An Efficient Parallel Ray Tracing Scheme for Highly Parallel Architectures

Didier Badouel and Thierry Priol

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 when the load balancing has been balance and the partition of the data domain has been performed. This paper presents a parallel ray tracing algorithm for DMPC using a Shared Virtual Memory (SVM) which solves these two classical problems. This algorithm has been implemented on a hypercube iPSC/2 and results are given.

1 Introduction

The ray tracing technique is well known both for its ability to provide high quality images and its requirement in memory and computation power. Despite recent research for improving ray tracing algorithms, they are still too slow. The use of highly parallel computers is one way to decrease 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 research to develop 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 portable environment based on a shared virtual memory for DMPCs. This environment provides an easy way to efficiently parallelize the ray tracing algorithm. The shared virtual memory is used for accessing data manipulated by the ray tracing algorithm. The paper is organized as follows: Section 2 gives a brief background on ray tracing, highly parallel computers and how they can be programmed. Section 3 describes our algorithm and shows why emulating a SVM is not so paradoxical. A real implementation on an iPSC/2 hypercube allows us to present several encouraging results. Section 4 describes two approaches to efficiently implement a SVM on DMPCs.

delivered by EUROGRAPHICS DIGITAL LIBRARY www.eg.org diglib.eg.org

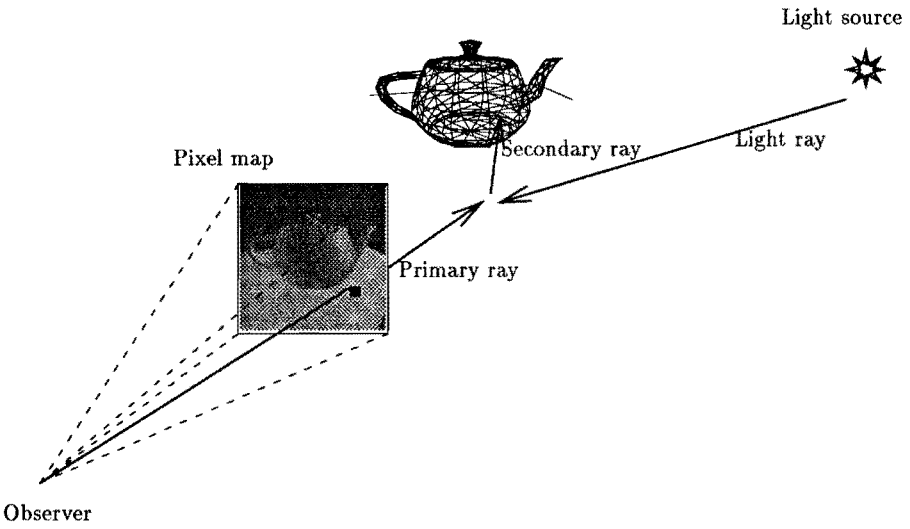


Fig. 1. The ray tracing principle

2 Backgrounds

2.1 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 effects such as shading, reflection and refraction into account. It acts as a light probe, following *light rays* in the reverse direction (Figure 1). 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, if one exists, 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 [12,21,35]. 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 million of rays and several hundred thousand 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

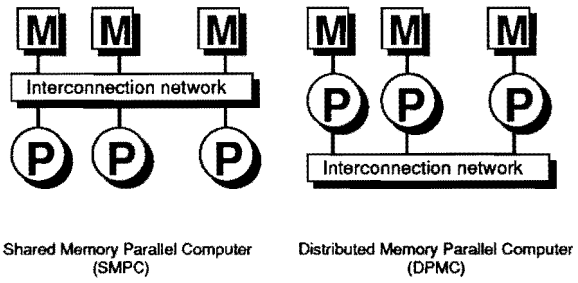


Fig. 2. MIMD architectures

boxes [24,33] or on space subdivision [2,8,14,15,23]. A parallelization of the ray tracing algorithm must address the problem of using a object access structure.

2.2 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. As this paper focuses on the use of MIMD architectures, we describe them briefly. Highly parallel MIMD computers may be split in two categories depending on how the processors are connected to the memory units (Figure 2).

Shared Memory Parallel Computers (SMPCs) constitute 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 large numbers of processors. Moreover, caches for speeding up remote memory accesses and for avoiding *hot spots* in the multistage network cannot be implemented easily.

Machines in the second category avoid these problems. The design of Distributed Memory Parallel Computers (DMPCs) is very simple. Furthermore, they are scalable. Processors are connected together by the interconnection network, which allows the exchange of messages between processors. A large number of DPMCs are available commercially. They are differentiated by 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 in order to simulate a large number of topologies.

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

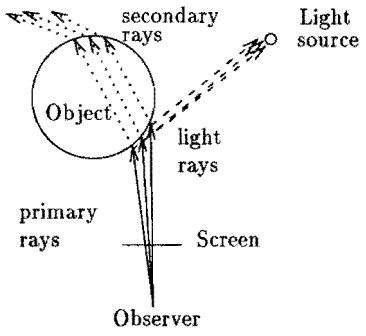


Fig. 3. Ray coherence property

2.3 Parallel Programming Methodologies for DMPCs

The programming of DMPCs consists of 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 the ray tracing algorithm. The first approach focuses on the parallelization of loops. Loops are analyzed in order to discover dependencies. A set of tasks are created that represents a subset of iterations. Communication primitives are added when a task needs remote data. This approach is called *control oriented* parallelization. The second approach consists in partitioning the data domain of the algorithm. Each sub-domain 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 of the first one, and is called *data oriented* parallelization.

Parallel ray tracing algorithms published in the literature can be grouped according to these two approaches. Algorithms based on *processing without dataflow* [9,28,30,34] or *with object dataflow* [3,17,18,19,31] have been parallelized with the first technique whereas those based on *processing with ray dataflow* [10,11,13,16,22,25,29,32] have been parallelized with the second technique. A survey of these methods is given in [4].

3 A Paradoxical Approach

Ray tracing is intrinsically parallel since the evaluation of one pixel is independent from 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 a data distribution and a balanced load 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

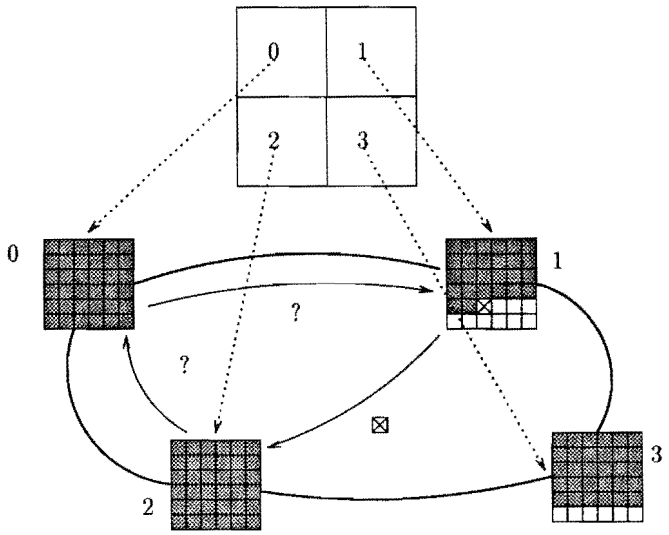


Fig. 4. Pixel map distribution and dynamic load balancing

The computation of one pixel $pixel[i,j]$ is a sum of various light contributions $contrib()$ according to the photometric model. Indeed, the recursive nature of the photometric model induces a dependency between the computation for the various contributions to one pixel. Searching all the data $space[a,b,c]$ used for the evaluation of one pixel is equivalent to the ray tracing itself.

We cannot afford to duplicate the database in every node as it will involves a severe limitation on the size of the database we can render. When choosing parallel architectures, the goals are both to speed up the execution and to be able to use larger databases.

The study of models of parallelism which can be implemented on DMPCs [3,32] leads us to advocate the use of a Shared Virtual Memory (SVM) for the parallelization of the ray tracing algorithm. In fact, this paradoxical approach for DMPCs can ensure a efficient distribution of the data while allowing all the nodes to access the entire database. Then, as described in the next section, the load can be balanced dynamically during the execution phase. This approach is classified as a *control oriented* parallelization. In the Section 3.2, we describe how the data is distributed and the management accesses to the SVM which contains the database.

3.1 Distributing Computations

In this section we consider the distribution of computation. This distribution must ensure that each processor does roughly the same amount of work. This can be done by distributing pixels among processors. Two approach can be used. The first (called *static scheduling*) consists in subdividing the screen in as many parts as processors. Each processor is responsible for computing all pixels belonging to its part of the screen. This approach is not satisfying because the computation time for every pixel is not the same and consequently the amount of work associated with each processor is not identical. The other approach (called *dynamic scheduling*) consists in assigning pixels to idle processors. As soon as a processor has finished computing a pixel, it asks a server for a new

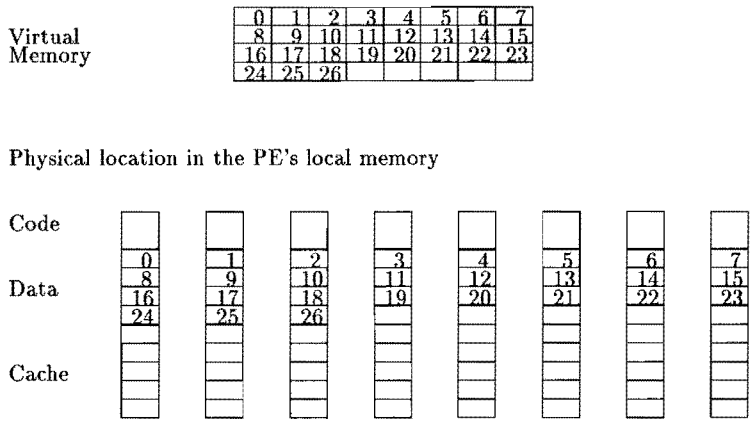


Fig. 5. Distribution of a virtual memory

pixel. This solution ensures a balanced load however it does not take into account the ray coherence property which can be used to improve remote data accesses described in the next section. The ray coherence property is showed in Figure 3: two rays shot from the observer through two adjacent pixels have a high probability of intersecting the same objects. This property is also true for all the rays spawned from the two primary ray. 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. The static scheduling technique improves the remote data accesses whereas the dynamic scheduling technique solves the load balancing problem.

As shown in Figure 4, each node owns a part of the pixel map. For example, if we use 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 achieved. 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 this *work item*. If its size is minimal (i.e. work item = one pixel), then we have the best load balance we can obtain, assuming that the computation of one pixel is indivisible over the set of nodes. However the communication cost is high. In balancing the load, we must not generate more work in communication activity than in computation. Experimental results show that a work item about 3×3 pixels is a good compromise.

3.2 Distributing Data

The use of a Shared Virtual Memory for a DMPC was first investigate by K. Li [27]. The large number of nodes allows the use of the combined local memories to store a SVM. In this distributed context, we observe two levels of data access : for each node, the set of node memories represents a larger address space than its own local memory but with slower access. As a remote data access is quite expensive with respect to a local access a cache mechanism must be used in order to increase the ratio of local data accesses.

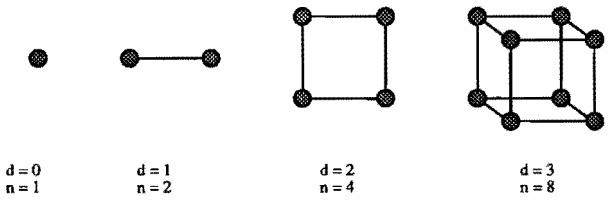


Fig. 6. The hypercube topology

In the ray tracing algorithm, the shared database contains the photometric and geometric parameters of the objects of the scene, together with the object access structure. The mechanism we use to manage the virtual shared memory is called *Object Paging* where an *object* (a polygon, a voxel of the grid ... etc) is an item of a *page*. A page is the unit for the data exchange between local memories. The paging mechanism allows uniform virtual memory management independent of the type of objects shared in this memory. An object belongs to one and only one page, and thus its memory location is contiguous.

In our algorithm, the database is first evenly distributed over the set of nodes without any particular strategy. Therefore each node's memory almost contains the same number of pages. Each local memory is divided in three parts: the process code, a part of the database, and the cache memory. The two last parts are divided into pages to allow memory management (Figure 5).

During the synthesis task, the application can potentially access the entire database through a memory management routine. 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 receive the page, it stores it in its cache memory according to a LRU (Least Recently Used) policy. This search is done during the communication of the new page, and thus causes no extra cost.

The use of these classical mechanisms, data paging and cache memory, has two main advantages: first, they dynamically exploit the data access locality of an algorithm such as ray tracing where data accesses can not be statically determined. This is an improvement parallelization factor. Second, they provide a portable environment which simplifies the parallel code and which can be used as a basis for parallelizing other algorithms.

3.3 Experimental Protocol

Results described in Section 3.4 were obtained on an Intel iPSC/2. Processors are linked together according to a hypercube topology (Figure 6). This kind of topology is characterized by a dimension d which is related to the number of processors N by the formula $N = 2^d$. Figure 6 shows how the processors are connected for different values of d.

Architecture of the iPSC/2

The iPSC/2 system consists of two main components: the cube and the system resource manager (Figure 7).

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. It is equipped with the Direct Connect Module (DCM) for high speed routing message between nodes. These communication

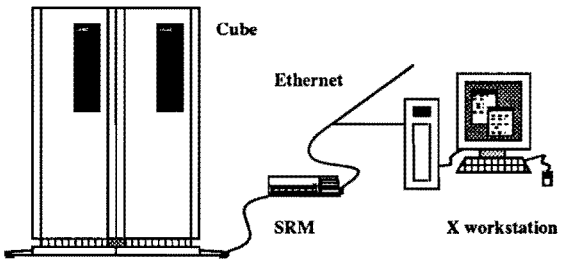


Fig. 7. The hypercube iPSC/2

processors allow programmers to view the network as a complete communication graph. Each processor can send a message directly to any other processors. This is very useful for implementing our shared virtual memory because the communication graph is not known in advance.

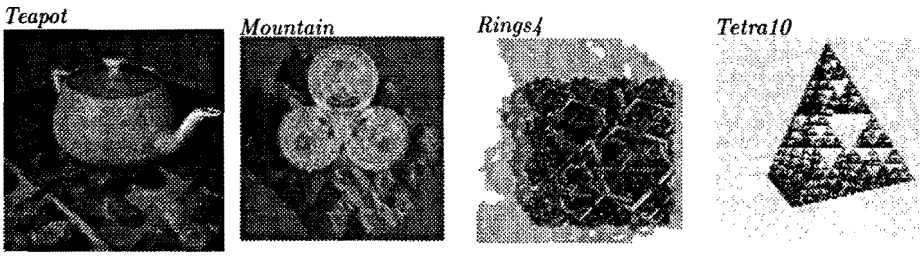
Software development tools are available on the System Resource Manager (SRM), which is connected via a special link to node 0. The SRM performs compilation, program loading and I/O operations for the cube. A process running on the SRM can act as an X-windows client which allows the display of images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

The NX/2 Operating System

The operating system of the iPSC/2 is made up 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. The second part 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. This latter method is used in our parallel ray tracing algorithm for implementing the virtual shared memory. When a processor receives a request for a page, an interrupt is sent to the user process and a user handler is executed. This handler satisfies the request by sending the page to the processor which requested it.

Experimenting on Other DMPCs

Implementing our parallel ray tracing on other DMPCs requires that messages can be sent from a processor to all other processors. This can be done by a communication processor like the iPSC/2 or by the operating system using a *store and forward* message passing mechanism. The virtual shared memory can be implemented either by a communication interrupt driven like the iPSC/2 or by *lightweight* processes. In the latter case, two such processes are needed, one for computing pixels and one for receiving and satisfying page requests. These requirements can be found in a majority of DMPCs and show that our approach is well suited for DMPCs.



Database	# polygons	# rays	Shared memory size	average pages/pixel
Teapot	3 754	1 397 473	793 Kbytes	58.19
Mountain	9 920	1 722 415	2 031 Kbytes	61.94
Rings4	18 002	1 872 991	4 632 Kbytes	125.91
Tetra9	262 144	303 239	36 189 Kbytes	17.45
Tetra10	1 048 576	300 962	138 851 Kbytes	33.00

Table 1. Databases characteristics and the rendering result

1	2	4	8	16	32	64	
3h12mn2s	1h39mn57s	51mn32s	26mn05s	13mn06s	6mn35s	3mn20s	*Teapot*
4h45mn3s	2h30mn38s	1h17mn06s	39mn10s	19mn45s	10mn00s	5mn07s	*Mount.*
	4h58mn46s	2h33mn24s	1h17mn35s	38mn56s	19mn41s	10mn04s	*Rings4*
				4mn55s	2mn26s	1mn18s	*Tetra9*
						3mn46s	*Tetra10*

Table 2. Synthesis times for an image resolution of 512 × 512 pixels

3.4 Results

Our experiments have been performed using a set of scenes called *Standard Procedural Databases* (SPD) provided by Eric Haines [20] and the famous *Teapot* from the university of Utah. These databases are presented in Table 1. Because of their geometric and the photometric diversity, they constitute a representative test set.

Synthesis times are shown in Table 2. A first result is that the distribution of the database enables the rendering of scenes like *Tetra10* which lies far beyond the memory capacity of one node. However as a result it is difficult to analyze the behavior of the algorithm for such large databases which cannot be executed with a small number of node. For small databases, the measurement of the parallel efficiency is straightforward while for the large databases it requires the use of a profile analysis to estimate the parallel overhead. For the test set, using up to 64 nodes, we always obtain an efficiency better than 78%. This work is presented in [3].

Following up on this encouraging result, in this paper, we will focus on an experiment which has given some interesting information concerning the behavior of our algorithm: We present some measurements of algorithm behavior as a function of cache size (the size of a page is 1Kbytes). These results are given in Figure 8. They illustrate the efficiency, the miss ratio and the page communication for various sizes for the cache memory. On these graphs, we represent 50% as a threshold below which the efficiency can be (arbitrarily) considered insufficient. Using a cache memory, when the number of pages becomes very

small, it results a large cache fault ratio (or miss ratio) which entails a large number of page communications (Figure 8).

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

Concerning the evolution of the miss ratio, we notice that when using all the node memory capacity (about 3.2 Mbytes/node for the shared virtual memory management), we are far from the miss ratio corresponding to the critical threshold, and thus far from the network saturation. This saturation has been obtained with the *Rings4* database when using only 204 pages for the cache (Figure 8.3). Bomans [7] have shown that the peak communication rate on the iPSC/2 is 0.9 Mbytes/sec when using message size of 1 Kbytes (our page size), while the absolute peak communication rate is 2.75 Mbytes/sec. Thus we notice that the peak performance we have measured corresponds to about 70% of the network for this message size (and around 23% with respect to the absolute peak communication rate).

If we reduce the cache size once more, the communication performance decreases: this phenomenon corresponds to a network congestion similar to the *Hot Spot* which appears when using shared memory architectures [1].

4 Implementing a SVM on DMPCs

The SVM described in this paper is implemented inside the ray tracing algorithm, therefore it is easily portable on other DMPCs. However, the management of pages, the use of high level communication primitives and satisfying page requests add substantial overheads. In order to minimize these overheads, SVM can be incorporated in the operating system or can be implemented by designing special VLSI devices.

4.1 An Operating System Approach

Incorporating the SVM inside the operating system allows the use of fast, low-level communication primitives and the Memory Management Unit (MMU) available in each node. In a paper by K. Li and R. Schaefer [27], a SVM for an iPSC/2 is presented. They use the MMU of the Intel 80386 to yield a large virtual address instead of physical addresses for memory references. The virtual address space is a set of pages, each of which 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 Gigabytes. Their results show that a kernel implementation can provide at least 23% improvement. We are working on such an implementation. Our approach differs in that it uses very low-level communication primitives which bypass the NX/2 protocol.

Unfortunately, this approach cannot be implemented on transputer-based machines due to the lack of a memory management unit.

4.2 A Hardware Approach

The implementation of a SVM requires that each processor is 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. This approach is similar to the one which consists of implementing the routing

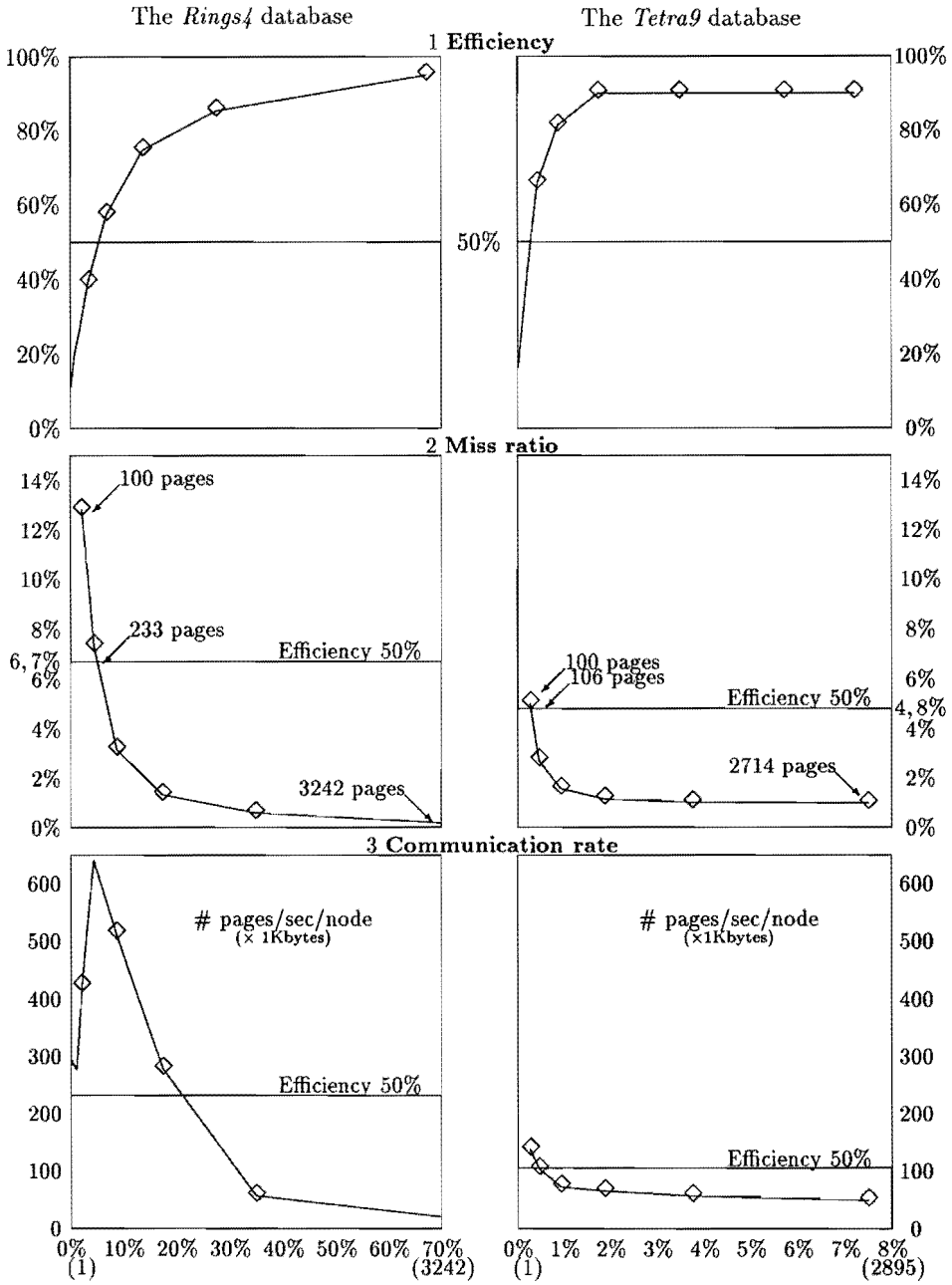


Fig. 8. Efficiency, cache miss, and page communication curves with respect to cache size

algorithm in VLSI router chip. In recent papers, R. Bisiani and M. Ravishankar present the PLUS machine [5,6] which is a distributed memory architecture. Global memory mapping and coherence management are performed by a hardware module implemented with PLD's and PAL's. The topology of the PLUS machine is a mesh. Routing is performed by a mesh router from Caltech. This architecture will offer the advantage of DMPC (simple design, scalability) and SMPC (easy programming).

5 Conclusion

In this paper, we have described a parallel scheme which appears quite paradoxical for a DMPC: the use of a Shared Virtual Memory to manage a distributed database. In fact, this mechanism efficiently exploits all the distributed resources of these architectures : computation, storage and communication resources.

Our current work concerns the implementation of the SVM in the kernel of the NX/2 operating system [26]. This implementation is based on the hardware MMU of the node processor (Intel 80386) which supports a virtual address space. We hope to obtain better absolute synthesis times by exploiting the fact that the virtual to physical address translations which were made by software will be faster using the MMU.

Remarks

Concerning distribution, VM_pRAY (the ray tracing algorithm described in this paper) 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] G. Almasi and A. Gottlieb.: *Highly Parallel Computing.* ISBN 0-8053-0177-1. Benjamin Cummings, 1983.

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

[3] 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.

[4] 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.

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

[6] R. Bisiani and M. Ravishankar.: Programming the PLUS Distributed-Memory System. In *Fifth Distributed Memory Computing Conference*, 1990.

[7] L. Bomans and D. Roose.: Communication Benchmarks for the iPSC/2. In F. André and J.P. Verjus, editors, *Hypercube and Distributed Computers*, pages 93–103, Rennes, France, October 1989. INRIA.

[8] 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.

[9] 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.

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

[11] J. G. Cleary, B. Wyvill, G. Birtwistle, and R. Vatti.: Multiprocessor ray tracing. Research Report 83/128/17, University of Calgary, October 1983.

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

[13] 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.

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

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

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

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

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

[19] S. Green, D. 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*, UK, January 1988. University of Leeds.

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

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

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

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

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

[25] 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.

[26] Z. Lahjomri.: Mise en œuvre d'une mémoire virtuelle distribuée sur l'iPSC/2. Rapport de DEA. Institut de Formation Supérieure en Informatique et Communication (IFSIC). Rennes, September 1990.

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

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

[29] 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.

[30] 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.

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

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

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

[34] T. Takahashi, M. Yoshida, and T. Naruse.: Architecture and performance evaluation of the dedicated graphics computer : SIGHT. In *COMPINT'87*, pages 153–160. IEEE, November 1987.

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