Hybrid Scheduling for Parallel Rendering using Coherent Ray Tasks

Erik Reinhard[*] University of Bristol
Alan Chalmers[†] University of Bristol
Frederik W Jansen[‡] Delft University of Technology

ABSTRACT

Parallelising ray tracing with a data parallel approach allows rendering of arbitrarily large models, but the inherent load imbalances may lead to severe inefficiencies. To compensate for the uneven load distribution, demand-driven tasks may be split off and scheduled to processors that are less busy. We propose a hybrid scheduling algorithm which brings tasks and data together according to coherence between rays. The amount of demand-driven versus data-parallel tasks is a function of the coherence between rays and the amount of imbalance in the basic data-parallel load.

Processing power, communication and memory are three resources which should be evenly used. Our current implementation is assessed against these requirements, showing good scalability and very little communication at the cost of a slightly larger memory overhead.

CR Categories: D.1.3 [Programming Techniques]: Parallel Programming; I.3.0 [Computer Graphics]: General I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism

Keywords: Parallel computing, hybrid scheduling, ray tracing

1 INTRODUCTION

Physically correct rendering of artificial scenes requires the simulation of light behaviour. Such lighting simulations can be very expensive in terms of computation and memory requirements. This means that an accurate lighting simulation may take between a couple of hours to several days to compute, even with current state of the art hardware. Moreover, the scene may include several millions of polygons, enhanced with megabytes of texture data. For radiosity and diffuse ray tracing, additional meshing and illuminance caching data structures may also take significant amounts of memory.

Parallel processing offers the possibility of speeding up realistic rendering algorithms. Previous approaches can be roughly categorised into demand driven and data parallel techniques. Demand driven [8], usually equivalent to image space partitioning [5, 13, 1], subdivides the screen into a number of regions, where each region represents a task. A number of processors executes these tasks and whenever a task is completed, a processor requests a new one from the master - hence the name demand driven.

If the entire scene description can be replicated over the processors, this approach may lead to very good speed-ups and is known to be ‘embarrassingly parallel’. However, for very large data sets, this assumption may not hold and data must be distributed over the processors. Data communication then becomes inevitable, which has an adverse effect on the speed-up. Some performance may be regained by employing caching techniques. However, the amount of data communication and the lack of data coherence between cached objects may destroy the efficiency.

Data parallel approaches [6, 3, 11, 1] are usually object space partitioning techniques. Here, the geometry is distributed over the processors and ray tasks are migrated to the processors that hold the relevant data. If a ray traced by a certain processor does not intersect any objects, it is sent to the processor that holds the neighbouring data. That processor will then continue tracing the ray; a process repeated until either the ray leaves the environment or an intersection is found.

Due to the distribution of data, very large scenes may be rendered. Unfortunately, both view point and light sources may prove to be hot-spots in the scene, and will lead to load imbalances. Furthermore, the number of rays that migrate to a different processor may be very high, resulting in a large communication overhead.

Overlaying demand driven and data parallel tasks could overcome the disadvantages of both these techniques. Such hybrid scheduling techniques [18, 10] can lead to a good load balance, while still being able to render very large scenes. An obvious scheme would be to subdivide the work into demand driven ray traversal and data parallel ray intersection tasks [18, 12]. The spatial subdivision structure is then replicated, while the geometry is distributed. However, the demand driven tasks should be sufficiently expensive to compensate for the load imbalance caused by the data parallel component. Otherwise, control over work load and communication distributions may be lost and performance degrades as in data parallel approaches. Ray traversal typically takes less than 10% of the total rendering time, and hence a good load balance cannot be achieved using ray traversal only.

Alternatively, coherence could be the criterion to subdivide tasks into data parallel and demand driven parts [16]. Coherent tasks, which will require a small data set while being sufficiently expensive, are good candidates to re-schedule as demand driven tasks, while the remainder of the tasks is handled in data parallel fashion. In [16] the demand driven component only consisted of primary rays, which ran out early during the process, leaving the rest of the computation with the low efficiency normally associated with data parallel processing. In this paper we extend the hybrid approach to include the demand-driven scheduling of shadow ray tracing. We also envisioned a further increase of the total processing load by using the ray tracer Radiance [20] that can also sample the diffuse inter-reflection. However, this aspect is, due to the severe demands on memory, not yet part of the work reported here.

In the next sections our hybrid scheduling algorithm is presented. It discusses task and data management techniques employed within our parallel Radiance implementation, as well as the implications of sampling diffuse inter-reflection and tracing area light sources

[*] Dept. of Computer Science, Merchant Venturers Building, Woodland Road, Bristol BS8 1UB, United Kingdom, e-mail: reinhard@cs.bris.ac.uk

[†] Same address, e-mail: Alan.Chalmers@bris.ac.uk

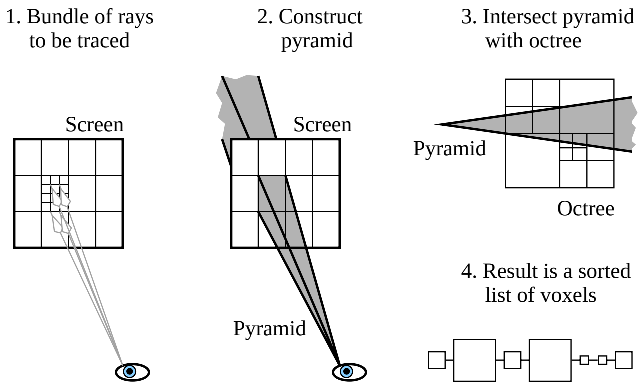
[‡] Fac. of Information Technology and Systems, Zuidplantsoen 4, 2628 BZ Delft, The Netherlands, e-mail: f.w.jansen@its.tudelft.nl

on the parallelisation. The current implementation and its performance are assessed in sections 6 to 8 and finally in section 9 the relevant conclusions are drawn. More elaborate overviews of different scheduling techniques for parallel rendering can be found in [4, 2, 15].

2 HYBRID SCHEDULING

As indicated in the previous section, the scheduling technique presented in this paper is of the hybrid variety and its implementation is similar to the one in [16]. This section briefly discusses the essentials of that algorithm, as well as the differences with our current implementation.

The algorithm consists of a demand driven part and a data parallel part. Each processor handles both types of tasks, but gives data parallel tasks a higher priority. This ensures that no new tasks are requested from the master as long as tasks are still available. This mechanism automatically balances the load. Obviously, enough computationally intensive demand driven tasks should be available to balance the load right until the end of the computation.



1. Bundle of rays to be traced
2. Construct pyramid
3. Intersect pyramid with octree
Screen
Screen
Pyramid
Octree
4. Result is a sorted list of voxels
Pyramid

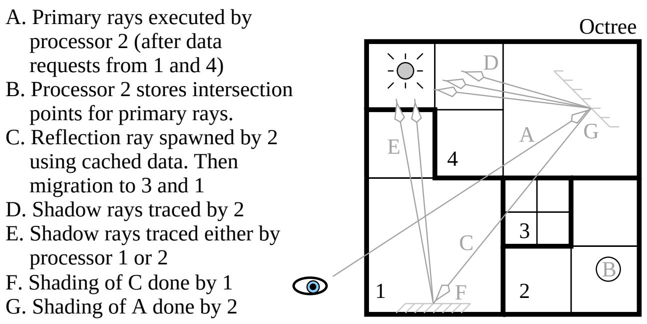
Figure 1: Pyramid clipping is an efficient data selection algorithm.

Besides a high workload, demand driven tasks should need relatively little data, so that scheduling such tasks does not involve an excessive amount of data communication. Our approach to accomplish this uses a screen space subdivision to generate bundles of primary rays. Prior to the execution of such bundles, it is necessary to establish whether all the required data is available. This data selection is implemented using a process called pyramid clipping [22]. A pyramid is constructed around the bundle of rays and it is intersected with the spatial subdivision structure, resulting in an in depth order sorted list of voxels which contain the objects that are needed to complete the task (figure 1). This list is then used to fetch any missing objects.

Next to primary rays, shadow rays (assuming area light sources) form bundles which can be handled together instead of separately. The same pyramid clipping algorithm can be applied to these rays to find a small subset of data which is valid for all the rays in the bundle and thus creating a suitable demand driven task.

Reflected and refracted rays do not exhibit sufficient coherence, making it difficult to predict which data will be necessary to trace them. In order to avoid having to communicate large quantities of object data, incoherent ray tasks are migrated between processors.

Primary rays are scheduled demand-driven to any processor that requests for work. A processor that executes a primary ray task, will also execute the (primary) shadow rays. Specular reflection and diffuse inter-reflection rays are initiated by the same processor but are further processed in a data parallel way. Subsequently these are transferred to the processors that store the (possibly cached) cells that they traverse. After finding an intersection, secondary shadow rays are processed either locally or returned to the first processor for demand-driven processing, thus avoiding local hot spots. This latter task is the main component in our demand-driven scheduling (see also section 5). Finally, the shading of intersection points always takes place with the processor that found the corresponding intersection point. See figure 2 for an example of a primary ray and its descendants.



A. Primary rays executed by processor 2 (after data requests from 1 and 4)
B. Processor 2 stores intersection points for primary rays.
C. Reflection ray spawned by 2 using cached data. Then migration to 3 and 1
D. Shadow rays traced by 2
E. Shadow rays traced either by processor 1 or 2
F. Shading of C done by 1
G. Shading of A done by 2
Octree
D
E
A
G
4
3
C
B
1
F
2

Figure 2: Example showing which processors execute which tasks.

In summary, this hybrid scheduling approach features an automatically balanced load by assigning demand driven tasks to processors that are not busy executing data parallel tasks. Data communication is minimised by migrating incoherent tasks, while the demand driven component should require sufficient computer power to last until the end of the computation and thus minimise idle time. This is accomplished by treating both primary and shadow rays as demand driven tasks (the latter was not yet implemented in [22]).

3 DATA DISTRIBUTION

Ideally, a data distribution should be such that each partition attracts more or less the same workload, while minimising the amount of (data) communication. The memory requirements for each partition should be more or less the same as well. Our data distribution is generated according to a cost function applied to all the cells of an octree. In our initial implementation, the objects in the leaf cells are counted, while the cells higher in the hierarchy have a cost function equal to the sum of the cost functions of their children. Once the cost function for the octree is known, the tree is split as near to the root as possible into partitions which will reside at different processors. The resulting distribution therefore adheres to the equal memory requirement. More elaborate schemes which take into account all of the above requirements, are currently under investigation [17].

A second data distribution, which may be regarded as a worst case scenario, would be to have one processor keep the entire scene description, while all the other processors have just the empty octree. This scheme is used to assess the load balancing capabilities of the hybrid scheduling algorithm by artificially placing a hot-spot at one processor. This processor will initially attract all the data parallel rays, while at a later stage, when sufficient objects have been cached elsewhere in the system, the data parallel workload should even out more. However, data fetches necessary to execute demand driven tasks, will all involve this single processor. Both object counting and worst case data distributions are evaluated in section 7.

Given these data distributions, the data parallel part of our hybrid scheduling algorithm is in place. Each processor will store the entire octree, but only the cells assigned according to the data distribution, will actually contain geometry data. Replication of internal octree nodes and distribution of scene data is similar to the data distribution described in [18].

When a processor receives data from a remote processor to handle a demand driven task, the objects will be cached in the octree. The data structure will therefore remain a single unified octree for each processor, despite having cached data. Each octree cell stores additional information indicating whether the data contained within is cached or not, and which processor contains the data permanently. Storing cached objects in the octree has the advantage that these objects can be transparently accessed for handling data parallel tasks as well as demand driven tasks. A least recently used (LRU) cache replacement policy is invoked whenever memory overflow is detected. When this happens, some performance is lost, but at such moments any system is more concerned with survival than with efficiency. Implications on memory usage are discussed further in section 8.

4 DIFFUSE INTER-REFLECTION

As our parallel implementation is based on Radiance, there is an additional challenge in the form of diffuse inter-reflection. This involves constructing a hemisphere around each intersection point and shooting diffuse inter-reflection rays into each direction [21] (figure 3). Sampling a hemisphere would indicate that on average, half the scene is sampled. The potentially huge amount of data required for this task implies data parallel rendering.

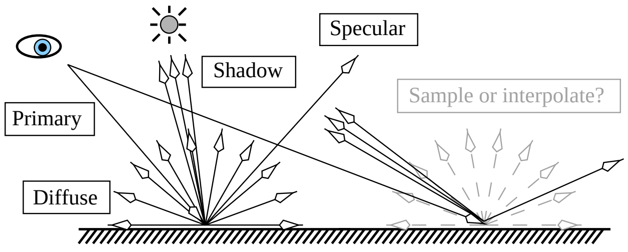


Figure 3: Types of ray within a complete ray tracer (left section). Sampling a hemisphere or interpolating between previously sampled ones depends on position (right section).

When, for diffuse inter-reflection rays, one of the termination criteria is reached, no more diffuse sampling is performed, but shadows as well as specularity are still tested. This accounts for the fact that despite the large number of inter-reflection rays, the number of (coherent) shadow rays is still larger. The ratio of demand driven work to data parallel work is therefore still in favour of the demand driven component.

As the cost of sampling diffuse inter-reflection is huge, once a hemisphere is sampled, its result is stored in an irradiance cache [21]. When sufficient data is collected, for new intersection points the cached values will be interpolated, rather than shooting a new hemisphere (figure 3). This saves an enormous amount of computation, but relies on new hemispheres being shot after previous ones have been completed. In our parallel ray tracer, this leads to subtle data dependencies, which reduce performance, as the previous hemisphere computation may not yet be complete before a decision to possibly interpolate is required. There seem to be two options to circumvent this issue. The first is for each processor to sample at most one hemisphere at a time. This would lead to extra idle time, as the arrival of shading results depends on the workload of other processors.

The other option is to take the extra amount of work for granted and shoot multiple hemispheres. Extra work is performed, because inter-reflection rays that would otherwise be interpolated, are now traced. This provides an extra workload which would not be generated in the sequential algorithm. Currently, sampling diffuse inter-reflection is an issue under investigation.

5 AREA LIGHT SOURCES

Area light sources are usually subdivided into patches, where a (jittered) ray is shot towards each of the patches. The shadow rays then all originate from the same intersection point and travel towards adjacent patches (figure 3). Coherence implies that such a bundle of rays probably needs only a small subset of the data which can be easily determined using the pyramid clipping algorithm [22].

The main difference between primary ray tasks and shadow ray tasks, is that the latter can be generated by any processor, while primary ray tasks are always distributed by the master processor. In order to determine the processor which should receive the primary ray task, some additional scheduling needs to be performed. Scheduling demand driven tasks would be straightforward if each processor were aware of its workload relative to all the other processors. However, this would involve broadcasting status reports to each of the other processors on a regular basis. In order to avoid this overhead, a number of alternative scheduling techniques is proposed.

First of all, each processor that generates a demand driven task, may execute the task as well (in the example in figure 2 step E would be executed by processor 1). This has the advantage that there is no additional task communication, although data may still have to be fetched. Unfortunately, keeping tasks local means that if one processor becomes a hot-spot, it cannot offload a number of tasks to less busy processors. If many diffuse inter-reflection rays are to be computed, the data parallel component of our algorithm will then lead to a load imbalance which is not resolved by keeping shadow tasks locally.

Alternatively, processors that create demand driven shadow tasks, could send these to the master processor which will then allocate them to the first processor that requires more work. Dependent on the number of shadow tasks created, the number of processors asking for work and the size of the object cache, this may lead to a communication bottleneck with the master processor. If diffuse inter-reflection is sampled, then this method should properly balance the workload.

Finally, one method of scheduling is to execute the shadow tasks at the processor that spawned the primary ray leading up to these tasks. This increases the size of the demand driven primary ray task and offloads work from the processor that owns the intersected patch (processor 1 in figure 2). In this way the master processor still schedules shadow tasks, although indirectly.

6 IMPLEMENTATION

The techniques discussed in this paper are implemented in Radiance [20], while the communication is handled using PVM [7]. This ensures that the software runs on a wide variety of platforms, and may thus be useful to a large user base. In the current parallel implementation, we have maintained the full functionality of Radiance, except that there is no super-sampling for primary rays. For the moment we also had to defer using diffuse inter-reflection, due to memory use.

All the tests in the next sections are performed on a Parsytec CC with 32 nodes consisting of a 133 MHz PowerPC 604 and 96 MB of memory. Our tests have shown that about 64 MB per node is available to the user. Communication is handled via a network with a maximum sustained speed of 27 MB/s. Each node runs the PARIX operating system with a homogeneous version of PVM, called PowerPVM. This version of PVM is optimised for speed and a small memory overhead is reported. Latency and throughput are good for communication between neighbours in the network, but for global communication the throughput is much worse and especially set-up times are in the order of seconds [9]. For this reason, our application employs a message buffering scheme which uses PVM buffers to collect tasks into larger messages [14]. This reduces communication overhead substantially for less important tasks. Tasks that require speedy delivery are sent quicker.

The most important tasks are those that involve data fetches (requesting and sending the data). These are sent as soon as a data item is needed, i.e. the bundling is effectively disabled for these messages. The second most important type of messages are shading results. These messages need to be communicated fairly quickly, because upon arrival they result in freeing up memory. After that, all data parallel tasks have a yet lower priority, but not as low as all demand driven tasks. The relative priority given to data parallel and demand driven tasks is the key to achieving load balancing within a hybrid scheduling algorithm.

Unfortunately, it appears that PowerPVM does not allow a graceful exit when memory for one of the processors overflows. This may occur during runs where the load is particularly unbalanced, causing one processor's input buffer to overflow. For this reason we were forced to limit the size of the model to just over 1 MB, leaving about 3 MB for caching and storing intermediate results and a sizeable amount of memory for input buffers. Also, instead of rendering larger models to test the algorithm's behaviour when relatively small subsets of data are stored with each processor, we have opted to reduce the size of the object cache while keeping the scene constant.

Object statistics	Studio	Conference
Total number of objects	5311	3951
Primitives	4606	3737
Instances	0	6151
Materials and textures	705	214
Octree statistics		
Leaf nodes	25733	8667
Internal nodes	3676	1238
Object references in octree	51925	29569

Table 1: Statistics for the scenes and their spatial subdivisions. Object references indicate the number of pointers from the octree to objects. If an object intersects more than one voxel, it will be pointed to by more than one voxel.

7 PERFORMANCE

The results presented in this section all pertain to the studio model, with only a few results replicated for the conference room model (figure 4 and table 1) to verify that the results are not scene-dependent, but indeed measure the behaviour of our algorithms. Image resolution was 512^2 pixels and other parameters include soft shadows, but no diffuse inter-reflection. All optimisations such as adaptive super-sampling have been switched off, with the exception of optimisations involving shadow rays. Shadow rays are therefore sorted according to their contribution to the shading result and the least important ones are approximated without tracing [19].

The next section discusses efficiency issues for the case when the cache for storing objects for demand driven tasks is sufficiently large. The most appropriate scheduling technique for this case is to keep shadow tasks with the processor that generates them. When the size of the object cache is artificially reduced, simulating the case when the scene cannot be replicated, the scheduling mechanism becomes more important. This case is investigated in section 7.2.

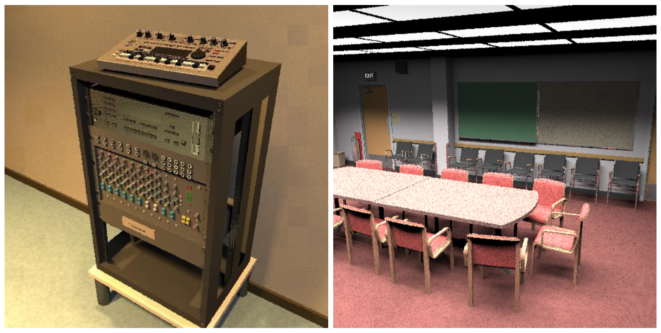


Figure 4: Studio model (left) and conference room (right).

7.1 Speed-up and efficiency

Speed-up graphs for our algorithm are presented in figure 5. When measured with respect to the one processor parallel case, the graphs indicate good scalability. Although this is a common way of measuring speed-up, we have also included fair speed-up graphs, i.e. with respect to the sequential algorithm. From these graphs, it can be learned that a significant overhead is incurred, most of which is processor independent, as argued below. Correcting for this overhead, i.e. computing the efficiency by taking the total rendering time and subtracting the time spent in overhead routines, a different picture emerges. The efficiency is then around 64% for one processor and further reduced to about 55% for sixteen processors (figure 6). The conference room model scores better with an efficiency of between 84% and 73%.

Given the rendering parameters used, computing the same images with the sequential algorithm takes 5.02 hours (18072 s.) for the studio model and 23.78 hours (85611 s.) for the conference room. The long rendering time is mainly due to the sampling of the rather large area light sources. Our measurements on the parallel algorithm indicate a slightly higher workload of 5.46 and 25.88 hours. The main reason for this is that this workload includes the cost of performing timing measurements and instrumenting the code (the results of which can be seen in figures 6 to 11). The speed-up graphs produce a nearly straight line up until the maximum of 32 processors, indicating good scalability. The performance loss due to parallelising the algorithm has a number of different reasons.

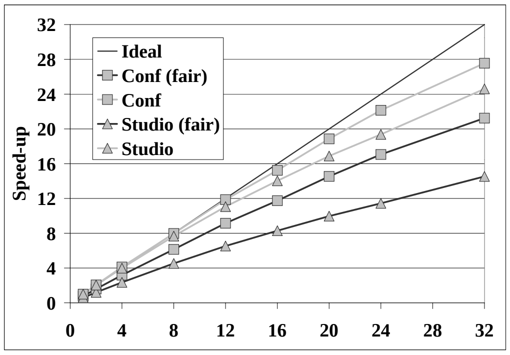


Figure 5: Speed-up graphs for the studio and conference room models with respect to the one processor parallel case and the sequential algorithm (fair).

First of all, data selection, necessary for establishing which data is required for demand driven tasks, constitutes a significant overhead (figure 7). The total amount of time spent in the pyramid clipping routine depends on the number of demand driven tasks created, and is therefore not dependent on the number of processors. The number of shadow rays shot per light source determines the efficiency of the pyramid clipping algorithm. In the studio model, the light sources are relatively far away and can therefore be sampled with relatively few shadow rays. Hence the pyramid clipping algorithm begins to dominate the efficiency of the computation. The conference room has a geometry such that the light sources are quite close to the objects they illuminate. Therefore, the number of voxels between any intersection point and the light sources is small. This makes pyramid clipping relatively cheap. Also, this closeness accounts for the fact that a much larger number of shadow rays is shot to account for soft shadows. For this reason the efficiency of the rendering is much higher.

The second most important form of overhead is idle time, which is mainly introduced by waiting for data for demand driven tasks and lack of new tasks at the end of the computation (figure 8). Waiting for data, rather than starting a new demand driven task, reduces storage requirements. As memory is not available in unlimited quantities, we have chosen to accept some idle time. The distribution of idle time over the different processors appears to be uneven, but this is due to starvation at the end of the computations. Otherwise, idle time is suffered by all processors in equal amounts. This indicates that hybrid scheduling is indeed capable of providing an even load. Idle time near the end of the computation could be reduced by distributing smaller tasks when some processors nearly run out of work.

As our implementation shows a rather substantial idle time when fetching data, which can be attributed to the long start-up time for communicating under PVM, we have looked into ways to improve the message bundling scheme (as discussed in section 6). For data fetches we modified the algorithm to only send requests for data after finishing pyramid clipping for a shadow task, rather than sending a request whenever an object was needed. This delayed sending the first request for data, but at the same time reduces the number of separate messages. Measured on 32 processors for the studio model, this modification resulted in an efficiency improvement of 3.8% which was due to a reduction in idle time by 17% and a reduction in task communication time of 86%[1].

Task communication (figure 9) and data communication for demand driven tasks (figures 10 and 11) appear not to be bottlenecks within this algorithm. The times reported for sending and receiving object data are currently negligible, but these are likely to deteriorate when larger scenes are considered. Then, the entire scene will cease to fit into a single processor's memory and therefore objects are likely to be fetched more than once. It appears that it is possible to increase the number of object communications quite drastically before it starts to dominate the computation.

Figure 11, depicting how much time each processor spends sending data to other processors, gives an indication of the success with which the scene data was initially distributed. Only five processors are ever requested for data, which suggests that the scene was unevenly distributed (an even data distribution would equalise the number of object requests over all processors). This graph coincides with our initial data distribution, which indicates that our caching scheme currently removes these processors as hot-spots. When very large scenes are to be rendered, investing more work into finding good initial object distributions would be worthwhile, as in that case caching alone would not be sufficient. Note also that the more processors that participate in the calculations, the easier it is to find an even object distribution. This is especially true for scenes that have an uneven object distribution over space (such as the studio model).

[1]Note that the speed-up graphs in figure 5 are based on results without this improvement.

7.2 Scheduling and caching

If all the objects that are required to complete a demand driven task, can be stored in memory, without having to replace previously cached data, the hybrid scheduling algorithm resembles a demand driven approach. No special scheduling is required to keep the workload balanced, other than the distribution of demand driven primary ray tasks. Shadow rays can be effectively computed by the processor that spawns them. However, when the size of the object cache is reduced, so that only a portion of the entire scene description fits, the capability to perform demand driven tasks is reduced. Rays that can not be entirely executed in demand driven fashion, need to resort to data parallel scheduling. If the object cache becomes smaller and smaller with respect to the total scene size, the amount of data parallel work grows, making the hybrid scheduling algorithm resemble a data parallel approach.

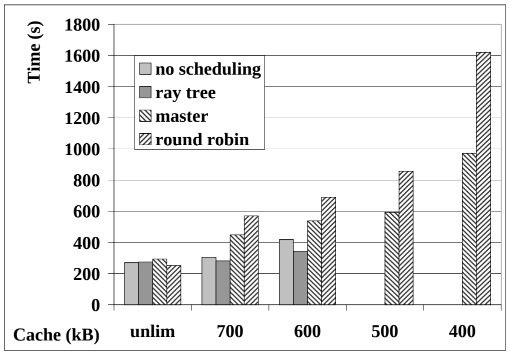


Figure 12: Rendering time (in seconds) on 8 processors for the studio model at 128^2 pixels for different object cache sizes and scheduling techniques.

For cases where only part of the scene can be cached, the way demand driven shadow tasks are scheduled, is important, as can be seen in figure 12. Here, the studio model is rendered on eight processors using a reduced problem size (128^2 pixels). The three scheduling methods discussed in section 5 are compared with each other and with a simple round robin scheme for the distribution of demand driven shadow tasks. If the cache size is not artificially clamped to a fixed value, seven processors need just over 1 MB to store non-local objects each plus 200 kB for their resident sets. The uneven initial object distribution for this scene awards one processor 1.25 MB initially, so that virtually no objects are fetched for this node. This processor becomes the bottleneck in the experiments discussed below.

As the cache size is reduced, the load balancing capability is maintained best by the algorithm which reroutes all shadow tasks via the master. Round robin scheduling also seems to distribute the workload, although far less successful. This is because this technique does not distinguish between heavily loaded processors and less busy ones. There seems to be no significant difference between scheduling shadow tasks with the processor that spawned the ray tree and using no scheduling at all. Both break down to the extent that a processing bottleneck is created with one processor, causing input task queues to overflow when the object cache size is made sufficiently small. This problem becomes worse when the workload is increased by rendering larger images.

When tasks are scheduled through the master, the total rendering time can be broken down into its component parts, as shown in figure 13. The computation time is almost completely dominated by idle time, which is equally shared by all but one processor. The idle time is also spread evenly over time, showing that one processor becomes the bottleneck which dominates the computation;

behaviour normally associated with data parallel approaches. This is no surprise, as the cache size becomes too small to complete most demand driven shadow tasks. Hence, the data parallel component gains in significance, without the demand driven part being capable to combat its associated load imbalance. Injecting more work into the system would not solve this problem, because this would result in more rays being transferred to the overloaded processor as data parallel ray tasks.

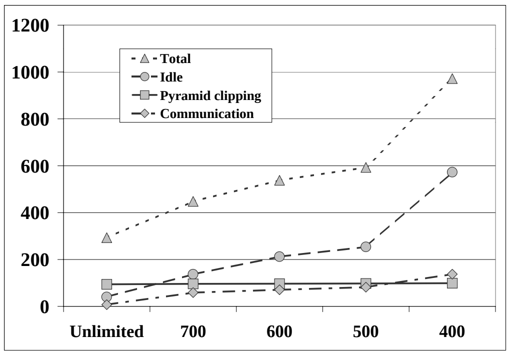


Figure 13: Rendering time (in seconds) on 8 processors for different aspects of the algorithm (studio model at 128^2 pixels). Shadow tasks are rerouted through the master. “Total” represents the total rendering time, while the other times are average times per processor.

In summary, the one scheduling method which uses global information (by rerouting all shadow tasks through the master processor, which distributes these tasks on demand) maintains an even workload best when the cache size is reduced. This is also true for larger problem sizes (we tested this scheduling technique for 512^2 images and found similar results). Unfortunately, this method is inherently not scalable with the number of processors. These results indicate that scheduling techniques need to be developed which incorporate global information regarding the load.

Finally, our hybrid scheduling algorithm sits inbetween demand driven and data parallel execution modes, both in concept and behaviour. Therefore, the cache chosen should therefore not be chosen too small, as this would kill the efficiency. On the other hand, choosing the cache size too big would turn this into a demand driven algorithm, for which pure implementations have proved to give best performance. Hence, the size of the scenes that can be rendered with some degree of efficiency should be larger than with demand driven algorithms, whereas increasing the scene size further will result in performance comparable to data parallel approaches.

8 MEMORY

Besides storage space used for caching objects, memory is required for temporary storage of intermediary results and task queues etc., requiring an additional memory overhead. This extra memory consumption should be less than the memory saved by distributing the scene.

In our implementation there are a number of areas which take up storage space, which include: memory occupied by the octree and a subset of the objects; storage of intersection points for which shading results are pending; and memory taken by storing a task queue for demand driven tasks that are waiting for object data to arrive. Depending on the workload balance, the buffers used by PVM to hold incoming messages take up a significant amount of memory as well. If we examine the case of the studio model for a rendering performed on 16 processors using an object cache which is large enough to fit all data that is requested, its memory breakdown is presented in table 2.

Table 1 shows that the number of references from the octree to an object is significantly higher than the total number of objects in the scenes. This is a common, but often overlooked feature of octrees. However, it does have an impact on the storage requirements of the octree itself.

Memory type	Maximum
Intersections	266 kB
Shadow tasks	207 kB
Pyramid clipping	14 kB
Primary ray tasks	96 kB
Cached objects	∼1 MB
Objects	268 kB
Octree	922 kB

Table 2: Memory usage for processor 1 (of 16). These figures are typical for all processors during these renderings.

Intermediary results require a modest amount of storage space. These results are stored while shading results are pending, which may be longer if the data parallel part is more prominent. For 512^2 pixel renderings of the studio model on 8 processors, the maximum amount of memory used during the computation (including all dynamically allocated memory, but excluding memory used for input buffers and memory allocated during start-up) ranges from 1.3 MB for an unlimited object cache size to 982 kB for a cache of 400 kB.

Dependent on the significance of the data parallel component, for example in the presence of diffuse inter-reflection or when many shadow rays need to be migrated due to a small object cache, a considerable amount of memory is taken by PVM’s input buffers. This may easily cause one processor to run out of memory. As measuring these buffers influences the performance of the algorithm considerably, separate measurements were taken (on 8 SGI O2’s in this case). The processor constituting a bottleneck, has a maximum input buffer size which ranges from 617 kB for an unlimited object cache to 16 MB for a 600 kB cache. Clearly, the total amount of work in the system at any one time needs to be controlled more rigorously than is currently the case. Therefore, scheduling needs to be based on global information to detect if and when processors become bottlenecks. Alternatively, the number of data fetches could be increased to reduce the influence of the data parallel component.

9 CONCLUSIONS

The hybrid scheduling algorithm presented in this paper scales well with the number of processors, although a fairly substantial amount of overhead is recorded. This stems from the fact that most of this overhead is not processor dependent, but remains relatively constant regardless of the number of processors used. Due to the required data selection, which is performed for each primary and shadow ray task, most of the computational overhead is scene and parameter dependent. In cases where the cost of data selection (pyramid clipping) is greater than the performance benefit, which occurs when too few shadow rays constitute a shadow task, it may be better to schedule these rays as data parallel tasks. We hope to investigate this aspect in the near future.

Idle time is mostly incurred when waiting for object data to be sent from other processors. This can be attributed to the long set-up times measured for global communication when using PowerPVM on the Parsytec PowerXplorer [9]. It would be possible to prepare more tasks while data is pending, but this would require extra memory for storing intermediary results. Slower networks tend to require more memory per node and if that is not available, then some idle time needs to be accepted. Our application is no exception.

Algorithms which distribute scene data are targeted at rendering large to very large scenes. The smaller the subset of the scene that can be cached with each processor, the more our algorithm starts to behave like a pure data parallel approach. It appears that the size of the object cache should be at least half the scene size, restricting the maximum size for which our algorithm retains some efficiency. However, this is still better than can be achieved with pure data parallel approaches. If, on the other hand, sufficient cache memory is available, our algorithm resembles a proper demand driven solution.

Also, our current implementation has some weaknesses concerning memory overhead. The most important one is the cost of storing input buffers for reduced cache sizes. Improving efficient memory usage is currently being investigated. Another area for improvement is in finding suitable initial data distributions, so that the workload for the data parallel component is more even. This would reduce the strain on the demand driven component, especially if the the number of objects that can be cached for demand driven tasks is limited. However, finding a data distribution that equalises the workload is a non-trivial task which is currently unsolved.

Finally, the results presented in this paper are partially comparable to our previous results which were based on a parallel implementation of Rayshade [16]. Then, the efficiency was very high until some point fairly early in the computation after which it degradated to the level expected with pure data parallel implementations. This was attributed to the fact that only primary rays were scheduled demand driven. Currently, shadow tasks are handled in demand driven fashion as well, which is why the efficiency remains constant throughout the computation.

Acknowledgements

This work was sponsored by the European Commission under TMR grant number ERBFMBICT960655 and by the Stichting Nationale Computerfaciliteiten (National Computing Facilities, NSF) for the use of supercomputer facilities, with financial support from the Nederlandse Organisatie voor Wetenschappelijk Onderzoek (Netherlands Organisation for Scientific Research, NWO).

References

[1] D. Badouel, K. Bouatouch, and T. Priol. Distributing data and control for ray tracing in parallel. *IEEE Computer Graphics and Applications*, 14(4):69–77, 1994.

[2] A. Chalmers and E. Reinhard. *Parallel and Distributed Photo-Realistic Rendering*. ACM Siggraph, July 1998. Course notes for SIGGRAPH 98.

[3] J. G. Cleary, B. M. Wyvill, G. M. Birtwistle, and R. Vatti. Multiprocessor ray tracing. *Computer Graphics Forum*, 5(1):3–12, March 1986.

[4] T. W. Crockett. An introduction to parallel rendering. *Parallel Computing*, 23(7):819–843, July 1997. Special Issue on Parallel Graphics and Visualisation.

[5] F. C. Crow, G. Demos, J. Hardy, J. McLaugglin, and K. Sims. 3d image synthesis on the connection machine. In *Proceedings Parallel Processing for Computer Vision and Display*, Leeds, 1988.

[6] M. A. Z. Dippé and J. Swensen. An adaptive subdivision algorithm and parallel architecture for realistic image synthesis. In H. Christiansen, editor, *Computer Graphics (SIGGRAPH '84 Proceedings)*, pages 149–158, July 1984.

[7] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM 3 User's Guide and Reference Manual*. Oak Ridge National Laboratory, Oak Ridge, Tennessee, May 1993. Included with the PVM 3 distribution.

[8] S. A. Green and D. J. Paddon. Exploiting coherence for multiprocessor ray tracing. *IEEE Computer Graphics and Applications*, 9(6):12–26, November 1989.

[9] A. G. Hoekstra, P. M. A. Sloot, F. van der Linden, M. van Muiswinkel, and J. J. J. Vesseur. Native and generic parallel programming environments on a transputer and a PowerPC platform. *Concurrency: Practice and Experience*, 8(1):19–46, January-February 1996.

[10] F. W. Jansen and A. Chalmers. Realism in real time? In M. F. Cohen, C. Puech, and F. Sillion, editors, *4th EG Workshop on Rendering*, pages 27–46. Eurographics, June 1993. held in Paris, France, 14–16 June 1993.

[11] 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):197–209, 1988.

[12] W. Lefer. An efficient parallel ray tracing scheme for distributed memory parallel computers. In *1993 Parallel Rendering Symposium*, pages 77–80. ACM-SIGGRAPH, 1993.

[13] T. T. Y. Lin and M. Slater. Stochastic ray tracing using SIMD processor arrays. *The Visual Computer*, 7(4):187–199, 1991.

[14] E. Reinhard and A. Chalmers. Message handling in parallel radiance. In M. Bubak, J. Dongarra, and J. Waśniewski, editors, *Proceedings EuroPVM-MPI'97*, pages 486–493. Springer - Verlag, November 1997.

[15] E. Reinhard, A. Chalmers, and F. W. Jansen. Overview of parallel photo-realistic graphics. In *Eurographics STAR – State of the Art Report*, pages 1–25, August-September 1998.

[16] E. Reinhard and F. W. Jansen. Rendering large scenes using parallel ray tracing. *Parallel Computing*, 23(7):873–886, July 1997. Special issue on Parallel Graphics and Visualisation.

[17] E. Reinhard, A. J. F. Kok, and A. G. Chalmers. Cost distribution prediction for parallel ray tracing. In K. Bouatouch, A. Chalmers, and T. Priol, editors, *Proceedings of the Second International Workshop on Parallel Graphics and Visualisation*, pages 77–90, September 1998.

[18] I. D. Scherson and C. Caspary. Multiprocessing for ray tracing: A hierarchical self-balancing approach. *The Visual Computer*, 4(4):188–196, 1988.

[19] G. J. Ward. Adaptive shadow testing for ray tracing. In *Photorealistic Rendering in Computer Graphics (Proceedings of the Second Eurographics Workshop on Rendering)*, pages 11–20, New York, 1994. Springer-Verlag.

[20] G. J. Ward. The RADIANCE lighting simulation and rendering system. In A. Glassner, editor, *Proceedings of SIGGRAPH '94*, pages 459–472, July 1994.

[21] G. J. Ward, F. M. Rubinstein, and R. D. Clear. A ray tracing solution for diffuse interreflection. *ACM Computer Graphics*, 22(4):85–92, August 1988.

[22] M. van der Zwaan, E. Reinhard, and F. W. Jansen. Pyramid clipping for efficient ray traversal. In P. Hanrahan and W. Purgathofer, editors, *Rendering Techniques '95*, pages 1–10. Springer - Vienna, June 1995.

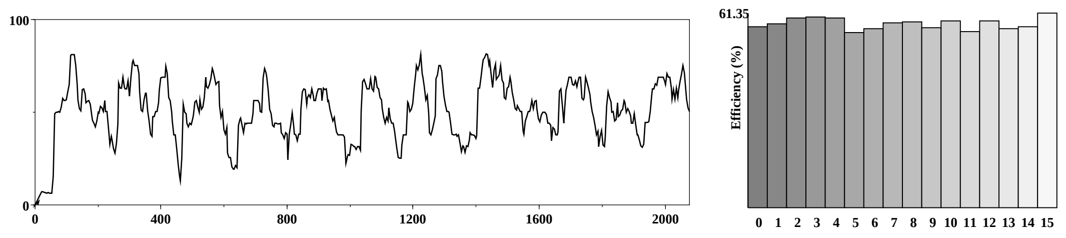


Figure 6: Efficiency for 16 processors (studio model). The left graph shows the efficiency over time, while the right graph gives the efficiency per processor (both graphs in %)

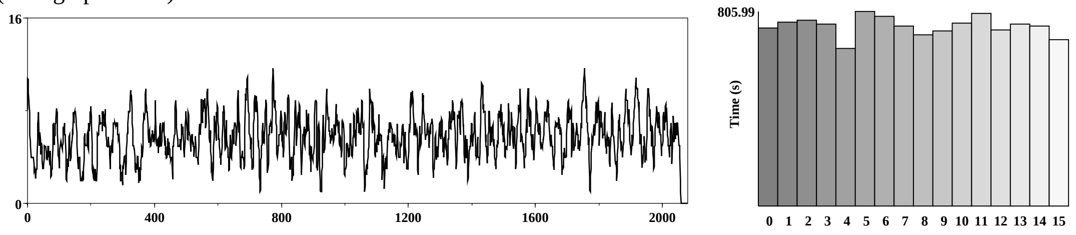


Figure 7: Number of processors performing pyramid clipping overhead (left) and amount of time spent pyramid clipping per processor (right)

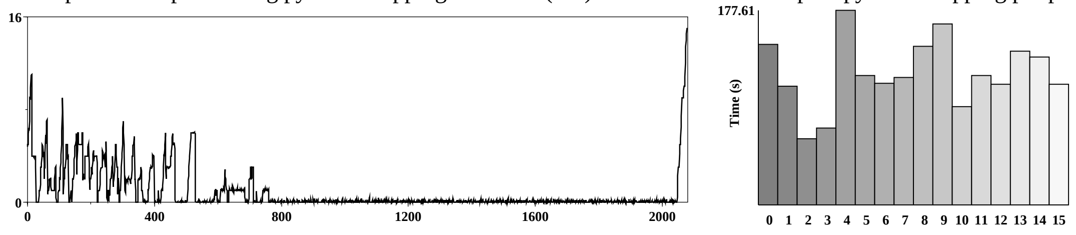


Figure 8: Number of processors idle per time unit (left) and idle time per processor (right).

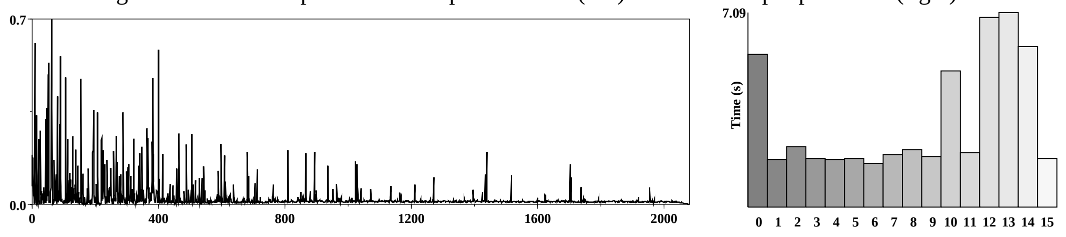


Figure 9: Number of processors busy communicating per time step (left) and communication time per processor (right). These graphs include both task and data communication. The left graph is scaled to fit the height of the image

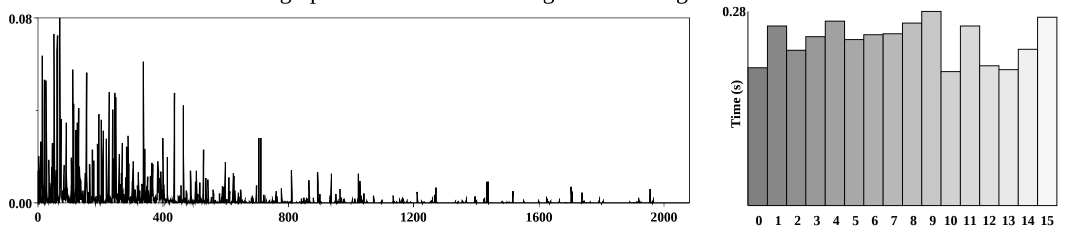


Figure 10: Number of processors fetching object data per time unit (left) and time spent fetching data per processor (right). Note that the graph on the left is scaled to fit maximum height.

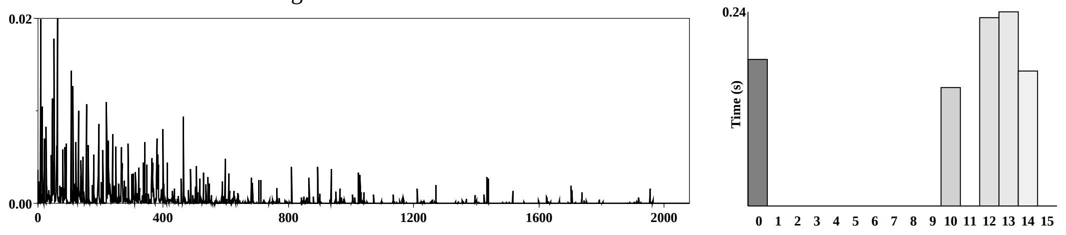


Figure 11: Number of processors sending object data per time unit (right) and time spent sending data per processor (right). The left graph is scaled to fit the height of the image.