MPIPOV: a Parallel Implementation of POV-Ray Based on MPI*

Alessandro Fava[1], Emanuele Fava[1], and Massimo Bertozzi[2]

[1] Dipartimento di Ingegneria Industriale, Università di Parma
Parco Area delle Scienze 181/A, I-43100 Parma, Italy
fava@ce.unipr.it
[2] Dipartimento di Ingegneria dell'Informazione, Università di Parma
Parco Area delle Scienze 181/A, I-43100 Parma, Italy
bertozzi@ce.unipr.it

Abstract. The work presents an MPI parallel implementation of Pov-Ray, a powerful public domain ray tracing engine. The major problem in ray tracing is the large amount of CPU time needed for the elaboration of the image. With this parallel version it is possible to reduce the computation time or to render, with the same elaboration time, more complex or detailed images. The program was tested successfully on ParMa2, a low-cost cluster of personal computers running Linux operating system. The results are compared with those obtained with a commercial multiprocessor machine, a Silicon Graphics Onyx2 parallel processing system based on an Origin CC-NUMA architecture.

1 Introduction

The purpose of this work is the implementation of a distributed version of the original code of Pov-Ray [1], that is a well known public domain program for ray tracing. The parallelization of this algorithm involves many problems that are typical of the parallel computation. The ray tracing process is very complex and requires a notable quantity of computations that take, for the most complex images, from many hours up to days of processing. From that the need to increase the elaboration speed of these operations trying to compute the image in parallel.

We have used MPICH, a freely available, portable implementation of the MPI standard message-passing libraries [7], that allow to develop parallel programs on distributed systems, using standard programming language (C and Fortran) [3].

2 Ray tracing with POV-Ray

POV-Ray (Persistence Of Vision Raytracer - www.povray.org) is a three-dimensional rendering engine. The program derives information from an external text file, simulating

* The work described in this paper has been carried out under the financial support of the Italian *Ministero dell'Università e della Ricerca Scientifica e Tecnologica* (*MURST*) in the framework of the MOSAICO (Design Methodologies and Tools of High Performance Systems for Distributed Applications) Project.

the way the light interacts with the objects in the scene to obtain a three-dimensional realistic image.

The elaboration consists in projecting a ray for each pixel of the image and following it till it strikes an object in the scene that might determine the color intensity of that pixel. Starting from a text file that contains the description of the scene (objects, lights, point of view), the program can render the desired image. The algorithm works line by line with an horizontal scan of each pixel.

An interesting option of POV-Ray is the antialiasing. The antialiasing is a technique that helps to remove sampling errors producing a better image. Generally the antialiasing renders smoother and sharper images. Using the antialiasing option, POV-Ray starts tracing a ray for each pixel; if the color of the pixel differs from that of its neighborhood (the pixel on the left hand and that above), of a quantity greater than a threshold value, then the pixel is supersampled tracing a fixed number of additional rays. This technique is called supersampling and could improve the final image quality but it also increase considerably the rendering time. After the input data parsing, POV-Ray elaborates all the pixels of the image to render, trough an horizontal scan of each line from left to right. Once finished the elaboration of a line, this is written on a file or displayed on screen and then the following line is considered up to the last one (see Fig. 1).

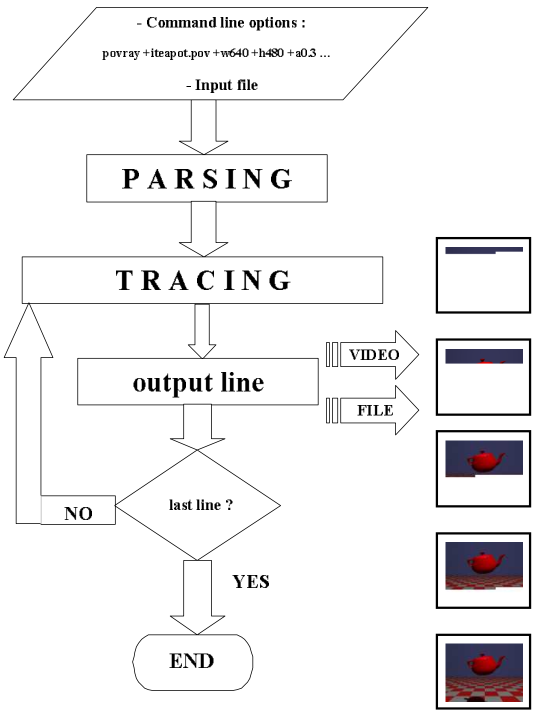

Fig. 1. Scheme of serial POV-Ray.

3 Parallel Implementation

Due to the high computational power required by POV-Ray, a parallel implementation is straightforward and other research groups tried to develop a parallel version of this application. The most famous is PVMPOV based on PVM [2, 6]. There are other porting based on MPI [4, 5] that, altough deeply investigates the partitioning of the problem, seem not to face the complex problem of antialiasing feature of POV-Ray.

Conversely, the main goal of this work was obtaining an high performance parallel version of POV-Ray with all its features available (i.e. antialiasing).

Two different approaches have been investigated, the first (MPIPOV1) features a static division of the work determined before the run of the computation. This approach is suitable for homogeneous distributed architectures. The second approach (MPIPOV2) provides load-balancing assigning dynamically more work to faster nodes, thus fitting heterogeneous architectures.

The results, as concern the speedup have been very effective in both cases, even if the release with dynamic assignment is the worst from the point of view of the absolute performance, because of the larger number of communications needed respect to the static approach. This second release is potentially more efficient in the case of an heterogeneous cluster.

3.1 MPIPOV1: Static Parallelization

The first type of parallelization is based on a static partitioning of the work. Each node is assigned a section of the line to elaborate. The nodes elaborate their section in parallel and, once finished, they send it to a master process that, after processing its section, reconstructs the complete line and then display or save it on a file (see Fig. 2). This procedure is repeated up to the last line of the image.

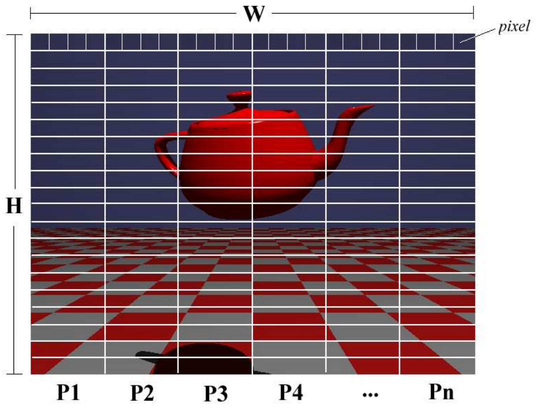

Fig. 2. Image subdivision in MPIPOV1: W = horizontal resolution (pixels), H = vertical resolution (pixels), n = number of processes, and $P1, \ldots, Pn$ = processes.

Each line is divided in n sections of W/n pixels. There is a control for the possible remaining pixels forcing the last section to have the value W as upper limit. The complete image will result composed of n vertical strips of H sections of line. The master process receives $(n-1) \times H$ sections while each slave sends H sections.

This implementation works fine on homogeneous parallel systems where each processing node features the same computation power. In fact a line is completed (and then written on disk or displayed on screen to continue the elaboration of the following) only when the slower process finishes the computation of its section while the faster processes remain idle, wasting useful time of elaboration. Moreover there is a small difference with the result produced by the original version: the antialiasing is, in fact, an intrinsically serial operation and so this function does not take into account the first pixels of each section that compose the complete line. This difference does not affect the image quality in noticeable manner.

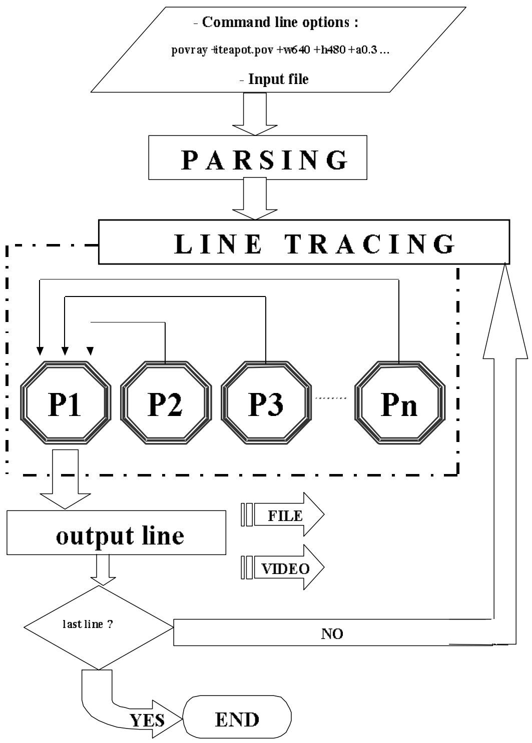

Fig. 3. MPIPOV1 Scheme.

3.2 MPIPOV2: Dynamic Parallelization

In the second approach the load of each process is dynamically balanced: in this way we assign more work to the faster processes: each process will elaborate a number of sections in proportion to its elaboration speed. In this way we try to find a solution to the slower process limitation, optimizing the use of each processor.

Each line of the image is now divided into a number of sections greater than the number of the available processes. Initially each processor is assigned a single section of the line to be computed; the master process, that does not have any section to elaborate, has the task to assign a new section to the first process that end its work. The elaboration goes on till no line sections are left. The procedure is iterated, also in this case, line after line (see Fig. 5).

The elaboration speed of each process could be different due to the internal characteristics of the architecture or to the machine load, but it also depends on the particular section of the image to process that could be more or less complex.

Now it is the user that can choose the number of vertical strips in which the image will be divided. The optimal value results difficult to find because it depends on many factors including the number of the available processes, the horizontal resolution and the machine load. In fact the lower grain of the computation can potentially improve the load balancing effect but also increase the communications.

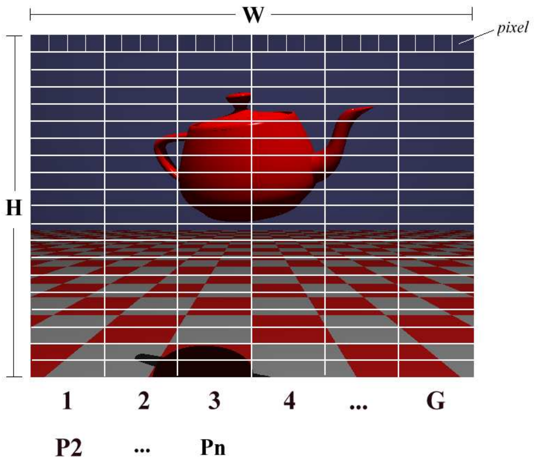

Fig. 4. Image subdivision in MPIPOV2: W = horizontal resolution (pixels) H = vertical resolution (pixels) n = number of processes $P1, \ldots, Pn$ = processes G = number of section for each line.

Figure 4 shows more in detail how the image is now divided. The algorithm requires that the number of line sections G must be a value between $n1$ and W.

Each line is divided in: G sections of W/G pixels. There is a control for the possible remaining pixels forcing the last section to have the value W as upper limit.

The complete image will result composed from: G vertical strips of H sections of line.

The advantage obtained from the dynamic management of the sections to elaborate, that is appreciated only in cases of inhomogeneous load of the machines used, produces an expressive increase of the number of communications among the processes. The consequence is the increase of the time of broadcast and of synchronization due to the sending of the orders to the slaves and to the larger number of sections to manage. The number of communications increases further if the antialiasing option is used. In this case, each slave have to know the complete line previously elaborated and some information concerning the antialiasing of the same line; all this is sent from the master that reconstructs the various partial data received from the slaves during the computation of the individual lines. Also with this approach, remains the problem concerning the antialiasing of the first pixel of each line section.

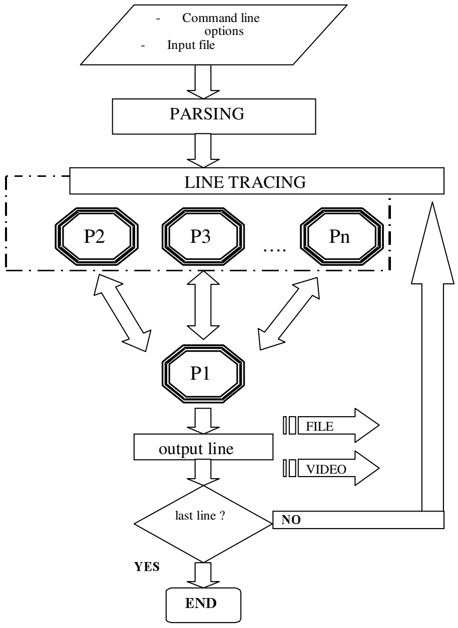

Fig. 5. MPIPOV2 Scheme.

4 Results

The examples has been tested on the distributed architecture ParMa^2, developed by Computer Engineering Department of the University of Parma [9]. ParMa^{2}is a cluster of PC running Linux as operating system, composed of four Dual Pentium II 450 MHz interconnected by a 100 Mbit/s switched Fast Ethernet network, forming an 8-processor system.

The program is also been ported and tested on a commercial multiprocessor machine an SGI Onix2 (with 8 R10000 195 MHz processors).

The results obtained rendering the same sample image with different numbers of processors are presented in Tables 1 and 2.

# Processors	Time (sec.)		Speedup	
	ParMa^2	Onix2	ParMa^2	Onix2
1	60	76	1	1
2	30	40	2	1.9
3	24	31	2.5	2.45
4	19	23	3.15	3.3
5	15	20	4	3.8
6	12	16	5	4.75
7	10	14	6	5.42
8	9	12	6.66	6.33

Table 1. Speedup of MPIPOV1 on the Pc-cluster ParMa^{2}and Onix2.

For MPIPOV1 we have for both architectures fairly linear speedup and for what concerns absolute performances the best time is obtained with the cluster ParMa^2.

# Processors	Time (sec.)		Speedup	
	ParMa^2	Onix2	ParMa^2	Onix2
1	60	76	1	1
2	67	80	0.89	0.95
3	37	41	1.62	1.85
4	28	29	2.14	2.62
5	24	22	2.5	3.45
6	23	18	2.6	4.22
7	21	16	2.85	4.75
8	20	15	3	5.06

Table 2. Speedup of MPIPOV2 on the Pc-cluster ParMa^{2}and Onix2.

For MPIPOV2, that we can define communication-intensive, speedup and absolute performances are better for the Onix2 that can take advantage of the fast interconnection structure.

5 Conclusions

We have developed an efficient parallel version of POV-Ray, a well known and freely available ray tracing engine, which is particularly suited for the parallelization. The ray tracing process is very complex and requires a notable quantity of computations and of time to obtain realistic three-dimensional images and so the need to increase the elaboration speed.

For the development of the parallel version, it has been used the MPI message-passing library, which can exploit the potential of low-cost architectures such as clusters of personal computers. The MPI parallel code is also easily portable on different architecture and operating system.

Two method has been described for parallel ray tracing. The first is a static algorithm that distributes equal work to each node of the parallel system, whereas in the second, the dynamic one, each node elaborates an amount of work in proportion to its elaboration speed.

The parallel release has been tested on a Linux based cluster (ParMa^2) producing excellent results. Test included compares the use of static vs dynamic version and ParMa^2 cluster vs SGI Onix2 commercial multiprocessor machine. Final benchmark testing, using a well known sample image, yielded the best results using ParMa^2 system, showing the potentiality of the cluster architecture and therefore their efficiency in terms of cost/performance ratio with respect to traditional multiprocessor architectures.

A highly optimezed dynamic version of POV-ray 3.1 that can also completely perform the antialiasing computation, is now under development [8].

References

1. O. Aftreth, G. Emery, and W. Morgan. The Online POV-Ray Tutorial, 1996. Available at `http://library.advanced.org/3285/`.
2. A. Dilger. PVM patch for POV-Ray, 1998. Available at `http://www-mddsp.enel-.ucalgary.ca/People/adilger/povray/pvmpov.html`.
3. E. Fava. Realizzazione in Ambiente Distribuito di un Programma per Raytracing. Master's thesis, Università degli Studi di Parma - Facoltà di Ingegneria, 1999.
4. B. Freisleben, D. Hartmann, and T. Kielmann. Parallel Incremental Raytracing of Animations on a Network of Workstations. In *Procs. Intl. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, volume 3, pages 1305–1312, July 1998.
5. B. Freisleben, D. Hartmann, and T. Kielmann. Parallel Raytracing: A Case Study on Partitioning and Scheduling on Workstation. In *Procs. 30th Hawaii Intl. Conf. on System Sciences*, volume 1, pages 596–605, Jan. 1998.
6. A. Haveland-Robinson. POVbench home page, Apr. 1999. Available at `http://www.haveland.com/povbench/`.
7. Mathematics and Computer Science Division, University of Chicago, Chicago. *User's guide for MPICH, a Portable Implementation of MPI*, 1995.
8. R. Sbravati. Parallelizzazione di POV-ray 3.1 utilizzando la libreria MPI. Technical report, Dipartimento di Ingegneria dell'Informazione, Università di Parma, June 1999.
9. D. Vignali. ParMa^2 White Paper. Technical report, Dipartimento di Ingegneria dell'Informazione - University of Parma, Sept. 1998. Available at ftp://ftp.ce.unipr.it/pub/ParMa2.