Message Handling in Parallel Radiance

Erik Reinhard and Alan Chalmers

Department of Computer Science,
University of Bristol, United Kingdom
E-mail: reinhard@compsci.bristol.ac.uk

Abstract. Photo-realistic computer graphics is an area of research which tries to develop algorithms and methods to render images of artificial models or worlds as realistically as possible, by carrying out lighting simulations. Such algorithms are known for their unpredictable data accesses and their high computational complexity. Rendering a single high quality image may take several hours, or even days. For this reason parallel processing must be considered as a viable option to compute images in a reasonable time. The nature of data access patterns and often the sheer size of the scene to be rendered, means that a straightforward parallelisation, if one exists, may not always lead to good performance. This paper discusses a suitable parallelisation of an advanced ray tracing algorithm using PVM, and presents a method of reducing the number of messages required for data and task communication by bundling a number of tasks or data items into a single message.

1 Introduction

Physically correct rendering of artificial scenes requires the accurate simulation of light behaviour. Such simulations are computationally very expensive and may take several hours or even several days.

Radiance [1] is an advanced ray tracing algorithm [2] which simulates light by following for each pixel of the image one or more rays into the scene. If such a primary a ray hits an object, the light intensity of that object is assigned to the corresponding pixel (figure 1). In order to model shadows, from the intersection point of the ray and the object, new rays are spawned towards each of the light sources. These rays, called shadow rays, are used to test whether there are other objects between the intersection point and the light sources, indicating the intersection point is in shadow, or whether the intersection point was directly lit. Mirroring reflection and transparency may be modelled similarly by shooting new rays into the reflected and/or transmitted directions (figure 1 left). These reflection and transparency rays are treated in exactly the same way as the primary rays. Hence, ray tracing is a recursive algorithm.

This recursive process has to be carried out for each individual pixel separately. A typical image therefore consists of at least a million primary rays and a multiple of that in the form of shadow, reflection and transparency rays. The most expensive part of the algorithm is the visibility calculation. For each ray, the object that is intersected by the first ray, must be determined. To do this,

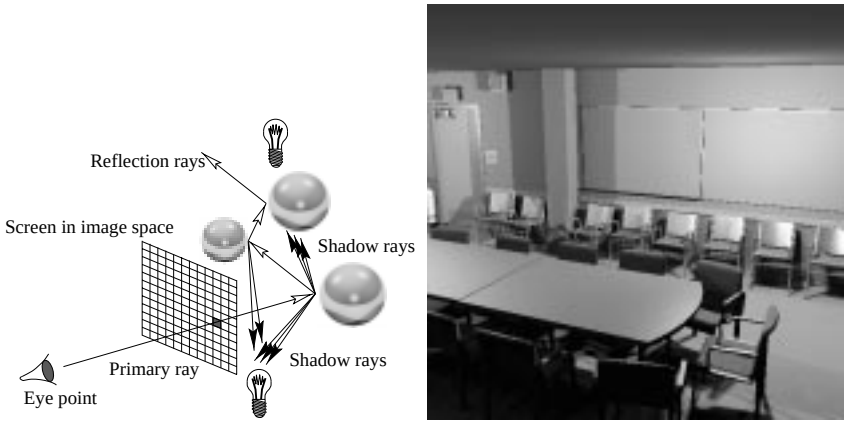

Fig. 1. Ray tracing in a nutshell (left) and test scene used for evaluation (right).

a potentially large number of objects will have to be tested against each ray. Spatial subdivision techniques [3] have reduced that number dramatically, but a significant number of intersection tests is still required. Hence the need for parallel processing, as further sequential algorithmic improvements are not imminent.

The most obvious parallelisation of the ray tracing algorithm is called screen space subdivision [4] [5] [6]. This subdivides the image into a number of regions, and each region is allocated to a processor when it has finished its previous region. In parallel processing terms this is a standard demand driven approach [7] [8]. The amount of communication can be kept to a minimum, while load imbalances are virtually non-existent. However, this approach only works well when the scene is small enough to be duplicated with each processor. If that is not feasible, for example because the scene is larger than a single processor's memory, the scene data must be distributed over the local memories, and data communication will occur. Caching strategies may reduce the amount of data communication, but unfortunately data access is quite unpredictable, which will therefore almost inevitably lead to low cache hit rates.

Alternatively, the data may be distributed over the processors, and instead of fetching the required data, each task is migrated to the processor that stores the relevant data items [9] [10] [11]. This approach, termed data parallel processing, will allow almost arbitrarily large scenes to be rendered. Again, data access will be unpredictable, which means that the load of each processor is extremely difficult to determine beforehand. Load balancing is the major problem with this approach. Communication will be intense too, but can be kept local, i.e. between neighbouring processors only.

An alternative approach is to try and combine demand driven and data parallel execution of tasks on the same set of processors[12] [13] [14]. A basic, if uneven, load is established by the data parallel component, while idle processors may request additional tasks from a master processor. This should bring the

best of both worlds: a good load balance due to the demand driven component, and the ability to render very large models due to the data parallel component. The name *hybrid scheduling* is coined for this type of approach.

Overlaying data parallel and demand driven tasks, however, brings about some problems in the way tasks and data are to be communicated. Some tasks should have higher priority than others, and additionally, some task and data messages are very good candidates to be bundled into larger messages. Sending various tasks in a single message has the simple advantage that start-up latencies, necessary to establish a connection between two processors, are reduced. However, collecting different tasks and data items into a single message, means that sending the message has to be deferred until enough components are collected. This paradox can potentially reduce performance, unless a well balanced method is devised to keep each processor busy, while at the same time allowing tasks and data to be bundled into sufficiently large messages. The following section presents such a general scheme, while in section 3, its merits using PVM [15], are assessed. Further measurements giving confidence in the technique are presented in section 4 and the results are finally discussed in section 5.

2 Buffering techniques

Our implementation adopts an asynchronous mode of operation. Each processor will query its input buffer and when a message is found, the associated task is executed. This procedure is repeated until the input buffer is empty, or an exit message is received. When the input buffer is empty, a new task is requested from the master processor and a wait state is entered. This is the only way idle time may occur. Whenever a processor generates a new task, or is requested to send a data item to another processor, its associated data is packed in an output buffer, and the message is sent. This means that each message contains only one task or one data item. In a parallel ray tracing algorithm, the number of messages can be quite prohibitive if this overly simple message handling algorithm is used.

The number of messages can be reduced by bundling a number of tasks or data items, or even mixtures of both, into a single message. In the new scheme, each message has a priority level associated with it. Any number of levels may be chosen, but for simplicity we have used four: highest, high, medium and low. Each message sent has a PVM message tag reflecting its priority.

There still is only one input buffer for each processor in this method. However, a message received in this buffer may contain a number of different tasks, all having the same priority as the message itself. Each task, or data item, is preceded by a tag indicating the type of task. Based on this tag, the appropriate unpacking routine is called, followed by a call to the appropriate function which will execute the task. This procedure is repeated until all tasks in the message are executed. The order in which incoming messages are handled, is important as well. Obviously, the highest priority messages should be executed first.

Instead of having a single output buffer, which is sent after packing each task, the new scheme utilises a number of output buffers. For each priority and each of the remaining processors there is an output buffer. When a task is to be sent to another processor, the $< destination, priority >$ tuple determines in which buffer the task has to be packed. Whenever a task is packed into a buffer, the number of tasks in that buffer is checked. When a priority level dependent threshold is reached, the buffer is sent. High priority tasks are sent sooner than low priority tasks. The threshold for each of the priority levels is a user parameter.

This basic algorithm does provide an opportunity for starvation to occur. Two processors may be starved when they have tasks for each other, but refrain from sending because the respective output buffers are not yet filled. When the master processor starts distributing the last few tasks remaining, it broadcasts a message to all processors. Once received, each processor will send its output buffers whenever its input buffer becomes empty. Deadlock does not occur because the (ray tracing) algorithm is implemented as a fully asynchronous system.

The main parameter to alter the performance of this message handling approach, is the *send threshold* for each of the priority levels. Sending too often is suboptimal because of start-up latencies, while keeping the messages in their output buffers too long will cause other processors to become idle. The number of priority levels is a parameter largely determined by the type of application. A simple demand driven or data parallel approach may need fewer than four levels, while a hybrid scheduling algorithm needs more.

3 Buffering applied to ray tracing

The basic message handling algorithm described in the previous section was applied to a hybrid scheduling parallel ray tracer. Recall that load balancing in the hybrid algorithm is achieved by overlaying demand driven and data parallel tasks on the same set of processors. Each processor is therefore capable of executing both sollicited and unsollicited tasks. The unsollicited tasks (data parallel) are given a higher priority in our system than the demand driven tasks. This ensures that demand driven tasks are only executed when a processor becomes idle.

Within the group of data parallel tasks, a distinction is made between tracing rays and shading rays. Shading a result, means that some book keeping memory may be freed [14]. In order to conserve as much memory as possible, shading tasks get a higher priority than trace tasks.

The highest priority, however, should be given to data requests. If a processor becomes idle because it has requested data from another processor, the other processor should send the data as quickly as possible, thereby minimising the amount of idle time. The hierarchy of priorities for our parallel ray tracer is therefore in descending order: data requests and data items, shading, secondary rays and primary rays.

4 Measurements

The tests in this section were performed on a cluster of Sun Sparc workstations running PVM. The message handling technique was implemented in a hybrid scheduling parallel ray tracer, based on the Radiance lighting simulation package [1].

The test scene used for evaluation is large (23,177 polygons) with clusters of objects dotted around the scene (figure 1 right). This reflects the type of model that is likely to be rendered in industrial applications. Furthermore, the scene has many light sources, adding to the computational requirements of the rendering task. The images rendered were all 512 x 512 pixels, with primary ray tasks having a size of 16 x 16 pixels. This gives a total of 1024 primary ray tasks to be distributed in demand driven fashion over the available processors.

Messages are bundled according to their importance, i.e. important ones are sent sooner than less important ones. In table 1, for each priority level the number of tasks allowed per message is given. The last two columns present the average number of tasks per message as measured during the test runs. These two averages, one for 4 processors and one for 8 processors, are used in subsequent graphs to index the idle time and communication time that has occured. Note that during the first measurement (row 1, table 1), for reference purposes, messages were not bundled at all.

Rendering	Priority level				Average	
	Low	Medium	High	Highest	4 Procs	8 Procs
1	1	1	1	1	1	1
2	16	8	4	2	1	1
3	32	16	8	4	1.97	1.97
4	64	32	16	8	3.83	3.83
5	128	64	32	16	7.30	7.32
6	256	128	64	32	12.91	13.15
7	512	256	128	64	23.21	23.23
8	1024	512	256	128	31.39	32.39

Table 1. Bundling for 8 different measurements performed on the Conference room.

Keeping tasks in a buffer longer, would suggest that idle times increase, while the time spent in communication routines would decrease. Figures 2 and 3 give idle and communication times for the 4 and 8 processor cases.

The communication times decrease according to our expectations and are not much different for the four and eight processor cases. If communication is the only concern, then bundling of as many tasks into a single message as possible, is the best sollution.

Unfortunately the idle times recorded fluctuate somewhat. This may have something to do with a varying workload on different processors as a result of other users running tasks on the same workstations. As a general trend, the

least idle time recorded was when about 7 tasks were bundled into a single message. Sending smaller messages clogs up the system with large numbers of small messages, while larger messages will starve the receiving processors.

Summing both idle times and communication times, as in figures 2 and 3, gives the overall time wasted due to message bundling. Again, the mimimum for this graph would indicate that about 7 tasks per message is optimal.

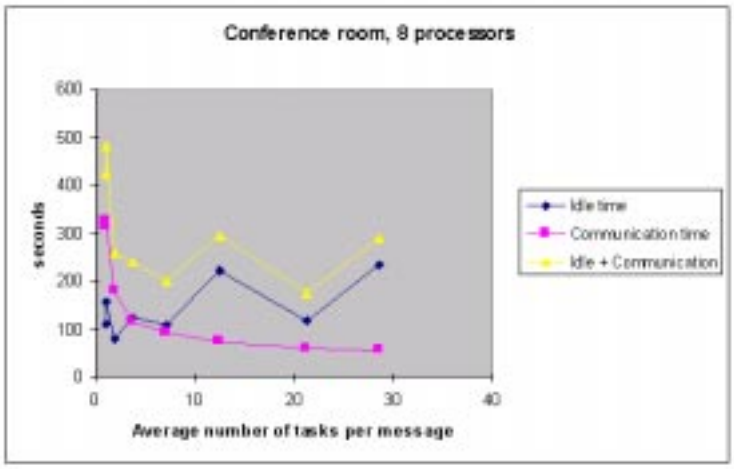

Fig. 2. Idle time, communication time and the sum of these two as function of the average number of tasks per message for the conference room with eight processors.

In our implementation, both idle time and communication time do not play an overly significant role in the total performance. The efficiency, measured here as the time spent doing useful computations divided by the total rendering time, is between 90% and 92% for four processors and between 89% and 91.5% for eight processors. Most of the efficiency loss is due to idle time at the end of the computation. During the bulk of the computation, hardly any time is spent on communication and most processors can be kept busy.

5 Discussion

This paper set out to prove that a good efficiency can be maintained using a hybrid scheduling algorithm, where the large data set is distributed amongst the processors. By carefully choosing which tasks to migrate to the processors which store the relevant data and for which tasks to fetch data from remote processors, high efficiencies can indeed be obtained. Furthermore, a general approach is presented for sending messages between different slaves and between slaves and the master processor. By giving each task a priority, different tasks can be bundled into a single message, provided they are to be sent to the same processor. The

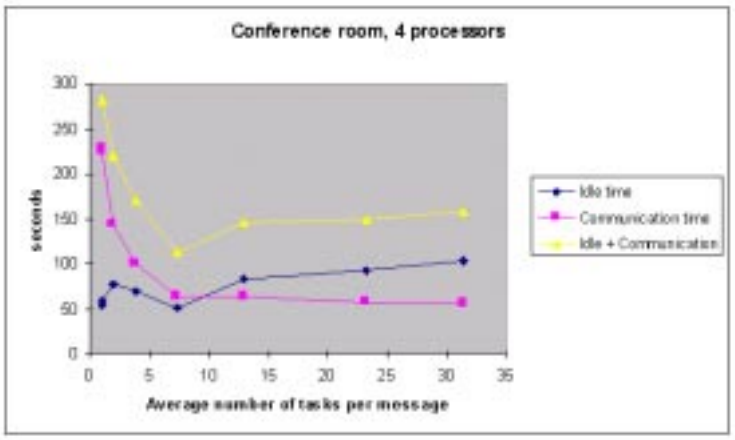

Fig. 3. Idle time, communication time and the sum of these two as function of the average number of tasks per message for the conference room with four processors.

number of tasks that can be bundled this way can be controlled by specifying a threshold for each priority. Typically, the lower priority tasks get bundled to a higher degree than important tasks. In our implementation, for instance, data fetches are deemed very important, because as long as a processor is waiting for data, it may not have any work to do.

The downside of bundling messages is a theoretical increase in idle time. Processors may have to wait longer for data and tasks to arrive. Unfortunately, testing on machines which are used by others at the same time, may result in biased measurements. If a processor gets less CPU time than other processors, it may hold up sending data to other processors, which consequently may suffer from idle time. This may account for the fluctuations in idle times between different renderings perceived in figures 2 and 3.

On average, our parallel ray tracing implementation benefitted from having an average of 7 tasks per message. This means that more important tasks are sent sooner, and less important ones later. Reordering of message types, i.e. giving tasks a different priority, may affect the overall performance, although on theoretical grounds, the chosen ordering is expected to be near optimal. Future work will consider rearranging priority levels.

Acknowledgements

This work was funded by the EC under TMR grant number ERBFMBICT960655.

References

1. Ward, G.J.: The radiance lighting simulation and rendering system. *ACM Computer Graphics*, SIGGRAPH '94 Proceedings. (1994) 459–472

2. Whitted, T.: An improved illumination model for shaded display. *Communications of the ACM*, **23** (1980) 343–349
3. Glassner, A.S., editor.: *An Introduction to Ray Tracing*. Academic Press, San Diego, (1989)
4. Plunkett, D.J., Bailey, M.J.: The vectorization of a ray-tracing algorithm for improved execution speed. *IEEE Computer Graphics and Applications*, **5** (1985) 52–60
5. Crow, F.C., Demos, G., Hardy, J., McLaugglin, J., Sims, K.. 3d image synthesis on the connection machine. In *Proceedings Parallel Processing for Computer Vision and Display*, Leeds, (1988)
6. Lin, T.T.Y., Slater, M.: Stochastic ray tracing using SIMD processor arrays. *The Visual Computer*, bf 7 (1991) 187–199
7. Green, S.A., Paddon, D.J.: Exploiting coherence for multiprocessor ray tracing. *IEEE Computer Graphics and Applications*, (1989) 12–27
8. Shen, L.S., Deprettere, E., Dewilde, P.: A new space partition technique to support a highly pipelined parallel architecture for the radiosity method. In *Advances in Graphics Hardware V, proceedings Fifth Eurographics Workshop on Hardware*. Springer-Verlag, (1990)
9. Dippé, M.A.Z.,Swensen J.: An adaptive subdivision algorithm and parallel architecture for realistic image synthesis. *ACM Computer Graphics*, **18** (1984) 149–158
10. Cleary, J.G., Wyvill, B.M., Birtwistle, G.M., Vatti, R.: Multiprocessor ray tracing. *Computer Graphics Forum*, (1986) 3–12
11. Kobayashi, H., Nishimura, S., Kubota, H., Nakamura, T., Shigei, Y.: Load balancing strategies for a parallel ray-tracing system based on constant subdivision. *The Visual Computer*, **4** (1988) 197–209
12. Scherson, I.D., Caspary, C.: A self-balanced parallel ray-tracing algorithm. In P. M. Dew, R. A. Earnshaw, and T. R. Heywood, editors, *Parallel Processing for Computer Vision and Display*, Wokingham, Addison-Wesley Publishing Company.**4** (1988) 188–196
13. Jansen, F.W., Chalmers, A.: Realism in real time? In *4th EG Workshop on Rendering*, (1993) 1–20
14. Reinhard, E., Jansen, F.W.: Rendering large scenes using parallel ray tracing. In A. Chalmers and F. W. Jansen, editors, *First Eurographics Workshop on Parallel Graphics and Visualisation*, Bristol, Alpha Books. (1996) 67–80
15. Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., Sunderam, V.: *PVM 3 User's Guide and Reference Manual*. Oak Ridge National Laboratory, Oak Ridge, Tennessee, (1993) Included with the PVM 3 distribution.

This article was processed using the LaTeX macro package with LLNCS style