SOLVING IRREGULAR PROBLEMS THROUGH PARALLEL IRREGULAR TREES

Fabrizio Baiardi
Dipartimento di Informatica
Università di Pisa
via Buonarroti, 2 - 56127 Pisa
email: baiardi@di.unipi.it

Paolo Mori
Istituto di Informatica e Telematica
CNR - Pisa
via Moruzzi, 1 - 56124 Pisa
email: paolo.mori@iit.cnr.it

Laura Ricci
Dipartimento di Informatica
Università di Pisa
via Buonarroti, 2 - 56127 Pisa
email: ricci@di.unipi.it

ABSTRACT
This paper presents PIT, a library for the parallelization of irregular problems on distributed memory architectures. All the strategies underlying the definition of the library can be expressed in terms of operations on a PITree, a parallel version of the tree data structure oriented to irregular problems. We consider the application of PIT to two well known irregular problems: adaptive multigrid and hierarchical radiosity methods. Performance figures that prove the effectiveness of PIT are presented.

KEY WORDS
Parallelization Tools; Irregular Problems; Distributed Memory Architectures; Load Balancing; Locality

1 Introduction

Most irregular scientific problems may be modelled as an iterative computation of properties of each of a set of elements. These elements are distributed in a problem dependant domain in an irregular and dynamic way. While the exact definition of the set of properties that characterizes each element depends upon the problem of interest, some properties define the position of the element in the domain. The system evolution is due to the interactions among the elements that update the system state by updating some of the properties of the elements, and it is a function of both time and the required accuracy of the results. A locality property holds because the evolution of the properties of an element e depend upon those of a small subset of other elements, the neighbours of e, $n(e)$. $n(e)$ changes at run time according to the system evolution due to the property updates. The Barnes-Hut method, adaptive multigrid methods and hierarchical radiosity [1] [2] [3] are well known problems in this class. In the following, the specific problem that is consider is denoted as the target problem.

With reference to this class of problems, we define PIT, a programming library to support the parallelization methodology proposed in [4]. The key concept underlying the methodology and the library is that both the sequential and the parallel application can be described in terms of trees. While a sequential application can be described in terms of operations on a classical tree, a parallel one can be structured in terms of operations on a parallel irregular tree, PITree, an evolution of the tree data structure that is oriented to distributed memory systems. Hence, a parallel application may be developed from a sequential one by transforming the operations on a tree into operations on PITrees. The goal of the PIT library is to simplify this transformation for those users that are familiar with the target problem but that are not expert in the development of parallel programs. Alternative approaches for the parallelization of irregular problems have been presented in [5], [6] and [7].

The paper is structured as follows. Sect. 2 briefly introduces PITrees and their operations. The PIT library and the operations it implements are described, respectively, in Sect. 3 and 4. Experimental results of the PIT parallel versions of an adaptive multigrid method and a hierarchical radiosity mehod are presented in Sect. 5.

2 A Hierarchical Representation of the Domain

Our methodology describes the element distribution through a multi level hierarchical representation, defined in terms of a recursive decomposition of the problem domain. Each level partitions both the domain into subdomains and the set of elements into subsets, according to the element positions. The partition strategy we adopt is Geometric Recursive Bisection, GRB. GBR is applied to the initial domain and, recursively, to the resulting subdomains, until any subdomains A satisfies a condition $C(A)$. $C(A)$ is problem dependent and it is a function of both the number of elements in A and the accuracy of the results the user requires. If $C(A)$ does not hold, GRB splits each side of A to produce 2^n subdomains where n is the number of dimensions of the domain. The distribution of the elements in a subdomain determines the number of times it is partitioned.

A tree can naturally represent both the hierarchical relation among the resulting subdomains and the distribution of the elements in the subsdomains. A hierarchical tree, *Htree*, is a tree that represents a domain decomposition, and its nodes are denoted *hnodes*. The hierarchical relation among the hnodes represents the one among the subdomains resulting from the decomposition. The root of a Htree represents the whole problem domain, while each hnode H at level l represents either an element or a subdomains produced by splitting l times the initial domain. A hnode that represents an element is always a leaf of the Htree. A leaf represents either an element or an unpartitioned, empty, subdomain. In the following, *SeqHtree* denotes the Htree representing all the subdomains resulting from a domain decomposition. A sequential application may be defined in terms of operation on the *SeqHtree* that represents the element distribution in the domain.

Fig. 1 shows an example of hierarchical decomposition and the corresponding SeqHtree. In this example, $C(A)$ holds iff the subdomain A includes at most one element. Hence, the hnode representing an element e is a son of the hnode that represents the smallest subdomain that includes e.

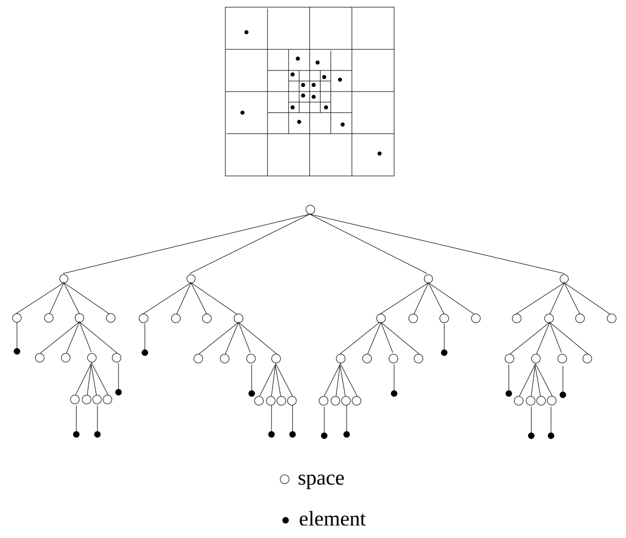

Figure 1. Domain Decomposition and HTree

2.1 *Parallel Irregular Tree*

A PITree is a distributed version of the Htree that describes a mapping of a SeqHTree onto the np processing nodes, pnodes, of a distribute memory architecture. A PITree pt is an ordered tuple of $np + 1$ subsets of a SeqHtree s:

$$pt \equiv \langle h_0, .., h_{np-1}, mht \rangle$$

Each h_i, $i \in 0, ..., np - 1$ is a forest that includes all the subtrees of s whose hnodes have all been mapped onto the i^{th} pnode. For the sake of simplicity, we assume that any h_i includes just one tree, the *private Htree* of i^{th} pnode. Each private Htree represents one of the subsets resulting from the partition of the domain. Each subdomain resulting from the hierarchical decomposition belongs to one of these subsets only, and it is represented in just one private Htree. Consequently, there is no intersection between the subdomains and the elements represented by two distinct private Htrees of the same PITree. Each hnode of a private Htree stores the properties of either the sudomain or the element it represents. Distinct PITrees are equivalent iff they describe alternative mappings of the same SeqHTree.

The $np+1^{th}$ subset of the PITree, the *mapping Htree*, mht, is a further tree that represents both the hierarchical relation among the private Htrees and their mapping onto the pnodes. mht is the subset of the SeqHtree that includes the root R of the SeqHtree and all the hnodes on the paths from R to the roots of all the private Htrees. Each hnode of mht corresponds also to a hnode of a private Htree h_i and it records the identifier i. If the mapping strategy preserves locality, the size of the mapping Htree is fairly lower than that of the SeqHtree. The mapping Htree is replicated in the local memory of each pnode and it is the only information on the mapping of the elements available to each pnode. Fig. 2 shows a PITree that represents the SeqHTree of Fig. 1.

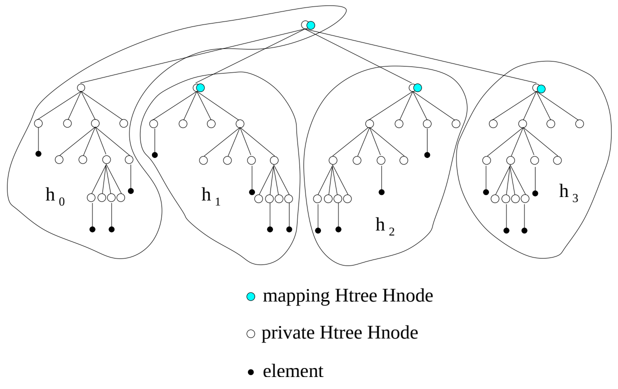

Figure 2. Mapping Htree

The main operators defined on a PITree are: *Creation*, *Completion*, *Correction* and *Balance*. The parallelization methodology underlying the definitions these operators has been proposed in [4]. A detailed and more formal description of the operators can be found in [8].

Creation creates a PITree from the set of elements in problem domain in two steps. The first one decomposes the initial domain as described in the previous section, and it partitions the resulting hierarchy of subdomains into np subsets so that each subset, that defines a further subdomain, preserves locality as much as possible. To this aim, the subdomains produced by the hierarchical decomposition are ordered into a sequence S according to a space filling curve [9]. This way of defining S exploits the properties of the space filling curves to guarantees that: *i)* if two subdomains are close in S then they are close in the domain, *ii)* if two subdomains are close in the domain, are close in S with an high probability. The domain partition is defined by partitioning S into np subsequences, each with the same computational load. All the subdomains in the same subsequence are mapped into the same pnode. After transforming each subsequence into a private Htree, the mapping Htree is built according to the roots of all the private Htrees.

Completion transforms each private Htree h_i into the corresponding essential Htree eh_i. eh_i is equal to the union of h_i with the set of the hnodes representing the neighbours of the elements in h_i. Hence, eh_i includes all the elements whose properties are required to update those of the elements in h_i.

Correction updates a PITree to avoid inconsistencies due to the updated values of some properties. After an update, the properties of an element e may be inconsistent with respect to the mapping of e described by the PITree. As an example, in the N body problem, properties are related to the position of a body in the domain. After an update, the position of the body may belong to a subdomain mapped onto another pnode.

Balance transforms a PITree into an equivalent one but that describes a mapping that balances the computational load among the pnodes. This operation should preserve at most locality among the elements and minimize the number of hnodes whose mapping changes and, as a consequence, are transferred from one private Htree to another one.

3 PIT Library

Parallel Irregular Tree, PIT, library is a simple, complete and effective tool for the parallelization of irregular applications on distributed memory architectures. PIT is addressed to those users that want to transform a sequential irregular application into a parallel one but are not familiar with parallel programming.

The key assumption underlying PIT is that both the sequential and the parallel versions of the application are structured in terms of operations on Htrees. In particular, the sequential code is structured as a sequence of visits of a SeqHtree. For each visited hnode e, the neighbors of e are collected and the proper interaction is computed to update the properties of e. These visits are denoted as *target problem, tp,* operators, and they define a kernel of both the sequential and the parallel applications. PIT preserves the code of this kernel when transforming a sequential application into a parallel one. The main problem to be faced is that the sequential version applies the tp operators to a SeqHtree stored in the local memory of a single pnode, the parallel one applies them to a PITree through a parallel visit where each pnode considers a distinct subset of the PITree.

The code of a tp operator can be preserved when parallelizing an application provided that, as in the case of PIT, the implementation of an operator on a PITree solves any communication and synchronization issue arising because of the distributed representation of the Htree. The next section briefly describe how this is implemented in the PIT library.

4 PIT Functions

PIT defines a set of functions on the PITree. Each function, $PITfn$, is applied to a PITree in a Single Program Multiple Data (SPMD) style, where each pnode applies $PITfn$ to a distinct subset of the PITree, i.e. to its private Htree. Distinct pnodes cooperate in the execution of the same $PITfn$.

Each of the PIT operators defined in Sect. 2.1 is implemented through one or more PIT functions. We refer to [8] for a detailed description of the implementation. PIT defines two APIes, a standard and an advanced one. The standard API only includes those functions that implement the PIT operations in the most general case. The advanced API defines a larger set of functions, to enable a parallelization expert user to choose the most appropriate functions and the most effective way to compose them when parallelizing an application. The main PIT functions are the following.

Creation. It implements the *Creation* operation without introducing a centralization point. The elements are distributed among all the pnodes at the beginning of Creation and exchanged at the end of the operation according to the adopted mapping. In this way, there is no need to store all the element in the local memory of one pnode and the size of this local memory does not constraint the total number of elements. The inputs of `Creation` are: np, the initial set of elements, and some functions to manage the element properties. In this way, PIT is independent of the target problem and, in particular, of the element properties, because the user can define both the structure to record the properties of an element e and the functions to handle them. As an example, one of these functions implements the decomposition of the element of the target problem, and is exploited by `Creation` to perform the Geometric Recursive Bisection. The results of `Creation` are the private Htree of the pnode and the mapping Htree. By setting $np = 1$, a sequential application can exploit this function to decompose the domain and build the SeqHtree.

Completion The standard interface defines a single function to implement the *Completion* operator, `Completion`, to be applied just before each tp operator. Obviously, this simplifies the development of the parallel version at the expense of efficiency. The expert user, that knows which element properties each tp operator exploits and updates, can use the advanced API that implements *Completion* by composing two PIT functions, `Det_neighbors` and `Exch_neighbors`. The former exchanges among the pnodes some information about the PITree that will be exploited by the latter to determine more efficiently the neighbors of each element. `Det_neighbors` is invoked after `Creation` as well as after each tp operator that updates the properties that influence the neighbourhood stencil. `Exch_neighbors` actually exchanges the hnodes among the pnodes. It is applied before any tp operator that needs the updated values of the elements properties. The inputs of these PIT functions are: the private Htree of the pnode and a function that implement the neighborhood stencil of the target problem operator.

Correction and Balance The standard PIT interface defines one function, `Update`, that implements both the *Correction* and the *Balance* operations. This function updates the mapping of the PITree taking into account both the elements that violate the mapping strategy and the load unbalance. To prevent inconsistencies in the PITree, the function is applied after any update to the element distribution. Hence, in the simplest case, `Update` is invoked after each tp operator. The advanced interface includes two distinct functions, `Correction` and `Balance` that, respectively, updates the mapping of the elements that give rise to an inconsistency and balances the load. `Correction` has to be invoked each time the element distribution changes, and takes as input the private Htree of the pnode. `Balance`, instead, is invoked after each tp operator that changes the workload paired with the elements. This function takes as input the load balance threshould T and the private Htree of the pnode. At first, it collects the workload of each pnode, and computes the maximum unbalance of the current mapping. The balancing procedure is applied only if this unbalance is larger than T. The workload of each element is recorded in a field of the corresponding hnode that can be updated by the user. As an example, if the workload of e changes during the computation, the user can record in the field of the proper hnode an estimate of the number of operations to update the properties of e.

Table 1 shows the skeleton of a simple parallel application developed through the PIT library.

In this example, `tp_op_1`,...,`tp_op_n` are the target problem operators, `stencil_1`,...,`stencil_n` are the related neighbourhood stencils, and `dom` represents the set of elements of the domain. At first, let us suppose that only `tp_op_j` and `tp_op_n` update both the Htree and the neighborhood relation among the elements. In this case, `det_neighbors` is applied after `creation`, after the application of `tp_op_j`, `tp_op_n` and of the `balance` function. Since the neighborhood relation, as determined by the `det_neighbors` function applied before `tp_op_1`, is exploited by `exch_neighbors` to collect the hnodes to apply `tp_op_1, .., tp_op_j`, the neighborhood stencil exploited by `det_neighbors` is the union of the stencils of these tp operators. In the same way, `det_neighbors` applied before `tp_op_j+1` exploits a neighborhood stencil that is the union of the `tp_op_j+1, .., tp_op_n` ones.

The `exch_neighbors` function, instead, is invoked before each tp operator, to collect the updated properties values to be used by the tp operator. However, if `tp_op_j-1` and `tp_op_j` exploit the same stencil, while `tp_op_j` does not exploit the properties updated by `tp_op_j-1`, then `tp_op_j-1` and `tp_op_j` can both exploit the data collected by the instance of `exch_neighbors` applied before `tp_op_j-1`.

```
thnode *pht_root
pht_root =creation(np, dom, dec_el,
                   inc_el, rem_el)
while (not solution_computed) {
  det_neighbors(pht_root, stencil_1+
                +...+stencil_j)
  exch_neighbors(pht_root, stencil_1)
  tp_op_1(pht_root)
  ....
  ....
  exch_neighbors(pht_root, stencil_j-1)
  tp_op_j-1(pht_root)
  tp_op_j(pht_root)
  pht_root = correction(pht_root)
  pht_root = balance(pht_root, T)
  det_neighbors(pht_root, stencil_j+1+
                +...+stencil_n)
  exch_neighbors(pht_root, stencil_j+1)
  tp_op_j+1(pht_root)
  ....
  ....
  exch_neighbors(pht_root, stencil_n)
  tp_op_n(pht_root)
  pht_root = correction(pht_root)
}
```

Table 1. Example of PIT parallel code

`correction` is invoked after `tp_op_j` and `tp_op_n` only, because the other tp operators do not update the Htree. Notice that `det_neighbors` has been invoked after `correction`, because, in general, this function updates both the PITree and the mapping of some hnodes.

Finally, in this example we suppose that only the tp operator `tp_op_j` updates the computational load of each element. Hence, `balance` has been applied only after `tp_op_j`. Since also `correction` is applied after `tp_op_j`, `balance` is applied after `correction` because the PITree could be in an inconsistent state after the execution of `tp_op_j`. In this case, `correction` has to be applied to the PITree before any other PIT function.

5 Experimental Results

To evaluate the effectiveness of both our methodology and PIT, we have exploited PIT to parallelize two well known irregular problems, adaptive multigrid methods and hierarchical radiosity methods, on two distributed memory architectures. The first one is a cluster of 10 PCs, each equipped with an Intel Pentium II CPU (266 MHz) and 256 MB of local memory. The interconnection network is a switched 100Mbit Fast Ethernet. The other architecture is an IBM Linux Cluster with 64 nodes, each with 2 Intel Pentium III (1.133 GHz) processors and 1Gbyte of local memory. The interconnection network is a Miricom LAN C with a bandwidth of 264 MB.

Adaptive multigrid methods are fast iterative methods for the numerical resolution of partial differential equations in two or more dimensions [2] [10]. The considered problem is a very complex instance of the Poisson problem on the unit square in two dimensions, i.e. the Laplace equation:

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

with $f(x,y) = 0$ and subject to the Dirichlet boundary conditions:

$$h(x,y) = 10\cos(2\pi(x+y-1))\frac{\sinh(2\pi(x-y+3))}{\sinh(8\pi)}$$

Fig. 3, shows the highly irregular grid built after just 10 iterations.

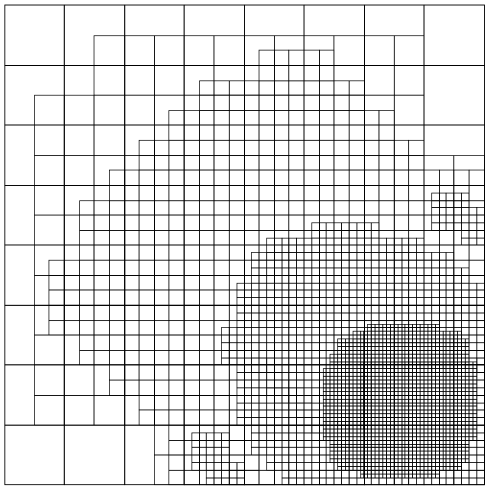

Figure 3. Grid hierarchy after 10 iterations

The first experiment evaluates the impact of the load balancing on the performance of the PIT version. Fig. 4 shows the completion time of 50 iterations on the PC cluster for an initial uniform 64×64 points grid and for different values of the threshold that determines the largest unbalance that is tolerated. The graph shows that the best result from the point of view of the completion time and, consequently, of efficiency are achieved with a threshold of 10%. This curve also shows the influence of load balancing on the completion time because it shows that a load balancing threshold of 100% results in an increase of the completion time of 126% wrt the completion time with the optimal threshold. Hence, better speed ups can be achieved by proper exploiting the load balance strategy offered by the library.

Another set of experiments evaluates the performances of the PIT version of the adaptive multigrid methods. Figures 5 shows the efficiency for a variable number

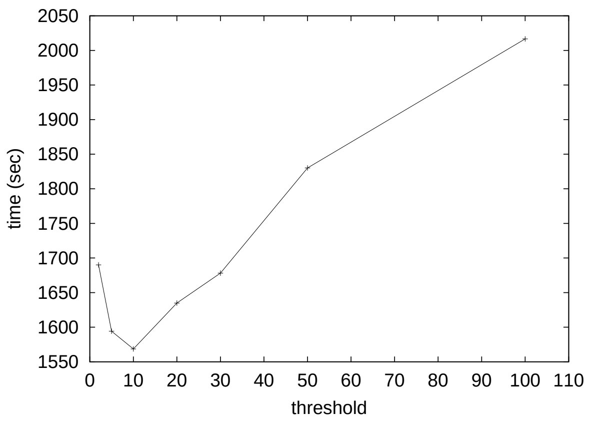

Figure 4. Completion time with alternative load balancing thresholds

of pnodes in the case of 50 iterations on the two architectures. In all the experiments, the load balancing threshold has the optimal value previously determined and very small initial grid, 64×64 points, is considered. This corresponds to a worst case from the scalability point of view, because the number of points and, consequently, the computational load, of each pnode decreases as the number of pnodes increases, while the synchronization overhead increases with the number of pnodes.

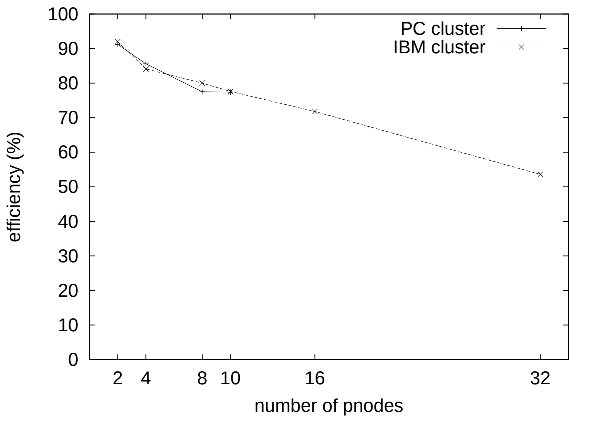

Figure 5. Efficiency for Adaptive Multigrid

Fig. 5 shows that, on both the architectures, our implementation achieves an efficiency larger than 75%, even on 10 pnodes. On the IBM cluster, the efficiency is larger than 50% even on 32 pnodes. The efficiency is even larger for a lower number of pnodes, e.g. on 4 pnodes it is larger than 85%.

Hierarchical radiosity [3] is another well known irregular method. It computes the global illumination of a set of objects in a scene by modelling the exchange of light among the surfaces that compose the scene [11]. For the sake of simplicity, we consider flatland [12], a bidimensional world and a very simple scene including 896 initial segments that compose 192 polygons. Again, this is a worst case because real images are much larger and include several millions of initial elements.

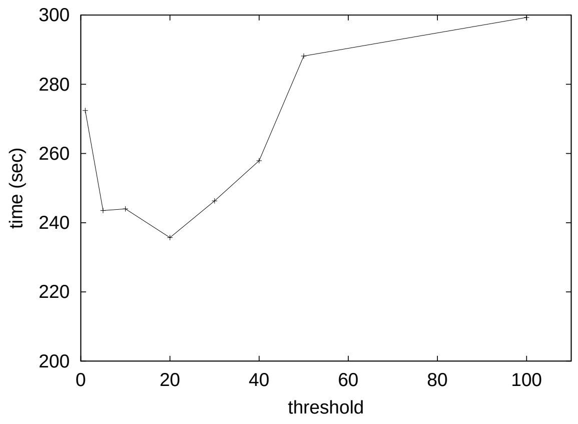

Figure 6. Completion time with alternative load balancing thresholds

The first results concern again the impact of load balancing on the efficiency of the parallel application. Fig. 6 shows the completion time of 30 iterations of the parallel application for different values of the load balancing threshold on the 10 nodes cluster. The best completion time is achieved with a 20% threshold. These results confirm the importance of load balancing for irregular problems, because a threshold of 100% results in a completion time that is 116% of the optimal one. Furthermore, the results confirm the effectiveness of our methodology as well.

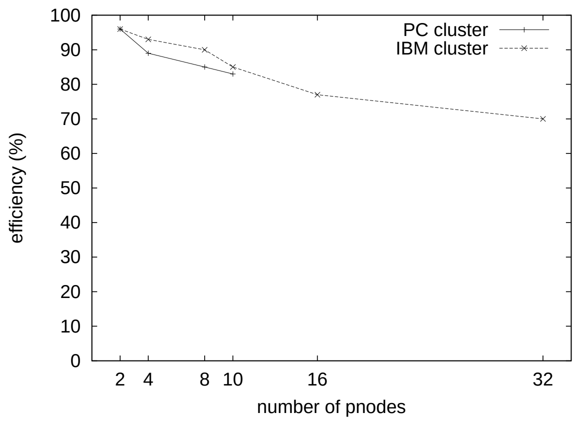

Figure 7. Efficiency for Hierarchical Radiosity

Fig. 7 shows that, on both architectures, and in the worst case we have considered, the parallel application achieves an efficiency larger than 80%, even on 10 pnodes. On the IBM cluster, the efficiency is larger than 70% even on 32 pnodes. This is mostly due the ability of the PIT strategies to exploit at best the high locality of the scene.

6 Acknowledgements

This work was partially supported by CINECA.

References

[1] J.E. Barnes and P. Hut. A hierarchical O(nlogn) force calculation algorithm. *Nature*, 324:446–449, 1986.

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

[3] M.F. Cohen, D.P. Greenberg, D.S. Immel, and P.J. Brock. An efficient radiosity approach for realistic image synthesis. *IEEE Computer Graphics and Applications*, 6(2):26–30, 1986.

[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*, 17:969–975, 2001.

[5] A.J. Field, P.H.J. Kelly, and W. Qian. M-tree: A parallel abstract data type for block-irregular adaptive applications. In *Proceedings of EuroPar 97; LNCS*, volume 1300, 1997.

[6] S.R. Kohn and S.B. Baden. A robust parallel programming model for dynamic non-uniform scientific computations. In *Proceedings of the 1994 Scalable High Performance Computing Conference*, pages 509–517, 1994.

[7] 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 *Proc. of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 68–79, 1995.

[8] P. Mori. *Solving Irregular Problems on Parallel Architectures: a Methodology and a Supporting Library*. PhD thesis, Dipartimento di Informatica, Università di Pisa, 2003.

[9] 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.

[10] W. Briggs, V.E. Henson, and S.F. McCormick. *A multigrid tutorial Second Edition*. SIAM, 2000.

[11] C.M. Goral, K.E. Torrance, D.P. Greenberg, and B. Battaill. Modelling the interaction of light between diffuse surfaces. *Computer Graphics*, 18(3):213–222, 1984.

[12] P.S. Heckbert. Radiosity in flatland. *Computer Graphics Forum Eurographics 92*, 11(3):181–192, 1992.