, with the consequence that the last checkpoint from a worker can be different to the ones of others (see [6]). Therefore, to assure that the transferred state is consistent, when workers are issued to perform an operation switching they have to first identify the last checkpointing step which all have passed (a.k.a a *recovery line*), and then transfer the corresponding checkpoints. The selection of the recovery line can be performed by running a two-phase leader-based protocol as described in [6].

4 Experiments

We have tested the behavior of the rollback protocol on a flood emergency management application, developed in the context of the Italian In.Sy.Eme. project (Integrated System for Emergency). This application includes three ParMods: a *Generator* of environmental data, providing input tasks describing the current state of each point of a discretized space of a river basin; a *Forecasting ParMod* implementing a flood forecasting model, that for each input point resolves a system of differential equations. The numerical resolution of the system gives place to four tri-diagonal linear systems, which are solved according to an optimized direct method (see [9] for further details). This ParMod includes two operations, respectively based on a task-farm and on a stencil data-parallel parallel program; the last ParMod implements a *Client* visualizing the results.

In these tests the task-farm is mapped to a cluster including 30 production workstations. The data-parallel operation is executable on two multi-core architectures: (1) a dual-processor Intel Xeon E5420 Quad-Core featuring 8 cores; (2) a dual-processor Intel Xeon E5520 Quad-Core featuring 8 cores. The operations have been implemented using the MPI library: on the cluster architecture we supported the task-farm with the LAM/MPI implementation, while on the multi-cores we have used the shared-memory version of MPICH.

The generic rollback protocol has been tested on the operation switching from the task-farm to data-parallel version. The optimized protocol is exploited for switching between two instances of the data-parallel operation mapped to the two multi-cores. Figures 3a and 3b show the behavior of the the reconfiguration overhead L_{reconf} (i.e. total time in seconds for applying the reconfiguration) by varying the parallelism degree of the source operation and the size of the solved systems. Consider the case of systems of size $32\,MB$ and parallelism degree equals to 8. In this case, the mean service time of each worker is $10.25\,sec.$ and by applying the performance models of parallelism schemes introduced in Section 2 and the reconfiguration cost model introduced in Section 3.3 we obtain that $L_{reconf} \leq (N+4)\,T_W = 123.023\,sec.$. This value is a slight overestimation



(a) L_{reconf} by varying the parallelism degree and the task size (1, 2 and 4 MB).

(b) L_{reconf} by varying the parallelism degree and the task size (8, 16 and 32 MB).

Fig. 3. Evaluation of rollback protocol for Task-Farm to Data-Parallel switching

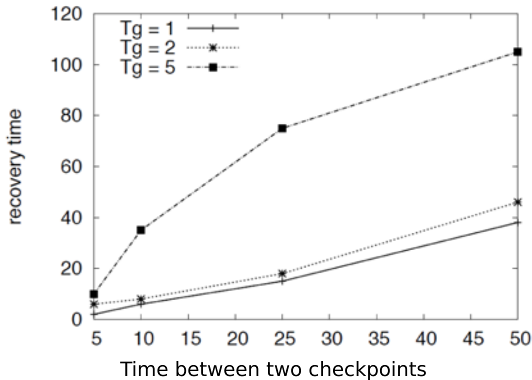


Fig. 4. Recovery time on the target operation by varying the checkpointing frequency and the G grain

than the real experienced one (i.e. 103.6233 *sec.*), due to the fact that in this experiment the number of rolled-back elements has been 11 instead of 12.

Figure 4 shows the time needed to perform the optimized recovery protocol on the target operation by varying the checkpointing frequency and the cost of applying the function G on the local partition of each worker. Clearly, this time also depends on the instant at which reconfiguration is issued, aside of the checkpointing frequency: for instance, if we perform checkpointing with low frequency but the reconfiguration must be applied immediately after a checkpoint, clearly the recovery cost is small. Nevertheless, the behavior of the recovery cost, in average, decreases with the checkpointing frequency. As we can see with these tests, with a grain of $T_G = 5\,sec.$, the selected pattern for issuing reconfiguration commands makes the recovery time strongly increase when the checkpointing

frequency decreases. From this, we can see that higher T_G times should be supported by more frequent checkpointing operations, if our target is to minimize the recovery time.

5 Conclusions

In this paper we have introduced two consistent reconfiguration protocols for ASSISTANT applications, supporting version switching activities. The first protocol supports the switching between any kind of parallel computations. The second protocol represents an optimization when the source and target operations are "similar" data-parallel programs. It is based on a periodic checkpointing protocol and on transferring the state of the last recovery line from the source to the target operation. For the protocols we have introduced performance models to predict their overhead, and we have assessed the expected results by real experiments.

References

1. Kennedy, K., et al.: Toward a framework for preparing and executing adaptive grid programs. In: Proceedings of the 16th International Parallel and Distributed Processing Symposium, IPDPS 2002, pp. 322–326. IEEE Computer Society, Washington, DC (2002)
2. Blumofe, R.D., Lisiecki, P.A.: Adaptive and reliable parallel computing on networks of workstations. In: Proceedings of the Annual Conference on USENIX Annual Technical Conference, p. 10. USENIX Association, Berkeley (1997)
3. Bertolli, C., Mencagli, G., Vanneschi, M.: A cost model for autonomic reconfigurations in high-performance pervasive applications. In: Proceedings of the 4th ACM International Workshop on Context-Awareness for Self-Managing Systems, CASEMANS 2010, pp. 3:20–3:29. ACM, New York (2010)
4. Cole, M.: Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. Parallel Comput. 30, 389–406 (2004)
5. Arvind, Nikhil, R.S., Pingali, K.K.: I-structures data structures for parallel computing. ACM Trans. Program. Lang. Syst. 11, 598–632 (1989)
6. Bertolli, C., Vanneschi, M.: Fault tolerance for data parallel programs. Concurrency and Computation: Practice and Experience 23(6), 595–632 (2011)
7. Elnozahy, E.N.M., Alvisi, L., Wang, Y.M., Johnson, D.B.: A survey of rollback-recovery protocols in message-passing systems. ACM Comput. Surv. 34, 375–408 (2002)
8. Bertolli, C., Mencagli, G., Vanneschi, M.: Consistent reconfiguration protocols for adaptive high-performance applications. In: The 7th International Wireless Communications and Mobile Computing Conference. Workshop on Emergency Management: Communication and Computing Platforms (2011) (to appear)
9. Bertolli, C., Buono, D., Mencagli, G., Vanneschi, M.: Expressing Adaptivity and Context Awareness in the ASSISTANT Programming Model. In: Vasilakos, A.V., Beraldi, R., Friedman, R., Mamei, M., et al. (eds.) Autonomics 2009. LNICST, vol. 23, pp. 32–47. Springer, Heidelberg (2010), doi:10.1007/978-3-642-11482-3_3