ELSEVIER

Future Generation Computer Systems 17 (2001) 969–975

FGCS
FUTURE GENERATION COMPUTER SYSTEMS

www.elsevier.nl/locate/future

Integrating load balancing and locality in the parallelization of irregular problems☆

Fabrizio Baiardi*, Sarah Chiti, Paolo Mori, Laura Ricci

Dipartimento di Informatica, Università di Pisa, Corso Italia 40, 56125 Pisa, Italy

Abstract

An irregular problem models the evolution of a system where several elements are irregularly distributed in a domain. The evolution modifies this distribution in a way that cannot be foreseen and the behavior of each element depends upon the elements close to it according to a problem dependent relation. Starting from a hierarchical representation of the domain, we define a parallelization methodology that includes a load balancing strategy that preserves this locality property and a strategy to collect information distributed onto the processing nodes. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Irregular problems; Distributed memory architectures; Adaptive multigrid methods; Load balancing; Locality

1. Introduction

Several complex systems are modeled through time dependent partial differential equations (PDEs), solved through adaptive iterative algorithms that compute some properties, i.e. speed, position, illumination etc., for each element in a domain of interest. The iterations either simulate the system evolution in an interval of time or improve the accuracy of the results. The properties of an element e_i depend upon those of other elements, the neighbors of e_i. A problem dependent neighborhood relation determines the neighbors of e_i, but the probability that e_j is a neighbor of e_i is inversely related to the distance between e_i and e_j. This property is denoted as *locality*. A problem is irregular if the elements are irregularly distributed in the domain and their number and/or their distribution change in a way that cannot be anticipated. A parallel application to solve an irregular problem can achieve a satisfactory speed up only if the mapping of the elements onto the processing nodes, p-nodes, of the architecture balances the computational load and it preserves locality, i.e. it maps onto the same p-node elements close to each other. Furthermore, since the computational load of each element is a function of the distribution, the mapping has to be updated as the distribution changes. Among the methods that are important examples of irregular problems, we recall Barnes–Hut [2], adaptive multigrid [5] and hierarchical radiosity [7].

This paper defines a methodology to solve an irregular problem through a distributed memory architecture where the p-nodes are connected by a sparse network. Starting from the methodology, a package will be defined to support the development of parallel applications. Alternative approach to irregular problems have been described in [9,10]. Each of the next three sections introduces one of the component of the proposed methodology, namely, a hierarchical representation of the distribution, the load balanc-

☆ This work has been partially supported by CINECA.

* Corresponding author. Tel.: +39-050-2212762; fax: +39-050-2212726.

E-mail addresses: baiardi@di.unipi.it (F. Baiardi), mori@di.unipi.it (P. Mori), ricci@di.unipi.it (L. Ricci).

PII: S0167-739X(01)00039-5

ing strategy that maps the element onto the p-nodes and updates the mapping at run time and the strategy to collect the properties of elements mapped onto another p-node. The two strategies exploit the hierarchical representation to deduce the current mapping of the elements. The experimental results of the methodology in the case of the adaptive multigrid method (AMM) are discussed in Section 5. Those of the Barnes–Hut method have been presented in [1].

2. A hierarchical representation of the element distribution

All the strategies of our methodology exploits a hierarchical representation of the distribution of the elements, the H-Tree. Each node N of the H-Tree, hnode, represents a subspace of the domain, *space*(N), and it describes the elements in *space*(N). As discussed in the following, the detail of the information increases with the depth of N in the H-Tree. If the elements in *space*(N) do not satisfy a problem dependent condition, *space*(N) is partitioned into equal subspaces by halving each of its dimensions. The distribution of the elements in the subspaces are represented by the sons of N. If *space*(N) is not decomposed, N is a leaf of the H-Tree. Because of the irregular distribution, the depths of two distinct sub-trees rooted in the same hnodes may be very different. During the computation, the domain decomposition and the H-Tree are updated according to the evolution of the system. As soon as *space*(N) is partitioned, the corresponding hnodes are inserted, while these hnodes are pruned if *space*(N) is no longer partitioned. In any real applications, the number of elements and that of hnodes is so large that the H-Tree cannot be replicated in each p-node. Hence, the H-Tree is partitioned into $np+1$ subsets, where np is the number of p-nodes. One subset of the H-Tree, the *replicated H-tree*, is replicated in all the p-nodes. Each of the other subsets is stored in one p-node only and it defines the *private H-Trees* of the p-node.

3. Initial mapping and run time reallocation

To take locality into account while balancing the computational load, we define a mapping of the spaces onto the p-nodes by ordering the spaces in the hierarchical representation and by partitioning the resulting sequence. The spaces are ordered through a *space filling curve* [8] built on the hierarchical representation of the element distribution, starting from the lowest level spaces, i.e. from the first partition of the problem domain. The spaces at the same level are visited in the order stated by the characteristic figure of the adopted curve. If a space S has been partitioned, then all its subspaces are visited in a recursive way, before the next space at the same level of S. Because of the properties of a space filling curve, this space is always a neighbor of S. Any space filling curve *sf* also defines a visit $v(sf)$ of the H-Tree that returns a sequence $Sq(v(sf)) = [N_0, N_1, \ldots, N_m]$ of hnodes. Alternative curves may be adopted because the aspects of $v(sf)$ that depend upon *sf* may be encapsulated into a function that returns the next hnode to be visited. The *load* of a hnode N is a problem dependent metric that evaluates the amount of computations due to the elements in *space*(N). According to the considered problem, this load can be (i) constant and equal for all the hnodes; (ii) constant but distinct for each hnode; or (iii) variable and distinct for each hnode. In the last case, the program has to be instrumented to measure the load during the computation. In those parallel architectures where the cost of a communication depends upon the communicating p-nodes, the *np* p-nodes are ordered too. A p-node A immediately precedes B in the ordered sequence *SP*, if the cost of an interaction between A and B is not larger than the cost of the same interaction between A and any other p-node following B. Since each p-node executes one process, in the following, P_k denotes both the kth p-node of *SP* and the process executed by the p-node. To preserve the ordering, the spaces are mapped onto the p-nodes through a blocking strategy that partitions $Sq(v(sf))$ into *np* segments, i.e. into *np* subsequences of consecutive hnodes. The first segment is mapped onto P_0, the second onto P_1 and so on. The resulting mapping satisfies the **range property**: *if the hnodes N_i andN_{i+j} are assigned to P_h, then all the hnodes in-between N_i andN_{i+j} in $Sq(v(sf))$ are assigned to P_h as well*. This property guarantees that the spaces assigned to P_h are close to each other. The load of each segment should be as close as possible to *average_load*, the ratio between the overall load and the number of p-nodes. We cannot assume that the load of each segment is equal to *average_load*, because each hnode, and its load, can

be assigned to one segment only. However, due to the large number of elements, the difference between *average_load* and the load of each segment is negligible. Starting from the chosen mapping, each process P_h builds the replicated H-Tree and its private H-Trees. An hnode N belongs to one private H-Tree of P_h if *space*(N) is assigned to P_h. The replicated H-Tree is the union of the paths from the root of the H-Tree to those of the private H-Trees. Each hnode N of the replicated H-Tree records the position of *space*(N) in the domain and the identifier of the owner process, while each hnode N of one of the private H-Trees records the properties of *space*(N). In some problems, the intersection among a private H-Tree and the replicated H-Tree includes the roots of the private H-Tree only. In other problems, the private H-Trees and the replicated H-Tree are partially overlapped.

Due to the system evolution, the initial mapping may later result in an unbalanced load. We define a procedure to update the mapping while respecting locality and minimizing the corresponding overhead. To detect when the mapping has to be updated, each process periodically broadcasts its workload and computes *max_unbalance*, the largest difference between *average_load* and the workload of each process. To avoid a too frequent execution, the procedure is invoked only if *max_unbalance* is larger than an user defined threshold T. In the following, $= (Se, C)$, where Se is a segment of hnodes and C is a constant, denotes the segment Se whose load is as close as possible to C. Let us suppose that the workload of P_h is $average\_load + C$, $C > T$, while that of P_k, $h < k$, is $average\_load - C$. To recover the unbalance, P_h cannot send to P_k a set of hnodes whose load is equal to C because the resulting mapping violates the range property. Hence, any process P_i in between P_h and P_k is involved in a shift of the spaces from P_h to P_k. Let us define Pre_i and Suc_i as the two sequences of processes $[P_0, \ldots, P_{i-1}]$ and $[P_{i+1}, \ldots, P_{np-1}]$ that, respectively, precede and follow P_i in SP. Furthermore, $Unb(Pre_i)$ and $Unb(Suc_i)$ are, respectively, the global load unbalances of Pre_i and Suc_i. If $Unb(Pre_i) = C > T$, i.e. processes in Pre_i are overloaded, P_i receives from P_{i-1} a segment S_1 where $= (S_1, C)$. If, instead, $Unb(Pre_i) = C < -T$, P_i sends to P_{i-1} a segment S_2 where $= (S_2, C)$. The same procedure is applied to $Unb(Suc_i)$ but, in this case, the hnodes are either sent to or received from P_{i+1}. To respect the range property, if $[N_q, \ldots, N_r]$ is the segment assigned to P_i, then P_i sends to P_{i-1} a segment $[N_q, \ldots, N_s]$, while it sends to P_{i+1} a segment $[N_t, \ldots, N_r]$, with $s \leq r$ and $q \leq t$.

4. Collecting properties from other p-nodes

To compute the properties of each element e in a space it has been assigned, a process needs those of the neighbors of e, that may have been allocated onto other p-nodes. The simplest strategy to collect such remote data is *request/answer*. As soon as it needs the properties of an element e in a space S mapped onto another p-node, P_h analyses the replicated H-Tree to discover the process P_k where S is allocated, it suspends the computation and sends a request to P_k. In this way, two communications have place for these properties for, respectively, the request and the reply.

To reduce this overhead, we introduce the *fault prevention* strategy. Each process P_h, for each of its spaces S, determines which processes require the properties of the elements in S, and it sends to these processes the data, without any explicit request. To determine all the data to be sent to P_k, P_h exploits the neighborhood stencil and the information on the subspaces assigned to P_k in the replicated H-Tree. In this way, one communication suffices to collect a remote data from another p-node. Moreover, strategies such as communication merging can be easily applied to reduce the communication overhead. In general, P_h approximates the data that P_k requires, because the replicated H-Tree records a partial information only. The approximation is always safe, i.e. it includes any data P_k needs, but, due to the approximation, some of the data sent may be useless for P_k. If the information in the replicated H-Tree does not enable the processes to compute an accurate approximation, *informed fault prevention* can be adopted where the processes exchange some information about their private H-Trees before the fault prevention phase.

5. The adaptive multigrid method

The AMM is a fast iterative method based upon multilevel paradigms to solve multidimensional PDEs [5,6], that can be applied to a large set of problems

from different fields [3,4]. Starting from a uniform grid, the level 0 of the hierarchy, the AMM discretizes the domain through an irregular grid hierarchy built according to the considered PDE. Each grid of the hierarchy partitions a subset of the domain into a set of square spaces. The values of the equation are computed in each square corner; these points are the elements considered by our methodology. Let us suppose that, at a level l, a square A has been discretized through the grid g. To improve the accuracy of the values in A, a finer grid is added at level $l + 1$. The new grid represents A but, to double the accuracy of the discretization, it doubles the number of points of g on each dimension of A. As the computation goes on, finer and finer grids are added to the hierarchy until the desired accuracy has been reached in each square. To solve the PDE, several operators are applied to the points of each grid of the hierarchy in a predefined order, the V-cycle; for a complete description see [6].

5.1. Mapping the grids onto the p-nodes

Two aspects of locality of the AMM have to be considered, because the value of a point p on the grid g at level l is function of the values of the neighbors of p on the same grid g for the some operators (intra-grid or horizontal locality) and on the grids at level $l + 1$ (if it exists) and $l - 1$ for other operators (inter-grid or vertical locality). The ordering of the hnodes through a space filling curve takes into account both aspects, because the recursive definition of the curve preserves the inter-grid locality, while intra-grid locality is preserved by the ordering stated by the characteristic figure. For each square it has been assigned, with the exception of these on the border of a grid, a process which computes one point only, the rightmost downward corner of the square. This avoids duplicate computations.

The private H-Tree of process P_h includes all the hnodes representing the squares assigned to P_h, while the replicated H-Tree includes all the hnodes on the paths from the root of the H-Tree to that of one private H-Tree. A hnode can belong both to the replicated H-Tree and to a private H-Tree, because the computation is executed on all the hnodes. Consider a hnode N assigned to P_h. If one of its descendants has been assigned to P_k, $h \neq k$, N belongs to the private H-Tree of P_h, because P_h computes the value of the points in *space*(N), and to the replicated H-Tree, because it belongs to the path from the root of the H-Tree to that of the private H-Tree of P_k. The number of operations is the same for each point of a grid and does not change during the computation. Hence, the same computational load is assigned to each square, i.e. to each hnode, and we map the same number of squares to each p-node.

5.2. Informed fault prevention

In the following, we denote by $Do(P_h)$ the subdomain assigned to process P_h. Each process P_h applies the AMM operators, in the order stated by the V-cycle, to the points in the squares in $Do(P_h)$. Due to the locality of our mapping strategy, the squares required by P_h to apply the operators have been assigned to P_h as well, with the exception of some square in the border of $Do(P_h)$. For each operator *op*, P_h has to collect the updated properties of the points in these squares from the other processes. Let us define $Bo(op, P_h)$, the boundary of $Do(P_h)$, as the sets of the squares in $Do(P_h)$ such that one of their neighbors does not belong to $Do(P_h)$. $Bo(op, P_h)$ depends upon the neighborhood relation of the considered operator *op*. Let us define $I_h(op, l)$ as the set of squares not belonging to $Do(P_h)$ and including the points whose values are required by P_h to apply *op* to the subgrid at level l of $Do(P_h)$. The values of points in $I_h(op, l)$ are exchanged among the processes just before the application of *op*, because they are updated by the operators preceding *op* in the V-cycle. If fault prevention is adopted, P_h does not compute $I_h(op, l)$. Instead, for each process P_k, P_h determines through the replicated H-Tree which squares in $Do(P_h)$ belongs to $I_k(op, l)$, $\forall k \neq h$. Since the information in a hnode N does not fully describe the elements in *space*(N), P_h computes an approximation $A_h I_k(op, l)$ of $I_k(op, l)$ that, because of safety, includes all the squares that could be a neighbor of a square at level l in $Do(P_k)$. Then P_h sends to P_k, without any explicit request, the values of the points in $A_h I_k(op, l)$. To show that some of these values may be useless for P_k, suppose that $Do(P_h)$ and $Do(P_k)$ share a side, that they have been uniformly partitioned until, respectively level l and level $l - m$, and that the neighborhood stencil of *op* for the point p includes points on the same level of p only. Since P_h does not know $l - m$, it could send to P_k some

of its squares at levels higher than $l - m$ that are useless for P_k, that has no spaces on these levels. When adopting informed fault prevention, instead, P_k sends to P_h, before the fault prevention phase, the depth of each square in $Bo(op, P_k)$ that could have a neighbor in $Do(P_h)$. This allows P_h to improve the approximated set of squares sent to P_k because, to determine whether to send a square, P_h can use both the data in the replicated H-Tree and the depth of the squares received by P_k. P_k sends the depth information at the beginning of each V-cycle and this information is correct until the end of the V-cycle, when new grids may be added to improve the discretization. Hence, all the informed fault prevention phases for the operators of the V-cycle can exploit this information. If the load balancing procedure has been applied, at the beginning of the V-cycle, P_k sends the depth of all the squares in $Bo(op, P_k)$. Otherwise, since grids can be added but not removed, it sends information on the new grids only.

5.3. Implementation in MPI

Our parallel version of the AMM has been developed starting from a sequential version in C that has been extended through a set of procedures written in C plus MPI primitives. The most important of these procedures are those that, respectively, balance the load among the p-nodes, evaluate the current load unbalance and implement the informed fault prevention strategy. The procedure implementing the AMM operators are of the same sequential version because, after the informed fault prevention phase, each p-node owns all the information it needs and it can apply each operator as in the sequential case. This is an important advantage of fault prevention strategy.

The load balancing procedure is invoked at the end of the V-cycle. It computes *max_unbalance* by collecting the current load of each p-node through a MPI_Allgather. If the load is to be rebalanced, the elements are shifted among the p-nodes through MPI point-to-point communications. Then, the processes exchange the roots of their new private H-Trees to update the replicated H-Tree. Through a MPI_Allgather, each process communicates to any other how many roots it is going to send, i.e. how many trees belong to $Do(P_h)$. This information is used to allocate a buffer to be exploited by a MPI_Allgather that implements the actual exchange of the roots.

The informed fault prevention strategy is implemented through MPI point-to-point, non-blocking communications. Collective primitives, i.e. MPI_Scatter, have not been adopted, because this requires the creation, for each P_h, of a distinct communicator including any neighbor of P_h. To this aim, P_h should determine the neighbors of any process, but it does not have enough information to do so. Moreover, MPI_Comm_split cannot be adopted because the communicators associated with two processes are not disjoint. Furthermore, since the mapping of the elements may be updated, the neighbors of P_h change and new communicator should be created after each update of the mapping or each refinement. Lastly, since collective communications are blocking, they have to be properly reordered to prevent the deadlock. In order to overlap a communication with useful computation, non-blocking primitives are exploited and each process determines the data to be sent to other processes while it is waiting for the data from its neighbors. Moreover, more data to be sent to the same process are merged into one message to reduce both the number of communications and the setup overhead. This is a further advantage of this strategy and it is implemented as follows: each process P_k issues an MPI_Irecv from MPI_ANY_SOURCE to declare that it is ready to receive from any other process. While waiting for these data, P_h determines the data to be sent to all the other processes, i.e. it computes $A_h I_k(op, l)\ \forall k \neq h$. When a predefined amount of data for the same process has been determined, P_h sends it using an MPI_Issend. Subsequently, P_h checks through an MPI_Test the status of the pending MPI_Irecv. If the communication has been completed, P_h inserts the received data in its copy of the replicated H-Tree and it posts another MPI_Irecv. In any case, the computation of the data to be sent goes on. After sending $A_h I_k(op, l)$ for any k, P_h broadcasts, through $np - 1$ MPI_Isend, a synchronization message and it continues to receive data from its partners. Since P_h does not know how many data it will receive, it waits for the synchronization message from all the other processes. A MPI barrier cannot be used to synchronize the processes because, after issuing an MPI_Barrier, P_h is blocked and it cannot collect data from other processes.

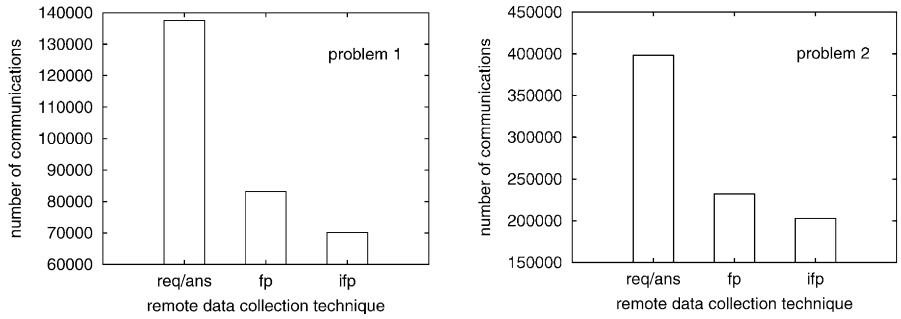

Fig. 1. A comparison of remote data collection techniques.

5.4. Experimental results

We present some experimental results of the parallel version of the AMM resulting from our methodology. The parallel architecture we consider is a Cray T3E. Each p-node has a DEC Alpha 21164 processor and 128 Mb of memory. The interconnection network is a torus.

The simulations solve two PDEs derived from the *Poisson differential equation* in two dimensions, subject to the *Dirichlet boundary conditions*:

$$-\frac{\mathrm{d}^2u}{\mathrm{d}x^2}-\frac{\mathrm{d}^2u}{\mathrm{d}y^2}=f(x,y) \quad \text{in } \Omega=]0,1[\times]0,1[$$
$$u=h(x,y) \quad \text{in } \delta\Omega$$

$$h(x,y)=10 \tag{1}$$

$$h(x,y)=10\cos(2\pi(x-y))\frac{\sinh(2\pi(x+y+2))}{\sinh(8\pi)} \tag{2}$$

where $f(x, y) = 0$ and (1) and (2) are two different boundary conditions. With respect to other PDEs, such as the Navier–Stokes one, this equation is a more significant test for a parallel implementation because the ratio between computational work and parallel overhead is among the lowest ones.

Fig. 1 compares the remote data collecting techniques. We plot the overall amount of data exchanged for request/answer (req/ans), for fault prevention (fp) and for informed fault prevention (ifp). In both problems, the informed fault prevention strategy is more convenient because the number of communications of fault prevention and of informed fault prevention

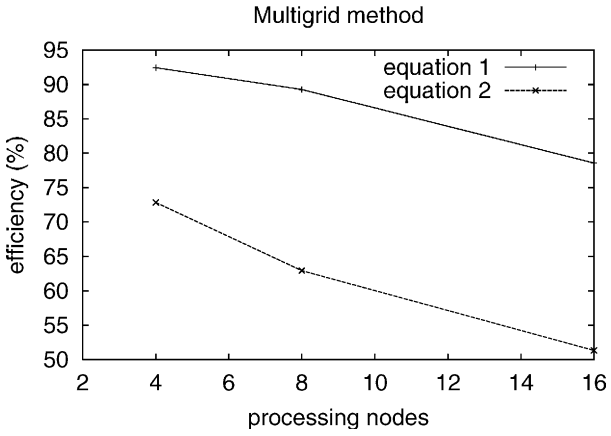

Fig. 2. Efficiency for problems with fixed data dimension.

are, respectively, less than 61 and 52% than those of request/answer. Fig. 2 shows the efficiency of our implementation for the two PDEs, for a fixed number of initial points, 2^{14}, the same maximum grid level, 12, and a variable number of p-nodes. These simulations adopt informed fault prevention. Communication merging has been exploited and each message includes the properties of 15 points. The low efficiency achieved in the second problem is due to a highly irregular grid hierarchy. However, even in the worst case, our solution achieves an efficiency larger than 50% even on 16 p-nodes.

6. Conclusions

This paper has presented a methodology for the parallelization of irregular problems based upon the hierarchical structuring of the domain, a load balancing strategy based upon space filling curves and a technique, informed fault prevention, that reduces the communication overhead. Our experimental results in the case of the AMM together with those of [1] show that this approach achieves good performances on high parallel distributed memory architectures. We are currently defining a package based upon the methodology to simplify the development of parallel solutions to irregular problems.

References

[1] F. Baiardi, P. Becuzzi, P. Mori, M. Paoli, Load balancing and locality in hierarchical *N*-body algorithms on distributed memory architectures, in: Lecture Notes in Computer Science, Vol. 1401, Springer, Berlin, 1998, pp. 284–293.
[2] J.E. Barnes, P. Hut, A hierarchical O($n \log n$) force calculation algorithm, Nature 324 (1986) 446–449.
[3] P. Bastian, S. Lang, K. Eckstein, Parallel adaptive multigrid methods in plane linear elasticity problems, in: Numerical Linear Algebra with Applications, Vol. 4, No. 3, Wiley, New York, 1997, pp. 153–176.
[4] P. Bastian, G. Wittum, Adaptive multigrid methods: the UG concept, in: Adaptive Methods — Algorithms, Theory and Applications, Notes on Numerical Fluid Mechanics, Vol. 46, Vieweg, Braunschweig, 1994, pp. 17–37.
[5] M. Berger, J. Oliger, Adaptive mesh refinement for hyperbolic partial differential equations, J. Comput. Phys. 53 (1984) 484–512.
[6] W. Briggs, A Multigrid Tutorial, SIAM, Philadelphia, PA, 1987.
[7] P. Hanrahan, D. Salzman, L. Aupperle, A rapid hierarchical radiosity algorithm, Comput. Graphics 25 (4) (1991) 197–206.
[8] J.R. Pilkington, S.B. Baden, Dynamic partitioning of non-uniform structured workloads with space filling curves, IEEE Trans. Parall. Distr. 7 (3) (1996) 288–299.
[9] S.S. Mukherjee, S.D. Sharma, M.D. Hill, J.R. Larus, A. Rogers, J. Saltz, Efficient support for irregular applications on distributed-memory machines, ACM SIGPLAN Notices 30 (80) (1995) 68–79.
[10] A. Sohn, R. Biswas, H.D. Simon, A dynamic load balancing framework for unstructured adaptive computations on distributed memory multiprocessors, in: Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures, SIGARCH, ACM, New York, 1996, pp. 189–192.

Fabrizio Baiardi is currently an Associate Professor at the Dipartimento di Informatica, Università di Pisa, Italy. His main research interests are in highly parallel systems and network security.

Sarah Chiti received the graduation in computer science in 2000 with a Master Thesis on AMMs on highly parallel architectures.

Paolo Mori is currently a PhD student in computer science at the Dipartimento di Informatica, Università di Pisa since 1999. His research interests include the study of irregular adaptive algorithms on highly parallel architectures.

Laura Ricci is currently an Assistant Professor at the Dipartimento di Informatica, Università di Pisa. Her research interests include the design of abstract interpretation based tools for parallel systems, the parallelization of irregular adaptive applications and the study of methodologies and tools for teaching concurrency.