KOAN : a versatile tool for parallelizing realistic rendering algorithms

Didier Badouel, Kadi Bouatouch, Zakaria Lahjomri, Thierry Priol

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
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 1505

Programme 1
Architectures parallèles, Bases de données,
Réseaux et Systèmes distribués

KOAN : A VERSATILE TOOL FOR PARALLELIZING REALISTIC RENDERING ALGORITHMS

Didier BADOUEL
Kadi BOUATOUCH
Zakaria LAHJOMRI
Thierry PRIOL

Septembre 1991

RR-1505

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
35042-RENNES CÉDEX
FRANCE
Téléphone: 99.84.71.00
Télex: UNIRISA 950 473 F
Télécopie: 99 38 38 32

KOAN : A versatile tool for parallelizing realistic rendering algorithms
KOAN : Un outil général pour paralléliser des algorithmes de synthèse d'images réalistes

Didier BADOUEL, Kadi BOUATOUCH, Zakaria LAHJOMRI, Thierry PRIOL
IRISA
Campus de Beaulieu- 35042 Rennes Cedex - France
e-mail :priol@irisa.fr

Juillet 1991
Publication Interne n° 598 - 28 pages - Programme I

Abstract

The production of realistic image generated by computer requires a huge amount of computation and a large memory capacity. The use of highly parallel computers allows this process to be performed faster. Distributed memory parallel computers (DMPCs), such as hypercubes or *transputer*-based machines, offer an attractive performance/cost ratio. However, several crucial problems have to be solved efficiently : load balancing, data domain decomposition. In order to overcome these problems, a tool named KOAN, which is an implementation of a shared virtual memory for a iPSC/2 hypercube, has been developed. It makes the parallelization of realistic rendering algorithms easier. An experiment with a parallel ray-tracing algorithm using KOAN is presented in order to prove the efficiency of KOAN. A Parallel radiosity algorithm, which would take benefits by KOAN, is discussed.

Résumé

Le calcul d'images réalistes par ordinateurs nécessite un nombre très important de calculs ainsi qu'une grande capacité mémoire. L'utilisation d'ordinateurs massivement parallèles permet d'accélérer ces traitements. Les ordinateurs parallèles à mémoire distribuée, tels que les hypercubes ou les machines à *transputers*, offrent un rapport performance/coût intéressant. Cependant, les problèmes cruciaux, tels que l'équilibrage des charges ou la décomposition du domaine des données, doivent être résolus. Dans le but de faciliter cette tâche, un dispositif de mémoire virtuelle partagée, appelé KOAN, a été conçus puis mis en œuvre. Il permet une parallelisation facile des algorithmes de calcul d'images réalistes. Une expérimentation sur un algorithme de lancer de rayon est présentée dans le but de prouver l'efficacité du système KOAN. Un algorithme parallèle de calcul de la radiosité, utilisant KOAN, est proposé.

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
(L. A. 227)
UNIVERSITÉ DE RENNES 1 I. N. S. A. DE RENNES

INSTITUT NATIONAL DE RECHERCHE
EN INFORMATIQUE ET EN AUTOMATIQUE
(LABORATOIRE DE RENNES)

1 Introduction

Realistic computer generated imagery is getting more and more important in several fields of application such as science engineering, architecture and animation. To attain realism, it is necessary to account for all the light transport mechanisms, that is, how each object interacts with the others. All these mechanisms are handled by a global illumination model which closely simulates the physical propagation of light through the environment.

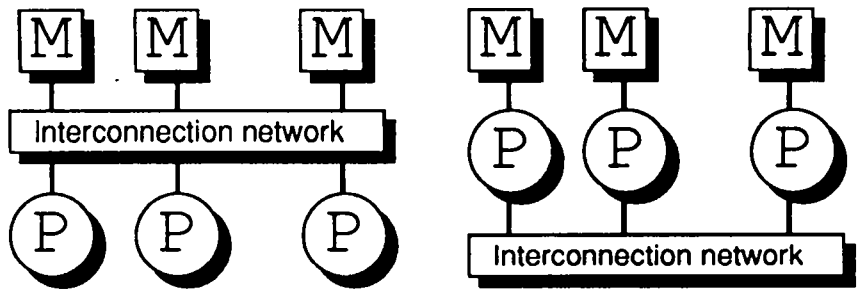
Several approaches have been tempted to implement a global illumination model [36, 61, 13, 58, 54]. The first attempt is ray tracing [62] which has the advantage of efficiently evaluating the global specular component of the radiance of a point within the scene. But the drawback of this method is its incapability to precisely evaluate the global diffuse component. This drawback is overcome by the radiosity method [29, 21, 22, 23] which allows an accurate evaluation of the global illumination effects in a perfectly diffuse environment. The third approach which combines ray tracing and radiosity [61, 57, 58, 14, 56] accomplishes realism more precisely since it handles all kinds of materials: diffuse, specular, and transparent.

Unfortunately, the realism obtained with these methods requires large resources in both computation power and memory. The use of highly parallel computers is then necessary to fulfill these requirements.

Using highly parallel computers is one way to decrease the synthesis time. These machines offer memory and computing resources which can be scaled. Intel and DARPA have announced the Touchstone project for the development of a highly parallel computer (2000 processors) with a peak performance of 150 Gflops. IBM has a similar research project with the VULCAN parallel computer that will deliver a peak performance of 1.2 Teraflops in the 90s. Inmos (SGS-Thomson) in Europe is also involved in the design of a high performance RISC processor (H1) that will replace the Transputer for building parallel computers. These new architectures will outperform supercomputers like CRAY or FUJITSU. However, the lack of tools and environments for these new architectures discourage potential users.

This paper advocates the use of a new environment, named KOAN, based on a shared virtual memory for DMPCs. This environment provides an easy way to efficiently parallelize realistic rendering algorithms. The shared virtual memory is used for accessing data manipulated by such algorithms.

The paper is organized as follows. Section 2 gives a brief background on highly parallel computers and how they can be programmed. Section 3 describes the KOAN shared virtual memory. Section 4 shows the use of KOAN for parallelizing the ray-tracing algorithm. A real implementation on an iPSC/2 hypercube allows us to give several encouraging results. Finally, some ideas for parallelizing the radiosity algorithm, and taking advantage of KOAN, are presented.



Shared Memory Parallel Computer (SMPC)

Distributed Memory Parallel Computer (DPMC)

Figure 1: MIMD architectures.

2 Background

2.1 Highly parallel computers

Large improvements in computing speed can be obtained by highly parallel computers which are made up of many microprocessors (more than a hundred). They can be either Single Instruction Multiple Data (SIMD) architectures like the well known *Connection Machine* or Multiple Instruction Multiple Data (MIMD) machines such as arrays of *transputers* or hypercube computers. Since this paper focuses only on the use of MIMD architectures, we describe them briefly. Highly parallel MIMD computers may be split into two categories depending on how the processors are connected to the memory units (Figure 1).

Shared Memory Parallel Computers (SMPCs) fall in the first category. Processors which share a single address space are connected to local memories through an interconnection network. Each processor can physically access any local memory. The network can be either a bus (e.g. SEQUENT and ENCORE computers) or a multistage network (e.g. BBN and IBM RP3 computers). Since the bandwidth of a bus is limited, the multistage network is the only way to make highly parallel shared memory computers. The cost of such a network is prohibitive for a large numbers of processor. Moreover, caches for speeding up remote memory accesses and for avoiding *hot spots* in the multistage network cannot be easily implemented.

Machines of the second category avoid these problems. As for Distributed Memory Parallel Computers (DMPCs), their design is very simple. Furthermore, they are scalable. Processors are connected together by an interconnection network which allows the exchange of messages between processors. A large number of DPMCs are available commercially. They differ in their interconnection network. Hypercube topologies are used in the Intel iPSC and the NCUBE2. Transputer-based machines like the Telmat T-Node and the Parsys SN1000 are based on a reconfigurable interconnection network allowing the simulation of a large number of topologies.

Type of parallelism	Communication	Data Structure	References
Control	Without dataflow		Nishimura et al. [44]
		Tree of extents	Bouville et al. [15] Naruse et al. [42]
	Object dataflow	Space subdivision	Green et al. [32, 30, 31] Badouel et al. [5]
		Bounding volumes	Potmesil et al. [46]
Data	Ray dataflow	Space subdivision	Dippé et al. [25] Cleary et al. [20] Nemoto et al. [43] Kobayashi et al. [39, 40] Priol et al. [48, 49, 47] Caubet et al. [17] Jevans [35]
		Tree of extents	Goldsmith et al. [28] Caspary et al. [16]
Pipeline			Ullner [60] Plunket et al. [45]

Table 1: Parallel ray-tracing algorithms.

However, there is another side of the picture: programming a DPMC is more difficult than programming a SMPC because programmers have to manage data themselves. They must partition data used by the algorithm and add message-based primitives for a remote data access. The next section describes some programming methodologies for DMPCs.

2.2 Parallel programming methodologies for DMPCs

The programming of DMPCs consists in subdividing the problem to be solved into a set of communicating tasks. The lack of *general purpose* automatic parallelization tools makes this work difficult. However, several programming methodologies exist and can be applied to sequential algorithms.

The first approach focuses on the parallelization of loops. Loops are analyzed in order to discover dependencies. A set of tasks are created that represent a subset of iterations. This approach requires that each task has an access to the data. Operating systems associated to DMPC do not offer shared memory services. Therefore, the user must take into account this lack and, in addition, he must add communication primitives to allow a task to access remote data. This approach is called *control oriented* parallelization.

The second approach consists in partitioning the data domain of the algorithm into sub-domains. Each of them is associated with a processor. Computations are assigned to processors which own the data used by these computations. They are sent to processors

by mean of messages. In fact, this is the dual approach to the first one, and is called *data oriented* parallelization.

The last approach focuses on a functional decomposition. The original algorithm is split as a set of functions. Each function is associated with a processor. Data flows between the processors. This kind of parallelism is called *pipelining* or *systolic computing*.

A *control oriented* parallelization is not often used on DMPC because users think that emulating a shared memory is time expensive. We will see in section 4 that in some circumstances, this is not true. Therefore, the natural way for parallelizing algorithms on DMPC is to use *data oriented* and *systolic* parallelization since it requires only the exchange of messages.

As an illustration example, table 1 classifies the proposed parallel ray tracing algorithms with respect to the parallel programming methodology they use. Works in [6] shows that *control* oriented parallelization is better than *data* oriented parallelization when implementing ray-tracing on DMPC. Therefore, we have decided to promote the use of a shared virtual memory for DMPC. Next section shows KOAN which is a SVM embedded in the iPSC/2 hypercube.

3 KOAN : A Shared Virtual Memory for the iPSC/2 hypercube

3.1 Shared Virtual Memory

Shared Virtual Memory for a DMPC was first investigated by K. Li [41] for a network of workstations. A SVM is a virtual address space that is shared by a number of processes running on different processors of a distributed memory parallel computer. The address space is split into several pages as shown in figure 2. Pages are spread among local processor memories according to a mapping function. Each page has a unique identifier : an integer. A mapping function could be $m(pgid) = pgid \ modulo \ N$ where $pgid$ is the page number, N the number of processors, and the function m gives the processor number associated with a page. The mapping function is used to initialize the page tables of the Memory Management Unit (MMU) associated with each processor in order to provide the user with a linear address space.

Each local memory acts as a large *software cache*. When a cache miss is detected (i.e. the page is not in its local memory) then a request (i.e. a message) is sent to the processor which is the owner of this page. When the processor receives the page, it stores it in its cache memory according to a LRU (Least Recently Used) policy. Pages can be accessed either in a *read-only* or *write* mode. Multiple copies of *read-only* pages are allowed. However, to keep the SVM coherent, there is only one copy of a page in write mode. When a processor writes into a page, it sends an invalidation message to each processor which has a copy of this page. The processor that has a page in a *write* mode is the owner of this page. A write mode page cannot be flushed from the cache.

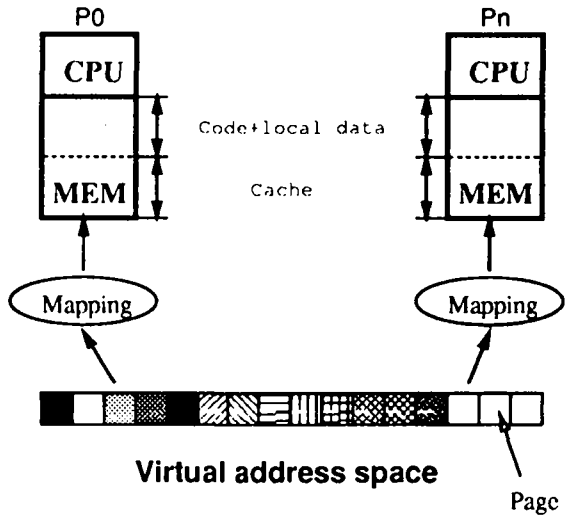


Figure 2: Mapping a Shared Virtual Memory on DMPC.

Implementing a SVM on DMPCs requires several hardware devices such as a Memory Management Unit (MMU) and a communication processor (routing). A Memory Management Unit (MMU) is needed for transforming a virtual address to physical one, while the communication processor allows the exchange of page requests, page invalidation and pages between non-neighboring processors. In addition to these hardware devices, an appropriate operating system, running in each processor, must contain specific servers to handle page requests as well as page invalidations.

Since, *Transputer* does not contain a MMU, a SVM cannot be implemented on transputer-based DMPCs.

3.2 KOAN: A Shared Virtual Memory

KOAN is an implementation of a SVM on a hypercube iPSC/2. It is embedded in the operating system. It allows the use of fast and low-level communication primitives as well as a Memory Management Unit (MMU). It differs from the work described in [41] in that it is an operating system based implementation.

3.2.1 Architecture of the iPSC/2

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 by the hypercube network. It consists of several cabinets (up to 4). Each of them houses up to 32 computational nodes. Each node consists of one Intel 80386 microprocessor augmented by an 80387 floating point co-processor and 4 Mbytes of local memory. A MMU of the Intel 80386 yields a large virtual address space instead of physical one. The virtual address space is a set of pages, each of one

has a size of 4096 bytes. A 128 nodes iPSC/2 with 16 Mbytes of local memory allows a virtual shared address space up to 2 Giga bytes. It is equipped with the Direct Connect Module (DCM) for high speed routing message between nodes. These DCM allow programmers to view the network as a complete communication graph. Each processor can send a message directly to any other processor. This is very useful for implementing our shared virtual memory because the communication graph is not known in advance.

Software development tools are available on the System Resource Manager (acting as a host processor), which is connected to node 0 via a special link. The SRM performs compilation, program loading and I/O operations for the cube. A process running on the SRM can act as an X-windows client which allows the display of images on an X-windows server connected to an ethernet network.

3.2.2 The NX/2 operating system

The operating system of the iPSC/2 consists of two parts. The first part runs on the SRM and consists of several UNIX processes. Several users can run their programs simultaneously. The operating system splits the cube into sub-cubes. Each of them is assigned to a user. Several commands have been added to allow the management of sub-cubes or parallel processes. As for the second part, it is a small kernel called NX/2 which runs on each processor of the cube. This kernel implements an asynchronous communication paradigm. Communication libraries have been added to C and FORTRAN to allow the exchange of messages between nodes and the management of parallel processes. Communication can be blocking, non-blocking or interrupt driven.

3.2.3 Using KOAN SVM

In order to implement KOAN, we have modified several parts of NX/2. Servers for processing page requests and page invalidations have been added to the NX/2 kernel, and the size of the page tables has been increased to allow a virtual address space. A library is available for the user to manage the shared virtual memory. This library is a set of routines for allocating and freeing a shared region, initializing the shared region with data from the host processor and synchronizing processes (semaphore and barrier).

Let us now outline the use of KOAN. We will focus only on how a shared address space is allocated and initialized by the processors. To dynamically allocate (resp. free) a shared region, each processor calls the routine `shared_alloc()` (resp. `shared_free()`). This latter, when called by **all** the processors, returns to them a pointer (a virtual address) to the shared region. Therefore, each processor can read or write in this shared region. With regards to the initialization of the shared region, data located on the host processor can be loaded, thanks to routine `init_svm()` on the host processor and `load_svm()` on the node processors. As an example, in case of ray-tracing, data could be objects, voxels, textures, etc... Figure 3 shows an example of using a shared region which contains a linear array.

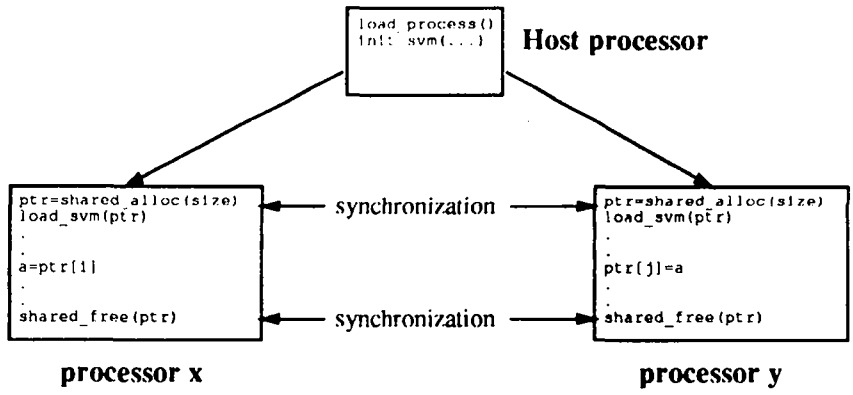


Figure 3: Using a shared region with KOAN.

4 Using KOAN for parallelizing the ray-tracing algorithm

4.1 The ray tracing principle

The ray tracing algorithm is used in computer graphics for rendering high quality images. It is based on simple optical laws which take into account effects such as shading, reflection and refraction. It acts as a light probe, following *light rays* in the reverse direction (Figure 4). The basic operation consists in tracing a ray from an *origin* point towards a *direction* in order to evaluate a light contribution. The closest intersection (*impact* point) between the ray and the scene determines the object, which contributes to this evaluation. The computation of each pixel of a simulated screen plane consists in shooting a ray from an *observer* through this pixel (*primary* rays). When an impact point is found, the contribution of various light sources to the intensity of the pixel are computed by shooting rays (*light* rays) from this point to each light source to determine if the relevant point is shadowed. According to the photometric properties of the intersected object, new rays are shot from the impact point, in order to take into account the contribution of the neighboring objects [24, 34, 62]. If the object is transparent (reflective) a ray is shot in the refracted (reflected) direction (*secondary* rays).

Geometric computations are used to find the closest intersection point between a ray and the objects in the scene. Their number increases with the *photometric* complexity of the scene (i.e. with the number of rays) and with the *geometric* complexity of the scene (i.e. with the number and the shape of the objects). Computing realistic images requires several millions of rays and several hundred thousands of objects. It is this large number of ray/object intersections which makes ray tracing a very expensive method. Several attempts have been proposed to minimize the number of ray/object intersections. These solutions are based on what we call an *object access structure* which allows a fast search for objects along a ray path. These structures are based on a tree of bounding boxes [38, 53] or on space

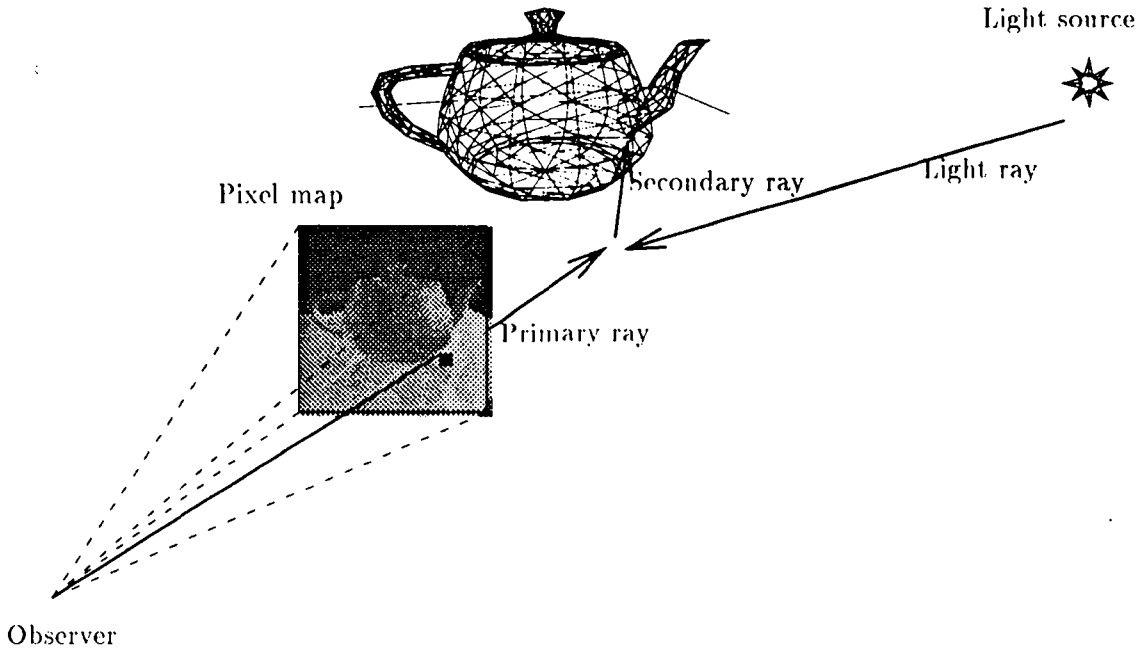


Figure 4: The ray tracing principle.

subdivision [4, 12, 26, 27, 37]. A parallelization of the ray tracing algorithm must address the problem of distributing the object access structure.

Ray tracing is intrinsically parallel since the evaluation of one pixel is independent of the others. The difficulty in exploiting this parallelism is to simultaneously ensure that the load be balanced and that the database be distributed evenly across the memory of the processors. The parallelization of such an algorithm raises a classical problem when using distributed parallel computers: how to ensure both data distribution and load balancing when no obvious relation between computation and data can be found ? This problem can be illustrated by the following schematic ray tracing algorithm :

for $i = 1, xpix$ **do**
  for $j = 1, ypix$ **do**
    $pixel[i,j] = \Sigma(contrib(\ldots, space[f_x(\ldots), f_y(\ldots), f_z(\ldots)], \ldots))$
  done
done

The computation of one pixel $pixel[i,j]$ is the accumulation of various light contributions $contrib()$ depending on the lighting model. Indeed, the recursive nature of the lighting model induces a dependency between the the various contributions to one pixel. $space[\ldots]$ is the data structure associated with the space subdivision. The value of f_x, f_y and f_z are unknown before the computation. Searching for all the data $space[a,b,c]$ used for the evaluation of one pixel is equivalent to the ray tracing itself. Therefore, relationships between computations

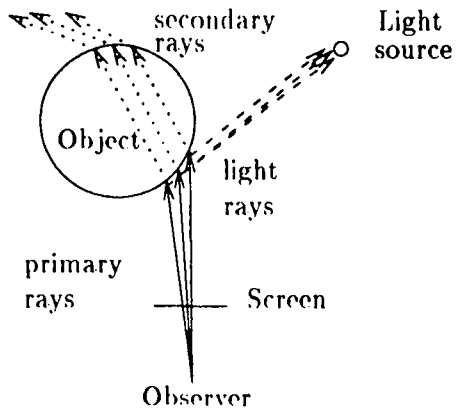


Figure 5: Ray coherence property.

and data are unknown. Using *data oriented* parallelization does not allow the computation of a data domain decomposition that ensures both a data distribution and a balanced load [6]. This leads us to consider another kind of parallelism (control oriented) which needs a shared virtual memory. This paradoxical approach for DMPCs can ensure both a distribution of the data and the workload. As described in the next section, the workload can be balanced dynamically during the running of the algorithm.

4.2 Distributing computations

This section deals with the distribution of computation among processors. This distribution must ensure that each processor does roughly the same amount of work. This can be done by distributing pixels among processors. Two approaches can be used. The first (called *static scheduling*) consists in subdividing the screen into as many parts as processors. Each part is assigned to one processor. This approach is not satisfying because the computation time needed by different pixels are not similar. Consequently, the workload is not equally distributed over the processors. The other approach (called *dynamic scheduling*) consists in assigning pixels to idle processors. As soon as a processor has completed the computation of a pixel, it asks a server for a new pixel. This solution ensures a balanced load but does not take into account the ray coherence property which can be used to reduce the remote data accesses. The ray coherence property is shown in Figure 5: two rays shot 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 these two primary rays. The dynamic scheduling technique does not ensure that the same processor treats neighboring pixels. Consequently we advocate the use of a mixture of the two techniques for a reason of efficiency. Indeed, the static scheduling technique improves the remote data accesses whereas the dynamic scheduling technique solves the load balancing problem.

As shown in Figure 6, each node owns a part of the pixel map. For example, if we use

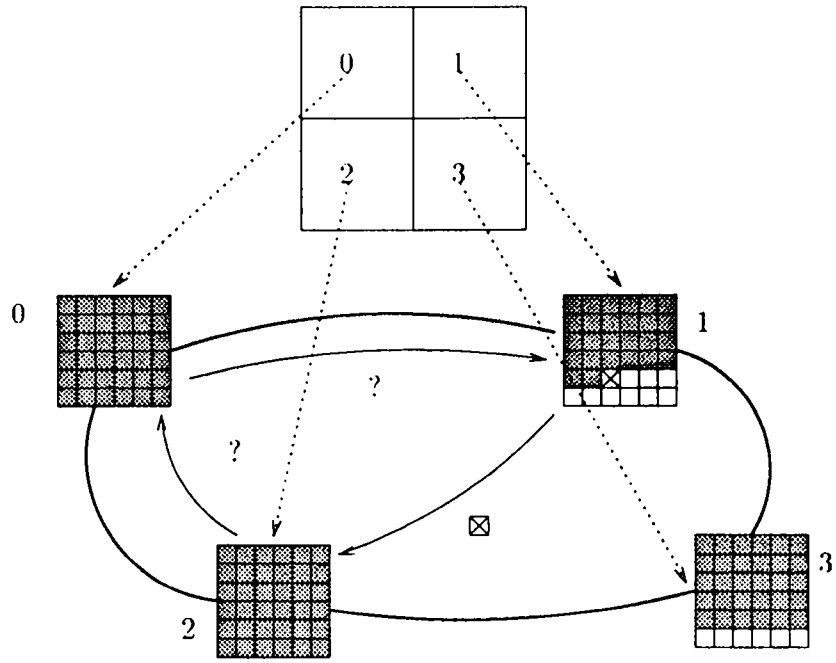


Figure 6: Pixel map distribution and dynamic load balancing.

4 nodes to compute an image with a 512×512 resolution, each node manages a 256×256 sub-pixmap. We use a square (or nearly square) sub-pixmap in order to exploit as much as possible the ray coherence property. When a node completes the computation of its sub-pixmap, it sends a request to get a *work item* (i.e a set of pixels) from a node still working on its own sub-pixmap. This request moves along a ring topology. If this request goes back without satisfaction, the node knows that the image computation is completed. This local termination detection is sufficient for our application.

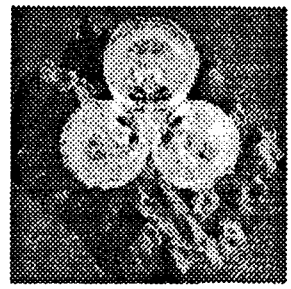
In order to insure a balanced load, the only parameter to be determined is the size of a *work item*. If its size is minimal (i.e. work item = one pixel), then we have the best balanced load we can obtain, assuming that the computation of one pixel is indivisible over the set of nodes. However, since the communication cost is high, to balance the workload, we must not generate more works in communication activity than in computation. Experimental results show that a work item of about 3×3 pixels is a good compromise.

4.3 Distributing data

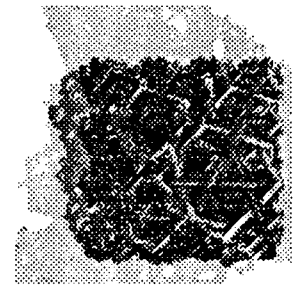
Data distribution is made by KOAN and is transparent for the user. Data which must be shared by the processors are the photometric and geometric parameters of the objects of the scenes, the grid elements (voxels) as well as textures. All these data are mapped into the virtual linear address space and are accessed by pointers as if they are located in the physical local memory of each processor.

Teapot

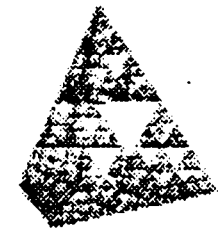
Mountain



Rings4



Tetra10



Database	# polygons	# rays	Shared memory size
Teapot	3 754	1 397 473	793 Kbytes
Mountain	9 920	1 722 415	2 031 Kbytes
Rings4	18 002	1 872 991	4 632 Kbytes
Tetra9	262 144	303 239	36 189 Kbytes
Tetra10	1 048 576	300 962	138 851 Kbytes

Table 2: Databases characteristics and the rendering result.

4.4 Analysis

Our experiments have been performed by using a set of scenes called *Standard Procedural Databases* (SPD) provided by Eric Haines [33] and the well-known *Teapot* from the university of Utah. These databases are presented in Table 2. Because of their different geometric and photometric properties, they allow to have a representative test set.

The two next sections present results given by two different experiments. Results of section 4.5 are obtained with the parallel ray tracing algorithm named *VM_pRAY* (described in [7]) for which the shared virtual memory is implemented by the user in the algorithm. This allowed us to fit in different parameters such as page size, cache size, page replacement policy, etc... However implementing a SVM in the application adds a lot of overhead, which is emphasized by results in section 4.6 obtained with an OS-based SVM such as KOAN.

4.5 Results with non OS-based SVM

Synthesis times are shown in Table 3. As a first result, the distribution of the database enables the rendering of scenes like *Tetra10* which lies far beyond the memory capacity of one node. However, it is difficult to analyze the behavior of the algorithm for such large databases and for a small number of nodes because of the limited size of the local memories. For small databases, the measurement of the parallel efficiency is straightforward while for large databases, the use of a profile analysis is required to estimate the parallel overhead. For the test set, using up to 64 nodes, we always obtain an efficiency that is better than 78%. This work is presented in [5].

1	2	4	8	16	32	64	
11522	5997	3092	1565	786	395	200	*Teapot*
17103	9038	4626	2350	1185	600	307	*Mount.*
	17926	9204	4655	2336	1181	604	*Rings4*
				295	146	78	*Tetra9*
						226	*Tetra10*

Table 3: Synthesis times (in seconds) with a non OS-based SVM.

1	2	4	8	16	32	
5927	2964	1483	743	372	187	*Teapot*
11042	5525	2766	1384	697	351	*Mountain*
		6075	2858	1413	704	*Rings4*

Table 4: Synthesis times (in seconds) with KOAN.

In this paper, we will focus on an experiment which gives measurements on the algorithm behavior as a function of the cache size. Note that the local memory of each processor used to simulate the SVM is subdivided into two parts : one for storing pages associated with the processor and the other one acts as a cache. Several experiments have shown that a better efficiency has been obtained with a page size of 1Kb. The obtained results are given in Figure 7. They give the efficiency, the miss ratio and the page communication for various sizes of the cache memory. On these graphs, 50% is a threshold below which the efficiency can be (arbitrarily) considered as insufficient. A small cache size entails a large cache fault ratio (or miss ratio) which increases the amount of communications (Figure 7).

Among the different tested databases, *Rings4* and *Tetra9* correspond to the two extreme behaviors: *Tetra9* uses a small average number of pages per pixel (17.45) and thus can be efficiently rendered with a small cache size, while *Rings4* which uses the greatest average number of pages per pixel (125.91) requires a larger cache memory to keep the efficiency threshold above 50%.

With regards to the evolution of the miss ratio, we ascertained that when using all the node memory capacity (about 3.2 Mbytes/node for the shared virtual memory management), we are from far above the miss ratio corresponding to the critical threshold, and thus far from the network saturation. This saturation has been obtained with the database *Rings4* when using only 204 pages for the cache (Figure 7.3). If we reduce the cache size once more, the communication performance decreases: this phenomenon corresponds to a network congestion similar to *Hot Spot* which appears on shared memory architectures [2].

4.6 Results with the KOAN SVM

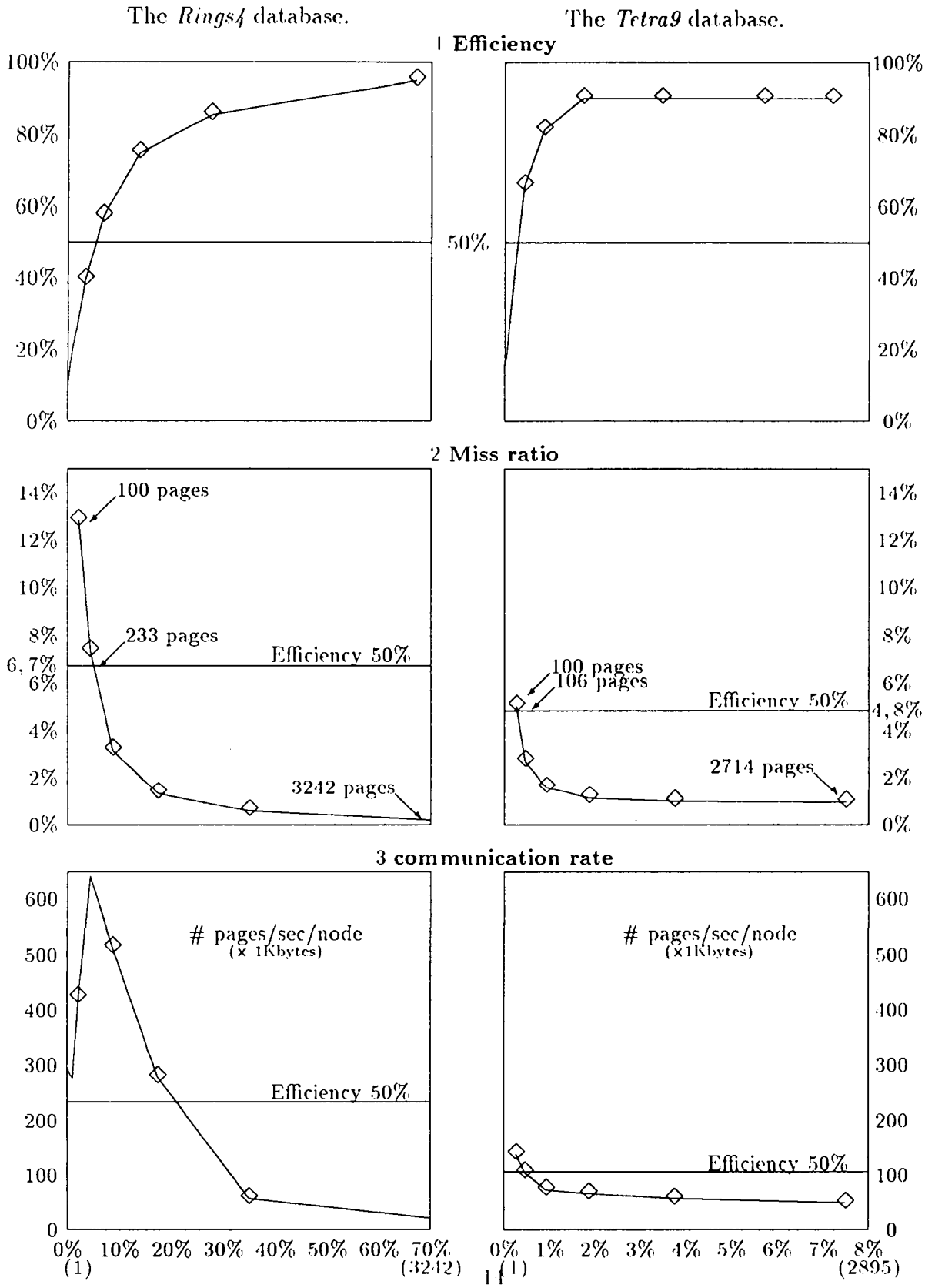


Figure 7: Efficiency, cache miss, and page communication curves with respect to cache size.

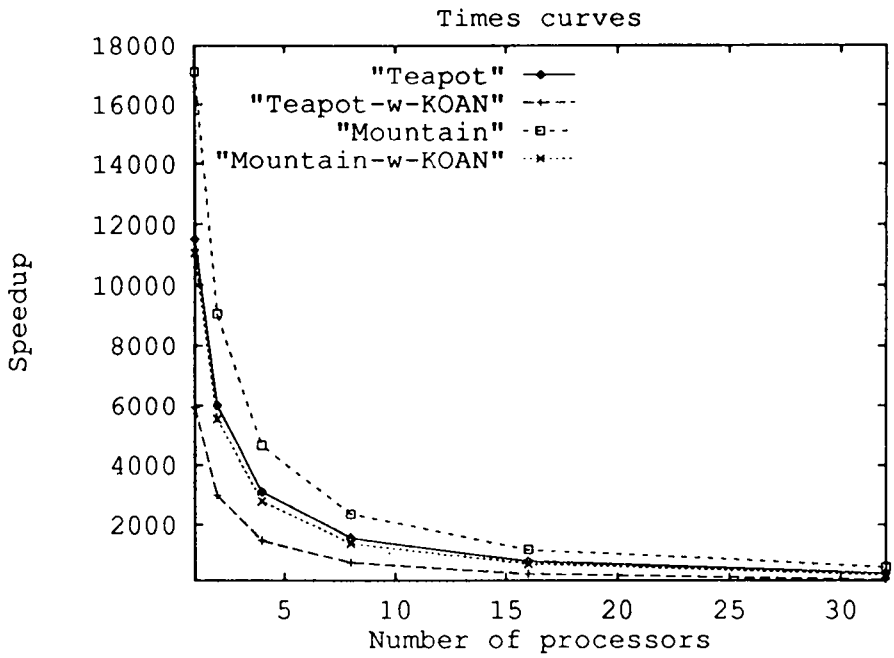


Figure 8: Comparing times for non OS-based and KOAN SVM.

This section gives some results obtained with our parallel ray-tracing algorithm which uses all the services offered by KOAN. These results allowed us to make the following remarks.

First, the size of a page is 4 Kbytes, while it is equal to 1Kbytes for the non-OS based version of the algorithm. In fact, this size is imposed by the MMU of the 80386 processor. Second, the size of cache of one processor is 2 Mbytes and the parallel machine, used to experiment with KOAN, contains only 32 processors. Third, table 4, figures 8 and figure 9 show that the speed-up for the non-OS based SVM and KOAN are similar. But the synthesis time obtained with KOAN is lower. Indeed, the synthesis time is decreased by a factor 1.51 to 2.11, according to the number of processors and to the image to be computed. This improvement is due to the fact that the pages are managed by the MMU of each processor node, and the communication primitives are faster since these latters are low level primitives of the operating system, whereas those used by the non-OS based SVM are high level primitives (belonging to the user's library).

5 KOAN and parallel radiosity

5.1 Radiosity algorithm

The radiosity method calculates the energy per unit time and per unit surface (this radiant quantity is also called radiosity) emitted by each patch in a closed environment. Given a number of N patches, their emissivity and reflectivity for each wavelength in the visible

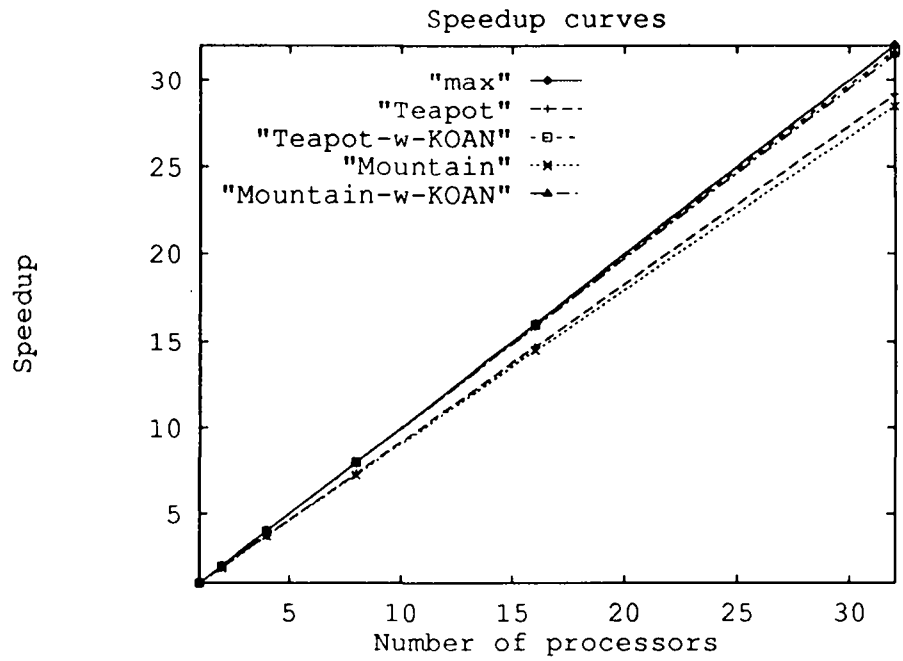


Figure 9: Comparing speedup for non OS-based and KOAN SVM.

spectrum, the relationship between the radiosities of all the patches is expressed as a set of equations:

$$B_i = E_i + \rho_i \sum_{j=1}^{j=N} F_{ji} B_j \tag{1}$$

where :

- B_i : radiosity of patch i;
- E_i : emitted energy per unit time and unit surface;
- ρ_i : reflectivity ;
- F_{ij} : form-factor;
- N : number of patches in the scene.

Form-factors F_{ji} can be computed using Hemi-cubes [21], or Hemispheres [59], or ray tracing [8], while the linear system of equations can be solved by Gauss-Siedel method (gathering), or by progressive refinement (shooting) [23].

5.2 Previous works on parallel radiosity

So far, parallel schemes of the radiosity algorithm have been proposed in the literature. The employed parallelism is based on the use of either graphics hardware [55, 9], or coarse grain computer networks [51, 19], or transputer based machines [50, 18]. These parallel algorithms are closely dependent on the form-factor calculation method or the way the linear system of equations is solved.

We propose below a parallel algorithm that efficiently exploits the advantages of KOAN.

5.3 Our parallel algorithm

Unlike our ray tracing algorithm, the parallel radiosity algorithm we propose has not been yet implemented. For these reasons, we will focus only on those parts of the algorithm which take advantage of the shared virtual memory, such as data distribution, form-factor calculations and system resolution.

5.3.1 Data distribution

As explained in the previous section describing our shared virtual memory, the patches are automatically distributed over all the processors by the host' system, according to a placement function. For example, this latter can be such that each processor is responsible for a similar number of patches. Each processor sees the whole database as a global memory without having to know where the patches are located.

5.3.2 Form-factor calculation

Each processor is responsible for a subset of patches. It calculates the form-factors between a patch of ots subset and the others. This is made possible thanks to KOAN. To efficiently use KOAN, the successive processings involved by a form-factor calculation must operate on nearly the same data in order to meet the data locality criterium. This point will be developed later in this section. Note that the form-factors of all the patches can be accessed by every processor through KOAN.

Computing the form-factors by the hemi-cube method consists in:

- placing a hemi-cube around the center of a patch;
- transforming all the patches into the 5 coordinate systems corresponding to the 5 hemi-cube' faces;
- clipping all the patches in these coordinate systems;
- scan-converting the clipped patches.

All the above processings make the hemi-cube method very time expensive. For this reason, we propose another method which uses a hemisphere and the ray-casting method which avoids all these processings.

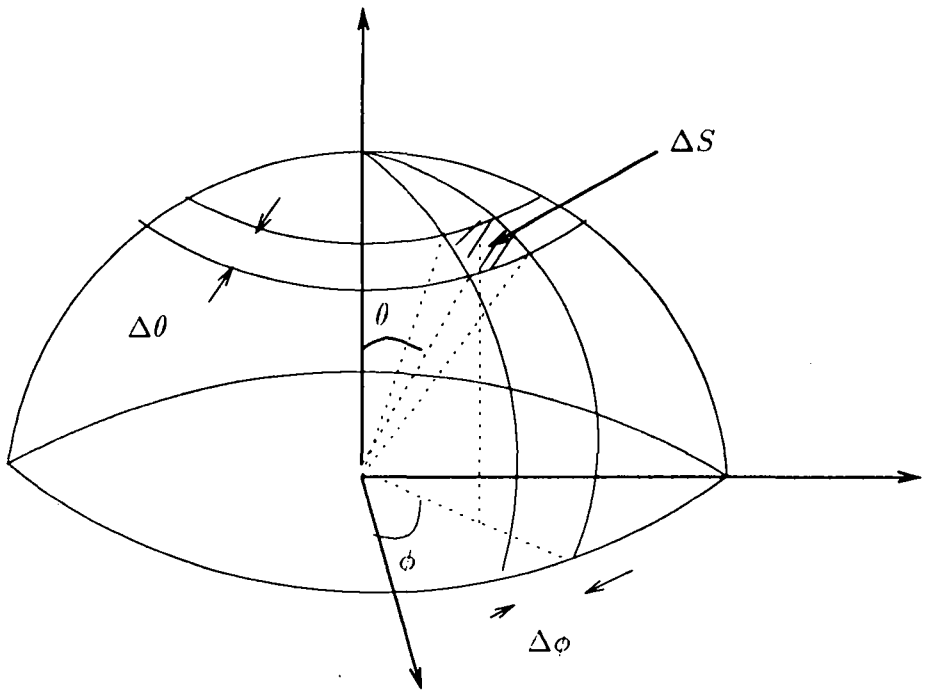


Figure 10: Hemisphere

Hemisphere

A hemisphere is placed at the center of a patch and is discretized by uniformly sampling the two polar angles θ and ϕ as shown in figure 10. The hemisphere is then discretized into surface elements ΔS of different solid angles.

Ray casting

To each surface element ΔS on the hemisphere of a patch corresponds a solid angle subtented by ΔS and whose apex is the center of the patch. These solid angles are precomputed and equal to $\sin\theta\Delta\theta\Delta\phi$, the corresponding delta form-factors are given by $(\cos\theta\sin\theta\Delta\theta\Delta\phi)/\pi$. To calculate these delta form-factors, a ray is cast from the hemisphere center and through each surface element ΔS of this hemisphere, i.e. in directions (θ_i, ϕ_i). Note that each ray corresponds to a delta form-factor. The intersection process between a ray and the scene may result in several points. Only the point closest to the ray origin is considered, and the identifier of the patch containing it is stored in an item buffer. Once all the rays have been cast from the origin of a patch i toward all directions (θ_k, ϕ_k), the form-factors from this patch are calculated by scanning the item buffer and summing the delta form-factors associated with the rays along which a particular patch j is visible.

To fully take advantage of KOAN, it is necessary to reduce the remote data accesses by exploiting the ray coherence property which satisfies the data locality principle required

by KOAN. Indeed, two subsequent rays must be close to each over. In other words, their directions may be (θ_i, ϕ_i) and $(\theta_i + \Delta\theta, \phi_i + \Delta\phi)$.

Space subdivision

To accelerate the ray casting process, the bounding volume of the scene is subdivided by the host into a $3D$ grid whose elements (named voxels) are automatically distributed over all the processors according to a placement function, in the same manner as for patches. Each processor sees the whole $3D$ grid as part of the shared virtual memory, which allows it to easily apply a grid traversal algorithm [3].

5.3.3 Solving the linear system

The linear system can be solved by applying one of the parallel solving algorithms proposed in the literature [52]. Note that the elements of the system matrix depend on the form-factors which are part of the shared virtual memory as pointed out previously.

6 Hardware perspectives : toward new architectures

The implementation of a SVM requires that each processor must be able to respond, as soon as possible, to page requests coming from other processors. Therefore, user tasks are often interrupted for replying to these requests. Special VLSI devices can be designed for doing this task. In recent papers [10, 11, 1], new distributed memory architectures are presented. They are based on the same kind of architecture design : shared virtual memory are build with hardware device (see figure 11). The network is a mesh and routing is performed by a hardware router. This kind of new architectures offers the advantages of both DMPC (simple design, scalability) and SMPC (easy programming).

7 Conclusion

In this paper, we have proposed a parallel scheme, based on a shared virtual memory, which appears paradoxical for a DMPC. Hence the Japanese name KOAN (related to paradox in Japanese) we gave to this OS-based implementation of a shared virtual memory on a iPSC/2 hypercube. This mechanism efficiently exploits all the resources of a distributed memory architectures : memory and computation.

This efficiency has been proved by the implementation of a parallel ray-tracing algorithm. Moreover, we have tried to emphasize the importance of a SVM in case of parallel radiosity.

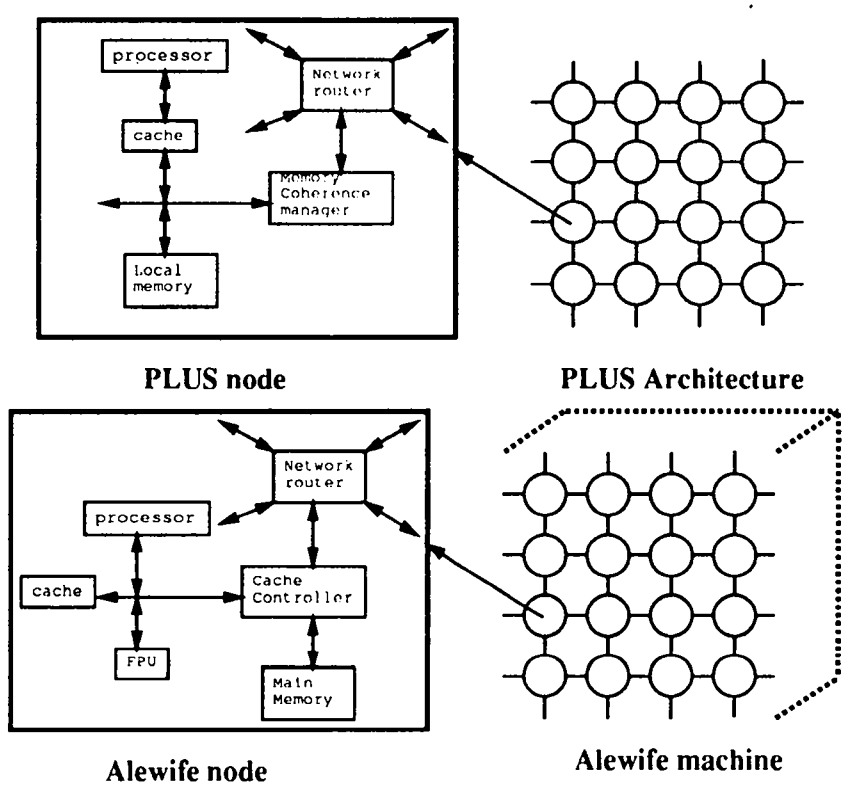


Figure 11: New distributed memory parallel computers.

8 Remarks

With regards to distribution, VM_pRAY (the parallel ray-tracing algorithm using a non-OS SVM) can be obtained by anonymous FTP on *irisa.irisa.fr (131.254.2.3)* in the directory iPSC2/VM_pRAY. Scenes in NFF format are available in iPSC2/NFF. A copy of VM_pRAY may also be obtained by sending electronic mail to either : `badouel@irisa.fr` or `priol@irisa.fr` for those who do not have access to `fnet`.

References

[1] A. Agarwal, B.H. Lim, D. Kranz, and J. Kubiatowicz. April: a processor architecture for multiprocessing. In *17th International Symposium on Computer Architecture*, May 1990.

[2] G. Almasi and A. Gottlieb. *Highly Parallel Computing. ISBN 0-8053-0177-1*, Benjamin Cummings, 1983.

[3] J. Amanatides and A. Woo. A fast voxel traversal algorithm for ray tracing. In *EUROGRAPHICS'87*, pages 3-9, Amsterdam, 1987.

[4] B. Arnaldi, T. Priol, and K. Bouatouch. A new space subdivision for ray tracing CSG modelled scenes. *Visual Computer*, 3(2):98-108, August 1987.

[5] D. Badouel. *Schémas d'exécution pour les machines parallèles à mémoire distribuée. Une étude de cas : le lancer de rayon.* PhD thesis, Université de Rennes I - IFSIC, Rennes, October 1990.

[6] D. Badouel, K. Bouatouch, and T. Priol. Ray tracing on distributed memory parallel computers: strategies for distributing computations and data. In S. Whitman, editor, *Parallel Algorithms and architectures for 3D Image Generation*, pages 185-198, ACM Siggraph'90 Course 28, August 1990.

[7] D. Badouel and T. Priol. *VM_pRAY: An efficient ray tracing algorithm on a distributed memory parallel computer.* Publication Interne 1198, INRIA, Rennes, March 1990.

[8] D. R. Baum, H. E. Rushmeier, and J. M. Winget. Improving radiosity solution through the use of analytically determined form-factors. *Computer Graphics*, 23(3):335-344, July 1989.

[9] D. R. Baum and J. M. Winget. Real time radiosity through parallel processing and hardware acceleration. In *ACM Workshop on Interactive 3D Graphics*, pages 68-76, March 1990.

[10] R. Bisiani and M. Ravishankar. Plus: a distributed shared-memory system. In *17th International Symposium on Computer Architecture*, May 1990.

[11] R. Bisiani and M. Ravishankar. Programming the plus distributed-memory system. In *Fifth Distributed Memory Computing Conference*, 1990.

[12] K. Bouatouch, M.O. Madani, T. Priol, and B. Arnaldi. A new algorithm of space tracing using a CSG model. In *EUROGRAPHICS'87*, August 1987.

[13] C. Bouville and K. Bouatouch. A unified approach to global illumination models. In *PIXIM'89 Conference*, page , September 1989.

[14] C. Bouville, K. Bouatouch, P. Tellier, and X. Pueyo. Theoretical analysis of global illumination models. In *Eurographics Workshop on Photosimulation, Realism and Physics in Computer Graphics*, pages 53-66, June 1990.

[15] C. Bouville, R. Brusq, J.L. Dubois, and I. Marchal. Synthèse d'images par lancer de rayons: algorithmes et architecture. *ACTA ELECTRONICA*, 26(3-4):249-259, 1984.

[16] E. Caspary and I.D. Scherson. A self balanced parallel ray tracing algorithm. In *Parallel Processing for Computer Vision and Display*, University of Leeds, UK, january 1988.

[17] R. Caubet, Y. Duthen, and V. Gaildrat-Inguimbert. Voxar: a tridimentional architecture for fast realistic image synthesis. In *Computer Graphics 1988 (Proceedings of CGI'88)*, pages 135–149, May 1988.

[18] A. G. Chalmers and D. J. Paddon. Parallel radiosity methods. In *OCCAM and TRANSPUTER, Research and Applications Proceedings*, pages 183–193, 1990.

[19] S.E. Chen. *A Progressive Radiosity Method and its Implementation in a Distributed Processing Environment.* Master's thesis, Cornell University, January 1989.

[20] J.G Cleary, B. Wyvill, G.M. Birtwistle, and R. Vatti. *Multiprocessor Ray Tracing.* Research Report 83/128/17, University of Calgary, October 1983.

[21] M. Cohen and D. Greenberg. The hemi-cube, a radiosity solution for complex environments. *ACM Computer Graphics*, 19(3):31-40, July 1985.

[22] 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):26–35, March 1986.

[23] Michael F. Cohen, Shenchang E. Chen, John R. Wallace, and Donald P. Greenberg. A progessive refinement approach to fast radiosity image generation. In *SIGGRAPH'88 Conference Proceeding*, pages 75–84, August 1988.

[24] R.L. Cook and K.E. Torrance. A reflectance model for computer graphics. *ACM Transactions on Graphics*, 1(1):7–24, January 1982.

[25] M. Dippé and J. Swensen. An adaptative subdivision algorithm and parallel architecture for realistic image synthesis. In *SIGGRAPH'84*, pages 149–157, New York, 1984.

[26] A. Fujimoto, T. Tanaka, and K. Iwata. ARTS: accelerated ray_tracing system. *IEEE Computer Graphics and Applications*, 6(4):16–26, April 1986.

[27] A.S. Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, 4(10):15–22, October 1984.

[28] J. Goldsmith and J. Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications*, 14–20, May 1987.

[29] M.C. Goral, K.E. Torrance, D.P. Greenberg, and B. Battaile' Modeling the interaction of light between diffuse surfaces. In *SIGGRAPH'84*, pages 213–222, ACM, July 1984.

[30] S.A. Green and D.J. Paddon. Exploiting coherence for multiprocessor ray tracing. *IEEE Computer Graphics and Applications*, 6:12–26, November 1989.

[31] S.A. Green and D.J. Paddon. A highly flexible multiprocessor solution for ray tracing. *Visual Computer*, 5(6):62–73, March 1990.

[32] S.A. Green, D.J. Paddon, and E. Lewis. A parallel algorithm and tree-based computer architecture for ray traced computer graphics. In *Parallel Processing for Computer Vision and Display*, University of Leeds, UK, january 1988.

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

[34] A. Roy Hall and Donald P. Greenberg. A testbed for realistic image synthesis. *IEEE Computer Graphics and Applications*, 3(8):10–20, November 1983.

[35] D.A.J. Jevans. Optimistic multi-processor ray tracing. In *Computer Graphics 1989 (Proceedings of CGI'89)*, pages 507–522, Leeds, 1989.

[36] J.T. Kajiya. The rendering equation. In *SIGGRAPH'86*, pages 143–150, August 1987.

[37] M.R. Kaplan. Space-tracing, a constant time ray tracer. In *SIGGRAPH'85 tutorial on the uses of spatial coherence in ray tracing*, 1985.

[38] T.L. Kay and J.T. Kajiya. Ray tracing complex scenes. *ACM Computer Graphics*, 20(4):269–278, August 1986.

[39] H. Kobayashi, T. Nakamura, and Y. Shigei. Parallel processing of an object space for image synthesis using ray-tracing. *The Visual Computer*, 3(1):13–22, February 1987.

[40] H. Kobayashi, T. Nakamura, and Y. Shigei. A strategy for mapping parallel ray-tracing into a hypercube multiprocessor system. In *Computer Graphics International'88*, pages 160–169, Computer Graphics Society, May 1988.

[41] K. Li and R. Schaefer. A hypercube shared virtual memory system. In *1989 International Conference on Parallel Processing*, pages 125–132, 1989.

[42] T. Naruse, M. Yoshida, T. Takahashi, and S. Naito. Sight : a dedicated computer graphics machine. *Computer Graphics Forum*, 6(4):327–334, 1987.

[43] K. Nemoto and T. Omachi. An adaptative subdivision by sliding boundary surfaces for fast ray tracing. In *Graphics Interface'86*, pages 43–48, May 1986.

[44] H. Nishimura, H. Ohno, T. Kawata, I. Shirakawa, and K. Omuira. Links-1: a parallel pipelined multimicrocomputer system for image creation. In *Proc. of the 10th Symp. on Computer Architecture*, pages 387–394, 1983.

[45] D. Plunkett and M. Bailey. The vectorisation of a ray tracing algorithm for improved execution speed. *IEEE Computer Graphics and Applications*, 5(8):52–60, August 1985.

[46] M. Potmesil and E.M. Hoffert. The pixel machine : a parallel image computer. In *SIGGRAPH'89*, ACM, Boston, 1989.

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

[48] T. Priol and K. Bouatouch. Experimenting a parallel ray_tracing algorithm on a hypercube machine. In *EUROGRAPHICS'88*, Nice France, September 1988.

[49] T. Priol and K. Bouatouch. Static load balancing for a parallel ray tracing on a mimd hypercube. *Visual Computer*, 5:109–119, March 1989.

[50] Werner Purgathofer and Michael Zeiller. Fast radiosity by parallelization. In *Eurographics Workshop on Photosimulation, Realism and Physics in Computer Graphics*, pages 173–184, June 1990.

[51] R. J. Recker, D. W. George, and D. Greenberg. Acceleration techniques for progressive refinement radiosity. In *ACM Workshop on Interactive 3D Graphics*, pages 59–66, March 1990.

[52] Yves Robert. *The impact of Vector and Parallel Architectures on the Gaussian Elimination Algorithm. Algorithms and Architectures for Advanced Scientific Computing*, Manchester University Press, Oxford Road, Manchester M13 9PL, UK, 1990.

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

[54] H. Rushmeier and K. Torrance. Extending the radiosity method to include reflecting and translucent materials. *ACM Transaction on Graphics*, 9(1):1–27, January 1990.

[55] Holly E. Rushmeier, D. Baum, and D. E. Hall. Accelerating the hemi-cube algorithm for calculating form-factors. In *ASME Heat Transfer Conference*, pages 45–52, June 1990.

[56] B. Le Saeck and C. Schlick. A progressive ray tracing based radiosity with general reflectance functions. In *Eurographics Workshop on Photosimulation, Realism and Physics in Computer Graphics*, pages 103–116, June 1990.

[57] M. Shao, Q. Peng, and Y. Liang. A new radiosity approach by procedural refinements for realistic image synthesis. In *SIGGRAPH'88 Conference Proceeding*, pages 93–101, ACM, August 1988.

[58] F. Sillion and C. Puech. A general two-pass method integrating specular and diffuse reflection. In *SIGGRAPH'89*, pages 335–344, Boston, July 1989.

[59] S. Spencer. The hemisphere radiosity method : a tale of two algorithms. In *Eurographics Workshop on Photosimulation, Realism and Physics in Computer Graphics*, pages 127–136, June 1990.

[60] M.K. Ullner. *Parallel machines for computer graphics*. PhD thesis, California Institute of Technology, 1983.

[61] J.R. Wallace, M.F. Cohen, and D.P. Greenberg. A two-pass solution to the rendering equation: a synthesis of ray-tracing and radiosity methods. In *SIGGRAPH'87*, pages 311–320, ACM, July 1987.

[62] T. Whitted. An improved illmination model for shaded display. *Computer Graphics and Image Processing*, 23(6):343–349, June 1980.

LISTE DES DERNIERES PUBLICATIONS PARUES EN 1991

PI 582 PROGRAMMING REAL TIME APPLICATIONS WITH SIGNAL
Paul LE GUERNIC, Michel LE BORGNE, Thierry GAUTIER, Claude LE MAIRE
Avril 1991, 36 Pages.

PI 583 ELIMINATION OF REDUNDANCY FROM FUNCTIONS DEFINED BY SCHEMES
Didier CAUCAL
Avril 1991, 22 Pages.

PI 584 TECHNIQUES POUR LA MISE AU POINT DE PROGRAMMES REPARTIS
Michel ADAM, Michel HURFIN, Michel RAYNAL, Noël PLOUZEAU
Mai 1991, 10 Pages.

PI 585 TOWARDS THE CONSTRUCTION OF DISTRIBUTED DETECTION PROGRAMS, WITH AN APPLICATION TO DISTRIBUTED TERMINATION
Jean-Michel HELARY
Michel RAYNAL
Mai 1991, 24 Pages.

PI 586 OPAC : A COST-EFFECTIVE FOATING-POINT COPROCESSOR
André SEZNEC, Karl COURTEL
Mai 1991, 26 Pages.

PI 587 ON FAILURE DETECTION AND IDENTIFICATION : AN OPTIMUM ROBUST MIN-MAX APPROACH
Elias WAHNON, ALbert BENVENISTE
Mai 1991, 24 Pages.

PI 588 BOUNDED-MEMORY ALGORITHMS FOR VERIFICATION ON THE FLY
Claude JARD, Thierry JERON
Mai 1991, 14 Pages.

PI 589 UNE APPROCHE MULTIECHELLE A L'ANALYSE D'IMAGES PAR CHAMPS MARKOVIENS
Patrick PEREZ, Fabrice HEITZ
Juin 1991, 32 pages.

PI 590 THE IDEMPOTENT SOLUTIONS OF THE SEMI-UNIFICATION PROBLEM
Pascal BRISSET, Olivier RIDOUX
Juin 1991, 16 pages.

PI 591 AVARE UN PROGRAMME DE CALCUL DES ASSOCIATIONS ENTRE VARIABLES RELATIONNELLES
Mohamed OUALI ALLAH
Juin 1991, 32 pages.

PI 592 SCHEDULING IN DISTRIBUTED SYSTEMS : SURVEY AND QUESTIONS
Yasmina BELHAMISSI, Maurice JEGADO
Juin 1991, 36 pages.

PI 593 APPLICATION OF BELLEN'S PARALLEL METHOD TO ODE's WITH DISSIPATIVE RIGHT-HAND SIDE
Philippe CHARTIER
Juin 1991, 24 pages.

PI 594 PROGRAMMATION D'UN NOYAU UNIX EN GAMMA
Pascale LE CERTEN, Hector RUIZ BARRADAS
Juillet 1991, 48 pages.

PI 595 CALCULATING THE BUSY PERIOD DISTRIBUTION OF THE M/M/1 QUEUE
Louis-Marie LE NY, Gerardo RUBINO, Bruno SERICOLA
Juillet 1991, 11 pages.

PI 596 EFFICIENT CODE GENERATION FOR DISTRIBUTED MEMORY MACHINES*
Françoise ANDRE, Olivier CHERON, Jean-Louis PAZAT, Henry THOMAS
Juillet 1991, 14 pages.

PI 597 KOAN : A SHARED VIRTUAL MEMORY FOR THE iPSC/2 HYPERCUBE
Zakaria LAHJOMRI, Thierry PRIOL
Juillet 1991, 32 pages.

PI 598 KOAN : A VERSATILE TOOL FOR PARALLELIZING REALISTIC RENDERING ALGORITHMS
Didier BADOUEL, Kadi BOUATOUCH, Zakaria LAHJOMRI, Thierry PRIOL
Juillet 1991, 28 pages.

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

ISSN 0249-6399