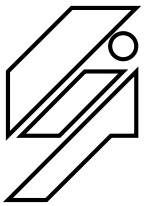


Laboratoire de l'Informatique du Parallélisme

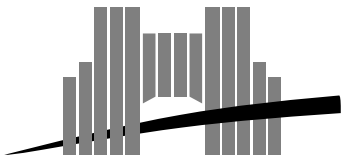
Ecole Normale Supérieure de Lyon
Unité de recherche associée au CNRS n°1398

Parallel Visualization of Texture-Mapped Digital Elevation Models

Sylvain Contassot-Vivier
Serge Miguet

February 14, 1996

Research Report N^{o} 96-02



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

Parallel Visualization of Texture-Mapped Digital Elevation Models

Sylvain Contassot-Vivier
Serge Miguet

February 14, 1996

Abstract

We propose in this paper, a parallel implementation of a ground visualization algorithm. Our input data consist in a Digital Elevation Model (DEM) covering a rectangular region, together with a raster image of the same area (the texture). The goal of the algorithm is to compute in parallel, images of the DEM from any point of view while mapping the texture onto the surface. The main originality of our approach concerns the distribution of the data, leading to a load-balanced and scalable parallel algorithm. We use a workload estimation to partition the output image, and then redistribute the input data according to this division. A special attention is payed on the data structures used for minimizing the cost of communications.

Keywords: Parallelism, 3D Visualization, Texture-Mapping, Load Balancing, Data repartition

Résumé

Nous proposons dans ce rapport, une implémentation parallèle d'un algorithme de visualisation de terrains. Les données initiales consistent en un Modèle Numérique de Terrain (MNT) couvrant une région rectangulaire, ainsi qu'une image raster de la même région (la texture). Le but de l'algorithme est de calculer en parallèle, des images du terrain de n'importe quel point de vue tout en plaquant la texture sur la surface. La principale originalité de notre approche concerne la distribution des données, menant à un algorithme ayant un bon équilibrage des charges et s'adaptant bien au nombre de processeurs utilisés. Nous utilisons une estimation de la charge pour partitionner l'image à calculer, puis nous redistribuons les données initiales sur les processeurs en fonction du partitionnement. Une attention particulière est portée sur les structures de données utilisées de façon à réduire le coût des communications.

Mots-clés: Parallélisme, Visualisation 3D, Plaquage de Texture, Equilibrage de Charges, Répartition des données

1 Introduction

In geology, 3D visualization of texture-mapped grounds is an important tool for scientists. Researchers need to visualize interactively the images of grounds from any angle, and to make measurements such as altitude reading, contour levels tracing, curvature or slope computations, etc. To fulfill these needs, we have designed the *Volter* (for *surVOL de TERrain* = ground flying over) software that includes an interactive module as well as an animation engine. The problem that occurs with actual data is that their huge size is not compatible with real time constraints. SPOT images typically present a resolution of 6000×6000 pixels.

The goal of the study presented here is to parallelize the computation of these images in order to minimize the response time of the system. We have used the *PPCM* [1] library, developed in the LIP laboratory. This library allows the use of several MIMD distributed memory machines in a portable way.

The paper is organized as follows: section 2 presents different possible strategies used in the literature for the parallelization of general visualization techniques. Section 3 gives more details on the specific data structure we use for textured ground representation, as well as on the sequential algorithm we employ for texture mapping. We explain our parallelization scheme of this algorithm in section 4, and give results on its performances in section 5. Finally, we discuss in section 6 some possible improvements of this parallel algorithm, including the handling of frame-to-frame coherence and multiresolution techniques.

2 Parallel visualization techniques

When dealing with computer graphics, we classically have to handle with two distinct spaces, namely object space (the scene to visualize), and image space (the screen where the scene has to be represented).

These two sets of data can lead to three major parallelization schemes, depending on which space is decomposed (objet, image or both).

2.1 Object space division

There are several ways to divide object space. A software or hardware pipeline can be used as in [CDH+88, OHA93, Ell94], but the main problem consists in the synchronizations which are required between each step, that can significantly slow down the process. We can also proceed as in [FK90], while processing in parallel either the objects or some sets of polygons, but we also encounter a problem of synchronization. Finally, another solution used in [CWBV83, CS89, BBP90, Cha91], is to divide object space into areas and to assign one task per area.

The main problem associated with object space approaches is the bottleneck generated by simultaneous access to the image space, when rendering in parallel different objects.

2.2 Image space division

As for the object space, there are several ways to divide the image space. We can proceed by working over the pixels or groups of pixels. These methods are the most used and have been the subject of many studies [KG79, CJ81, HF85, Rog85, Whe85, GP86, Rob88, CO91, Whi92b, Whi94, MCEF94]. The most appropriate techniques for a software issue are those which associate groups of adjacent pixels (horizontal or vertical strips or rectangles) to elementary processors.

[1] Parallel Portable Communication Module

The main problem with this approach concerns the distribution of the scene. If it is duplicated to all the local memories of the processors, redundant computations occur, leading to a non-scalable solution. Partial duplication of the scene is possible, but leads to important communication overheads. Some algorithms were made to minimize the communications by statically distributing the data as in [CJ81], but they were confronted to the problem of the load imbalance. Whelan [Whe85] and Roble [Rob88] tried to solve this problem by using a static decomposition taking into account the load balance.

The grouping of adjacent pixels to a same processor allows to take advantage of spatial coherence when rendering objects. On the other hand, it can suffer from load balancing problem, since some regions can be dramatically more loaded than others. The bloc-cyclic allocation of image data can be used to find a compromise between balancing the load and exploiting coherence.

Finally, there is another method which is based on the parallel drawing of each polygon [FFR83, All91], but there are more constraints and the parallelism is limited. Again, performances are restricted by synchronization or access conflicts problems.

2.3 Conclusion

Our bibliographical study has enabled to emphasize some important criterias for a good parallel visualization algorithm. They are:

- The granularity (size of tasks).
- The load balancing.
- The data distribution and access (memory management,...).
- The use of coherence.
- The scalability.

We could observe in the literature that no algorithm can at present, satisfy all the criterias listed above — maybe because it is not really possible. The goal of our study is to try to solve several of these issues in the particular case of the texture-mapped grounds.

Due to their lack of scalability, image-space only and object-space only solutions are unusable for large scenes and important frame-buffer resolutions. We proposed in a previous study a two-steps algorithm based on a mixed approach [CLM95]. The objects are first geometrically transformed in parallel to the screen coordinates, allowing to compute a load estimation of each row of the image. A strip-wise partition is then computed according to this estimation, so that the workload is well balanced among each region. The scene is then reshuffled according to this new image partition, and the sequential algorithm can then be applied on each subregion.

3 Sequential Visualization of DEMs

The texture-mapped grounds are composed of two input data structures. On one hand, we have a Digital Elevation Model (DEM), organized as an orthogonal mesh of points whose altitudes are given (see figure 1).

On the other hand, a raster image, representing the texture of the ground, that can be superimposed to the DEM. The number of points in the texture has typically a factor of one to ten times more than in the DEM, in each dimension. Each grid

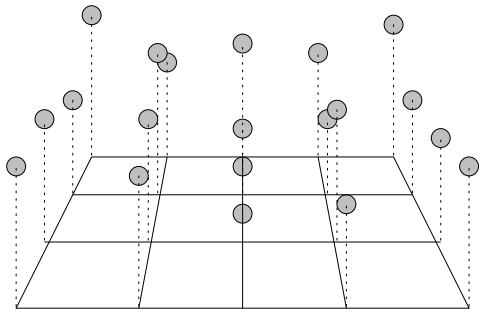


Figure 1: Digital Elevation Model

point of the DEM can be associated to a corresponding position in the texture image. A usual way to visualize the textured ground is to decompose the DEM into triangles which are scanned sequentially. Each triangle is then colored according to the the texture coordinates of its inner points, that are interpolated from the texture coordinates of its three vertices. A more detailed study over this technique can be found in [CV93]. Figure 2 shows a synthetic example where a square texture is mapped onto a sinusoidal surface.

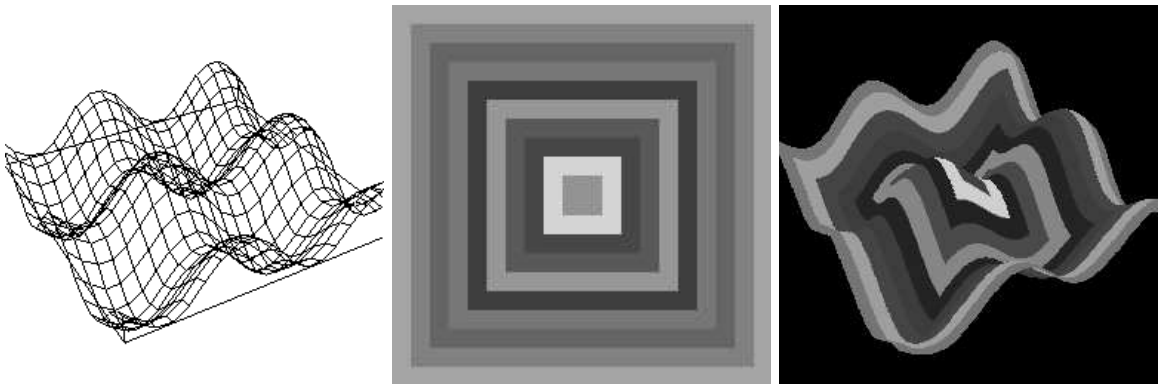


Figure 2: Example of visualization (right) starting from a DEM (left) and a texture image (middle).

4 Parallel Algorithm

According to our bibliographical studies, we chose a hybrid object and image based method similar to the one used in [CLM95]. The main originality of this new algorithm consists in the data structures used for the scene representation, and for interprocessor communications. But before entering into details, we first explain how the data are initially allocated, and give some hints on our load balancing strategy.

4.1 Load balancing

In the following, we assume that we have a p processors distributed memory machine, where processors are numbered from 0 to $p-1$.

The DEM is initially distributed using a strip-wise subdivision. Like in [CLM95], we estimate the workload associated to a given row of the picture, by counting the

number of polygons intersecting this row. Each processor does this computation for the portion of DEM it has been allocated. All the local informations are then exchanged (but not centralized) by a parallel reduction operation. An irregular strip-wise partition of the image is then computed, such that the load of each strip is approximately constant. We use the "Elastic" load balancing scheme introduced in [MR91] for computing this new partition.

In figure 3 can be seen the partition into four strips (4 processors) for a DEM of Mars [2]. The white lines represent the frontiers between the strips.

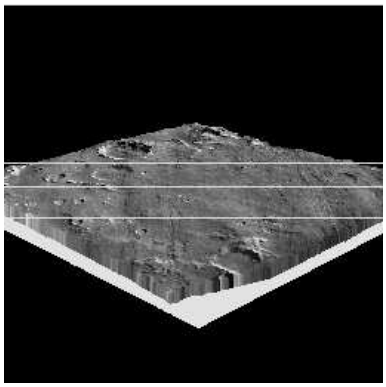


Figure 3: Division in four strips of a Mars DEM

In the following section, we explain the data structure we use for interprocessor communications. Assume first that the texture is duplicated in all the local memories of the processors. We show in section 4.3 how to remove this constraint.

4.2 Redistribution of the DEM

Once the image is partitioned, the DEM has to be redistributed among the processors according to the subdivision of the screen. In order to save memory and communication time, we want to send to a given processor, only the data it will need to perform its rendering.

A first solution would be to represent explicitly the polygons, and to use the communication scheme employed in [CLM95] for general polygon-based scenes. Nevertheless, this would amount to very large messages since we would loose the compactness of the DEM representation of the surface. Thus, we chose to redistribute DEM points themselves in a much more compact data structure, allowing nevertheless the receiver of the message to rebuild triangles.

Each processor computes p buckets for the p possible destinations of the DEM points it owns initially. A DEM point goes into bucket q if its projection falls into subregion q of the image. Additionally some points need to be added to bucket q to allow processor q to reconstitute the triangles overlapping frontiers of region q.

In the figure 4, it can be seen on the left, the chosen triangulation of the DEM and the points added to a single vertex. On the right are represented the points added to a general subset of the DEM. It is important to note that depending on the topography of the surface, the set of DEM points that are sent to a given processor might form several connected regions.

Two neighboring DEM points have a high probability to be projected into the same subregion. This remark makes it possible to code efficiently the message that a processor will send to an other one.

[2] Courtesy of the Science of Earth Department of the ENS Lyon

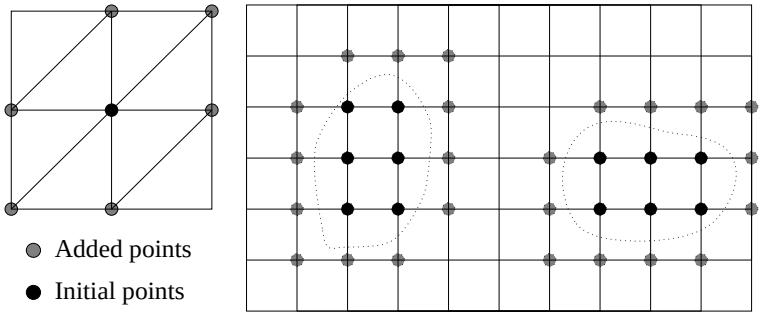


Figure 4: Triangulation of the DEM and points added to a single DEM vertex (left) and to a general set of DEM vertices (right).

A sparse structure has to be constructed in order to communicate all the data in a single message. We use for this purpose the regular grid structure of the DEM. The set of DEM points that were inserted into the same bucket, is decomposed into a set of horizontal segments. In the example of figure 4 (right hand-side), the set of points is coded with 9 horizontal segments. The message between two processors includes thus the following data: the total number of segments that have to be sent, and then the list of segments. For each segment, we include the number of points that compose it, the position of the first point in the grid of the DEM (x and y), and the list of the image coordinates (X,Y and Z) of each point in the segment. The coordinates of the points are given in the image space (which increases the size of the messages) to avoid to reproject them, since it is more time consuming on our target machines than having longer message. The two times we had to compare were on one side about 25 floating-point operations for the re-projection of a point, and on the other side, the communication of 8 additional bytes of data (three floats instead of one). This choice was made after the test of both strategies.

Finally, once the data of the DEM are reshuffled by a so-called *multi-scattering* all-to-all communication, the visualization can be done, restricting the draw space to the strip assigned to the processor. A clever scanning of the segments is used to reconstruct the triangles efficiently, refer to [CV95] for more details.

We assumed in this section that the texture was duplicated in all the local memories of the processors. If it is not possible because of memory constraints, we use a similar mechanism as for the DEM distribution, as described in the following section.

4.3 Distribution of the texture

To map the texture onto the ground, each processor must have at least the piece of texture corresponding to the piece of DEM it owns. The informations collected while constructing the DEM's messages are used to build the texture's ones. Thus, the messages have the same structure as before except that in place of the image coordinates of the points, there are the pixels' colors of the texture. Since the texture image can be at a higher resolution than the DEM, the number of segments may be larger.

Once the texture is shared out, we have to retrieve a given pixel from its absolute coordinates in the texture image, since these are used for the mapping. The pixels are placed in a linear array. We use a reference table which contains the informations about each segment (position in the texture and number of pixels) and their places in the array. So, to retrieve a pixel, we find the segment containing it in the reference table, and then, we can get it in the array.

Using such a reference table creates a small overhead. Nevertheless, the search time of the right segment can be accelerated by bucket sorting the segments in the reference table by y coordinate, and beginning the search with the last segment used. Indeed, when drawing a triangle, the pixels of texture that are used consecutively are often very closed at each other.

5 Results

The results given here were obtained on a *Volvox* machine. The DEM used has a size of 700×680 points and the texture is a 8 bits image of the same size (in pixels). These relatively small input data are imposed by the small amount of memory available on each node. Since we want to compare the parallel execution time to the time on one node, the whole dataset has to fit into the memory of a single node. The graphs, in the figure 5, show the times to compute one image as a function of the number of processors, with two different sizes of the output image.

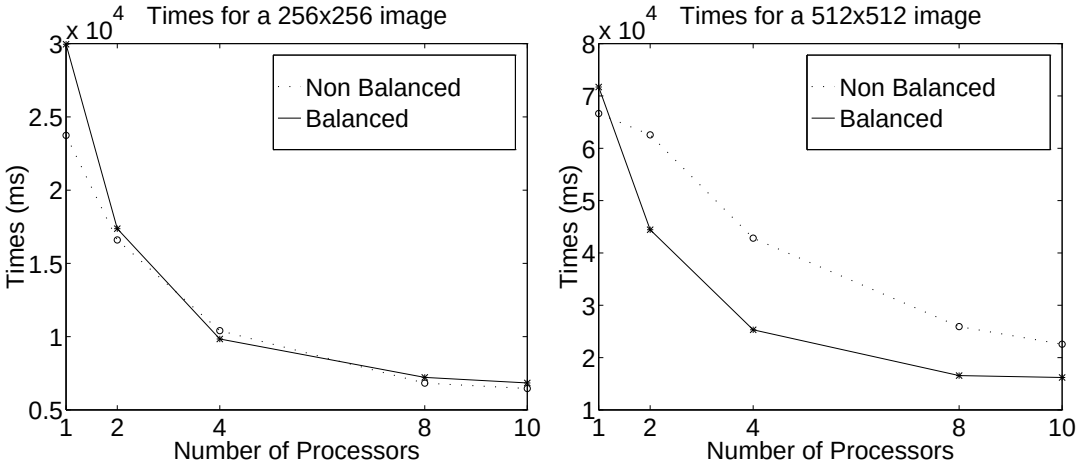


Figure 5: Times to compute one image in function of the number of processors for two sizes of output images.

Two series of tests have been made for each size of output image, either using the load balancing algorithm or not (the static version consists in dividing the image into p equal sized strips). In order to emphasize the time to compute the loads, a distinction is even made with only one processor between the balanced and the static versions.
In the left graph, we observe that the static version is sometimes better than the balanced one. Nevertheless, in the other graph, the balanced version is always better. This difference between the two graphs is due to the influence of the size of the output image on the performances.
We have separated on figure 6, the times of the different kinds of computations (communications, visualization, load balancing when it is relevant,...) over each processor and for the two versions of the algorithm; using a configuration of eight processors and a 512×512 pixels output image. The balancing of the work clearly appears in the visualization times (it is the part of the algorithm we balance). We also observe that the times to compute the loads are the same over the processors and the communication times are slightly different (we can't directly supervise them since they rely on the data redistribution and the internal configuration of the parallel machine).

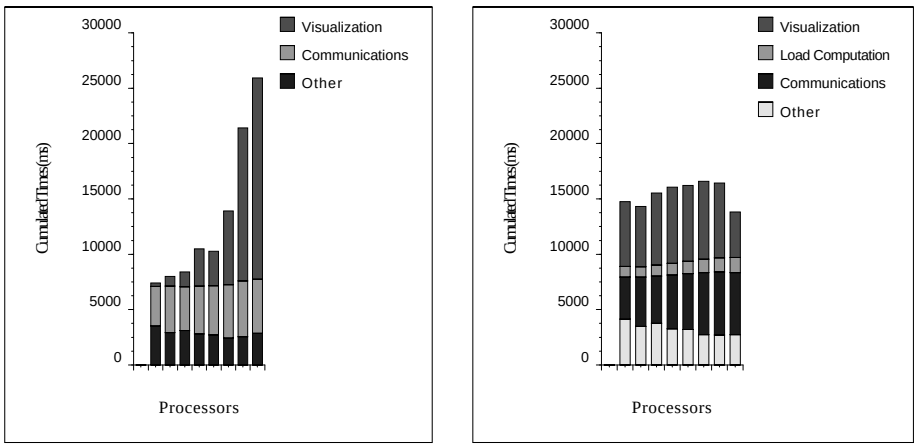


Figure 6: Cumulated times for a 512×512 image on 8 processors. Static allocation (left) – Dynamic Load Balancing (right)

We observed in figure 5, that the execution time decreases when the number of processors increases (which was the purpose of the parallelization). Figure 7 illustrates the speedup of the algorithm as a function of the number of processors, both for the balanced and unbalanced version. The speedup seems to reach quickly an upper bound, but this is due to the small dataset (ground and texture) we used for our experiments, as well as the small image resolution used for rendering.

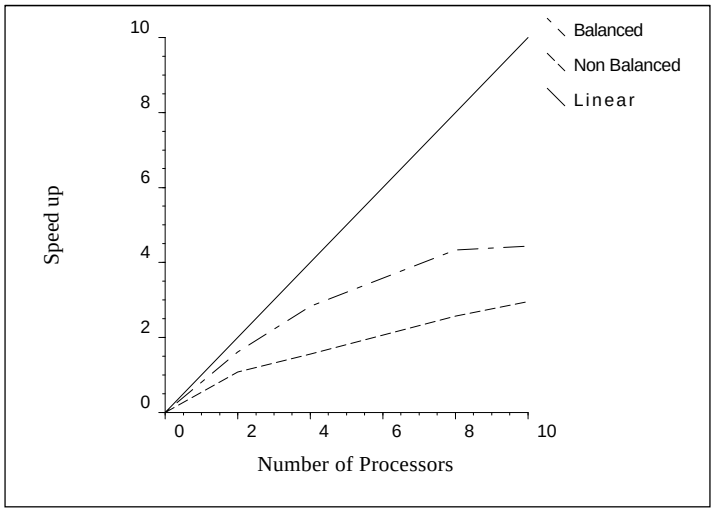


Figure 7: Speedup with the 700×680 DEM in a 512×512 image

To illustrate this dependence, we have plotted in figure 8, the efficiency of the parallelization as a function of the size of the output image. We clearly see that increasing the resolution of the image increases the efficiency and therefore the speedup of the algorithm.

This dependence can be explained by two main reasons: the first one is that the

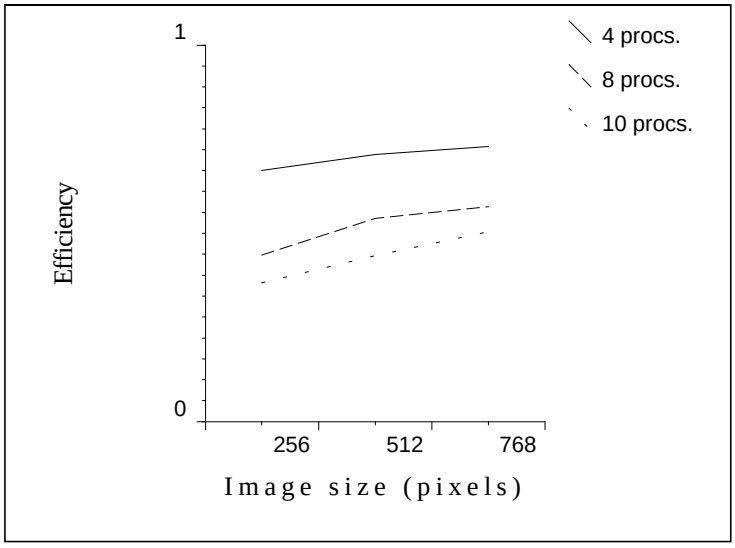


Figure 8: Efficiency as a function of the size of the output image

smaller is the output image, the smaller is the visualization time. Therefore, the overheads of the parallelization take the most important part of the computations. This is why, in the first graph of figure 5, the non balanced version is sometimes faster, since there is no time spent to load balance.
The second reason is the precision of the strips division. If the image is small, the projected DEM is observed smaller and more polygons are intersecting each lines. The strips division is therefore less accurate (it is more difficult to have the same amount of work on each strip). In the other case, the DEM is better spread on the image and the lines have less important loads; hence, it is easier to obtain an accurate division, giving a better repartition of the work.

At last, another parameter which is important for the performances is the position of the observer, and more particularly his distance to the ground. Giving a DEM and the size of the output image; the closer we are to the ground, the smaller is the part we see. Thus, there are much less data to process for an equal time of visualization since the viewed triangles are bigger in the image. Consequently, there are less communications and the overheads are reduced. Figure 9 illustrates this fact by a plot of the speedup as function of the distance between the observer and the DEM; each line corresponds to a given configuration (number of processors).
This ends the analysis of our parallel program's results. These results are promising and they give a good base to study various optimizations. The following part briefly gives several interesting issues for further developments.

6 Possible optimizations

Although the current parallel version of our program gives good results, it is yet possible to improve it at several levels. We give here some ideas which seem interesting to study more precisely.

Multi-resolution : The idea consists in taking into account only those points of the DEM which are significant enough, for instance because they are at a minimum distance of each other in the output image. Therefore, the visualization

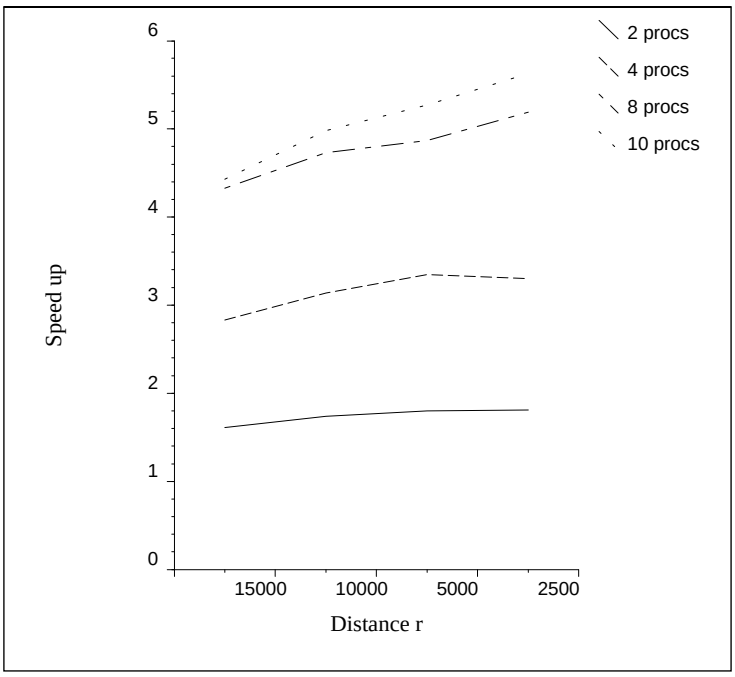


Figure 9: Speedup evolution according to the DEM-observer distance

quality would be almost the same but the amount of data would be smaller, representing an important gain in time.

Coherence : When computing an animation, the use of the coherence between successive images allows not to redo some computations whose results are nearly the same. We can therefore, apply this to the redistribution of the points and the computation of the loads.

Rectangular division : One of the major problem with the strips division is the limitation of the number of processors used, especially with small images. Moreover, for a given number of processors, the rectangular division gives better results than strips one, because we can have a more precise repartition of the works. Thus, We can easily think of doing such a division based on a computation of the loads for the lines and the columns of the image.

7 Conclusion

The parallel version of Volter was presented. It is based on an irregular strip-wise subdivision of the output image, leading to a balanced execution; the DEM and the texture are shared out in order to have just the necessary data on each processors. The implementation was developed under the *PPCM* environment created at the LIP. This environment allows the use of several parallel machines like an iPSC, a Volvox, a Paragon, a network of stations (PVM), and a Cray T3D.
This study permitted to point out the important parameters influencing the performances of this kind of programs. The most important are the size of the output image, the size of the data to visualize, and the amount of data effectively in the image (that directly depends on the observer-ground distance).
The domain of the visualization often requires large computation times. In order to reach interactivity or to create animations, these times must be as small as pos-

sible. The technique presented here allows to process very large amount of data, at reasonable speed. We showed how to take into account the specific data structure of the grounds represented by DEMs. Moreover, we proposed some optimizations that should make it possible to still improve our absolute performances.

References

[All91] M. Allison. *Private Communication*, July 1991.

[BBP90] D. Badouel, K. Bouatouch, and T. Priol. Ray tracing on distributed memory parallel computers: Strategies for distributing computations and data. *Siggraph*, pages 185–198, 1990.

[CDH+88] F. C. Crow, G. Demos, J. Hardy, J. McLaughlin, and K. Sims. 3D image synthesis on the connection machine. *Proceedings of the International Conference on Parallel Processing for Computer Vision and Display*, January 1988.

[Cha91] J. Challinger. Parallel volume rendering on a shared memory multiprocessor. *UCSC-CRL-91-23, University of California, Santa Cruz*, 1991.

[CJ81] P. Chang and R. Jain. A multi-processor system for hidden surface removal. *Computer Graphics*, 15(4):405–436, December 1981.

[CLM95] Henri-Pierre Charles, Laurent Lefevre, and Serge Miguet. An optimized and load-balanced portable parallel ZBuffer. In *SPIE Symposium on Electronic Imaging: Science and Technology*, 1995. to appear.

[CO91] T. W. Crockett and T. Orloff. A parallel rendering algorithm for MIMD architectures. *ICASE Tech. Rept. No. 91-3, NASA Langley Research Center*, June 1991.

[Cro87] G. A. Crocker. Screen-area coherence for interactive scanline display algorithms. *IEEE Computer Graphics and Applications*, September 1987.

[CS89] E. Caspary and I. D. Scherson. A self-balanced parallel ray-tracing algorithm. *Parallel Processing for Computer Vision and Display, P. M. Dew, T. R. Heywood, R. A. Earnshaw (Addison-Wesley)*, pages 408–419, 1989.

[CV93] S. Contassot-Vivier. Reconstruction 3D de terrains texturés. Rapport de stage de licence, August 1993.

[CV95] Sylvain Contassot-Vivier. Visualisation parallèle de MNT texturés. Master's thesis, Ecole Normale Supérieure de Lyon, June 1995.

[CWBV83] J. G. Cleary, B. M. Wyvill, G. M. Birtwistle, and R. Vatti. Multiprocessor ray tracing. *Tech. Rept. 83/128/17, University of Calgary*, 1983.

[Ell94] D. A. Ellsworth. A new algorithm for interactive graphics on multicomputers. *IEEE COMPUTER*, July 1994.

[FFR83] E. Fiume, A. Fournier, and L. Rudolph. A parallel scan conversion algorithm with anti-aliasing for a general-purpose ultracomputer. *ACM Computer Graphics*, 17(3):141–149, July 1983.

[FK90] W. R. Franklin and M. S. Kankanhalli. Parallel object-space hidden surface removal. *ACM Computer Graphics (Siggraph)*, 24(4):87–94, August 1990.

[GP86] D. Ghosal and L. M. Patnaik. Parallel polygon scan conversion algorithms: Performance evaluation of a shared bus architecture. *Computers and Graphics*, 10(1):7–25, 1986.

[GVL95] D. Germain, G. Vézina, and D. Laurendeau. Visibility analysis on a massively data-parallel computer. In Vincent Van Dongen, editor, *High Performance Computing'95*, pages 267–278, Montreal, Quebec, July 1995. CRIM.

[HF85] M. C. Hu and J. D. Foley. Parallel processing approaches to hidden-surface removal in image space. *Computer and Graphics*, 9(3):303–317, 1985.

[Hor84] C. Hornung. A method for solving the visibility problem. *IEEE Computer Graphics and Applications*, July 1984.

[KG79] M. Kaplan and D. P. Greenberg. Parallel processing techniques for hidden surface removal. *Computer Graphics, Proceedings for Siggraph*, 13(2):300–307, July 1979.

[MCEF94] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications*, July 1994.

[MR91] Serge Miguet and Yves Robert. Elastic load balancing for image processing algorithms. In H. P. Zima, editor, *Parallel Computation*, Lecture Notes in Computer Science, pages 438–451, Salzburg, Austria, September 1991. 1st International ACPC Conference, Springer Verlag.

[OHA93] F. A. Ortega, C. D. Hansen, and J. P. Ahrens. Fast data parallel polygon rendering. *ACM*, 1993.

[Rob88] D. R. Roble. A load balanced parallel scanline Z-Buffer algorithm for the iPSC hypercube. *Proceedings of Pixim'88, Paris, France*, October 1988.

[Rog85] D. Rogers. *Procedural Elements for Computer Graphics*. McGraw-Hill, 1985.

[War69] J. E. Warnock. A hidden-surface algorithm for computer generated halftone pictures. *Tech. Rept. TR 4-15, University of Utah, NTIS AD-753 671*, June 1969.

[Wat70] G. S. Watkins. A real-time visible surface algorithm. *Tech. Rept. UTEC-CSc-70-101, University of Utah, NTIS AD-762 004*, June 1970.

[WHC94] S. Whitman, C. D. Hansen, and T. W. Crockett. Recent developments in parallel rendering. *IEEE Computer Graphics and Applications*, July 1994.

[Whe85] D. S. Whelan. *Animac: A Multiprocessor Architecture for Real-Time Computer Animation*. PhD thesis, California Institute of Technology, 1985.

[Whi92a] S. Whitman. A brief summary of parallel graphics rendering algorithms. *MPCI Report*, 1992.

[Whi92b] S. Whitman. *Multiprocessor Methods for Computer Graphics Rendering*. Jones and Bartlett, 1992.

[Whi94] S. Whitman. Dynamic load balancing for parallel polygon rendering. *IEEE Computer Graphics and Applications*, July 1994.