Proceedings of the 21st IASTED International Conference
Parallel and Distributed Computing and Systems (PDCS 2009)
November 2 - 4, 2009 Cambridge, MA, USA

A PROGRAMMING MODEL FOR HIGH-PERFORMANCE ADAPTIVE APPLICATIONS ON PERVASIVE MOBILE GRIDS

Carlo Bertolli, Daniele Buono, Silvia Lametti, Gabriele Mencagli, Massimiliano Meneghin,
Alessio Pascucci and Marco Vanneschi
Department of Computer Science
University of Pisa
L. B. Pontecorvo, 3, I-56127 Pisa, Italy
email: {bertolli, d.buono, lametti, mencagli, meneghin, pascucci, vannesch}@di.unipi.it

ABSTRACT
Pervasive Grids are emerging distributed computing platforms featuring high degrees of dynamicity and heterogeneity. They are composed of fixed and mobile nodes, interconnected through wireless and wired networks. *Adaptivity* is the key point for applications to efficiently exploit Pervasive Grids. We focus on High-Performance pervasive applications, such as emergency management. We present a novel programming model which allows programmers to express high-performance adaptive applications in terms of their parallel and distributed structures. In this programming model a parallel module can be programmed in multiple *versions*, each optimized for a specific platform configuration (e.g. mobile nodes or central servers). The module is programmed to select the best version to handle/serve certain events, such as wireless link failure and user request for a better *QoS*. We experimentally show the efficacy of this approach for a test-based application on a Pervasive Grid.

KEY WORDS
High-Performance Computing, Context Awareness, Adaptivity, Pervasive Grid

1 Introduction

Pervasive Grid computing platforms [13] are composed of a variety of fixed and mobile nodes, interconnected through multiple wireless and wired network technologies. In these platforms the concept of *context* represents the state of logical and physical resources and of the surrounding environment (e.g. sensor data). An example of Pervasive Grid application is risk and emergency management [3]. These applications include data- and compute-intensive processing (e.g. forecasting models) 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 mobile users. For this purpose applications must deal with the dynamic changes of the context: *adaptivity* is the key point for applications to efficiently exploit Pervasive Grids.

We introduce a novel programming model for High-Performance applications on Pervasive Grids, featuring context awareness and adaptivity. We enable application adaptivity in terms of their (parallel and distributed) structures and used algorithms. In this paper we consider a risk and emergency management application. Traditionally, these applications are performed on centralized servers, but we can map (part of) them also onto decentralized nodes, such as PDAs and embedded multicore technology. In fact, alternative computations should be defined to perform the same application task (e.g. forecasting model). These computations are different in the used implementation mechanisms and in the provided Quality of Service (QoS). In general, they can be also mapped onto different resources. This mapping should be performed dynamically, to handle certain context events, such as user requests for different QoS constraints or catastrophic events such as link failures. In this paper we consider the term QoS as a set of metrics, reflecting the experienced behavior of an application such as: its memory occupation, the estimated performance (e.g. the average service time for a stream computation or the completion time of a single task) and the user degree of satisfaction, such that the precision of computed results.

To enable this kind of behavior, applications have to be defined as adaptive. Adaptivity requires that specific programming constructs are provided to programmers to express (a) different "versions" of the same application module, and (b) the dynamic selection of the best version for specific context situations. The programming model, introduced in this paper, allows programmers to express multiple versions of a same computation, optimized to be respectively executed on different resources (e.g. mobile nodes or high-performance central servers). The key point is that these versions are described in a same program, providing constructs to express the dynamic switching to handle programmer defined context events and, in general, to control their performances and the adaptivity strategies.

In Sect. 2 and 3 we summarize the related works and we introduce a test-case application related to emergency management. In Sect. 4 we introduce our programming model. In Sect. 5 we describe a prototype implementation of the programming model related to the test-case and we present and discuss experimental results, validating our approach. Finally in Sect. 6 give the conclusion of this work.

2 Related Work

Adaptivity has been introduced for mobile and pervasive applications by exploiting the concept of context [2]. Context definition includes environmental data (e.g. air temperature, the state of network links and processing nodes and high-level information). Smart Space systems [14] mainly consist in providing context information to applications, which possibly operate on controllers to meet some user defined requirements. Some works focused on the issue of abstracting useful information from raw sensor data in such a way that they can be used to define application adaptivity. For instance, [4] exploits ontologies to model various context information, to reason and share or disseminate them.

General mobile applications must adapt themselves to the state of the context. For instance, a mobile application can exploit optimized algorithms [9] or protocols [6]. In this vision, it is the run-time support (e.g. the used protocol) which is in charge of adapting its behavior to the context. In a more advanced vision adaptivity can be defined as part of the application logic itself [3]. For instance, in Odyssey [10] an operating system is responsible of monitoring the system resources. Significant changes in resource status is notified to applications, which adapt themselves to meet a *fidelity* degree. Adaptivity is expressed in terms of the choice of the used services.

High-performance for context-aware applications is introduced in [8]. In this work high-performance computations are defined as data stream flows of transformations, data fusions and feature extractions. Anyway, they are executed on centralized servers, while mobile nodes are only demanded to result collection and presentation. We go beyond this vision by: (i) allowing programmers to express multiple versions of a same program exploiting different QoS; (ii) allowing programmers to execute proper versions also on mobile nodes.

Independently of pervasive environments, several research works are focused on adaptivity for high-performance parallel programs. In [1] it is shown how hierarchical management can be defined in the case component-based applications are developed according to known parallelism paradigms. Adaptivity for service-oriented applications is also targeted in [11], but it is only discussed for large-scale simulations of non-linear systems. We inherit and extend these research works in our programming model. In this paper we mainly focus on the programming model mechanisms to express adaptivity between multiple versions of a same computation, and on their performances according to known cost models.

3 A Test Case: Flood Applications

We consider a schematic view of an application for fluvial flood management (see Fig. 1). During the "normal" situation several parameters are periodically monitored through sensors and possibly by other services (meteo, GIS). E.g. sensors can monitor the current value of flow level, surface height, flow density in each spatial coordinate.

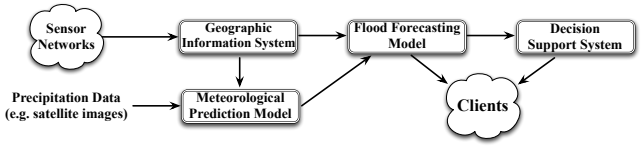

Figure 1. Scheme of a flood management application.

A forecasting model is periodically applied for specific geographical areas. Examples are MIKE 21 [18] or TUFLOW [15], which are based on mass and momentum differential equations. Their discrete resolution requires, for each time slice, the resolution of a very large number of linear (tri-diagonal) systems, which size depends on the model precision required. Parallel techniques are available (e.g. parallel cyclic reduction method) that make it possible reasonable response times in a scalable manner.

Alternatively a human operator can ask the execution of the model for a specific area with specific parameters. A critical situation is that in which the network connection of the human operator(s) with the central servers is down or unreliable. This is possible because we are making use of a (large) set of wired and wireless interconnection links which are geographically mapped onto the critical area. To manage the potential crisis in real time, *we can execute the forecasting model and visualization tools on a set of decentralized resources*, whose interconnection is currently reliable.

Just limiting to this scenario, *it is clear that there is a complex problem in dynamic allocation of software components to processing and communication resources*. Some resources may have specific constraints in terms of storage, processing power, power consumption: the same version of the software components may be not suitable for them, or even may be impossible to run it. Thus, the application must be designed with several levels of adaptivity.

4 A Programming Model for Parallel Adaptive Applications

Our novel programming model *ASSISTANT* (ASSIST with Adaptivity and Context Awareness) is based on the *ASSIST* [16] experience, which has been a starting point to define a novel framework realizing High-Performance pervasive applications with adaptive and context-aware behaviors. ASSIST [16] is a programming environment for expressing parallel and distributed computations according to the structured parallel paradigm. In ASSIST it is not possible to natively express an adaptive application, which is one of the intended goals of ASSISTANT. An ASSIST application is expressed in terms of a set of ParMods (i.e. Parallel

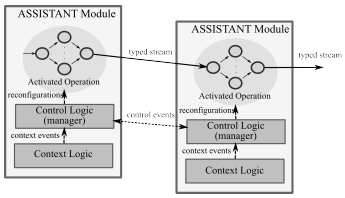

Figure 2. Cooperating ASSISTANT ParMods.

Modules) interconnected by means of typed streams. The ParMod construct includes three sections:

- **input_section**: it is used to express the distribution of received data from input streams to the parallel activities performing the computation, according to primitive constructs (e.g. *on-demand* and *scatter*) or user-programmed ones;
- **virtual_processors**: they are used to express the parallel computation applied to each input data, possibly producing an output result. Virtual processors are the abstract units of parallelism in ASSIST, which are mapped onto a set of implementation processes;
- **output_section**: in this section we express the collection of virtual_processors results and their delivery to output streams, by means of primitive strategies (e.g. *gather*) or user-programmed ones.

4.1 General Semantic of Adaptive Applications

In ASSISTANT we target adaptivity by allowing programmers to express how the computation evolves reacting to specified events. We enable this expressivity in a single programming construct, as an extension of the previous ASSIST ParMod. We can characterize three logics (Fig. 2) that can be used to describe the semantics of an ASSISTANT ParMod:

- **Functional** logic: this includes all the computations (sequential or parallel) performed by the ParMod and expressed according to the ASSIST model. We allow programmers to define multiple versions of the same computation, solving the same problem (e.g. tridiagonal systems solving), and exploiting different QoS. Functional logics of ParMods communicate by means of typed stream of data.
- **Control** logic or *manager*: this includes all the adaptivity strategies and *reconfigurations* performed to adapt the ParMod behavior as a response to specified events. A *cost model* for a certain parallelization schema can express the interested QoS parameters of a computation in function of architecture-dependent parameters such as: the communication latency between processes and the completion time for a specific task. Using proper cost models the control logic can select the best version to execute when a certain context situation is verified, maximizing or minimizing specific QoS parameters (e.g. the module response time or its memory occupation). In ASSISTANT the programmer is provided with high-level constructs to directly express in a simple way this control logic. Moreover managers can interact by means of *control events*, to exploit global reconfigurations which involve a set of ParMods.
- **Context** logic: this includes all the aspects which link the ParMod behavior with the surrounding context. The programmer can specify events which correspond to sensor data, monitoring the environmental and resource state (e.g. air temperature and network bandwidth). It can also specify events related to the dynamic state of the computation (e.g. the module service time).

The main construct of the ParMod, which is used to express these three logics, is *operation*: a set of operations reflects the concept of different versions. Only one operation for each ParMod can be currently activated by its control logic. When a ParMod is started, a user-specified *initial* operation is performed. During the execution, the ParMod context logic or the managers of other ParMods can notify one or more events. The ParMod control logic exploits a mapping between these events and reconfigurations, defined by the programmer, to either select a new operation to be executed, or modify the run-time support of the current operation (e.g the parallelism degree of a parallel computation as described in [17]).

The control logic of an ASSISTANT ParMod can be described as a state graph (Fig. 3). In this graph, nodes are ParMod operations and arcs are events (or their combinations). In the exemplified graph, the initial operation is OP_0. In the case an event EV_0 occurs, we continue executing OP_0 but we modify some aspects of its implementation. For instance we modify its parallelism degree. That is, self-arcs, starting and ending in the same node, correspond to run-time system reconfigurations. Consider now the arc from OP_0 to OP_1 injected by event EV_1. In this case the programmer specifies that if we are executing OP_0

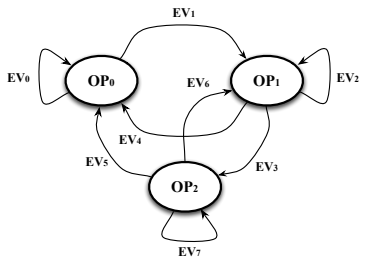

Figure 3. Example of event-operation graph.

and event EV_1 occurs, we stop executing OP_0 and we start OP_1. This switching can include pre- and post- elaborations: e.g. we can reach some consistent state before moving from OP_0 to OP_1 in order to allow the former operation to start from a partially computed result, instead of from the beginning. In the next section we describe the structure of ASSISTANT applications and the constructs that can be used to express them.

4.2 Structure of Applications and Adaptivity Constructs

Fig. 4 shows a syntactic scheme of an ASSISTANT ParMod. It has a *name* and a set of *input* and *output streams*. It can feature a *global state* shared between operations and it can define *events* which it is interested to sense.

Multiple operations can be specified inside a ParMod, each operation possibly featuring a *local private state*. As stated above, each operation inside a same ParMod defines a different version of the same computation and they must all feature *the same input and output interfaces*, in terms of streams. Each operation expresses *all* logics of a ParMod, not only the functional one, and they feature different behaviors in terms of distribution and collection of input and output data, parallelism and distribution strategies (e.g. different algorithmic skeletons [5]). Concerning the control logic, each operation features a different way of managing the QoS, the parallelism degree, the atomicity of its computation and the cooperation with other ParMods.

4.2.1 Events and On_Event construct

We have seen that the ParMod behavior is driven by *events*, which are defined and used by programmers as the basis to define adaptivity. In general, the programmer specifies an initial operation, which is automatically executed when the ParMod is instantiated. Next, we execute reconfigurations in the case of certain events happen. Events are asynchronous and can be generated from either context interfaces or control logic of other ParMods. The programmer can specify conditions on the current state of the ParMod to enable the monitoring of certain events. Thus, reconfigurations can be performed in the case: (a) some predetermined events happen and/or (b) some predicates on the ParMod state are satisfied. That is, the control logic of a ParMod is stateful. Syntactically, the programmer makes use of nondeterministic clauses as shown in Fig. 5.

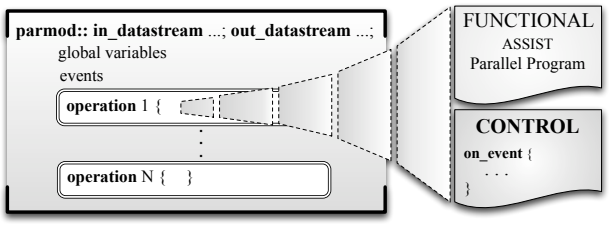

Figure 4. Syntactic view of a ParMod.

```
on_event:
   event_combination_0:
      do
         ... //New parallelism degree:
         parallelism value;
      enddo
   ...
   event_combination_S-1:
      do
         ... //New operation executed:
         operation_Name.start();
      enddo
```

Figure 5. Scheme of the on_event construct.

Event combinations are logical expressions of both events and conditions on the global/local state. Classically, an event is verified if (a) it is present and (b) its local guard on the state (if present) is verified. If the *event_combination* logical expression is satisfied, the corresponding reconfiguration code is executed, exploiting the modification of the parallelism degree (*parallelism* construct) of the actual operation or the activation of another operation. In the latter case the parallelism degree of the new operation is computed (implicitly by the run-time support) before starting it. We notice that the on_event construct specified inside an operation states which are the transitions *starting* from the operation and not the whole set of transitions in the event-operation graph.

5 Implementing a Flood Management Application

We show how to apply our programming model for the flood application described in Section 3. For simplicity, we consider a part of the whole application (see Fig. 1) corresponding to the hydrodynamic model [15, 18]. The modules are depicted in Fig. 6.

The *Generator Module* emulates the previous phases of the forecasting model. It generates a stream of double precision floating point data obtained from sensors, including information related to each river point [15]. It can be programmed in such a way that it provides a different discretization of the river basin.

The *Tridiagonal Solver Module* (TSM) implements a part of the hydrodynamic model which is applied to each input stream element. The most compute-intensive part consists in solving a set of partial differential equations on two dimensions for each specified point. Mainly, this can be reduced to solving a large set of tridiagonal linear systems, which sizes depend on the required accuracy of results. In this implementation we choose a direct cyclic method [7] and we express its parallelism as a *task farm*

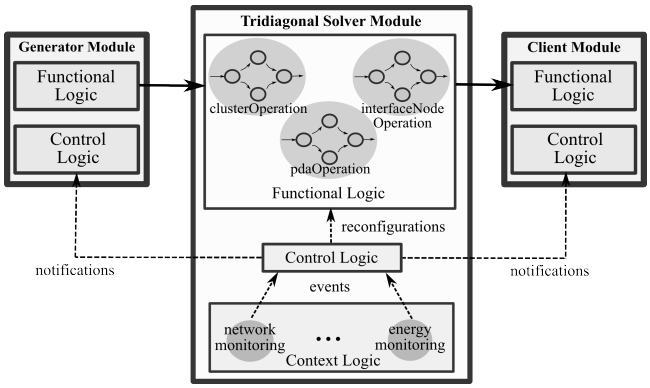

Figure 6. A part of a flood management application.

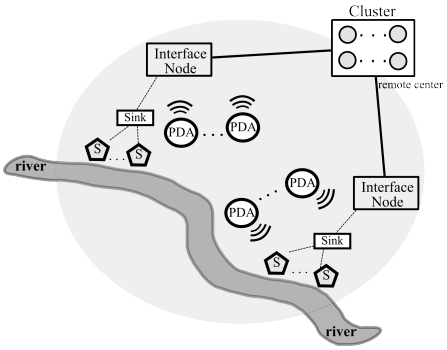

Figure 7. Logical scheme of a Pervasive Grid platform for flood emergencies.

structure: distribution to workers is performed according to an on-demand policy; each worker solves the partial differential equation for each assigned point; collection from workers is performed according to a FIFO policy. The result data are a stream of arrays representing the force components in the X and Y directions on each point of the river. The *Visualization Module* implements the post-processing activities taking as input the solutions of the linear systems.

In our first prototype we map these modules onto a specific Pervasive Mobile Grid infrastructure which scheme is depicted in Fig. 7. Sensor networks, which provide input data through their sink nodes, are located along the course of the river. Mobile nodes such as PDAs, which (at least) are required to run a visualization software, are utilized by the personnel of civil protection. Central server(s), composed of parallel architectures, are located on remote sites. Moreover we can consider the presence of a set of *interface nodes* which are used to interconnect (i.e. exploiting routing activities) mobile nodes and sensor devices with remote central server(s) and performing some computations on-demand (e.g. they can be equipped with multicore processors). All interconnections are implemented by means of both wireless and wired technologies, whose robustness depends on mobility factors and on failures due to emergency conditions.

In this scenario our application needs to adapt according to different context situations (e.g. status of networks, presence of high latency interconnections, energy-related problems, user-defined QoS constraints):

- we can execute the forecasting model on a central server, which allow programmers to express the parallel computations with highest parallelism degrees. Due to emergency condition, the Visualization Module can request a lower service time threshold to reduce the overall completion time. In this case the module adapts its performance modifying its parallelism degree;
- if the connection between the remote central server and the corresponding interface node is no more available, or it can not guarantee the requested QoS, we can execute the model directly on the interface node;
- if the interface node has no sufficient computational power (it performs only routing activities), or if the connections with user devices are temporarily unreliable, we choose to perform the forecasting computations directly on a set of mobile nodes.

In these cases we can reconfigure our application by executing difference versions on different resources. For brevity we describe a part of the implementation of the most critical module, i.e. the Tridiagonal Solver Module.

5.1 Tridiagonal Solver Module

In this section we do not concern on linguistic aspects of the programming model, but we are interested in describing the semantics of its mechanisms. The module implementation is shown in Figures 8,9. In Fig.8 we show the module declaration (i.e. its name and its in-out interfaces).

```
//Tridiagonal Solver Module definition:
module TSM(in_datastream task_in tin,
           out_datastream result_out rout) {

 //Global state shared between operations:
 int actualOperation;
 double averageServiceTime;
 double serviceTimeThreshold;
 ...
 //Definition of events:
 define event serv_time=Visualization_Module.
      newServTime();
 define event con_fail=netMonitorInterface.conFail();
 define event con2_fail=netMonitorInterface.con2Fail();
 ...see next figure...
}
```

Figure 8. Definition of the Tridiagonal Solver Module.

We can assume that the operation performed on the central server will provide a better quality of results w.r.t.

the other operations: it could require more precise input data and it will provide more precise results. This maps in the size of the single elements passed on the output streams. In the pseudo-code the input stream data type *task_in* is a structure of 8 *double*s and the output stream data type are two arrays of *double*s. The size of results, which give the precision of the forecast, is not specified to reflect its dynamic selection.

The module features a set of global attributes which represent the current operation executed (*actualOperation*) and the average service time of the module (*averageServiceTime*). This metric is automatically updated by the run-time system, by exploiting the knowledge of the implemented parallelism structure of the current operation executed. Moreover a client can specify a maximum threshold (i.e. a QoS constraint) for the average service time (*serviceTimeThreshold*) of the module. The module defines three kinds of events which can be notified either from its context interfaces or from the control logic of another module:

- *serv_time*: an event received from the Visualization Module, which value is a new maximum threshold for the average service time;
- *con_fail*: an event produced by a context interface (*netMonitorInterface*) of the module, providing a boolean information related to the connection capability (based on current latency and connection status) between the central server and a considered interface node.
- *con2_fail*: an event produced by *netMonitorInterface*, providing a boolean information related to the connection capability between the interface node and the user mobile nodes.

Fig. 9 shows the pseudo-code of the *clusterOperation* executed on the cluster central server.

The operation program includes definitions of *local private attributes*, which can specify some performance values such as the actual parallelism degree of the operation. Next, follows the operation functional logic as a parallel program. For brevity, we avoid to show it.

The *on_event* construct specifies the adaptive behavior of this operation. Suppose that the client requires a lower QoS in terms of a lower average service time. The control part of the Visualization Module sends a proper control event to the TSM module. Its effect is to re-compute the best parallelism degree, using a proper cost model. The cost model (*costmodel* function) is instantiated with the actual cluster parameters (e.g. communication latency between processes).The *parallelism* construct re-configures the operation to reach the computed *bestDegree*.

The other two event combinations induce an operation switching for the TSM, and they concern the current status of networks. In both cases the control parts of the Generator and Visualization Modules are notified of the decisions with specific *notify* function. Next, the clusterOperation is suspended (*stop*) and the new operation is started (*start*)

```
initial operation clusterOperation {
  //Local variables of the operation:
  int parallelismDegree;
  ...
  //...parallel implementation...
  ...
  //Management section of the operation:
  on_event {
    serv_time(requestedThreshold) && (
        averageServiceTime > requestedThreshold):
      do
         serviceTimeThreshold = requestedThreshold;
         int bestDegree=costmodel(requestedThreshold);
         //Modification of parallelism degree:
         parallelism bestDegree;
      enddo
    con_fail() && (!con2_fail()):
      do
        notify(Generator_Module, con_fail);
        notify(Client_Module, con_fail);
        //Stop the operation consistently:
        this.stop();
        //Activation of the new operation:
        nodeInterfaceOperation.start();
      enddo
    con2_fail():
      do
        ...
        pdaOperation.start();
      enddo
  }
}
...see next figure..
```

Figure 9. Operation of the TSM module for central server.

with a proper parallelism degree and system size, chosen by the run-time support.

5.2 Prototype implementation

We have implemented a first prototype of the programming model applied to the flood application. The prototype (Fig.6) has been tested on an emulation of the Pervasive Grid (Fig.7), based on localized resources (a cluster, some PDA nodes, multicore-based nodes). This prototype consists on a set of processes communicating through sockets.

Fig. 6 shows the first implementation: we emulate the sink node of the sensor network with a single workstation, on which the Generator Module is executed. The centralized server is emulated with a cluster architecture on which we can map the clusterOperation. The cluster is composed of 30 nodes Pentium III 800 Mhz with 512 KB of cache, 1 GB of main memory and connected with a 100 Mbit/s Ethernet. The interface node is emulated with a multicore architecture (IBM CELL Broadband Engine with 6 cores) on which we can execute the nodeInterfaceOperation. The CELL BE features 256 MB of main memory and each core has 256 KB of local memory. Finally the mobile devices are a set of PDAs with 300 Mhz ARM processors (with 64 MB of main memory) on which we map the pdaOperation.

5.3 Experimental Results

We have tested how our prototype can adapt itself to context events and maintaining a specific QoS constraint with an user. In fact suppose that an user requests a maximum service time threshold for system solver module, to reduce the entire execution time of the forecasting computation. In our experiment this threshold is set to 3 seconds. Fig.10 (top) shows the scalability of clusterOperation with different system sizes (i.e. 8, 16 and 32 MB).

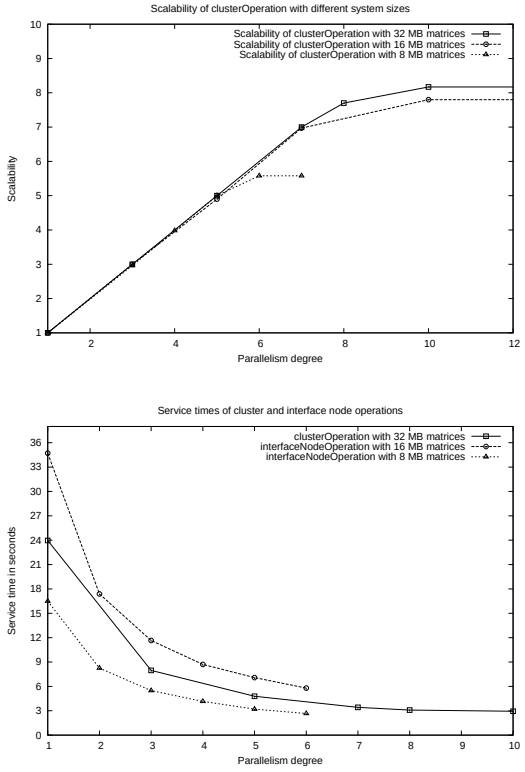

Figure 10. Scalability of clusterOperation with different system sizes (top). Average service times of clusterOperation and interfaceNodeOperation (bottom).

This operation has an expected scalability up to 10 cluster nodes with 32 MB systems. With an higher parallelism degree the scalability of the task farm is limited by the collector process due to communication latencies. If the Visualization Module, in response to a corresponding user intention, requests a maximum service time threshold of 3 seconds, the TSM manager computes the best parallelism degree and the greatest system size to respect the new QoS constraint, according with the cost model. In this test it obtains an average service time of ~ 2.93 sec, consistent with the user constraint.

We have simulated a *con_fail* event by disconnecting the connection with cluster. In this case a reconfiguration starts, and the *interfaceNodeOperation* is activated by the TSM manager. To maintain the QoS constraint, we need to decrease the system size and the precision of forecasting results. In Fig.10 (bottom) the run-time support must reduce the system size from 32 MB to 8 MB, obtaining an average service time of ~ 2.66 seconds (using 6 cores i.e. the maximum parallelism degree in the CELL BE multicore).

When the *con2_fail* event occurs we can execute the computation over an ad hoc network of homogeneous PDAs. The run-time support computes the necessary parallelism degree and system size, to respect the QoS constraint. Table1 shows the average sequential computation time (T_{mob}) for each stream element on a single PDA, in function of different system sizes.

We consider a simple cost model for this case, in which we just consider performance-related factors. More complex models will be developed as future work, including memory and communication bandwidth constraints. The average service time of the pdaOperation can be estimated as following: $T_{pdaOp} = max\{T_{gen}, T_{farm}, L_{com}^{g-m}\}$, and

$$T_{farm} \;=\; max\{T_{dist},\; \frac{T_{mob}}{N},\; T_{coll}\}$$

where T_{gen} is the average service time of Generator Module (in our test ~ 1 second), T_{farm} is the average service time of the task farm computation mapped onto an ad-hoc network of PDAs, and L_{com}^{g-m} is the average process communication latency between Generator Module and the processes executed on PDA devices. The average service time of the task farm is the maximum between T_{mob}/N where N is the parallelism degree, T_{dist} and T_{coll} (i.e. the average service time of distribution and collection processes). In our experiments the two latter values are completely dominated by the communication latency between processes inside the ad-hoc network (L_{com}^{mob}). Multiple hops between the emulated sink node and the mobile ad-hoc network of PDAs can be necessary: we will show the case of 2 hops. As input tasks are 64 bytes in size, the latency for 2 hops covering a range of 100m can be estimated to ~ 0.23 ms (L_{com}^{g-m}). To maintain the QoS constraint (i.e. 3 seconds), when the pdaOperation is activated the run-time support chooses a lower quality of results and a proper parallelism degree. In this case using 1 MB systems, the communication latency L_{com}^{mob} between processes on mobile nodes (average distance of 50m) is ~ 1.36 sec. So:

$$N_{best} \;=\; \frac{90.04\ sec}{3\,\mathrm{sec}} \;= \sim 30$$

Data Size	**Average Computation Time (T_{mob})**
1 MB	90.04 seconds
2 MB	193.20 seconds
4 MB	414.28 seconds
8 MB	882.68 seconds

Table 1. Computation time on a PDA device.

with 30 PDAs we have an average service time of ~ 3 seconds, similar to the one obtained with clusterOperation but with a lower precision (i.e. from 32 MB matrices to 1 MB).

6 Conclusion

In this paper we introduced the ASSISTANT programming model for high-performance context-aware applications targeting Pervasive Mobile Grids. The programming model includes constructs to express multiple versions of a same module (i.e. operation) and a control logic to express the adaptivity w.r.t. the events (i.e. on_event construct).

We have introduced a test-case modeling a part of a flood emergency application. In this situation the application performs intensive forecasting models. It requires to adapt its behavior according to the context changes (e.g. status of connections). For this test-case we have developed a first implementation of the run-time support of our programming model. The main module features three different operations best-suitable for different resources. Experiments show how our application can reconfigure the current operation executed, maintaining a QoS behavior specified by the user but reducing the precision of the results.

References

[1] M. Aldinucci, M. Danelutto, P. Kilpatrick, Co-design of distributed systems using skeletons and autonomic management abstractions, *Proc. Euro-Par 2008*, Gran Canaria, Spain, 2008, 403-414.

[2] M. Baldauf, S. Dustdar, F. Rosenberg, A survey on context-aware systems, *Int. J. Ad Hoc Ubiquitous Computing*, 2(4), 2007, 263-277.

[3] C. Bertolli, R. Fantacci. G. Mencagli, D. Tarchi, M. Vanneschi, Next generation grids and wireless communication networks: towards a novel integrated approach, *Wireless Comm. and Mobile Computing*. 9(4), 2009, 445-467.

[4] T. Chaari, D. Ejigu, F. Laforest, V. M. Scuturici, A comprehensive approach to model and use context for adapting applications in pervasive environments, *Journal of System and Software*, 80(12), 2007, 1973-1992.

[5] M. Cole, Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Par. Comp.* 30(3), 2004, 389-406.

[6] R. Curtmola, C. N. Rotaru, BSMR: Byzantine-Resilient Secure Multicast Routing in Multi-hop Wireless Networks, *IEEE Trans. on Mobile Comp*. 8(4), 2009, 263-272.

[7] R.W. Hockney, C. R. Jesshope, *Parallel Computers: Architecture, Programming and Algorithms* (Bristol: Institute of Physics Publishing, 1981).

[8] D. J. Lillethun, D. Hilley, S. Horrigan, U. Ramachandran, MB++, An Integrated Architecture for Pervasive Computing and High-Performance Computing, *Proc. 13th IEEE Intl. Conf. on Embedded and Real-Time Computing Systems and Applications*, Daegu, Korea, 2007, 241-248.

[9] A. Mishra, V. Shrivastava, D. Agrawal, S. Banerjee, S. Ganguly, Distributed channel management in uncoordinated wireless environments, *Proc. 12th Intl. Conf. on Mobile Computing and Networking*, 2007, Los Angeles, USA, 170-181.

[10] B. D. Noble, M. Satyanarayanan, Experience with adaptive mobile applications in Odyssey, *Mob. Netw. Appl*. 4(4), 1999, 245-254.

[11] B. Plale, D. Gannon, J. Brotzge, K. Droegemeier, J. Kurose, D. McLaughlin, R. Wilhelmson, S. Graves, M. Ramamurthy, R. D. Clark, S. Yalda, D. A. Reed, E. Joseph, V. Chandrasekar, CASA and LEAD: Adaptive Cyberinfrastructure for Real-Time Multiscale Weather Forecasting, *Computer*, 39(11), 2006, 56-64.

[12] S. Helal and W. Mann and H. El-Zabadani and J. King and Y. Kaddoura and E. Janse, The Gator Tech Smart House: A Programmable Pervasive Space, *Computer*, 38(3), 2005, 50-60.

[13] T. Priol, M. Vanneschi, *From Grids To Service and Pervasive Computing* (Heidelberg: Springer, 2008).

[14] M. Román, C. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, K. Nahrstedt, A Middleware Infrastructure for Active Spaces, *IEEE Perv. Comp*. 1(4), 2002, 74-83.

[15] B. Syme, *Dynamically Linked Two-Dimensional/One-Dimensional Hydrodynamic Modelling Program for Rivers, Estuaries and Coastal Waters* (Spring Hill: WBM Oceanics, 1991).

[16] M. Vanneschi, The programming model of ASSIST, an environment for parallel and distributed portable applications, *Par. Comp*, 28(12), 2002, 1709-1732.

[17] M. Vanneschi, L. Veraldi, Dynamicity in distributed applications: issues, problems and the ASSIST approach, *Par. Comp*. 33(12), 2007, 822-845.

[18] J. R. Thompson, H. Refstrup Sorensen, H. Gavin, A. Refsgaard, Application of the coupled MIKE SHE/MIKE 11 modelling system to a lowland wet grassland in southeast England, *J. of Hydrology*, 293(1-4), 2004, 151-179.

[19] J. T. Feo, D. C. Cann, R. R. Oldehoeft, A report on the Sisal language project, *J. Parallel Distributed Computing*, 10(4), 1990, 349-366.