A new parallel algorithm provided by a computation time model

Jean-Christophe Nebel
E-mail: jc@dcs.gla.ac.uk

17 Lilybank Gardens Computer Science Department
University of Glasgow
Glasgow, G12 8QQ, UK

Abstract

Photo realistic algorithms as used in Computer Graphics are resource-hungry. The most promising way of resourcing these algorithms is by using parallel distributed machines.

Ray tracing is widely used both directly to address specular reflection realistically and indirectly in form-factor calculation in radiosity, so is present in the most powerful but most computationally expensive global illumination algorithms used today. Both ray tracing and radiosity thus benefit from improvements in the efficiency of the ray casting component which can result from a good mapping onto distributed resources.

Two distinct algorithms have been advocated for this purpose: object dataflow and ray dataflow, but there is no comparative literature. Here we offer a method for comparison from which we derive the means by which they can be combined to give a more effective, dynamically determined balancing of load over all resources than would be possible with either method alone. The paper concludes with a discussion of results obtained for this new algorithm on a set of standard scenes.

Key-words

Computer graphics, rendering, ray tracing, parallelism, DMPC, MPP, load balancing, radiosity, computation model.

1. Introduction

Rendering algorithms used in computer graphics are expensive in time and memory resources. The most promising way for resolving both of these limitations is the use of parallel distributed machines.

The ray tracing algorithm is the first photo realistic algorithm to simulate specular reflection and refraction through transparent objects [Wh80], which stimulated a lot of research on parallelization of this algorithm. Now new rendering algorithms are appearing, which use the parallelization techniques developed for ray tracing.

We start this paper by studying the two most popular parallel algorithms used for rendering. Next, we describe a computation time model which allows us to compare these algorithms, and to propose an original approach to parallelizing rendering applications with a concurrent algorithm, which uses both object and ray dataflow. Finally, we present experimental results obtained by the implementation of our new technique for the ray tracing algorithm.

2. **Some efficient parallel algorithms for rendering**

Our paper deals with the parallelization of rendering algorithms on parallel computers with distributed memory (Figure 1). Because we want to render photo realistic quality pictures, we are concentrating on parallel methods which allow for more processing memory and CPU power. Accordingly we will not be interested in parallelization methods based on the duplication of the whole scene on the local memory of every processor of the parallel machine.

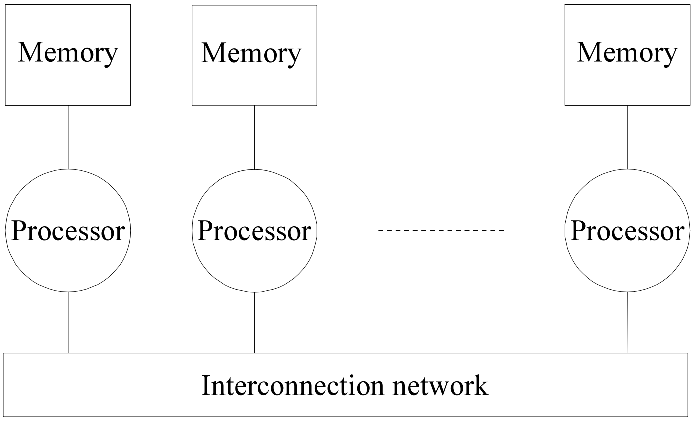

Figure 1: Architecture of a parallel computer with distributed memory.

Because ray tracing was the first photo realistic method but very time consuming, a lot of parallelization techniques have been proposed for this algorithm. Some of these techniques are now used for the parallelization of other photo realistic models, most notable radiosity [Go84, Ni85 and Wa87].

We will now present the two main efficient parallel algorithms which can be used for ray tracing and other rendering methods.

2.1. Algorithms with object dataflow

Objects describing the scene to be rendered are distributed among the local memories of the parallel machine. Then each processor has to compute a subset of all independent computations. For the ray tracing algorithm, the subset is of the pixels in the image.

Because the size of the database is supposed to be bigger than the size of a single local memory, computations cannot be perform without communication. So when a computation cannot be performed because of a missing object, this object is fetched on the other local memory of the parallel computer. Then the processor keeps a copy of this object in its local memory, and the computation can continue. In this way each processor can compute the rays which it has in charge.

This method is used in ray tracing [Gr88, Gr90 and Ba90], and also in the progressive radiosity algorithm [Bo93].

2.2. Algorithms with ray dataflow

The principles of this algorithm are quite different from these of the previous one. Here each processor is in charge of a subspace of the 3D space and so owns objects which belong to its subspace in its local memory. Then each processor performs only computations which are in relation with the objects of its 3D space.

When a computation can only be performed by using an object belonging to another subspace, this computation is sent to the processor owning the data. Then this processor will continue the computation until this computation needs an object from another subspace. By travelling from processor to processor computations can be fully performed.

Many different algorithms based on this approach have been proposed for ray tracing [Ko88, Sa88, Is91, Le93 and Ba94], but also for progressive radiosity [Gu95].

2.3. Conclusion

We have seen that the two efficient classical algorithms for ray tracing are also used for the parallelization of the progressive radiosity algorithm. Now we would like to know which one of these parallel algorithms is the most efficient. Unfortunately, the literature does not give a proper answer to this problem.

3. Model and analysis of the two dataflow schemes

We do not know which of these classical algorithms is the most efficient and we do not want to choose arbitrarily between them. So we will propose our own comparison. The implementation of these two algorithms being a heavy task, we thought that it would be easier and efficient enough to perform a complexity analysis of the two schemes. This analysis leads us to a model of the computation time of the dataflow algorithms. We use the ray tracing example in order to limit our field of investigation.

3.1. Algorithms and assumptions

Our models are for a qualitative comparison purpose only. We just want to determine which of the two parallel dataflow algorithms is the most efficient.

In order to be able to propose models for these two algorithms, we simplify the ray tracing algorithm:

At first the ray tracing algorithm may be seen as a loop, where, at each step, a ray-object intersection is performed for all rays which have been generated. A ray-object intersection involves obviously the intersection computation, but also message passing may be performed before this computation, if the ray or the object is remote.

During the preprocessing step each object is associated to a single processor, which allows each processor to know where to get a missing object.

Being interested in a comparison between the two algorithms, the generation of shadow rays and refracted rays would only complicate the algorithms without adding any relevant information, so we will not deal with them.

Finally no dynamic load balancing schemes are supposed to be implemented on the algorithms we study.

We describe now the two algorithms to which we are going to give a computation time model:

1. Objects are distributed, each of the *nb* processors receiving a fixed percentage, *x*, of the scene objects.
2. The n_0 primary rays are distributed, each processor receiving n_0/nb rays.
3. At each step, *(1-x)* of the rays need objects which are not owned by their processor.
4. Then an object or a ray dataflow strategy is used:

a. Object dataflow algorithm:

- Each time a ray needs an object, a request is sent to the processor which owns the missing object.
- This processor sends back a copy of the missing object and the ray computation is performed.

b. Ray dataflow algorithm:

- Rays needing an object are sent to the processor which owns the object (rays are buffered, each processor sends only one message to each of the other processors).
- When all rays are on the processor possessing the needed object, ray computations are performed.

5. When ray processing is done, the ray is deleted and a new ray is generated. From a given step, rays start to disappear because it is supposed that they have finished their journey through the scene. As such, some deleted rays are not replaced.
6. After all processors have finished computing their rays, return to step '3'.
7. Terminate when all rays have disappeared.

Before presenting our computation models, we present some assumptions for the parallel ray tracing algorithms we want to model:

- Object distribution is supposed to be well balanced.
- The number of rays to be computed is larger than the number of objects.
- All objects have the same probability for intersecting at every step of calculation.
- We assume that the size of an object is bigger than the size of a ray and that each object has the same size.
- We assign a fixed computation time to every distinct operation.

3.2. Computation time models

In order to compare the two classical dataflow algorithms, we propose two computation time models for the previous algorithms. These models are based on recurrence relations, which gives computation time for the object and ray dataflow algorithms, depending on the percentage of objects owned by each processor and on the number of processors.

We use the following notations:

n_0 is the total number of rays to compute at the beginning of the algorithm.
nb is the number of processors used, $nb>1$.
x is the fraction of the database owned by each processor, $x \in]0,1]$.
$r_{i,j,k}$ is the number of rays which cannot be computed on processor i at stage k without communication with the processor j.
$n_{i,k}$ is the number of rays owned by the processor i at stage k, $n_{i,k} \in \mathbb{N}^*$.

In the following, equations are given for the *permanent regime*, that is the regime in which the total number of rays is constant on the simulated parallel machine. Each processor owns the fraction *x* of the database, so only *x* of the rays being on a processor *i* can be computed without any communication:

$$\forall (i,k) \in (N\text{x}N^*), 0 \leq i < nb, r_{i,i,k} = x * n_{i,k-1}$$

Then, *(1-x)* of the rays owned at step *k-1* cannot be computed on processor *i* without any communication. There are *nb* processors, so:

$$\forall (i,j,k) \in (N\text{x}N\text{x}N^*), 0 \leq i,j < nb, i \neq j, r_{i,j,k} = \frac{1-x}{nb-1} * n_{i,k-1}$$

The previous equations do not involve any load unbalance between processors. In order to have more realistic models, we add a variable $\zeta_{i,j,k}$ which models unbalancing at every step *k* for each processor *i* in relation with a processor *j* ($\zeta_{i,j,k} \in [-1,1]$). The previous equations now become:

$$\begin{aligned}
&\forall (i,j,k) \in (N\text{x}N\text{x}N^*), 0 \leq i,j < nb, \\
&\text{if } i = j,\ r_{i,j,k} = x * n_{i,k-1} * (1 + \xi_{i,j,k}) \\
&\text{if } i \neq j,\ r_{i,j,k} = \frac{1-x}{nb-1} * n_{i,k-1} * (1 + \xi_{i,j,k})
\end{aligned} \tag{1}$$

Now we express the number of rays *n* that a processor *i* has to compute at a given step *k*.

For the object dataflow algorithm, this number is constant in the *permanent regime*:

$$\forall i \in N, 0 \leq i < nb, n_{i,k} = \frac{n_0}{nb}$$

But for the ray dataflow algorithm, this number is the number of rays needing the processor *i* in order to be computed at step *k*.

$$\forall (i,j) \in (N\text{x}N), 0 \leq i,j < nb, n_{i,k} = \sum_j r_{j,i,k} \tag{2}$$

The equation (1) is true if $\zeta_{i,j,k}$ does not induce any change in the total number of rays. So,

$$\forall i \in N, 0 \leq i < nb, \sum_i n_{i,k} = n_0 \tag{3}$$

By replacing $n_{i,k}$ in the equation (3) by the expression (2), we get:

$$\forall (i,j) \in (N\text{x}N), 0 \leq i,j < nb, \sum_i \sum_j r_{j,i,k} = n_0 \tag{4}$$

Finally in replacing $r_{i,j,k}$ in equation (4) by its expression in equation (1) we have a relation between the $\zeta_{i,j,k}$:

$$\forall(i,j,k) \in (N\mathrm{x}N\mathrm{x}N^*), 0 \le i, j < nb,$$
$$\sum_j \left(x * n_{i,k-1} * (1+\xi_{i,j,k}) + \sum_{i \neq j} \frac{1-x}{nb-1} * n_{i,k-1} * (1+\xi_{i,j,k}) \right) = n_0 \tag{5}$$

Now we express the computation time of each algorithm. Message passing with asynchronous communication is the protocol of communication we want to simulate. So the data transmission costs are not relevant for our study, nevertheless we cannot ignore the startup times and the packing and unpacking times (copy of data to or from a communication buffer).

Here is the notation for the computation time of each single operation:

T_{ray} is the computation time of a ray.
T_{sending} and T_{receipt} are the startup times for sending and receiving a message.
$T_{\mathrm{pack\_object}}$ and $T_{\mathrm{unpack\_object}}$ are packing and unpacking times for an object.
$T_{\mathrm{pack\_ray}}$ and $T_{\mathrm{unpack\_ray}}$ are packing and unpacking times for a ray.
$T_{\mathrm{pack\_req}}$ and $T_{\mathrm{unpack\_req}}$ are packing and unpacking times for an object request.

The computation time for a processor *i* to perform the step *j* using the ray dataflow algorithm is noted by $TC_{i,j}$. Its expression is:

$$TC_{i,j} = n_{i,j} * T_{ray} + T_{sending} * (nb-1) + T_{pack\_ray} * \sum_{k \neq i} r_{i,k,j} + T_{receipt} * (nb-1) + T_{unpack\_ray} * \sum_{k \neq i} r_{k,i,j}$$

We can deduce that the needed time for performing one full step *j* is the time of the slowest processor at step *j*:

$$TC_j = Max_i(TC_{i,j})$$

So the computation time for performing all steps with the ray dataflow algorithm is:

$$TC = \sum_j TC_j = \sum_j Max_i(TC_{i,j})$$

Now we do the same thing for the object dataflow algorithm. The computation time for a processor *i* to perform the step *j* using the object dataflow algorithm is noted by $TO_{i,j}$. Its expression is:

$$TO_{i,j} = n_{i,j} * T_{ray} + \left(T_{sending} + T_{pack\_req} + T_{receipt} + T_{unpack\_object}\right) * \sum_{k \neq i} r_{i,k,j}$$
$$+ \left(T_{sending} + T_{pack\_object} + T_{receipt} + T_{unpack\_req}\right) * \sum_{k \neq i} r_{k,i,j}$$

So the computation time for performing all steps with the object dataflow algorithm is:

$$TO = \sum_j Max_i(TO_{i,j})$$

3.3. Results

We tested our models for various numbers *nb* of processors and for various fractions of the database owned by each processor *x*. We made several experiments by changing the ratios between computation times of each single operation. With realistic ratios, we always obtained charts with the following shapes:

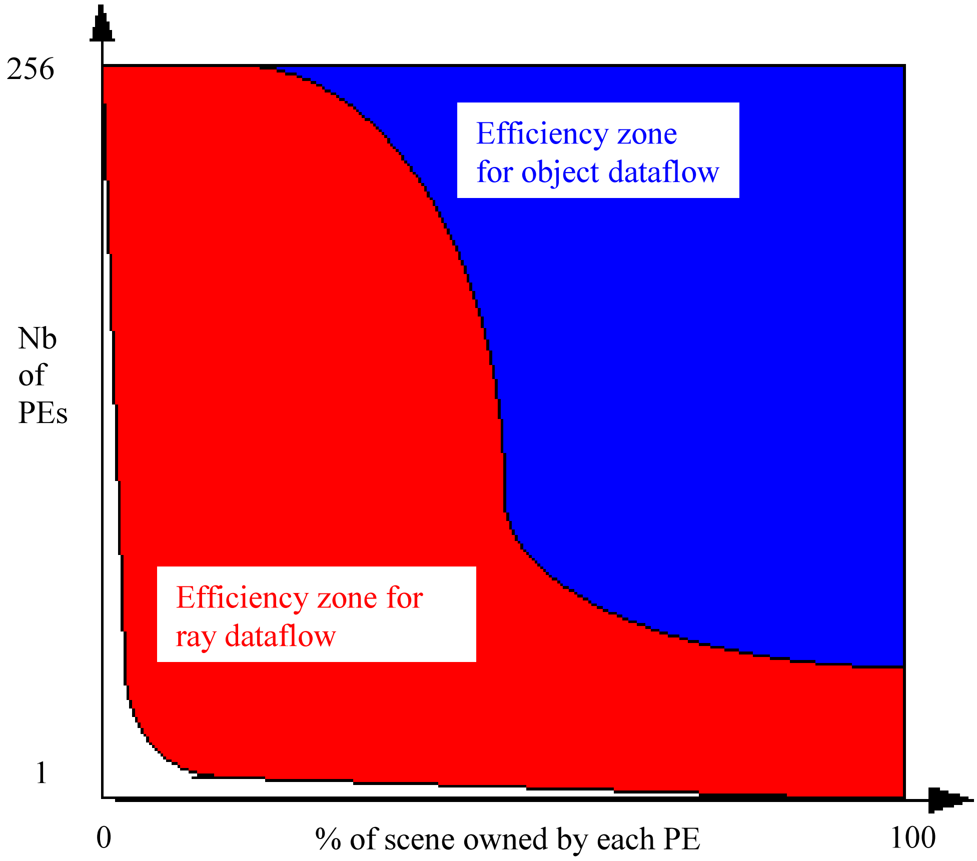

Figure 2: Comparison between the two classical dataflow algorithms using models.

Shaded areas on Figure 2 indicate whether the ray dataflow or object dataflow algorithm was more efficient. First, we see that the object dataflow scheme is best when each processing element (PE) deals with a large part of the scene. The result is reversed when processors have only a small part of the scene. Furthermore, we notice that increasing the number of processors seems to favour the object dataflow algorithm.

Although our model is based on a lot of simplifications (no consistency, no load balancing, step-by-step…), the results raise some questions:

1. Should we choose between the two algorithms depending on the number of processors and the scene distribution?
2. Should we choose dynamically, instead of doing it at the preprocessing time?

Our goal is to propose a parallel algorithm which could be used for a large range of applications, so only a dynamical choice seems to be suitable. We will explore this idea in the following.

4. A concurrent dataflow algorithm

Using results obtained previously, we give the main principles of the concurrent dataflow algorithm we propose. Finally we show some experimental results obtained with this new algorithm implemented for the ray tracing algorithm.

4.1. Principles

Previously we have seen that both dataflow algorithms may be useful to increase efficiency depending on the number of processors and the size of the local memory available. That is why our new algorithm will give the ability to every processor to choose the most efficient type of dataflow dynamically. This arose from the development of a parallel algorithm that included the two modes of dataflow, i.e. a concurrent dataflow algorithm.

With this algorithm, every processor can exchange both objects and computations at any moment. So the key is the way to choose between these two kinds of dataflow for optimal efficiency. This choice has to be guided by two main goals:

1. Keeping optimal load balancing between nodes of the parallel machine.
2. Reducing the amount of communication.

In order to achieve these goals, every processor needs several parameters for obtaining optimal choice. Some are local, but others are not. Every processor needs to know:

1. The load of every processor.
2. The number of computations which cannot be computed in the absence of communication.
3. The list of all the remote objects needed for the computations which cannot be performed.

Most of these parameters can be computed locally, but the load on other processors cannot. This new algorithm needs a scheme which allows every processor to know the load of the others. Furthermore, considering that we wish to work also on massively parallel machines, we want to avoid "master-slave schemes" because it leads to communication bottlenecks. Global information should be transmitted by a non-centralized strategy, which assures scalability and does not produce extra messages. Thus each time a message is sent because objects are missing, it includes a header, which contains the load of the sender i.e. the number of computations still to be computed. After a few messages, each processor has the knowledge of the load of every other processor in the parallel system.

With this scheme, every processor gets sufficiently precise information for a very low cost (only the size of messages is increased). Using this information and its own parameters a node has the capability to choose between object and ray dataflow at any instant.

The following is an outline of the algorithm:

```
IF my load is superior to a minimal load
        (I can send some computations to perform to another processor)
THEN
        FOR every processors
                IF my load is superior to the processor load
                THEN
                        Send computations to the processor
                ENDIF
        ENDFOR
ENDIF
FOR every missing piece of data
        (each piece of data is supposed to be own by a single processor)
        IF the number of computations needing the piece of data is sufficient
         THEN
                Ask for the piece of data
        ENDIF
ENDFOR
```

For example, if a processor is busy, it will choose to send computations to processors which have a lower load. If a processor is idle, it will choose to ask for objects in order to perform computation locally.

Finally, our algorithm offers a dynamic management of concurrent dataflow, which intrinsically assures dynamic load balancing.

4.2. Experimental results

Now that the principles of our algorithm have been explained, we present our experimental results (the entire results can be found in [Ne97]).

Our strategy has been implemented using the sequential ray tracer OORT [Wi94]. The results presented here have been obtained using a CRAY T3E with 128 processors interconnected by a 3-D torus topology. We used PVM as a message-passing library. The size of each picture is 512x512 pixels and the maximal ray depth is 5. Scenes, i.e. *Mountain5*, *Ring1, Tetrahedron5* and *Tetrahedron6*, come from the SPD database [Ha87] and contain between 341 and 8688 objects.

4.2.1 Comparison between the two classical algorithms

Our new algorithm including the two modes of dataflow, so it is possible by setting some parameters to use it in each of the classical modes. In this way, we can compare the two classical algorithms experimentally and check if the results obtained are similar to those obtained by the analytical method.

The comparison between these two algorithms is expressed in term of the *time profit* of the object dataflow algorithm, defined here as the processing time needed by the ray dataflow algorithm divided by the processing time needed by the object dataflow algorithm.

The next figure shows the time profit for the scene *Tetrahedron5* using various numbers of processors:

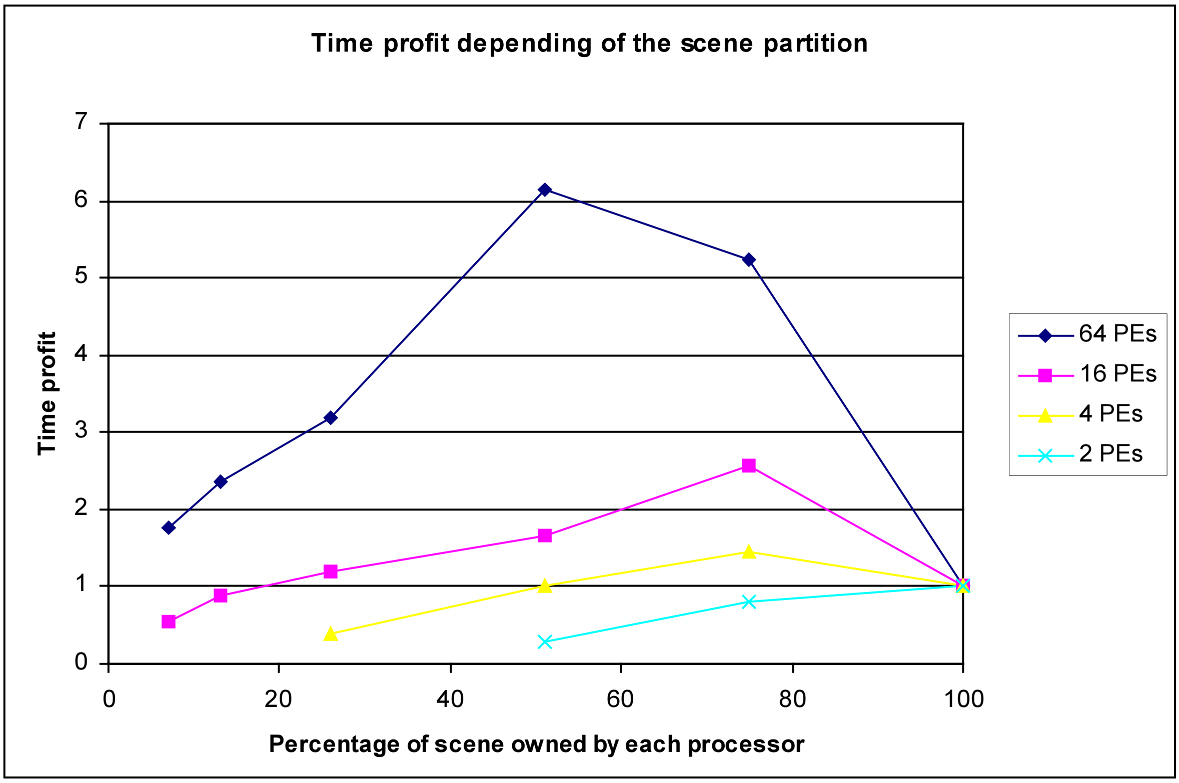

Figure 3: Experimental comparison between the two classical dataflow algorithms.

We find again the results of our model (Figure 2):

1. The object dataflow scheme is the best when the scene is lightly partitioned (right side of Figure 3).
2. The ray dataflow scheme may be the best only when the scene is extensively partitioned (left side of Figure 3).
3. The object dataflow scheme is the best when the number of processors is high.

Our computation time model appears to be efficient enough to predict these experimental results, so we can assume that the new algorithm to which it contributed to create should be efficient.

4.2.2 Results provided by our new algorithm

We compare the results given by our concurrent dataflow algorithm with those of a classical object dataflow algorithm, which appears to be more efficient than the classical ray dataflow algorithm for the studied configuration (a machine with 64 processors). The comparison between these two algorithms is expressed in term of the *time profit* of the concurrent dataflow algorithm.

The next figure shows the time profit obtained by the concurrent dataflow algorithm for various scenes using 64 processors of the CRAY T3E:

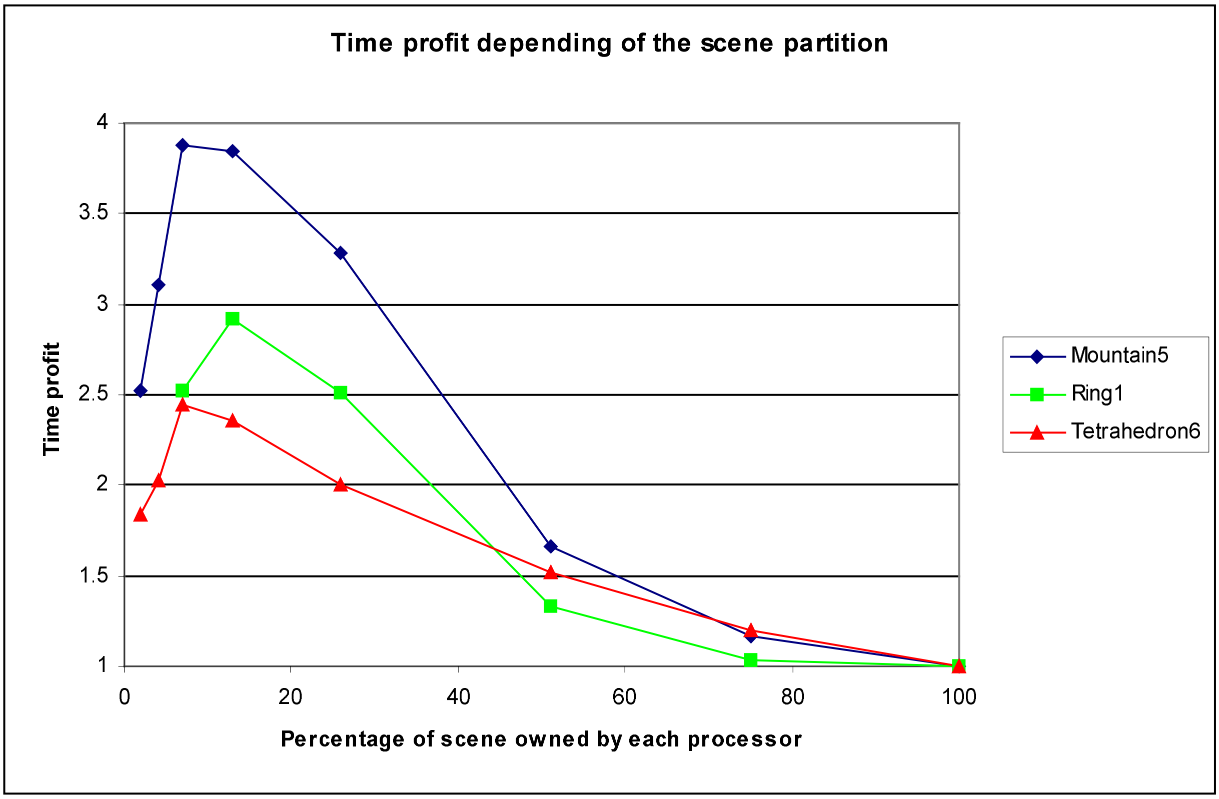

Figure 4: Time profit depending of the scene partition for the concurrent dataflow algorithm.

At first it appears that whatever scene is rendered, the time profits provided by our new algorithm are always quite high: the processing time of the *Mountain5* scene has been divided by almost 4! Moreover charts have very similar shapes, which suggests that the general behavior of the algorithm is scene independent. The best results are obtained when each processor owns a low percentage of the scene. When processors own less than the half of the object database, profits vary between 3.9 and 1.5, otherwise they are between 1.7 and 1.

This fact can be easily explained by the way in which global information is transmitted to processors. This information is distributed by messages generated for missing objects. When each processor owns most of the scene objects, very few messages are sent, so refresh rates for this information is quite low. Accordingly its precision may not be good enough to permit the best choice in every case. One solution to resolve this lack of precision would be to generate extra messages providing load information when the refresh rate is too low.

Our algorithm needs a sufficient flow of communications to reach its highest efficiency. When there is too much communication - when less than 10 % of the scene is owned by each processor -, some messages cause wait states. Then time profits provided by our algorithm decrease. (Nevertheless they continue to be better than 1.8).

One goal of our concurrent algorithm was to reduce computation time by reducing the number of messages which are sent. In the next figure, we show the *message profits* induced by our new algorithm, defined here as the number of messages sent by the object dataflow algorithm divided by the number of messages sent by the concurrent dataflow algorithm.

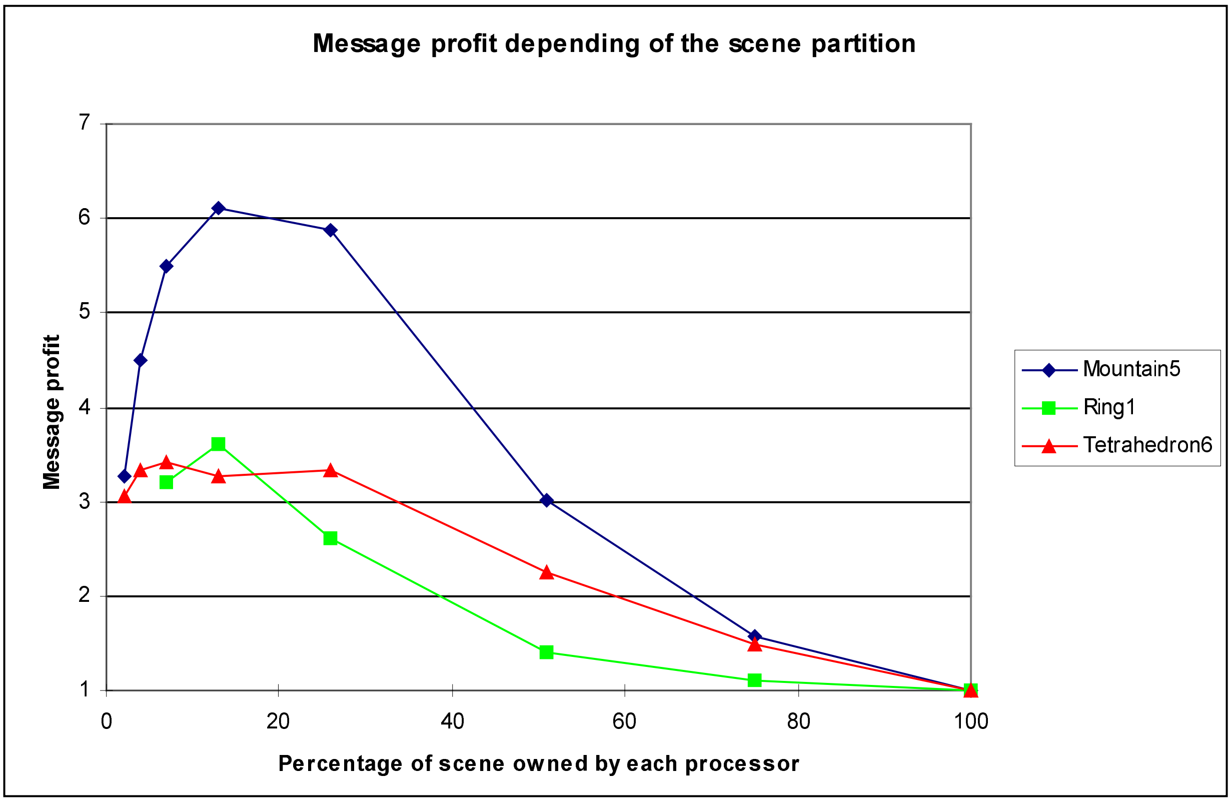

Figure 5: Message profit depending of the scene partition for the concurrent dataflow algorithm.

The charts on figure 5 are very similar to these of the figure 4, which clearly indicates that our concurrent algorithm improves computation times due to the reduction of message numbers. Message profits are between 6.1 and 1.7 when processors have loaded less than half of the database. Again we see that waiting for messages decreases profits.

Our algorithm using concurrent dataflow gives very encouraging results, as the computation time for rendering pictures and the number of exchanged messages are reduced by significant factors in relation to a classical dataflow algorithm. This reduction of the magnitude of the flow of data communication reduces the risk of network saturation.

The previous results show that our algorithm is efficient on an architecture of 64 processors. In order to express its scalability, on the next figure we present its acceleration as a function of the number of processors. The charts are given for the picture *Mountain5* for various percentage of scene owned by each processor.

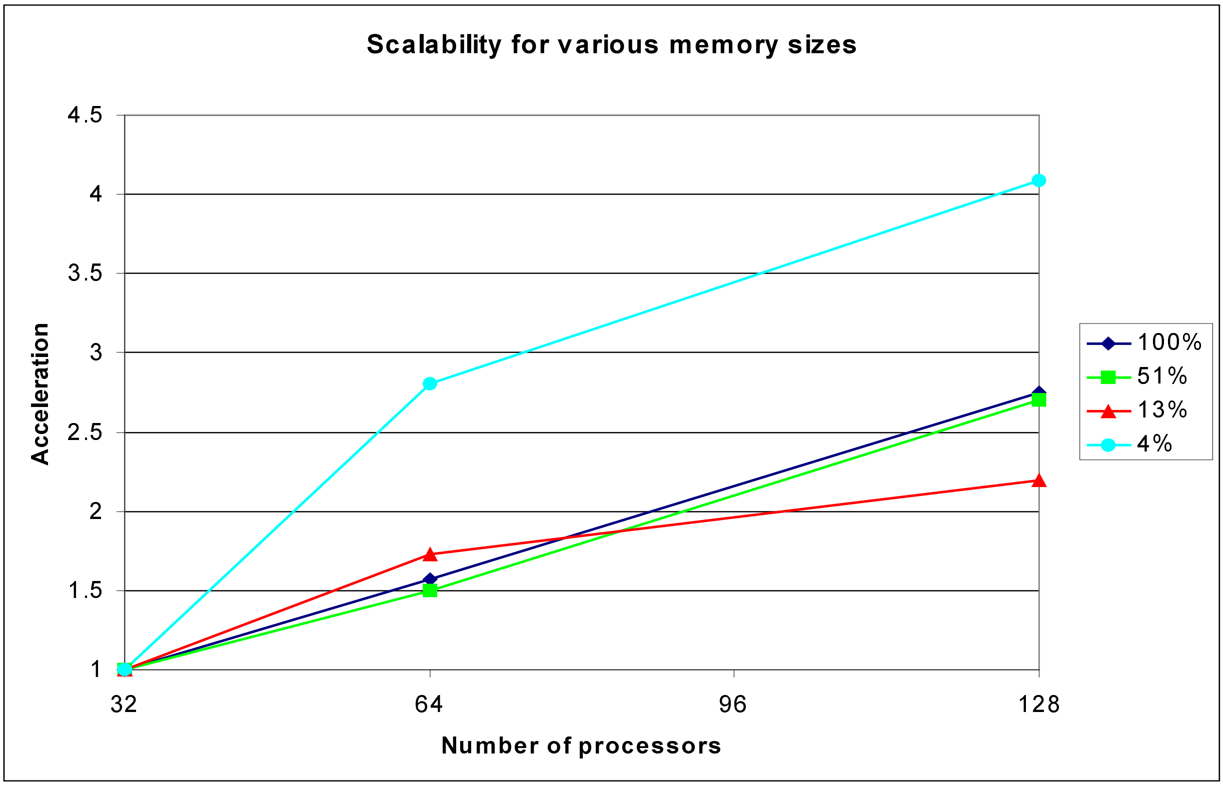

Figure 6: Scalability of the concurrent dataflow algorithm for various memory sizes.

Whatever the percentage of scene owned by each processor, the acceleration between 32 and 128 processors is over 2.2. So this algorithm appears as being scalable on our architecture. That was predictable because our algorithm provides a significant reduction of communication and a dynamic load balancing scheme.

5. Conclusion and future work

In this paper, we have shown that the classical parallel algorithms used for ray tracing may be used for other rendering algorithms. Then we presented a model for some parallel ray tracing algorithms. The results lead us to propose a new parallel algorithm for rendering, based on concurrent dataflow.

As performances on a CRAY T3E show, the implementation of this algorithm for the ray tracing application gives encouraging results seeing that the computation time for a picture has been reduce by 3.9 in relation to the performance of the best classical dataflow algorithm. Moreover communication flows are substantially reduced and the algorithm is scalable.

Our concurrent scheme is currently used for the parallelization of an algorithm of particle simulation.

Acknowledgements

We thanks Eric Haines for supplying the SPD package [Ha87] and Dr. John Patterson for helpful comments on the paper. This work was supported by the CEA (Commisariat à l'Energie Atomique) and the EMSE (Ecole des Mines de Saint-Etienne).

References

[Ba90] D. Badouel and T. Priol. An efficient parallel ray tracing scheme for highly parallel architectures. *Advances in computer hardware v. rendering, ray tracing audiovisualisation systems. Lausanne, CH, 2-3 September 1990*, pages 93-106, September 1990.

[Ba94] D. Badouel, K. Bouatouch and T. Priol. Distributed data and control for ray tracing in parallel. *IEEE computer graphics and applications*, 14(4), pages 69-76, July 1994.

[Bo93] K. Bouatchou, D. Menard and T. Priol. Parallel radiosity using a shared virtual memory. *First Bilkent computer graphics conference on advanced techniques in animation, rendering and visualization, Ankara,* 1993

[Go84] C. M. Goral, K. E. Torrance, D. P. Greenberg and B. Battaile. Modeling the interaction of light between diffuse surfaces. *SIGGRAPH'84*, 18(3), pages 213-222, July 1984.

[Gr90] S.A. Green and D.J. Paddon. A highly flexible multiprocessor solution for ray tracing. *The visual computer*, 6(2), pages 62-73, March 1990.

[Gr88] S.A. Green, D.J. Paddon and E. Lewis. A parallel algorithm and tree-based computer architecture for ray traced computer graphics. *Parallel processing for computer vision and display. Leeds, UK, 1988*, January 1988.

[Gu95] P. Guitton, J. Roman and G. Subrenat. *Implementation results and analysis of a parallel progressive radiosity. Proceedings of parallel rendering symposium, Atlanta, 30-31 October 1995,* pages 31-38, October 1995.

[Ha87] E. Haines. A proposal for standard graphics environments. *IEEE computer graphics and applications*, 7(11), pages 3-5, November 1987.

[Is91] V. Isler, C. Aykanat and B. Ozguc. Subdivision of 3-D space based on the graph partitioning for parallel ray tracing. *Proc. second eurographics workshop on rendering, univ. of Catalonia, Barcelona*, 1991.

[Ko88] H. Kobayashi, S. Nishimura, H. Kubota, T. Nakamura and Y. Shigei. Load balancing strategies for a parallel ray-tracing system based on constant subdivision. *The visual computer*, 4(4), pages 197-209, October 1988.

[Le93] W. Lefer. An efficient parallel ray tracing scheme for distributed memory parallel computers. *Proc. of parallel rendering symposium (1993), San Jose, California, October 25-26*, pages 77-80, October 1993.

[Ne97] J.-C. Nebel. *Développement de techniques de lancer de rayon dans des géométries 3-D adaptées aux machines massivement parallèles*. PhD thesis, Université de Saint-Etienne, Saint-Etienne, December 1997.

[Ni85] T. Nishita and E. Nakamae. Continuous tone representation of three-dimensional objects taking account of shadows and interreflection. *SIGGRAPH'85*, 19(3), pages 23-30, July 1985.

[Sa88] J. Salmon and J. Goldsmith. A hypercube ray-tracer. Proc. 3rd conference on hypercube concurrent computers and applications. 2, pages 1194-1206, January 1988.

[Wa87] J. R. Wallace, M. F. Cohen and D. P. Greenberg. A two-pass solution to the rendering equation: A synthesis of ray tracing and radiosity methods. *SIGGRAPH'87*, 21(4), pages 311-320, July 1987.

[Wh80] T. Whitted. An improved illumination model for shaded display. *Communication of the ACM*, 23, pages 343-349, 1980.

[Wi94] N. Wilt. *Object-oriented ray tracing*. Wiley, 1994.