reprinted from Proceedings of PARA 2002: LNCS 2367 pages 185-194

PIT: A Library for the Parallelization of Irregular Problems

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. A problem is irregular if its solution requires the computation of some properties for each of a set of elements irregularly distributed in a domain of interest. These problems satisfy a locality property because the properties of an element depend upon those of a few other elements, its neighbors, according to a dynamic, problem dependent stencil. The development of parallel algorithms for irregular problems on distributed memory architectures is not trivial, because the irregularity and the dinamicity of the distribution of the elements in the domain require complex strategies to manage the mapping of elements onto the processing nodes and to implement the processing nodes cooperation. This paper introduces PIT, a library to simplify the parallelization of irregular problems. The key assumption underlying the definition of PIT is that both the sequential and the parallel version of the application are structured in terms of operations on a tree that describes the distribution of the elements in the domain. In the parallel version, the tree is handled in parallel through the functions supplied by PIT in a way that is transparent to the user and that preserves most of the sequential code.

1 Introduction

Several physical phenomena, such as the motion of the stars in a galaxy or the illumination of objects in an scene, are modelled by time dependent partial differential equations systems that are usually solved through adaptive iterative algorithms. An iterative algorithm computes the final result through a sequence of approximations, each produced by updating the previous one. The domain of an irregular problem consists of a set of elements distributed in an irregular and dynamic way in a space of interest. Each element is characterized by its properties, as defined in the specific irregular problem. For instance, if the Barnes Hut method for the nbody problem is applied to the motion of stars, each element corresponds to a star and its properties are mass, position in the space and speed vector. The properties of an element are updated by computing its interactions with a set of other elements close to the considered one (*neighbors*). The rule to determine the set of neighbors of an element (*neighborhood stencil*) depends upon the specific problem, but, in general, the number of neighbors is different for distinct elements and it changes during the computation. The number of

elements changes as well. For instance, it increases to improve the accuracy of the final solution. Hence, due to the irregularity of the distribution of the elements and of the neighborhood stencil, the computation of the evolution of some subsets of the domain requires a larger computational effort than others and these subsets change as the computation goes on.

The development of parallel applications for the solution of irregular problems is not trivial, because of the irregular and dynamic distribution of the elements in the domain, that requires complex strategies to map the elements onto the processing nodes, *p-nodes*, and for the communication management. Since no useful information about the distribution of the elements can be deduced by a static analysis or by program profiling, sophisticated run time mapping strategies have to be adopted to produce a parallel code that can achieve satisfactory performance values. In particular, load balancing is critical to improve these values.

This paper presents PIT, *Parallel Irregular Trees*, a library to parallelize irregular problems derived from the methodology to structure a parallel application for irregular problems on distributed memory architectures presented and evaluated in [1–4]. The aim of PIT is to provide a simple and complete tool for the development of parallel solutions to irregular problems while preserving most of the code developed for the sequential application. The key point of the PIT approach is that both the sequential and the parallel version of the application may be structured in terms of operations on a tree that describes the distribution of the elements in the domain. In the parallel version, the tree is distributed among the local memories of the p-nodes of the architecture and is handled in parallel in a transparent way through the PIT functions. In the current version, PIT functions are implemented through C and MPI, however other languages can be exploited.

Alternative approaches to irregular problems are LPARX [5], Chaos and Multiblock Parti, [6] and that presented in [8]. However, these approaches are focused on the data mapping techniques.

In the following, we describe the methodology and the strategies underlying the definition of PIT. Sect. 2 describes the methodology to solve irregular problems, and how the sequential code can be transformed in the parallel one using the PIT functions. Sect. 3, 4, 5 describe the details of the main functions provided by PIT, the H-Tree_creation, the H-Tree_completion and the H-Tree_update.

Sect. 6 shows that the user knowledge of the specific irregular problem can be exploited to combine the PIT functions to produce a more efficient parallel code. Section 7 draws the conclusions.

2 PIT Parallelization Strategy

PIT is a library to develop parallel applications to solve irregular problems. Its main aim is to simplify the development of such algorithms on distributed memory architecture so that it can be exploited by users that are not acquainted with parallel programming or with the problems posed by distributed memory

systems. The functions of the library have been defined starting from a general methodology for the development of parallel applications to solve irregular problems that is described in [1–4]. In the following, at first we describe how to develop the sequential code according to our methodology so that it can be easily transformed through the functions supplied by PIT. Then, we presents the parallelization methodology and how the parallel code can be derived from the sequential one by properly exploiting the functions of PIT.

2.1 Sequential Code

In our approach, the distribution of the elements in the domain is represented through a tree, the *H-Tree*. The domain is recursively partitioned into equal spaces until a problem dependent condition on each resulting space is satisfied. The resulting space hierarchy is described through the H-Tree. Each node of the H-Tree, *h-node*, represents a space of the hierarchy; the root of the H-Tree represents the whole domain, the sons of the root the spaces derived by the first partition of the domain, and so on. The leaves of the H-Tree represents spaces that have not been partitioned. For the sake of simplicity, in the following, we suppose that each h-node is paired with one element even if, in some problems, some spaces may not be paired with elements. For instance, in the Barnes Hut method for the nbody problem, only the leaves of the H-Tree are paired with any element; the other h-nodes represent sets of elements used to compute approximated interactions. The interactions between the elements are computed by iteratively applying a sequence of operators to the H-Tree, *operator_1, .., operator_n*, until the solution has been computed. Each operator is implemented through a visit of the H-Tree. For each h-node visited, the set of neighbors of the corresponding element is determined through the neighborhood stencil of the operator, and the properties of the element are updated by applying the function corresponding to the operator. A very general scheme of the sequential code developed according to this approach is the following:

```
irregular_problem(elements)
    root = tree_creation(elements)
    while (not solution_computed)
        operator_1(root)
        ....
        operator_n(root)
    endwhile

operator_ j(node)
    neighbor_list =stencil_j(node)
    node_properties = rule_j(node,neighbor_list)
    for i from 0 to NSONS
        if exists(son(i)) then
            operator_j(son(i))
    endfor
```

In the previous code, stencil_j is the function that, given a h-node returns the list of its neighbors, and rule_j is the specific problem function that, given a h-node and the list of its neighbors, returns the updated values of the properties of the element paired with the h-node.

2.2 Parallelization Methodology

The considered parallelization methodology defines strategies to map the elements onto the p-nodes, to collect the properties of the elements mapped onto other p-nodes and to update the elements mapping during the computation to balance the computational load amog the p-nodes.

Obviously, to implement such strategies, the user has to be familiar with parallel programming techniques. In particular, the user has to understand the concepts of data mapping, local data, remote data and so on.

The PIT approach, instead, is based upon the H-Tree that describes the elements distribution in the domain. The strategies of the methodology are implemented by PIT as operation on the H-Tree. The parallel programming paradigm implemented by PIT is SPMD, where each p-node executes the same code on a distinct subset of the H-Tree.

¿From the user point of view, each PIT function implements an operation on the H-Tree, and the H-Tree is handled in parallel in a *transparent way* by these functions. For instance, the mapping of the elements onto the p-nodes is implemented by the *H-Tree_creation* function, the update of the mapping to balance the computational load among the p-nodes is implemented by the H-Tree_update function and so on. The PIT functions implement all the strategies of our methodology so that a parallel solution can be implemented by properly invoking the functions of the package only, and no additional parallel code has to be developed by the user. Any process synchronization or communication is completely solved within the PIT function. Hence, in the simplest solution, the user has only to choose the number of p-nodes to be exploited. PIT defines alternative APIes for different kinds of users. The simple API includes just a few functions that implement the strategies of the methodology in the most general way. The invocations to this API may be inserted into the sequential code according to a standard skeleton. The advanced API is addressed to users that are more familiar with parallel programming techniques. To define the advanced API, the strategies of the methodology have been decomposed into several steps, each implemented by a distinct function. In this way, the functions can be composed in the most suitable way for the considered problem. Obviously, the use of the advanced API results in better performances of the parallel application. Moreover, in the same parallel code both functions from the standard API and from the advanced one can be applied.

2.3 Standard Parallel Code

This section describes how the functions of PIT may be applied to transform the sequential version of an application into a parallel one. The user has to structure

the sequential application to solve the specific irregular problems as a sequence of operations on the H-Tree as shown in the previous section. Then, the parallel version of the application can be defined by one of the two following strategies: *i)* by choosing the simplest API and inserting the sequential code into a standard skeleton *ii)* by inserting invocations to the advanced API into the sequential code in the most suitable way for the specific irregular problem.

The following code is obtained by applying the standard skeleton to the sequential code showed in section 2:

```
irregular_problem(elements)
    root = Tree_creation(elements)
    while (not solution_computed)
        Htree_completion(root,stencil_1)
        operator_1(root)
        Htree_update(root)
        ....
        Htree_completion(root,stencil_n)
        operator_n(root)
        Htree_update(root)
    endwhile
```

As described in the following, the execution of the same PIT function is synchronized, so that it terminates simultaneously in distinct p-nodes. The first PIT function to be invoked in the parallel code is the *H-Tree_creation* one, to build the H-Tree shared among all the functions. This function returns an handle to the root of the H-Tree that is used by all the functions to refer the H-Tree. As described in the following, this function returns a distinct H-tree to each p-node, according to the SPMD paradigm. Since the H-Tree is distributed across the p-nodes, while the code developed in the sequential version for each operator assumes that all the data it needs are allocated into the local memory of the invoking process, the H-Tree_completion function has to be invoked before each operator. This function, through the neighborhood stencil, collects all the updated values of properties of the h-nodes mapped onto remote p-nodes that are required by the operator. The H-Tree_update function, instead, updates the H-Tree mapping when the elements distribution in the domain changes or the computational load assigned to the p-nodes is no longer balanced. All these functions are described in details in the following sections.

3 H-Tree Creation

As previously stated, the PIT library is based upon a multi level representation of the domain described through the H-Tree. In general, the H-Tree is too large to be stored into the local memory of a single p-node. Hence, it is partitioned and distributed among the p-nodes. Each p-node stores a distinct subset of the H-Tree, the *private H-Tree*. There are no intersections between the private H-Trees of two distinct p-nodes, and the union of all the private H-Trees is the H-Tree. the

private H-trees should be defined so that most of the neighbours of an element belong to the same private H-tree of the element itself. In order to enable each p-node to deduce the allocation of the elements mapped onto another p-node, a further subset of the H-Tree has been defined, the *replicated H-Tree*. As implied by its name, this tree is replicated in each p-node, and includes all the h-nodes on the path from the root of the H-Tree to the root of each private H-Tree. Each leaf of the replicated H-Tree records the identifier of the p-node where the private H-Tree rooted in that leaf has been mapped. The replicated H-Tree is partially overlapped with some private H-Trees and it defines the smallest amount of information allowing any p-node to determine the mapping of any h-node.

The private H-Tree and the replicated H-Tree are built by the H-Tree_creation function. Since not only the H-Tree, but even all the elements of the domain could not be stored in the local memory of one p-node, the H-Tree_creation function adopts a distributed strategy. To create the H-Tree, each p-node reads a subset of the elements of the problem domain and it recursively partitions the domain for a fixed number of times, to produce a set of equal spaces. Then, each p-node determines, for each space S, the number of elements it has read that belong to S, and exchange this number with all the other p-nodes, to determine the total number of elements included in each space. The spaces are ordered through a space filling curve, such as the Peano Hilbert one [7]. The resulting sequence of spaces is partitioned into np contiguous sub sequences, where np is the number of p-nodes. This guarantees that an element and most of its neighbours are assigned to the same private H-tree. To balance the load, we also require that the computational loads due to the elements in distinct subsequences differ for, at most, a predefined user defined constant. In this phase, we assume that two any elements have the same computational load. This phase cannot determine a balanced assignment of the spaces if at least one space includes too many elements. In this case, each space is partitioned one more time, and the procedure is iterated. When a balanced element partition has been computed, a communication phase begins where the p-nodes exchanges the elements to satisfy the new mapping, i.e. each p-node P sends to another p-node Q all the elements it has read but that belong to a space mapped onto Q. After this exchange has been completed, each p-node creates its private H-Trees and sends the roots of these trees to the other p-nodes. Since each p-node receives the roots of the private H-Trees of each other p-node, it can create the replicated H-Tree too. All the previous phases are separated by a barrier synchronization among the p-nodes.

¿From the user point of view, the H-Tree_creation function is the first one to be invoked. For the sake of simplicity, we suppose that one private H-Tree only is assigned to each p-node. In this case, the H-Tree_creation function returns the handle to the root of the private H-Tree. The user does not know how the H-Tree has been partitioned, however, the handle to the root make it possible to visit the H-Tree in the same way as a local H-Tree. Obviously, each p-node cannot access the whole H-Tree, but only the h-nodes it has been assigned. However, by

properly invoking the H-Tree_completion functions, each p-node can collect into its local memory the h-nodes it needs to compute the interaction of its elements, as described in the following.

4 H-tree Completion

Since the H-Tree has been partitioned among the p-nodes by the H-Tree_creation function, some of the neighbors of an element e have not been mapped onto the same p-node where e has been mapped. To compute the properties of e through the same operators of the sequential implementation, all the neighbors of the elements mapped onto a p-node have to be collected in the local memory before visiting the private H-Tree. Hence, before applying the operator, each p-node should receive, from the other p-nodes, some of the subtrees that have been mapped onto these p-nodes. In this way, each p-node, can collect the properties of interest. The subtrees to be sent to other p-nodes can be determined at run time only, through a problem dependent neighborhood relation, because they depend upon the current properties of the elements mapped onto the receiver p-node.

The strategy defined in our methodology for such a remote data collection includes two steps. In the first one, each p-node P determines, through the neighborhood stencil, the set of its elements such that at least one of their neighbors may have been mapped onto another p-node. Each element of this set is sent to the owner of the neighbor element, i.e. to the p-node Q where the neighbor has been mapped. Since this exchange has place only to enable Q to compute the set of its elements whose properties are required by another p-node, P does not send to Q all the properties of these elements, but only those required to apply the neighborhood stencil. In the second step, through the information received in the previous step, each p-node can determine which of its elements are the neighbors of an element e received in the previous step and send their properties to the owner of e. The receiver p-node inserts these nodes into its private H-tre. These hnodes will be replaced the next time that remote data are collected. The strategy has been defined as the composition of two distinct steps because the first one has to be executed each time the set of neighbors of at least an element changes, while the second step is executed each time the updated values of the properties of the elements are needed to apply an operator.

¿From the PIT user point of view, after the collection, the h-nodes paired with the neighbors of an element can be reached through a visit of the H-Tree, as in the sequential code, even if these h-nodes have been mapped onto other p-nodes. As stated by the methodology, these h-nodes are determined by the PIT functions by applying the neighborhood stencil in a dynamic way. Two possible APIes are defined by PIT to interface the user with the functions to collect remote data. The user that is not familiar with parallel programming can invoke a single function that executes both the steps previously described. In this case, the function has to be invoked just before the operator that needs the updated values of the elements. Instead, a more sophisticated user, that can determine which operators

updates the properties required to apply the neighborhood stencil, can exploit two distinct PIT functions, *H-tree_det_neighbors* and *H-tree_exchange_neighbors*, rather than one . These functions implement, respectively, one of the steps previously described. In this case, the H-tree_det_neighbors function is invoked after the H-Tree_creation function and after each operator that updates the properties that determine the neighborhood stencil. The H-tree_exchange_neighbors function, instead, is invoked just before the operator that needs the updated values of the elements. Obviously, the invocation of just one function simplifies the development of the parallel version at the expense of efficiency. The adoption of the two steps function, instead, makes it possible to achieve a better efficiency of the resulting parallel code, because it avoids useless data exchange among the p-nodes.

5 H-tree Update

During the computation, both the number and the distribution of the elements in the domain changes because the operators can create, delete and update elements in the domain. Hence, the mapping of the elements onto the p-nodes has to be updated, because some elements may violate the mapping strategy defined in sect. 3. For instance, in the Barnes Hut method, a star, due to the interactions with all the other stars, may change its position in the space, from the subdomain assigned to a p-node to that mapped onto another p-node. Moreover, these modifications in the domain may affect the computational load paired with each element, and the mapping has to be updated to correct load unbalances as well. The correct mapping of an element can be determined through the replicated H-Tree. To balance the computational load, instead, the methodology defines a strategy to update the mapping through the same space-filling curve used to define the initial mapping of the elements.

¿From the PIT user point of view, two possible interfaces can be used to implement the H-Tree update. The standard function, H-Tree_update, updates the mappings taking into account both the elements that violate the mapping strategy and the load unbalance. The user has to invoke this function each time the distribution of the elements in the domain changes, otherwise an incorrect H-Tree state could result. As a consequence, in the standard skeleton, this function is invoked after the execution of each operator. The advanced interface includes two distinct functions, H-Tree_correction and H-Tree_balance, that, respectively, update the mapping of the elements violating the mapping strategy and correct the load unbalance. While the H-Tree_correction function has to be invoked each time the elements distribution changes, to avoid incorrect H-Tree state, the H-Tree_balance function should be invoked less frequently because, to determine whether the current load distribution is unbalanced, each p-node exchange its current workload with all the other p-nodes. Taking into account the corresponding overhead, no real performance improvement may be achieved by executing this strategy if minor modifications to the domain have occurred only. The user can also choose whether the computational load due to each element has to be

considered fixed and equal for all the elements or it has to be measured as the computation goes on. In the latter case, it can further specified that the load due to an element is determined by the number of times a given operator is applied to the element.

6 PIT's Parallel Code

The parallel code presented in sect. 2.3 has been obtained through the standard skeleton. To describe how the more advanced API can be exploited, we suppose to know that only *operator_n* modifies the H-tree.The other operators only update some problem dependent properties of the elements, that do not affect the neighborhood relations. Starting from this information, we show how PIT functions can be composed in a more suitable way for the specific irregular problem, in order to improve the performances of the resulting parallel code.

```
irregular_problem(elements)
    root = Tree_creation(elements)
    Htree_det_neighbors(root, stencil_1+....stencil_n)
    while (not solution_computed)
        Htree_exchange_neighbors(root,stencil_1)
        operator_1(root)
        ....
        Htree_exghange_neighbors(root,stencil_n)
        operator_n(root)
        Htree_update(root)
        Htree_det_neighbors(root, stencil_1+....stencil_n)
    endwhile
```

In this case, the remote data collection is implemented through the advanced PIT interface defined in section 4. The H-tree_det_neighbors function is invoked after the H-Tree_creation function, to determine the initial neighborhood relations, and each time the neighborhood relation among the elements have to be updated, i.e. after the execution of the *operator_n*, the only operator that modifies the H-Tree. Since all the invocations to H-tree_exchange_neighbors function share the data collected by the same invocation of the H-tree_det_neighbors one, the neighborhood stencil passed to the H-tree_det_neighbors function is the union of the neighborhood stencils of all the operators. The H-tree_exchange_neighbors function, instead, is invoked before each operator, to collect the updated property values that will used by the following operator. For the update of the H-Tree, instead, the standard interface has been adopted. The H-Tree_update function is invoked after the *operator_n*, because the other operators does not modifies the domain. Notice that the H-tree_det_neighbors function has to be invoked after the H-Tree_update function, because both the H-Tree structure and its mapping have to be updated, to determine the new neighborhood relations,.

Finally, we notice that in both cases, our approach preserves most of the existing sequential code, the modifications to the sequential code are minimal

and that they do not require a detailed knowledge of parallel programming techniques.

7 Conclusions

This paper has briefly presented the guidelines for PIT, a complete library to develope parallel algorithms to solve irregular problems. As previously shown, the parallelization through PIT is very simple, and preserves most of the sequential code. Moreover, the library provides both a simple API to allow users that are not familiar with parallel programming to easily obtain a parallel version of their sequential applications, and an advanced APIs to be exploited by an expert user to develope a customized parallel code for the specific irregular problem.

References

1. 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*, 1900:218–222, August 2000.
2. F. Baiardi, S. Chiti, P. Mori, and L. Ricci. Adaptive multigrid methods in MPI. *Proceedings of Euro PVM/MPI 2000: Lecture Notes in Computer Science*, 1908:80–87, September 2000.
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, May 2000.
4. F. Baiardi, S. Chiti, P. Mori, and L. Ricci. Integrating load balancing and locality in the parallelization of irregular problems. *Future Generation Computer Systems: Elsevier Science*, 17:969–975, June 2001.
5. J.S. Fink, S.B. Baden, and S.R. Kohn. Efficient run-time support for irregular block-structured applications. *Journal of Parallel and Distributed Computing*, 50(1):61–82, 1998.
6. S.S. Mukherjee, S.D. Sharma, M.D. Hill, J.R. Larus, A. Rogers, and J. Saltz. Efficient support for irregular applications on distributed-memory machines. In *ACM SIGPLAN Notices*, volume 30, pages 68–79. 1995.
7. 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.
8. A. Sohn, R. Biswas, and H. D. Simon. A dynamic load balancing framework for unstructured adaptive computations on distributed memory multiprocessors. In *Proc. 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 189–192, 1996.