Laboratoire de l'Informatique du Parallélisme

Ecole Normale Supérieure de Lyon
Unité de recherche associée au CNRS n°1398

An optimized and load-balanced portable parallel Zbuffer

Henri-Pierre Charles
Laurent Lefèvre
Serge Miguet

October 3, 1995

Research Report N$^{\text{O}}$ 95-25

Ecole Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : (+33) 72.72.80.00 Télécopieur : (+33) 72.72.80.80
Adresse électronique : lip@lip.ens−lyon.fr

An optimized and load-balanced portable parallel Zbuffer

Henri-Pierre Charles
Laurent Lefèvre
Serge Miguet

October 3, 1995

Abstract

This paper describes the parallel implementation of the Zbuffer algorithm on different kinds of distributed memory machines. In computer graphics domain, the Zbuffer is one of the most popular and fastest technique to generate a surfacic representation of a scene consisting of objects in a 3-dimensional world. To improve this method we develop a parallel algorithm which uses an hypercube topology, load-balancing techniques and portable global communications phases.

Keywords: Parallelism, Load-balancing methods, Synthesis

Résumé

Nous présentons dans ce rapport une implémentation parallèle d'un algorithme de synthèse d'image : le Zbuffer. Cette implémentation a été portée sur différents types de machines parallèles à mémoire distribuée. En synthèse d'image, le Zbuffer est l'une des plus populaires et rapides techniques pour générer une représentation surfacique d'une scène constituée de facettes dans un monde en trois dimensions. Pour améliorer cette technique, nous présentons un algorithme parallèle qui tire pleinement profit d'une topologie en hypercube et de méthodes d'équilibrage de charges. Cette implémentation est basée sur des phases de communications globales générales qui assurent la portabilité du Zbuffer sur différentes plateformes de développement.

Mots-clés: Parallélisme, Equilibrage de charge, Synthèse d'image

An optimized and load-balanced portable parallel Zbuffer *

Henri-Pierre Charles, Laurent Lefèvre* and Serge Miguet
Laboratoire de l'Informatique du Parallelisme
Ecole Normale Superieure de Lyon
69364 LYON Cedex 07
France
(charles, llefevre, miguet)@lip.ens-lyon.fr

October 3, 1995

Abstract

This paper describes the parallel implementation of the Zbuffer algorithm on different kinds of distributed memory machines. In computer graphics domain, the Zbuffer is one of the most popular and fastest technique to generate a surfacic representation of a scene consisting of objects in a 3-dimensional world. To improve this method we develop a parallel algorithm which uses an hypercube topology, load-balancing techniques and portable global communications phases.

1 Introduction

In recent years, many authors have studied and implemented computer graphics algorithms on parallel architectures. The algorithms giving pictures with the best degree of reality are based on ray tracing or radiosity techniques. They have the advantage of being highly parallel in nature, and calculation intensive enough to produce good speedups. But they cannot be used to compute pictures in real time. The Z-Buffer algorithm has not as many features as ray tracing but it is much faster in calculation time, and produces nevertheless pictures with a good degree of reality.

In this paper, we have studied an implementation of the Zbuffer on parallel machines and the improvements we have done to optimize the execution time. The experimentations are based on different parallel computers based on the i860 processor, the Intel iPSC860 and the Archipel Volvox. The algorithm we will describe is optimized for this specialized processor and uses intensively hypercube communications.

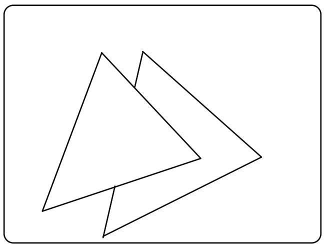
In a first part we briefly explain the sequential algorithm. Then, we present our parallel version of the Zbuffer algorithm and we discuss the improvements in terms of parallelism, optimization, load-balancing techniques and portability. Then we will finish with some experiments of our parallel implementation on medical scenes.

2 Sequential algorithm

The scene is made with a set of triangles. After the vertices have been projected in the coordinate system of the screen, the tiles have to be clipped to the size of the picture. Then, we have to determine all the pixels of the image which are inside the projected triangles. This is done by a so called scan conversion algorithm, which scans a triangle row by row. These pixels are then colored according to a given shading model. We are using the Gouraud model, where the color of a vertex is determined by the cosine between the normal to the surface and the direction of the light, and the colors of the pixels inside of the triangle result of a bilinear interpolation based on the color of the three vertices. The main problem is to eliminate the parts

*This work was supported by the Project C3 of the French Council for Research CNRS, and by the ESPRIT Basic Research Action 6632 "NANA2" of the European Economic Community.

of the objects which are hidden by others. This is what the Z-Buffer is used for: each time a pixel has been drawn in a position of the picture, we note in the same position in another picture (the Z-Buffer), the Z coordinate of that point, that is, the distance between the point and the observer. Afterwards, a pixel is drawn in a given position, if and only if, its Z coordinate is smaller than the one previously stored in that position of the Z-Buffer.



∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	6	∞	2	∞	∞	∞	∞	∞	∞
∞	∞	∞	6	5	2	2	∞	∞	∞	∞	∞
∞	∞	∞	6	6	5	2	2	∞	∞	∞	∞
∞	∞	6	6	6	5	5	2	2	∞	∞	∞
∞	∞	6	6	6	5	5	5	2	2	∞	∞
∞	7	6	6	5	5	2	2	2	∞	∞	∞
∞	7	7	∞	2	2	2	∞	∞	∞	∞	∞
∞	∞	∞	2	2	∞	∞	∞	∞	∞	∞	∞

Figure 1: Example of picture with 2 triangles and the associated Zbuffer

Then the sequential algorithm can be summarized by:

```
For All the triangles in the scene
       Project the triangle in the image coordinate system
       For all the pixel in a triangle
              If Z coordinate of pixel< Zbuffer[pixel]
                     Image[pixel] = Color(pixel)
                     Zbuffer[pixel] = Z coordinate of pixel
              EndIf
       EndFor
EndFor
```

Some parallelizations of the Zbuffer algorithm have been proposed in the last few years for shared memory machines[7] or distributed memory machines [1, 4, 6]. But few of these implementations use load-balancing techniques and many of them are limited by their large communication times. In this paper, we propose a new technique allowing to load-balance the work among the processors and to make a clever use of hypercube communications to lower the overhead of the algorithm. For sake of portability, all our communication schemes are expressed with help of P.P.C.M., our parallel portable communication module.

3 Portability

To specify our algorithm, the zbuffer is defined by sequence of global communications and computations phases. To implement it in a portable way, we use Portable Parallel Communication Module (P.P.C.M.), which allows us to execute our applications on all our distributed-memory machines with various topologies but without any modification of the applications.

Our current project is to design a parallel library for 3D image processing. Our goal is to write a parallel program that is directly executable on all the parallel computers that are available in our environment. To this purpose, we have written a library composed of a restricted set of macro-instructions which can be used on all our differents parallel computers.

This library is oriented towards DMPCs[1]. We want this library to be portable, i.e. machine independent.

PPCM can be viewed as a set of different layers for different topologies. We have chosen the most useful topologies (Ring, Grid, Hypercube). The choice of the size of a given topology is made at execution time with a configuration program.

[1] Distributed Memory Parallel Computer

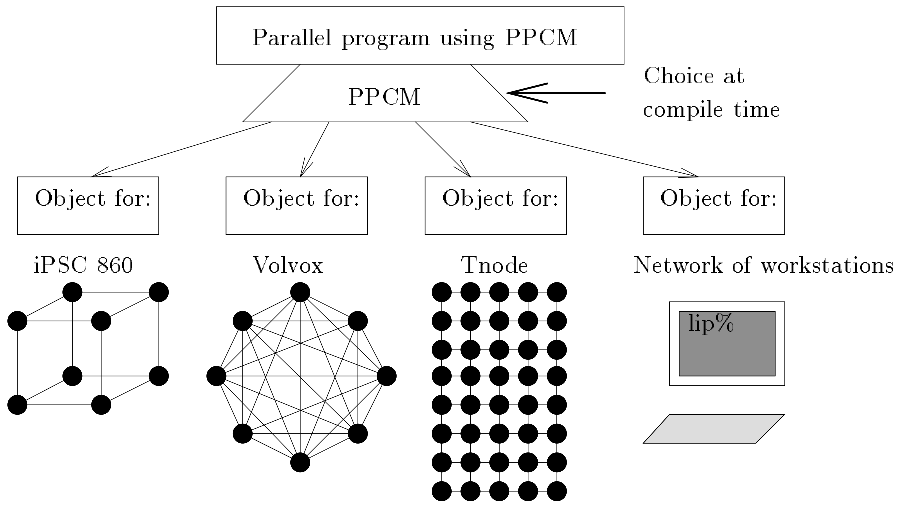


Figure 2: Synopsis of PPCM

Based on these topologies we have written a set of high level global communications libraries that are optimized for each particular interconnection scheme. This allows the programmer to write a parallel program without any knowledge of the underlying topology.

We briefly describe the main functionnalities available in these libraries:

Global Communication is used to gather or scatter data in the parallel computer. We have implemented the classical communication operations like broadcast, scattering, all to all for constant size messages or variable size messages.

LoadBalance allows to make simple but efficient load balance methods based on a data balancing in function of the load of each processor. These load balancing facilities are very useful for our image processing applications.

Reduction is used for the simulation of a global memory with an operator applied on the data. The operator can be and addition, a multiplication, a maximum, a minimum or a logical operator.

So our programming environment PPCM allows us to compile a same program on different parallel computer with a given topology in a transparent and portable way.

4 Our parallel approach :

We based our new implementation on the thesis of H.P. Charles[3] and the works of S. Miguet and J. Li[4] who have developped a parallel version of the Zbuffer on a ring of processors. Like them, our algorithm is dedicated to a distributed memory machine, but to reduce the communication costs, we use hypercube communications.

In this part, we will describe our parallel algorithm. In fact, we are not only interested in the computing of a single picture but also to the computation of a sequence of pictures for various observer's positions (rotation, zoom...). That's why we do the communication and computing phases for all the pictures of the animation.

The first version of our algorithm is:

```
      ParallelZbuffer();
      Begin
      Scatter(Vertices);
      Scatter(Triangles);
      For all picture to compute Do
//         Project vertices from object to screen coordinate system
           MultiBroadcast(Projected Vertices);
//         LocalLoad = Estimation(Locals Triangles);
           GlobalLoad = MultiReduce(LocalLoad);
           MultiScatter(Triangles);
//         Sequential Zbuffer
           Output the picture
           EndDo
      End
```

The parallel parts of this algorithm have been marked with //. The others lines are call to global communication routines.

In order to optimize the memory and computation requirement, our scene is represented by a two-levels data structure : a set of vertices and a set of triangles. A vertex is a set of 6 real numbers which define a point in a coordinate system and a normal for this point. A triangle is a set of 3 vertices' indices (3 integers).

The instructions of our algorithm can be grouped into three sets : the classical Zbuffer instructions, the load-balancing techniques and the global communication steps. We describe the differents parts of this algorithm :

Scatter(Vertices) All the vertices of the scene are equally distributed on the parallel computer with a PPCM function (global communication part). The vertices come from a disk or from a previous computation on the parallel computer.

Scatter(Triangles) With the same library call, we equally distribute triangles on the parallel computer. Note that the triangles, of a given processor, can make reference to vertices that might not be present in the local memory of that processor.

Project vertices from object to screen coordinate system The projections are done in parallel. For each vertex we have to do a matrix vector multiplication. Furthermore, we use the normal to shade the vertices and assign it a R.G.B. color.

MultiBroadcast(Projected Vertices) We multi-distribute the projected vertices with a library call of PPCM (global communication part). After this step, each processor knows all the projected vertices of the scene even if it doesn't use them.

LocalLoad = Estimation(Locals Triangles) Each processor computes in parallel an estimation of the load due to its own triangles. We approximate the load associated with each row of the picture, with the number of triangles intersecting that row (see section 5).

GlobalLoad = MultiReduce(LocalLoad) With a call to the PPCM library, we compute the global load. This global load allows to compute for each processor which part of the picture to treat in order to have a balanced workload.

MultiScatter(Triangles) Given the image partition, we can compute the triangles required by each processor. Then, with a library call we multiscatter the triangles (global communication part).

Sequential Zbuffer We compute in parallel a sequential zbuffer for the part of the image owned by each processor.

Write the picture When all the sequential zbuffers are performed, we write the image to an output device.

We move, now, to the desciption of the dynamic load-balancing methods, we have used to improve the execution time.

5 Load-balancing :

We have seen that the scene is sliced between the processors. But the objects are seldom uniformely distrbuted on all the picture. Therefore we have to equally distribute the workload to avoid for some processors to be idle while others have a large amount of work. We use a dynamic load-balancing method, based on an elastic distribution of [5] which is composed of three parts :

- Estimation of local workload : We consider that the time to scan-convert a triangle in the final picture is divided in two parts :
  - Initialisation : The time to compute line descriptors which are useful to scan the triangle frontiers. This time is fixed and doesn't depend on the size of the triangle.
  - Scan : The time needed to scan all the pixels on the triangles to know if these pixels are visible or hidden.

  But the computing of the workload has to be very small if we don't want to decrease the performances. Then we have chosen the following heuristic where the workload associated to each row of the picture, is the number of triangles which intersect that row. To avoid the scan of all the lines of each triangle to add its unit load in the workload of the picture, we use an optimization. Let's consider $v1(x1, y1)$ the upper vertex of the triangle and $v3(x3, y3)$ the lower one.

```
For each triangle do
       Workload[y1]=Workload[y1]+1
       Workload[y3+1]=Workload[y3+1]-1
```

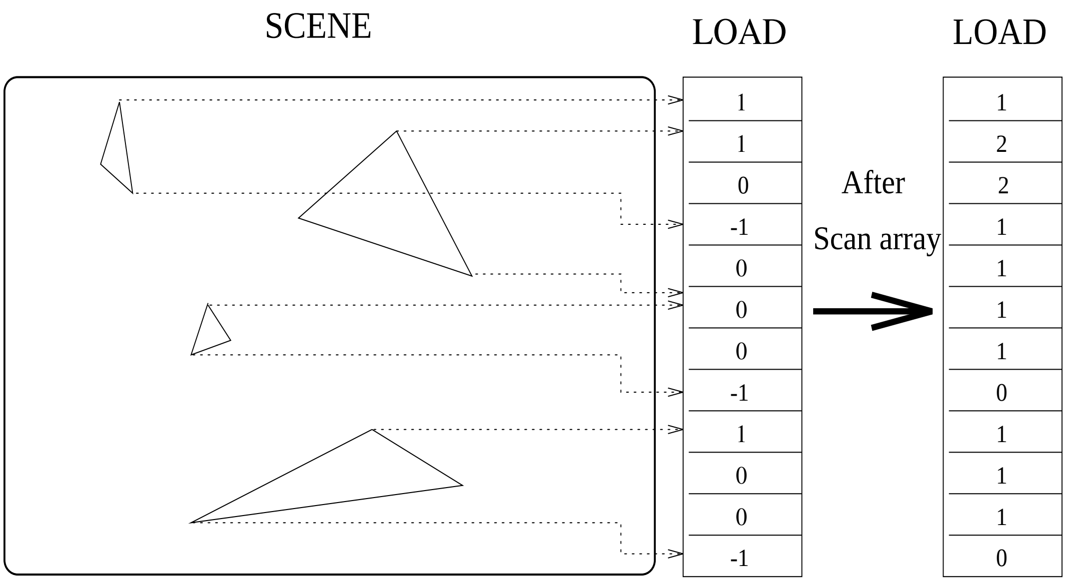


Figure 3: Computing of workload, before and after the scanning array phase

Afterwards, we only have to do a single scan of the load array, to update for each row, the number of triangles intersecting that row (see Figure 3). This array is updated in the following way :

```
For each row i of the picture do
       Workload[i]=Workload[i]+Workload[i-1]
```

- Computing of global load : Each processor must compute the global load concerning the picture to determine which part of the final scene it has to process. This is done by a multi-reduction operation of the local loads. Then, each processor knows its own slice and sends the unuseful triangles, presents in its local memory, to the real owners (multi-scattering step).

So the picture is not equally distributed between the processors in terms of rows but equally distributed in terms of work as we can see in table 1. In our first experiments, we use The University of Utah teapot, a small scene composed of 3572 triangles. We have chosen a size of 256 by 256 pixels for the output picture (see Figure 4).

Processors	Number of triangles	Workload	% of work	Number of rows
1	1304	650	26	63
2	1056	610	24	21
3	822	597	24	22
4	1056	632	25	149

Table 1: Triangles distribution and processors workload

We can note in Table 1 that the triangles that intersect the regions of two or more processors are duplicated to these processors. This explains that the sum of local triangles is larger than the total number of triangles (4238 triangles instead of 3572).

Figure 4: Example of slicing on the teapot between 4 processors

6 Experimentations :

We present next, some results of the experimentations we make with our load-balanced portable parallel Zbuffer. The first experimentations have been done on the teapot composed of 1970 vertices distributed beween 3572 triangles (see Figure 4). We have decided to compare our implementation on an hypercube topology with the one proposed by S. Miguet and J. Li[4] based on a ring of processors. The times are given for two sizes of pictures : 256 by 256 pixels and 512 by 512 pixels.

In Figure 5, we can see that the use of a parallel machine decreases widely the time of computing the Zbuffer on the teapot picture. We see the improvements between the sequential version (with one processor)

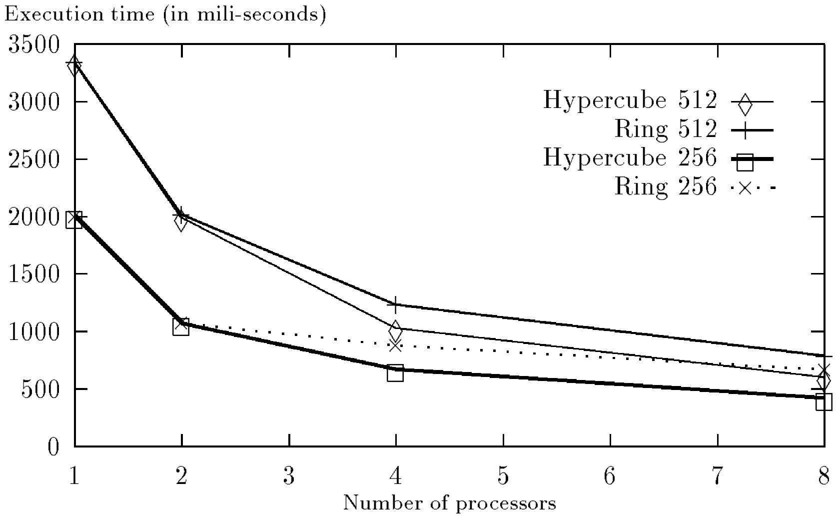


Figure 5: Execution time on teapot 256 and 512

and parallel implementation with an hypercube of dimension 1 to 3 (eight processors). When we use the ring there are small differences of time for the 256 by 256 resolution as compared to the 512 by 512 resolution. But to eficiency compare the two parallel versions, we must compare them to the sequential version. That's why, in the next figures, we plot the acceleration factor (speed-up) of our Zbuffer :

$$Speed\ up = \frac{Sequential\ version\ time}{Parallel\ version\ time}$$

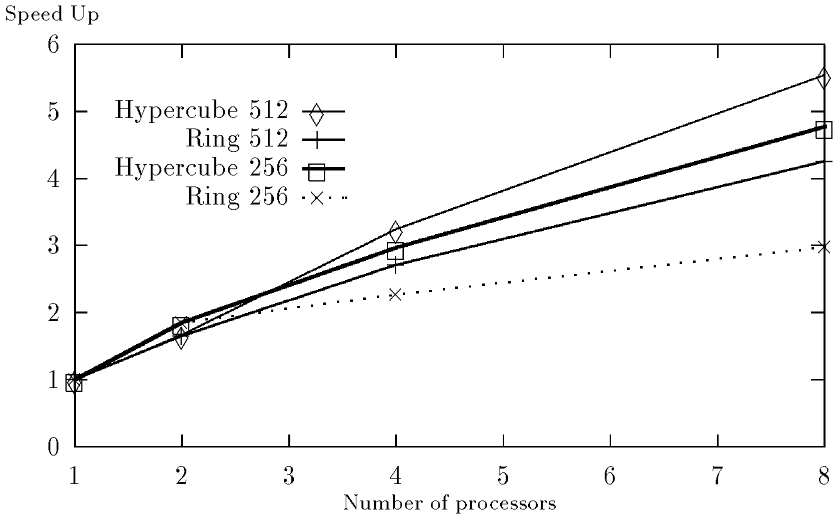


Figure 6: Speed-Up on teapot 256 and 512

For a small picture we can see on figure 6 that the acceleration factor quickly stagnates when we use a ring of processors. In a general way, we note that the bigger the picture size is, the better the acceleration factor is when we use an hypercube topology. This is due to the fact that the communication volume is independant of the resolution, whereas the computation linearly grows with the resolution. The relative importance of communications decreases with the size of the image and, since the workload is well balanced, the eficiency of our parallel Zbuffer gets close to 100%. On the teapot scene, we go 5.6 times faster than the sequential version, when we use an hypercube of dimension 3.

For the following experimentations we use a medical scene representing a skull (see Figure 8) composed of 39837 vertex distributed between 77408 triangles.

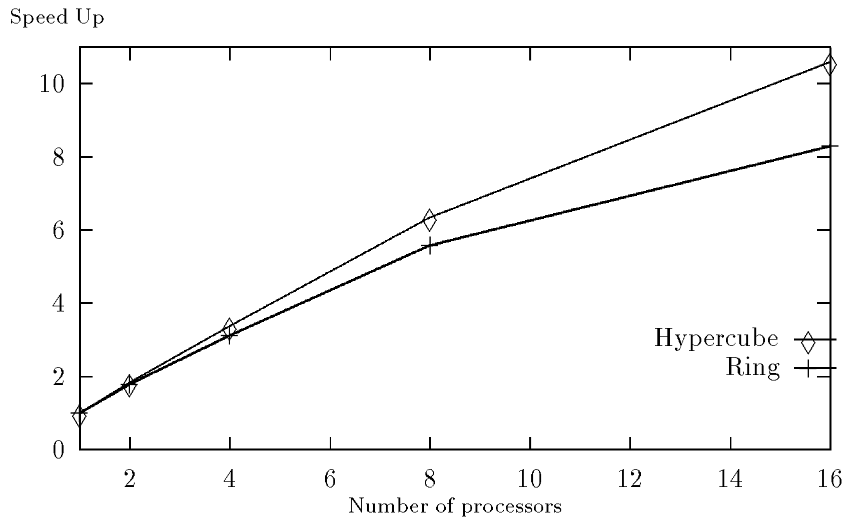


Figure 7: Speed-Up on skull scene

On the figure 7 we can see that the difference of acceleration between the two topologies are more important. When we use an hypercube of 16 processors we go 10.6 times faster while the speed-up of the teapot is limited to 8.3. The use of the hypercube topology is very interesting for this application, even if the communication time is not preponderant regarding of computing time of the Zbuffer.

7 Discussion

We have shown in the previous experiments that the relative performances of our parallel Zbuffer are good, but we also see that the absolute performances are not as interesting as we could expected. This is due to the fact that our sequential Zbuffer is not completely optimized for sake of portability. But on some of our target machine (iPSC 860 and Volvox machine), we could optimize the scanning phase to take advantage of the specialized hardware of i860 processor. For example we could hand-code the projection from the object to the screen coordinate system (use of pipelined multiplier and adder of the i860) and the interpolation part of the Zbuffer to use the graphic capabilities of the i860. Our parallel version could also be optimized by avoiding the multibroadcast step which distribute all the vertices to all the processors even if each processor doesn't need all the vertices. We could replace that by a two steps request-answer scheme, where each processor asks to its neighbours which vertices it needs and then send vertices required by other processors.

8 Conclusion and future work :

In this study we have shown that to achieve high level absolute performances it is necessary to optimize a parallel program for the specialized target topology and for a particular hardware. Parallel implementations of the Zbuffer algorithm achieve good speedup, but their absolute performances (polygons per second) cannot be compared to the performances of specialized hardware for computer graphics. But by using some hand-coded parts of the Zbuffer, we could improve the sequential time of the renderer and then optimize the parallel application.

The use of an hypercube topology and load-balancing methods have widely improved our parallel Zbuffer in terms of acceleration and eficiency. Moreover the use of our portable parallel communication module has allowed an easy developement of our application and the test of several topologies and machines in a portable way.

By parallelizing this algorithm we obtain very good performances with speedup from 9 to 10 for medical scenes (80000 triangles) with 16 processors using a hypercube topology. But our program will be more per-

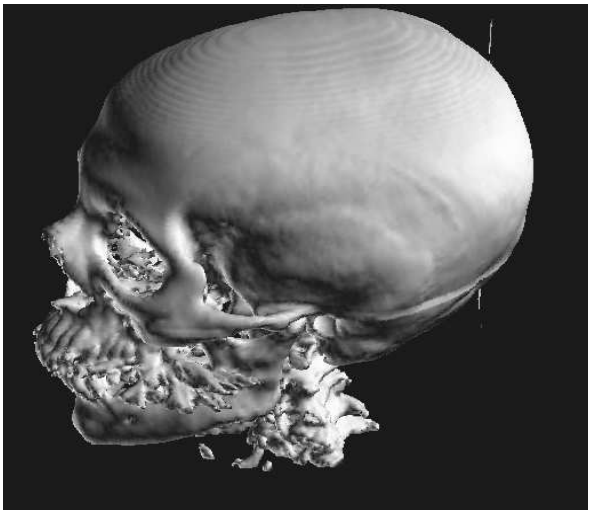


Figure 8: Example : Skull

formant for the computation of a set of images, when the polygons are already present in local memory and need only a global communication step to be correctly distributed between processors. Our next developements will concern the improvements of the Zbuffer to compute an animation with an use of a frame-to-frame coherence.

References

[1] Thomas W. Crockett, Tobias Orlof. A Parallel Rendering Algorithm for MIMD Architectures. *NASA 1-18605.* June 1991

[2] Henri-Pierre Charles. PPCM : A portable communication module. *Technical report. 92-04.* June 1992

[3] Henri-Pierre Charles. De la micro-optimisation à l'algorithme parallèle. *Thèse du LIP - Ecole Normale Supérieure de Lyon - France.* February 1993.

[4] Serge Miguet, Jian-Jin Li. Z-buffer on a transputer-based machine. *Report 90-30.* November 1990

[5] Serge Miguet, Yves Robert. Elastic load-balancing for image-processing algorithms. *Proceedings of First International Conference of the Austrian Center for Parallel Computation.* 1991

[6] T. Theoharis. Exploiting parallelism in the graphics pipeline. *Technical report PRG-54. Oxford University Computing Laboratory.* June 1986

[7] Scott Whitman. *Multiprocessors Methods for Computer Graphics Rendering.* Jones and Bartlett Publishers. 1992