A Cost Model for Autonomic Reconfigurations in High-Performance Pervasive Applications

Carlo Bertolli
Dept. of Computer Science
University of Pisa
Pisa 56127, Italy
bertolli@di.unipi.it

Gabriele Mencagli
Dept. of Computer Science
University of Pisa
Pisa 56127, Italy
mencagli@di.unipi.it

Marco Vanneschi
Dept. of Computer Science
University of Pisa
Pisa 56127, Italy
vannesch@di.unipi.it

ABSTRACT

In the last years we have seen the diffusion of platforms including high- performance nodes (e.g. multicores) and powerful mobile devices (e.g. smartphones) interconnected by heterogeneous networks. Relevant examples of applications targeting these kinds of platforms are Emergency Management and Homeland Protection which provide computing/communication activities characterized by user-defined Quality of Service constraints. In this paper we introduce the ASSISTANT programming model for adaptive parallel applications. ASSISTANT components are specified in multiple versions, each one dynamically selected according to an adaptation strategy aimed to target the required QoS levels. For these applications a key-issue is a well-defined adaptation semantics featuring a cost model which describes the overhead for reconfiguring a component (e.g. when switching between versions). In this paper we introduce our approach and we evaluate this cost on a flood management application.

Author Keywords

High-Performance Computing, Adaptivity, Autonomic Computing, Application Reconfigurations.

ACM Classification Keywords

D.1.3 Programming Techniques: Concurrent Programming—*Parallel Programming.*

General Terms

Design, Experimentation, Performance.

INTRODUCTION

Latest years have been characterized by a significant diffusion of several classes of parallel platforms (e.g. multi-/many-core) with different features in terms of interconnection networks, homogeneous or heterogeneous processors, caching hierarchy organizations and shared memory supports. According to the current trend in mobile technology, portable devices can also be equipped with these parallel chips and GPUs, thus rendering the embedding of compute- intensive functions quite feasible at low power consumption.

An increasing number of distributed platforms are characterized by the presence of multiple classes of computing resources, as in Grids [9], characterized by loosely coupled computing elements, and in next-generation Clouds [8] and Pervasive Grids [17] featuring the utilization of mobile nodes and communication networks. Programming distributed applications for these platforms is a critical issue, both for the highly dynamic behavior of the execution environment and for the presence of time-variable user requirements. This is especially true in many real-world scenarios characterized by hard and critical *Quality of Service* (QoS) requirements (e.g. in terms of quantitative and qualitative metrics as the computation performance and the user degree of satisfaction). For these reasons the application configuration, that is the specific identification of application components and their mapping onto the available computing resources, can not be statically defined but the application itself must be able to dynamically take *reconfiguration decisions*. The possibility to apply application reconfigurations is referred in many research works with the term *adaptivity* or autonomicity [14], and it is an unavoidable feature for many distributed systems such as risk and emergency management.

Adaptivity has been studied in both mobile and parallel programming contexts. For instance in [16] it is presented a framework for mobile applications which adapts the quality of visualized data according to the available network bandwidth. In [10] the execution platform is able to perceive the user motion and to automatically start task migrations between different environments. In [6] a distributed hierarchical control of parallel components based on algorithmic skeletons [5] is introduced, focusing on the possibility to modify the parallelism degree according to an adaptation strategy expressed by reactive policy rules. These approaches belong to a widespread methodology for adaptivity which requires a *transparent* approach to all the autonomic processes: i.e. application reconfigurations are completely subsumed by the run-time support which is responsible for granting the correctness and the consistency of the computation during the reconfiguration phase. This approach enormously simplifies the application development but requires a deep knowledge of the logical structure of the computation.

ASSISTANT is our research framework for programming high-performance adaptive applications. Each ASSISTANT component is able to exploit a parallel computation according to well-known parallelism schemes (i.e. the Structured Parallel Programming approach [5]). The run-time support provides multiple classes of reconfigurations including: (i) the run-time variation of the component performance by modifying the parallelism degree; (ii) the definition of multiple versions of the same component, each one featuring a parallelism scheme, a sequential algorithm and optimizations suitable for specific classes of computing resources and for different QoS requirements. In ASSISTANT the run-time support is responsible for preserving the computation consistency and correctness during these reconfigurations, whereas the programmer is mainly involved in defining multiple versions of the same component and the adaptation strategy which decides when a reconfiguration can bring to a significant gain in the application execution (e.g. a performance improvement or a better quality of the results).

Reconfiguration activities induce a necessary run-time overhead to make the new application configuration effective. In many real-time situations the predictability of this cost and its influence on the expected performance is of special interest. In this paper our main objective is to present the ASSISTANT programming model and its definition and to clearly describe a *reconfiguration cost model* for quantifying the overhead introduced by the autonomic behavior. We also present a first study in which we quantify the reconfiguration overhead of an existing ASSISTANT application for real-time flood emergency according to an application-dependent notion of consistency.

In this paper we will introduce the ASSISTANT programming framework describing its main constructs and its adaptation cost model. Next, a first implementation of ASSISTANT will be presented discussing specific optimizations and implementation hints. The proposed programming model approach will be applied to a real-world scenario, a distributed system for flood emergency management, introducing the required real-time constraints and quantifying the reconfiguration overhead according to different execution conditions and implementation strategies.

THE ASSISTANT PROGRAMMING MODEL

An ASSISTANT application is composed of distributed and interconnected application modules. By using a proper programming construct (i.e. the *application* construct), the programmer can express an ASSISTANT application as a direct graph of modules interconnected by means of streams of data, i.e. sequences possibly of unlimited length of typed elements. The set of in-going and out-going data streams to and from a module identifies its input and output interfaces. In ASSISTANT we distinguish between two classes of modules (see Figure 1):

- ***Parallel and Adaptive modules*** expressed by means of the **ParMod** construct. With this construct the programmer can express both the functional logic of the module (e.g. the parallel computation) and its control logic. The definition of this logic is a crucial issue in order to efficiently deal with a dynamic execution environment and time-variable QoS requirements;

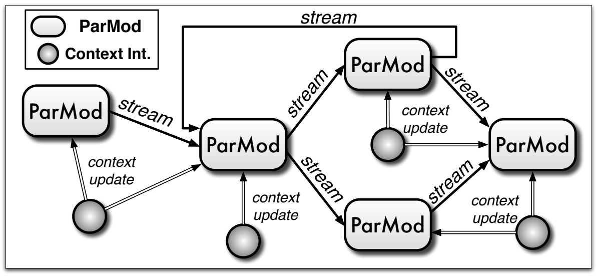

Figure 1. General graph of an ASSISTANT application.

- ***Primitive Context interfaces*** expressed by means of the **primitive_interface** construct. These are sequential modules which periodically exploit application (e.g. to obtain a module service time) and platform monitoring activities (e.g. available network bandwidth). We refer to this information as the current execution *context*.

Context interfaces provide ParMods with the information that they need to select and apply proper adaptation strategies. In some cases raw context values are passed to the ParMod, and they are modeled as key-value pairs, where each value is related to a specific sensor or probe deployed on the platform. In other cases some pre-processing is applied by context interfaces on raw sensor data, possibly synthesizing a whole data set in a single value. An example of this latter case is given by a context interface providing the average temperature of a given area as the average of the values provided by a set of sensors deployed in the area. This solution also helps in reducing the complexity of adaptation strategies given by a possible large number of input context values.

In this paper we focus our attention on adaptation strategies describing how they can be expressed in a ParMod and how they can be implemented, and focusing on the main issue related to the cost of applying them. We will show in future work the development in ASSISTANT related to context interfaces.

The ASSISTANT ParMod

The ParMod construct is used to express parallel adaptive computations. From an abstract point of view a ParMod can be seen as composed of two cooperating entities:

- the **Operating Part** executes the *Functional Logic* of the module. Its execution is activated by the reception of elements from its input interfaces (i.e. in-coming data streams), and it produces results on its output interfaces (i.e. out-going data streams). The elaboration is applied to each input element and it can be a sequential or a parallel computation expressed according to any scheme of Structured Parallel Programming [5] even in complex and compound forms (e.g. stream-parallel schemes such as pipe, task-farm but also data-parallel schemes such as map and different classes of communication stencils). For defining the functional logic of a ParMod we completely inherit the parallel programming model approach which has been developed in our past research works [19]. We can express several parallelism schemes by defining: (1) a proper distribution strategy for the input tasks (e.g. multicast, on-demand and scatter); (2) a set of logical units (i.e. virtual processors) that execute a sequential function on their data and which are mapped onto a set of implementation processes (i.e. worker) responsible for their execution; (3) a collection strategy for the results produced by virtual processors (e.g. gather and FIFO);

- the **Control Part** of the ParMod executes the *Control Logic* of the module. In our approach to adaptivity the run-time support provides a set of possible and predefined ParMod reconfigurations, whereas, the adaptation strategy, i.e. when and how the computation has to be reconfigured, must be directly expressed by the programmer by means of a specific programming construct. From our point of view this is an effective way to optimize application execution w.r.t. the state of the underlying platform and variable QoS requirements.

In Figure 2 it is depicted the ParMod abstract structure. The Operating Part is activated either according to a *data-flow* scheme (i.e. the module waits for values from all the ingoing streams) or to a *non-deterministic* behavior (e.g. a CSP-like semantics based on guarded commands). As hinted, the Operating Part can perform a sequential or a parallel computation, hence it can be implemented by means of a set of interconnected and distributed processes. The Control Part is started whenever a monitoring message is received: (1) from the set of primitive context interfaces connected to it, as described above; (2) from the Operating Part, which can periodically send execution monitoring information concerning the actual performance of the computation (e.g. its throughput) and other execution metrics (e.g. memory occupation). In both cases we refer to this information as a *monitoring update*. When a monitoring update is received, a set of reconfigurations can be decided according to a programmer-defined adaptation strategy. Many strategies can be expressed to deal with several execution conditions and to configure the ParMod behavior with the goal of guaranteeing in time specific QoS objectives (e.g. as introduced in our previous works [1]). For an ASSISTANT ParMod the run-time support provides two classes of reconfiguration activities:

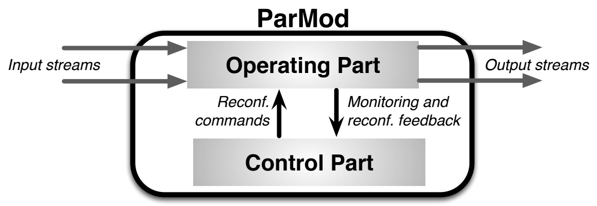

Figure 2. Abstract overview of an ASSISTANT ParMod.

- ***Functional Reconfigurations***. pervasive environments are characterized by strongly heterogeneous computing and network resources, featuring a time-variable degree of availability. The ParMod computation can be mapped onto different classes of resources (e.g. different kinds of high-performance architectures and mobile devices). Based on the actual resource state, the control logic can select which is the best platform onto which mapping the computation. "Best" in the sense that, according to the actual platform conditions and QoS requirements, the control logic can infer the mapping which probably leads to a better expected performance. The dynamical change of the execution mapping is a critical reconfiguration activity, not only for well-studied implementation issues (e.g. task migration and how to preserve consistency and correctness) but also for the relevant differences between previous and new available resources. For instance, a parallel computation for a cluster architecture could not be efficiently executed on a set of mobile nodes, e.g. due to their limitations in terms of memory and processing capacity. For this reason we provide the programmer with the possibility of expressing, for each ParMod, multiple alternative versions of the same computation each suitable for specific classes of computing resources. These operating modes can feature different sequential algorithms (e.g. with different memory occupations) and/or parallelism schemes (e.g. ensuring better scalability and performance on a specific class of architectures). Nevertheless, they preserve the same input and output interfaces of the ParMod in such a way that the selection of an alternative version does not modify the definition of the global application graph;
- ***Non-Functional Reconfigurations*** are *structural changes* which involve the modification of some implementation aspects (e.g. the parallelism degree of the operating part computation), without modifying the currently executed version. As is known, these kinds of reconfigurations are especially useful when we deal with the resolution of irregular and highly variable parallel algorithms, but also in many distributed scenarios characterized by dynamic resource utilization conditions both for communication networks and computational nodes.

The logical interconnections between the Operating and Control Part (Figure 2) are required to implement the reconfiguration protocol. When the control logic has selected the reconfiguration which must be executed, proper *reconfiguration commands* are sent to the Operating Part (to all involved processes). When the reconfiguration is completed, the Operating Part sends a corresponding *feedback message* to its Control Part which notifies the completion of the reconfiguration phase. In the rest of this section we describe the control logic of an ASSISTANT ParMod an the interaction with its functional logic.

ParMod Control Logic

The adaptive behavior of a ParMod has the main objective of maintaining desired execution properties despite time-varying execution conditions. These requirements can be expressed according to different specifications:

- we might require to optimize some execution parameters (e.g. the number of completed tasks), i..e. the Control Part is responsible for solving an ***utility optimization*** problem;

- we might require to maintain specific execution parameters within a user-defined range (e.g. keep the service time in a given range). In this case we refer to this approach as a ***threshold specification*** problem;
- we might require to maintain some execution parameters as closer as possible to a set of desired reference values, as in a classical ***set-point regulation*** problem [12].

Following these specifications, in ASSISTANT we can express two classes of adaptation strategies:

- *Reactive Control*: the programmer expresses a proper mapping between specific run-time operation conditions and corresponding module reconfiguration activities (both functional or non-functional);
- *Proactive Control*: instead of merely react to stimuli, being *proactive* means that a ParMod can consciously involve acting in advance of a future situation. This approach needs a systematic use of predictions (e.f. for time-varying workloads and resource utilization) and on-line optimization techniques [4].

Though the second approach is an interesting research issue in the field of autonomic and self-adaptive computing [14], in this paper our attention is focused on the first approach and the general structure of our programming model. Our results, as the adaptation cost model which will be introduced in the next section, are sufficiently general to be extended with different control strategies such as predictive approaches which will be studied in future works. In ASSISTANT it is possible to express a reactive behavior by defining how the Operating Part must be reconfigured in response to specific events or conditions. The Control Part periodically receives monitoring updates allowing it to identify if the actual ParMod configuration behaves as the user expects. Hence, the periodically receiving of such updates makes it possible to identify the presence of some *QoS violations*: (i) some execution parameters are not equal to certain desired reference values; (ii) some execution parameters are not within a required range (e.g. the average service time is higher than a threshold).

The main essence of such kind of control is to properly react to these QoS violations by automatically modifying the actual ParMod configuration, in such a way as to reach the desired execution goals as soon as possible. The mapping between undesired conditions and reconfiguration activities is a key-issue. A proper mapping can be defined exploiting a form of model reflecting the computation behavior. As hinted, the parallel computations of an ASSISTANT ParMod are well-known parallelism schemes, characterized by specific interaction patterns between parallel processes and a clear and well-defined semantics, which make it feasible the definition of proper *performance models*. For performance model we intend an analytical formulation for:

- the expected performance of the computation, for instance its average service time, in function of the parallelism degree and the current interarrival time. These models are based on Queuing Theory [15], by considering the set of messages exchanged between ParMods as a traffic flow between service nodes. In previous works we have studied the performance models of ASSISTANT applications in several schemes and configurations [1];
- the overall memory occupation of a parallelism scheme, in function of the parallelism degree, the task size and the memory occupation of the sequential algorithm. In [2] memory utilization models are dynamically instantiated to configure a parallel computation executed on a set of mobile nodes equipped with limited memory capacity.

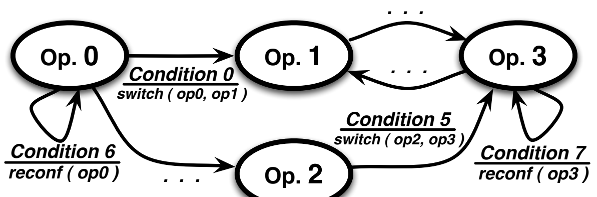

Figure 3. Example of a ParMod control automaton.

In ASSISTANT the reactive control logic is formally expressed by means of a *control automaton*, in which:

- each *internal state* of the automaton corresponds to a different operation (i.e alternative version);
- *input states* are logical combinations of QoS-related or platform-related boolean expressions, e.g. stating the presence of a certain QoS violation and a certain level of network and computing resource availability;
- *output states* are reconfiguration actions related to each transition. They can be: non-functional reconfigurations, denoted with the reconf(OP_i), an example of which is the use of the *parallelism* N construct, to modify an operation parallelism degree; or functional ones, denoted with switch$(\mathrm{OP}_i, \mathrm{OP}_j)$, which identifies a switching between two different operating modes (i.e. from OP_i to OP_j).

In Figure 3 it is depicted a general overview of a control automaton. Its starting state is the operating mode which the programmer has specified with the keyword *initial* in the operation definition. According to the ParMod semantics only one operating mode can be marked as initial. In our automaton we can observe that non-functional reconfigurations are self-transitions, whereas transitions between different internal states are functional reconfigurations. A control automaton is syntactically expressed as scattered in each operation definition within a ParMod: each operation defines, aside of its functional logic, how the module can react to specific events when that operating mode is currently executed. To do so, the programmer makes use of a set of nondeterministic clauses expressed in a specific programming construct (i.e. *on_event* Figure 4). They identify the outgoing possible transitions from the current operating mode.

```
on_event:
    condition 0:
        do
            //Non-functional reconfiguration:
            value = cost_model(. . .);
            parallelism value;
        enddo
    ...
    condition N-1:
        do
            //Functional reconfiguration:
            operation_Name.start();
        enddo
```

Figure 4. The on_event construct.

Finally note that reconfiguration commands can only be received at certain points of the Operating Part execution in which computation consistency can be guaranteed by exploiting specific reconfiguration protocols. A full description of consistency issues for parallel computations is out of the scope of this paper, but in our model the Operating Part is able to process reconfiguration commands only when its parallel computation reaches a so-called ***reconfiguration safe-point***. In general terms we can identify two classes of reconfiguration points:

- when one of the input guard, defining the input_section of the Operating Part, is activated. These points characterize the beginning of two successive elaborations;
- further finer points can be defined. Programmer-transparent points are consistency lines obtained by means of coordination protocols, by using semantics properties of the parallel programs (e.g. after a global reduce the state is consistent), or literature protocols from the distributed system world [7].

Therefore, after a notification of reconfiguration commands from the Control to the Operating Part, it may be needed to first reach a consistent state before applying the reconfiguration activities. Consistency can be obtained by making processes implementing the Operating part run proper distributed algorithms. In ASSISTANT these algorithms can be implicitly defined by using the parallel structure definition or explicitly implemented by the programmer. When performing an operation switching two kinds of implicit solutions can be characterized:

- **Rollforward** techniques, in which the Operating Part processes continue going on with the computation until a consistent state is reached. In the experiment section we will see how this can be simply implemented for task-farm structures;
- **Rollback** techniques, in which the Operating Part processes periodically perform some check-pointing protocols. When the operation switching is notified, a previous common recovery line is computed and the new target operation will go on from that point.

ParMod Reconfiguration Cost Model

Regardless of the particular control logic defined by the programmer, we can formally describe the interaction pattern (Figure 5) between Control and Operative Part. A ParMod can be executed by using different alternative ***configurations***. For configuration we mean the used operating mode (i.e. the current active operation) and its implementation (e.g. its parallelism degree). Whenever the Control Part receives a monitoring update, the corresponding control logic is started and its main objective is to select a new ParMod configuration which is more suitable for the current execution condition. To achieve this new configuration, the Control Part must select a set of reconfiguration activities. The average execution time for this selection algorithm is a parameter $T_{Control}$ of our abstract model.

When reconfigurations have been decided, the Control Part is also responsible for instantiating the new configuration. We can identify different run-time support activities:

- if the control logic decides to perform a non-functional reconfiguration, it has to interact with the processes implementing the operating part and to possibly instantiate new ones. For instance, if a parallelism increase is decided, it has to instantiate a set of workers on available computing nodes;
- if the control logic decides to switch to a different operation, the entire Operating Part must be instantiated on a proper set of computing nodes.

In the model we denote the cost of both cases as the average time $T_{Deployment}$. This overhead can be heavily influenced by different factors:

- if we suppose a static knowledge of the underlying distributed platform (i.e. target nodes are known), all or a subset of processes implementing the Operating Part must be dynamically deployed on the selected computing nodes. The parameter $T_{Deployment}$ identifies the average time needed to transfer the process source codes (or the corresponding executables) and to instantiate them on the corresponding processing nodes;
- in many real-world cases, especially for dynamic distributed systems (as in [17]), the Control Part has not a static knowledge of available nodes. Each operation corresponds to a general class of execution platforms (e.g. cluster archi-

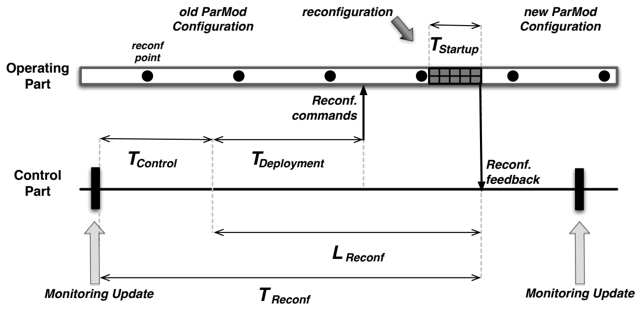

Figure 5. Interaction between the Operating and the Control Part.

tectures or multicore nodes). When a reconfiguration decision is made, the Control Part has to discover a set of compatible nodes on which it can start the selected operation or a set of workers. This discovery phase is part of the $T_{Deployment}$ overhead.

Once the reconfigurations have been decided and when all the necessary deployment actions have been done, the Control Part notifies specific reconfiguration commands to the Operating Part. When the Operating Part elaboration reaches a reconfiguration safe-point, the reconfiguration commands can be processed and the new configuration can become effective. At this point, according to the class of reached safe-points and of reconfigurations which are required, some run-time activities could be necessary: e.g. in the case of a functional reconfiguration we must switch to a different operating mode. When reconfiguration commands are received, the processes implementing the new operation have been already deployed on the selected nodes. In some cases, we may need to wait for the completion of the consistency protocol which is executed. Next, the new operating mode is activated and connected to the linked application modules. In general the average time to make a reconfiguration effective is the overhead $T_{Startup}$ depicted in Figure 5. In our adaptation cost model we can identify two important reconfiguration metrics:

- the ***total reconfiguration time*** (i.e. T_{Reconf}) is the time between the reception of a monitoring update and the time in which the new configuration is being executed;
- the ***reconfiguration latency*** is the time needed to reconfigure the ParMod, that is the time for the deployment activities and for starting the new module configuration.

By referring to Figure 5 these two metrics can be described by the following expressions:

$$L_{Reconf} = T_{Deployment} + \delta + T_{Startup} \tag{1}$$

$$T_{Reconf} = T_{Control} + L_{Reconf} \tag{2}$$

δ is the average time between the reception of reconfiguration commands and the feedback externalization. δ depends on the average time between two subsequent safe-points.

The presented adaptation cost model makes it possible a partial overlapping between the total reconfiguration time and the "normal" execution of the Operating Part. For instance we can observe from Figure 5 that the $T_{Control}$ and $T_{Deployment}$ overheads are always overlapped with the Operating Part execution. Therefore the Control Part sends the reconfiguration commands only when: the new configuration has been decided (i.e. the control logic execution is completed) and after the completion of all the necessary deployment actions. In our model $T_{Startup}$ is the only overhead which is not overlapped with the ParMod computation. It is an unavoidable critical overhead which must be paid to make the new configuration effective. We can also observe that, when no reconfigurations are necessary, the $T_{Control}$ delay is completely overlapped with the Operating Part elaboration, $T_{Deployment} = 0$ and the $T_{Startup}$ overhead is negligible because it only consists in the delay of making an asynchronous notification of the feedback message from the Operating Part to the Control Part.

In the next section we will describe a first ASSISTANT implementation. Our focus is on specific implementation details, different optimizations, reconfiguration protocols, consistency issues and the quantification by experiments of the $T_{Startup}$ overhead in different scenarios.

ASSISTANT IMPLEMENTATION

This implementation design is related to the case in which ASSISTANT is mapped onto parallel and distributed platforms (e.g. wired/wireless networks of clusters and multicore nodes) in which we support the Operating and Control Parts in two distinct sets of processes/threads.

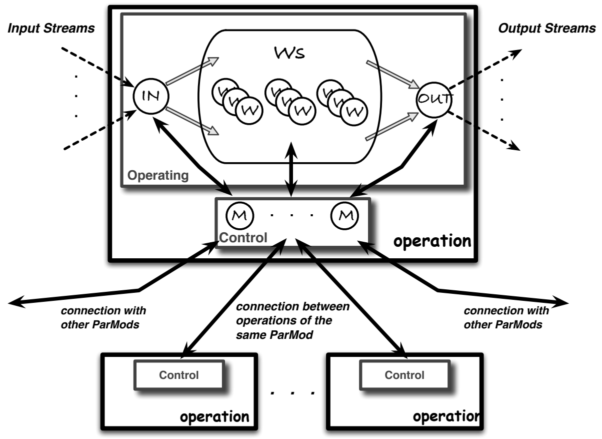
Input Streams
Output Streams
Ws
IN
W
OUT
Operating
M
M
Control
operation
connection with other ParMods
connection between operations of the same ParMod
connection with other ParMods
Control
operation
Control
operation

Figure 6. Implementation scheme of a ParMod.

Figure 6 shows the chosen implementation design of a ParMod. The figure includes multiple operations belonging to the same ParMod and we describe in more detail the upper one. According to the ASSISTANT semantics the operations must share the same input and output streams. In this figure these streams are connected to the upper operation which is the currently active operating mode. Each operation includes a set of processes belonging both to the Operative Part and to the Control Part of the ParMod. For the upper operation the Operating Part processes are:

- *IN*: this can be a single process, or a replicated set of processes, implementing the **input_section** of the functional logic of the operation (see the ASSISTANT section);
- *W*: as described in the previous section, workers are the parallel activities of the operation. Multiple workers can operate independently (e.g. as in a task-farm or in a map) or in cooperation (e.g. as in a data-parallel with stencil). In the latter case, additional links between workers are needed to implement cooperations;
- *OUT*: this can be a single process, or a replicated set of processes, implementing the **output_section** of the functional logic of the operation.

In the figure note that arrows between IN, {W}, OUT are represented in gray color, input/output streams are represented by dashed arrows, while interactions between Control Part and Operating Part processes are black arrows.

In this ASSISTANT design each operation includes a set of processes (called *managers*, *M* in short) which are responsible for locally implementing the ParMod control logic, denoted with the *control* box in each operation. That is, the Control Part of the ParMod is implemented by a set of locally replicated processes scattered among each ParMod operation. In the ASSISTANT definition this choice corresponds to the *on-event* construct (see ASSISTANT section). For each operation, the implementation of the Control Part needs to access the relevant context values to implement the chosen control logic. Therefore, the Control Part of a ParMod, as well as its context view, is partially partitioned amongst the operation instances. In the figure we can also note that the Manager processes implementing the Control Part have multiple interactions:

- interactions with the Operating Part processes of the same operation: Managers may notify some reconfiguration commands. Therefore, these notifications implement the abstract interaction pattern between the Control and the Operating Parts described in the previous section. Note that, unlike the manager processes which are continuously waiting for context events on all its input interfaces, the Operating Part processes are mainly responsible for performing the parallel computation of the ParMod. For this reason, we need to implement the so-called *reconfiguration safe-points*, in which the Operating Part processes can be interrupted from the managers;

- manager processes of the same ParMod interact between themselves for implementing the overall control logic in a distributed way. Information passed between manager processes within the same ParMod can be related to context values shared between different operations (e.g. network latencies on input/output streams). Managers of different operations can be issued to implement an operation switching: suppose that the manager decides to switch to another operation (e.g. a transition between different internal states on the control automaton is verified). In this case the manager may perform some protocol to gracefully stop the processes implementing the current operation (i.e. a rollfoward) before actually performing the operation switch, and only next it can notify the manager process of the target operation to start the computation;

- interactions with Control Parts of other ParMods: these kinds of interactions are both used to share context information across different ParMods, and to notify other ParMods that something has changed inside the ParMod (e.g. an operation switching). For instance, neighbor modules could need to close old connections to the previous operation processes and open new ones to the activated operation. This clearly depends on the kind of support we give to interactions between ParMods: ideally, from a functional viewpoint, one would prefer to make operation switching within a ParMod transparent to other ParMods; practically, we will see that if we select off-the-shelf communication facilities the most efficient and straightforward implementation solution is to create an interaction protocol between linked ParMods to dynamically modify their interconnections.

Consider now the mentioned reconfiguration points of processes implementing an operation functional logic. In ASSISTANT these are specific points in the computation of the related processes in which the operation managers can issue some request, or some relevant context information is notified from the Operating to the Control Part. In more details, if the Operating Part is executing a stream-based parallel program, a straightforward way to define such points coincides with process activations whose semantics depends on their role in the Operating Part:

- *IN*: for each operation these processes implement the input_section guarded command. Activation is hence defined as the selection of one of such guards and interruptions from the managers can be sensed at each subsequent activation. Depending on the activation frequency (e.g. low frequencies), it could be needed to check interruptions more frequently. This can be solved by enabling interruptions also while the *IN* processes are blocked on the alternative command, waiting for one guard to be fired. A further issue is given by the time needed to serve each activation: if this time is large, we may require to enable interruption sensing also during the serving. Note that, in this case, interruption handling may not be consistent with the application semantics: below we show how consistency can be automatically introduced for the parallel structure used in the experiment. Otherwise, if consistency cannot be automatically guaranteed, the programmer can insert proper rollback/rollforward procedures to guarantee consistency;

- *W*: the behavior of each worker is to cyclically receive an input task, to solve it possibly in cooperation with other workers, and to return a result. Activation of these processes is hence defined as the receiving of the input task. As for the *IN* case, we can define implicit interruption sensing points during the resolution of a task for certain classes of structured parallel programs. Moreover, we give the programmer the opportunity of inserting specific interruption points and related rollback/rollforward procedures;

- *OUT*: similarly to the *IN* case, interruptions are automatically sensed during the delivering a result to output streams and during result collection. If needed, the programmer can also insert specific interruption sensing points and consistency guaranteeing procedures.

Finally, in the figure we avoided to represent a further lower-level set of processes responsible of transferring and activating operation processes, which we call *deployers*.

EXPERIMENTAL SCENARIO AND RESULTS

A relevant experiment in which adaptivity and its reconfiguration costs are a crucial issue is given by time-critical real-time applications like Emergency Management [19]. The most important feature of such applications is their dynamic nature and the presence of real-time requirements. In this scenario adaptivity is an unavoidable feature to efficiently deal with variable degree of availability of computing and network resources and of the surrounding environment (e.g. resource utilization level and emergency conditions). In this situation the application configuration must be adapted to target the application QoS requirements (e.g. the application performance or the user degree of satisfaction).

In this paper we focus on a river flood emergency application, limited to only the disaster prediction. In this application the users require the real-time monitoring of the river area (or different sub-areas) and the execution of forecasting models, producing flood predictions (e.g. for the flow level and the corresponding environmental damages). The forecasting model is periodically applied to specific geographical areas. In the first system implementation we have considered the TUFLOW [18] hydrodynamical model, which is based on the real-time resolution of mass and momentum partial differential equations to describe the predicted flow variation at surface, considering a proper discretization of the river basin (e.g. a set of bi-dimensional points). Their numerical resolution (e.g. finite difference methods) yields to a set of linear systems of equations which involve large and banded systems (in our case tridiagonal systems). Such elaborations include data- and compute-intensive processing not only for off-line centralized activities, but also for on-line, real-time and decentralized activities: these computations must be able to provide prompt and best-effort information to the users.

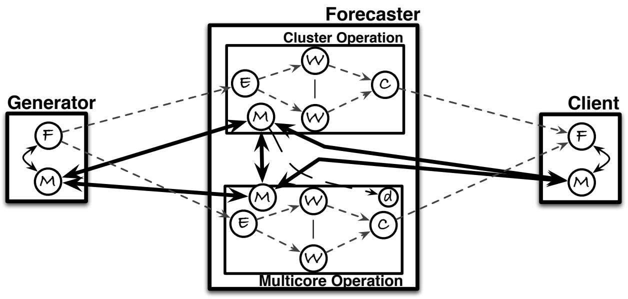

Figure 7. Experimental application scheme.

In ASSISTANT the flood prediction application can be defined with the scheme of Figure 7. A single *generator* (G) ParMod emulates a Wireless Sensor Network (WSN), providing input data to a forecasting ParMod. Each input data is related to a point of the space discretization, and it includes information as the river depth and its speed components. The *flood forecasting* ParMod numerically solves, for each point, the corresponding system of bi-dimensional differential equations (see above), practically consisting in solving four tridiagonal linear systems [18]. We employ a flavor of cyclic reduction algorithm [13] for this task, which is one of the most efficient sequential algorithm to solve banded systems. Results are delivered to a further *client* (C) ParMod, which emulates the visualization tools on a user device. Depending on the mapping, the client emulates a central institutional center, where the overall activities to manage the flood are controlled, or a user mobile node (e.g. a PDA), simulating an active operator near the emergency area.

For the generator and client ParMod we have decided to make use of a single process for the operating part, executing all the functional activities (IN, OUT and Workers) in sequential. This is due to the fact that we assume that the generator and client tasks are not a bottleneck for the application performance, hence they do not require, in these experiments, a parallel support. The quality of the computation can be controlled by: selecting a coarser/finer grain of discretization of the river basin in points; by generating, from the differential equation system, tri-diagonal systems with higher/lower size. The former case influences the length of the stream (i.e. the number of stream elements) for each forecast over the whole monitored river area. The latter case influences the time needed to solve a single task, independently of the selected parallelism form.

In the experiments the flood forecasting ParMod includes two operations: one targeting a cluster of workstations, emulating a central server; another one targeting a multicore processor, emulating an interface node. Both operations are developed according to a task-farm model, where the task is defined as the resolution of a whole differential equation system for each discretization point. For all ParMods we have decided to limit the manager replication degree (for each operation) to 1, in such a way that the monitored operation switching costs are not affected by replica management algorithms, needed to make manager progress in a consistent way. The introduction of full replication support for manager process will be studied in future work.

All ParMods are developed in C++ using MPI for communications inside each operation, and a raw socket interface between different operations and ParMods. The task-farm on the cluster is supported by the LAM/MPI implementation [3], while the operation on the multicore is supported by MPICH for shared memory machines [11]. For the task-farm scheme we have also defined further interruption points for the IN and OUT processes (respectively denoted with *E*, for Emitter, and *C*, for Collector). These points are automatically derived from the program parallel structure:

- the IN process checks for interruptions from the manager before performing a receive from the input streams and the receive itself can be blocked by an interruption. As task scheduling is performed on-demand, the process may remain blocked while waiting for one of the workers to notify that it is free. It can automatically check interruptions from its manager also during this wait;
- similarly, OUT process can be interrupted after sending a result on the output streams, while being blocked in a send operation (in case the related buffer is full) and when waiting for a worker to send a result.

In this paper we show the experimental results related to the cost of operation switching in two notable cases, related to the situation in which the forecasting ParMod is executing the cluster version and the control logic decides to switch to the multicore operation due to context changes. The two cases are:

- **Case 1** the two operations are both active, and the generator and client are connected to the cluster operation. When the cluster manager decides to switch to the multicore operation, it notifies this decision to all IN, $\{W\}$ and OUT cluster processes, to the generator and client, and to the manager process of the multicore. The generator and the client, when getting notified of the switching, close their connections to/from the cluster and open new connections to/from the multicore. The switching is performed in a consistent[1]way according to a *rollforward* technique: when notified of the operation switching, the client first waits for the cluster operation to deliver results of all currently executed tasks (i.e. those assigned before deciding the operation switching);

- **Case 2** the multicore operation is not active and the cluster operation manager must activate the corresponding processes during the operation switching. For this purpose, we map a proper deployment process onto the multicore, whose objective is to start the corresponding operation (i.e. its processes) when requested. In this test case we make use of the same consistency guarantee protocol of the second case (i.e. a rollforward technique).

Clearly, the described cases are not the only ones which can happen in an operation switching scenario. Nevertheless, as we will see, these cases arise some interesting hints related to the design and implementation of ASSISTANT, which we discuss below.

Numerical Results

We have performed tests on an emulated distributed environment:

- the cluster includes 30 nodes Pentium III 800 MHz with 512 KB of cache, 1 GB of main memory and interconnected with a 100 Mbit/s Fast Ethernet;

- the multicore is an Intel E5420 Dual Quad Core processor, featuring 8 2.50 GHz cores, 12 MB L2 Cache and 8 GB of main memory.

All nodes are interconnected through a Fast Ethernet technology (100 Mbit/sec) which is an optimistic scenario. Note that, altought the difference between the computing power of the two architecture is high, the operation switching time is not specifically affected by this factor. Rather, the switching overhead is mainly affected by the communication latencies between managers. The choice of this optimal Ethernet-based configuration is that the goal of these experiments is to study the behavior of the switching delay in a case in which it is not affected by underlying overheads. Indeed, we will study the switching overhead in future works by using slower wired/wireless communication technologies featuring dynamic behaviors (in terms of throughput and latency).

[1]We say that the computation is consistent for this application if all generated input elements are elaborated and the corresponding results are delivered to the client.

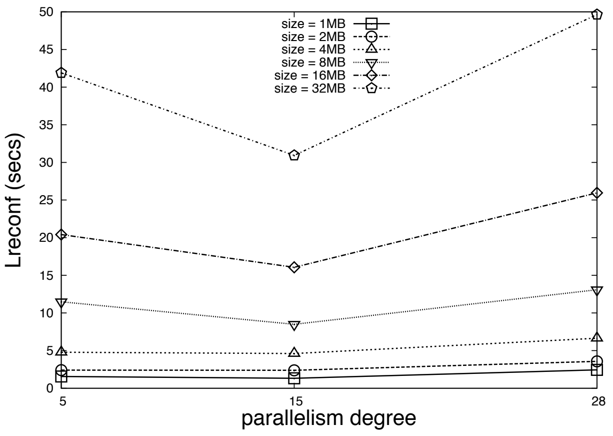

Figure 8. Overhead (L_{reconf}) for Case 1 w.r.t. parallelism degree.

Figure 8 refers to experiments for Case 1. Its is straightforward to note that the rollforward consistency protocol causes the L_{reconf} behavior to be dependent on the task granularity. As hinted, in this protocol we have a first phase in which the cluster operation has to serve all pending requests from the generator and a second phase in which the termination is notified in a flooding fashion from the IN process to workers (a broadcast) and finally to the OUT process. We can note that the time needed to perform the first phase becomes lower by increasing the parallelism degree, while the time needed to perform the second phase becomes larger, as we need to notify the termination to each worker.

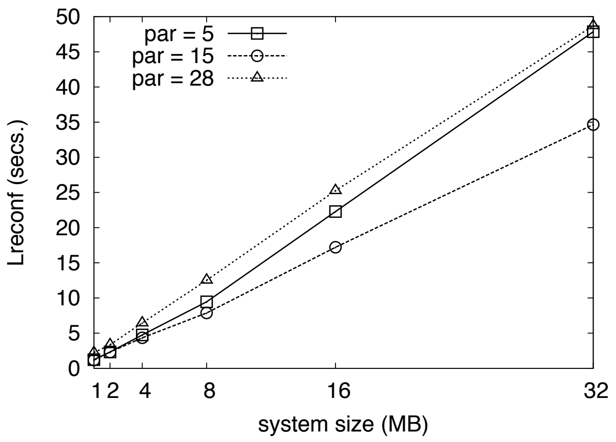

Figure 9. Overhead (L_{reconf}) for Case 2 w.r.t. task granularity.

Also note that the curve with parallelism degree 15 is lower than the other ones, i.e. with parallelism degree 5 and 28. The cause of this behavior can be seen in Figure 8, in which we show the L_{reconf} behavior w.r.t. the parallelism degree. In the figure all curves present their lower value with parallelism 15: this is given by the fact that 15 best approximates the optimal parallelism degree, i.e. adding further workers is useless. For the chosen consistency protocol note that the use of an increasing number of workers is reflected in a higher termination overhead: hence, in this case, adding workers beyond the optimal degree not only is useless in terms of parallel efficiency, but it also increase the rollforward overhead resulting in lower L_{reconf} performances.

Finally, Figure 9 shows the L_{reconf} behavior w.r.t. task granularity for Case 2. We can note two facts: the very same behavior is obtained w.r.t. Case 1, hence the main overhead is given by the consistency protocol; moreover, the dynamic start-up of the target operation during the switching does not add a significant impact to L_{reconf} in this platform configuration. Clearly, as this cost certainly depends on the communication latency between cluster manager and multicore deployer, we will experience higher overheads when passing to lower performance communication networks. We can also note that we do not consider additional overheads for this case, e.g. we suppose that no other deployment and discovery actions are necessary, both for the computing nodes (which are clearly identified in this scenario) and for the source codes or executables (which are already present in the filesystem of the target architecture).

CONCLUSION

In this paper we have presented the ASSISTANT programming model oriented towards the definition of self-adaptive high-performance applications. Target platforms are those in which several classes of strongly heterogeneous computing nodes are dynamically available, hence giving place to the need of an efficient reconfiguration support. In this paper we have introduced an adaptation cost model for ASSISTANT reconfigurations and, by focusing on a flood emergency management application, we have experimented reconfiguration overheads on an emulation of a distributed platform. Results show that the adaptation cost model well describes the expected reconfiguration overheads, and hence these can be taken into account in the overall application performance evaluation with the goal of guaranteeing dynamic user's requests in terms of specific QoS parameters.

REFERENCES

1. C. Bertolli, D. Buono, G. Mencagli, and M. Vanneschi. Expressing adaptivity and context-awareness in the assistant programming model. In *Proceedings of the Third International ICST Conference on Autonomic Computing and Communication Systems*, volume 23, pages 32–47, September 2009.
2. C. Bertolli, G. Mencagli, and M. Vanneschi. Analyzing memory requirements for pervasive grid applications. In *The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, Washington, DC, USA, 2010, to appear. IEEE Computer Society.
3. G. Burns, R. Daoud, and J. Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.
4. E. F. Camacho and C. A. Bordons. *Model Predictive Control in the Process Industry*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997.
5. M. Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Comput.*, 30(3):389–406, 2004.
6. M. Danelutto and G. Zoppi. Behavioural skeletons meeting services. In *ICCS '08: Proceedings of the 8th international conference on Computational Science, Part I*, pages 146–153, Berlin, Heidelberg, 2008. Springer-Verlag.
7. E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.
8. I. Foster. Computing outside the box. In *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, pages 3–3, New York, NY, USA, 2009. ACM.
9. I. Foster and C. Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.
10. D. Garlan, D. Siewiorek, A. Smailagic, and P. Steenkiste. Project aura: Toward distraction-free pervasive computing. *IEEE Pervasive Computing*, 1(2):22–31, 2002.
11. W. D. Gropp and E. Lusk. *User's Guide for* `mpich`*, a Portable Implementation of MPI*. Mathematics and Computer Science Division, Argonne National Laboratory, 1996.
12. J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.
13. R. W. Hockney and C. R. Jesshope. *Parallel Computers Two: Architecture, Programming and Algorithms*. IOP Publishing Ltd., Bristol, UK, UK, 1988.
14. M. C. Huebscher and J. A. McCann. A survey of autonomic computing—degrees, models, and applications. *ACM Comput. Surv.*, 40(3):1–28, 2008.
15. L. Kleinrock. *Theory, Volume 1, Queueing Systems*. Wiley-Interscience, 1975.
16. B. D. Noble and M. Satyanarayanan. Experience with adaptive mobile applications in odyssey. *Mob. Netw. Appl.*, 4(4):245–254, 1999.
17. T. Priol and M. Vanneschi. *Towards Next Generation Grids: Proceedings of the CoreGRID Symposium 2007*. Springer Publishing Company, Incorporated, 2007.
18. B. Syme. Dynamically linked two-dimensional/one-dimensional hydrodynamic modelling program for rivers, estuaries and coastal waters. Technical report, WBM Oceanics Australia, 1991. available at: http://www.tuflow.com/Downloads/.
19. M. Vanneschi. The programming model of assist, an environment for parallel and distributed portable applications. *Parallel Comput.*, 28(12):1709–1732, 2002.