Parallel raytracing on the IBM SP2 and CRAY T3D

Igor-Sunday Pandzic, MIRALab, University of Geneva
Michel Roethlisberger, IBM Geneva
Nadia Magnenat - Thalmann, MIRALab, University of Geneva

Abstract

Raytracing is a widely used method for generating realistic-looking images on a computer, but it still requires considerable computing power, especially when rendering complex scenes. This paper presents a parallel raytracing algorithm with dynamic workload distribution, based on public domain raytracing software Rayshade. The implementation on the IBM SP2 and CRAY T3D is presented, with the discussion of the differences between the two machines with respect to the ease of implementation and the performance.

Motivation

Raytracing is a widely used method for generating realistic-looking images on a computer. Currently, most of the 3D modelling and animation systems use raytracing for the final image rendering. However, the algorithm is quite computing-intensive and the demand on processing power grows as the scenes to visualise become more and more complex.

MIRALab is a Computer Graphics research group and one of its activities is the production of computer generated films. As our modelling and animation algorithms enable us to produce ever more complex scenes, the rendering becomes a bottleneck for the production. The rendering of a really complex film, as the one mentioned in /1/ would take several weeks. In order to be able to produce more complex scenes and increase our production speed we have decided to implement a parallel raytracer on the IBM SP2 and the CRAY T3D. The software we used as a base is the public domain raytracing package Rayshade.

Principles of raytracing

The input to the raytracing algorithm is the scene - the description of 3D objects geometry, together with the definition of objects' materials, definition of lights and the definition of the camera. The output is the image of the scene as seen by the defined camera. More precisely, the output is the colour of each pixel of the image (the frame).

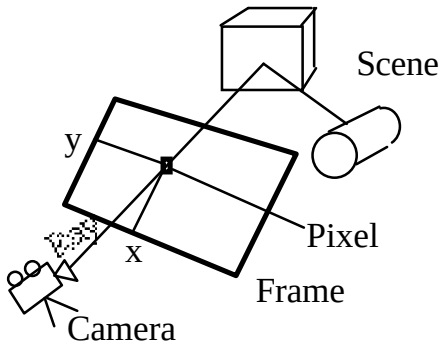


Figure 1: Principles of raytracing

In order to compute the colour of the pixel *x,y* a ray is casted from the camera through the frame at the position *x,y* into the scene and the intersection with the first object is found (if any). Based on the position of the intersection point, the surface normal at this point, the position and colour of lights and the material of the intersected surface, the light intensity and colour at the intersection point is computed. Then, the ray is reflected and/or refracted based on the reflectivity and transparency of the surface and the process is repeated recursively with the reflected/refracted rays, adding the light intensities at all intersection points to get the final colour of the traversed pixel in the frame. The recursive process stops when the light contribution gets below a threshold, or at user-defined depth of recursion.

Parallelization strategy

As explained in the previous chapter, in the basic raytracing algorithm one ray is casted for each pixel of the frame and all the rays are independent from each other. This makes the basic parallelization strategy quite obvious - distribute a certain number of rays to each processor, i.e. each processor works on one part of the frame. Each processor must have the complete scene description. This might seem trivial at the first glance, but careful analysis shows some serious problems:

- Each ray will take a different amount of processor time to compute, depending on the complexity of the intersections that occur. If a big region of the frame is distributed to each processor, this will lead to load balance problems as illustrated in figure 2. Obviously the processors 0, 1 and 2 process the rays that don't intersect with the scene, and P3 will do all the work. As it is impossible to predict the complexity of the intersections for a given part of the frame, the only solution is to allocate smaller, scattered portions of the frame to each processor.
- Rayshade software uses an advanced raytracing algorithm using the information from neighbouring rays' intersections to improve speed. This brings a big improvement in speed, but the rays are not independent anymore. In order to profit from the advanced algorithm, contiguous blocks of frame pixels (i.e. rays) should be processed by each processor. The larger the blocks, the bigger the benefit from the advanced algorithm.

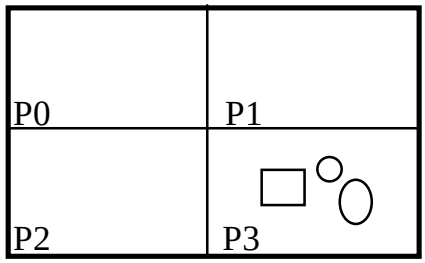


Figure 2: Load balance problems when distributing big frame regions

These two requirements are clearly contradictory - allocating small scattered regions to improve load balance vs. allocating big contiguous regions to profit from the advanced algorithm using neighbouring information. In order to find an optimum between the two opposing requirements we have devised a dynamic workload allocation mechanism that will distribute the workload optimally for every scene. The basic strategy is to begin by allocating relatively large portions, but as the job comes to the end to allocate smaller and smaller portions, thus adjusting quite precisely the load balance.

The algorithm

We will first present the algorithm regardless of the actual implementation on a particular machine. While discussing the algorithm we will accept the notion of global memory, accessible by all the processors, that will hold the final image and the global variables that need to be unique and shared by all the processors. The actual implementation of this global memory will be discussed later.

Because of the structure of the existing Rayshade software we decided to take a line of the frame as the basic distributable unit. So, when a processor gets a job, it is determined by the starting line (counting from the bottom of the frame) and the job size expressed in number of lines.

The global workload counter is crucial for the dynamic workload allocation algorithm. It contains the number of the next line to be processed (initially 0) and the global workload left, i.e. the number of lines left to be processed (initially equal to the number of lines of the frame). The global counter is continuously updated by all the processors as they do the work.

The flowchart (figure X) represents the algorithm executed on each processor. First the database is loaded and the variables initialised. In each step of the outer loop the processor will access the global counter to determine the job to do. The size of each job is proportional to the workload left

to be done and inverse-proportional to the number of processors. In this way we insure that the jobs get smaller as the work proceeds, with the effect of fine-tuning the load balance towards the end of the processing When the size of the job is determined the pointer to the next line to be processed is updated in the global counter, i.e. increased by the size of the job. As the job is being done, at each line the global workload in the global counter is decremented by one.

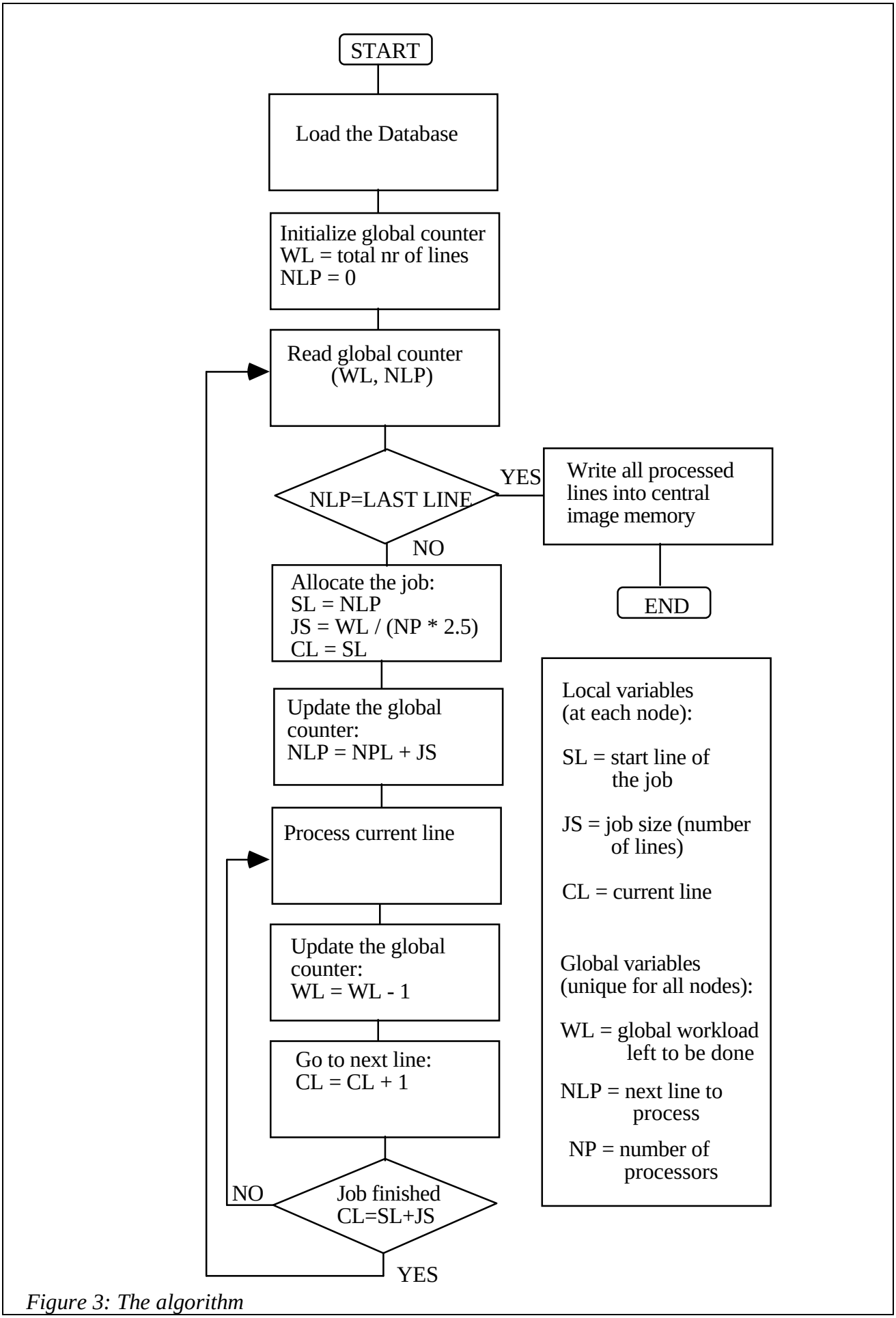


Figure 3: The algorithm

The implementation

The algorithm has been implemented on two parallel computers: a 256-node CRAY T3D and a 14-node IBM SP2. C programming language was used on both architectures, but with different parallel programming subroutine sets. The Shared Memory model was used on the T3D, while on the SP2 we used MPI.

While most of the algorithm is quite straightforward to implement, the real challenging issue was to implement the "central memory" for the global counter. The demand is that all processors have read/write access to the global counter. At the same time conflicts have to be avoided (simultaneous writing by two processors).

Both machines have a distributed memory architecture, so no central memory exists as-is. The solution is to place the global counter on one of the nodes and let the other nodes access it. The solutions for this access vary with the programming model used.

The Shared Memory model allows asymmetric communication, i.e. node A can read/write the data in the memory of node B, without node B needing to send/receive the data explicitly. This already solves the problem of the access to the global counter - it is put on one processor, shared memory links are established and all nodes can access the global counter for read and write operations. To avoid conflicts, each node has to reserve the write access before writing to the global counter. This is implemented using a semaphore in the "global memory". Before writing, the node checks the state of the semaphore. If it is free, it puts it's ID number into the semaphore, reserving the write access to the global counter. However, since the reading and writing of the semaphore is not instantaneous, there is still a small risk of a conflict: if two nodes access the semaphore at the same time, both will see it as free and both will send their ID to the semaphore. Obviously one ID will be overwritten by the other. To eliminate this risk, the node will wait a short time after writing the ID to the semaphore, then read the semaphore again to check if it's ID is there. Only then it is completely safe to write into the global counter.

In the MPI programming model there is no possibility to access another node's memory directly. If node A wants data from node B, B has to send it either explicitly or through a global communication scheme. In any case, for each communication a more or less explicit communication pair is needed. There is however the possibility to have non-blocking send and receive commands. Practically, if node A knows that node B will send a message, it can issue a non-blocking receive command, continue it's work and from time to time check if the message has been received.

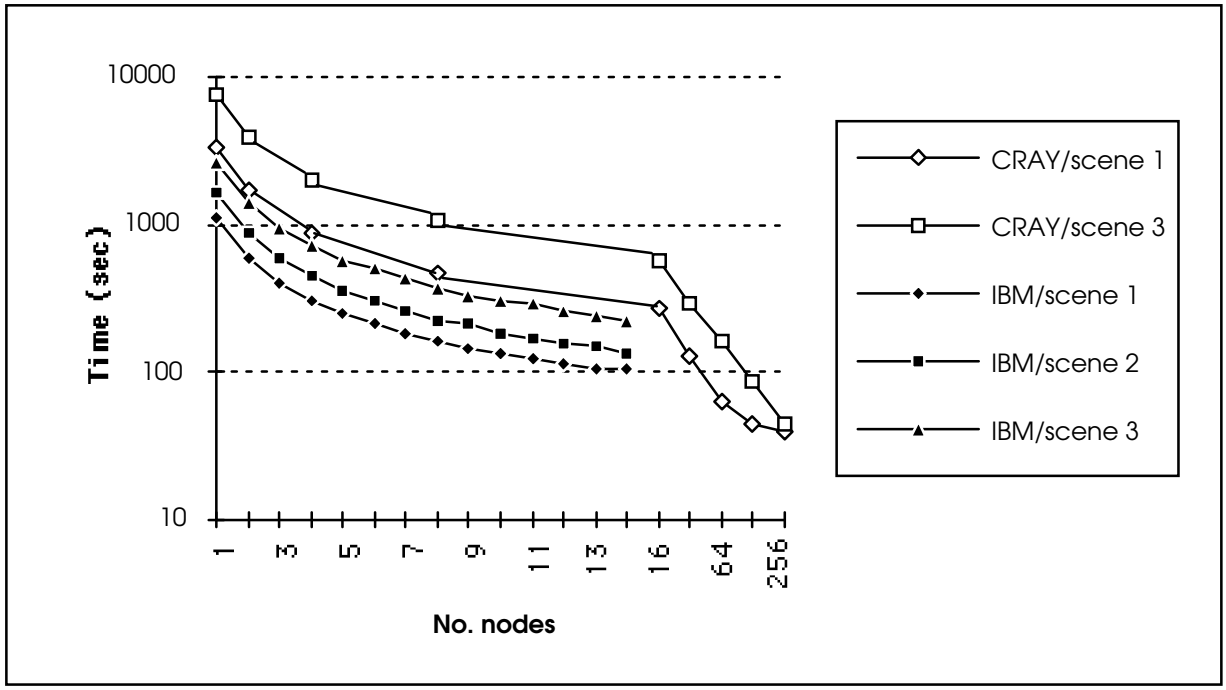
As it is impossible to implement a passive "central memory", we have used the notion of the master processor, whose task is to keep the global counter and distribute the jobs to other processors. However, this task is relatively small compared to the real processing, so it would be very inefficient to waste one processor just to act as the master. For this reason we have implemented an interrupt mechanism where the master processor actually does a slave's work most of the time and gets interrupted each 20 ms to perform the function of the master.

The master will issue non-blocking receive commands allowing all slaves to send him messages asking for a job. When the master receives such message, it will allocate the job according to the state of the global counter and send the job command to the slave. This is a slight change of the original algorithm where the slave performs the job allocation itself. As the slave processes each line it will send messages (non-blocking) to the master to keep him up to date about the state of the work so that the master can update the global counter.

Performance results

Computation have been undertaken for the Cray T3D and for the IBM SP2 on scenes of different complexity. 1 scene of 18000 polygons has been computed once at normal image size 720 x 576 (scene 1) and once at a larger image size 1000 x 1000 (scene 3). An additional scene of

higher complexity, 56000 polygons (scene 2) has also been computed on the SP2. The performance results obtained are presented in Figure 3 where we represent measured elapsed time and measured efficiency on both parallel systems. When running on n nodes, the efficiency is defined as Eff = T1 / (Tn * n), where T1 is the time when run sequentially on 1 node and Tn the time on n nodes.



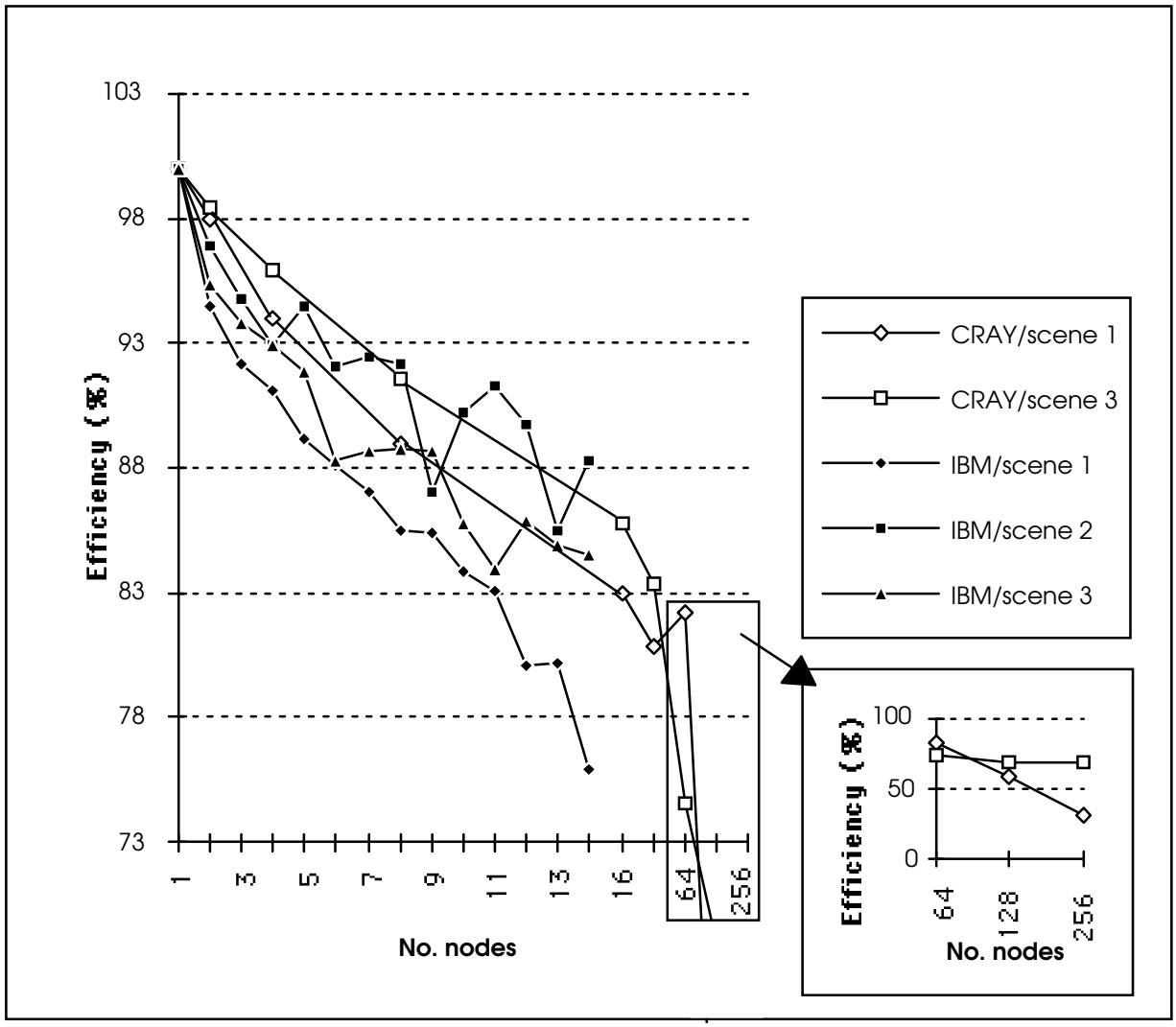


Figure 3: Time and efficiency measurements

Following comments can be drawn:

1) SP2 processor is about 3 time faster than the T3D processor on this application.

2) T3D exhibits about 4% better efficiency than the SP2 for scenes of modest complexity. This gap tends to reduce for higher scene complexity. Unfortunately, it has not been possible to run case 3 with 56000 polygons on the T3D, as not enough memory was currently available. 2 factors may contribute to efficiency: 1) load balancing and 2) the factor $f = \alpha * \beta$, with α = (speed of communication) / (speed of computation) and β = (amount of computation) / (amount of communication). As we just saw, the SP2 processor is faster than the T3D and the T3D has a faster communication system than the SP2, especially when we use the Shared memory model. Moreover, in the MPI implementation, we introduced an "application latency" of the order of 10ms on the master interrupt mechanism. For all these reasons, α is higher on the T3D than on the SP2, which explains the differences we measured in efficiencies. On the other hand, computing scenes with higher complexity increases β and therefore have a positive effect on efficiency. We believe that load balancing performs equally good in both systems with the slight exception master node also has to do some job management tasks in the MPI implementation. Scalability As the application still relies on Host-node model, it was not expected to get full scalability both in term of computing time and data distribution. However, the T3D demonstrates good efficiencies even for high number of nodes. As our SP2 was a 14 parallel nodes system, it has not been possible to run similar tests.

Figure 4 shows a rendered image of the complex test scene.

Conclusion

- Both systems have proven to be very efficient on parallel Raytracer. Film production time can now be reduced by one order of magnitude by using a reasonable number of nodes, typically 16. This provide a definite advantage of using parallel raytracer in a production environment.

- Our load balancing mechanism proved to be very efficient on both machines.

- The implementation was slightly easier on the T3D with the proprietary shared memory model than in the SP2 with MPI.

- The T3D has demonstrated a faster communication system even for a high number of nodes. However as our application did not require a lot of communications, global performance is higher on the SP2 which can take full advantage of his high performance node. This conclusion might change for a very fine grained parallel application.

- It was possible to compute very complex scenes on the SP2 as more memory was available per node.

Acknowledgement

We would like to thank B. Mathews / University of Geneva for kindly making available the whole SP2 for our tests, as well as Jean-Claude Moussaly, MIRALab, University of Geneva for creating the final rendered image.

References

1, Nadia Magnenat-Thalmann, Igor Sunday Pandzic, Jean-Claude Moussaly, Daniel Thalmann, Zhyong Huang, Jianhua Shen, "The Making of Xian Terra-cotta Soldiers", Computer Graphics: Developments in Virtual Environments, pp. 281-295, Academic Press, 1995.

2. Craig E. Colb "Rayshade User's Guide and Reference Manual", 1991.

3. IBM Corporation, "IBM AIX Parallel Environment Parallel Programming Subroutine Reference", 1994.
4. Cray Research, Inc. "Cray T3D Applications Programming", 1994.

Figure 5: An example of a rendered image