Expressing Adaptivity and Context Awareness in the ASSISTANT Programming Model

Carlo Bertolli, Daniele Buono, Gabriele Mencagli, and Marco Vanneschi

Dept. of Computer Science, University of Pisa, Largo Pontecorvo 3, Pisa I-56127 Italy
bertolli@di.unipi.it,
WWW home page: http://www.di.unipi.it/∼bertolli

Abstract. Pervasive Grid computing platforms are composed of a variety of fixed and mobile nodes, interconnected through multiple wireless and wired network technologies. Pervasive Grid Applications must adapt themselves to the state of their surrounding environment *(context)*, which includes the state of the resources on which they are executed. By focusing on a specific instance of emergency management application, we show how a complex high-performance problem can be solved according to multiple parallelization methodologies. We introduce the ASSISTANT programming model which allows programmers to express multiple versions of a same parallel module, each of them suitable for particular context situations. We show how the exemplified programs can be included in a single ASSISTANT parallel module and how their dynamic switching can be expressed. We provide experimental results demonstrating the effectiveness of the approach.

Key words: Adaptivity, Context Awareness, Parallel Programming, High-Performance Computing

1 Introduction

Pervasive Grid computing platforms [15] are composed of a variety of fixed and mobile nodes, interconnected through multiple wireless and wired network technologies. In these platforms the term *context* represents the state of logical and physical resources and of the surrounding environment (e.g. acquired by sensor data). An example of Pervasive Grid application is risk and emergency management [4]. 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 ones: these computations must be able to provide prompt and best-effort information to mobile users. In general these applications are composed of multiple software modules interconnected in some graph structure (e.g. work- or data-flow). In abstract terms each module is responsible for solving a specific sub-problem. Clearly, each problem can be solved according to different methods featuring different characteristics. They are suitable for different parallelization techniques and optimized for being mapped onto different resources. For instance, a method can be optimized for the parallelization according to task farm instead of data parallel. These computations can also

be different in the provided Quality of Service (QoS). In this paper we consider the term QoS as a set of metrics, which reflect the run-time behavior of a computation w.r.t. factors such as its memory occupation, its estimated performance (e.g. the average service time for a stream computation or the completion time of a single task) and the the quality of computed results. In this paper we show how multiple *versions* of a same parallel module can be introduced to target performance issues for different computing nodes, to face with the dynamic nature of pervasive grids and to meet dynamic user requests. We show that each version is best suited to be selected depending on conditions which are verified only at run-time (e.g. failures and user requests). This contribution is synthesized in the novel ASSISTANT programming model (ASSIST [18] with Adaptivity and Context-Awareness) which allows programmers to:

- express multiple versions of a same parallel module, exploiting the structured parallelism paradigm [6] (e.g. skeletons). This feature is inherited from the previous ASSIST parallel programming model [18];
- dynamically select which parallel version must be performed, in response to user-defined events or context changes (e.g. related to resource availability and sensor data). This feature can be expressed by exploiting performance models of structured parallel computations [19].

Thus, an ASSISTANT parallel module can be used to implement a fully autonomic computation.

We show a prototype implementation of ASSISTANT and we use it on a specific problem which is part of flood management application. A main module is involved in computing a flood forecast and it includes the resolution of a large number of tridiagonal linear systems. We show some different methods to solve tridiagonal systems, we discuss their properties and how these influence their parallelization scheme. We show experimental results of the execution of two parallel programs on best suited computing platforms. Thus, in this paper we show an example focusing on self-healing, self-configuring and self-optimization properties, leaving to future work the description of the remaining self- properties.

The paper is organized as follows: Sect. 2 discusses related works. Sect. 3 introduces the flood management application. Sect. 4 describes the different versions solving tridiagonal systems. Sect. 5 introduces the ASSISTANT programming model. Sect. 6 describes the implementation of the tridiagonal solver methods in the ASSISTANT model and it shows experimental results.

2 Related Work

Adaptivity has been introduced for mobile and pervasive applications by exploiting the concept of context [3]. Context definition includes environmental data, such as air temperature, the state of network links and processing nodes, and high-level information. Smart Space systems [16] mainly consist in providing context information to applications, which possibly operate on controllers

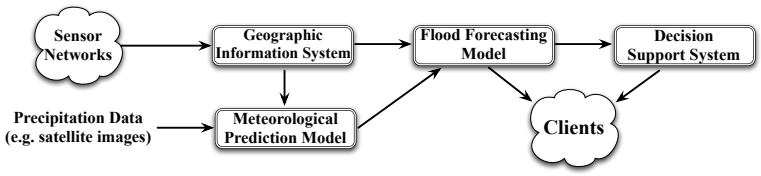

Fig. 1. Scheme of the flood management application.

to meet some user defined requirements. Some works focus on abstracting useful information from raw sensor data for adaptivity purposes. For instance, [5] exploits ontologies to model various context information, to reason, share and disseminate them.

General mobile applications must adapt themselves to the state of the context. For instance a mobile application can exploit optimized algorithms [12], protocols [7] or systems [2]. 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 [4]. For instance, in Odyssey [13] an operating system is responsible of monitoring resources. Significant changes in resource status are 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 [11]. Computations are defined as data stream flows of transformations, data fusions and feature extractions. 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 with 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 programs [19]. In [1] it is shown how hierarchical management can be defined for structured parallel component-based applications [6]. Adaptivity for service-oriented applications is also targeted in [14], but application adaptivity 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 a known cost model.

3 A Flood Management Application

We consider a schematic view of an application for fluvial flood management (see Fig. 1). During the "normal" situation several parameters are periodically monitored and acquired through sensors and possibly by other services (meteo

and GIS). For instance sensors can monitor the current value and the variation of flow level and surface height. A forecasting model is periodically applied for specific geographical areas and for widest combinations of these areas. An example is the TUFLOW [17] hydrodynamic model, which is based on mass and momentum partial differential equations to describe the flow variation at the surface. Their discrete resolution requires, for each time slice, the resolution of a very large number of linear (tri-diagonal) systems. The quality of the forecasts also depends on the size of the tridiagonal systems. There exist parallel techniques which allow us to obtain reasonable response times in scalable manner.

During the execution, the forecasting model may signal abnormal situations which could lead to a flood. Thus, *performance is a critical parameter* concerning the response time of the forecasting model *per se* and also concerning all graphic and visualization activities.

Consider an example of a critical situation: 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 mobile interconnection links which are geographically mapped onto a critical area. To manage the potential crisis in real time, *we can think to execute the forecasting model and visualization tools on a set of decentralized resources* whose interconnections are 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 in order to be able to cover different resource availability situations and dynamic QoS requirements. In this paper we show how multiple versions can be introduced for the flood emergency management application.

4 Defining Parallel Versions

We focus on the problem of solving tridiagonal linear systems of equations and we show how multiple parallel versions can be introduced each with different characteristics. There exist several resolution methods for *generic* linear systems: if the system is tridiagonal specialized techniques can be employed [9]. For performance modeling purposes, in this paper we focus on direct methods, which attempt to find an exact solution in a fixed, statically known, number of steps. Examples of direct approaches for tridiagonal systems are twisted factorization and cyclic reduction [9]. In this paper we are interested in defining multiple versions by exploiting different parallelization schemas of the same method: we focus on cyclic reduction methods because they can be easily generalized to banded and block tridiagonal systems [9]. In [10] two algorithms are introduced for solving tridiagonal systems of generic size N. For brevity, we avoid to introduce the mathematical formulations of these algorithms: interested readers can refer to [10].

First Algorithm This algorithm includes two main parts. The first part (denoted by *transformation*) transforms in $q-1$ steps ($q = log_2(N+1)$) the input system. At each step l we consider all rows i such as $i \ mod \ 2^l = 0$. We solve:

$$\begin{aligned} a_i^l &= \alpha_i a_{i-2^{l-1}}^{l-1} & c_i^l &= \gamma_i c_{i+2^{l-1}}^{l-1} \\ b_i^l &= b_i^{l-1} + \alpha_i c_{i-2^{l-1}}^{l-1} + \gamma_i & k_i^l &= k_i^{l-1} + \alpha_i k_{i-2^{l-1}}^{l-1} + \gamma_i k_{i+2^{l-1}}^{l-1} \\ \alpha_i &= -a_i^{l-1}/b_{i-2^{l-1}}^{l-1} & \gamma_i &= -c_i^{l-1}/b_{i+2^{l-1}}^{l-1} \end{aligned} \tag{1}$$

where a_i, b_i, c_i and k_i are the diagonal coefficients and the constant term of the i-th row. The superscripts denote the computational step at which their values are taken. α and γ are used in this notation to make equation reading easier. The stencil, i.e. the functional dependencies between successively computed values, refers the same element i and two neighbors: rows $i - 2^{l-1}$ and $i + 2^{l-1}$.

The second part of this algorithm is denoted *resolution*. We compute the solutions of the system, according to a fill-in procedure. It includes q steps for $l = q, q-1, \ldots, 1$. At each step l we consider all rows i for which $i \ mod \ 2^l = 0$

$$x_i = (k_i^{l-1} - a_i^{l-1} x_{i-2^{l-1}} - c_i^{l-1} x_{i+2^{l-1}})/b_i^{l-1} \tag{2}$$

In this case we do not need multiple x values for each computation step. The stencil is the same of the first part of the algorithm.

Second Algorithm The second algorithm includes two parts as the previous one. The first part includes q steps. Unlike the first algorithm, we solve the same equations (1) but for *all* rows at each step. The second part includes only a single step in which we directly get all the solutions of the system. These are computed in the following way: $x_i = k_i^q / b_i^q$. Notice that we only need the last values of the transformed system, instead of all the ones computed in the first part.

Discussion We discuss the features of each algorithm to define the best parallelization schemas. *In this paper we focus on context events including the state of the used computing resources and their associated performance.* We avoid to consider environmental data (e.g. sensor data) influencing the version selection policy (demanded to future work).

The performance features of the described algorithms can be characterized as following:

- **Number of steps**: the first algorithm performs $q-1 = log_2(N-1)-1$ steps during the transformation part and $q = log_2(N-1)$ in the resolution one. The second algorithm performs less steps: $q = log_2(N-1)$ transformation steps and only one resolution step.
- **Number of Operations**: the first algorithm performs a lower number of operations in the first part w.r.t. the second algorithm. This because the second algorithm applies, at each step, the equations (1) to all system elements, instead of only a subset of them. The second part of both algorithms involves the same number of operations.

– **Number of Functional Dependencies** The first algorithm includes a lower number of functional dependencies because, at each step of the transformation part, equations 1 are solved only for a subset of elements.

In our application we need to solve a stream of tridiagonal systems, i.e. a possibly unlimited sequence of systems. We need to consider the parallel efficiency on the single system and on the stream of systems. Looking at the algorithms above we can think to use two kinds of parallel structures:

– **Task Farm**: the systems (tasks) belonging to the input stream are scheduled w.r.t. several replicated workers according to a load balancing strategy, each worker executing the sequential algorithm. An output stream of results is produced. As known, this parallelism paradigm does not decrease the processing latency of a single element of the stream, but it decreases the service time (increases the throughput), provided that the stream interarrival time is sensibly less than the sequential processing time (stream processing situation in the true meaning).
– **Data Parallel**: each tridiagonal system is partitioned (scattered) onto several replicated workers, each one performing the sequential algorithm for its respective partition. In the considered algorithms, workers cooperate during each step according to a proper communication stencil. The whole result is obtained by gathering the partial results. With respect to the farm structure, this parallelism paradigm works both in a stream processing situation, and when only a single system has to be processed (i.e. equivalently, when the stream interarrival time is greater than the sequential processing time for a single task). Moreover, it is able to decrease the processing latency of a single tridiagonal system and the memory size per node. In a stream situation, the disadvantage of a stencil-based data parallel structure, w.r.t. the farm paradigm, is a potential load unbalance and a more critical impact of the communication/computation time ratio, thus in general a greater service time.

Two structuring modalities of the whole computation have to be distinguished: an acyclic graph structure or a cyclic one. In the former case, a pipeline-like effect is present, provided that a real stream processing situation occurs. In the latter, the overall computation is a client-server schema. Each client sends the input data to the tridiagonal solver module (i.e. the server) and it waits for the corresponding results. To increase the performance of each client we can parallelize the server with a proper parallelism degree:

– in a task farm structure it is equal to the number of clients;
– in a data parallel structure it is independent of the number of clients and can be obtained by the proper cost model of the parallel structure. Moreover, the reduced latency time contributes to decrease the server response time, thus the client service time.

All the described situations (stream vs single element processing, acyclic graph vs client-server structure) can be taken into account in an adaptive and context-

aware computation. The farm and the data- parallel paradigms are able to optimize specific situations. In general it may be convenient, or necessary, to switch from one structure to another one dynamically, thus to switch from a version of the computation implemented according to a parallelism paradigm to another version, implemented according to the other parallelism paradigm: this feature characterizes our approach to high-performance adaptive and context-aware application design.

We have seen that the second algorithm performs more operations than the first one (but less steps), but also more communications. These can be buffered and, provided that their support is efficient, the second algorithm can be parallelized according to the data parallel structure. Communication efficiency characterizes multicores: communications between cores are implemented as accesses to shared variables and the computation can take advantage of the hardware caching support. Thus, we implement this version on a multicore architecture (see below).

The first algorithm minimizes the number of operations performed in the whole computation. Thus, it seems reasonable to: (a) parallelize it according to the task farm model, which has not the strong requirements, in terms of communication efficiency, of the data parallel; (b) implement it for both cluster and multicore architectures. We show experimental results for both versions and we discuss how the best version is dynamically selected according to specific context situations.

We show the data parallel program for the second algorithm in the ASSIST syntax. For brevity, we avoid to show the program of the task farm version.

4.1 The ASSIST Model

ASSIST [18] 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 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.

4.2 Parallel Programs in ASSIST

We implement an ASSIST parmod for the data parallel version using the second algorithm (see Fig. 4.2). The parmod *TSM-DP* receives a stream of tridiagonal

```
1  parmod TSM-DP(input_stream syst_row input_syst[N] output_stream
       solutions sols) {
2    topology array [i:N] vp;
3    attribute syst_row computing_syst[2][N] scatter S[][*i] onto VP[i];
4
5    do input_section {
6      guard1: on , , input_syst {
7        distribution input_syst[*s] scatter to comput_syst[0][s];
8      }
9    } while (true)
10
11   virtual_processors {
12     solve_system (in guard1 out sols) {
13       VP i {
14         for(l = 1; l <= q; l++)
15           transform(i, computing_syst[i][l-1], computing_syst[i-pow(2,l
                 -1)],computing_syst[i+pow(2,l-1)],sols[i]);
16
17         solve(i, comput_syst[i][], sols[i]);
18       }
19     }
20   }
21
22   output_section {
23     collects sols from ALL vp[i];
24   }
25 }
```

Fig. 2. Data parallel program based on the second cyclic reduction algorithm.

systems (*syst_row* data structure) as input tasks (line 1). For each system it computes the correct solution (*solutions* data structure). The topology command (line 2) gives integer numbers (from 1 to N) as names of virtual processors. Virtual processors are assigned a single system row on which they apply the algorithm. In the implementation, multiple virtual processors are mapped onto a set of implementation processes. At line 3 an attribute (a ParMod variable) is used to store two successive system values during its transformation and it is scattered amongst the virtual processors. The input section (line 5) is fired when a system is received (line 6) on the input stream. The system is scattered onto the first position of the attribute (line 7). In the virtual_processors section (line 11) each VP_i in parallel: (a) transforms the input system in q steps (line 15); (b) computes the results (line 17). In the output_section we gather all computed results (*from ALL* keyword), which are automatically delivered onto the output stream.

4.3 Experiments

We have tested the parallel efficiency of the different tridiagonal solver versions on a emulation of a pervasive grid. The aim of this section is to experimentally

show that different versions can be used to target different context situations, related to the state of the used computing resources (e.g. their availability) and on their performance. The experiments are performed on a prototype of ASSISTANT: we avoid to show the reconfiguration costs (i.e. version selection) because it is out of the scope of this paper. The pervasive grid is emulated by the following nodes:

– a centralized server, emulated with a cluster architecture. The cluster is composed by 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. We map the task farm version onto this platform;
– an interface node between the cluster and the mobile distributed platform of PDA nodes and sensor devices. The interface node is emulated with an Intel E5420 Dual Quad Core multicore processor, featuring 8 cores of 2.50 GHz, 12 MB L2 Cache and 8 GB of main memory. Both data parallel and farm versions are mapped onto this architecture.

Fig. 3, 4 and 5 show the results in terms of service time and scalability. We define the service time as the time passing between the consuming of two successive systems from the input stream (not their complete resolution but only their consuming). Scalability can be defined as the ratio between the sequential computation time (parallelism equal to 1) and the parallel one: it represents the quality of parallelization of the module. Notice that the cluster service time is higher than the multicore one because of the sensible difference between the processing power of their single nodes (800 MHz versus 2.5 GHz). For comparable processing powers, the cluster would provide higher scalability and parallelism degrees. As discussed at the end of Sect. 4, for acyclic graph application structures:

– the farm version is effective only when the computation operates on a stream processing situation: in this case it performs better than the data-parallel solution;

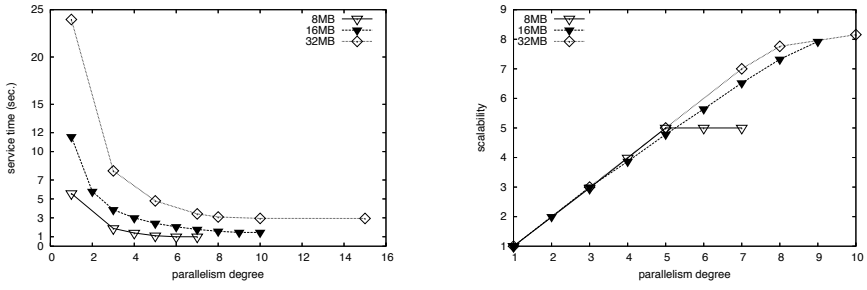

Fig. 3. Experimental results of the cluster task farm version (first algorithm): service time (left) and scalability (right).

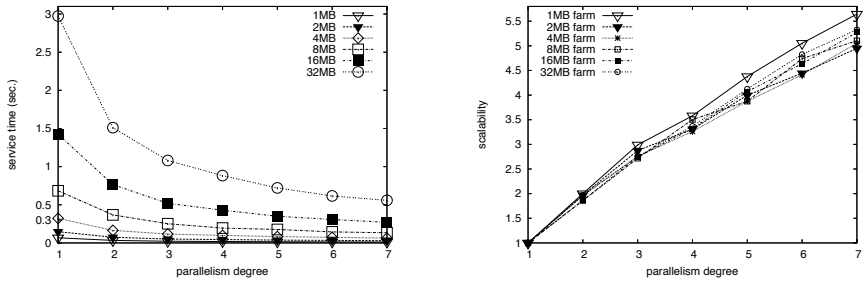

Fig. 4. Experimental results of the multicore task farm version (first algorithm): service time (left) and scalability (right).

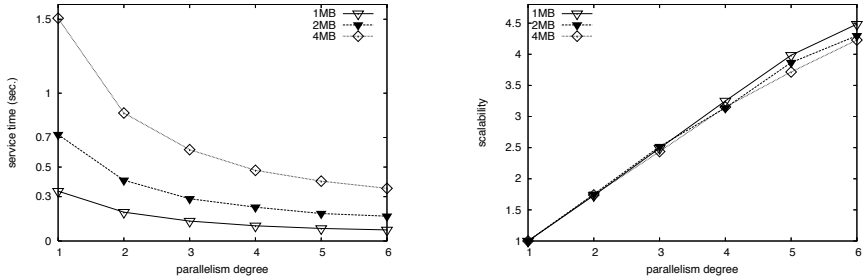

Fig. 5. Experimental results of the multicore data parallel version (second algorithm): service time (left) and scalability (right).

– the data parallel version has to be adopted when the computation operates on a single tridiagonal system (i.e. too large interarrival time).

For client-server cyclic graph application structures, both versions are potentially feasible: one or the other will be selected dynamically according the performance comparison between the farm and the data parallel version, i.e. according to the optimal parallelism degree of the two versions.

From this discussion we can conclude that pervasive grid applications must be programmed as multiple parallel modules, each provided in multiple versions. Moreover, *the programmer needs to express the conditions (or events) under which each version is dynamically selected*, according to the available resources and user needs. This can be done by specifying different parallel programs and by introducing a policy for the dynamic selection of the best version. The actual implementation of version selection can be automatized in the following points:

– low-level version switching: this includes the re-direction of input data streams between the versions;

– data consistency: the channel re-direction must guarantee that no input elements are lost or re-ordered.

In the next section we introduce an high-level programming model in which all these actions are automatized, while the application programmer just focuses on the abstract events inducing a version switching.

5 The ASSISTANT Programming Model

We introduce the novel ASSISTANT programming model for high performance pervasive applications with adaptive and context-aware behaviors. With ASSISTANT we target application adaptivity by allowing programmers to express how the computation evolves reacting to specified events. We enable this kind of expressivity with a new ParMod construct. We can characterize three main logics (Fig.6 (left)) that can be used to describe the semantic and the behavior of a ParMod:

– **Functional** logic: this includes all the versions solving the same problem in the ASSIST syntax. Functional logics of different ASSISTANT ParMods communicate by means of typed data streams.
– **Control** logic: this includes the adaptivity strategies, i.e. the *reconfiguration* actions performed to adapt the ParMod behavior in response to specified events. For instance, the control logic can select the best version between multiple ones, according to specific cost models. The programmer is provided with high-level constructs to directly express the control logic with the corresponding adaptivity strategies. Control logics of different application ParMods can interact by means of *control events*.
– **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 service time of a ParMod). These *context events* can be provided by the run-time support of the programming

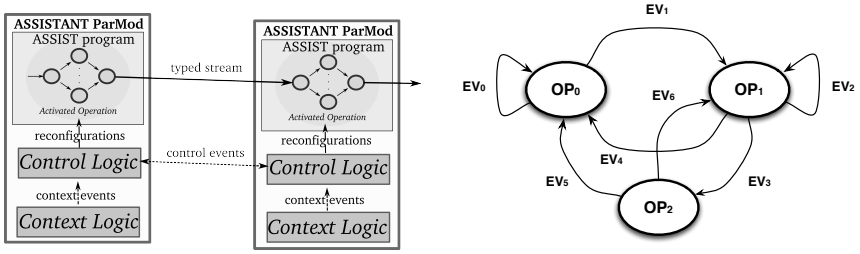

Fig. 6. Example of ASSISTANT ParMods (left) and of event-operation graph (right).

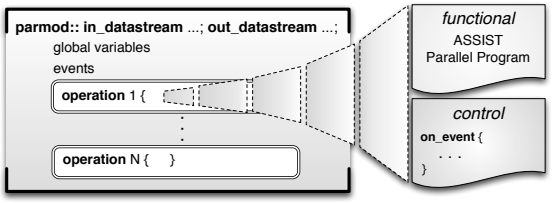

Fig. 7. Syntactic view of a ParMod.

model, or in other cases primitive *context interfaces* (e.g. failure detectors) which communicate with the application modules by context events.

The different versions of a same ParMod are expressed by means of the *operation* construct. Each ParMod can include multiple operations (see Fig. 7), all solving the same problem according to different algorithms and parallel structures. All operations of the same ParMod must feature the same input and output interfaces, in terms of streams. Each operation includes its own part of functional, control and context logic of the ParMod in which it is defined. That is, each operation features its own parallel algorithm, but also its own control and context logics. Notice that operations are not merely alternative sections of code inside a module definition: they are the adaptation and deployment units of the versions of the same Assistant module. For this reason, a new suitable construct is required to achieve our goals.

Syntactically, the ParMod 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. Events may be context ones, whose monitoring can be provided by context interfaces, or control events obtained from the control logic of other ParMods. Semantically, 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, possibly deploying it on dynamically discovered resources. During the execution the context logic of a ParMod, or the control logic of other modules, can notify one or more events. The control logic exploits a mapping between these events and reconfiguration actions, 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 [19]). The control logic of an ASSISTANT ParMod can be described as a graph (see Fig. 6(b)(right)): nodes are operations of the ParMod and arcs are events (or their combinations with logical expressions on the actual ParMod state). Semantically a ParMod control logic is a sequential automaton: this is done to avoid nondeterministic behaviors. In the case of concurrent events we serialize and manage them according to a priority defined in the control logic itself (i.e. their definition sequence). In the example, the initial operation is OP_0. If the event EV_0 occurs, we continue executing OP_0 but we modify some aspects of its implementation (e.g. 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 fired by event EV_1. In this case the programmer specifies that if we are executing OP_0 and event EV_1 occurs, we stop executing OP_0 and we start OP_1. This switching can include pre- and post- elaborations: for instance, 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.

Reconfigurations can be performed in the case: (a) some pre-determined events happen and/or (b) some predicates on the module state are satisfied. That is, the control logic of a ParMod is stateful. This behavior is expressed in each operation of a ParMod by means of the *on_event* construct. Syntactically, the programmer makes use of nondeterministic clauses which general structure is described as shown in Fig. 8:
If the *event_combination* logical expression is satisfied, the corresponding recon-

```
event_combination:
  do   <reconfiguration code>
  enddo
```

Fig. 8. General structure of the on_event construct.

figuration code is executed. Programmers are also provided with a *parallelism* construct, to specify a modification of the parallelism degree of the current operation (i.e. a run-time system reconfiguration).

6 Programming Adaptivity for the Flood Application in ASSISTANT

We show how to encapsulate the different versions, to solve tridiagonal systems, in a single ParMod.The flood management application is composed of the following ParMods:

- **Generator**: this module emulates all the application phases preceding the flood forecasting model. It generates a stream of double precision floating point data related to the conditions of each point in the river [17].
- **Tridiagonal Solver Module** (TSM): this module implements the forecasting model (see Sect. 4), which is applied to each input stream element received by the Generator. For each input data, it generates and solves four tridiagonal systems (e.g. see [17]). The TSM includes three different operations.
- **Visualization**: this module implements the post-processing activities, visualizing forecast results on the user's display.

In this program we consider three different operations for the TSM: the first one is *clusterOperation* which is the task-farm mapped onto the cluster architecture as described in Sect. 4. The second one is *interfaceNodeFarm*) which is the

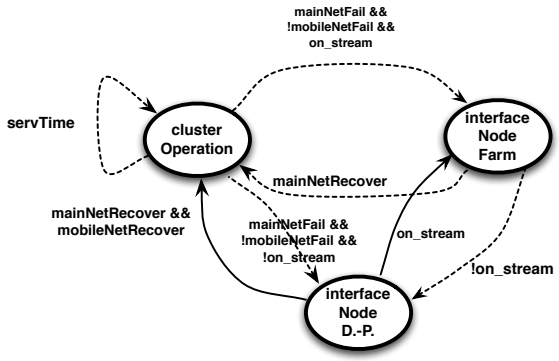

Fig. 9. Event-Operation graph of the parmod TSM. Bold arrows are implemented in Fig. 6.

```
parmod TSM(..) {
 operation interfaceNodeD.-P. {
   //Parallel Computation in an ASSIST-like fashion:
   see Section 3..

   //Management section of this operation:
   on_event {
    mainNetRecover && mobileNetRecover:
      do
        notify(Generator_Module, con_fail);
        notify(Client_Module, con_fail);
        //Stop the operation consistently:
        this.stop();
        //Activation of the new operation:
        clusterOperation.start();
      enddo
    on_stream():
      do
        ...
        interfaceNodeFarm.start();
      enddo
   }
 }
...
```

Fig. 10. Control part of *interfaceNodeD.-P.* operation inside the parmod TSM definition.

task farm version executed on interface multicore nodes. The third operation is *interfaceNodeD.-P-* which is the data parallel version executed on interface multicore nodes. As the TSM functional logic has been described in Sect. 4, we are interested in the control logic. This is responsible of deciding which context changes have to be monitored and which ones cause a reconfiguration. The considered context changes are:

– a network event from the TSM context logic related to the current status of network connections (e.g. their availability or presence of high-latency links).

We have considered only two disconnection events: *mainNetFail* and *mobileNetFail* which provide a boolean information related to the connection capability (based on current latency and connection status) between the cluster and the considered interface node (the former) and the interface node and the user's PDAs (the latter);
– the average interarrival time to the ParMod TSM is lower than a maximum threshold. This event is denoted with *on_stream*.

Fig. 9 shows the event-operation graph of the TSM: bold arrows are those expressed in the *interfaceNodeD.-P.* control part. Dotted arrows are expressed in the other two operations. Fig. 6 shows the corresponding *on_event* section of the operation inside the TSM definition, implementing its adaptive behavior when the *interfaceNodeD.-P.* is executed. This *on_event* instance describes two different conditions: while executing *interfaceNodeD.-P.* the *mainNetRecover* and *mobileNetRecover* events can be received. We choose to execute the forecasting model on the cluster, because it enables higher parallelism degree than the multicore and, consequently, lower service times. If the *on_stream* event is received, the input stream interarrival time is now lower than the TSM service time. In this case we can switch to the *interfaceNodeFarm* operation, which provides better on stream scalability (see Sect. 4). We also need to notify the generator and client modules of this re-configuration (i.e. the *notify* function).

7 Conclusions

In this paper we have shown how adaptivity for pervasive grid applications can be defined by exploiting multiple versions for the same application module. We have exemplified our approach on the specific problem of solving tridiagonal systems of linear equations, introducing two resolution algorithms and parallelizing them. The two algorithms are shown to be best suited for being executed on a cluster architecture and on a multicore one. The experimental results show that: the cluster version and the interface node version, as well as the respective farm and data parallel schemas, have clear pros and cons that can drive the selection of the best adaptation strategy at run time. As an example, the cluster version has to be preferred to the interface node one if the network status provides a reasonable communication latency between the cluster and the mobile users. We have introduced the novel ASSISTANT programming model, providing constructs to express multiple versions of a same parallel module and to adapt its execution by dynamically selecting the best one. We have implemented a flood forecasting module exploiting the ParMod construct, including the two resolution algorithms and their dynamic selection policy.

References

1. Aldinucci, M., Danelutto, M., Kilpatrick, P.: Co-design of distributed systems using skeletons and autonomic management abstractions. In: Csar, E., Alexander, M.,

Streit, A., Trff, J.L., Crin, C., Knpfer, A., Kranzlmller, D., Jha, S. (eds.) Euro-Par 2008. LNCS, vol. 5415, pp. 403-414. Springer, Heidelberg (2009).
2. Balasubramanian, A., Levine, B.N., Venkataramani, A.: Enhancing interactive web applications in hybrid networks. In: 14th ACM International Conference on Mobile Computing and Networking, pp. 70-80. ACM, New York (2008).
3. Baldauf, M., Dustdar, S., Rosenberg, F.: A survey on context-aware systems. Int. J. Ad Hoc Ubiquitous Computing. 2, 263-277 (2007).
4. Bertolli, C., Fantacci. R., Mencagli, G., Tarchi, D., Vanneschi, M.: Next generation grids and wireless communication networks: towards a novel integrated approach. Wireless Comm. and Mobile Computing. 9, 445-467 (2009).
5. Chaari, T., Ejigu, D., Laforest, F., Scuturici, V.M.: A comprehensive approach to model and use context for adapting applications in pervasive environments. Journal of Syst. Softw. 80, 1973–1992 (2007).
6. Cole, M.: Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. Par. Comp. 30, 389-406 (2004).
7. Curtmola, R., Rotaru, C.N.: BSMR: Byzantine-Resilient Secure Multicast Routing in Multi-hop Wireless Networks. IEEE Trans. on Mobile Comp. 8, 263-272 (2009).
8. Danelutto, M.: QoS in Parallel Programming through Application Managers. In: 13th Euromicro Conf. on Parallel, Distributed and Network-Based Processing, pp. 282-289. IEEE Press, Washington (2005).
9. Duff, I.S., Van der Vorst, H.A.: Developments and trends in the parallel solution of linear systems. Par. Comp. 25, 1931-1970 (1999).
10. Hockney, R.W., Jesshope, C.R.: Parallel Computers: Architecture, Programming and Algorithms. Institute of Physics Publishing, Bristol (1981)
11. Lillethun, D.J., Hilley, D., Horrigan, S., Ramachandran, U.: MB++: An Integrated Architecture for Pervasive Computing and High-Performance Computing. In: 13th IEEE Intl. Conf. on Embedded and Real-Time Computing Systems and Applications, pp. 241-248. IEEE Press, Washington (2007).
12. Mishra, A., Shrivastava, V., Agrawal, D., Banerjee, S., Ganguly, S.: Distributed channel management in uncoordinated wireless environments. In: 12th Intl. Conf. on Mobile Computing and Networkin, pp. 170-181. ACM, Los Angeles (2006).
13. Noble, B.D., Satyanarayanan, M.: Experience with adaptive mobile applications in Odyssey. Mob. Netw. Appl. 4, 245-254 (1999).
14. Plale, B., Gannon, D., Brotzge, J., Droegemeier, K., Kurose, J., McLaughlin, D., Wilhelmson, R., Graves, S., Ramamurthy, M., Clark, R.D., Yalda, S., Reed, D.A., Joseph, E., Chandrasekar, V.: CASA and LEAD: Adaptive Cyberinfrastructure for Real-Time Multiscale Weather Forecasting. Computer. 39, 56-64 (2006).
15. Priol, T., Vanneschi, M.: From Grids To Service and Pervasive Computing. Springer, Heidelberg (2008).
16. Román, M., Hess, C., Cerqueira, R., Ranganathan, A., Campbell, R.H., Nahrstedt, K.: A Middleware Infrastructure for Active Spaces. IEEE Perv. Comp. 1, 74-83 (2002).
17. Syme, B.: Dynamically Linked Two-Dimensional/One-Dimensional Hydrodynamic Modelling Program for Rivers, Estuaries and Coastal Waters. WBM Oceanics (Aus) (1991).
18. Vanneschi, M.: The programming model of ASSIST, an environment for parallel and distributed portable applications. Par. Comp. 28, 1709-1732 (2002).
19. Vanneschi, M., Veraldi, L.: Dynamicity in distributed applications: issues, problems and the ASSIST approach. Par. Comp. 33, 822-845 (2007).