Workload Distribution for Ray Tracing in Multi-Core Systems

Miguel Nunes
Dep. de Informática, Universidade do Minho
Campus de Gualtar, 4710-057 Braga
mdccnunes@gmail.com

Luís Paulo Santos
Dep. de Informática, Universidade do Minho
Campus de Gualtar, 4710-057 Braga
psantos@di.uminho.pt

Abstract

One of the features that made interactive ray tracing possible over the last few years was the careful exploitation of the computational power and parallelism available on modern multicore processors. Multithreaded interactive ray tracing engines have to share the workload (rays to be processed) among rendering threads. This may be achieved by storing tasks on a shared FIFO-queue, accessed by all threads. Accessing this shared data structure requires a data access control mechanism, which ensures that the data structure is not corrupted. This access mechanism must incur minimal overheads such that performance is not penalized. This paper proposes a lock-free data access control mechanism to such queue, which avoids all locks by carefully reordering instructions. This technique is compared with a classical lock-based approach and with a conservative local technique, where each thread maintains its local queue of tasks and shares nothing with other threads. Although the local approach outperforms the other two due to very good load balancing conditions, we demonstrate that the lock-free approach outperforms the lock-based one for large processor counts. Efficient and reliable sharing of data structures within a shared memory system is becoming a very relevant problem with the advent of many core processors. Lock free approaches are a promising manner of achieving such goal.

Keywords

Interactive Ray Tracing, Workload Distribution, Parallel Graphics

1. INTRODUCTION

Ray tracing based renderers are able to simulate global illumination and physically based light transport effects, resulting in high fidelity images that can be used in a predictive manner [Greenberg 99]. Ray tracing is, however, a very time consuming algorithm, traditionally seen as suitable for offline rendering only. Over the last decade many improvements in the field of ray tracing and in computing hardware have made it possible to interactively render many of the global effects, such as specular phenomena, direct illumination and even indirect diffuse interreflections [Wald 07b, Debattista 09]. Interactive ray tracing has been achieved mainly by resorting to multi-level parallelism, by using clever sampling strategies and by carefully exploiting memory locality and ray coherence [Wald 07a].

Within these interactive rendering systems parallelism is usually exploited along at least three levels: the vectorial capabilities of modern multiprocessors (SIMD instructions), symmetric multiprocessing by resorting to multithreading in multi-core systems and parallel execution among multiple nodes of distributed memory clusters. An additional, seldom exploited, level is the simultaneous utilization of GPUs to share the workload with the CPUs. The exploitation of a multiplicity of computing resources, with the goal of increasing performance, requires partitioning and distributing the workload among these resources, while simultaneously assuring that overheads are kept to a minimum. Overheads include communication/synchronisation costs, resources idle times due to sub-optimal load partitioning (the load balancing problem) and redundant computations due to work replication.

This paper focuses on the multithreading/multicore level of parallelism exploitation. Within this level all processing cores see a single shared address space. Workload distribution is often performed by maintaining some shared data structure from where individual threads can retrieve tasks [Bigler 06]. Within an interactive ray tracing context a task, corresponding to processing a set of rays, can originate new tasks, which correspond to secondary rays shot at deeper levels of the ray tree. The typical recursive, depth-first, ray tracing algorithm is thus substituted by an iterative, breadth-first, approach by storing newly generated tasks on this data structure, rather than using the execution stack. Since all rendering threads can read and write on the shared data structure, a data access control mechanism is required to ensure that it is not corrupted. These data access control mechanisms incur additional costs, such as serialisation of data accesses, which might compromise performance on a multithreading processing environment. Careful design and evaluation of these mechanisms is thus required in order to maintain acceptable efficiency levels.

In this paper we discuss and evaluate three different such mechanisms, which are used to control access to a shared FIFO-queue holding tasks (sets of rays) to be processed by the rendering threads. Access control to shared memory is typically performed by resorting to mutual exclusion, using locks to guarantee that only one thread executes a critical section at each time. Lock-free synchronisation techniques can avoid all locks by carefully reordering instructions, drastically reducing contention and context switching costs [Herlihy 08]. In this paper we compare one such lock-free access control mechanism with a traditional lock-based approach and a conservative local technique, where each thread maintains a local work queue, thus preventing any type of work sharing while simultaneously dispensing with access control.

The relevance of these alternative, not so well-known, data access control mechanisms is becoming fundamental with the ever increasing processor count in multicore systems. The major contributions of this paper are the proposal of a lock-free data access control mechanism for distributing tasks within a multithreaded interactive ray tracer and a comparison of the respective performance with more traditional approaches.

2. RELATED WORK

2.1. Interactive Ray Tracing

At the core of any renderer lies a visibility algorithm that evaluates whether two points in space are mutually visible, i.e., if there is any entity occluding the shortest path between them. The two most well known techniques for visibility evaluation are rasterization-based approaches and ray tracing based approaches. The former are supported very efficiently on special purpose hardware (GPUs), but are usually limited to process sets of rays sharing a single origin. The latter, on the other hand, support arbitrary visibility queries, thus allegedly more suitable to compute global illumination effects. Ray tracing, however, has traditionally exhibited larger latencies than rasterization, which has precluded its utilization within interactive contexts. With advances in processing hardware combined with improvements in algorithms and sampling strategies, ray tracing has reached a stage where its utilization within interactive settings is now possible.

Ray tracing performance has been increased by resorting to multi-level parallelism, by using clever sampling strategies and by carefully exploiting memory locality and ray coherence [Wald 07a]. Multi-level parallelism includes exploiting modern CPUs vectorial capabilities, the multiplicity of cores available within a single machine and multiple machines interconnected by some suitable network. Sampling was improved by using more effective sampling patterns [Kollig 02] and by resorting to bidirectional particle-based algorithms [Keller 97], thus reducing the number of rays required for the same perceived image quality.

Ray coherence refers to the fact that neighboring rays tend to traverse the same regions of space and intersect the same geometric primitives. Packeting these coherent rays together and processing them in packets allows for increased memory coherence and enables effective utilisation of modern CPUs vector processing capabilities. By combining these techniques with careful code optimization – to adapt the algorithms to the properties of the underlying hardware – the ray tracing algorithm runs almost completely in cache, reducing expensive accesses to main memory [Wald 07a]. This approach has been thoroughly exploited over the last years, resulting in many alternative packet-based approaches [Wald 02b, Reshetov 05, Wald 08]

Descriptions of interactive ray tracers' architectures do not abound in the specialized literature. Bigler at al. [Bigler 06] argue that the shift towards multi-core systems requires that high performance programs address thread-level and instruction stream (SIMD) parallelism explicitly. They present the Manta interactive ray tracer, based on an multi-threaded scalable engine and a collection of data structures, based on wide ray packets, for coordinating these threads' work. They do not evaluate different data structures used to share work among the rendering threads. In [Wald 02a] the authors present a list of requirements an interactive ray tracer must comply to, together with results showing that their ray tracer - OpenRT - complies to these requirements, but they do not discuss lower level implementations details.

In this paper we present the generic structure of our interactive ray tracer - iRT - focusing on the data structure used to share work among threads.

2.2. Synchronization

Access control to shared data structures is usually performed by resorting to lock-based (or blocking) mechanisms, which ensure mutual exclusion within critical sections of the code. Access to shared data structures is serialised, resulting in high performance penalties when contention is significant. Since contention increases with the level of concurrency, typically lock-based approaches perform worst as the number of threads increases. Furthermore, locking often requires expensive context switches, which might be intolerable within interactive applications.

Alternatively, one can use lock-free synchronisation methods, which rely on atomic conditional primitives to control access to shared data structures [Debattista 03, Herlihy 08] (listing 1 presents the functionality of Compare-and-Swap, a well known atomic operation used throughout this paper). These algorithms may either be non-blocking or wait free. Non blocking algorithms terminate in finite time, but are based on retries. Wait-free algorithms are guaranteed to finish on a finite number of steps, thus being immune to the priority inversion problem and eliminating deadlock and starvation.

Wait-free and lock-free access control mechanisms are seldom used within the graphics community, in spite of their increasing relevance due to the ever increasing number of cores on modern processors. In [Dubla 09] the authors present a wait free mechanism to share the irradiance cache

```
atomic CAS(addr location, val cmpVal, val newVal)
{
    if (*location == cmpVal)
    {
        *location = newVal;
        return true;
    }
    else return false;
}
```

Listing 1. Compare and swap

among multiple cores. They compare the achieved performance with those achieved with a lock-based approach and a local approach, where threads do not share locally computed irradiance values.

In this paper we present and evaluate a lock-free approach to access a shared queue holding tasks for the rendering threads, and compare its performance with those achieved with a lock-based approach and a local approach, where threads do not share locally generated tasks.

3. iRT ARCHITECTURE

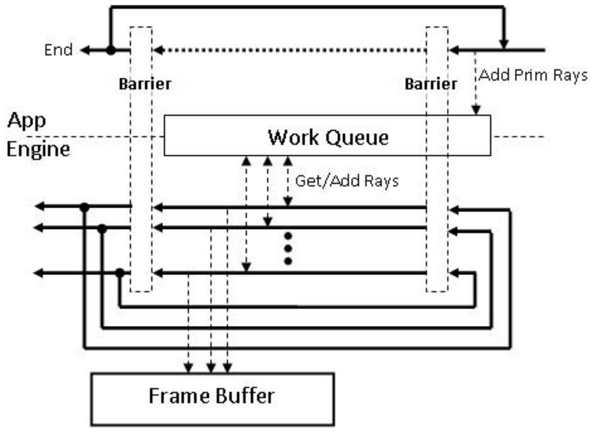


Figure 1. iRT organization

Our interactive ray tracing engine - iRT - runs a set of symmetric rendering threads, which get their tasks from a global, shared work queue (see figure 1). The threads are symmetric in the sense that they all execute the same algorithm, which consists on retrieving tasks from the global queue and, for each ray in the task, traverse the 3D space (using a Bounding Volume Hierarchy), intersect the ray with candidate triangles and then shade the intersection point. Shading may result in shooting additional rays, which is achieved by adding new tasks to the work queue. Shading, specially in the case of shadow rays, may result in contributions to the frame buffer, which are concurrently added by each thread. Transforming the traditional recursive ray tracing algorithm into this iterative one, requires that each ray carries with it information about which pixel it contributes to (this is equivalent to the ID of the parent primary ray) and also a weight factor that is equal to the product of the cosines and BRDFs at all intersection points along the current path. When the queue is empty this means that the current frame has been rendered; each thread will then join a barrier and wait for further work or a terminate tag that closes the rendering engine (see listing 2).

```
1  thread_RenderLoop(WorkQueue wl) {
2      while (!END) {
3          barrier(); // wait for a new frame to start
4          while (wl.getRays (&task) != EMPTY) {
5              for each ray in task {
6                  TraverseBVH ();
7                  Intersect ();
8                  Shade (); // may add new rays to queue
9                  FrameBuffer.Update ();
10             }
11         }
12         barrier(); // wait for all threads to finish
13     }
14 }
```

Listing 2. iRT - rendering threads main loop

iRT has to be linked with an application program that implements the application logic. Besides supporting all the application functionality, this application program is responsible for initializing iRT, loading initial tasks (eventually corresponding to primary rays) into the work queue and then release the iRT rendering threads. This last step is achieved by joining the barrier where the rendering threads are waiting (line 3 of listing 2 and line 7 of listing 3). The application program must then wait for the rendering threads to finish, which is achieved by joining the second barrier (line 12 and 9 of listings 2 and 3, respectively). This main loop is repeated until the application is terminated, in which case the END flag is raised, causing the rendering threads to finish.

```
1  main() {
2      iRT_init();
3      application_init ();
4      while (!finished) {
5          application_logic1 ();
6          Generate_PrimRays (); //write into the work queue
7          barrier (); // release render threads
8          application_logic2 ();
9          barrier (); // wait for frame to finish
10         FrameBuffer.Output ();
11     }
12     set END flag to finish threads
13     barrier(); // release threads and finish
14 }
```

Listing 3. iRT - application loop

Using this architecture rendering, or ray tracing, is decoupled from the application logic. Although the ray tracing engine was developed essentially for image rendering, it can easily be used for different goals, such as collision detection. It is the application who decides which rays are shot and how are the respective results used. The manner how each ray's contribution is evaluated depends on the particular shader being used (line 8 of listing 2); different iRT engines can be built using different shading functions. The shader could actually be a dynamically loadable component - dispensing with building different engines for different shaders - but we decided to keep it statically linked in order not to hurt performance. Also note that the particular implementation of the WorkQueue is hidden behind the respective class interface. We will be using different work queues without changing the thread RenderLoop code.

The above described software architecture implies that two data structures are shared among the application and all rendering threads: the work queue and the frame buffer. Accessing the former is discussed in the following sections. The latter, which is where the rendering results are accumulated by the renderers and read back by the application thread, is protected by a user space spinlock [Herlihy 08], thus reducing context switches among threads. Since the results in the frame buffer are a linear combination of several rays' contributions, access to it could be wait-free among the rendering threads by resorting to hardware-supplied atomic floating-point add instructions.

iRT is a preliminary prototype and most of the development effort focused on the mechanisms used to share tasks among threads, whose results are reported in this paper. Some other very important issues (performance-wise) have been handled in a much straight forward manner, which results in some performance penalties. These issues include vectorisation, coherent rays packeting and locality of memory accesses [Reshetov 05, Wald 07b]. All these three are intimately related and are determinant with respect to the ray tracer final performance; these will be addressed in the near future.

4. ALGORITHMS

In this section the three different data access control mechanisms are presented. The lock-based one is referred to as LOCK, the local one as LOCAL and the lock-free as LFREE. In order to reduce workload distribution time overhead each element of these queues consists on a set of 20 rays, rather than a single one, thus increasing task granularity.

4.1. Lock-Based Queue

The traditional LOCK implementation uses a single lock to protect all accesses to the shared queue. This results in serialising all calls to getRays() and addRays(), that is, reads and writes do interfere among them. However, reads and writes operate on different ends of the queue, so they must be able to proceed without interference if the queue is neither full nor empty. We use the Unbounded Total queue algorithm described in [Herlihy 08]: the queue is a linked list of tasks and different locks are used for reads and writes, thus reducing contention. A sentinel node, whose next field is NULL, is initially inserted. Readers always check whether this is the head node of the queue; if it is, then the queue is empty (see listing 4).

```
addRays (RayType *ray) {
    addlock.lock();
    QueueNode *node = new QueueNode (ray);
    tail->next = node;
    tail = node;
    addlock.unlock();
}

getRays (rayType **ray) {
    getlock.lock();
    if (head->next==NULL) {
        getlock.unlock();
        return EMPTY;
    }
    QueueNode *actual = head;
    head = head->next;
    getlock.unlock();
    *ray = actual->value;
    delete actual;
    return OK;
}
```

Listing 4. Lock-based queue

4.2. Local Queue

The reasoning behind the LOCAL approach is that each thread maintains its own work queue. The application program rather than writing all primary rays to a single queue, statically distributes the tasks among all work queues in a round-robin fashion to ensure better load balancing. Each rendering thread then processes its initially assigned rays and adds secondary rays to its own queue. This approach has no sharing, thus it requires no data access control mechanism; the work queues implementation is similar to that shown in listing 4, but without the locks. Since there is no contention or serialisation of accesses this approach has the potential to outperform the other two if a balanced load distribution is guaranteed. Note that load distribution is statically done by the application program; the round robin distribution of work and the fine granularity of tasks (i.e., the number of rays associated with each of the queue's node) assure a reasonable load distribution for many images and for shallow ray trees, such as those typically found in interactive ray tracing contexts.

4.3. Lock-Free Queue

The lock-free algorithm [Herlihy 08, Michael 96] does not rely on locks to guarantee mutual exclusion. It relies on the Compare-and-Swap atomic synchronisation primitive described in listing 1 and on retries, i.e., it contains a loop (this is the reason why it is not a wait-free method: it is not guaranteed to finish in a finite number of steps). Additionally, the addRays() method is lazy, meaning that insertion of new nodes happens in two different steps; in particular, threads may need to help one another in order to advance tail (see listing 5).

```
addRays (RayType *ray) {
    QueueNode *node = new QueueNode (ray);
    while (true) {
        QueueNode *last = tail;
        QueueNode *next = last->next;
        if (last==tail) {
```

```
 7            if (next==NULL) {
 8                if (CAS(last−>next, next, node)) {
 9                    CAS (tail, last, node);
10                    return ;
11                }
12            } else {
13                CAS (tail, last, next);
14            }
15        }
16    }
17 }
18
19 getRays (rayType **ray) {
20    while (true) {
21        QueueNode *first = head;
22        QueueNode *last = tail;
23        QueueNode *next = first−>next;
24        if (first==head) {
25            if (first==last) {
26                if (next==NULL) {
27                    return EMPTY;
28                }
29                CAS (tail, last, next);
30            } else {
31                *ray = next−>value;
32                if (CAS (head, first, next)) {
33                    delete first;
34                    return OK;
35                }
36            }
37        }
38    }
39 }
```

Listing 5. Lock-free queue

The addRays() method creates a new node (line 2), reads tail and finds the node that appears to be last (lines 4 and 5). It then checks whether that node is still last (line 6) and whether the node has a successor (line 7). If the node does have a successor then it was inserted by other thread; this thread will help the others by trying to advance tail to the next node, but only if tail is still equal to last (line 13) - it will then try again to insert the new node. If, however, the node still does not have a successor (lines 7 – 12), then it performs a trial to append it to the queue (line 8). If it succeeds, it tries to update tail to the new node (line 9); this CAS operation may fail, but this does not represent a problem, since it will only fail if tail has already been advanced by other thread. If, however, appending the node to the queue failed (line 8), then the thread will try again (this CAS operation may fail because some other thread might have appended other node).

The getRays() method is very similar to its lock-based homonym. It will check if the queue is empty by verifying if the successor of head is NULL (line 26); if the queue is non empty, then it will try to advance head to its successor and return the previous head node (lines 31 – 35). There is however a subtlety in this lock-free algorithm: before advancing head the algorithm has to make sure that tail is not left referring to the sentinel node that is about to be removed from the queue (this may happen because some thread added a node to the queue but was not able to update tail). Thus if head equals tail (line 25) but the head successor is not NULL (line 26), then the queue can not be empty; the thread will try to advance tail to the sentinel's node successor (line 29) and will iterate again.

5. RESULTS

5.1. Experimental Setup

In order to assess the effectiveness of the three different data access control strategies (LOCK, LFREE and LOCAL) experiments were performed with three different scenes: the conference room (190951 triangles), the office scene (20769 triangles) and the Stanford bunny (69463 triangles) (see figure 2). All scenes include 4 point light sources. The illumination model used shoots one shadow ray per light source and one specular reflection ray per intersection point (if the material has a specular reflection coefficient larger than 0). Images were rendered at a resolution of 300x300 pixels, with one primary ray per pixel. All images were rendered with the above described prototype of iRT, which does not use nor vectorial (SIMD) instructions, neither ray packeting.

Experiments were run in three different multicore systems, in order to assess the proposed techniques efficiency and scalability under different conditions:

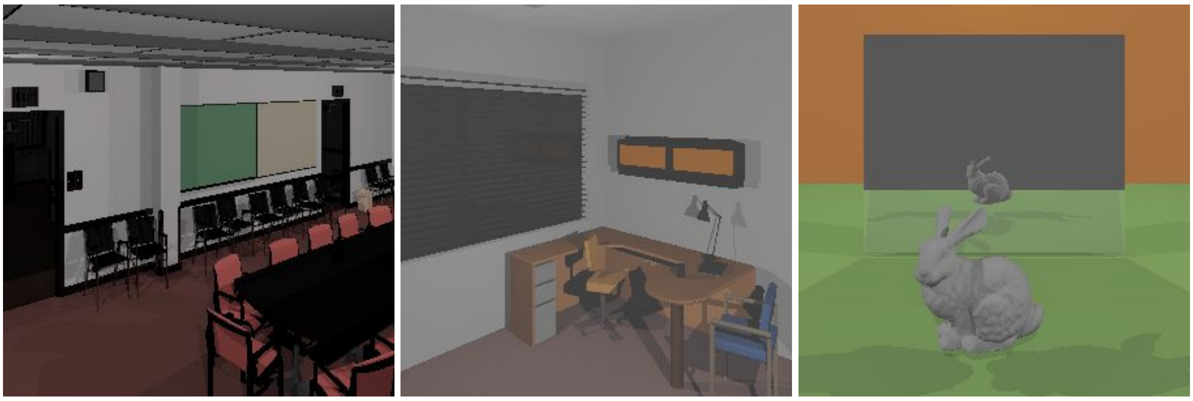
- an 8-core server based on two quad-core Intel Xeon 5420 processors, running at 2.50 GHz, with 8 GB RAM;
- an 8-core server based on two quad-core Intel Xeon 5520 processors (Nehalem architecture), running at 2.26 GHz, with 12 GB RAM; these include support for HyperThreading, thus reporting 16 processors to the operating system;
- a 24-core server based on four hexa-core Intel Xeon 7450 processors (Dunnington architecture), running at 2.40 GHz, with 64 GB RAM.

All results are the average over all frames of three executions of the ray tracer, where each execution renders 200 frames. The goal is to eliminate interferences from other processes activity (such as the operating system) and cache warm-up.

5.2. Analysis

Figures 3, 4 and 5 present all the results in graphical form. The left axis of each figure depicts frames per second and is associated with the line graphs. The right axis depicts speed-up, computed relatively to the sequential version, and is associated with the bar graphs. The horizontal axis represents the number of threads for the corresponding data.

All graphs show that the LOCAL version outperforms the other two. It presents a larger frame rate under all evaluated settings and scales better with the number of cores, thus

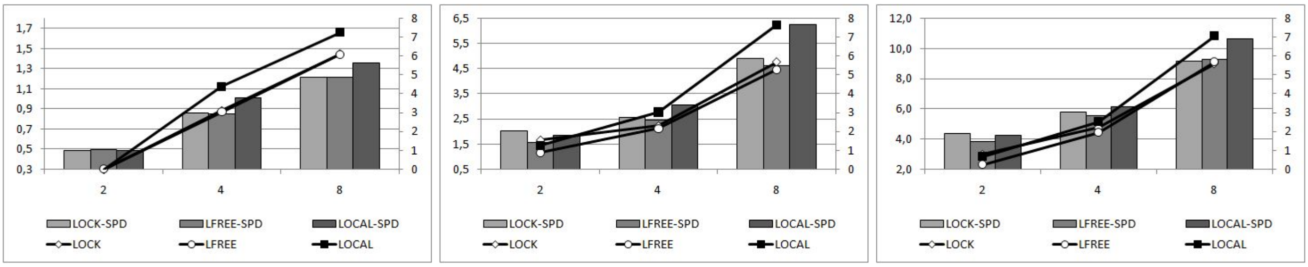


(a) The conference room

(b) The office scene

(c) The Stanford bunny

Figure 2. The scenes used for the experiments

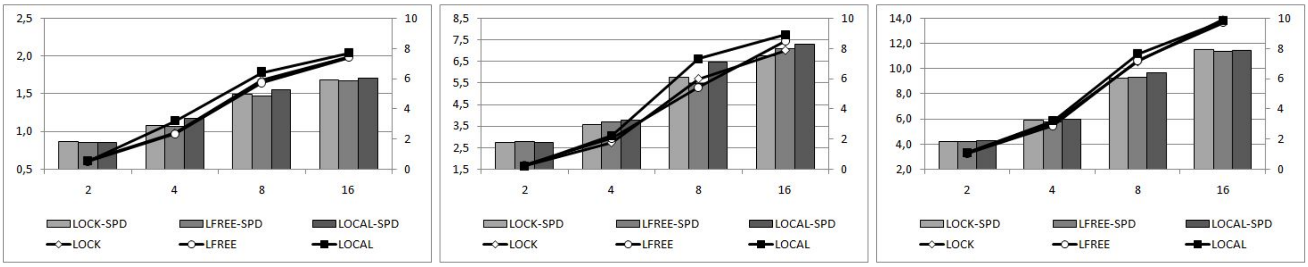


(a) Conference

(b) Office

(c) Bunny

Figure 3. Results for the 8-core Xeon 5400 server. Axis: left- fps, right- speed-up, horizontal- number of threads.

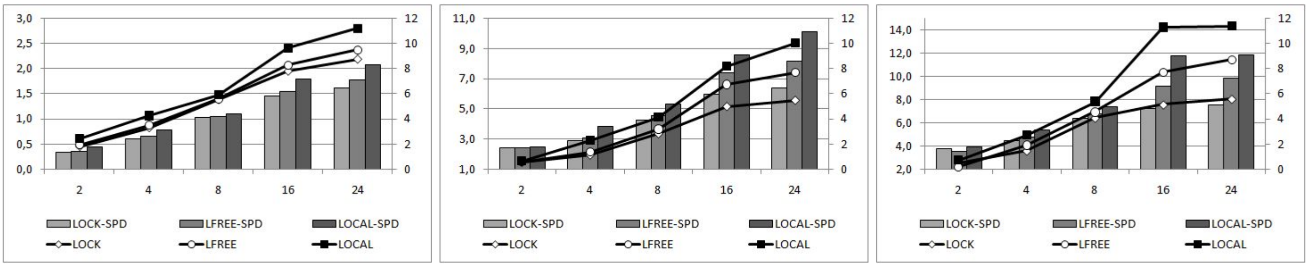


(a) Conference

(b) Office

(c) Bunny

Figure 4. Results for the 8-core (plus HT) Xeon 5500 server. Axis: left- fps, right- speed-up, horizontal- number of threads.



(a) Conference

(b) Office

(c) Bunny

Figure 5. Results for the 24-core Xeon 7400 server. Axis: left- fps, right- speed-up, horizontal- number of threads.

presenting the larger speedup. This version incurs no data access control overheads, but may suffer from load balancing problems: image plane blocks are assigned to each thread in a round-robin fashion, and each thread must process all rays (primary and secondary) associated with those blocks. However, due to the fine granularity of the image blocks (20 pixels per block) and since these are distributed cyclically among threads, in practice the load imbalance is negligible, and LOCAL achieves the best results. Figure 6 presents the average time, in seconds, spent idle per thread and per frame waiting for other threads to finish their assigned image space blocks (results for 24 threads on the 24-core server). Although the time spent waiting for other threads due to load imbalances is larger for the LOCAL approach, this difference is small and can not be responsible for the corresponding differences in frame rate. In fact, figure 7 shows that, for 24 threads, the time spent accessing data in the work queues is completely negligible for the LOCAL approach, when compared with LFREE and specially with LOCK. Load balance can however be compromised, for the LOCAL approach, if the image is very heterogeneous with respect to the ray tree depth per pixel. In this case some threads might be busy finishing their workload while other threads have already finished.

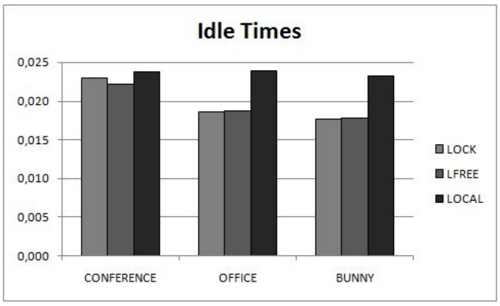


Figure 6. Average idle times, in seconds, per frame for 24 threads on the 24-core Xeon 7400 server. (Conference scene values are divided by 10)

A direct comparison between the LOCK and LFREE approaches shows that their behaviour is very similar up to 8 threads, with LOCK outperforming LFREE in some cases. However, when the number of cores, and associated threads, increases above 8, such as with the 24-core server, then LFREE starts showing its true potential. With 24 cores LFREE clearly outperforms LOCK, and its derivative is larger than that of LOCK for all three scenes, suggesting that LFREE still has margin to keep scaling if the number of cores increases (this is clearly not the case of LOCAL for the bunny scene, where the inflection point seems to have been reached - see figure 5(c)). Figure 7 shows aggregate data access times for the three scenes on the 24 core server; aggregate data access time is the average time spent waiting for data access in the work queue(s) summed over all threads. These are constant for LOCAL, scale sublinearly for LFREE, but grow exponentially for LOCK. This result clearly shows that LOCK performance degrades as the number of threads increases, while the other two approaches are still quite far from their inflection point. LFREE and LOCAL have thus the potential to scale to larger numbers of cores/threads, which is an important result given the actual trend to increase the core count in modern processors.

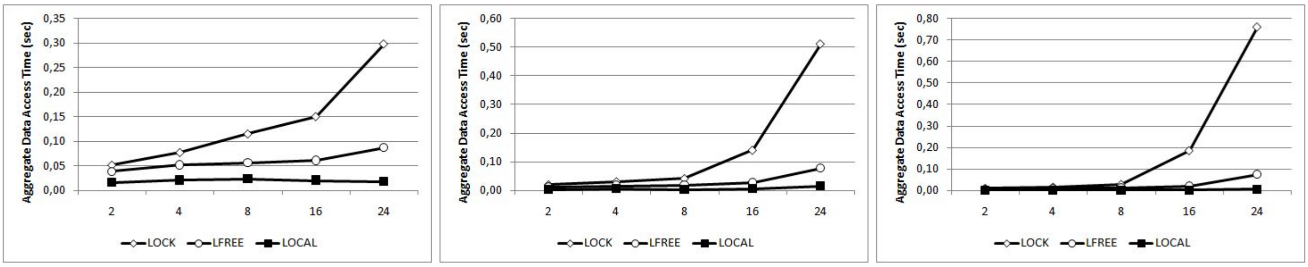
With respect to absolute results it is clear that iRT is not scaling well for a large number of cores. This specially true for the 24 core server, with efficiency values under 50%. Given that ray tracing is an embarrassingly parallel algorithm this suggests some major inefficiencies in the ray tracing engine. Figure 7 shows these inefficiencies do not reside in accessing the work queue. On the other hand, idle times due to load balancing are quite significant for all approaches as shown in figure 6. Additionally, interactive ray tracing is known for being memory bandwidth limited; the poor speedups reported might be due to a memory wall accessing scene data, such as the acceleration data structure and geometric primitives for intersection. iRT is still a preliminary prototype; these issues will be dealt with in the near future.

6. CONCLUSIONS

This paper compares three different data access control mechanisms, used to share access to a FIFO-queue holding tasks (sets of rays) for a multithreaded interactive ray tracer. One approach is based on using locks to provide mutual exclusion to critical regions. This approach, which uses different locks for reading and writing thus reducing contention and serialisation, is referred to as LOCK. The lock-free synchronisation approach (LFREE) avoids all locks by carefully reordering instructions. Finally, within the local approach each thread maintains a local work queue, preventing work sharing but also dispensing with access control mechanisms. These approaches were evaluated on three different multicore systems: two different 8-core servers based on the Intel Xeon 5400 and on the Intel Xeon 5500 (supporting HyperThreading) and a 24-core system based on the Intel Xeon 7400.

Results have shown that the LOCAL approach outperforms the other two both in raw performance and scalability. This result is explained by the fact that this approach incurs no data access control overheads. Overheads due to load imbalance do not occur due to both the fine granularity of the tasks and to the homogeneity and shallowness of the per pixel ray trees depths.

LOCK and LFREE perform similarly for a moderate number of cores. However, as the core count increases, the time spent waiting to enter critical sections with locks starts to grow exponentially. With the lock free approach this overhead increases sublinearly, having a significant impact on the achieved frame rates. This result is specially relevant due to the ever increasing core count in modern processors. The performance of future shared memory manycore systems will be, with high probability, dependent on the ability to efficiently and robustly share data structures.



(a) Conference

(b) Office

(c) Bunny

Figure 7. Average aggregate time spent accessing data, in seconds, per frame on the 24-core Xeon 7400 server (time summed across threads)**.**

iRT, the prototype interactive ray tracer engine used to produce these results, is still in a preliminary development stage. Important features have not been properly addressed, such as resorting to explicit vectorial programming, packeting of coherent rays and maximising locality of memory references. These issues, fundamental to achieve interactive frame rates within a global illumination context, which requires the shooting of several additional rays, will be addressed in the near future.

7. Acknowledgements

This work was partially supported by research project *IGIDE* – FCT grant PTDC/EIA/65965/2006.

References

[Bigler 06] J. Bigler, A. Stephens, and S. Parker. Design for Parallel Interactive Ray Tracing Systems. In *IEEE Symp. on Interactive Ray Tracing*, 2006.

[Debattista 03] K. Debattista, K. Vella, and J. Cordina. Wait-free cache affinity thread scheduling. In *IEE Proc. on Software*, volume 150, pages 137–146, 2003.

[Debattista 09] K. Debattista, P. Dubla, F. Banterle, L. Santos, and A. Chalmers. Instant Caching for Interactive Global Illumination. *Computer Graphics Forum*, 2009. (to appear).

[Dubla 09] P. Dubla, K. Debattista, L. Santos, and A. Chalmers. Wait-Free Shared-Memory Irradiance Cache. In *EG Symp. on Parallel Graphics and Visualization*, 2009.

[Greenberg 99] D. Greenberg. A Framework for Realistic Image Synthesis. *Communications of the ACM*, 42(8):44–53, August 1999.

[Herlihy 08] M. Herlihy and N. Shavit. *The Art of MultiProcessor Programming*. 2008.

[Keller 97] A. Keller. Instant Radiosity. In *SIGGRAPH '97: Proc. of Conf. on Computer graphics and interactive techniques*, pages 49–56, 1997.

[Kollig 02] T. Kollig and A. Keller. Efficient Multidimensional Sampling. *Computer Graphics Forum*, 21(3), 2002.

[Michael 96] M. Michael and M. Scott. Simple, fast and practical non-blocking and blocking concurrent queue algorithms. In *Proc. of ACM Symp. on Principles of Distributed Computing*, pages 267–275. ACM Press, 1996.

[Reshetov 05] A. Reshetov, A. Soupikov, and J. Hurley. Multi-Level Ray Tracing Algorithm. *ACM Transaction on Graphics*, 24(3):1176–1185, 2005.

[Wald 02a] I. Wald, C. Benthin, and P. Slusallek. A Flexible and Scalable Rendering Engine for Interactive 3D Graphics. Technical Report TR-2002-01, Saarland Univ., 2002.

[Wald 02b] I. Wald, C. Benthin, and P. Slusallek. OpenRT - A Flexible and Scalable Rendering Engine for Interactive 3D Graphics. Technical report, Saarland Univ., 2002.

[Wald 07a] I. Wald, C. Gribble, S. Boulos, and A. Kensler. SIMD Ray Stream Tracing - SIMD Ray Traversal and Generalized Ray packets and on-the-fly Reordering. Technical Report UUSCI-2007-012, SCI - Univ. of Utah, August 2007.

[Wald 07b] I. Wald, W. Mark, J. Gunther, S. Boulos, T. Ize, W. Hunt, S. Perker, and P. Shirley. State of the Art in Ray Tracing Animated Scenes. In *Eurographics Conf. - State of the Art Reports*, 2007.

[Wald 08] I. Wald, C. Benthin, and S. Boulos. Getting Rid of Packets - Efficient SIMD Single-Ray Traversal using Multibranching BVHs. In *IEEE Symp. on Interactive Ray Tracing*, 2008.