Volume 0 (1981), Number 0 pp. 1–13 COMPUTER GRAPHICS *forum*

Distributed Out-of-Core Stochastic Progressive Photon Mapping

Tobias Günther[1] and Thorsten Grosch[2]

[1]Visual Computing, University of Magdeburg, Germany
[2]Computational Visualistics, University of Magdeburg, Germany

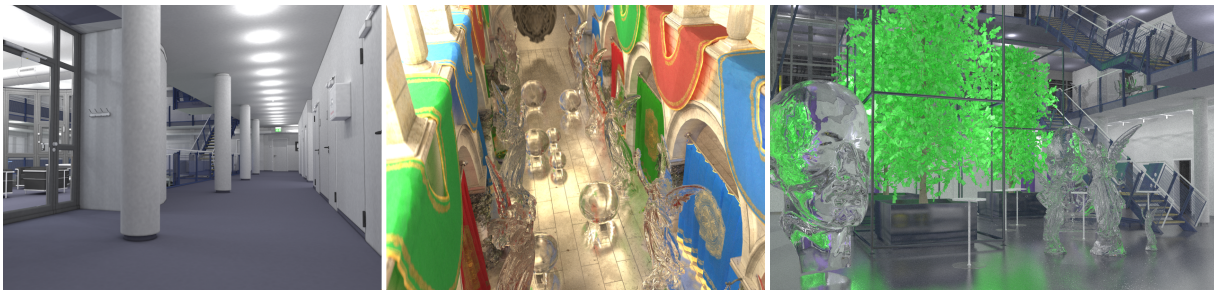

Figure 1: *Consistent global illumination for out-of-core scenes. Left to right: 70 M triangles and many light sources, 40 M triangles with glossy surfaces and depth-of-field, and 90 M triangles with a large number of glass objects.*

Abstract

At present, stochastic progressive photon mapping (SPPM) is one of the most comprehensive methods for a consistent global illumination computation. Even though the number of photons is unlimited due to their progressive nature, the scene size is still bound by the available main memory. In this paper, we present the first consistent out-of-core SPPM algorithm. In order to cope with large scenes, we automatically subdivide the geometry and parallely trace photons and eye rays in a portal-based system, distributed across multiple machines in a commodity cluster. Moreover, modifications of the original SPPM method are introduced that keep both the utilization of tracer machines high and the network traffic low. Therefore, compared to a portal-based single machine setup, our distributed approach achieves a significant speedup. We compare a GPU-based with a CPU-based implementation and demonstrate our system in multiple large test scenes of up to 90 million triangles.

This is the authors preprint. The definitive version is available at http://diglib.eg.org/ and http://onlinelibrary.wiley.com/.

1 Introduction

A realistic image synthesis is required in many applications today. Architecture, urban design and especially movie production often require the processing of extremly large scenes, which is challenging due to their sheer size. A frequent choice for a *consistent*, i.e., photometrically correct, rendering solution is the stochastic progressive photon mapping (SPPM) of Hachisuka and Jensen [HJ09]. Yet, it was not efficiently applied to large scenes, even though it proved to converge faster for specular-diffuse-specular lighting effects, compared to traditional Monte Carlo-based renderers, e.g., bi-directional path tracing [LW93]. Out-of-core photon mapping, such as the portal-based approach of Fradin et al. [FMH05] (not yet consistent), proved rather memory unfriendly, since photons follow highly scattered paths and incoherently bounce through the entire scene. For consistent renderings, this requires frequently swapping the scene data at *highest detail* to continue the tracing, because pho-

© 2014 The Author(s)
Computer Graphics Forum © 2014 The Eurographics Association and John Wiley & Sons Ltd. Published by John Wiley & Sons Ltd.

tons may revisit the same scene parts multiple times. Therefore, the simulation quickly becomes memory-bound due to the expensive and frequent loading of scene data. Furthermore, there is a large variation in ray computation time, which causes periods of under-utilization of parallel processors, when only few rays are traced toward the end of each iteration. Another open question is the choice of hardware for such a configuration. While the GPU-based SPPM implementation of Hachisuka and Jensen [HJ10] is faster than their original CPU-based SPPM [HJ09] for scenes fitting into GPU memory, it remains unclear how well both techniques scale in an out-of-core setting—especially since GPUs have less memory to operate on.

In this paper, we extend both the CPU-based and GPU-based SPPM implementation of Hachisuka and Jensen [HJ09, HJ10] to cope with large scenes by using a portal-based system, similar to Fradin et al. [FMH05], based on an *automatic* scene subdivision. We study and compare a GPU-based, CPU-based and hybrid implementation to compile a guideline, which method to choose in a given scene. To increase the throughput in all cases, we distribute the tracing of photons and eye rays across multiple tracer machines in a commodity cluster. As a result, we reduce the scene loading operations, since effectively more memory becomes available. Our configuration uses multiple client machines for the tracing and one server to estimate the radiance per pixel. Thereby, the effective utilization of multiple machines in the network is a challenging problem, due to potential periods of under-utilization toward the end of each iteration. We were able to increase the photon throughput significantly, compared to a straightforward out-of-core implementation, by running the photon and eye ray tracing in parallel. More specifically, we coalesce tracing jobs and trace photons from multiple iterations at the same time, which in turn allows to guarantee that a certain number of tracing jobs is present in the network at all times, keeping all machines busy. We show that our method traces in a GPU-based, distributed system 5 times more photons and 30 times more hit points than a standard CPU-based SPPM implementation. The main contributions of this paper are:

- the consistent extension of stochastic progressive photon mapping to out-of-core scenes based on an automatic scene subdivison,
- a client-server architecture that distributes the computation to the CPUs or GPUs of multiple machines in a local network,
- optimizations for processing and transporting photon and eye ray paths in the network,
- a comprehensive guideline for the implementation method (GPU/CPU/hybrid).

Being able to obtain photometrically correct global illumination on large scenes provides a ground truth that is of use to validate the correctness of approximations used in production rendering.

In the remainder, we first describe related work in Section 2. In Section 3, we describe our distributed and hybrid out-of-core extension of the CPU-based and GPU-based SPPM and afterwards elaborate on the details of its efficient implementation in Section 4. Finally, we evaluate and discuss our results in Section 5, including a guideline to decide when to use CPU-based, GPU-based or hybrid SPPM, and conclude and encourage further research in Section 6.

2 Related Work

2.1 On the Development of Photon Mapping

The original photon mapping [Jen96] is a two-step algorithm for solving the rendering equation [Kaj86], i.e., to integrate the irradiance, reflected by the BRDF toward the viewer. First, *photons* are traced from the light source, are reflected based on BRDFs, and are stored in a k-d tree when hitting a diffuse surface. In a second step, paths are traced from the eye through the pixels to generate *hit points*, at which the photon density is estimated via range query. This radiance estimate contains noise due to the finite number of photons and a bias due to the fixed search radius. For this reason, Hachisuka et al. [HOJ08] introduced progressive photon mapping (PPM), a consistent extension that iteratively generates new photons and only stores *photon statistics* instead of all photons. This allows to increase the number of photons while shrinking the search radius. It was further extended in [HJ09] to stochastic progressive photon mapping (SPPM), which generates new hit points in each iteration, in order to integrate the radiance over the area projected onto the pixel. Thereby, all hit points of a pixel share the statistics, which allows for an efficient computation of distributed ray tracing effects like depth of field and glossy materials.

Several approaches exist to implement photon mapping on the GPU, cf. Ritschel et al. [RDGK12]. Among those, Hachisuka and Jensen [HJ10] suggested to store the photons of the current iteration using stochastic spatial hashing. There, the storage index is obtained by binning the photon position into a regular cell grid and by mapping the cell ID via hashing function to a memory address. Among the photons that were binned into a cell, only one is chosen for storage by Monte Carlo sampling. Due to the random writing order, the photons can overwrite each other and the selection probability is aquired by atomically counting the collisions. For the update of photon statistics, photons in cells inside the search radius around the hit point are considered. The whole procedure for GPU-based SPPM is visualized in Figure 2.

Knaus and Zwicker [KZ11] showed how to reduce the memory requirements by sharing statistics among all pixels. Recent work introduced more robust combinations of SPPM with bi-directional path tracing [HPJ12, GKDS12] that importance sample the full light path, i.e., multiple importance sampling. An adaptive PPM by Kaplanyan and Dachsbacher [KD13] proved better convergence rates by estimating the optimal search radius.

© 2014 The Author(s)
Computer Graphics Forum © 2014 The Eurographics Association and John Wiley & Sons Ltd.

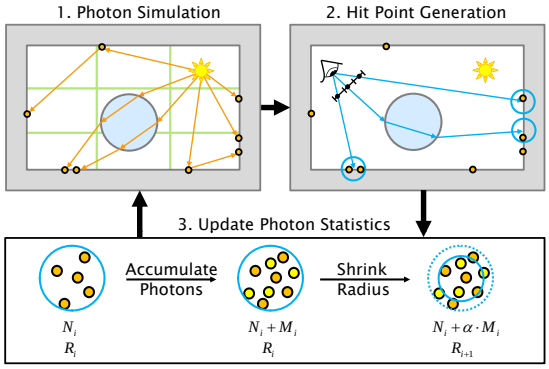

Figure 2: *GPU-based SPPM consists of three repeating steps: First, photons are traced from the light source and are stored using spatial hashing (shown as green grid). In the second step, hit points are traced from the eye. In the third step, photons are collected at the hit points in a search radius and the photon statistics are updated.*

2.2 Out-of-Core Global Illumination

Over the years, most standard global illumination algorithms were extended to out-of-core scenes, including radiosity [TFFH94], ray tracing [PKGH97, CLF*03, YM06], path tracing [BBS*09, ENSB13] and many-light techniques [WHY*13]. In production rendering, hybrid combinations of out-of-core GPU-based ray tracing with precomputed radiance transfer [SKS02], as in Pantaleoni et al. [PFHA10], and point-based approximations as in Kontkanen et al. [KTO11] proved useful. Photon mapping—at this point still inconsistent—was regarded by Christensen and Batali [CB04] for large out-of-core production scenes. They proposed to agglomerate the irradiance at all photon positions in regular octrees, assembling an irradiance atlas, which is designed to enable efficient caching. Lighting is done by final gathering and irradiance interpolation in the atlas. Later, Fradin et al. [FMH05] used a system based on cells and portals to distribute photons in large buildings. They manually divided the scene into rooms connected by portals, traced photons in the rooms successively and recorded photons in portals to be continued later in adjacent rooms. They computed indirect illumination by photon mapping and casted rays to compute the direct lighting from light sources visible from the point at hand. Since a gathering kernel with a constant size was used for the photon density estimation, discernible artifacts occurred at portal transitions.

The computation of caustic maps, as a part of the rendering solution, was distributed in commodity clusters. For an interactive application, Wald et al. [WKB*02] proposed a screen space subdivision of the workload for scenes fitting into the memory of each machine. Their technique uses instant radiosity [Kel97] to compute diffuse interreflections and additionally constructs a caustic photon map for rendering caustics. Kato and Saito [KS02] published results of their parallel global illumination renderer Kilauea, which utilizes a PC cluster consisting of 16 nodes. They combined a Monte Carlo-based ray tracer with final gathering on a photon map. The scene size, however, is limited to the memory available in the cluster. So, we observe that none of the existing photon mappers attains a consistent solution for out-of-core scenes.

3 Distributed Out-of-Core SPPM

In the extension of photon mapping [HJ09, HJ10] to out-of-core scenes by portal-based ray tracing [FMH05], we face two main challenges: *scene loading latency* and *under-utilization of resources*. The first issue is due to incoherent bouncing of photons in the scene, causing frequent loading of geometry data at highest detail from disk. A solution to reduce the scene loading operations is to increase the available memory. An economic option—and the one we chose—is to invest in a cheap commodity cluster, as in [KS02]. The under-utilization of resources occurs in the end of each tracing iteration, since we usually must wait until the last rays have finished. In previous methods, the photon and hit point tracing were executed alternately. This is especially wasteful if only a few rays are bouncing between scene parts. Not only individual CPUs or multi-processors on the GPUs, but even worse entire machines start idling. Our solution to this problem is to parallelize the tracing of photons and hit points, so that (a) workload can be efficiently distributed across the network, and (b) scene loading operations are minimized.

3.1 Overview

An overview of our out-of-core simulation is shown in Figure 3: In a pre-process, the large scene is automatically subdivided into c smaller *chunks* that fit into the system's memory, i.e., main memory or video memory depending on the chosen architecture. Similar to Fradin et al. [FMH05], the chunks are separated by *portals*. For reference, we first explain a naïve out-of-core SPPM implementation on a single machine that suffers from the aforementioned problems. Afterwards, we introduce our distributed solution.

Single Machine On a *single* machine, the three repeating steps of GPU-based SPPM (Figure 2) are processed *se-*

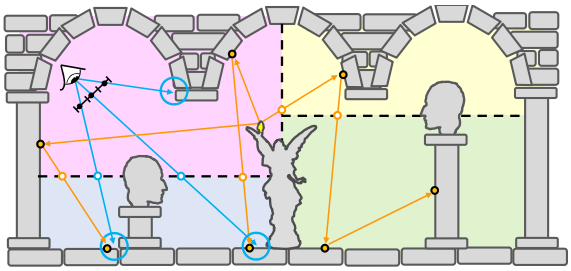

Figure 3: *Out-of-core SPPM: A large scene is subdivided into chunks of equal memory, separated by portals (dashed lines). When a photon or eye ray hits a portal, it is temporally stored and continued when the adjacent chunk is activated.*

© 2014 The Author(s)
Computer Graphics Forum © 2014 The Eurographics Association and John Wiley & Sons Ltd.

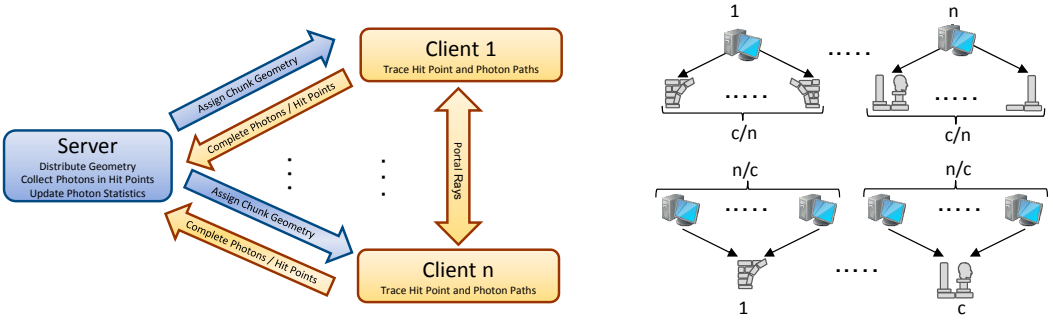

Figure 4: *Left: Client-Server Model. Right: Distribution of c geometry chunks to n client machines. In case of more geometry chunks than client machines ($c > n$), a fixed set of c/n chunks is assigned to each client and processed in a round-robin fashion (top right). If more client machines than geometry chunks are available ($c < n$), multiple clients use the same, fixed chunk geometry for tracing (bottom right).*

quentially. One chunk after another is loaded for tracing the photon and afterwards hit point rays. Whenever a ray hits a portal, it is temporally stored and continued later when the adjacent chunk gets activated. The CPU-based implementation uses traditional k-d trees for photon storage and range queries.

Network In our *network* approach, the chunks are processed *in parallel*. To distribute the work across multiple machines, we use one server and n clients, as shown in Figure 4. Thereby, we support the following four configurations in our distributed out-of-core SPPM (DOSPPM):

- CPU-based DOSPPM (clients and server use CPU),
- GPU-based DOSPPM (clients and server use GPU),
- Hybrid GC (GPU-based clients, CPU-based server),
- Hybrid GS (GPU-based server, CPU-based clients).

The scheduler of the server assigns chunks to the available client machines. Each client loads one of its assigned chunks of scene geometry and traces photon and eye ray paths inside this chunk—either on the GPU or CPU. If multiple chunks are assigned to a client, the client unloads the current chunk and loads the next chunk when all its jobs are processed. Whenever a ray hits a portal, it is forwarded to the client responsible for the adjacent chunk to be continued there. Deposited photons and hit points of eye rays are sent back to the server. Depending on the chosen architecture, the arriving photons are either stored stochastically in a spatial hash map on the GPU [HJ10], or in a k-d tree in main memory (CPU-based). The photon statistics are updated when all hit points of the current iteration arrived at the server. In this setup, we are able to trace eye rays and multiple photon iterations *simultaneously*. In fact, we emit new photons if the current number of pending photon tracing jobs in the network falls below a certain threshold (in our experiments 1k jobs), to keep all machines busy at all times. Typically, SPPM carries out two sampling processes: a sampling of the area projected to the pixel by *emitting hit points* and a sampling of the flux at the hit points by *emitting photons*, which are both explained in the following.

3.2 Hit Point Sampling

The update procedure of the photon statistics is similar to the original, sequential SPPM [HJ09] and works as follows (Figure 2): Given N_i photons inside the search region S of a pixel after i iterations. In iteration $i+1$, M_i additional photons arrive in search radius R_i at the currently sampled hit point $\vec{x} \in S$. In sequential SPPM, these photons were emitted in iteration $i+1$. In our method, these are the photons that arrived most recently, originating in possibly different iterations. Thus, as a contribution, the radiance estimate treats photons of different iterations equally, which allows to trace them in parallel in a joined set. Then, similar to [HJ09], only a fraction $\alpha \in (0,1)$ of these new photons is considered:

$$N_{i+1}(S) = N_i(S) + \alpha M_i(\vec{x}). \tag{1}$$

Assuming a uniform photon distribution, the search radius is decreased to cover the new number of photons:

$$R_{i+1}(S) = R_i(S)\sqrt{\frac{N_i(S) + \alpha M_i(\vec{x})}{N_i(S) + M_i(\vec{x})}}. \tag{2}$$

Thereby, α steers the rate of radius reduction. The BRDF-weighted, total flux of the M_i photons found in iteration $i+1$ is given as:

$$\Phi_i(\vec{x},\vec{\omega}) = \sum_{p=1}^{M_i(\vec{x})} f_r(\vec{x},\vec{\omega},\vec{\omega}_p)\Phi_p(\vec{x}_p,\vec{\omega}_p), \tag{3}$$

with f_r being the BRDF (for viewing direction $\vec{\omega}$ and photon direction $\vec{\omega}_p$) and Φ_p being the photon flux. The value Φ_i is added to the total BRDF-weighted flux τ_{i+1} and both are corrected for the decreased search radius R_{i+1}:

$$\tau_{i+1}(S,\vec{\omega}) = (\tau_i(S,\vec{\omega}) + \Phi_i(\vec{x},\vec{\omega}))\frac{R_{i+1}(S)^2}{R_i(S)^2}. \tag{4}$$

(Assuming uniform photon distribution within R_{i+1}.) The final radiance L of a pixel is obtained by normalizing τ_i with the total number of photons emitted after i iterations $N_e(i)$:

$$L(S,\vec{\omega}) = \lim_{i\to\infty} \frac{\tau_i(S,\vec{\omega})}{N_e(i)\pi R_i(S)^2}. \tag{5}$$

© 2014 The Author(s)
Computer Graphics Forum © 2014 The Eurographics Association and John Wiley & Sons Ltd.

3.3 Photon Sampling

Our modifications to run SPPM efficiently in a network mainly concern the photon sampling. As shown in Eq. 3, SPPM uses an estimate of the flux to compute the radiance at $\vec{x}$. To describe the incoming flux at $\vec{x}$, we use the notion of a *flux map* $\Psi^{\vec{x}}_{\vec{x}_L,\omega_L} : \mathbb{R} \to \mathbb{R}$, which tells how much differential flux arrives at a differential area around position $\vec{x}$, when emitting differential flux at $\vec{x}_L$ in direction $\vec{\omega}_L$ (see Figure 5). This map implements the tracing and BRDF-dependent reflection of photons. The incoming flux $\Phi(\vec{x})$ is estimated by Monte Carlo integration of arriving flux over the emitters F:

$$\Phi(\vec{x}) = \int_F \Psi^{\vec{x}}_{\vec{x}_L,\vec{\omega}_L}(\mathrm{d}\Phi_L) \tag{6}$$

$$= \lim_{i\to\infty} \frac{1}{N_e(i)} \sum_{k=1}^{N_e(i)} \Psi^{\vec{x}}_{\vec{x}_L,\vec{\omega}_L}(\Phi_k) \tag{7}$$

$$\approx \frac{1}{N_e(i)} \sum_{k=1}^{N_e(i)} \Psi^{\vec{x}}_{\vec{x}_L,\vec{\omega}_L}(\Phi_k) \tag{8}$$

with $\mathrm{d}\Phi_L$ being the differential flux emitted at location $\vec{x}_L$ in direction $\vec{\omega}_L$, and Φ_k being the discrete version, i.e., the flux of a photon. In each iteration, photons are emitted from all light sources by importance sampling according to the radiant intensity. Due to the delay in the network, photons may arrive in a later iteration (see Figure 5 right). In contrast to [HJ09], the photons are subject to a permutation and they arrive in slightly different numbers. In a given iteration, this changes the statistical values, but in the limit, or when the yet started tracing processes are finished to store intermediate results, this does not influence consistency. The reason is that every photon is processed once, none is forgotten and the order is still independent of the hit point location. Eq. 8 integrates into the PPM radiance estimate at location $\vec{x}$, viewed from direction $\vec{\omega}$:

$$L(\vec{x},\vec{\omega}) = \lim_{i\to\infty} \frac{\tau_i(\vec{x},\vec{\omega})}{N_e(i)\pi R_i(\vec{x})^2}, \tag{9}$$

There, the BRDF-weighted accumulated flux $\tau_i(\vec{x},\vec{\omega})$ is used, which is approximately:

$$\hat{\tau}_i(\vec{x},\vec{\omega}) = \sum_{k=1}^{N_e(i)} f_r(\vec{x},\vec{\omega},\vec{\omega}_k)\,\Psi^{\vec{x}}_{\vec{x}_L,\vec{\omega}_L}(\Phi_k), \tag{10}$$

with $\tau_i(\vec{x},\vec{\omega}) \approx \hat{\tau}_i(\vec{x},\vec{\omega})$. In practice, the accumulated flux depends on the search radius, but in the limit, both approximations approach the same value by definition. As the SPPM radiance estimate (Eq. 5) is based on the PPM update by sharing photon statistics, our modification applies to SPPM, too. Since hit point sampling and photon sampling are separate processes, the photon and hit point iterations do not need to alternate. Thus, one contribution of our work is the decision to execute the photon tracing and hit point tracing in parallel.

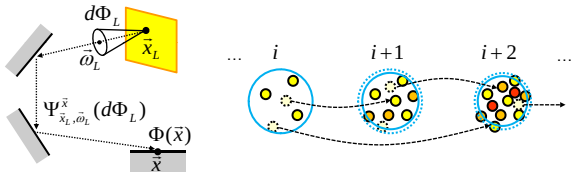

Figure 5: *Left: Flux map. Right: Photons emitted in iteration i may arrive in a later iteration, since they are still stored in portals when the photon statistics are updated.*

4 Details and Implementation

In this section, we explain the automatic subdivision of the scene, consistency regards and the optimizations needed for efficent computations in the network.

4.1 Automatic Scene Subdivision

In a preprocess, we automatically subdivide the scene into chunks that fit into the system's memory, i.e., the video memory of a GPU or the available main memory for CPU-based tracing, depending on the chosen architecture. Ideally, rays travelling through the scene should visit as few chunks as possible. We therefore aim to avoid chunks with a complicated border shape, as rays travelling in the scene from end to end might be subject to a large number of ray intersections. Thus, a subdivision into simple box-shaped chunks is desirable. We obtain such a subdivision by a simple five-stage algorithm, illustrated in Figure 6: First, we obtain the polygon count for each cell of a uniform grid of user-defined size by binning all polygons into the grid (similar to [PFHA10]; for our test scenes, the grid dimension is listed later in Section 5). The second step is a top-down construction of a k-d tree. For each cell, we inspect all three axes and select the split position which minimizes the difference in the polygon counts of the two halves. We proceed the splitting recursively until the chunks fit into the available memory. Third, we place portals (i.e., virtual quads) between neighboring chunks to mark transitions. Fourth, we insert the polygons into the chunks. If a polygon intersects a portal plane, it is inserted in both chunks. Fifth, we build a data structure for each chunk to accelerate the ray tracing, i.e., a SBVH tree [SFD09], as it performed best in our experiments. Figure 6 (right) shows a color-coded subdivision of a scene. This simple, yet efficient subdivision scheme leads to box-shaped chunks of roughly equal size.

4.2 Out-of-Core Radiance Estimates

The portal-based, sequential chunk-by-chunk processing in tracing photons and eye rays is similar to Fradin et al. [FMH05], though we aim for a *consistent* solution. The implementation of Fradin et al. contained an additional bias at the portals, since hit points did not account for the photons inside adjacent chunks. Although the query radius is shrinking progressively in PPM, the resulting bias remains visible in many practical cases, as shown in [KD13]. Depending on the chosen architecture—the GPU or the CPU—different

© 2014 The Author(s)
Computer Graphics Forum © 2014 The Eurographics Association and John Wiley & Sons Ltd.

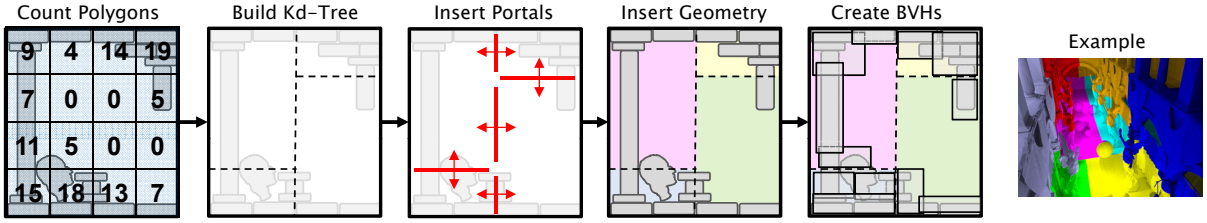

Figure 6: *Left: Subdivision pipeline. Right: Visualization of the subdivision of the CryTek Sponza scene into eight chunks.*

options arise for estimating the radiance on the server side. If the CPU is chosen, we construct a *unified* k-d tree to which photons from all chunks are added. Thus, each range query has access to photons from all chunks and no seams arise at portals. In a GPU setting, we circumvent the problem of visible seams at the portals by using a *single* spatial hash map that is confined by the bounding box of *all hit points*. The bounding box is extended by the initial search radius in all directions to ensure that all relevant photons are collected. This means that photons in a cell of the hash map can be located in different chunks of geometry. Therefore, a hit point always collects photons located in all chunks and no additional bias appears at the transition region (see Figure 7). Note that the stochastic spatial hash map requires a *constant* amount of storage for storing the photons.

In addition, it is possible to subdivide the spatial hash map into *multiple* smaller maps and to update the photon statistics in each map separately. Thereby, the bounding boxes of the maps *must overlap* by extending the box by the initial search radius. Thus, a photon can be contained in multiple maps. A hit point, however, must be contained in *exactly one* map by considering the non-overlapping, original box size. Doing so allows to obtain less hash collisions and therefore faster convergence, but at the cost of more computationally expensive photon statistics updates.

4.3 Batching and Culling

To further improve the performance of our photon mappers we implemented several optimizations. Two of them are specifically designed to increase the utilization of GPU multi-processors, these are: *client batching* and *server batching*. The third optimization improves both the CPU and GPU implementation, i.e., *photon culling*. In the following, we explain each in turn.

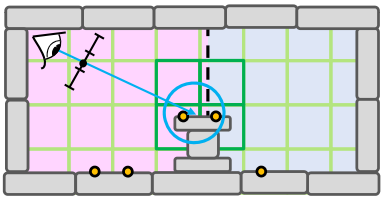

Figure 7: *Correct portal transitions (GPU-based): A hit point located close to a portal collects all photons inside the search region. Since the spatial hash map stores photons of the whole scene, photons from both chunks are located.*

Client Batching On the client side, all incoming ray tracing jobs are *batched*, i.e., coalesced to form larger packages that better utilize the GPU in their parallel processing. An example for photon rays is shown in Figure 8. Eye rays are batched likewise. This is especially beneficial for high chunk numbers and at a high number of ray indirections, since ray packages split when they enter different chunks, i.e., one package is sent to every visited neighbor chunk. (Unbatched, the number of ray packages grows exponentially.)

Server Batching The server collects photons in batches, too, before carrying out a photon statistics update. For this, the server awaits the arrival of a certain number of photons. (In our experiments we chose 1M photons.) This value is a trade-off between barely-filled hash maps, causing underutilization of the GPU, and a high number of hash collisions that cause loss of variations, i.e., slower convergence. At the end of the simulation, photon emission is stopped and the arrival of all remaining jobs is awaited. The final batch is processed even if it is not yet completely filled.

Photon Culling For the update of photon statistics, we are only interested in photons that are located in the search radius of at least one hit point. To reject unused photons before sending them to the server, we conservatively cull them using the bounding box of all yet found hit points, extended by the maximum search radius in all directions. More specifically, we cull the photons already on-the-fly on the client's GPU to prevent the memory transfer to RAM, or in the intersection handling code on the CPU, respectively. Photons outside this region can safely be ignored, since the box converges to the true bounding box in the limit (see Figure 9). Initial errors due to missing photons disappear over time. In doing so, we reduce the network traffic to the server. The

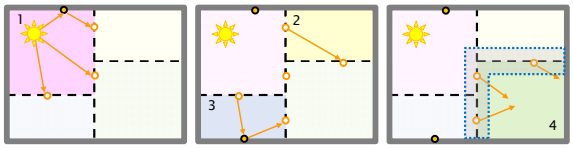

Figure 8: *Photon batching example: A photon job is emitted in chunk 1. The portal photons are traced in smaller jobs in chunks 2 and 3. When chunk 4 gets activated, all incoming portal photons are coalesced into a single tracing job.*

© 2014 The Author(s)
Computer Graphics Forum © 2014 The Eurographics Association and John Wiley & Sons Ltd.

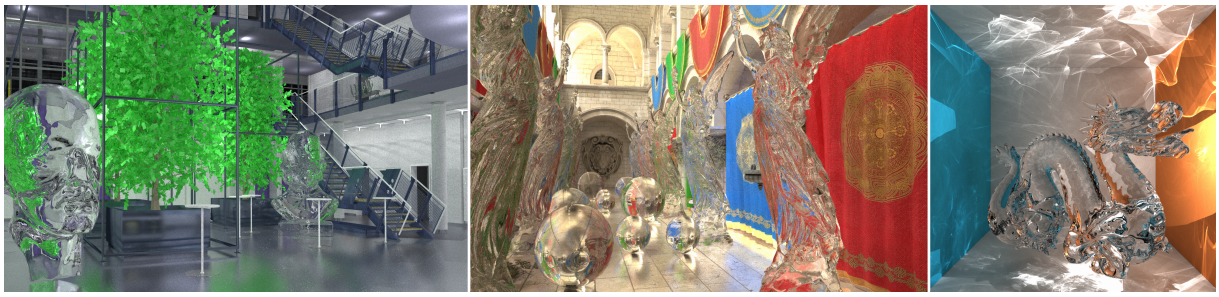

Figure 10: *Our test scenes: MPI building (70M triangles, 129.0h), CryTek Sponza (40M triangles, 82.4h) and Cornell Dragon (102K triangles, 18.2h)*

current hit point bounding box is included in a photon emission job and is expanded when photon tracing jobs are joined to avoid broadcasts by the server that are otherwise needed to report the new box to the clients. Culling photons is especially useful if the camera sees only a fraction of a scene.

5 Results

We obtained our results using a cluster of 9 equal machines (1 server and 8 clients) with a 1 GBit/s connection. Each machine is equipped with an Intel Xeon X3480 CPU with 3 GHz, 16 GB RAM and an Nvidia GTX 460 GPU with 1 GB VRAM. We used Nvidia's GPU ray tracing engine OptiX [PBD*10] for the intersection tests on the GPU and a single-threaded SSE optimized ray tracer on the CPU. For the communication between the machines we used the message passing interface MPICH2 [Arg].

We evaluated our method with several test scenes (see Figures 1(right) and 10), for which scene characteristics are summarized in Table 1. Our smallest scene is the Stanford Dragon in a Cornell box, and among our test scenes it is the only one that fits entirely into GPU memory. We used it for comparisons with standard SPPM and for parameter studies (number of chunks and tracers, and architecture studies). The CryTek Sponza scene is test ground for distributed ray tracing effects: It contains ten Stanford Lucys (each 4 M triangles) and many glossy materials. In the MPI building we evaluated the behavior of our method in case of many light sources (here, 623). Moreover, it is the largest scene that our GPU cluster can load completely into the distributed VRAM. For our last and largest test scene, we added 200 glass Stanford Lucys (each 100 k triangles) to the MPI building, resulting in a scene that is only loadable by a single machine into RAM so that we can draw comparisons to standard CPU-based SPPM.

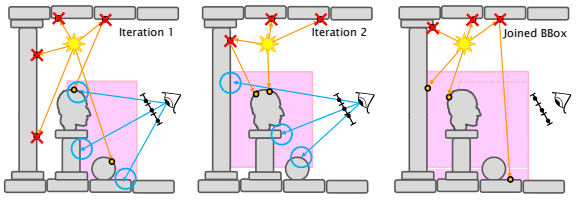

Figure 9: *Photon culling: Photons outside the common hit point bounding box are not transferred back to the server.*

In all scenes, the time spent on scene subdivision and SBVH construction was orders of magnitude smaller than the rendering time. In the MPI building, the subdivision took longer, since large triangles overlapped more bins, which therefore needed to be tested. The SBVH construction, on the other hand, was faster, as the insertion needed less cuts.

5.1 Comparison to Sequential SPPM

By using the scene that still fits into the memory of one GPU (i.e., the Dragon scene), we compare GPU-based and CPU-based SPPM with our out-of-core extension on a single machine (OSPPM), shown in Figure 11. In general, we list the throughput of photons and hit points per second, divided by the number of participating machines, i.e., the throughput per machine second (Ma Sec). It can be seen that the CPU-based method (throughput multiplied by four to account for a multi-threaded implementation) scales far better than the GPU-based method. The GPU, however, reaches far better

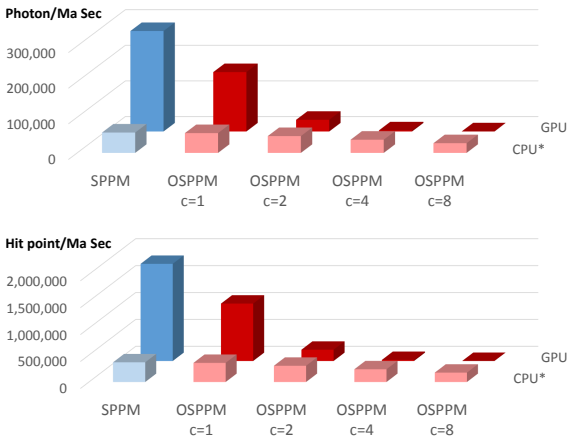

Figure 11: *Comparison in the Dragon scene of GPU-based SPPM (●) with OSPPM (●) and CPU-based SPPM (●) with OSPPM (●). The CPU* marks that the CPU timings are conservatively multiplied by four to account for multi-threading.*

© 2014 The Author(s)
Computer Graphics Forum © 2014 The Eurographics Association and John Wiley & Sons Ltd.

Scene	Triangles	Subdivision (bins)	BVH building & size (GPU/CPU)		Lights	Photon/It.	Resolution
Dragon	102 k	2 sec (4 k)	14 sec / 12 MB	2 sec / 14 MB	1	100 k	800×800
Sponza	40 M	2.3 min (262 k)	19 min / 2.9 GB	46 sec / 5.3 GB	5	500 k	960×600
MPI	70 M	20 min (262 k)	3 min / 4.8 GB	2 min / 9.0 GB	623	10 M	1720×1060
MPI Lucy	90 M	48 min (262 k)	–	3 min / 12 GB	623	10 M	1720×1060

Table 1: *Table showing for the main test scenes the complexity (number of triangles), timings for scene subdivision and BVH building, the number of lights, the number of photons emitted in one iteration and the viewport resolution. All scenes were subdivided into 8 chunks and were traced on 8 client machines. The spatial hash map had a resolution of* $512 \times 128 \times 512$ *and we traced photons up to ten indirections.*

peak performance at a small number of chunks. In fact, if the scene fits into the VRAM of a single GPU, standard GPU-based SPPM is the fastest method and therefore advisable. The overhead arising by the copying of photons and hit points between VRAM and RAM can be seen at the throughput of GPU-based OSPPM ($c = 1$), compared to SPPM (drop by 41 %). The performance of GPU-based OSPPM drops rapidly when increasing the number of chunks, because of the high number of portal transitions and the switching to the ray tracing acceleration data structure of the next chunk. The asynchronous loading of the next chunks is still not fast enough to prevent the GPU from idling, though it benefits the CPU-based method, as it loads faster.

Additionally, we compare scene subdivisions into several c chunks and numbers of tracer machines n in our distributed out-of-core approach (DOSPPM) for all possible architecture choices, i.e., GPU-based, Hybrid GS, Hybrid CS, and CPU-based, shown in Figure 12. For the distributed approaches, mere scaling of timings cannot account for multithreading, as the tracing rate and the network bandwidth depend on each other. In the following, timings are w.r.t. a single-threaded implementation. It can be seen that in nearly all configurations an increase in the number of chunks and tracers ($c = n$) leads to a loss of performance due to portal overhead. For this reason, dividing into as few chunks as necessary is advisable. Since hit point iterations are not parallelized, their throughput further drops. Clearly, the GPU-based method achieves the best performance, whereas the CPU-based method performed worst. The throughput of the hybrid methods indicates that the server is the bottleneck, as the choice of GPU or CPU on the clients has a small effect.

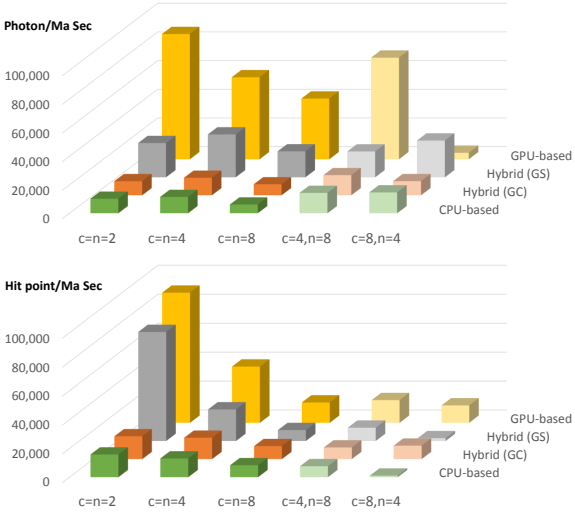

Figure 12: *Comparison in the Dragon scene of architecture options for DOSPPM, i.e., purely CPU-based (●●), Hybrid GC that is GPU-based clients and a CPU-based server (●●), Hybrid GS that is a GPU-based server and CPU-based clients (●●), and purely GPU-based (●●). The throughputs are given for various numbers of chunks c and tracers n.*

Doubling the number of tracers from $n = 4$ to $n = 8$ while keeping the number of chunks constant ($c = 4$) increases the overall throughput of photons and hit points. In an $n < c$ setting, overhead arises due to swapping of scene data, but the performance still increases if more tracer machines are used: Four tracers ($n = 4$), compared to one tracer ($n = 1$), increase the photon throughput for 8 chunks ($c = 8$) by factor 14.9 (GPU) and 2.1 (CPU). The hit point throughput increases by factor 6.1 (GPU) or drops by factor 0.2 (CPU), as network workload arises that is not compensated, since hit point iterations are traced successively.

Based on the results from Figures 11 and 12, Table 2 shows the speedup attained in the Dragon scene by using GPU-based DOSPPM over GPU-based OSPPM for several scene subdivisions. While a small subdivision introduces some overhead, using more chunks leads to an up to 138 times speedup for photons and 7 times speedup for hit points. Similarly, we measured the speedup factor for our CPU-based implementation. However, as Figure 11 suggests, the CPU-based OSPPM method barely loses performance, since the asynchronous loading of the chunks hides most of the

	# Chunks	2	4	8
Dragon, GPU	Photons	2.67	18.20	138.7
	Hit points	0.43	1.93	7.19
Dragon, CPU	Photons	0.86	1.25	0.91
	Hit points	0.21	0.22	0.19
MPI Lucy, CPU	Photons	0.61	0.64	4.46
	Hit points	5.55	3.52	11.67

Table 2: *Tracing speedup factor attained by using DOSPPM compared with OSPPM. Listed for GPU-based and CPU-based runs in the Dragon scene and a larger run in the MPI Lucy scene. One chunk per tracer was used, i.e., $c = n$.*

© 2014 The Author(s)
Computer Graphics Forum © 2014 The Eurographics Association and John Wiley & Sons Ltd.

Scene	Photon/Ma Sec	Hit point/Ma Sec	GPU load (srv/cln)	Bandwidth (srv/cln)
Dragon	40,469.35	13,793.10	14 % / 93 %	45 % / 21 %
Sponza	12,727.01	3,047.62	6 % / 86 %	20 % / 7.5 %
MPI	24,818.63	27,711.34	7 % / 87 %	19 % / 9.8 %
MPI Lucy	Does not fit into distributed GPU memory.			
Scene	Photon/Ma Sec	Hit point/Ma Sec	CPU load (srv/cln)	Bandwidth (srv/cln)
Dragon	6,137.99	8,315.41	47 % / 52 %	14 % / 8.9 %
Sponza	5,011.82	1,823.51	52 % / 47 %	17 % / 4.0 %
MPI	1,924.15	1,020.54	35 % / 86 %	6 % / 2.5 %
MPI Lucy	2,053.54	978.56	35 % / 89 %	10 % / 4.1 %

Table 3: *Table showing the utilization of our system running on GPUs (top) or CPUs (bottom), the required network bandwidth and the achieved photon and eye ray throughput (1 server, 8 clients, 8 chunks).*

portal loading overhead. In larger scenes, for instance the MPI Lucy scene, asynchonous scene loading operations cannot be hidden. Thus, our distributed approach that reduces the loading operations starts paying off in a CPU setting as well. Though, the improvement is not as drastical as in the GPU-based implementation, i.e., a speedup of factor 4.46 for photons and 11.67 for hit points in an $c = n = 8$ setting.

5.2 Performance and Hardware Utilization

We list the performance, measured for 8 clients and 8 chunks, in Table 3 for all test scenes, using both a GPU-based and a CPU-based implementation. Moreover, it contains both for server and clients the average CPU/GPU load and the arising bandwidth utilization. Both the CPU/GPU and network utilization depend on each other, since a slowed transmission means less work arriving at the clients.

GPU implementation The GPU utilization on the clients is quite high (86-93 %), though there is still room for improvements, e.g., by a better pipelining of the memory I/O and tracing. In all cases, the server's GPU utilization was lower (6-14 %). One option to fill the idle times is to spend time on more accurate radiance estimates, i.e., by subdividing the spatial hash map as described in Section 4.2. The network utilization, in our experiments never higher than 45 %, becomes critical if saturation is approached. It depends on a number of factors: the camera position (i.e., photon culling), the scene (materials, geometry arrangement, light sources), the scene subdivision (imbalance of workload) and the number of photons and hit points (viewport resolution).

CPU implementation The CPU is—as expected—in most cases slower than the GPU. The CPU clients spend 52-89 % of the time on tracing, which is still improvable. This is because the processing rate is lower and individual machines tend to idle, when waiting for the swamped clients to process their jobs. A possible next step is to further optimize the processing performance.

Conclusion The previous two sections showed that GPUs are favorable on tracer machines for scenes fitting into the distributed VRAM, due to their high processing rate. On larger scenes, a CPU tracer setup is advisable as it scales better, also because in a CPU setting fewer chunks are necessary due to more available RAM. The following sections further discuss scalability.

5.3 How Many Clients Can a Server Maintain?

Observing the individual timings of the simulation steps allows to estimate how many clients a single server can maintain. Thus, in Table 4, we show a breakdown of the individual processes of our simulation for the Dragon scene. Note that the GPU-based method is faster on both the client and the server.

GPU implementation A tracing command of 100k photon rays and the copy of the resulting 269 k deposited photons to the CPU take together 255.8 ms on a client. The server processes photons in batches of 1 M photons, thus $1\,\mathrm{M}/269\,\mathrm{k} = 3.72$ photon emissions are processed by one photon statistics update. Their tracing takes 951.6 ms in total. On the server, the transfer of the batched photons to the GPU and the photon statistic update take 93.3 ms in total. Hence, in the worst case (i.e., without inter-client communication) one server can maintain about $951.6/93.3 = 10.2$ GPU-based photon tracing machines, before it becomes a bottleneck. (Assuming the hit point tracing runs in parallel.)

CPU implementation In a CPU-based setting, 1 M pho-

Simulation process	Time in ms	
	GPU	CPU
Photon tracing (100k → 269k)	179.1	2,633.1
Copy photons: GPU to CPU	76.7	–
Copy photons: CPU to GPU	74.9	–
Hit point tracing (640k → 617k)	152.7	3,659.7
opy hit points: GPU to CPU	49.2	–
Copy hit points: CPU to GPU	46.7	–
Update Photon Statistics	18.4	792.5

Table 4: *Timing breakdown of the simulation processes in the Dragon scene on a single machine. Emitted and final counts are shown for photons and hit points. Note that the CPU implementation is single-threaded.*

© 2014 The Author(s)
Computer Graphics Forum © 2014 The Eurographics Association and John Wiley & Sons Ltd.

tons are traced in 9795.1 ms. The respective photon statistics update requires 792.5 ms. Thus, a CPU-based server maintains up to $9795.1/792.5 = 12.4$ CPU-based tracers.

Hybrid implementation The only hybrid setting of interest when considering large scenes is a GPU-based server and CPU-based client tracers, since CPU-based tracers show better scalability. In this setting, a GPU-based server maintains up to $9795.1/93.3 = 105$ CPU-based tracers.

Conclusion As our cluster comprises 8 tracer machines, neither in a pure GPU-based nor in a pure CPU-based setting, the server will become a bottleneck. In large clusters, GPU-based servers are advisable.

5.4 Scalability on Large Scenes

In the following, we study the scalability of our method on our largest scene. The results are normalized by the number of used machines and are summarized in Figure 13. It is noteable that the CPU-based approach scales nicely with the growing number of chunks and tracers ($c = n$). As expected by the server bottleneck estimate in Section 5.3, the server is not the limiting component for our 8 tracing machines. This is reflected by the roughly equal throughput of the GPU-based and CPU-based server. Since hit point iterations are traced sequentially, their throughput drops as anticipated when increasing the number of chunks and tracers.

Compared with a likewise subdivided MPI Lucy scene ($c = n = 8$), our CPU-based DOSPPM achieves a speedup over CPU-based OSPPM with asynchronous loading of scene data. Specifically, 1.12 times more photons and 2.92 times more hit points are processed. The speedup is improvable by splitting emission jobs, as especially the CPU-based

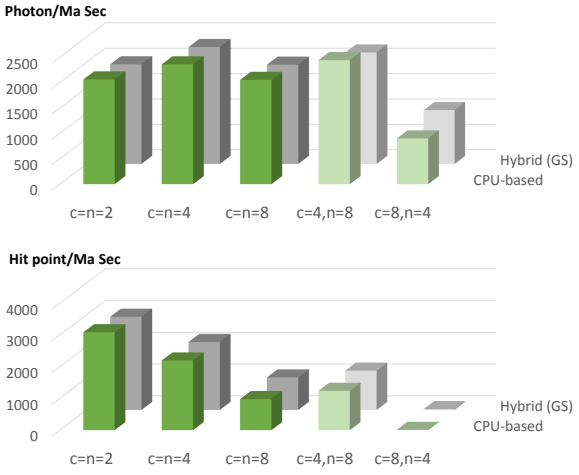

Figure 13: *Comparison in the MPI Lucy scene with DOSPPM for purely CPU-based (●●) and hybrid (GS) that is a GPU-based server and CPU-based clients (●●). The throughputs are given for various numbers of chunks c and tracers n.*

tracers spend a particularily long time on hit point emission, cf. Table 4, which is even longer in larger scenes. In that case, a client can quickly become a bottleneck.

In practice, one would subdivide the scene into as few chunks as possible. Thus, for the CPU-based approach, we can use fewer chunks due to more available memory and therefore have less portal overhead compared to the GPU-based method. When using the largest possible scene, i.e., the MPI building, to compare our GPU-based DOSPPM in the cluster with standard CPU-based SPPM on a single machine, we obtain a photon speedup of 4.82 and a hit point speedup of 29.5.

Conclusion For our number of tracers, the GPU and CPU are likewise appropriate to carry out photon statistics updates on the server. However, when working on larger clusters, a GPU-based server is highly advisable, since it scales far better as explained in Section 5.3. In such a setting, CPUs are recommended to carry out the tracing on the clients, since far less chunks are needed due to the amount of available memory, resulting in less portal overhead. Another issue is the arising bandwidth limitation, which can be delayed by compaction of the transmitted data, e.g., photons and hit points forwarded between machines.

5.5 Convergence and Optimizations

To verify the correctness of our method, we use the Dragon scene and compare the standard GPU-based SPPM to our method, using a subdivision into eight chunks (see Figure 14). Note that we obtain the same caustic pattern without visible seams, even though the portals cross the glass object.

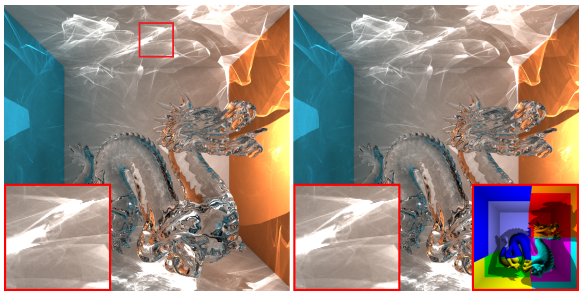

Figure 14: *Comparison of the standard GPU-based SPPM (left) with DOSPPM (right). The inset shows the subdivision into eight chunks.*

Figure 15 shows the impact of the optimizations in the GPU-based setting, by disabling them in turn, i.e., no batching of jobs on the clients, no batching of jobs on the server, alternately tracing of photons and eye rays (waiting for each pass to complete), compared to all optimizations enabled. For multiple chunks, the performance significantly drops if one of the optimizations is disabled. A CPU-based setting does not benefit from batching. The non-sequential tracing increases the photon rate by factor 1.1 ($c = n = 1$), 5.1 ($c = n = 2$), 4.3 ($c = n = 4$), and 4.6 ($c = n = 8$). The hit point

© 2014 The Author(s)
Computer Graphics Forum © 2014 The Eurographics Association and John Wiley & Sons Ltd.

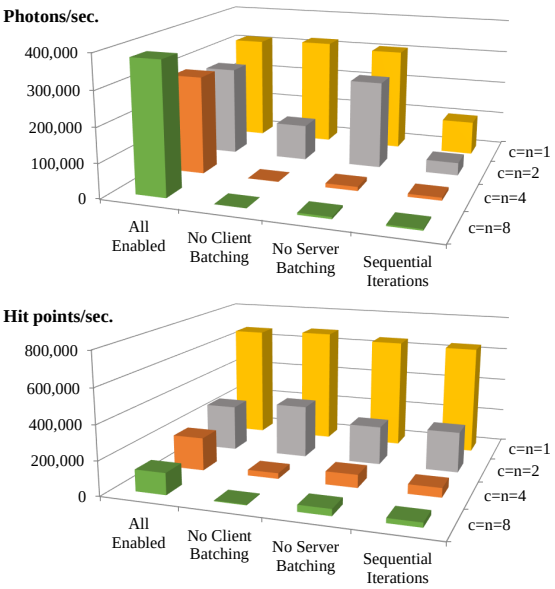

Figure 15: *Optimization impact in GPU-based DOSPPM.*

rate behaves as in the GPU setting. When activating photon culling, we observed a $10-13\,\%$ speedup for both MPI scenes, because of photons culled in invisible scene regions (rooms). The other test scenes did not show a significant speedup, since nearly all parts of the scenes were visible.

5.6 Potential Bottlenecks

We observed three classes of potential bottlenecks in our distributed implementation:

1. It is problematic if the server processes photons too slowly. Then, jobs are swamping the server's queue, whereas the clients start idling. The server cannot commit new jobs until it has processed the currently pending jobs. This happens when the batching of tracing jobs is disabled on the server side.
2. The transfer in the network may impose a bottleneck. In that case, machines are idling more frequently, since fewer jobs are processed in the same amount of time, as the jobs' lifetime is mostly spent on sending.
3. An individual tracer can become a bottleneck, if the scene geometry is either unequally distributed or tracing jobs have unequal workload. The latter might occur, if emission jobs are not split, e.g., the emission of camera rays, which can be observed in the CPU-based DOSPPM at large screen resolutions.

The optimal configuration is a sufficient network bandwidth where the server can process the incoming photons in time. This configuration was present in all test scenes when using one tracer per chunk. In the CPU-based setting, the client tracers were a bigger bottleneck than in the GPU-based setting, thus future optimization efforts will be spent there.

6 Conclusions and Future Work

In this paper, we described the first *consistent* out-of-core photon mapping approach and its efficient implementation based on both CPUs and GPUs in a distributed environment that *reduces scene loading operations* and prevents *periods of resource under-utilization*. With nine machines working together, we attained a significant speedup by factor 138 for photon processing and factor 7 for hit point processing in a distributed GPU-based setting, compared to nine machines doing GPU-based out-of-core tracing individually. In a CPU-based setting we achieve a speedup by factor 5 for photons and 12 for hit points. We demonstrated in our largest test scene that our method traces in a GPU-based, distributed system $5\times$ more photons and $30\times$ more hit points than standard CPU-based SPPM. We extended both the GPU-based and CPU-based SPPM to work more efficiently in an out-of-core scenario by first separating the tracing of photons and hit points and second batching and culling of tracing jobs. In our network, we employed a server for all photon statistics updates, and multiple clients to do the photon and hit point tracing in a portal-based, automatically acquired subdivision of the scene. We found that a hybrid combination of a GPU-based server and CPU-based clients shows the best scalability on large scenes and clusters.

We raise a number of challenges for future research. The maximum number of client machines in our network was eight. Evaluating the behavior in larger networks is therefore an avenue of future work, i.e., to inspect the network traffic. A single server can process only a fixed, scene-dependent number of clients. The investigation of a server hierarchy or array is therefore a possible next step. In case of a server array, we assume that broadcasting hit points to all servers and then letting clients randomly select the server to send photons to would be a better choice than using a screen-space subdivision, in which each server is responsible for a fraction of the viewport. The latter requires to either broadcast all photons to all servers, causing bandwidth issues, or to determine on the client side for every photon the servers that have hit points to which the photon contributes. This in turn requires the presence of all hit points and range queries on all clients. The culling of photons could be further improved by clustering of hit points to find better hull approximations. We also investigate the inclusion of volumetric illumination and spectral effects [KZ11]. Even though the scene subdivision is quite fast compared to the simulation, it can be further accelerated by using a hierarchy when inserting into the binning grid and it could as well be parallelized in the network. So far, we do not trace multiple hit point iterations at once. This extension requires to weight the hit points, as faster hit point paths are updated more frequently, which would result in a non-uniform sampling. Our automatic scene subdivision strives for an equal size of the chunks, but does not take into account the resulting workload. Considering the room topology of indoor scenes would be of help to guide

© 2014 The Author(s)
Computer Graphics Forum © 2014 The Eurographics Association and John Wiley & Sons Ltd.

the portal placement, cf. Fradin et al. [FML06] and Horna et al. [HDMB07]. Taking into account the visibility of light sources after a certain number of indirections could be used to cull lights whose photons never reach the viewed area.

Acknowledgements: The Dragon and Lucy models are courtesy of the Stanford University. Vlastimil Havran created the MPI Building. The Max Planck head is by AimAtShape. The Otto-von-Guericke head is courtesy of the University of Magdeburg. The CryTek Sponza scene was created by Frank Meinl and the Sibenik scene is by Marko Dabrovic. We thank the anonymous reviewers for their valuable comments. The work was sponsored by grant no. GR 3833/3-1 of the German Research Foundation (DFG).

References

[Arg] ARGONNE NATIONAL LABORATORY: MPICH2, http://www.mcs.anl.gov/research/projects/mpich2/. 7

[BBS*09] BUDGE B., BERNARDIN T., STUART J. A., SENGUPTA S., JOY K. I., OWENS J. D.: Out-of-core data management for path tracing on hybrid resources. *Computer Graphics Forum (Proc. Eurographics) 28*, 2 (2009), 385–396. 3

[CB04] CHRISTENSEN P. H., BATALI D.: An irradiance atlas for global illumination in complex production scenes. In *Proc. Eurographics Symposium on Rendering* (2004), pp. 133–141. 3

[CLF*03] CHRISTENSEN P. H., LAUR D. M., FONG J., WOOTEN W. L., BATALI D.: Ray differentials and multiresolution geometry caching for distribution ray tracing in complex scenes. *Computer Graphics Forum (Proc. Eurographics) 22*, 3 (2003), 543–552. 3

[ENSB13] EISENACHER C., NICHOLS G., SELLE A., BURLEY B.: Sorted deferred shading for production path tracing. *Computer Graphics Forum (Proc. Eurographics Symposium on Rendering) 32*, 4 (2013), 125–132. 3

[FMH05] FRADIN D., MENEVEAUX D., HORNA S.: Out-of-core photon mapping for large buildings. In *Proc. Eurographics Symposium on Rendering* (2005), pp. 65–72. 1, 2, 3, 5

[FML06] FRADIN D., MENEVEAUX D., LIENHARDT P.: A hierarchical topology-based model for handling complex indoor scenes. *Computer Graphics Forum 25*, 2 (2006), 149–162. 12

[GKDS12] GEORGIEV I., KŘIVÁNEK J., DAVIDOVIČ T., SLUSALLEK P.: Light transport simulation with vertex connection and merging. *ACM Trans. Graph. (Proc. SIGGRAPH Asia) 31*, 6 (Nov. 2012), 192:1–192:10. 2

[HDMB07] HORNA S., DAMIAND G., MENEVEAUX D., BERTRAND Y.: Building 3D indoor scenes topology from 2D architectural plans. In *Conference on Computer Graphics Theory and Applications* (Mar. 2007), pp. 37–44. 12

[HJ09] HACHISUKA T., JENSEN H. W.: Stochastic progressive photon mapping. *ACM Trans. Graph. (Proc. SIGGRAPH Asia) 28*, 5 (Dec. 2009), 141:1–141:8. 1, 2, 3, 4, 5

[HJ10] HACHISUKA T., JENSEN H. W.: Parallel progressive photon mapping on GPUs. In *SIGGRAPH Asia Sketches* (2010), ACM, pp. 54:1–54:1. 2, 3, 4

[HOJ08] HACHISUKA T., OGAKI S., JENSEN H. W.: Progressive photon mapping. *ACM Trans. Graph. (Proc. SIGGRAPH Asia) 27*, 5 (Dec. 2008), 130:1–130:8. 2

[HPJ12] HACHISUKA T., PANTALEONI J., JENSEN H. W.: A path space extension for robust light transport simulation. *ACM Trans. Graph. (Proc. SIGGRAPH Asia) 31*, 6 (2012), 191. 2

[Jen96] JENSEN H. W.: Global illumination using photon maps. In *Proc. Eurographics Workshop on Rendering Techniques* (1996), Springer-Verlag, pp. 21–30. 2

[Kaj86] KAJIYA J. T.: The rendering equation. *Computer Graphics (Proc. SIGGRAPH) 20*, 4 (Aug. 1986), 143–150.

[KD13] KAPLANYAN A. S., DACHSBACHER C.: Adaptive progressive photon mapping. *ACM Trans. Graph. 32*, 2 (2013), 16:1–16:13. 2

[Kel97] KELLER A.: Instant radiosity. In *Proc. SIGGRAPH* (1997), Annual Conference Series, pp. 49–56. 3

[KS02] KATO T., SAITO J.: "Kilauea": parallel global illumination renderer. In *Proc. Eurographics Workshop on Parallel Graphics and Visualization* (2002), pp. 7–16. 3

[KTO11] KONTKANEN J., TABELLION E., OVERBECK R. S.: Coherent out-of-core point-based global illumination. In *Proc. Eurographics Symposium on Rendering* (2011), pp. 1353–1360. 3

[KZ11] KNAUS C., ZWICKER M.: Progressive photon mapping: A probabilistic approach. *ACM Trans. Graph. 30*, 3 (May 2011), 25:1–25:13. 2, 11

[LW93] LAFORTUNE E. P., WILLEMS Y. D.: Bi-directional path tracing. *Proc. Compugraphics* (1993), 145–153. 1

[PBD*10] PARKER S. G., BIGLER J., DIETRICH A., FRIEDRICH H., HOBEROCK J., LUEBKE D., MCALLISTER D., MCGUIRE M., MORLEY K., ROBISON A., STICH M.: Optix: a general purpose ray tracing engine. *ACM Trans. Graph. (Proc. SIGGRAPH) 29* (July 2010), 66:1–66:13. 7

[PFHA10] PANTALEONI J., FASCIONE L., HILL M., AILA T.: Pantaray: fast ray-traced occlusion caching of massive scenes. *ACM Trans. Graph. (Proc. SIGGRAPH) 29* (July 2010), 37:1–37:10. 3, 5

[PKGH97] PHARR M., KOLB C., GERSHBEIN R., HANRAHAN P.: Rendering complex scenes with memory-coherent ray tracing. In *Proc. SIGGRAPH* (1997), Annual Conference Series, pp. 101–108. 3

[RDGK12] RITSCHEL T., DACHSBACHER C., GROSCH T., KAUTZ J.: The state of the art in interactive global illumination. *Computer Graphics Forum 31*, 1 (2012), 160–188. 2

[SFD09] STICH M., FRIEDRICH H., DIETRICH A.: Spatial splits in bounding volume hierarchies. In *Proc. High-Performance Graphics* (2009), pp. 7–11. 5

[SKS02] SLOAN P.-P., KAUTZ J., SNYDER J.: Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. *ACM Trans. Graph. (Proc. SIGGRAPH) 21*, 3 (July 2002), 527–536.

[TFFH94] TELLER S., FOWLER C., FUNKHOUSER T., HANRAHAN P.: Partitioning and ordering large radiosity computations. In *Proc. SIGGRAPH* (1994), Annual Conference Series, pp. 443–450. 3

[WHY*13] WANG R., HUO Y., YUAN Y., ZHOU K., HUA W., BAO H.: GPU-based out-of-core many-lights rendering. *ACM Trans. Graph. (Proc. SIGGRAPH Asia) 32*, 6 (2013), 210:1–210:10. 3

[WKB*02] WALD I., KOLLIG T., BENTHIN C., KELLER A., SLUSALLEK P.: Interactive global illumination using fast ray tracing. In *Proc. Eurographics Workshop on Rendering Techniques* (2002), Eurographics Association, pp. 15–24. 3

[YM06] YOON S.-E., MANOCHA D.: Cache-efficient layouts of bounding volume hierarchies. *Computer Graphics Forum 25*, 3 (2006), 507–516. 3

© 2014 The Author(s)
Computer Graphics Forum © 2014 The Eurographics Association and John Wiley & Sons Ltd.