Adaptive Multigrid Methods in MPI*

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

Dipartimento di Informatica, Universitá di Pisa
Corso Italia 40, 56125 - PISA (ITALIA)
{baiardi, chiti, mori, ricci}@di.unipi.it

Abstract. Adaptive multigrid methods solve partial differential equations through a discrete representation of the domain that introduces more points in those zones where the equation behavior is highly irregular. The distribution of the points changes at run time in a way that cannot be foreseen in advance. We propose a methodology to develop a highly parallel solution based upon a load balancing strategy that respects the locality property of adaptive multigrid method, where the value of a point p depends on the points that are "close" to p according to a neighborhood stencil. We also describe the update of the mapping at run time to recover an unbalancing, together with strategies to acquire data mapped onto other processing nodes. A MPI implementation is presented together with some experimental results.

1 Introduction

Multigrid methods are iterative methods based upon multilevel paradigms to solve partial differential equations in two or more dimensions. Combined with the most common discretization techniques, they are among the fastest and most general methods to solve partial differential equations [6, 7]. Moreover, they do not require particular properties of the equation, such as the symmetry or the separability and are applied to problems in distinct scientific fields [4, 5, 12].

The adaptive version of multigrid methods, AMM, discretizes the domain at run time by increasing the number of the points in those zones where the behavior of the equation is highly irregular. Hence, the distribution of the points in the domain is not uniform and not foreseeable.

Since the domain usually includes a large number of points, the adoption of a parallel architecture is mandatory. We have defined in [1, 2, 3] a parallelization methodology to develop applications to solve irregular problems on distributed memory parallel architectures. This paper describes the application of this methodology to develop a MPI implementation of the AMM. Sect. 2 describes the main features of AMM, sect. 3 shows the MPI implementation resulting from applying our data mapping technique to the AMM. Sect. 4 describes the technique to gather information mapped onto other processing nodes and the problems posed by the adoption of MPI collective communications. The experimental results on a Cray T3E are discussed in sect. 5.

* This work has been partially supported by CINECA

J. Dongarra et al. (Eds.): EuroPVM/MPI 2000, LNCS 1908, pp. 80–87, 2000.
© Springer-Verlag Berlin Heidelberg 2000

2 Adaptive Multigrid Methods

An AMM discretizes the domain through a hierarchy of grids built during the computation, according to the considered equation. In the following, we adopt the finite difference discretization method. For sake of simplicity, we assume that the domain belongs to a space with two dimensions and each grid partitions the domain, or some parts of it, into a set of squares. The values of the equation are computed in the corners of each square. We denote by $g(A, l)$ the grid to discretize a subdomain A at level l. To improve the accuracy of the discretization provided by $g(A, l)$, a finer grid, $g(A, l+1)$, that is obtained by halving the sides of each square of $g(A, l)$, is introduced. In this way, at run time, finer and finer grids are added till the desidered accuracy has been reached. Even if, in practice, the first k levels of the hierarchy are built in advance, to simplify the description of our methodology, we assume that the initial grid is one square, i.e. $k = 0$.

The AMM iteratively apply a set of operators on each grid in a predefined order, the V-cycle, until the solution has been computed. The V-cycle includes two phases: a descending one, that considers the grids from the highest level to the lowest one, and an ascending one, that considers the grids in the reverse order. Two versions of the V-cycle exist: the additive and the multiplicative; we adopt the additive one and briefly describe the involved operators [7]. The *smoothing operator* usually consists of some iterations either of the Gauss-Seidel method or the Jacobi one to improve the current solution on each grid. The *restriction operator* maps the current solution on $g(A, l)$ onto $g(A, l-1)$. The value of each point on $g(A, l-1)$ is a weighted average of the values of its neighbors on $g(A, l)$. The *prolongation operator* maps the current solution on $g(A, l)$ onto $g(A, l+1)$. If a point exists on both grids, its value is copied. The value of any other point of $g(A, l+1)$ is an interpolation of the values of its neighbors on $g(A, l)$. The *norm operator* evaluates the error of the current solution on each square that has not been further partitioned. The *refinement operator*, if applied to $g(A, l)$ adds a new grid $g(A, l+1)$.

Our methodology represents the grid hierarchy through a quad-tree, the H-Tree. A quad-tree is well suitable to represent the hierarchical relations among the squares and it is intrinsically adaptive. Each node N at level l of the H-Tree, hnode, represents a square, *sq(N)*, of a grid $g(A, l)$ of the hierarchy. The squares associated to the sons of N, if they exist, represent $g(sq(N), l+1)$. Because of the irregularity of the grid hierarchy, the shape of the H-Tree is irregular too. The quad-tree has been adopted in [9], while alternative representations of the grid hierarchy have been adopted in [8, 10]. The multigrid operators are applied to $g(A, l)$ by visiting all the hnodes at level l of the H-Tree. All the operators are applied to $g(A, l)$ before passing to $g(A, l+1)$ or $g(A, l-1)$.

3 Data Mapping and Load Balancing

This section describes the load balancing strategies that, respectively, map each square at any level of the hierarchy onto a processing node, p-node, and update

the mapping during the computation to recover an unbalancing. Both strategies take into account two locality properties of an AMM: the value of a point on *g(A,l)* is function of the values of its neighbors *i)* on the same grid for operators such as smoothing and norm (intra-grid or horizontal locality); *ii)* on *g(A,l+1)* (if it exists) and *g(A,l-1)* for the prolongation, restriction and refinement operators (inter-grid or vertical locality). In the following, we assume that any p-node executes one process and that the *np* p-nodes have been ordered so that two p-nodes close in the interconnection structure of the considered architecture are close in the ordering as well. P_h denotes the process executed by the *h*-th p-node.

Our methodology defines a data mapping in three steps: *i)* determination of the computational load of each square; *ii)* squares ordering; *iii)* order preserving mapping of the squares onto the p-nodes. In the AMM the same load is statically assigned to each square, because the number of operations is the same for each point and does not change at run time. To preserve the locality properties of the AMM, the squares are ordered through a space filling curve [11] built starting from the lowest grid of the hierarchy. After a square S in *g(A,l)*, the curve visits any square in *g(S,d)*, $d > l$, before the next square in *g(A,l)*. The recursive definition of the space filling curves preserves the vertical locality. Moreover, if an appropriate curve is chosen, like the Peano Hilbert or the Morton one, the horizontal locality is partially preserved. A space filling mapping has been adopted in [8, 10] too. Since each square is paired with an hnode, any space filling curve *sf* defines a visit *v(sf)* of the H-Tree that returns an ordered sequence $S(v(sf)) = [N_0, ..., N_{m-1}]$ of hnodes. To preserve the ordering among squares, $S(v(sf))$ is mapped onto the ordered sequence of p-nodes through a blocking strategy. $S(v(sf))$ is partitioned into *np* subsequences of consecutive squares; the *h*-th subsequence includes m/np hnodes and it is assigned to P_h.

The resulting mapping satisfies the **range property**: *if the hnodes* N_i *and* N_{i+j} *are assigned to* P_h*, then all the hnodes in-between* N_i *and* N_{i+j} *in* $S(v(sf))$*, are assigned to* P_h *as well.* This property is fundamental to exploit locality. The domain subset assigned to P_h, Do_h, includes squares at distinct levels of the hierarchy. To avoid replicate computations, for each square in Do_h, P_h applies the operators of the V-cycle to the rightmost downward corner only.

Our methodology assumes that the whole H-Tree cannot be fully replicated in each p-node because of memory constraints. Hence, each p-node stores two subtrees of the H-Tree: the replicated H-Tree and the private H-Tree. The private H-Tree of P_h includes all the hnodes representing squares in Do_h. Even if, in general, the squares in Do_h may correspond to disjoint subtrees of the H-Tree, for sake of simplicity, we assume that Do_h is represented by one connected private H-Tree only. The replicated H-Tree represent the relation among the private H-Trees and the H-Tree. It includes all the hnodes on the paths from the root of the H-Tree to the roots of each private H-Tree, and it is the same for each process. Each hnode N of the private H-Tree records all the data of the rightmost downward corner of *sq(N)*, while each hnode N of the replicated H-Tree records the position of *sq(N)* in the domain and the identifier of the owner process.

To determine, during the computation, where the refinement operator has to introduce finer grids, the processes estimates the current approximation error through the norm operator. This requires the exchange of the local errors among all the processes at the end of a V-cycle and the computation of a global error. Any process estimates its local error and the global error is computed through the MPI_Allreduce primitive. At the end of a V-cycle, to check if the creation of finer grids has leaded to a load unbalance, the processes exchange their workloads through the MPI_Allgather primitive. Then, each process computes *max_unbalance*, the largest difference between *average_load*, the ratio between the overall load and *np*, and the workload of each process. If *max_unbalance* is larger than a tolerance threshold $T > 0$, then each process executes the balancing procedure. T prevents the procedure from being executed to correct a very low unbalance. Let us suppose that the workload of P_h is $average_load + C$, $C > T$, while that of P_k, $h < k$, is $average_load - C$. The balancing procedure cannot map some of the squares in Do_h to Do_k because this violates the range property. Instead, it shifts the squares involving each process P_i in-between P_h and P_k. Let us define $Prec_i$ as the set of processes $[P_0...P_{i-1}]$ that precede P_i and $Succ_i$ as the set of processes $[P_{i+1}...P_{np-1}]$ that follow P_i. Furthermore, $Sbil(Prec_i)$ and $Sbil(Succ_i)$ are, respectively, the global load unbalances of the sets $Prec_i$ and $Succ_i$. If $Sbil(Prec_i) = C > T$, i.e. processes in $Prec_i$ are overloaded, P_i receives from P_{i-1} a segment S of hnodes. If, instead, $Sbil(Prec_i) = C < -T$, P_i sends to P_{i-1} a segment S of hnodes whose overall computational load is as close as possible to C. The same procedure is applied to $Sbil(Succ_i)$ but, in this case, the hnodes are either sent to or received from P_{i+1}. To respect the range property, if $[N_q....N_r]$ is the subsequence of hnodes it has been assigned, P_i sends to P_{i-1} a segment $[N_q....N_s]$, with $q \leq s \leq r$, while it sends to P_{i+1} a segment $[N_t....N_r]$, with $q \leq t \leq r$. P_i communicates with processes P_{i-1} and P_{i+1} only. All communications exploit the synchronous mode with non–blocking send and receive primitives. Non–blocking primitives overlap communication and computation, while the choice of synchronous mode is due to the MPI implementation on the considered parallel architecture, Cray T3E, that provides system buffering. If the MPI standard mode is used, a deadlock may occur if a large amount of pending non–blocking operations has exhausted the system resources. At the end of the load balancing procedure, all the processes exchange, through MPI_Allgather and MPI_Allgatherv, the roots of their private H-Trees to update the replicated H-Tree. Each process, using MPI_Allgather, declares to any other one how many data it is going to send, i.e how many roots it owns. Then, the MPI_Allgatherv implements the exchange of the roots through a buffer allocated according to the number of roots returned by the MPI_Allgather.

4 Collecting Data from Other P-Nodes

Each process P_h applies the multigrid operators, in the order stated by the V-cycle, to the points in Do_h. While in the most of the cases, any information that P_h needs is stored in the private H-Tree, for some points in the border of Do_h,

P_h has to collect the values of points in squares assigned to other processes. We outline the MPI implementation of our remote data collecting procedure, denoted *informed fault prevention*, where processes exchanges remote data before applying the multigrid operators. This procedure allows P_h to receive any data it needs to apply the operator *op* without requesting it to the owner processes, before applying *op*. In this way, when P_i applies *op* to $g(A, l)$, it can visit the H-Tree in any order because it has already collected the data it needs. The advantages of this technique are discussed in [3].

The *informed fault prevention* technique consists of two steps: the *replicated H-Tree extension* step, executed at the beginning of each V-cycle, and the *fault prevention* step, executed before each operator in the V-cycle. Let us define $Bo_h(op, l)$ as the set of the squares S_i in Do_h at level l, such that one of the neighbors of S_i, as defined by the neighborhood relation of *op*, does not belong to Do_h. Furthermore, let $I_h(op, l)$ be the set of squares outside Do_h corresponding to the points whose values are required by P_h to apply *op* to the points in the squares in $Bo_h(op, l)$.

In the replicated H-Tree extension step the processes exchange some informations about their private H-Trees. For each point p_i in $\cup_{op} \cup_l \; Bo_h(op, l)$ such that one of its neighbors belongs to Do_k, P_h sends to P_k the level of the hnode N where *sq(N)* is the smallest square including p_i. This information is sent at the beginning of the V-cycle and it is correct until the end of the V-cycle, when the refinement operator may add finer grids. Since the refinement operator cannot remove a grid, if a load balancing has not been executed, at the beginning of a V-cycle each process sends information on the new grids only.

In a fault prevention step, P_k determines $A_k I_h(op, l) \; \forall h \neq k$, i.e. the squares in Do_k belonging to $I_h(op, l)$ by exploiting both the information in the replicated H-Tree about Do_h and the one received in the replicated H-Tree extension step. Hence, P_k sends to P_h, without any explicit request, the values of the points in $A_k I_h(op, l)$. These values are exchanged just before applying *op* to $g(Do_h, l)$, because they are updated by previous operators in the V-cycle. Notice that P_h can compute $I_h(op, l)$ by simply merging the subsets $A_k I_h(op, l)$ received by its neighbors.

It is worth noticing that, in the case of the refinement operator, $A_k I_h(op, l)$ is approximated. In fact, whether P_h, that owns the square including the point p, needs the value of the point q, in a square owned by P_k, depends not only upon the neighborhood stencil but also upon the value of the points. Since P_h, in the replicated H-Tree extension phase, sends to P_k the depth of the hnodes corresponding to the square in $\cup_{op} \cup_l \; Bo_h(op, l)$, but not the values of the points in these squares, P_k cannot determine exactly $A_k I_h(op, l)$. To guarantee that P_h receives all the data it needs, P_k determines the squares to be sent according to the neighborhood stencil only, and it could send some useless values.

Both steps are implemented through MPI point to point communications. Collective communications, i.e. MPI_Scatter, have not been adopted, because each process usually communicates with a few other processes. This implies the creation, for each process P_h, of one communicator $\mathcal{C}_h$ including any neighbor

of P_h. To this aim, P_h should determine the set of the neighbors of each process in MPI_COMM_WORLD, but it has not enough information to do so. Moreover, MPI_Comm_split cannot be exploited because the communicators associated with two neighbors processes are not disjoint. Furthermore, at the end of a V cycle, because of the refinement operator and of load balancing procedure, the neighbors of P_h changes; this requires the elimination of old communicators and the creation of new ones. Also notice that, since the collective communications are blocking, they have to be properly reordered to prevent the deadlock.

In order to overlap a communication with useful computation, in the fault prevention procedure, each process determines the data to be sent to other processes while is waiting for the data from its neighbors. Moreover, data to be sent to the same process are merged into one message, to reduce the number of communications and the setup overhead. This is a noticeable advantage of informed fault prevention 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 the sets $A_k I_h(op, l)$ from any P_k. While waiting for these data, P_k determines the data to be sent to all the other processes, i.e. $\forall h \neq k$ it computes $A_k I_h(op, l)$. When a predefined amount of data to be sent to the same process has been determined, P_k sends it using an MPI_Issend. Subsequently, P_k checks through an MPI_Test the status of the pending MPI_Irecv. If the communication has been completed, P_k inserts the received data in its replicated H-Tree and it posts another MPI_Irecv. In any case, the computation of the data to be sent goes on. This procedure is iterated until no more data has to be exchanged. After sending $A_k I_h(op, l)$ for any h, P_k sends, through *np-1* MPI_Isend, a syncronization message to any other process and it continues to receive data from them. Since P_k does not know how many data it will receive, it waits for the syncronization message from all the other processes. Then, P_k begins to apply the op to Do_k. A MPI barrier has not been used to syncronize the processes because it is a blocking primitive; hence, after issuing an MPI_Barrier, P_k cannot collect data from other processes. A data exchange among a pair of processes involves variables with distinct datatypes. In order to merge these values in one message, we have compared the adoption of MPI_Pack/ MPI_Unpack against that of derived datatype; both techniques achieve similar execution times.

5 Experimental Results

We present some experimental results of the MPI parallel version of the AMM. The parallel architecture we consider is a Cray T3E; each p-node includes a DEC Alpha EV5 processor and 128Mb of memory. The interconnection network is a torus. MPI primitives are embedded in the C language.

We consider the Poisson problem on the unit square in two dimensions, i.e. the Laplace equation subject to the Dirichlet boundary conditions:

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

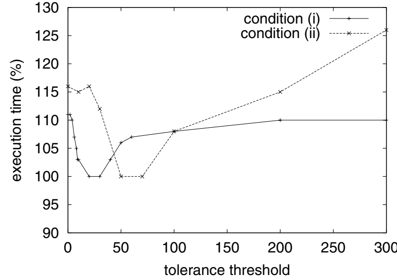

Fig. 1. Load balance

with $f(x, y) = 0$ and two boundary conditions:

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

The solution of the Poisson problem is simpler than those of other equations such as the Navier-Stokes one. Hence, the ratio between computational work and parallel overhead is low and this is a significant test for a parallel implementation. The points distribution in the domain in the case of boundary condition *(ii)* is more irregular than the one of *(i)*. In fact, given the same maximum H-Tree depth, the final number of hnodes of H-Tree *(i)* is three times that of H-Tree *(ii)*

In order to evaluate the effectiveness of the informed fault prevention technique, we have measured that, for both the conditions, the data sent in the informed fault prevention are less than 104% than the data required. As previously explained, due to the refinement operator, this percentage cannot be equal to 100%, but the amount of useless data is less than 4%.

Fig. 1 shows the execution time for different values (in percentage) of the tolerance threshold T. The balancing procedure considerably reduces the execution time; in fact, in the worst case, the execution time of an unbalanced computation may be 25% higher than the optimal one. However, if T is less than the optimal value, no benefit is achieved, because the cost of the balancing procedure is larger than the unbalance recovered. Fig. 1 also shows that the optimal value of T depends upon the points distribution in the domain. In fact, the same value of T results in very different execution times for the two conditions; also the lowest execution times have been achieved using distinct values of T for the two equations.

Figure 2 shows the efficiency of the AMM for the two problems, for a fixed initial grid with $k = 7$, see sect 2, the same maximum grid level, 12, and a variable number of p-nodes. The low efficiency resulting in the second problem is due to an highly irregular grid hierarchy. However, even in the worst case, our solution achieves an efficiency larger than 50% even on 16 p-nodes.

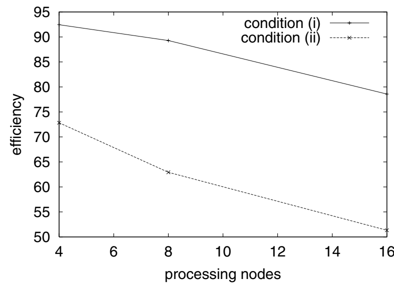

Fig. 2. Efficiency for problems with fixed data dimension

References

[1] F. Baiardi, P. Becuzzi, P. Mori, and M. Paoli. Load balancing and locality in hierarchical N-body algorithms on distributed memory architectures. *Proceedings of HPCN 1998: Lecture Notes in Computer Science*, 1401:284–293, 1998.

[2] F. Baiardi, P. Becuzzi, S.Chiti, P. Mori, and L. Ricci. A hierarchical approach to irregular problems. *Proceedings of Europar 2000: Lecture Notes in Computer Science.*

[3] F. Baiardi, S.Chiti, P. Mori, and L. Ricci. Parallelization of irregular problems based on hierarchical domain representation. *Proceedings of HPCN 2000: Lecture Notes in Computer Science*, 1823:71–80, 2000.

[4] P. Bastian, S. Lang, and K. Eckstein. Parallel adaptive multigrid methods in plane linear elasticity problems. *Numerical linear algebra with appl.*, 4(3):153–176, 1997.

[5] P. Bastian and G. Wittum. Adaptive multigrid methods: The UG concept. In *Adaptive Methods – Algorithms, Theory and Applications*, volume 46 of *Notes on Numerical Fluid Mechanics*, pages 17–37, 1994.

[6] M. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53:484–512, 1984.

[7] W. Briggs. *A multigrid tutorial.* SIAM, 1987.

[8] M. Griebel and G. Zumbusch. Parallel multigrid in an adaptive pde solver based on hashing. In *Proceeding of ParCo 97*, pages 589–599, 1998.

[9] W.F. Mitchell. Refinement tree based partitioning for adaptive grids. In *Proceedings of the 27th Conference on Parallel Processing for Scientific Computing*, pages 587–592, 1995.

[10] M. Parashar and J.C. Browne. On partitioning dynamic adaptive grid hierarchies. In *Proceeding of the 29th annual Hawaii international conference on system sciences*, 1996.

[11] J.R. Pilkington and S.B. Baden. Dynamic partitioning of non–uniform structured workloads with space filling curves. *Transaction on parallel and distributed systems*, 7(3):288–299, 1996.

[12] Y.N. Vorobjev and H.A. Scheraga. A fast adaptive multigrid boundary element method for macromolecular electrostatic computations in a solvent. *Journal of Computational Chemistry*, 18(4):569–583, 1997.