Ray Tracing Complex Scenes: Sequential or In Parallel?

Amo Formella*
Computer Science Department, University of the Saarland, 66041 Saarbrücken, Germany
formella@cs.uni-sb.de

Keywords: cost–effective speedup, shared memory architecture, image processing, benchmarking

Abstract

We present a discussion whether current parallel machines or, preferable, fast sequential computers should be used to render images using ray tracing. Based on the definitions of cost–effective speedup and efficiency, we will show that shared memory machines have advantages over distributed memory machines. Moreover, the SB–Pram appears to be an architecture which allows for cost–effective absolute speedup on large data bases.

The communication overhead is too large to be handled by the message passing systems. Our implementations on a KSR–1 [2] show that with the shared memory approach at least a cost–effective relative speedup can be obtained on that machine. However, the SB–Pram [3] provides a cost–effective absolute speedup, which is higher than any published data of other machines.

Cost–effective speedup and efficiency are defined in Section 2. Section 3 introduces ray tracing, optimization methods and approaches how the problem can be implemented in parallel. Section 4 characterizes the machines and the benchmark suites. Section 5 summarizes some benchmarking results. In Section 6 the conclusions are drawn.

1 Introduction

According to the parallel computing community and the manufacturers of parallel machines, parallel systems are suited to solve large and complex problems. In [1] it is said: The main purpose of parallel processing is to perform computations faster than can be done with a single processor by using a number of processors concurrently. ... The need for “faster solutions” and for “solving larger–size problems” arises in a wide variety of applications. Ray tracing is a large and complex task which offers the possibility to generate high quality images. Thus, it is an excellent application to examine the facilities of parallel systems and to compare the performance of parallel systems with sequential ones.

To render fotorealistic images large data bases providing enough details of the scenes are needed. As long as single images are generated, the data base is accessed read-only. However, if sequences of frames are to be calculated, a fast algorithm might change the data base as well. Fast algorithms exploiting coherence properties of the scene usually perform highly irregular accesses to the data base. Due to the nature of the problem tracing a huge number of rays, there is enough inherent parallelism to keep a large number of processors busy.

We point out that current parallel implementations of the ray tracing kernel on distributed memory machines (DMMs) cannot be considered being efficient.

2 Speedup and Efficiency

From a theoretical point of view [1], speedup $s$ and efficiency $\eta$ are defined for parallel algorithms on a machine with $p$ processors as: $s = T_s/T_p$ and $\eta = s/p$, where $T_s$ is the run time of the sequential algorithm and $T_p$ is the run time of the parallel algorithm. These definitions do neither consider the relation between the algorithms, nor their space or memory requirements. Let $T^*$ be the run time of the best known sequential algorithm to solve the problem. The absolute speedup $s_{abs}$ and the relative speedup $s_{rel}$ are defined as: $s_{abs} = T^*/T_p$ and $s_{rel} = T_1/T_p$; thus, the absolute efficiency $\eta_{abs}$ and the relative efficiency $\eta_{rel}$ as: $\eta_{abs} = s_{abs}/p$ and $\eta_{rel} = s_{rel}/p$.

Because of $T^* \leq T_1$, $s_{abs} \leq s_{rel}$; and for certain machines and algorithms even $p \leq s_{rel}$ is possible. Such a superlinear relative speedup can be observed due to memory hierarchies and remote accesses in parallel machines (especially DMMs). For the same reason, a superlinear absolute speedup is possible too. $s_{rel}$ is used to demonstrate how good a certain algorithm can be parallelized. But note, that a slow and inefficient algorithm often can be executed in parallel with large $s_{rel}$. $s_{abs}$ appears in many cases to be quite low, especially for a large number of processors, limited memory in a node processor and a lot of interprocessor communication.

In [4] additionally to the conventional speedup definition, a costup $c$ of a parallel machine as $c = (f(p) + g(m_p))/(1 + g(m))$, where

*This research was partly supported by DFG (German Science Foundation) under SFB 124 - TP D4 and by Universidade de Vigo/Xunta de Galicia
\( f(p) \) is the fixed cost of a parallel system with \( p \) processors normalized to the fixed cost of a uniprocessor not counting memory; \( g(m_p) \) is the memory cost of the parallel machine and \( g(m_1) \) the memory cost of the sequential machine. They consider a parallel machine more cost-effective, if the speedup is greater than the costup, i.e., \( s_{abs} > c \). If memory becomes a significant part of the hardware cost, a parallel system can be more cost-effective than the uniprocessor.

If \( f(p) \) is large, high speedup is necessary to achieve a good cost-effectiveness. But focusing on large problems, let us assume that the memory cost is dominating the cost of the machines. We define the memory costup as: \( c_{mem} = S_p/S_n \), where \( S_p \) is the size of memory for the data required on the parallel machine and \( S_n \) the size on the sequential machine.

The property that a parallel implementation is only cost-effective if the obtained speedup is larger than the costup can be made more descriptive by defining the cost-effective absolute speedup \( \sigma_{eff} \) and the cost-effective absolute efficiency \( \eta_{eff} \) as:

\[
\sigma_{eff} = s_{abs}/c_{mem} \quad \text{and} \quad \eta_{eff} = \sigma_{eff}/p.
\]

A cost-effective efficiency of 1 can be obtained by a parallel algorithm which computes the result \( p \) times faster than the sequential version without using more memory. In [4] a parallel implementation is considered already cost-effective if \( \eta_{eff} > 1/p \). Cost-effective relative speedup and efficiency are defined similarly.

### 3 Ray Tracing

We concentrate only on the geometrical part of ray tracing, i.e., the portion dealing with illumination model, texture mapping, light source modeling etc, although being crucial for a realistic appearance of the images are not of main interest. Their contribution to the overall run time is usually small.

The ray tracing algorithm is quite simple. From a virtual camera, rays are sent out into the scene. The objects of a scene are defined through an intersection routine, which returns whether or not a given ray has an intersection point with the surface of the object. If the objects are bounded, i.e., they have a limited extension in each direction, intersection tests can be simplified, because tests with bounding volumes often allow for a fast rejection of possible candidates for a hit point. Reflections, transparencies as well as shadows can be calculated with the same principle effectively by tracing appropriate rays in a recursive manner.

The main task in the algorithm is: given a ray, find the object that the ray hits first. For a complex scene, most of the run time is spent in this part of the algorithm. In the following, we deal only with the time consuming part.

Fast algorithm have been developed to speedup the ray-object intersection tests. The main ideas are: i) grouping bounding volumes which are close together into larger bounding volumes recursively (tree of extents), ii) sorting the objects into a regular grid structure, iii) building an adaptive regular subdivision (e.g., octree) such that every voxel contains more or less the same number of objects and iv) performing a binary space partitioning and sorting the objects according to the dividing planes.

The performance of the different subdividing strategies depends heavily on the scene, i.e., the distribution and shape of the objects. Generally, the grid structure performs best on somehow evenly scattered small objects, whereas the tree methods appear to be faster if the objects build clusters. A BSP-tree usually generates a similar space subdivision as a hierarchical bounding volume approach.

Besides the optimization of the data structures to reduce the time to find an intersection point, other properties of the geometry in the scene can be exploited. Image coherency, i.e., data locality, provides the possibility of further improvement of the basic algorithm.

Coherency can be found in many areas: surfaces are continuous, only a certain number of objects can be seen in a small solid angle, a lot of rays are traversing the scene almost parallel, and for animations the differences from frame to frame are often small.

Some means to speedup the basic algorithm which are based on data locality use software caching strategies. Examples for such caches are light caches, storing the last objects producing shadows, ray mailboxes, maintaining the last ray which has been intersected with a complex object, or transformation matrices of a hierarchical scene description.

At first sight, it seems that ray tracing can be parallelized easily. In principle, the intersection calculation of each ray can be performed independently from any other ray. In a multi-processor environment each processor calculates a portion of the final image and linear speedup seems to be achievable. There is enough parallelism: millions of rays must be traced in scenes containing several hundred thousand of objects.

Parallel implementations try to find an optimal point for the run time considering all the trade-offs for the specific machine they are implemented for. They all, [5, 6, 7, 8, 9, 10, 11], use one or more of the following methods. Duplication all data of the scene and calculating with each processor only a part of the image is the simplest method. There is almost no communication necessary and each processor can use the best sequential algorithm. However, this approach cannot deal with large scenes, because too much memory is used. The cost-effective speedup is at most \( 1/p \). Load balancing can be static by distributing the pixel area or dynamic by implementing task queues based on ray flow. For a static load balancing strategy a previous subsampling of the image can provide information how to distribute the pixels. In both approaches one has to decide if a ray or an object message should be transferred between processors. The amount of rays that must be
communicated can be quite large and hot spots, e.g. at light sources, can lead to serialization, to saturation of message buffers, or to deadlocks. If object messages are transferred, one has virtually implemented a shared memory emulation. It depends on the subdivision strategy which and how many objects must be sent between the processors.

Efficient implementations are not available for solving the problem for a large scene on a massively parallel general purpose computer; absolute speedups are published nowhere. Many of the optimizations depicted above, which are very effective to speed up the sequential version, are not included in parallel implementations. Most of them would increase the communication amount of the algorithm trying to decrease the amount of computation. Hierarchical data structures which allow to trace large scenes on a single processor can only be handled by the shared memory approach efficiently.

4 Machines and Benchmarks

We have implemented ray trace programs to run on the architectures listed in Tab. 1. The programs for the SB–Pram [3] were executed on the simulator (the machine is still under construction). We measured the run times as wall clock time while being single user on the machines.

We use two benchmark suites to analyze the performance of the different algorithms and machines. The set of nine images of the SPD–benchmark (developed by Eric Haines) tests the performance of ray tracing programs for various features. The scene descriptions are not hierarchical, there is no subdivision provided. To achieve fast run times, a ray tracer must perform the subdivision automatically or a default value must be chosen. We designed the U–benchmark especially to analyze the run time of a ray trace kernel for an increasing number of objects [12]. The scenes consist of non reflecting spheres, which are distributed uniformly in a cubic area. The area is surrounded by a highly reflective cube. The uniform distribution is best case for a grid method but worst case for a hierarchical algorithm, because the latter one always will generate a similar subdivision as a grid. The additional overhead due to the hierarchy usually decreases performance.

5 Results

We have ported three ray trace kernels onto the different machines. Rayshade, a public domain ray tracer, is known as one of the fastest based on grid subdivision. Art, another freely available tracer, uses an BSP–tree subdivision. The third one is Unitree, a ray trace kernel we built ourselves [12].

Tab. 2 shows the profile of a comparison between a one processor KSR–1 and a Sun–Sparc20 workstation using the Unitree algorithm on the SPD–benchmark.

<table>
<thead>
<tr>
<th>machine</th>
<th>p</th>
<th>MHz</th>
<th>MByte</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sun–Sparc2</td>
<td>7</td>
<td>40</td>
<td>32</td>
</tr>
<tr>
<td>Sun–Sparc20</td>
<td>1</td>
<td>60</td>
<td>256</td>
</tr>
<tr>
<td>SGI challenge</td>
<td>1</td>
<td>150</td>
<td>704</td>
</tr>
<tr>
<td>KSR–1</td>
<td>20(18)</td>
<td>20</td>
<td>640</td>
</tr>
<tr>
<td>SB–Pram</td>
<td>128</td>
<td>8</td>
<td>2048</td>
</tr>
</tbody>
</table>

Table 1: Machines

<table>
<thead>
<tr>
<th>machine</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>run time in seconds</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We have not been able to find an explanation why on gears (2) Sparc20 is almost 5 times faster than KSR–1 and on shells (6) KSR–1 is almost twice as fast as Sparc20.

Fig. 1 shows the run times for the different subdivision methods on a Sun–Sparc2 (grid size at 22x22x22). In scenes with irregularly distributed objects, the grid subdivision behaves worst. Art could not deal with the shell scene. The Unitree algorithm performs in all cases good and in many cases best.

We have distributed the image in the parallel version for KSR–1 column wise. Load balancing was not a problem (less than 5% differences). The relative efficiency was in the range of 0.70 to 0.94 and in most cases close to 0.85. Because of the small data bases (< 10000 objects), the cache memory can hold a copy of the whole scene within every node.

The U–benchmark allows to inspect the performance of different subdivision methods with increasing number of objects. The grid method has almost constant run time for a small number of objects, but with the fixed size of the grid, the run time increases rapidly if more objects are present (see Fig. 2 for 100000 objects). The Unitree algorithm performs almost three times better than the BSP–Tree algorithm. Choosing the grid size carefully leads to best run times (single points for 1 million objects correspond to grids of size 20x20x20, 30x30x30 and 40x40x40, respectively).

Due to the highly irregular access to the data base,
preprocessing for the Unitree algorithm was not possible on the KSR-1. We preprocessed the data on SGI challenge and worked with the already structured data base on KSR-1. Table 3 shows the relative and the cost-effective relative efficiencies on the U-benchmark for a cluster of the KSR-1 consisting of 18 nodes. As long as the entire data base could be held in the memory of a node almost linear relative speedup can be obtained. Exceeds the data base the local memory of a node, performance of KSR-1 drops significantly.

A summary of the run times for the different architectures is gathered in Fig. 3. We expect the fastest run time on the SB-PRAM. Simulations show that the 128 node machine will achieve a run time being seven times faster than the run time of a single processor SGI challenge. Note that the clock rate of the SB-PRAM is as low as 8 MHz (compare Tab. 1).

<table>
<thead>
<tr>
<th>#obj</th>
<th>10^0</th>
<th>10^1</th>
<th>10^2</th>
<th>10^3</th>
<th>10^4</th>
<th>10^5</th>
<th>10^6</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\eta_c)</td>
<td>0.922</td>
<td>0.867</td>
<td>0.898</td>
<td>0.865</td>
<td>0.795</td>
<td>0.115</td>
<td>0.056</td>
</tr>
<tr>
<td>(\eta_{eff})</td>
<td>0.051</td>
<td>0.048</td>
<td>0.050</td>
<td>0.048</td>
<td>0.044</td>
<td>0.056</td>
<td>0.056</td>
</tr>
</tbody>
</table>

**Table 3: Relative efficiency on KSR-1**

6 Conclusion

For ray tracing complex scenes we can conclude: one of the best algorithms is Unitree; distributed memory machines have severe problems to overcome the communication bottleneck; cost-effective relative efficiency of KSR-1 with 18 processors is close to 0.056 \(\approx 1/p\); cost-effective absolute speedup of KSR-1 with 18 processors to SGI challenge is less than 0.012; cost-effective relative efficiency of SB-PRAM is always close to 1; cost-effective absolute speedup of SB-PRAM with 128 physical processors to SGI challenge is greater than 7. In other words, to render large scenes, use a sequential fast computer or the SB-PRAM which offers the possibility to implement efficient parallel ray trace algorithms.

Until now, we have analyzed only the basic algorithm. Possible optimizations of the ray tracing algorithm reduce the computing part but increase the communication part. We believe that such efforts will decrease the performance of the KSR-1 (and probably other DMMs), while additional absolute speedup can be expected for the SB-PRAM.

**Acknowledgement:** We like to thank Volker Hofmeier for performing the benchmarks on the KSR-1 and the computing center of the University of Mannheim for giving us exclusive access to the parallel machine.

**References**


