Ray tracing on distributed memory parallel computers : strategies for distributing computations and data

Didier Badouel, Kadi Bouatouch, Thierry Priol

INRIA

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tel.(1)39 63 55 11

Rapports de Recherche

N° 1163

Programme 2
Structures Nouvelles d'Ordinateurs

RAY TRACING ON DISTRIBUTED MEMORY PARALLEL COMPUTERS : STRATEGIES FOR DISTRIBUTING COMPUTATIONS AND DATA

Didier BADOUEL
Kadi BOUATOUCH
Thierry PRIOL

Février 1990

RR-1163

IRISA INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CÉDEX
FRANCE
Téléphone : 99 36 20 00
Télex : UNIRISA 950 473 F
Télécopie : 99 38 38 32

Publication Interne n°508 - Janvier 1990 - 16 Pages

Ray Tracing on Distributed Memory Parallel Computers: strategies for distributing computations and data *

Didier Badouel
Kadi Bouatouch
Thierry Priol

Lancer de Rayon sur des Architectures à Mémoire Distribuée : Stratégie pour distribuer les calculs et les données

Abstract

The ray tracing algorithm produces high quality images, but it requires a lot of computations which makes it extremely time-consuming. Several attempts have been made to reduce the synthesis time by using high speed parallel computers. For example, distributed memory parallel computers, such as hypercubes or *transputer*-based machines, offer an interesting performance/cost ratio. Several studies have used parallel computers to speed up the ray tracing algorithm so as to render complex scenes with several thousands of objects. However few of them have led to real implementation due to the complexity of the solution proposed by the authors. This paper introduces a new approach emulating a read-only shared memory on a distributed memory parallel computer. Results are given and are compared to a previous parallel ray tracing algorithm, both are implemented on an iPSC/2.

Résumé

L'algorithme de lancer de rayon permet la production d'images de synthèse de grandes qualitées. Cependant il nécessite souvent un temps de calcul élevé. L'utilisation de nouvelles machines parallèles permet de réduire ces temps de calculs. Les architectures parallèles à mémoire distribuée, telles que les hypercubes ou les machines à *Transputer*, offrent un rapport performance/coût très intéressant. Des études ont été effectuées afin d'utiliser de telles architectures pour accélérer l'algorithme de lancer de rayon en vue de calculer des scènes complexes constituées de plusieurs milliers d'objets. Cependant, peu d'entre elles ont été suivies d'une mise en œuvre en raison de la complexité des solutions proposées par les auteurs. Cet article introduit une nouvelle approche qui utilise une mémoire partagée en lecture uniquement mise en œuvre sur une architecture à mémoire distribuée. Une expérimentation sur un iPSC/2 permet de comparer les résultats avec ceux obtenus avec un algorithme que nous avons décrit dans une publication antérieure.

1 Introduction

To render high quality images, the ray tracing algorithm needs to shoot different kinds of rays: from an observer and toward a screen plane (primary rays), in the reflected and refracted directions (secondary rays), toward the light sources (light or shadow rays).

Computing realistic images requires several millions of rays. Hence a large number of ray/object intersections are involved.

*This work has been supported by the French Coordinated Research Program C^3 and by the CCETT (Centre Commun d'Etudes de Télédiffusion et Télécommunications) under contract 86ME46

Several solutions have been proposed to reduce this number. They are based on what we call an *objects access structure* which allows a fast search for objects along a ray path. They can be grouped in two classes: those that build a tree (or a hierarchy) of bounding volumes [35, 25] and those based on a subdivision of the scene extent in order to take advantage of the spatial coherence [2, 5, 14, 15, 24].

Even when such mechanisms are used, the ray tracing algorithm is still too slow on sequential computers. Moreover, latest researches such as stochastic sampling [11, 27] and sophisticated light models [10, 23, 37] require more and more computations and new algorithmical improvements can not decrease substantially the synthesis time. Therefore, the only way for speeding up ray tracing algorithms is to use more powerful computers. Several supercomputers (Cray or Fujitsu) are available but their price often prohibits their use in computer graphics. On the other hand, low cost supercomputers are now commercially available. They are constituted by several powerful microprocessors and are known under the name of Distributed Memory Parallel Computers (DMPC).

Several attempts have been made in using such architectures for speeding up the ray-tracing algorithm but few of them have given results on real parallel architectures. The goal of this paper is to present and compare results of two algorithms which have been both designed for DMPC and implemented on an iPSC/2. Results have been performed on a set of scenes called *Standard Procedural Databases* provided by Eric Haines [19]. Therefore, readers can compare easily these results with their own.

This paper is organized as follows. After a brief description of the methodology for programming DMPCs, the previous works on parallel ray tracing algorithms are discussed. Then we present a ray tracing algorithm we have implemented on a parallel machine (iPSC/2) in order to emphasize the complexity of designing this kind of parallel algorithm. To overcome this complexity, we propose another parallel approach that consists in emulating a read-only shared memory on a distributed memory parallel computer.

2 Using a DMPC for speeding up ray tracing

A DMPC is a MIMD computer where each processor has a local memory used to store its own code and data. All the processors of a DMPC are connected through a network. Researches at Caltech [1] [36] have led to DMPCs organized according to a hypercube topology of dimension d, where the number of processors is $N = 2^d$. This topology has several interesting properties: topologies such as ring, mesh and tree can be embedded on a hypercube topology.

Various DMPC with hypercube topologies are sold by several companies: NCUBE/10 by Ncube, and iPSC/1 and iPSC/2 by Intel Scientific Computers. Parallel computers based on *transputers T800* also are DMPC; for example, the *Computing Surface* by Meiko, T-NODE by Telmat, Supernode from Parsys or the Supercluster from PARACOM. These transputer-based DMPC offer a programmable topology via a switch network.

2.1 An example of a DMPC: the Intel iPSC/2

For our experimentations, we have used an iPSC/2 which is configured into 64 nodes. The iPSC/2 system consists of two main components: the cube and the system resource manager.

The cube houses all the nodes which are connected through a hypercube network. Each node consists of an Intel 80386 microprocessor supplied with a 80387 floating point co-processor and 4 Mbytes of local memory. It is equipped with the Direct Connect Module (DCM) for high speed routing message between nodes. The performance of this configuration is approximately 256

[1] California Institute of Technology

MIPS and 20 MFLOPS. A node can support a vector extension board with a peak performance of 20 MFLOPS.

The System Resource Manager hosts the software development tools. It is connected via a special link to node 0. It performs compilation, program loading and I/O operations with the cube.

The iPSC/2 can be programmed using C or FORTRAN language. A communication library has been added to these languages to allow the exchange of messages between nodes.

2.2 Programming a DMPC

DMPCs can deliver high performances, but are difficult to program due to the lack of a powerful software environment. The standard programming methodology consists in subdividing the problem to be solved into a set of communicating tasks [21]. Each of these tasks is described with a conventional language such as C or FORTRAN. To take full advantage of the multicomputer resources, the programmer must try to follow several rules.

First, the task decomposition must ensure that the computations are evenly distributed among all the processors. Secondly, the programmer must take into account the cost of the remote data access: the data needed by a task must fit as often as possible in the local memory of the processor that runs the task so as to avoid a remote data access. Last but not least, the user must verify that there is no potential deadlock due to the communication in his parallel algorithm. Such situation occurs when for example two tasks are waiting for receiving a message and there is no message in the network.

2.3 Parallelizing ray tracing algorithms

A ray tracing algorithm consists of two distinct phases: the building of an *object data structure*, and the computation of the intensity of each pixel. This is the latter part that generates most of the computation and therefore that must be parallelized. Since the computation of each pixel is independent of the others, it can be easily distributed among the processors. As there are much more pixels than processors, load balancing can be achieved by using a server/client programming model: a server process assigns the computation of a pixel to a client process running on a non-busy processor.

We must determine which part of the database is needed by a processor to compute a pixel. Its size depends on the objects access structure which has been chosen for designing the ray tracing algorithm. For *space-tracing* algorithms, the objects access structure consists of a collection of adjacent 3D cells which may need several megabytes for its storage. The computation of a pixel by a processor potentially requires the knowledge of the whole objects access structure since the path followed by a ray is unpredictable. Several strategies can be used to assign the objects access structure to the different nodes:

1. *Duplication of the data in each processor.*

   The objects access structure is duplicated in each local memory of a DMPC. Load balancing is achieved according to a server/client programming model as said above. We name this strategy **processing without dataflow** since there is no interprocessor communication.

2. *Geometrical partitioning of the data.*

   Each processor is responsible for a sub-region of the scene extent. The distribution of the computations and of the data are made at the same time. The processor's load depends on the size of its associated sub-region. When a ray leaves a sub-region, it is communicated to the processor responsible for the next sub-region along the ray path. We name this strategy **processing with ray dataflow**.

3. *Random distribution of the data*

   The object data structure is randomly distributed among the local memo-

ries of the processors. If a processor needs a datum, such as an object or a voxel, which is not stored in its local memory, it can access it by sending a message to the processor containing this datum. This strategy needs a software service that provides a global memory abstraction. Load balancing also is achieved according to a server/client model. Since objects are exchanged between processors, we name this technique **processing with object dataflow**.

Several parallel ray tracing algorithms have been proposed, the next section classifies them according to the strategies they employ.

3 Previous works

3.1 Algorithms based on processing without dataflow

The simplest way to parallelize ray tracing algorithms is to duplicate the entire object data structure in the local memory of each processor. The CRISTAL parallel computer [6], the LINKS [31] and the SIGHT architecture [29, 39] use this technique. A sequential process running in each processor is in charge of computing a subset of pixels. This set is dynamically determined according to the load of a processor. In fact, this sequential process is almost the same as the one running on a sequential computer. Owing to this simplicity, many ray tracing algorithms running on DMPC as demonstration use this technique. However, the limited size of the local memory associated with each processor of a DMPC prohibits its use for rendering complex scenes. Efficient algorithms, such as space-tracing, cannot be used since the employed objects access structure would require a very large memory.

3.2 Algorithms based on processing with ray dataflow

Since using DMPCs aims at rendering very complex scenes, new researches have been made for distributing the objects access structure among the processors. Each processor is assigned one part of the whole database. There are mainly two techniques for distributing the database, according to the objects access structure which is chosen. The first one [8, 16] partitions the scene according to a tree of extents while the second [9, 12, 22, 26, 30] subdivides the scene extent into 3D regions (or voxels).

3.2.1 Tree of extents as an object data structure

The distribution of a tree of extents has first been proposed by Goldsmith and Salmon [16]. The method they propose involves the creation of a hierarchy of rectangular extents used to compute intersections. Only the upper tree (named forest by the authors) is replicated in each processor. The lowest levels of the forest are pointers to subtrees of the entire hierarchy: a pointer provides access to the processor in which the appropriate part of the database is stored. Thus, each processor controls a subset of pixels together with the forest and a subtree of extents corresponding to the associated portion of the database.

To compute the color of a pixel, a processor shoots a primary ray and follows it through the forest down to the lowest levels of this forest which are pointers. If this traversal leads to a pointer to this processor, this latter performs the necessary calculation, otherwise it sends the relevant ray to the concerned processor.

The drawback of this algorithm is the weakness of its load balancing technique which is only based on the distribution of the pixel array. Caspary and Scherson have addressed this problem in [8].

Algorithms based on tree of extent have the advantage of requiring less memory than those based on a spatial subdivision described below. But this is at the expense of a more important execution time caused by the traversal of the hierarchy (forest and subtrees associated with the portions of the whole database) which requires numerous

floating point operations.

3.2.2 Spatial subdivision as an object data structure

Each processor is assigned one or more regions and each region contains on the one hand a part of the whole database and on the other hand a data structure describing a spatial subdivision (octree, grid ...). Rays are passed from processor to processor via messages. Such algorithms have been proposed by Cleary et al. [9], Dippe et al. [12], Nemoto et al. [30], Kobayashi et al. [26] and Jevans [22].

As far as load balancing is concerned, several strategies have been proposed in [12] and [30]. The load balancing is performed dynamically by moving the boundaries of the regions allocated to the processors by means of redistribution messages passed between processors. Besides the numerous floating point operations and the important message traffic it involves, this kind of dynamic load balancing technique raises difficult issues. Since a dynamic load balancing proceeds by moving a corner or a face; which corner or face is selected first? What is the behavior of the algorithm when all the processors are adjusting their load at the same time? What is the periodicity of the load redistribution? This latter may be a source of oscillations. How to avoid it? How to solve the deadlock problem caused by a large message traffic?

3.3 Algorithms based on processing with object dataflow

Examples of such algorithms are those proposed by Green et al. [18] and Potmesil and al [32].

Green's algorithm is embedded on an architecture which is organized according to a tree of transputers, each one uses its local memory as a cache to reduce the communications with the other processors (ancestors, parents, children). The root processor contains the whole database representing the scene. In fact, the database consists of two parts: an octree description and objects descriptions. Each node processor has a copy of the octree description and maintain a local database (description of some objects) which, in general, is much smaller than the whole database. The main feature of this algorithm is that portions of the database must be communicated between processors to accomplish the ray/scene intersections. Requests for objects are then passed between processors to ensure a dynamic allocation of objects to the processors. Each node processor maintains a ray stack to be treated, passes requests for work to its parent processor when the stack is empty, pushes onto the stack the rays spawned from the last popped ray. This approach is well suited to ensure a load balancing.

The second object dataflow algorithm [32] is implemented on the Pixel Machine. This architecture offers a parallel processing thanks to an array of pixel nodes and a distributed frame buffer. The pixels are distributed among the pixel nodes, the CPU of which is a $DSP32$ processor with only 36 Kbytes for program and data storage. As this memory is too small, a virtual memory mechanism is used to render scenes exceeding 36 Kbytes.

For these two algorithms, the virtual memory is located in the host, which in our mind, increases the communication cost between the host and the nodes, and consequently may reduce their efficiency. For DMPCs, it is more efficient to distribute the virtual memory among all the nodes because a node to node communication is faster than a communication between the host and the nodes.

Later on, we will describe a new approach of parallel ray tracing which emulates a read-only shared memory on a DMPC. Unlike the previous algorithms, this approach distributes the virtual memory among all the nodes and uses both caching and paging mechanisms.

3.4 conclusion

In most cases, algorithms have been simulated in a sequential environment which does not take into account important features such as communication overhead and synchronization. Moreover, scenes used in these simulations are different and do not allow to compare the different strategies used in the parallel algorithms. Therefore, it is difficult to appreciate the performances of these algorithms. As for the algorithms based on processing without dataflow, they are simple to implement but, for complex scenes, they cannot be embedded on a DMPC because of the large memory they require. The last part of this paper shows experimentations we have done on an iPSC/2 hypercube. Two parallel ray tracing algorithms are presented in the two next sections. The former is based on processing with ray dataflow whereas the latter is based on processing with object dataflow. Results are given and used to compare their performances.

4 A ray dataflow approach

This section deals with the description and the evaluation of a parallel ray tracing algorithm based on processing with ray dataflow. Since rays are exchanged between processors via messages, it is well suited to DMPC using standard message-passing programming methodology. This algorithm is almost identical to the one described in [34], the only difference lying in the way the computation and the data are distributed to ensure load balancing.

4.1 Partitioning both the computations and the objects access structure

Cleary's [9] idea of distributing the objects among the processors is elegant. However, the distribution of the computation he proposes is difficult to implement. Indeed, the load associated with each processor depends on the size of the corresponding subregion and therefore depends on several parameters: a) the relative location of the sub-region with respect to the scene extent, b) the number, the location and the size of the objects contained in the sub-region, c) the photometric properties of the objects contained in the sub-region, d) the number of rays entering in the sub-region.

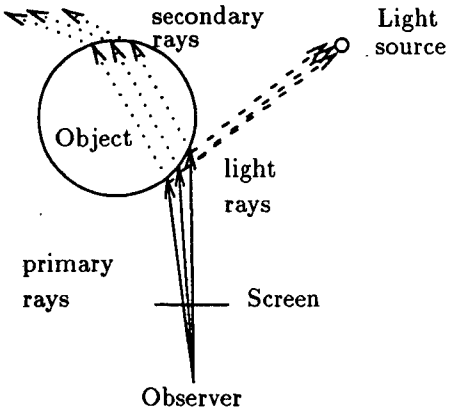

Figure 1: Ray coherence property.

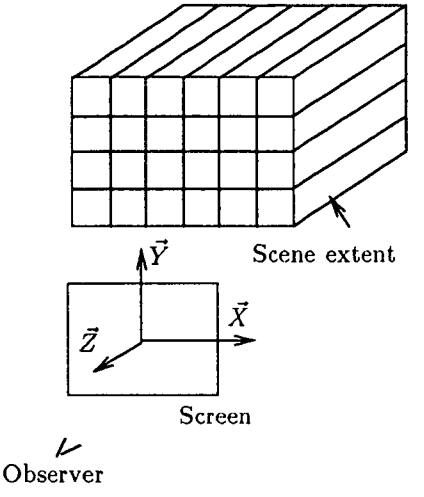

Figure 2: Subdivision of the scene extent using a 2D grid.

A modification of the size of a subregion entails the change of these parameters and consequently the corresponding load. The synthesis time related to one region cannot be precisely known in advance because of the important number of parameters involved in the computation. However, it can be estimated by using the ray coherence property [20, 38] As shown in figure 1, two rays shot

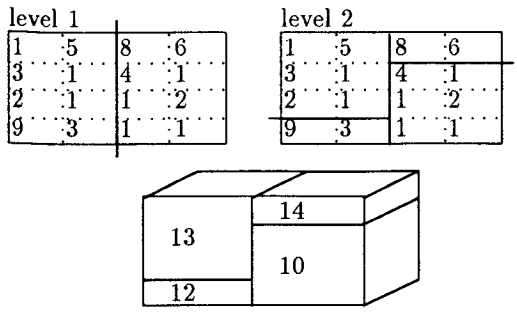

Figure 3: Clustering cells in equal load 3D regions

from the observer through two adjacent pixels have a high probability to intersect the same objects. This property is also true for all the rays spawned from the two primary rays. Consequently, the time needed to compute a pixel is close to that of the neighboring pixels. We use this ray coherence property to subdivide the scene extent into equal load 3D regions. Three steps are needed to achieve this space subdivision:

1. The scene extent is subdivided into cells by using a 2D grid as shown in figure 2.

2. The image is sub-sampled, so that only a subset of pixels is computed. A counter is associated with each cell and represents an evaluation of the amount of time needed to treat all the rays which enter this cell.

3. Values of counters are used to cluster cells in order to get equal loaded 3D regions (see Fig. 3). There are as many regions as processors.

The clustering technique is the following one. The scene extent is recursively subdivided using median planes perpendicular to the X and Y axes of the screen coordinate system. This technique is known as *BSP* [2] [13] in the field of computer graphics or *BDD* [3] [3] in the area of parallel computers. The *BSP* is convenient because it

[2] Binary Space Partitioning
[3] Binary Domain Decomposition

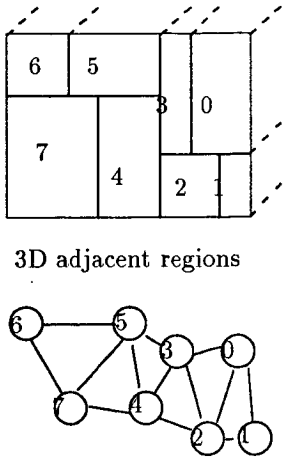

Figure 4: Communication graph.

provides a 2^N regions, which is the number of processors in a hypercube topology. Figure 3 shows a partitioning of depth 2. The subdivision provides a set of adjacent regions. The adjacency (or connectivity) of these regions may be modeled by a connectivity graph, whose vertices are 3D regions and edges represent the adjacency relation. Since each region is associated with a process performing a ray tracing operation, the connectivity graph can be seen as a graph of processes, where a vertex is a process and an edge represents the communication between processes (figure 4). It is this graph that should be mapped on a real architecture. The subdivision is performed so as to produce a number of processes equal to the number of processors. So, the aim of the mapping is to place communicating processes on neighboring processors.

4.2 Algorithm overview

The algorithm consists of two processes.

1. *Host process* :
   As described above, the host performs a static load balancing by subdividing the scene extent into $3D$ regions. Then, according to a mapping strategy, it assigns to each processor a region and a portion of the database as well as a part

of the pixels array. Once all the processors have accomplished, in their turn, a spatial subdivision of the region they are responsible for, they notify the host which sends them a message allowing them to start the synthesis phase. Finally, the host waits for the pixel intensity contribution messages coming from the nodes and updates the frame buffer accordingly.

2. *Node process* :
   A node process performs two tasks : synthesis and communication.

   - Synthesis task
     This task subdivides the region provided by the host into $3D$ cells. Then, it shoots the primary rays through the portion of the pixels array it is responsible for. These rays can give rise to secondary rays which are passed to other nodes by putting the associated messages into a FIFO queue. To avoid the saturation of the queue, this synthesis task treats, in priority, the rays coming from other processors. The fraction of the pixel intensity computed by this task is sent to the host which stores it in the frame buffer.
   - Communication task
     This task is in charge of managing the FIFO queue. Indeed, when the synthesis task wants to put a ray message in the queue, it calls the communication task which sends the oldest ray message in the FIFO so as to avoid saturation.

4.3 The deadlock problem

In our case a deadlock situation can only be caused by the saturation of the FIFO queues. Figure 5 illustrates such a situation. If the FIFO queue of processor P_1 is full, then P_1 cannot receive messages as these might create later new rays that could not be saved in P_1's queue. Moreover, let us assume that processor P_0 sends shadow ray

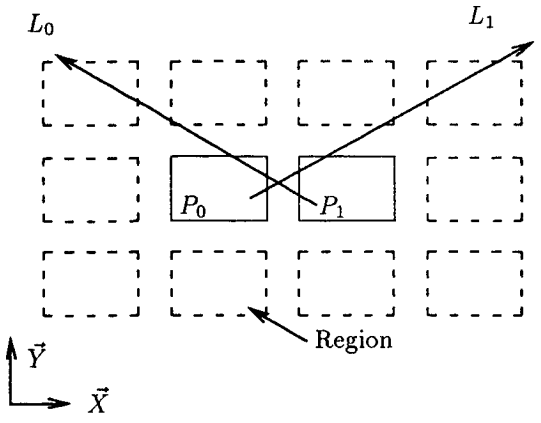

Figure 5: Example of a deadlock situation

messages to P_1 by saving them in its queue. After a while, P_0's queue is, in its turn, full because P_1 cannot receive these messages. This situation then entails a deadlock.

To remedy a deadlock situation, we opted for the following strategy. When a FIFO queue is full, the ray messages are sent to the host instead of the concerned processors. Then, the host processor saves these messages in its own memory and forwards them to the concerned processors as soon as possible. For some test images the host receives about 0.5 percent of all the exchanged messages.

4.4 Results and discussion

Tests of our parallel algorithms, this one as well as the other developed in the next section, have been performed on a set of scenes called *Standard Procedural Databases* (SPD) provided by Eric Haines [19].

Before giving the performances of our algorithm applied to these scenes, let us recall the definitions of the speed-up and the efficiency of an algorithm. The speed-up is the ratio of the running time of a processor to the running time obtained with p processors. This quantity represents, in fact, the number of processors effectively used during the parallel execution of the algorithm. As for the efficiency, it is equal to the ratio of the speed-up to the number of processors.

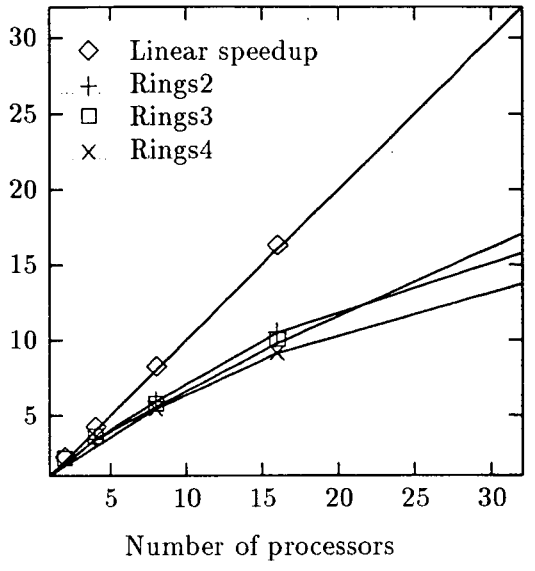

Figure 6: Speedup for the *Rings* images.

#	1	2	4	8	16	32
Rings2	100	90	86	74	65	49
Rings3	100	96	85	69	61	53
Rings4	100	–	–	64	53	43

Figure 7: Efficiency in percent for the *Rings* images.

It represents the average utilization of the processors.

Figures 6 and 7 give the speed-up and the efficiency of our algorithm for different scenes and different numbers of processors. With configurations of two or four processors, we have not been able to compute the efficiency for scene *rings4*, since the local memory of a processor is not large enough to contain the ray tracing algorithm and its database.

These results show that the efficiency of our algorithm decreases rapidly when the number of processors increases. This is mainly due to two reasons.

1. *The increase of computation due to a larger number of regions*

   When a processor receives a ray message, it must determine the entry cell. This requires a computation involving floating point operations. The cost of this computation increases with the number of regions.

2. *The increase of communication / computation ratio*

   An increase of the number of processors makes the size of regions smaller. This reduces the synthesis time involved by a ray but increases dramatically the message traffic.

As pointed out previously, the use of a static load balancing technique consists in subsampling the screen and subdividing the scene extent into equal loaded regions. The results, presented here, are related to scene *ring3*. The subsampling has been accomplished by choosing one pixel in a grid of 8×8 pixels. This represents 1.5% of the total image. With this load balancing technique, the average load of the processors is about 82.5%.

We have also implemented the technique which consists in subdividing the scene extent into equal sized regions and assigning one region to each processor. With this technique, the average load is smaller than 20%. This result proves that the first method is, from far, more efficient. However, we have found three drawbacks for this approach :

1. The behavior of our algorithm is based on empirical choices such as the size of the subsampling grid and the $2D$ grid used for the partitioning into equal loaded regions.

2. Several regions resulting from a spatial subdivision, may share the same object thus implying repeated intersections with this object when the ray moves from region to region. A solution has yet been brought to this problem of shared objects. It consists in using a data structure named *mail box* [2]. This idea of mail box can be extended to our parallel algorithm. This extension can be made by allowing a ray message to contain informations about the intersections performed by the processor which sends this message. This mail box technique could be embedded in our algorithm but would add an overhead for each exchange of a ray message.

3. Another problem not solved by our algorithm is related to the situation in which a light source lies in a region assigned to a processor. Indeed, since the database partitioning is performed statically, this processor receives a lot of shadow ray messages. This makes difficult a uniform load distribution, and may entail a network contention since the message traffic is increased.

In conclusion, we have seen that, in case of a parallel ray tracing algorithm, partitioning the database statically is very difficult. So, as shown in the next section, a dynamic partitioning is indispensable when the aim is to ensure a uniform load balancing.

5 An object dataflow approach

To overcome the difficulties of the message passing model of programming, several studies have tried to define mechanisms for implementing a shared data model in distributed systems [4, 7, 28]. In [4] and [28], strategies for maintaining data consistency between copies of modified variables are studied in order to offer a general purpose shared memory service. The aim here is to show that the emulation of a shared memory on a DMPC is the best way to parallelize algorithms such as ray tracing which use large read-only databases with no obvious data domain decomposition. With a DMPC, a portion of each node's memory can be used to store a part of the shared database and the remaining portion as a cache to speed up slow global accesses. The notion of cache, managed by software in our case, is the core of an efficient shared memory emulation.

Various characteristics of the ray tracing algorithm led us to design a software tool to emulate a global memory access in the context of distributed memories. These characteristics are as follows:

- the huge amount of memory necessary for this algorithm makes the database load balancing as important as the computation load balancing.
- due to the coherence property and topological property of 3D objects, only a small part of the whole database is required at a given time. Thus, a caching mechanism can be efficient for our problem.
- due to illumination effects (shading, reflection, refraction), the small part of database necessary to evaluate one pixel is very difficult to determine statically. Thus a dynamic database management is appropriate.
- the calculation of an image uses the database in a read-only way, therefore the problem of data coherence management is not posed.

5.1 Distributing and accessing objects in the shared memory

For our experimentation, we have chosen a regular $3D$ grid as in [14]. with the traversal algorithm described in [1]. The database which must be shared by the processors are the photometric and geometric parameters of the objects of the scene together with the voxels of the grid. The mechanism used to manage the shared memory is called *Object Paging*. The shared memory is constituted with a set of pages where polygons and voxels are stored. The pages are equally distributed over the set of nodes without any particular strategy. A local memory of a PE is organized as shown in figure 8. Each local memory is divided into three parts: one containing the code to execute the application, another to store the pages which represents a portion of the initial database, and the free space is used as a cache memory for storing temporary pages received from the other processors.

During the synthesis task, the application (ray tracing) can potentially access the whole database through the software memory management. For each node, when a cache miss is detected, i.e. the page is neither in its local database nor in the cache memory, then a request is sent to the node responsible for this page. When the node

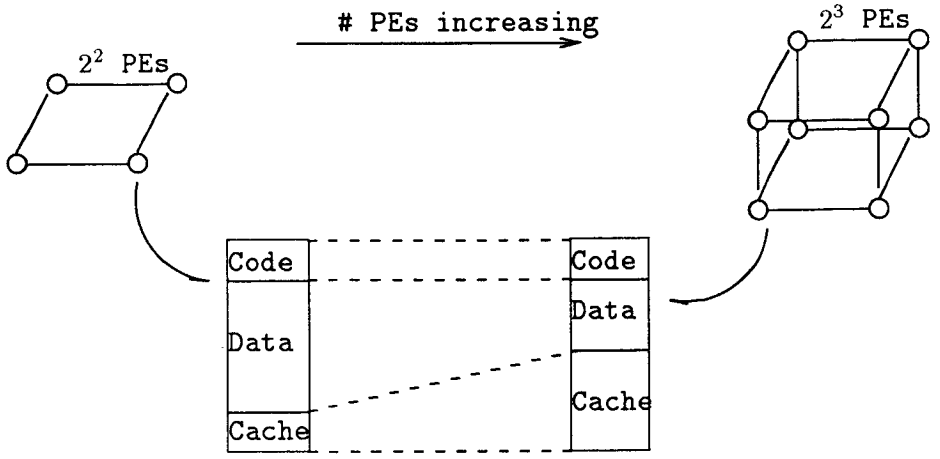

Figure 8: A user node memory description.

receives the page, it stores it in its cache memory according to a LRU (Last Recently Used) policy. This search is done during the communication of the new page, and thus causes no extra cost (cf. Fig 9).

5.2 Work distribution

With a shared database, ensuring a load balancing is quite simple. Each PE is owner of a part of the bitmap. For example, if we use 32 PEs to compute an image with a 512 x 512 resolution, each PE manages a 32 x 32 sub-bitmap. We use a square (or nearly square) sub-bitmap in order to exploit as much as possible the ray coherence property. If the PEs could directly address the frame buffer, a centralized control would not be necessary. As we do not have this facility on the iPSC/2, the host computer insures a global management of the different sub-bitmaps in order to fill in the frame buffer. The synthesis of each sub-bitmap requires frequent global data accesses at the beginning of the task, and the number of external requests progressively decreases as the memory cache keeps the pertinent items of the global database.

When a PE completes the computation of its sub-bitmap, it sends a request to get a *work item* (i.e a set of pixels) from a PE still working on its own sub-bitmap. This

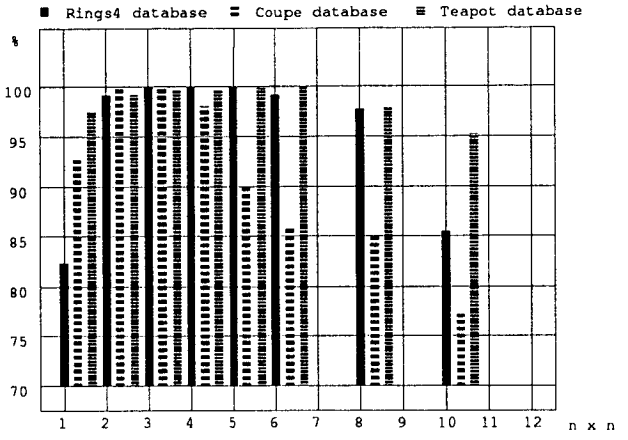

Figure 10: Relative efficiency using different size for a *work item*.

request moves along a ring topology. If this request goes back without satisfaction, the PE knows that the image computation is achieved. This local termination detection is sufficient for our application.

In order to insure a good load balancing, the only parameter to be determined is the size of this *work item*. If its size is minimal (i.e. work item = one pixel), then we have the best load balancing we can obtain, but the communication cost is then high. To take benefit of a good load balancing, we must not generate more communication activity than computation. Experimental results (see Fig. 10) show that a size of about 3 x 3 pixels offers a good compromise.

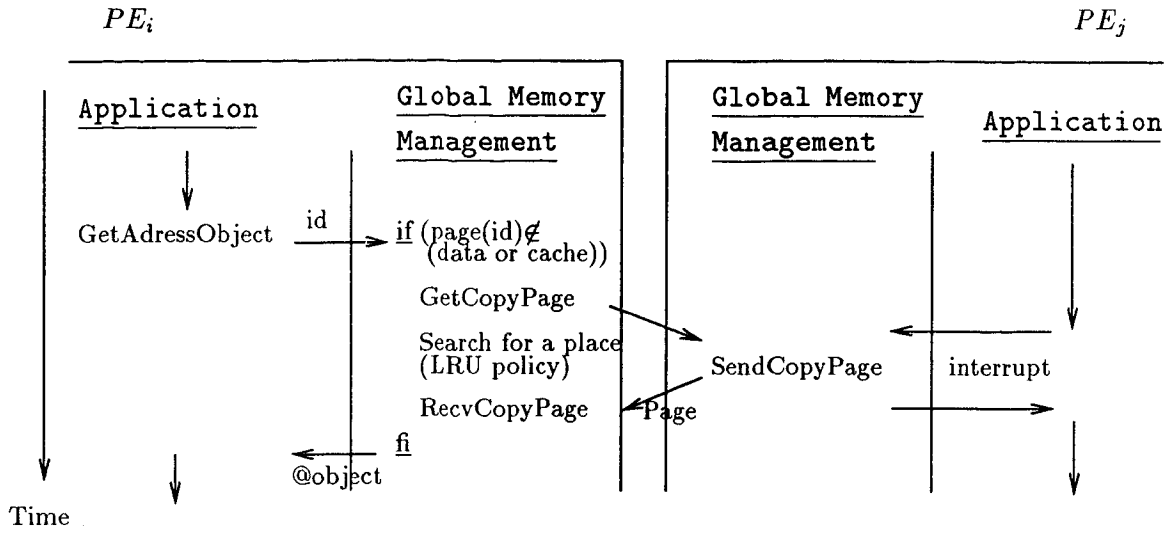

Figure 9: Accessing a global object.

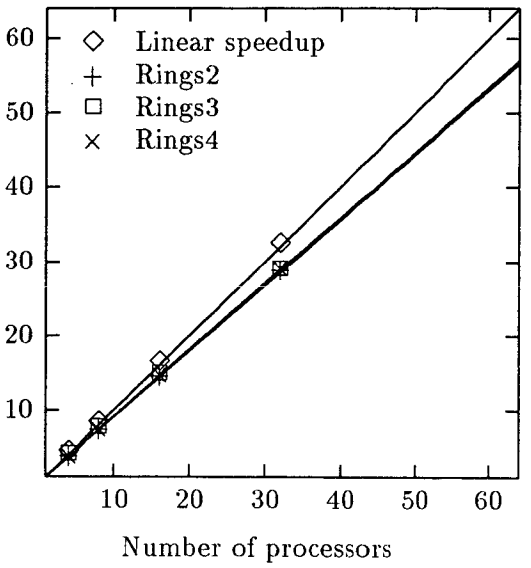

Figure 11: Speedup for the *Rings* images.

#	1	2	4	8	16	32	64
Rings2	100	95	93	91	91	90	89
Rings3	100	95	92	91	90	89	88
Rings4	100	95	93	92	91	90	88

Figure 12: Efficiency in percent for the *Rings* images.

5.3 Results and discussion

Figure 11 and 12 show the speedup and efficiency obtained with this algorithm. If we compare these results obtained with our previous work [34, 33], we can emphasize on the improvements brought by the shared database model of programming. The behaviour of this algorithm is what a user of parallel machines expects: the use of more PEs allows to solve problems faster, and to consider larger problems. This is due to the characteristics of the software global memory management:

- for a sufficient size of memory cache, the PEs can work rapidly since the number of requests to others is small;
- the size of the memory cache is flexible. Indeed, with a memory fixed-sized problem, i.e. a fixed-size database, using more PEs increases the computation power of course, but also provides a better memory management as the local cache memory increases (see Fig. 8).

A nearly similar approach has been proposed, independently of ours, by Green and Paddon in a recent paper [17]. The results given by their simulation as those given by our implementation confirm the interest of a cache mechanism for ray tracing. In a preprocessing step, they statically divide the local memory of each processor in two parts : the first one is resident while the other acts

as a cache. The difference between their method and ours is that, in our approach, the local memories are dynamically divided without requiring a preprocessing step.

One of our goals was to compute as large a database is possible. At present, we have rendered *tetra10* database which contains more than one million (1 048 576) polygons. The size of this scene and of its *objects access structure* requires the use of 109 452 pages (× 1 280 Bytes), which represents a shared memory of about 140 MBytes. The synthesis time with 64 nodes is 8 mn 46 sec. With a number of nodes ranging from 1 to 32, we ascertained that it is not possible to render this scene since the memory capacity is not sufficient (4 MBytes per node). For this reason, we had to *estimate* the speedup for 64 nodes. Its value is about 32.5.

6 Conclusion

Our goal was to design parallel ray-tracing algorithms which distribute the initial database among the local memories of the nodes of a DMPC. This feature is essential for rendering complex scenes. The two proposed algorithms meet this goal. However their performances are not similar. For scenes *rings* containing several thousands of polygons, the first algorithm seems to have a $O(logn)$ speedup while the second is in $O(n)$. These results can be explained by the fact that the first algorithm has two major drawbacks.

First, objects which are shared by several processors imply repeated intersection and decrease the efficiency of the algorithm when the number of processors increases. Solving this problem is difficult because a ray is processed by several processors. So, avoiding repeated intersections would require that the ray/object intersection points must be passed from processor to processor. Our experience has shown that the implied cost would be too prohibitive. The second algorithm does not have this drawback because a ray is always computed by one processor, thus standard solution explained in [2] can be applied.

Database	# Polygons	Synthesis Time
Teapot	3754	2 mn 49 sec
Coupe	15408	4 mn 02 sec
Rings4	18002	9 mn 05 sec
Tetra9	262144	2 mn 21 sec
Tetra10	1048576	8 mn 46 sec

Figure 13: Examples of synthesis time with 64 nodes and a resolution of 512×512 pixels.

The second defect concerns the light source. A processor that possesses a light source in its associated region, receives a lot of messages from the other ones. This causes a network congestion and decreases dramatically the efficiency of the algorithm. As for the second algorithm, it efficiently handles this situation since it uses cache memories. Indeed, a region that contains a light source will be frequently used by a processor and therefore it will be stored in the cache memory with a low probability to be overwritten.

This drawbacks allow us to say that algorithms based on processing with ray dataflow are not well suited to DMPC. This is due to the fact that the data domain decomposition cannot be made efficiently before the computation. Therefore, the shared model approach on DMPC is well suited because it achieves a dynamic data domain decomposition.

Last but not least, algorithms based on processing with ray dataflow are generally difficult to implement while those based on processing with object dataflow are simple, especially if the DMPC operating systems offer a software shared memory service. Such services are likely to be available soon in DMPC operating system.

References

[1] AMANATIDES, J., AND WOO, A. A fast voxel traversal algorithm for ray tracing. In *EUROGRAPHICS'87* (Amsterdam, 1987), pp. 3–9.

[2] ARNALDI, B., PRIOL, T., AND BOUATOUCH, K. A new space subdivision for ray tracing CSG modelled scenes.

Visual Computer 3, 2 (Aug. 1987), 98–108.

[3] BERGER, M., AND BOKHARI, S. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Transactions on Computers 36*, 5 (May 1987), 570–580.

[4] BISIANI, R., NOWATZYK, A., AND RAVISHANKAR, M. *Coherent Shared Memory on a Message Passing Machine.* Tech. Rep. CMU-CS-88-204, Carnegie Mellon, December 1988.

[5] BOUATOUCH, K., MADANI, M., PRIOL, T., AND ARNALDI, B. A new algorithm of space tracing using a CSG model. In *EUROGRAPHICS'87* (Aug. 1987).

[6] BOUVILLE, C., BRUSQ, R., DUBOIS, J., AND MARCHAL, I. Synthèse d'images par lancer de rayons: algorithmes et architecture. In *Premier Colloque Image* (May 1984), pp. 683–696.

[7] CARREIRO, N., AND GELERNTER, D. The s/net's linda kernel. *ACM Transactions Computer Systems* (May 1986).

[8] CASPARY, E., AND SCHERSON, I. A self balanced parallel ray tracing algorithm. In *Parallel Processing for Computer Vision and Display* (UK, january 1988), University of Leeds.

[9] CLEARY, J., WYVILL, B., BIRTWISTLE, G., AND VATTI, R. *Multiprocessor Ray Tracing.* Research Report 83/128/17, University of Calgary, October 1983.

[10] COHEN, M., AND GREENBERG, D. The hemi-cube, a radiosity solution for complex environments. *ACM Computer Graphics 19*, 3 (July 1985), 31–40.

[11] COOK, R. Stochastic sampling in computer graphics. In *SIGGRAPH'86 Tutorial* (1986), ACM.

[12] DIPPÉ, M., AND SWENSEN, J. An adaptative subdivision algorithm and parallel architecture for realistic image synthesis. In *SIGGRAPH'84* (New York, 1984), pp. 149–157.

[13] FUCHS, H. On visible surface generation by a priori tree structure. In *SIGGRAPH'80* (July 1980), pp. 149–158.

[14] FUJIMOTO, A., TANAKA, T., AND IWATA, K. ARTS: accelerated ray_tracing system. *IEEE Computer Graphics and Applications* (April 1986), 16–26.

[15] GLASSNER, A. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications 4*, 10 (Oct. 1984), 15–22.

[16] GOLDSMITH, J., AND SALMON, J. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications* (may 1987), 14–20.

[17] GREEN, S., AND PADDON, D. Eploiting coherence for multiprocessor ray tracing. *IEEE Computer Graphics and Applications* (November 1989), 12–26.

[18] GREEN, S., PADDON, D., AND LEWIS, E. A parallel algorithm and tree-based computer architecture for ray traced computer graphics. In *Parallel Processing for Computer Vision and Display* (UK, january 1988), University of Leeds.

[19] HAINES, E. A proposal for standard graphics environments. *IEEE Computer Graphics and Applications 7*, 11 (November 1987), 3–5.

[20] HECKBERT, P., AND HANRAHAN, P. Beam tracing polygonal objects. In *SIGGRAPH'84* (1984), pp. 119–129.

[21] HOARE, C. Communicating sequential processes. *Communications of the ACM 21*, 8 (1978), 666–677.

[22] JEVANS, D. Optimistic multi-processor ray tracing. In *Computer Graphics*

1989 (Proceedings of CGI'89) (Leeds, 1989), pp. 507–522.

[23] KAJIYA, J. The rendering equation. In *SIGGRAPH'86* (August 1987), SIGGRAPH, pp. 143–150.

[24] KAPLAN, M. Space-tracing, a constant time ray tracer. In *SIGGRAPH'85 tutorial on the uses of spatial coherence in ray tracing* (1985).

[25] KAY, T., AND KAJIYA, J. Ray tracing complex scenes. *ACM Computer Graphics 20*, 4 (August 1986), 269–278.

[26] KOBAYASHI, H., NAKAMURA, T., AND SHIGEI, Y. A strategy for mapping parallel ray-tracing into a hypercube multiprocessor system. In *Computer Graphics International'88* (May 1988), Computer Graphics Society, pp. 160–169.

[27] LEE, M., REDNER, R., AND USELTON, S. Statiscally optimized sampling for distributed ray tracing. *ACM Computer Graphics 19*, 3 (1985), 61–67.

[28] LI, K., AND HUDAK, P. Memory coherence in shared virtual memory systems. In *Symposium on Principles of Distributed Computing* (Calgary, Canada, 1986), ACM SIGACT-SIGOPS, pp. 229–239.

[29] NARUSE, T., YOSHIDA, M., TAKAHASHI, T., AND NAITO, S. Sight : a dedicated computer graphics machine. *Computer Graphics Forum 6*, 4 (1987), 327–334.

[30] NEMOTO, K., AND OMACHI, T. An adaptative subdivision by sliding boundary surfaces for fast ray tracing. In *Graphics Interface'86* (May 1986), pp. 43–48.

[31] NISHIMURA, H., OHNO, H., KAWATA, T., SHIRAKAWA, I., AND OMUIRA, K. Links-1: a parallel pipelined multimicrocomputer system for image creation. In *Proc. of the 10th Symp. on Computer Architecture* (1983), pp. 387–394.

[32] POTMESIL, M., AND HOFFERT, E. The pixel machine : a parallel image computer. In *SIGGRAPH'89* (1989), ACM.

[33] PRIOL, T. *Lancer de rayon sur des architectures parallèles: Etude et mise en œuvre.* PhD thesis, Institut de Formation Supérieure en Informatique et Communication, Rennes, juin 1989.

[34] PRIOL, T., AND BOUATOUCH, K. Static load balancing for a parallel ray tracing on a MIMD hypercube. *Visual Computer 5* (March 1989), 109–119.

[35] ROTH, S. Ray casting for modeling solids. *Computer Graphics and Image Processing 18*, 2 (February 1982), 109–144.

[36] SEITZ, C. The cosmic cube. *Comm. of the ACM 28*, 1 (January 1985), 22–33.

[37] SILLION, F., AND PUECH, C. A general two-pass method integrating specular and diffuse reflection. In *SIGGRAPH'89* (July 1989), SIGGRAPH, pp. 335–344.

[38] SPEER, L., DEROSE, T., AND BARSKY, B. A theorical and empirical analysis of coherent ray_tracing. *Graphics Interface'85* (May 1985), 11–25.

[39] TAKAHASHI, T., YOSHIDA, M., AND NARUSE, T. Architecture and performance evaluation of the dedicated graphics computer : sight. In *COMPINT'87* (November 1987), IEEE, pp. 153–160.

Liste des publications internes 1990

PI 508 RAY TRACING ON DISTRIBUTED MEMORY PARALLEL COMPUTERS: STRATEGIES FOR DISTRIBUTING COMPUTATIONS AND DATA.
Didier BADOUEL, Kadi BOUATOUCH, Thierry PRIOL
Janvier 1990, 16 Pages.

PI 509 STABILITY ANALYSIS AND IMPROVEMENT OF THE BLOCK GRAM-SCHMIDT ALGORITHM.
William JALBY, Bernard PHILIPPE
Janvier 1990, 24 Pages.

PI 510 TESTING FOR THE UNBOUNDEDNESS OF FIFO CHANNELS IN PROGRAMS.
Thierry JERON
Janvier 1990, 30 Pages.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique