2009 10th International Symposium on Pervasive Systems, Algorithms, and Networks

Adaptivity in Risk and Emergency Management Applications on Pervasive Grids

Carlo Bertolli, Gabriele Mencagli and Marco Vanneschi
Department of Computer Science, University of Pisa
Largo Bruno Pontecorvo, 3, Pisa, 56125 Pisa, Italy
{*bertolli,mencagli,vannesch*}*@di.unipi.it*

***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 which includes environmental data (e.g. collected from sensors) and the state of the used resources (e.g. network or node states). Adaptation is especially important if we consider complex High-Performance Pervasive Grid applications, such as intelligent transportation and emergency management. In this paper we investigate how to define adaptivity for complex Pervasive Grid applications by providing multiple versions of application parallel modules. The versions are defined by exploiting different sequential algorithms and parallelization techniques. We introduce performance analysis tools for versions, which allow us to define specific selection policies of the best version to be executed, depending on the context. We show how each version is best suited to be executed on two application scenarios, also by means of experiments. To synthesize the contributions of this paper we introduce the ASSISTANT programming model, for adaptive Pervasive Grid applications.**

***Keywords*-Pervasive Grid; High-Performance; Risk and Emergency Management; Adaptivity; Programming Models**

I. Introduction

Pervasive Grid computing platforms [1] 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. Complex Pervasive Grid 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. These requirements can be satisfied if applications can adapt themselves to the dynamic conditions of the platform. That is, *applications must adapt themselves to the context in which they are executed*. We can express application adaptivity in two main ways:

- provide applications with a dynamic support (e.g. [2]), which allows them to modify their run-time support to face changes in the availability of the used resources;
- provide multiple versions of same application modules, to enable their mapping onto heterogeneous resources. Versions can be based on different sequential algorithms and parallelization schemas.

In this paper we mainly focus on the second kind of adaptivity. In a Pervasive Grid, a parallel computation can be mapped onto several kinds of computing nodes. For instance it can be mapped onto: (a) a centralized parallel server; (b) a wireless network of users' PDAs; (c) a network of wireless interface nodes, possibly supported by multicore architectures. Each platform requires a different version of the same module, exploiting different sequential algorithms and parallelization techniques. A specific version can be optimized for a specific computing support and, in more general terms, context situation. *We show how performance analysis of structured parallel programming, which is based on common and well-studied parallelism forms (e.g. divide-and-conquer and parallel sort), is exploited to select the best version for each context situation*. The dynamic selection is done to face with catastrophic events affecting the platform (e.g. a link or service failure), in response to user requests and to optimize performance aspects.

Our research work in the Italian FIRB In.Sy.Eme. (Integrated Systems For Emergency) Project focuses on Emergency management applications [3]. These represent one of the most interesting test-case for Pervasive Grids as they include two main kinds of computations: (i) off-line computations to support environmental monitoring and emergency forecasting; (ii) on-line computations (real-time in some cases), to support the emergency management when it is act. In this paper we focus on flood management applications, and we show multiple parallel versions of a same flood forecasting module. We describe how the version selection policy can be defined for two main application scenarios by using proper performance models. Then we introduce the novel ASSISTANT programming model which inherits from our experience on the ASSIST [4] structured parallel programming. ASSISTANT includes full application adaptivity, w.r.t. the above two points.

The paper is organized as follows: in Sect. II we discuss related works. In Sect. III we briefly introduce the flood management application and we describe two versions of a flood forecasting model. In Sect. IV we define two application scenarios and we characterize their performance analysis. In Sect. V we describe the main features of the ASSISTANT programming model. Sect. VI derives conclusions and discusses future work.

978-0-7695-3908-9/09 $26.00 © 2009 IEEE
DOI 10.1109/I-SPAN.2009.92

II. Related Work

Our research work is based on previous works on adaptivity for parallel and distributed High-Performance applications. Adaptivity is mainly intended as the dynamic reconfiguration of the parallel implementation of programs (e.g. increasing or decreasing the parallelism degree [2]). Structured parallel programming has lead to optimized solutions [2], w.r.t. solutions based on general parallel programming models (e.g. MPI and OpenMP).

In our research work we extend the notion of adaptivity to include also the possibility of switching between different versions of the same computation. This idea has been studied also in the Grid Component Model (GCM): in GCM, components can be nested in arbitrary ways and sub-components can be dynamically replaced also to perform the switching between different versions (e.g. see [5]). The selection policy can be programmed in specialized management sub-components (i.e. the *application managers*). In GCM adaptivity focuses on the following issue: if a single performance-related Quality of Service (QoS) is specified for a whole GCM application, how this can be recursively translated in specific QoS for the sub-components? [6] presents a solution based on algorithmic skeletons [7]: the recursive translation can be defined if components are developed according to the skeleton paradigm, by exploiting their performance models. This recursive translation is reflected in the hierarchical design of the network of application managers.

Our research work can be thought as an extension to these works. Anyway, we avoid to tie our application control logic to a specific design pattern (e.g. hierarchical). We present our research framework to define version selection policies of versions. We demand at the implementation level the actual development of an optimized network of managers: this will exploit different structures (e.g. ring or centralized ones), depending on the application- and platform-specific features.

Adaptivity has been introduced also for mobile and pervasive applications, by exploiting the concept of context [8]. The context includes environmental data such as air temperature and the network and node states. Smart Space systems [9] mainly consist in providing context information to applications, which possibly operate on controllers to meet user requirements. Some works focus on abstracting useful information from raw sensor data for adaptivity purposes, possibly by means of ontologies [10].

Concerning adaptivity, a mobile application can exploit optimized algorithms, protocols or systems [11]. In this vision, it is the run-time support (e.g. the used protocol) which is in charge of adapting its behavior. Adaptivity can be also defined at the application level [3]. For instance, in Odyssey [12] adaptivity is expressed in terms of the choice of the services from which an application is composed.

In [13] High-performance application are introduced in the context of pervasive computing, for stream-based transformations, fusion and feature extractions. Unlike our work, these computations are mapped onto centralized servers.

In this paper we propose to attack the complexity of definition of adaptivity of Pervasive Grid applications by extending the concepts developed in the high-performance computing research field.

III. 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 of flow level and surface height. A forecasting

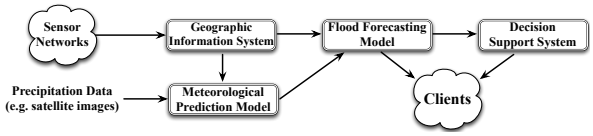

Figure 1. Scheme of the flood management application.

model (e.g. an hydrodynamic model as in[14]) is periodically applied for specific geographical areas and for widest combinations of these areas. The main computational cost is represented by the resolution of partial differential equations for each river discretization point, corresponding to resolving a large set of linear tridiagonal systems.

During the execution the forecasting model may signal abnormal situations that could lead to a flood. Thus, *execution performance is a critical parameter. In this paper we describe multiple application scenarios and we characterize the best parallel structure w.r.t. performance parameters.*

We consider two resolution algorithms for solving tridiagonal linear systems, based on cyclic reduction method [15]. Each algorithm is best suited for a different parallelization scheme which is characterized by a different performance model. We only give a brief description of the two algorithms and their parallelization: the interested reader can refer to [15] for a full description.

First Algorithm: This algorithm includes two main parts. The first part (denoted by *transformation part*) transforms the input system in $q-1$ steps ($q = log_2(N+1)$), where N is the system size. At each step l we consider all rows i such as $i \ mod \ 2^(l-1) = 0$, for which we solve a set of equations (see [15] for the equation definition). This part features functional dependencies (i.e. stencil) between equation resolutions: for each considered row i we need the previously computed values of rows $i - 2^{l-1}$ and $i + 2^{l-1}$. The second part of this algorithm is denoted *resolution part* in which we compute in q steps the solutions of the linear system, according to a fill-in procedure.

This algorithm minimizes (w.r.t. the second one, see below) the number of operations and communications. Con-

sequently, it features fine grain and it is best suited for a parallelization according to a task farm scheme. This scheme is applied to a stream of input data (the river points), which an emitter (E) schedules to a set of replicated workers (W), according to a load balancing strategy. Each worker, for each input data, generates and solves four tridiagonal systems, producing the X and Y components. Results are sent to a collector (C) which produces a stream of output results. We show the task farm service time (T_{farm}) and latency (L_{farm}). T_{farm} represents the average time between the beginning of the elaboration of two successive input elements. L_{farm} represents the time necessary to complete the elaboration of a single input element. If we denote with T_E, T_W and T_C respectively the service times of the emitter, worker and collector, and with n the parallelism degree (i.e. number of workers) we can define $T_{farm} = max(T_E, T_W/n, T_C)$ and $L_{farm} = T_E + T_W + T_C$. So the task-farm scheme is able to reduce the service time, but the latency is equal or higher than in the sequential case.

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 of the first algorithm but for *all* system rows at each steps. The second part includes only a single step in which we directly get all the solutions of the system.

Compared with the previous algorithm, the second one features coarser grain. Thus, it is best suited for parallelization according to a data parallel scheme: each generated tridiagonal system is scattered (S) onto several replicated workers (W), each one performing the sequential algorithm on its assigned partition. Workers cooperate during each step according to a proper communication stencil. The whole result is obtained by gathering (G) the partial results of each worker. Unlike the task farm, this paradigm works both in a stream processing situation and when only a single system at time has to be processed. In other words, it reduces the parallel efficiency also when the stream interarrival time is greater than the sequential processing time. This is obtained because it is able to decrease the processing latency of a single tridiagonal system and the memory size per node. The disadvantage of this data parallel version is that it can feature load unbalancing. Also for the data parallel we consider the service time T_{dp} and request latency L_{dp}. We denote with $T_{scatter}$, $T_{parcomp}$ and T_{gather} respectively the service times of the scatter, worker and gather modules. We can define $T_{dp} = max\ (T_{scatter}, T_{parcomp}, T_{gather})$, and $L_{dp} = T_{scatter} + T_{parcomp} + T_{gather}$. For brevity, we avoid to show how $T_{parcomp}$ can be defined.

IV. Application Scenarios

We consider two application scenarios, each characterizing a run-time configuration scheme of the flood application. The definition of the scenarios is given independently of the set of available computing nodes. Consequently, the introduced performance models characterize each scenario independently of the mapping between application modules and available nodes. To concretize our research contributions, we also perform experiments on actual computing nodes: a cluster platform and a multicore interface node.

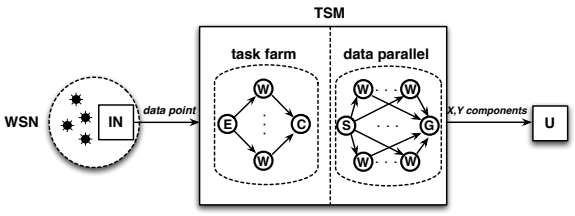

Figure 2. Representation of Scenario 1.

A. Scenario 1: Forecasting a Large River Area

In the first scenario users request for flood forecasting over a large river area. A Wireless Sensor Network (WSN) monitors the area and it provides input data for a forecasting model. As described above, input data reflect the discretization of the area in *points*. We assume that a single module *IN* collects all the sensed data from the WSN (see Fig. 2). *IN* sends the input data to a forecasting module (Tridiagonal Solver Module or *TSM*). TSM sends the forecast results to the users in real-time or it can perform some buffering. The performance modeling of this scenario depends on two quantities: (i) the input data interarrival time to the TSM (T_A); (ii) the parallel performance of the TSM module (T_{TSM}). Depending on the value of T_A we need to minimize either the TSM service time, or its request latency. Clearly, both values depends on its parallelization scheme. Notice that, for simplicity, we consider that communication and computation cannot be overlapped and T_{IN} includes also the communication latency of input data.

- **Case** $T_A < T_{TSM}$: the WSN is not the bottleneck of the system. The system performance depends on the service time of the forecasting module T_{TSM}. We choose the task farm parallel structure as it minimizes the forecasting module *service time*;
- **Case** $T_A >> T_{TSM}$: T_A is the bottleneck of the system and we can only minimize the single system resolution time. We choose the data parallel solution as it minimizes the *service latency*.

In a Pervasive Grid these two situations can be dynamically verified. For instance, when we start the application the input generator may not be the bottleneck (*on stream* case). Thus, we initially select the task farm version. During the execution, the input generator can become a bottleneck of the computation. This can happen when input data from sensor networks are not available anymore, and they must be partially simulated. The current TSM interarrival time becomes higher than the corresponding service time (i.e.

$T_A >> T_{TSM}$), also for the lowest parallelism degree of the TSM parallel structure. To manage this situation in real-time, we can switch from the task farm version, which reduces only the average service time, to the data parallel one, which reduces also the latency for a single request.

We can deduce two features for a high-level programming model for Pervasive Grid applications. The programming model should allow programmers to: (i) define a (parallel) module in multiple versions, and (ii) express the abstract switching conditions without entering in implementation details. Point (ii) avoids to mix in a same abstract code the version switching implementation and the abstract policy.

Experiments: We performed experiments on the task farm mapped onto a cluster architecture, which models a centralized server. The cluster features 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 mapped the data parallel onto an interface node between the WSN and mobile users, supported by a multicore architecture: this is motivated by the strong requirements, in terms of communication efficiency, of this version, which cannot be supported by an off-the-shelf cluster architectures. The interface node features a Intel E5420 Dual Quad Core multicore processor, with 8 cores of 2.50 GHz, 12 MB L2 Cache and 8 GB of main memory. To compare the algorithms we

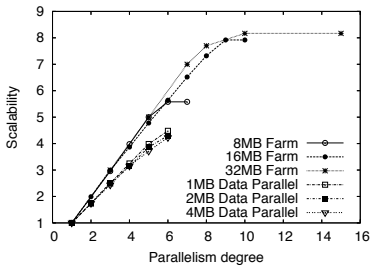

Figure 3. Scalability of the task farm and data parallel versions.

present their scalability, instead of their service time, to mask the difference between the single node computing power. In Fig. 3 we show the results. It can be noticed that the task farm version supports higher system sizes, w.r.t. the data parallel one. This is motivated by physical aspects of the used architectures. Results show that the task farm version provides better scalability also for large system sizes. In this scenario we select it, unless we need to minimize the single system resolution latency (see above).

B. Scenario 2: Forecasting Local River Areas

In the second scenario (see Fig. 4) we model the case of multiple mobile users (or clients), which are near the emergency area and they do not need a forecasting of the whole river, but only of their nearest area. Each client cyclically performs the following program: (1) collects input data; (2) sends input data to TSM; (3) receives corresponding results from TSM; (4) visualizes the received results. We denote with T_G the average time needed to perform steps (1) and (4). Unlike the previous scenario, the number of clients is now a critical parameter of the performance model. The performance of this scenario can be modeled as a *Queuing System*: a set of clients sends requests that are logically received by a TSM input queue (i.e. Q). We are interested in the performance experience of each client w.r.t. the system: the performance characterization of this scenario is given by the client service time (T_{Cl}). By exploiting the queuing theory [16], we derive that the T_{Cl} depends on the service latency (L_{TSM}) and the average waiting time in the TSM queue W_Q: this depends on T_{TSM} [16]. Unlike the first scenario, we are interested in optimizing both values (T_{TSM} and L_{TSM}) *at the same time*. As we described above, the data parallel version minimizes both values, while the task farm one only optimizes the T_{TSM} value. We performed experiments to show how, given an actual computing platform, we can derive the selection policies.

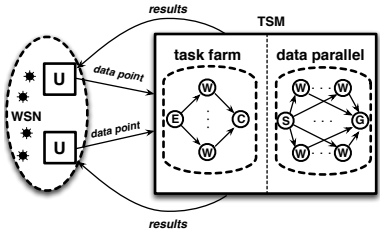

Figure 4. Representation of Scenario 2.

Experiments: Fig. 5 shows the version service times for system size of 4MB and 5 clients. The data parallel version should be preferred when the TSM interarrival time is larger (enough!) than its service time ($T_A >> T_{TSM}$). We can consider to map the task farm onto both the cluster and the multicore. The multicore can support up to 6 parallel processes: for client numbers larger than 6 it is not guaranteed which architecture is best suited. In the experiments we have not reached the sufficient client number which requires to switch from the task farm on the multicore to the one on the cluster: we will investigate this point in future work. For discussion, we define such value as Δ(=THRESHOLD). The version selection policy can be defined as following:

- if we are executing the task farm on the cluster and $T_A >> T_{TSM}$, we switch to the data parallel. We switch back if the number of clients becomes larger than Δ;
- if the number of clients becomes lower than Δ we switching from the task farm on the cluster to one of the other two versions: the best version is selected by checking T_A;
- if the T_A features a high variance our selection strategy may switch from the task farm on the multicore to the

data parallel one. As service times are comparable, we avoid the switching costs by permanently selecting the data parallel version.

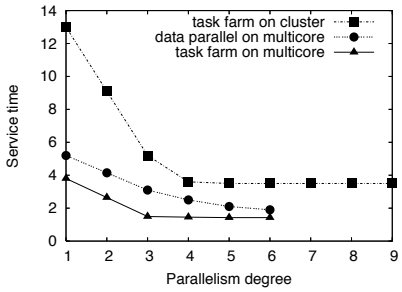

Figure 5. TSM service times for 5 clients.

V. The ASSISTANT Programming Model

ASSISTANT (*ASSIST with Adaptivity and Context Awareness*) inherits from our previous research experience in structured parallel programming [7] maturated in the ASSIST programming model [4]. ASSISTANT allows programmers to define parallel and adaptive applications as a graph of interconnected modules, each one defined by using a specific programming construct (the new parallel module, or *ParMod*). The parmod semantic is characterized by two different logics of the module and their interaction scheme:

- **Functional Logic**: this part encapsulates multiple versions of the parallel module, each one with a different behavior according to several parameters (e.g. memory utilization, estimated performance and user degree of satisfaction, e.g. the precision of computed results). In a certain time only one version can be currently executed (the version is active);
- **Control Logic**: this part includes all the adaptivity strategies and *reconfigurations* performed to adapt the module behavior as a response to specified events. Events can be generated by context interfaces which provide useful information about the surrounding execution environment (e.g. network disconnection events). By using proper performance models, the control logic can select the best version to execute when a certain context situation is verified, maximizing or minimizing specific execution parameters (e.g. the module response time or its memory occupation).

To express these logics we introduce a new construct, called *operation*, which is utilized inside the parmod definition. An operation is composed by a specific version (i.e. its functional part) and the corresponding adaptivity strategy (i.e. its control part) utilized when the version is executed. A parmod includes multiple operations which describe its functional and control logics.

The control logic of a parmod can be formally described as an automaton (see the example of Fig. 7), where: internal states are a mapping of the operation names; output states are a mapping of reconfiguration actions; input states are a mapping of event combinations admitted in each internal state of the automaton. Input states are defined also on conditions on internal variables of the parmod state. Self-arrows denote re-configurations of the same operation (e.g. changing the parallelism degree). Transitions between different operations denote operation switchings. The events labeling transitions are the condition firing the adaptivity actions. Syntactically, the control part of an operation implements the transitions outgoing from itself (also self-arrows). It is programmed by means of the *on_event* construct, which includes a set of nondeterministic clauses expressed as condition-action rules.

The control logic and the functional logic of a parmod follow the interaction scheme depicted in Fig. 6. The computation performed by the parmod can be reconfigurated in specific safe-points (i.e. *reconfiguration points* as in [2]) implicitly identified by the run-time system or explicitly defined by the programmer. As example, if the parmod processes a large sequence of input tasks, a typical reconfiguration point is defined when a new task is received and before starting the corresponding elaboration. When the execution reaches a re-

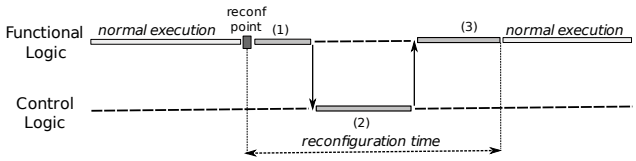

Figure 6. Interaction scheme between functional and control logics.

configuration point all the information necessary to evaluate the non-deterministic clauses of control logic are provided (1). The control logic evaluates the on_event construct of the active operation identifying the reconfigurations which must be executed (2). The functional logic reacts to the reconfiguration commands (3) by modifying its parallelism degree (i.e. new processes are allocated) or changing the active operation (a different version is executed).

A. Adaptive Flood Forecasting Application in ASSISTANT

We show a control logic part of the TSM parmod in the second scenario (see Sect. IV), which implements the described switching policy. Fig. 7 shows the event operation graph of the TSM parmod in this client-server scenario. Fig. 8 shows a part of the TSM control logic (i.e. when the *taskFarmMulticore* is activated). This on_event section implements the bold lines of the corresponding event-operation graph of Fig. 7, i.e. the ones outgoing from the *farmMulticore-Operation* internal state. The first nondeterministic clause regards the average interarrival time of the TSM, measured by a specific context interface of the module (updating the *TA* internal state variable). If this value becomes lower than a predefined threshold (i.e. *Ta-THS*, case

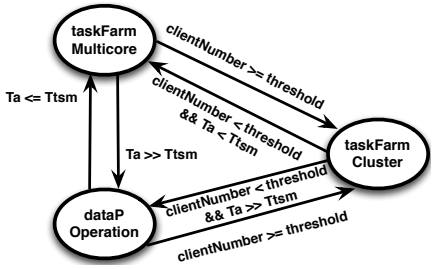

Figure 7. Event-operation graph of the TSM parmod in the second scenario.

```
parmod TSM(..) {
  operation farmMulticore-Operation
   //Functional logic: task-farm algorithm..
   on_event
     (Ta > Ta-THS) do
         farmMulticore-Operation.stop();
         dataParallel-Operation.start();
       enddo
     (Ta <= Ta-THS)&&(clientNumber > THRESHOLD) do
         farmMulticore-Operation.stop();
         farmCluster-Operation.start();
       enddo
}
```

Figure 8. Control logic of the multicore farm operation of the TSM parmod.

$T_A >> T_{TSM}$) we can only minimize the single system resolution time switching to the data parallel operation. The second nondeterministic clause regards the number of clients performing requests to the TSM module. If the interarrival time is lower than a specific value and the number of clients is higher than a certain threshold we select the task farm version mapped onto the cluster architecture, exploiting an higher parallelism degree than the multicore operations.

VI. Conclusions

In this paper we have described how adaptivity can be derived for emergency management applications on Pervasive Grids, by assuming two application scenarios and characterizing their performance models. For a specific application module we have defined two versions, based on two sequential algorithms and parallelization techniques. *We have shown how to define the version selection: this is given independently of any actual implementation of adaptivity, in terms of application manager modules/processes*. These abstract considerations are synthesized in the ASSISTANT programming model for pervasive adaptive applications, which is briefly presented in this paper in its main aspects.

References

[1] T. Priol and M. Vanneschi, Eds., *Procs. of the CoreGRID Symposium: From Grids To Service and Pervasive Computing*, ser. LNCS. Springer, 2008.

[2] M. Vanneschi and L. Veraldi, "Dynamicity in distributed applications: issues, problems and the assist approach," *Par. Comp.*, vol. 33, no. 12, pp. 822–845, 2007.

[3] C. Bertolli, R. Fantacci, G. Mencagli, D. Tarchi, and M. Vanneschi, "Next generation grids and wireless communication networks: towards a novel integrated approach," *Wireless Comm. and Mobile Computing*, 2008.

[4] M. Vanneschi, "The programming model of ASSIST, an environment for parallel and distributed portable applications," *Par. Comp.*, vol. 28, no. 12, pp. 1709–1732, 2002.

[5] R. M. Badia, M. Ejdys, U. Herman-Izycka, N. Lal, T. Kielmann, and E. Tejedor, "Integrating application and system components with gcm," in *From Grids To Service and Pervasive Computing*, ser. LNCS. Springer, 2008.

[6] M. Aldinucci, M. Danelutto, and P. Kilpatrick, "Co-design of distributed systems using skeletons and autonomic management abstractions," in *Workshops of Euro-Par 2008*, ser. LNCS, vol. 5415. Springer, 2009.

[7] M. Cole, "Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming," *Par. Comp.*, vol. 30, no. 3, pp. 389–406, 2004.

[8] M. Baldauf, S. Dustdar, and F. Rosenberg, "A survey on context-aware systems," *Int. J. Ad Hoc Ubiquitous Computing*, vol. 2, no. 4, pp. 263–277, 2007.

[9] M. Román, C. Hess, R. Cerqueira, A. Ranganathan, R. Campbell, and K. Nahrstedt, "A middleware infrastructure for active spaces," *IEEE Perv. Comp.*, vol. 1, no. 4, pp. 74–83, 2002.

[10] T. Chaari, D. Ejigu, F. Laforest, and V.-M. Scuturici, "A comprehensive approach to model and use context for adapting applications in pervasive environments," *J. Syst. Softw.*, vol. 80, no. 12, pp. 1973–1992, 2007.

[11] A. Balasubramanian, B. N. Levine, and A. Venkataramani, "Enhancing interactive web applications in hybrid networks," in *In Proc. of the ACM Intl. Conf. on Mob. Comp. and Netw.* ACM, 2008, pp. 70–80.

[12] B. Noble and M. Satyanarayanan, "Experience with adaptive mobile applications in odyssey," *Mob. Netw. Appl.*, vol. 4, no. 4, pp. 245–254, 1999.

[13] D. Lillethun, D. Hilley, S. Horrigan, and U. Ramachandran, "MB++: An integrated architecture for pervasive computing and high-performance computing," in *In Proc. of the IEEE Intl. Conf. on Emb. and Real-Time Comp. Syst. and Appl.* IEEE, 2007, pp. 241–248.

[14] B. Syme, "Dynamically linked two-dimensional/one-dimensional hydrodynamic modelling program for rivers, estuaries and coastal waters," WBM Oceanics Australia, Tech. Rep., 1991, available at: http://www.tuflow.com/Downloads/.

[15] R. Hockney and C. Jesshope, *Parallel Computers: Architecture, Programming and Algorithms*. Institute of Physics Publishing, 1981.

[16] L. Kleinrock, *Queueing Systems, Volume I: Theory*. Wiley Interscience, 1975.