Eurographics Symposium on Parallel Graphics and Visualization (2009)
J. Comba, K. Debattista, and D. Weiskopf (Editors)

Wait-Free Shared-Memory Irradiance Cache

Piotr Dubla[1], Kurt Debattista[1], Luís Paulo Santos[†2] and Alan Chalmers[1]

[1]The Digital Lab, WMG, University of Warwick, United Kingdom
[2]Departamento de Informática, Universidade do Minho, Portugal

Abstract

The irradiance cache (IC) is an acceleration data structure which caches indirect diffuse irradiance values within the context of a ray tracing algorithm. Previously calculated values can be stored and reused in future calculations; such reutilisation enables exploitation of spatial and, in static scenes, temporal coherence, resulting in an order of magnitude improvement in computational performance. In multithreaded shared memory parallel systems the irradiance cache must be shared among rendering threads in order to achieve high efficiency levels. Since all threads read and write from this shared data structure an access control mechanism is required, which ensures that the data structure is not corrupted. Besides assuring correct accesses to the IC this access mechanism must incur minimal overheads such that performance is not compromised.

In this paper we propose a new wait-free access mechanism to the shared irradiance cache. Wait-free data structures, unlike traditional access control mechanisms, do not make use of any blocking or busy waiting, avoiding most serialisation and reducing contention. We compare this technique with two other classical approaches: a lock based mechanism and a local write technique, where each thread maintains its own cache of locally evaluated irradiance values. We demonstrate that the wait free approach significantly reduces synchronisation overheads compared to the two other approaches and that it increases data sharing over the local copy technique. This is, to the extent of our knowledge, the first work explicitly addressing access to a shared IC; this problem is becoming more and more relevant with the advent of multicore systems and the ever increasing number of processors within these systems.

Categories and Subject Descriptors (according to ACM CCS): Computer Graphics [I.3.3]: Picture/Image Generation—

1. Introduction

Rendering global illumination light transport effects within a ray tracing context is a computationally very demanding task. Recent improvements in the field of ray tracing have made it possible to interactively compute many of the global effects, such as specular phenomena and correct shadows [WMG*07]. Indirect diffuse interreflections, however, require dense sampling of the hemisphere at each shading point, dramatically increasing rendering times. Ward et al. [War88] exploit the fact that the indirect diffuse component is generally a continuous smooth function over space not affected by the high frequency changes common with the specular component. They proposed the irradiance cache (IC) data structure to allow sparse evaluation of indirect diffuse irradiance. Sparsely calculated irradiance values are stored in the IC and reused to extra(inter)polate irradiance values at nearby locations. By exploiting spatial coherence, the IC offers an order of magnitude improvement in rendering time over unbiased Monte Carlo integration. Performance is further improved when rendering animations of static scenes, since the indirect diffuse irradiance remains constant and the IC records can thus be maintained.

In multithreaded shared memory systems the irradiance cache must be shared to avoid replicated computations of diffuse samples among rendering threads, thus increasing efficiency. Since all rendering threads can read and write from the IC, a data access control mechanism is required to ensure that the data structure is not corrupted. Such control mech-

† partially supported by project IGIDE, PTDC/EIA/65965/2006, funded by the Portuguese Foundation for Science and Technology

anism incurs overheads, such as serialisation of accesses to the shared data structure; it must thus be carefully designed in order not to compromise performance. Traditionally, access control to shared memory data structures is maintained via mutual exclusion, typically, using locks when frequent access is required. However, alternatives that avoid the serialisation and contention do exist, in the form of lock free synchronisation [Her91]. By carefully ordering instructions lock free algorithms can guarantee no form of serialisation of code, from the removal of all critical sections, and a drastic reduction in contention. Certain lock free synchronisation methods can guarantee to complete in a finite number of steps thus avoiding starvation. Such access mechanisms are said to be using wait-free synchronisation.

In this paper we propose and compare three different mechanisms to share the irradiance cache among threads on a shared memory system. The first is based on traditional locking techniques. The lock based algorithm locks the shared IC every time a thread accesses it, both for reading and writing. The second is a local copy method which avoids concurrent access control by maintaining a local IC, per thread, and merging at the end of each frame. The final method is a wait-free IC which allows concurrent access to the shared IC by all threads without using any locks or critical sections.

With the advent of multicore systems and the ever increasing number of processors available within these systems, efficient access to shared data structures becomes an important issue, with the potential to strongly influence the renderer performance. Efficient sharing of the irradiance cache in multithreaded systems is mandatory in order to achieve high efficiency levels, since computed irradiance values become readily available to all threads, thus avoiding work replication. This is specially relevant because utilization of the irradiance cache has increased significantly over the last few years, e.g., as a stand-alone algorithm for computing indirect (ir)radiance [SKDM05, TL04, KGPB05], as an acceleration data structure for rendering participating media phenomena [JDZJ08] or used in conjuction with photon mapping [Jen01]. This paper contributions are the proposal of an efficient wait-free algorithm for sharing the irradiance cache among rendering threads on shared memory systems and a comparison of the proposed algorithm efficiency with two traditional data access control mechanisms: a lock-based approach and a local copy one.

This paper is structured as follows. In the next section we present related work. In Section 3, we present the algorithms for the three data access control mechanisms. In section 4 we compare results and, finally, in Section 5 we conclude and describe possible future work.

2. Related Work

2.1. Shared Irradiance Cache

The IC is an acceleration data structure which caches indirect diffuse irradiance samples within the framework of a distributed ray-tracing algorithm. Initial samples, sparsely distributed over the scene, are calculated by densely sampling the hemisphere and the result is cached in the IC. Whenever a new indirect value is required the irradiance cache is first consulted. If one or more samples fall within the user-defined search radius of the indirect diffuse value to be computed, the result is extrapolated from the samples using a weighted averaging strategy. In order to accelerate range searches, performed to locate valid samples within the IC, an octree is incrementally built every time a new sample is added; writing to the cache requires both storing the new indirect diffuse irradiance value and updating the octree topology. Ward et al. [War88] demonstrated that the irradiance cache offers an order of magnitude improvement in overall computational time over traditional unbiased Monte Carlo integration. Performance is improved even further when rendering animations of static scenes, since the indirect diffuse irradiance remains constant.

In parallel systems each rendering process (or thread) might evaluate new indirect diffuse irradiance values and add them to the IC. In order to increase efficiency, the IC must be shared among all processes, thus avoiding replicated work, where one process evaluates an irradiance value that might already have been evaluated by another process. The IC becomes a shared data structure, thus requiring some sharing mechanism assuring that all processes can access the available data, that the data is not corrupted and that overheads do not compromise efficiency.

In distributed memory systems, such as clusters of workstations, each node has its own address space, resulting on multiple copies of the shared data structure that are regularly synchronised. The standard Radiance distribution [War94] supports a parallel renderer over a distributed system using the Network File System for concurrent access of the irradiance cache; this has been known to lead to contention and may result in poor performance when using inefficient file lock managers. Koholka et al. [KMG99] broadcast irradiance cache values amongst processors after every 50 samples calculated at each slave. Robertson et al. [RCLL99] presented a centralised parallel version of Radiance whereby the calculated irradiance cache values are sent to a master process whenever a threshold is met. Each slave then collects the values deposited at the master by the other slaves. [DSC06] propose restricting diffuse irradiance evaluations to a subset of the available processors and synchronising the IC among these at a higher frequency than with the remaining processors.

We are not aware of any publication describing a data access control mechanism for sharing the irradiance cache

```
1 atomic XADD(address location)
2 {
3     int value = *location;
4     *location = value + 1;
5     return value
6 }
```

Listing 1: Fetch and Add

```
1 atomic CAS(address location, value cmpVal, value newVal)
2 {
3     if (*location == cmpVal)
4     {
5         *location = newVal;
6         return true;
7     }
8     else return false;
9 }
```

Listing 2: Compare and swap

among rendering threads in a shared memory parallel system. This is, to the extent of our knowledge, the first work explicitly addressing and evaluating this issue.

2.2. Synchronisation

Traditionally, access control to shared data structures is maintained through either lock-based mechanisms or blocking which ensure safe access to critical sections of code. Such mechanisms serialise access to the shared data structure (both reads and writes), thus incurring intolerable performance penalties when this is frequently accessed, as would be the case with the shared IC. Furthermore, when using locks, contention will increase with the degree of concurrency, thus, rather than scaling, lock-based approaches will perform worse as the number of concurrent threads increases. Blocking threads typically requires a horizontal context switch which is too computationally expensive on modern operating systems to be used for frequent access to critical sections.

An alternative method of synchronisation is the use of lock-free structures and algorithms, which rely on atomic conditional primitives to protect shared data structures from unsafe concurrent access [Her91]. Lock free synchronisation dispenses critical sessions, thus avoiding deadlocks and the serialisation of concurrent tasks. These algorithms may be either nonblocking or wait-free. Non blocking algorithms are guaranteed to terminate in finite time, but they are based on retries, which can result in unpredictable delays and starvation. Wait-free algorithms, on the other hand, are guaranteed to complete in a fixed number of steps which means they remove deadlock, improve fault tolerance, are immune to the priority inversion problem and avoid starvation when accessing the shared data structure.

The construction of wait-free algorithms requires the use of powerful atomic primitives which are executed as a single instruction, without any interruption, on modern architectures. We show pseudo code, refer to Listings 1 and 2, for the two atomic instructions, compare and swap (CAS) and fetch and add (XADD), that we will be using for our wait-free IC.

3. Algorithms

In this section the algorithms for the three evaluated data access control mechanisms are presented. To begin with we show a traditional sequential irradiance cache with no access control in Listing 3. In the subsequent sections we demonstrate how the traditional approach can be modified to enable the different access control mechanisms.

```
1  IrradianceCache IC;
2
3  ComputeIndirectDiffuse() {
4      //get irradiance from IC if there are valid records
5      inIC = IC.getIrradiance ();
6      if (!inIC) { // no valid records found
7          // compute it by sampling the hemisphere
8          ICsample = ComputeIrradianceRT ();
9          // insert new IC sample into the octree
10         IC.insert (ICsample);
11     }
12 }
13
14 IrradianceCache::getIrradiance() {
15     <Traverse the octree searching for valid records>
16     if (found) return true;
17     else return false;
18 }
19
20 IrradianceCache::insert (ICsample) {
21     // recursively traverse the octree
22     // starting at root
23     IC.root.insert (ICsample);
24 }
25
26 ICNode::insert (ICSample) {
27     if (correct insertion node) {
28         IClist.Add (ICsample);
29     } else {
30         // go deeper in the octree
31         xyz = EvaluateOctant();
32         if (children[xyz] == NULL)
33             children[xyz] = new ICNode ();
34         children[xyz].insert (ICsample);
35     }
36 }
37
38 ICList::Add (ICsample) {
```

```
39      // insert new record in head of list
40      IClist.records[head++] = ICsample;
41  }
```

Listing 3: Traditional sequential IC

3.1. Lock-Based Irradiance Cache (LCK)

The lock-based access control algorithm locks the IC whenever a read or write is made to it. However, the code responsible for hemisphere sampling, ComputeIrradianceRT(), is not a critical region (Listing 4 lines 4 - 6, 12 - 14), thus allowing concurrent evaluation of irradiance. The major disadvantage of the LCK approach is that it serialises all accesses, both reads and writes, to the shared IC. As the number of threads increases, contention will also increase, preventing performance to scale with the degree of parallelism. Table 1 shows that the overhead associated with locks (time spent waiting to enter critical regions summed over all threads) increases substantially when going from two to eight threads.

```
1   ComputeIndirectDiffuse()
2   {
3       //get irradiance from IC if there are valid records
4       IC.lock();
5       inIC = IC.getIrradiance ();
6       IC.unlock();
7
8       if (!inIC) { // no valid records found
9           // compute it by sampling the hemisphere
10          ICsample = ComputeIrradianceRT ();
11          // insert new IC sample into the octree
12          IC.lock();
13          IC.insert (ICsample);
14          IC.unlock();
15      }
16  }
```

Listing 4: Lock-based IC

3.2. Local-Write Irradiance Cache (LW)

An alternative approach is to have a global IC readable by all threads and an additional local IC per thread; each thread writes only on its local IC but reads from both. At certain predefined execution points, such as the end of a frame, the local ICs are sequentially merged onto the global IC. This form of synchronisation uses an end of frame as a barrier, effectively this is a blocking approach to synchronisation.

The major drawback of this approach is that it does not allow for any sharing within a single frame, thus resulting in work replication; this is reflected in Table 1, where the LW algorithm has a much higher IC sample count than the other two approaches. The time taken to sequentially merge the caches is not significant, as can be seen in the overheads section of Table 1 (at least up to eight threads). Additionally, memory consumption is dictated by the number of threads being used and the complexity of the octree itself.

```
1   IrradianceCache IClocal[number threads], ICglobal;
2
3   ComputeIndirectDiffuse()
4   {
5       //get irradiance from IC if there are valid records
6       inIC = ICglobal.getIrradiance ();
7
8       if (!inIC)
9           inIC = IClocal[current thread].getIrradiance ();
10
11      if (!inIC) { // no valid records found
12          // compute it by sampling the hemisphere
13          ICsample = ComputeIrradianceRT ();
14          // insert new sample into the local cache
15          IClocal[current thread].insert (ICsample);
16      }
17  }
```

Listing 5: Local-Write IC

3.3. Wait-Free Irradiance Cache (WF)

The wait-free algorithm does not rely on any critical sections to both read and write to the shared IC. When adding samples to an IC node the atomic XADD operator is used, returning a unique index into the list of records, which ensures that samples are never over-written; simultaneously, the private index to the next free position is incremented. When adding a new child node to the octree, the new subtree is built using a temporary pointer. When fully built, the subtree is attached to the octree using the CAS operator. If the relevant child still does not exist, then CAS completes successfully. If, however, another thread wrote to the same child, then CAS will fail and this thread will discard both the created subtree and the associated sample after utilising it for the current computation. As can be seen in Table 1 the number of discarded samples is minimal, amounting to no more than 0.002% of the total samples.

The atomic primitives used in most wait-free algorithms still need a memory barrier in order to ensure out-of-order execution does not corrupt the shared data structure. Typically a memory barrier precedes the use of atomic primitives such as CAS and XADD. This can often be expensive since out-of-order execution typically accounts for an increase in performance. In our wait-free algorithm we keep memory barriers to a minimum by only calling them for when inserting IC samples into the octree and not when accessing the cache for interpolation. In this way the much more frequent IC interpolation requests do not entail any overheads over the serial methods, as the code is effectively the same as the sequential version.

The wait free approach ensures that the single shared IC can be accessed concurrently by all threads, and, as can be seen in Listing 6, requires little changes in the code from a traditional sequential irradiance cache. As we shall show in the next section, this results in faster execution times both when interpolating and creating IC samples and also it does

not suffer the substantially larger memory requirements of the LW approach.

```
1
2  ICNode::insert (ICSample) {
3      if (correct insertion node)
4          IClist.Add (ICsample);
5      else { // go deeper in the octree
6          xyz = EvaluateOctant();
7          if (children[xyz]==NULL) {
8              temp = new ICNode()
9              temp.insert (ICsample);
10             // Update new branch into the octree
11             // This only occurs on the first level of
12             // recursion subsequent levels just insert
13             // normally.
14             if (!CAS (children[xyz], NULL, temp))
15                 free temp;
16         }
17         else
18             children[xyz].insert (ICsample);
19     }
20 }
21
22 IClist::Add (ICsample) {
23   // get index of new sample in node list
24   int index = XADD (&head);
25   IClist.records[index] = ICsample;
26 }
```

Listing 6: Wait-Free IC

4. Results

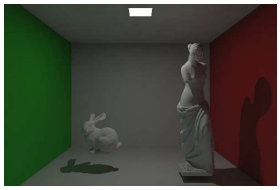
All presented results have been generated on an 8-core (dual quad-core) Intel Xeon machine running at 2.5GHz with 8 gigabytes of RAM, using our own interactive ray tracer. Experiments were run under CentOS 4 with the code being compiled with ICC 9.0. The renderer utilised does not make use of packetisation or SIMD operations except for the ray-bounding volume intersection test used when traversing the acceleration data-structure, which is a BVH implementation based on [WBS07].

Five different scenes (see Figure 4 for details) were utilised, providing a range of geometric complexity, physical dimensions and lighting conditions.

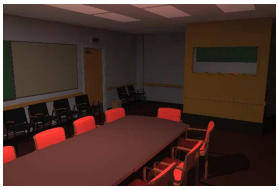
The experiments were run for both still images as well as walkthroughs within static scenes.

4.1. Still images

Table 1 provides results for one, two, four and eight threads with the time taken to calculate the frame, the number of IC samples generated, overheads associated with each algorithm and speed-up. Results for one thread were obtained using the traditional sequential approach (TRA) and speed-up is computed with respect to these results. For the lock-based approach (LCK) we report as an overhead the aggregate time



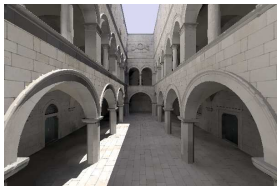
(a) Cornell (*48,339 polygons*)



(b) Conference Room (*190,953 polygons*)

(c) Desk (*12,424 polygons*)

(d) Office (*20,771 polygons*)



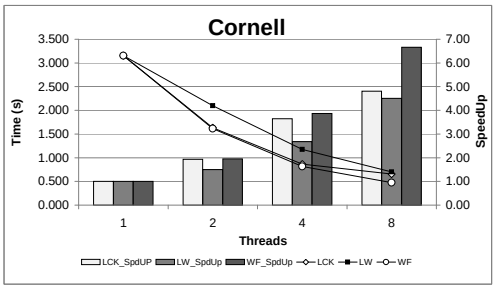
(e) Sponza (*66,454 polygons*)

Figure 4: The five scenes utilised in the experiments

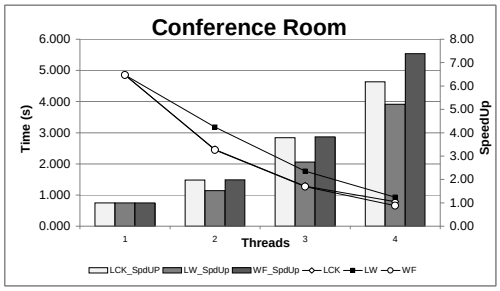
spent to enter critical regions summed over all threads. For local-write (LW) the reported overhead is the time taken to sequentially merge all local caches into the global one at the end of the frame, for wait-free(WF) it is the number of samples discarded. Each image was calculated with an empty irradiance cache to show a worst-case scenario with maximal irradiance calculations occurring. Graphs of all this data are presented in Figure 1; the left Y-axis and the accompanying line graph shows the time taken in seconds to render the frame while the right Y-axis and the bar graph shows the speed-up compared to the traditional single-threaded irradiance cache with no access control.

LW performs and scales worst than the two other algorithms. This is because no sharing is actually occurring since only one frame is rendered and merging of the local caches only happens at the end of the frame. Each thread must evaluate all irradiance samples that project into its assigned tiles of the image plane, leading to much work replication as can be seen by the number of evaluated irradiance samples.

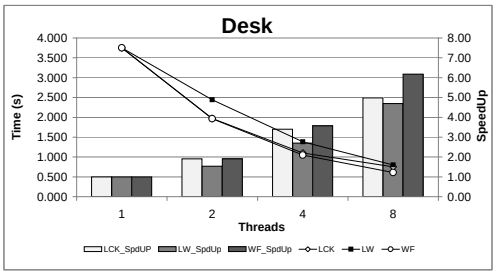
The performance difference between LCK and WF becomes evident as the number of threads increases. The aggregated time waiting for locks increases with the number of threads, resulting on a major performance loss. The wait-free algorithm scales much better because it does not serialise neither writes nor reads to the shared data structure. For a reduced number of threads LCK performs similar to WF since most of the time is spent evaluating new irradiance samples, which is not a critical region of the code. As the number of



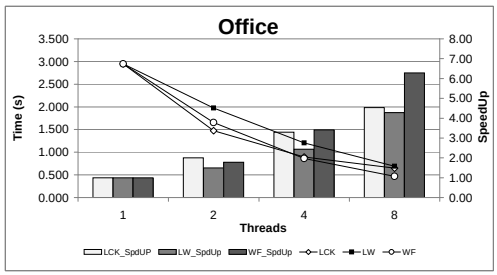
(a) Cornell



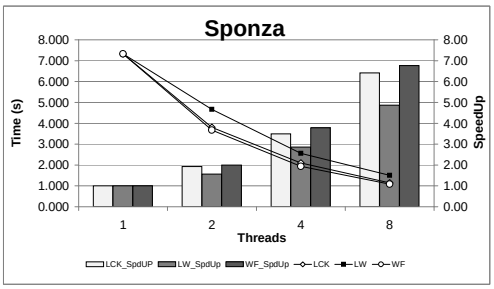
(b) Conference Room



(c) Desk



(d) Office



(e) Sponza

Figure 1: Still Images: Results for all scenes

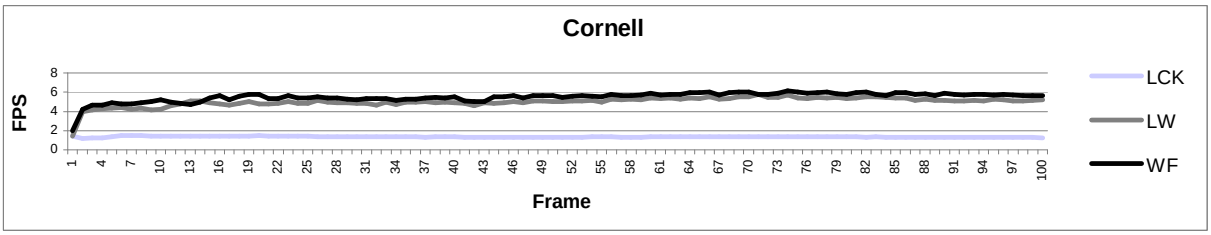


Figure 2: Animation results for Cornell Box (8 threads)

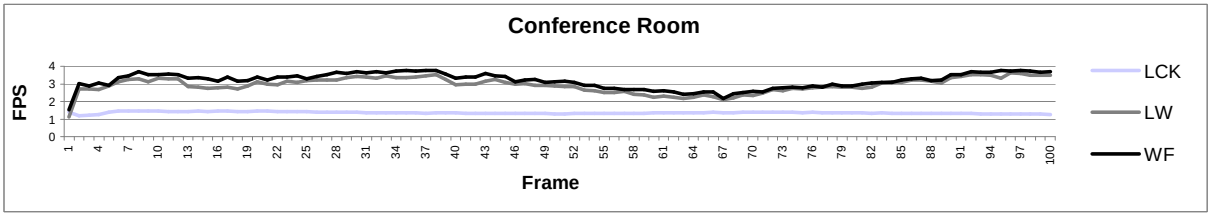


Figure 3: Animation results for Conference Room (8 threads)

	Cornell									
	1	**2**			**4**			**8**		
	TRA	*LCK*	*LW*	*WF*	*LCK*	*LW*	*WF*	*LCK*	*LW*	*WF*
Time (s)	3.152	1.633	2.096	1.614	0.863	1.178	0.814	0.656	0.700	0.473
IC samples	3463	2742	4339	2707	2483	4404	2473	2441	4440	2410
Overheads†	0	0.053	0.024	4	0.189	0.018	1	1.370	0.019	7
Speed-up	1.000	1.988	1.548	2.011	3.759	2.756	3.986	4.950	4.640	6.862
	Desk									
	1	**2**			**4**			**8**		
	TRA	*LCK*	*LW*	*WF*	*LCK*	*LW*	*WF*	*LCK*	*LW*	*WF*
Time (s)	3.749	1.971	2.444	1.965	1.104	1.385	1.049	0.753	0.798	0.607
IC samples	3477	3038	4282	2998	2748	4394	2775	2700	4378	2709
Overheads†	0	0.035	0.018	4	0.189	0.021	0	1.259	0.023	8
Speed-up	1.000	1.902	1.534	1.907	3.396	2.707	3.572	4.976	4.696	6.178
	Conference Room									
	1	**2**			**4**			**8**		
	TRA	*LCK*	*LW*	*WF*	*LCK*	*LW*	*WF*	*LCK*	*LW*	*WF*
Time (s)	4.854	2.460	3.180	2.445	1.283	1.768	1.269	0.786	0.929	0.658
IC samples	3065	2517	3663	2524	2272	3817	2256	2130	3842	2170
Overheads†	0	0.064	0.028	1	0.192	0.021	3	1.127	0.026	3
Speed-up	1.000	1.973	1.526	1.985	3.783	2.745	3.826	6.176	5.223	7.381
	Office									
	1	**2**			**4**			**8**		
	TRA	*LCK*	*LW*	*WF*	*LCK*	*LW*	*WF*	*LCK*	*LW*	*WF*
Time (s)	2.947	1.474	1.976	1.654	0.895	1.207	0.864	0.650	0.689	0.469
IC samples	2089	1881	2650	1976	1803	3199	1802	1766	3290	1785
Overheads†	0	0.042	0.019	1	0.337	0.018	3	1.397	0.022	1
Speed-up	1.000	1.999	1.491	1.782	3.291	2.442	3.412	4.531	4.276	6.287
	Sponza									
	1	**2**			**4**			**8**		
	TRA	*LCK*	*LW*	*WF*	*LCK*	*LW*	*WF*	*LCK*	*LW*	*WF*
Time (s)	7.330	3.802	4.672	3.676	2.100	2.563	1.935	1.143	1.505	1.083
IC samples	3357	3113	4166	3113	3032	4286	2958	2929	4379	2942
Overheads†	1	0.046	0.026	1	0.186	0.026	5	1.018	0.028	4
Speed-up	1.000	1.928	1.569	1.994	3.779	2.860	3.707	6.413	4.872	6.766

Table 1: Results for all scenes

†- Overheads are all in seconds except for WF which is number of irradiance samples discarded

threads increases, more range searches are performed; since these are serialised in LCK, a performance penalty arises.

4.2. Animations

Two scenes, Cornell and Conference Room, were selected and each was rendered, using 8 threads, for 100 frames while the camera did a 360 degrees rotation around the scene. Each frame in the sequence re-utilised previously created cache samples while simultaneously calculating new ones. This provides an overview of performance when a more balanced mix of evaluation and interpolation is occurring, unlike the case for the still images. The results for these particular experiments are displayed in Figures 2 and 3, showing the time taken to render each of the 100 frames for the Cornell and Conference Room scenes respectively for each of the three algorithms. For each of the scenes the first frame is the equivalent of the still images above, where the cache is totally empty and all the samples needed to be generated.

Clearly, LCK performs worse than LW and WF, except for the first frame. Since for the remaining frames the IC will not be empty, many irradiance samples can be reused; but LCK serialises all range searches performed to locate these samples, thus severely impacting on performance. WF outperforms LW because the former shares irradiance samples immediately without any extra overhead associated with reading, while the latter does not share samples within a frame, thus resulting on costly extensive evaluations of more indirect diffuse irradiance values. Summarising, LCK is mostly penalised by reading serialisation, whereas LW is penalised by work replication.

5. Conclusions

We proposed a new wait-free data access control mechanism for sharing the irradiance cache among multiple rendering threads on a shared memory parallel system and evaluate it against two traditional data access algorithms: a lock-based approach and a local write one. We demonstrate that the proposed approach outperforms the others and scales better with the number of threads.

The lock-based algorithm serialises all accesses to the shared data structure, reads included. Range searches performed in the octree to locate valid irradiance samples are serialised, resulting on huge performance losses; this problem is aggravated with the number of threads and the resulting contention. The local write algorithm does not share any irradiance values evaluated within each frame, thus suffering a performance penalty as a result of work replication. Neither of these two algorithms scales well as the number of threads increases.

The wait-free algorithm does not serialise accesses to the shared data structure and irradiance values are immediately shared among all threads without any overhead associated with reading. It exhibits the best frame rates for walkthroughs within static scenes and scales well with the number of threads, achieving an efficiency between 77% and 92% for 8 threads.

The proposed wait-free data access control mechanism is both efficient and simple to implement, requiring only minor modifications to a traditional sequential irradiance cache implementation. The design of efficient, scalable and reliable mechanisms to control access to shared data structures within shared memory systems is a relevant issue. With the advent of multicore systems and, in the near future, of many-core systems – where the degree of concurrency is expected to be much larger than with current machines – its relevance keeps on increasing.

5.1. Future Work

Although the wait-free algorithm has shown good scalability with up to eight threads we would like to investigate the limits of this trend by running the algorithm on machines with a larger number of processors sharing the same address space. Also the memory organisation, e.g., NUMA architectures, might impact on the performance of the proposed algorithm. The results presented indicate, however, that wait-free algorithms will most probably perform better than the evaluated alternatives.

Utilisation of the irradiance cache within dynamic environments, i.e., those where geometry might change between frames, requires the ability to remove from the shared data structure records which became invalid. We intend to investigate and assess wait-free synchronisation algorithms supporting this removal operation.

References

[DSC06] DEBATTISTA K., SANTOS L. P., CHALMERS A.: Accelerating the irradiance cache through parallel component-based rendering. In *Eurographics Symposium on Parallel Graphics and Visualization* (2006), Eurographics.

[Her91] HERLIHY M.: Wait-free synchronization. *ACM Trans. Program. Lang. Syst. 13*, 1 (1991), 124–149.

[JDZJ08] JAROSZ, DONNER, ZWICKER, JENSEN: Radiance caching for participating media. *ACM Transactions on Computer Graphics 27*, 1 (March 2008).

[Jen01] JENSEN H. W.: *Realistic image synthesis using photon mapping*. A. K. Peters, Ltd., Natick, MA, USA, 2001.

[KGPB05] KRIVANEK J., GAUTRON P., PATTANAIK S., BOUATOUCH K.: Radiance caching for efficient global illumination computation. *IEEE Transactions on Visualization and Computer Graphics 11*, 5 (2005), 550–561.

[KMG99] KOHOLKA R., MAYER H., GOLLER A.: MPI-parallelized Radiance on SGI CoW and SMP. In *ParNum'99: 4th Int. ACPC Conf.* (1999), Springer-Verlag, pp. 549–558.

[RCLL99] ROBERTSON D., CAMPBELL K., LAU S., LIGOCKI T.: Parallelization of radiance for real time interactive lighting visualization walkthroughs. In *ACM/IEEE Supercomputing* (1999), ACM Press, p. 61.

[SKDM05] SMYK M., KINUWAKI S., DURIKOVIC R., MYSKOWSKI K.: Temporally Coherent Irradiance Caching for High Quality Animation Rendering. *Computer Graphics Forum 24*, 3 (2005), 401–412.

[TL04] TABELLION E., LAMORLETTE A.: An Approximate Global Illumination System for Computer Generated Films. *ACM Transactions on Graphics 23*, 3 (2004), 469–476.

[War88] WARD G.: A ray tracing solution for diffuse interreflection. *Computer Graphics-SIGGRAPH'88 22*, 4 (August 1988).

[War94] WARD G.: The radiance lighting simulation and rendering. *Computer Graphics-SIGGRAPH'94* (1994), 459–472.

[WBS07] WALD I., BOULOS S., SHIRLEY P.: Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph. 26*, 1 (2007), 6.

[WMG*07] WALD I., MARK W. R., GÜNTHER J., BOULOS S., IZE T., HUNT W., PARKER S. G., SHIRLEY P.: State of the art in ray tracing animated scenes. In *STAR Proceedings of Eurographics 2007* (September 2007), Schmalstieg D., Bittner J., (Eds.), The Eurographics Association, pp. 89–116.