reprinted from Proc. of Euro PVM/MPI 2001: LNCS 2131 pages 304-310

Collecting Remote Data in Irregular Problems with Hierarchical Representation of the Domain

Fabrizio Baiardi, Paolo Mori, and Laura Ricci

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

Abstract. Irregular problems require the computation of some properties of a set of elements irregularly distributed in a domain. These problems satisfy a locality property because the properties of an element e depend upon those of its neighbors according to a problem dependent stencil. This paper proposes two strategies, fault prevention and informed fault prevention, to collect properties of elements mapped onto remote processing nodes that minimize the corresponding overhead. We describe an MPI implementation of informed fault prevention and the experimental results in the case of the adaptive multigrid method.

1 Introduction

Several phenomena, such as the motion of the stars in a galaxy or the illumination of objects in an image, are modeled by time dependent partial differential equations systems, that can be efficiently solved through numerical adaptive iterative algorithms. To apply these algorithms, at least one dimension of the problem, such as the time interval or the space to be analyzed, is discretized. Hence, the phenomenon is modeled as a set of elements in a domain of interest where some properties are computed for each element. The computation satisfies a locality property because the properties of an element depends upon those of a subset of all the other ones, according to a problem dependent neighborhood relation. The main feature of any irregular problem is the *non homogeneous and dynamic* distribution of the elements in the domain. This implies that some subsets of the domain require a larger computational effort than the other ones.

The adoption of parallel architectures for the resolution of these problems is mandatory because of the large number of elements, but this poses the problems of the mapping of the elements onto the processing nodes, p-nodes, and of the communication handling due to the irregularity and the dinamicity of the distribution. Since no information about the element distribution is statically available, a sophisticated run time support is required that is even more important on distributed memory architectures with a sparse interconnection network.

Any parallel implementation of these problems on this class of architectures should define, at least, three strategies that, respectively, *i)* define a load balancing mapping of the elements onto the p-nodes *ii)* update this mapping to recover an unbalancing due to the changes in the distribution and *iii)* collect remote data. These three strategies are built around a data structure that describe the mapping of the elements. In the following we assume that this structure describes a hierarchical decomposition of the domain. While the first two strategies are strictly correlated, the third one is independent from the other ones. In this paper, we focus on this strategy because it has a large influence on the overall efficiency. The strategies that define and update the mapping have been discussed in [1] and [2]. Alternative approaches to irregular problems are LPARX [4], Chaos and Multiblock Parti, [5] and that presented in [6]. However, the study is focused on the data mapping techniques.

The next section briefly describes the data structure that represents the mapping of the elements. Sect. 3 describes three strategies to collect remote data: fault handling, fault prevention and informed fault prevention. Sect. 4 discusses the MPI implementation of the informed fault prevention strategy. Experimental results are presented in the last section.

2 A Hierarchical Representation of the Element Mapping

The *Hierarchical Tree*, *H-Tree*, is the data structure that represents both the distribution of the elements in the domain and their mapping onto the p-nodes, assuming that the domain is an n-dimensional space. A tree has been adopted because it naturally represents the hierarchical relations and it is intrinsically adaptive. The H-Tree defines a multi level hierarchical representation that, at each level, partitions the domain into a set of equal subdomains, or spaces. The root of the H-Tree represents the whole domain, while each other node N, *hnode*, represents either a space produced by the decomposition, *space(N)*, or an element, and it records the corresponding information, see Fig. 1. If *space(N)* has been partitioned, the resulting subspaces are represented by the sons of N. Each element is paired with a space including it. If an element e is paired with *space(N)*, then the hnode L representing e is a son of N and is a leaf of the H-Tree. As the number of elements and their distribution change during the computation, the decomposition of the domain and the H-Tree are updated to represent the current distribution.

We do not discuss the mapping strategies that can be supported by the H-Tree. Here, we only assume that they do not violate the locality property and we denote by $Do(P_k)$ the subdomain including all and only the spaces mapped onto P_k. The mapping of the spaces defines np subsets of the H-Tree, one for each p-node. The subset assigned to P_h is the private H-Tree of P_h that includes an hnode N iff *space(N)* $\in Do(P_h)$. Notice that, in general, the hnodes assigned to a p-node could define distinct subtrees of the H-Tree but, for the sake of simplicity, we assume that they belong to just one subtree. A further subset of the H-Tree, the replicated H-Tree, is defined by the mapping strategy. This tree includes all the hnodes in the path from the root of the H-Tree to the root of any private H-Tree and is replicated in all the p-nodes. Each leaf L of the replicated H-Tree points to the p-node storing the private H-Tree rooted in L .

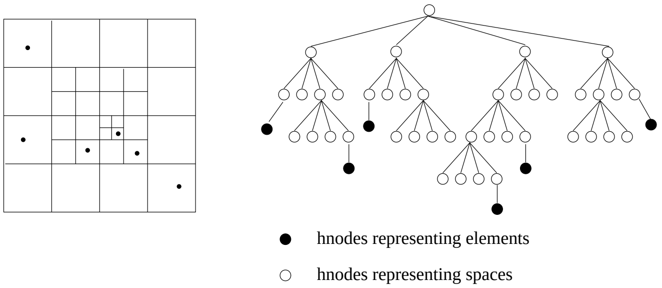

Fig. 1. Domain Decomposition and H-Tree

3 Remote Data Collection Techniques

At each iteration of the algorithm, P_h updates the properties of each element e_i in $Do(P_h)$ by applying one or more operators to the current properties of e_i and to those of its neighbors. Hence, P_h needs the properties of the neighbors of e_i, as defined by the neighborhood stencil of each operator. The abstraction level of these properties depends upon several problem dependent features, such as the operator that is applied or the value of the properties of the element. For this reason, in the following, we say that a p-node needs the properties of some spaces to compute the property of an element. Even if the mapping strategy respects the locality property, some of the neighbors of e_i have been allocated onto other p-nodes. Hence, the definition of a strategy to collect remote spaces, i.e. spaces allocated onto other p-nodes, is required.

However, a fundamental problem is that a p-node cannot know in advance where the spaces it needs have been mapped. Moreover, because of the dynamic mapping, even if, at each iteration, a p-node needs the same spaces, they may be mapped onto distinct p-nodes at distinct iterations. If locality is preserved, most of the neighbors of each element e are allocated onto the same p-node of e. This implies that: *i)* most of the spaces to be collected are allocated on the same p-node, and *ii)* the intersection between the set of the neighbors of two neighbors elements is large. Hence, a remote space should be reused to update the properties of several elements mapped onto the p-node (reuse property).

3.1 Fault Handling

The simplest strategy to collect remote spaces is *fault handling*. Each p-node, while computing the properties of its elements, interacts with the other p-nodes as soon as it needs some spaces. To compute the properties of each element e_i in $Do(P_h)$, P_h visits its private H-Tree. If P_h determines that it needs a space A, assigned to another p-node, it suspends the computation of the properties of e_i, and it requests the properties of A to the owner, determined by accessing the replicated H-Tree. The computation can be resumed only when P_h has received all the missing properties. These properties should be cached in the local memory of P_h because they may be used to compute the properties of other elements in $Do(P_h)$ by the current operator. The time inbetween the request of the remote spaces and the reply may be large. If, during this time, P_h is idle, a low efficiency will results. Since the computation of the properties of e_i is independent of that of each other element e_j, the waiting time can be overlapped with the computation of the properties of another element e_j, according to an excess parallelism strategy. If the computatation of e_j is suspended too, and all the spaces required for e_i have been received, then the computation of e_i can be resumed, otherwise the computation of another element is started.

This strategy is very general but, as a counterpart, it requires two communications to collect the properties of a remote space, one for the request and one to transmit the properties. Moreover, it is not trivial to optimize the communications. Since the size of the data sent in either a request or a reply is small, several messages for the same process could be merged into a single message, but this delays some messages. Hence, message merging increases the time between a request and the corresponding reply. In turns, this implies that a larger number of elements has to be computed in parallel by P_h and this may result in a large overhead. Furthermore, to fully exploit the reuse property, only one request for each space should be sent. This requires that each request for a space should be checked against those that have already been sent to the same p-node.

3.2 Fault Prevention

The *fault prevention* strategy generalizes the data exchange in the data parallel programming model where the compiler determines which spaces are required by each p-node and the run time support executes the communications before starting a new iteration. Hence, a p-node does not explicitly request the spaces it needs because the compiler can exactly determine the spaces to be exchanged.

In the fault prevention strategy, for each operator, the proper spaces are exchanged among the p-nodes before executing the operator. In this way, when P_h computes the properties of its elements, all the spaces it requires are available in its local memory. Taking into account the dynamic element mapping, P_k computes, before applying an operator, which of its spaces are required by P_h, $h \neq k$, by applying to each space A in $Do(P_k)$ the inverse of the operator neighborhood stencil. If A is a neighbor of any element in $Do(P_h)$, then P_k sends the properties of A to P_h. This fully exploits the reuse property because each space is sent just once.

Fault prevention assumes that all the p-nodes share some informations about the domain decomposition that is recorded in the replicated H-Tree. The main disavantage of this strategy is that the replicated H-Tree records a partial information only, hence, P_k approximates the set of spaces that P_h requires. However, P_h can execute its computation only if the approximation is *safe*, i.e. P_h receives all the spaces it needs. Safeness requires that, if the replicated H-Tree does not include enough information to determine whether A belongs to the neighborhood stencil of an element in $Do(P_h)$, P_k includes A in the data to be sent. If the approximation is not accurate, most of the spaces sent to P_h are useless.

3.3 Informed Fault Prevention

If the information in the replicated H-Tree does not allow a p-node to compute an accurate approximation, we propose an improved version of fault prevention, *informed fault prevention.* According to this strategy, the p-nodes exchange some information about their private H-Trees in a separate phase, *replicated H-Tree extension* phase, before the fault prevention one. During the fault prevention phase, each process exploits the information received in the replicated H-Tree extention phase to determine the spaces it has to send. Due to the locality property, the information sent by P_h to P_k in the replicated H-Tree extention phase usually describes the distribution of the elements on the boundary of $Do(P_h)$ that intersect that of $Do(P_k)$. In particular, the information sent by P_h to P_k is related to all the elements e_i in $Do(P_h)$ that could have at least one neighbor in $Do(P_k)$. To reduce the overhead of this phase, only the subset of properties of e_i useful to determine its neighborhod stencil are trasmitted. Moreover, this information is sent from P_h to P_k in the first replicated H-Tree extension phase after the creation of e_i and an invalidation message is sent when e_i is destroyed, remapped or when the subset of properties have been updated. When P_k receives the creation message, it stores the properties in its local memory and it will use this information until it receives an invalidation message.

With respect to fault prevention, the computation of the spaces to be sent is simplified, because each process P_k, for each element e_i received in the information phase, determines which of its spaces belongs to the neighborhood stencil of e_i and send them to the owner of e_i. Hence, while the fault prevention has to visit all the private H-Trees, the informed fault prevention considers the elements received in the replicated H-Tree extention phase only, that are considerably less than those in the private H-Trees.

An advantage of both fault and informed fault prevention is that they concentrate the communications in two small sections of the algorithm, the replicated H-Tree extension phase and the fault prevention phase. This implies that the trasmission of a set of a data can be delayed because they are not used immediatly. Hence, a group of data to be sent to the same p-node can be merged into one message. This strategy can be fundamental in cluster of workstations, where little communication support is provided and the time to set up a communication cannot be negletted. A further advantage of fault prevention strategies is that they preserve the sequential code. As a matter of fact, after the fault prevention phase, all the spaces required by the computation are available on each p-node and the sequential code can be executed.

An example of application of the methodology to adaptive multigrid method is showed in fig. 2. In adaptive multigrid method, [3], the final result is basically computed through the application of five operators: restriction, smoothing, prolongation, norm and refinement. The restriction and the prolongation operators are separately applied on each level of the hierarchical representation of the domain, while the smoothing operator is applied on the whole domain. The refinement operator is the only one that modifies the domain by adding new

```
data mapping and replicate and private h-trees creation

while (not global error < threshould)  {
   replicated h-tree extention(union_all_stencils, all_levels)

   for level from max_level downto min_level  {
     fault prevention(restriction_stencil, level)
     restriction(level)  }

   fault prevention(smoothing_stencil, all_levels)
   smoothing(all_levels)

   for level from min_level to max_level  {
     fault prevention(prolongation_stencil, level)
     prolongation(level)  }

   fault prevention(norm_stencil, all_levels)
   norm(all_levels)
   refinement(all_levels)

   data mapping and replicate and private h-trees update  }
```

Fig. 2. Example of Application of the Methodology

elements in those subdomains where the approximation error is too large. Hence the replicated H-Tree extention phase can be executed at the begining of each iteration only, and the values collected are valid until the end of the iteration, when the refinement operator is applied. The fault prevention phase, instead, has to be executed before each operator, because each operator updates the values.

4 MPI Implementation of Informed Fault Prevention

In the following, we focus on informed fault prevention, because this strategy is the most complex one, and consider how it can be implemented through the MPI primitives to manage the data exchange among the p-nodes.

Both in the replicated H-Tree extention phase and in the fault prevention one, each process determines the spaces to be sent while receiving the spaces from the other p-nodes. In this way, the latency of communications is overlapped with some useful computation. P_h issues an MPI_Irecv from each other p-node and starts the computation of the spaces to be trasmitted. The handles returned by each MPI_Irecv are recorded in the request array, in the position corresponding to the rank of the sender process. As soon as P_h determines that a space is to be sent, it issues an MPI_Isend, it records the corresponding handle in a buffer and it polls, through an MPI_Testany invoked on the request array, whether a new message has been received. If a message has been received from P_k, the corresponding hnode is inserted in the replicated H-Tree and a new MPI_Irecv from

P_k is posted. The handle returned by MPI_Irecv is recorded in the k^{th} position of the request array. The polling solution avoids any overhead of interrupt based solutions. Periodically, each p-node issues an MPI_testall on the buffer containing the send handles and frees the positions paired with the partners with which the exchange have been completed.

A process synchronization is executed before applying of each operator, to ensure that all the processes have terminated the data exchange. Since the number of messages to be exchanged is not known in advance, a barrier cannot be adopted to synchronize processes because the MPI_barrier is a blocking primitive and, once issued, only messages related to the barrier can be received. Hence, each process, after sending all the data, broadcasts a termination message to each other process through *np* point to point MPI_Isend primitives and waits for the corresponding termination messages from the other processes. In this way, termination messages are interleaved with the data ones.

If the properties of a space to be exchanged among the p-nodes have different data types, MPI derived data types may be adopted. Moreover, since the set of properties exchanged in the replicated H-Tree extention phase are different from those exchanged in the fault prevention phase, distinct derived data types have to defined. To reduce the communication overhead, message merging has been adopted. Hence, each process P_h defines a buffers of b positions for each of its communication partners and it stores in the k^{th} buffer any data to be sent to P_k to immediatly continue the computation of the data to be sent. As soon as the buffer paired with P_k is full, the content of the buffer is sent to P_k through an MPI_Isend . The value of b is choosen according to the features of the adopted architecture.

5 Experimental Results

In order to evaluate the performances and the effectiveness of the methodology and, in particular, of the informed fault prevention strategy, we have implemented a parallel version of an adaptive multigrid method, using the MPI primitives embedded in the C language.

The parallel architecture we consider is a cluster of workstations. Each workstation is a PC with an Intel Pentium II CPU (266 MHz) and 128 Mbyte of local memory. The interconnection network is a 100Mbit Fast Ethernet switch and the operative system running on each p-node is LINUX 2.2.13.

The adaptive multigrid method we consider solves an highly irregular partial differential equation derived from the Poisson problem. In fact, at some iterations of the computation, a very large number of elements is created in few subdomains. Hence, this is a good test for a parallel implementation.

Figure 3 shows the efficiency of our implementation for a variable number of p-nodes where the informed fault prevention strategy is adopted. The size of the problem, i.e. the initial number of elements in the domain, is the same for all the executions. The optimal size of the buffer used to implement the

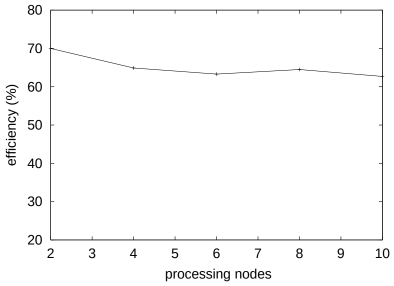

Fig. 3. Efficiency for Adaptive Multigrid Method

message merging strategy is 50, because in a cluster of workstations the cost of a communication is dominated by the time to setup the communication.

Figure 3 shows that our implementation arises an efficiency larger than 60%, even on 10 p-nodes. Moreover, from the point of view of scalability, we observe that the value of the resulting efficiency is almost independent of the number of p-nodes. The main reason of this efficiency value is that, even if the replicated H-Tree extention phase and the fault prevention phase take less than 20% of the total execution time, more than 10% of the time is lost due to an unbalanced load distribution in the refinement operator. An improovement of the methodolgy where, if necessary, two updates of the domain partitioning are executed in the same iteration, is under development.

References

1. Baiardi, F., Chiti, S., Mori, P., Ricci, L.: Integrating Load Balancing and Locality in the Parallelization of Irregular Problems. In: Future Generation Computer Systems, Vol. 17. Elsevier Science (2001) 969–975
2. Baiardi, F., Chiti, S., Mori, P., Ricci, L.: A Hierarchical Approach to Irregular Problems. In: Proc. of Europar 2000: LNCS, Vol. 1900. (2000) 218–222
3. Briggs, W.: A multigrid tutorial. SIAM (1987)
4. Fink, J.S., Baden, S.B., Kohn, S.R.: Efficient run-time support for irregular block-structured applications. In: Journal of Parallel and Distributed Computing, Vol. 50(1) (1998) 61-82
5. Mukherjee, S.S., Sharma, S.D., Hill, M.D., Larus, J.R., Rogers, A., Saltz, J.: Efficient support for irregular applications on distributed-memory machines. In ACM SIGPLAN Notices, Vol. 30(80) (1995) 68–79.
6. Sohn, A., Biswas, R., Simon, H.D.: A Dynamic Load Balancing Framework for Unstructured Adaptive Computations on Distributed Memory Multiprocessors. In: Proc. 8th Annual ACM Symposium on Parallel Algorithms and Architectures, (SIGARCH, ACM, 1996) 189–192.