Analyzing Memory Requirements for Pervasive Grid Applications

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

***Abstract*—Pervasive Grid Computing Platforms include centralized computing nodes (e.g. parallel servers) as well as decentralized and mobile devices. Pervasive Grid applications include data- and computing-intensive components which can be mapped also onto decentralized and mobile nodes. The effective and practical success of this mapping resides also in deriving proper configurations of applications which consider the** *limited memory capabilities* **of those resources. In this paper we target this issue by showing how we can study and configure the memory requirements of an Emergency Management application. We present our solutions by using the ASSISTANT programming model for Pervasive Grid applications.**

***Index Terms*—Parallel Processing, Parallel Language, Structured Programming, Memory Management**

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. 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.

Until now these complex applications have been mapped onto centralized parallel servers [2], while decentralized and mobile nodes were mainly used for back-end tasks (e.g. result visualization). The main motivation behind this choice is that those devices offer poor resources, which are generally thought to be insufficient for executing a complex application. From another viewpoint, we notice that this is true because complex computations are programmed and configured only to be mapped on a high-performance platform.

Our research work is based on the definition of the ASSISTANT [3] (ASSIST with Adaptivity and Context-Awareness) programming model which allows programmers to express multiple versions of a same parallel module targeting different execution platforms. In this paper we analyze the case in which the target platforms have constraints on the available memory space, which characterize decentralized and mobile computing nodes. As contributions of this paper, we will show that the features of the introduced programming model allow:

- to express parallel computations which memory requirements can be statically and dynamically analyzed and configure;
- to express adaptive behavior w.r.t. memory constraints of available computing nodes.

The main features of ASSISTANT are that it allows to express parallel computations by means of a general blend of algorithmic skeletons [4] (also called structured parallel programming). In this paper we show how the properties of algorithmic skeletons allow us to target the first contribution of above. To concretize this contribution we focus on a flood management application, we show multiple versions of a flood forecasting module and we provide its memory requirements analysis. We also performed experiments to assess the analytical results. ASSISTANT also offers high-level programming constructs to express dynamic selection and configuration of multiple parallel versions of a same program. We show how these feature allows us to target the second contribution of above and we concretize it by showing how adaptivity can be programmed for the flood management application.

The outline of this paper follows: Sec. II describes related works; Sec. III describes the flood management application; Sec. IV gives an overview of the main ASSISTANT constructs; Sec. V describes an implementation of the flood management application in ASSISTANT, highlighting memory-related logics. Sec. VI draws the conclusions of this work.

II. Related Work

Several research works aim at making possible the exploitation of mobile computing nodes to perform simple to complex adaptive computations. These works are based on the concepts of context [5]. The context includes environmental data such as air temperature and the network and node states. Smart Space systems 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 [5].

Concerning adaptivity, a mobile application can exploit optimized algorithms, protocols or systems in its run-time. Adaptivity can be also defined at the application level [6]. For instance, in Odyssey [7] adaptivity is expressed in terms of the choice of the services from which an application is composed.

In [2] High-performance application are introduced in the context of pervasive computing, for stream-based transformations, fusion and feature extractions. Unlike our work, high-

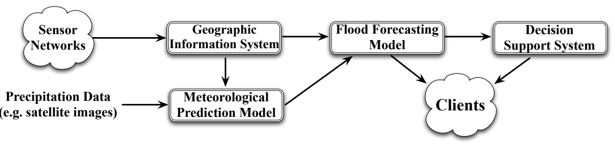
Sensor Networks
Geographic Information System
Flood Forecasting Model
Decision Support System
Precipitation Data (e.g. satellite images)
Meteorological Prediction Model
Clients

Fig. 1. Scheme of the flood management application.

performance computations are only mapped on centralized parallel servers.

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 by other services (meteo and GIS). For instance sensors can monitor the current value and 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 [8] hydrodynamic model, which is based on mass and momentum partial differential equations to describe the flow variation at surface. Their discrete resolution requires, for each time slice, the resolution of a very large number of tridiagonal systems. Parallel techniques are available that make it possible reasonable response times in a scalable manner. The quality of the forecasts also depends on the size of the tridiagonal systems. From the memory occupation viewpoint, larger system size induce a higher memory occupation for the resolution, independently of the used sequential and parallel algorithm.

To achieve high-performance we choose to map the forecasting module on a centralized high-performance parallel server (e.g. a cluster). Nevertheless, in a Pervasive Grid we are forced to consider also critical situations. An example of such a 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 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 nodes* whose interconnections are currently reliable. For this mapping to be feasible, *programmers must design their applications in such a way that multiple versions of modules (e.g. the forecaster) are provided, each featuring different memory requirements*.

In this paper we consider two cyclic reduction algorithms [9] for tridiagonal system solving. For brevity, we avoid to introduce their mathematical formulations: the interested reader can refer to [9].

Below, we show our methodology in designing and developing complex High-Performance Pervasive Grid applications w.r.t. the study of memory requirements.

IV. The ASSISTANT Programming Model

For brevity, we give a brief description of the main programming constructs of ASSISTANT. The reader can refer to [3] for a full description.

a) Parallelism: As introduced above, ASSISTANT provides constructs to express structured parallel computations composed by means of streams [10]. In particular, the following constructs are available:

- **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**: in this construct we specify the parallel computation applied to each input data, possibly producing an output result. Parallel computations that can be expressed are task farm and data parallel.
- **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.

The programmer can define multiple parallel modules (or ParMod) by means of this construct, and she/he can compose them in generic graphs by means of streams.

b) Adaptivity: The ParMod also includes constructs to express multiple versions of a same module and their dynamic control, i.e. its adaptivity. In a single ASSISTANT ParMod multiple parallel computations are encapsulated in multiple instances of the *operation* construct. This construct is the unit of adaptivity and deployment and it can be described by the following logics:

- **Functional** logic: this includes all the computations (sequential or parallel) performed by the ParMod and expressed according to the model described above.
- **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).
- **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 and to the dynamic state of the computation (e.g. the module service time).

A ParMod includes multiple operations each specifying its own logics of above. The general adaptive semantics of a ParMod can be described as following. When a ParMod is started, a user-specified *initial* operation is activated. Only one

Mem. Req. per Worker	Alg. 1	Alg. 2
Task Farm	$M_w(N,\delta,n) = (q-1)\cdot(4\cdot N\cdot\delta) + N\cdot\delta$	$M_w(N,\delta,n) = 5\cdot N\cdot\delta$
Data Parallel	$M_w(N,\delta,n) = \frac{1}{n}\left[(q-1)\cdot(4\cdot N\cdot\delta) + N\cdot\delta\right]$	$M_w(N,\delta,n) = \frac{1}{n}(5\cdot N\cdot\delta)$

TABLE I
MEMORY REQUIREMENTS PER WORKER

operation for each ParMod can be currently active. During the execution the 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).

V. ADAPTIVITY FOR FLOOD MANAGEMENT

We describe two cyclic reduction algorithms we parallelize them.

A. First Algorithm

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

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

where a_i, b_i, c_i and k_i are the three diagonal coefficients and the constant term of the i-th system row. The superscripts denote the computational step at which their values are taken.

The second part of this algorithm is denoted *resolution part* in which we compute the solutions to the linear 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 $i\ mod\ 2^l = 0$ and we compute:

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

In this case we do not need multiple x values for each computation step. The memory requirement of this algorithm is the sum of the following terms:

- as we have to keep all the values computed during the transformation part, as they are used in the resolution one, we need: $(q-1)\cdot(4\cdot N\cdot\delta)$ bytes, where we have considered 4 values for each row (a,b,c and the constant term k) and δ is the system size for double precision floating point values. As α and γ are temporary, we avoid to consider their cost;
- the resolution part only requires a single copy of the x array: the memory requirement is $N\cdot\delta$ bytes.

B. 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* system rows at each steps. 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 during the first part. Thus:

- the transformation part rewrites at each step the whole tridiagonal system. At each step we only need the values computed at the previous step and the new ones: $N\cdot 4\cdot\delta$ bytes;
- the resolution part requires the same memory amount of the first algorithm.

Thus, this algorithm requires $N\cdot 4\cdot\delta + N\cdot\delta = 5\cdot N\cdot\delta$ bytes.

C. Parallel Versions

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 on a different input system. Each worker performs a whole cyclic reduction algorithm on each assigned input system. Thus, the memory requirements for each worker correspond to the one of the whole algorithm.
- **Data Parallel**: each tridiagonal system is partitioned (scattered) onto replicated workers, each one performing the sequential algorithm in its partition. Workers cooperate during each step according to the communication stencil. The result is obtained by gathering local results. We assume that the system is equally partitioned into workers. For n workers the memory requirements for each one of them correspond to the algorithm memory occupation divided by n.

We compute the memory requirements per worker (see Table IV-0b) and per parallel module (see IV-0b) of both algorithms. The formulas can be simply derived by proper multiplication and division of the sequential formulas.

It is important to notice that the task farm version operates in parallel on p systems, while the data parallel one solves only a system at time. This is the reason behind the different memory requirements of task farm and data parallel paradigms.

From the analysis of the equations it can be noticed that:

Mem. Req. per Module	Alg. 1	Alg. 2
Task Farm	$M_p(N,\delta,n) = n \cdot [(q-1)\cdot(4\cdot N\cdot\delta) + N\cdot\delta]$	$M_p(N,\delta,n) = n\cdot(5\cdot N\cdot\delta)$
Data Parallel	$M_p(N,\delta,n) = [(q-1)\cdot(4\cdot N\cdot\delta) + N\cdot\delta]$	$M_p(N,\delta,n) = (5\cdot N\cdot\delta)$

TABLE II
MEMORY REQUIREMENTS PER PARALLEL MODULE

- for the single worker, the task farm memory requirements remain constant, equal to the memory requirements of the sequential algorithm. The data parallel memory requirements decrease linearly with the parallelism degree;
- for the whole parallel module, the task farm requirements increase linearly with the parallelism degree. Also in this case, the data parallel decrease linearly its memory requirements.

From these observations we can derive a general selection policy for the versions of above:

- the first algorithm parallelized as a task farm requires more memory than the other versions. We can map it on the centralized parallel server where each node has its own local memory: increasing the parallelism degree also increase the total memory available;
- the second algorithm parallelized as a task farm requires less memory than the first algorithm, but more than the data parallel version. We can map it on a decentralized interface node featuring a multicore architecture, which memory availability is larger than that of mobile devices but lower than that of a centralized server.
- the second algorithm parallelized according to the data parallel paradigm requires less memory than the other versions: it is feasible to be mapped on a decentralized set of mobile devices or on interface nodes with limited memory capabilities.

D. Programming Adaptivity

We show how adaptivity can be programmed in ASSISTANT, also taking into account memory requirements, in a simple example. Suppose that we are executing a data parallel (second algorithm) on an interface node and that the client PDAs become disconnected. In this case we start executing the forecasting module directly on the PDA network: before starting the execution we compute the optimal application configuration, w.r.t. generated system size, by analyzing the currently available memory of PDAs.

Fig. 2 shows the control logic of the operation executed on the interface node. The operation control logic is implemented in the *on_event* construct:

- in the case in which all connections fail, perform the actions in the **do**-**enddo** block;
- read the available memory per PDA from context interfaces.
- compute the maximum value for N (system size) which requires a total memory which is lower or equal to the one available on each PDA (we use the M_w function). The result must be lower than the detected available memory (*constraint* function);
- configure *pdaOP* to generate to the computed N value;
- instantiate the output interface for results in such a way that it corresponds to the size of results, which depend on the size of the generated and solved systems (half of the system size, in this forecasting problem).

Finally, we can start the *pdaOP*.

```
operation interfaceNodeOP {
  //Local variables of the operation:
  int pardeg;
  //... parallel implementation...
  on_event {
   ( (wireless_fail()) && (wired_fail()) )
   do
        this.stop();
        //get the available memory per PDA
        int avail_mem = context.pdaMem();
        //compute the max system size (N) for
            which the memory
        //req. are below the available one
        int N = constraint(max(N, Mw(N,delta,1))
            <= avail_mem);
        //configure the generated system size..
        pdaOP.setSystemSize(N);
        //..and output result size
        pdaOP.setInterface(result, N/2);
        pdaOP.start();
     enddo
  }
}
```

Fig. 2. Operation of the TSM module for the interface node.

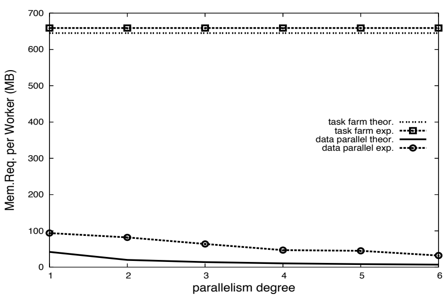

Fig. 3. Memory occupation per worker of the task farm and data parallel for $N = 2^{20} - 1$ by varying the parallelism degree.

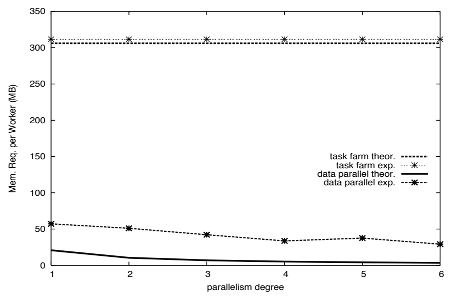

Fig. 4. Memory occupation per worker of the task farm and data parallel for $N = 2^{19} - 1$ by varying the parallelism degree.

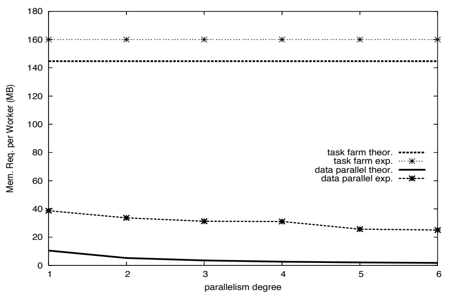

Fig. 5. Memory occupation per worker of the task farm and data parallel for $N = 2^{18} - 1$ by varying the parallelism degree.

E. Experiments

We performed experiments to show the difference between the memory requirements of the task farm with the first algorithm and the data parallel with the second algorithm.

For performance reasons, we mapped the task farm version on a cluster architecture, featuring 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 operation on a Intel E5420 Dual Quad Core multicore processor, featuring 8 cores of 2.50 GHz, 12 MB L2 Cache and 8 GB of main memory.

On both platforms we have implemented our operations in C++ exploiting the MPICH library for inter-process communications. At the implementation level we have to consider also the unavoidable costs of process and communication library implementation, which also depend on the selected platforms. Results show that this support cost dominates the application total memory requirements for smaller system sizes.

Fig. 3 shows the comparison for the task farm and data parallel between theoretically computed values and actually measured ones for the case of $N = 2^{20} - 1$ by varying the parallelism degree (1 to 6 processors). In the figure dotted lines represent experimental data, while continuous ones theoretical data: the latter values are computed by solving the M_w equations (see Sect. III). Notice that, as expected, the theoretical value is always below the corresponding experimental one: this happens because the M_w equations model the application state, not all the needed memory. In the experimental data data we have also to add the costs given by the process and communication implementation.

We have obtained similar results for the cases of $N = 2^{19} - 1$ (Fig. 4) and $N = 2^{18} - 1$ (Fig. 5). In all figures, notice that for low partition sizes (e.g. data parallel case), the memory requirements are dominated by the costs of process and communication implementations. This cost is dependent on the used platform, but also in the way in which we support our application.

To summarize, when the system size has a significant impact on the application memory occupation, our analysis based on structured parallelism models represents a good framework to express adaptivity aspects w.r.t. memory requirements. This is not true in the case in which the memory occupation is dominated by implementation mechanisms of processes.

VI. Conclusion

In this paper we have shown how memory requirements for complex Pervasive Grid applications can be analyzed by exploiting the structured parallel programming paradigms. We have described our analytical methodology on a flood management application and we have show how this can be used to program adaptive behavior in ASSISTANT.

References

[1] T. Priol and M. Vanneschi, *From Grids To Service and Pervasive Computing*. Springer, 2008.

[2] 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 Intl. Conf. on Emb. and Real-Time Comp. Syst. and Appl.* IEEE, 2007, pp. 241–248.

[3] C. Bertolli, D. Buono, G. Mencagli, and M. Vanneschi, "Expressing adaptivity and context-awareness in the assistant programming model," in *Procs. of the Third Intl. ICST Conf. on Autonomic Comp. and Comm. Syst.*, 2009.

[4] 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.

[5] 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.

[6] 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 Comp.*, vol. 9, no. 4, pp. 445–467, September 2008.

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

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

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

[10] 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.