Fourth Eurographics Workshop on Parallel Graphics and Visualization (2002)
D. Bartz, X. Pueyo, E. Reinhard (Editors)

"Kilauea" – Parallel Global Illumination Renderer

Toshi Kato† and Jun Saito

Square USA

Abstract

'Kilauea' is a revolutionary parallel renderer developed at Square USA. The goal of the R&D effort was to create a renderer which can compute global illumination on extremely complex scenes by using affordable PC cluster. This paper reports Kilauea's parallel processing methodology, implementation issues, and parallel performance.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Ray tracing

1. Introduction

The Kilauea Research Project took place in the R&D division of Square USA's Honolulu Studio from March 1998 till March 2002, to design and implement a groundbreaking high-end renderer. Two ultimate goals were set when the Kilauea Research Project began. The renderer aimed to:

1. Compute high quality images using global illumination;
2. Handle extremely complex and large scenes.

Compared to conventional renderers based on direct illumination models, achieving these goals evidently requires enormous amount of CPU and memory resources. Our solution to fulfill this requirement is to obtain the massive computational power from parallel processing using cost efficient PC Linux cluster consisting of multi-CPU machines interconnected by 100 BaseT Ethernet. Some of the reasons behind this decision are:

1. Ease in utilizing future hardware advancements
2. Flexibility in adjustment of computational resources

The benefits gained from the hardware advancements apply the same way even if multiple machines are used. Especially, the advancement in the networking environment will surely be the power to propel the use of PC clustering. By utilizing these trends in a clever way, the computational cost once thought to be intractable may become easily available, which in turn may give a rise to the new form of expression backed up by the groundbreaking image creation technology.

Hardware is constantly evolving. Faster and cheaper machines are pouring into the market everyday. However, the productivity of image rendering using only one machine is limited solely by the machine's performance. If more performance is desired, excessive investment must be made on the super-high-end hardware. Usually more than the highest possible hardware performance at that time is craved from the production. Thus, the promising methodology for the future is to combine the power of multiple cost-efficient, stable machines to satisfy the unlimited demand for computational resources. This solution also brings forth the way of gradual investment and long-time use of machines.

The architecture of such renderer is desired to prolong reliably for years. Therefore, the flexibility to allow incorporation of numerous additions and improvements is important. Furthermore, the renderer is intended to be the test bed for experimenting various rendering ideas, instead of specializing in creating certain images. That is, the renderer even needed to be flexible enough to incorporate unknown algorithms to be discovered in the future.

Carefully considering these premises, we have designed and developed a massively parallel global illumination renderer to run on a multi-CPU PC cluster – named Kilauea.

2. Technical overview

Kilauea renders multiple frames while the scene data stays resident in its memory. When Kilauea boots up, it displays a console prompt to accept commands to modify the scene data partially or entirely before starting the rendering of the next frame. The detailed rendering control commands thereafter are sent to this console either directly or through socket. Kilauea waits for commands until it is explicitly told to shut down the system.

† Now working for Rhythm & Hues Studios.

Everything inside Kilauea including ray tracing, shading, socket communication and command processing is designed to be processed in parallel. Shared memory parallel processing is implemented using Pthread to maximize the performance on multi-CPU systems.

All computations in Kilauea's distributed renderings using networked machines are done in the message passing manner. Data are processed in sequence just like the bucket brigade to allow the computation to continue over the network. Kilauea has its own implementation of the subset of MPI for the communication between multiple machines.

Monte Carlo ray tracing – Kilauea's base rendering algorithm – is simple yet powerful, and matches perfectly with parallel processing based on message passing. However, ray tracing requires all scene data to be loaded in the memory, which becomes an obstacle in achieving Kilauea's goal to render extremely large scenes. There are many proposed methods to overcome this problem, such as the one which pages in data from disk as required[2]. Kilauea's solution is to distribute the large scene data to multiple machines.

Photon mapping is used for the global illumination computation because the implementation of photon map on a Monte Carlo ray tracer is easy and the algorithm reliably works on any primitives which can be ray traced.

3. Distributed ray tracing engine

3.1. Guidelines to rendering algorithm

If the scene to be rendered is very simple and is small enough to fit in the memory of one machine, Kilauea simply sends exact copies of the entire scene data to multiple machines and executes ray tracing in parallel. In this case, the speed-up proportional to the number of machines is achieved. However, if the scene is too large to be stored in the memory of one machine, the ray tracing methodology must be carefully considered. Kilauea focuses on the following four points in consideration of the ray tracing algorithm:

1. Flexibility in describing shading computation
   Ideally, users should be able to describe the surface shading in the same way as in a sequential ray tracer.
2. Robustness to scene types
   An algorithm which performs well on all scene types is preferred over an algorithm whose performance is outstanding for certain scenes but critically slow on others.
3. Extensibility in supported primitives
   Rather than an algorithm specialized for certain primitives, an extensible algorithm which can support various primitive types is desired.
4. Simplicity in algorithm
   Keeping the algorithm simple and clean often eases code quality maintenance.

With these guidelines in mind, two rules for designing scene distribution and rendering algorithms are devised.

A. All scene data is to be loaded in memory
B. Ray data is passed between machines

The approach to swap out the scene data to disk requires exchanging of the order of ray tracing to improve the cache hit rate. This ordering may limit the type of shaders, which breaks the one of the guidelines. Thus, Kilauea took the approach to load the entire scene in memory.

Results of several experiments have shown that exchanging a part of the scene geometry data between machines as necessary inflicts too much communication overhead. This is because geometry data structure is complex and not suitable for partial transfers. As opposed to this, ray data are independent from each other and are easier to be transferred between machines.

3.2. Basic algorithm

Kilauea's ray tracing algorithm following the above mentioned guidelines under the distributed environment is explained. The algorithm is composed of two parts: scene distribution and distributed ray tracing.

3.2.1. Scene distribution and storage

Large scenes are distributed to and stored in multiple machines. The size of data to be held on each machine is determined by the physical memory it has, which in turn determines the number of minimum machines. Basically, the scene is distributed randomly to multiple machines in a primitive-based manner. As a result, the scene data held in one machine appears to be dispersed throughout the entire scene, instead of concentrated in some regions (Figure 1).

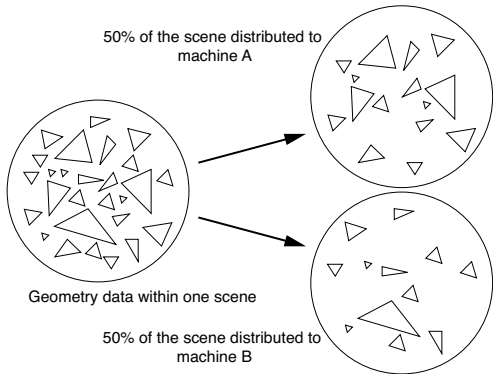


Figure 1: *Scene distribution*

Practically, primitive-based distribution can be too memory inefficient. For example, if a triangle-based distribution is performed on a triangle mesh data, many of the vertex data will not be shared. A better methodology used in Kilauea is to distribute in chunks of triangle meshes of up to certain size[5]. At each machine, the distributed data are stored in a

hierarchical uniform grid structure called accel grid to speed up ray tracing. This accel grid construction stage is executed in parallel on each machine.

3.2.2. Ray tracing on distributed scene

One ray tracing operation in the distributed environment is executed by first sending exact copies of a ray data originating on a certain machine to all machines with distributed scene data. For each machine, ray tracing is executed in its accel grid according to the received ray data. Eventually, the original machine receives ray tracing results from all machines with distributed scene data. The returned results are compared to find out the front-most intersection, which becomes the final ray tracing result (Figure 2).

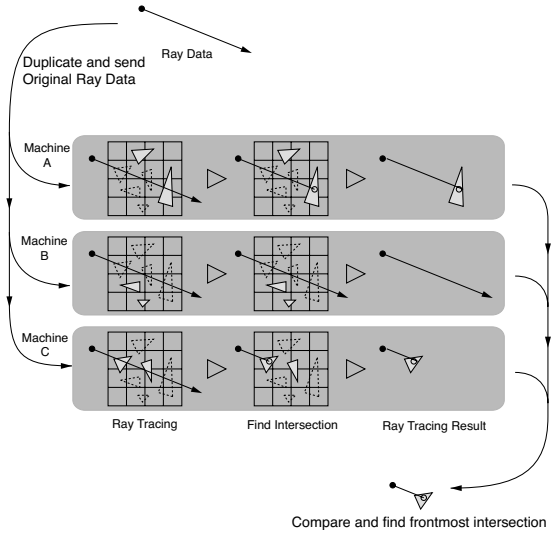


Figure 2: *Distributed ray tracing*

Let us assume that a scene is too large and needs to be distributed to three machines, as in Figure 2. During the actual rendering, numerous rays in the scene are processed in series. If there are six machines to distribute the scene, two sets of three machines to hold the entire scene can be prepared. In this case, numerous rays in the scene can be assigned to either one of the two sets to execute ray tracing in parallel and double the computation speed.

3.3. Parallel ray tracing

If the user can prepare multiple sets of machines for Kilauea to hold the entire scene data, ray tracing can be executed in parallel on each set independently.

First, the ray tracing is executed independently for each sample on the screen. Some of the screen rays will hit objects, where the shading computations start. The shading is processed in parallel with ray tracing. Depending on the shader, more rays such as shadow, reflection, and refraction rays may be shot. These rays after second generation are processed in parallel just like primary rays.

As a result, there will be a mixture of numerous rays from various generations during the rendering. Kilauea tries to process rays with higher generations first (see section 4.2). At the implementation level, this is handled by entering the ray data in the input queue of the ray tracing engine. The engine takes the queued data in series to process them. If there are multiple sets of machines to hold the scene data, the workload is balanced by letting them pull the queued data out by first-come-first-served basis.

3.4. Discussion on distributed ray tracing algorithm

Kilauea's distributed ray tracing approach behaves equally well, no matter where the origin or the direction of the ray is. In addition to classic ray tracing, Kilauea needs to ray trace photons and final gather rays, which makes meaningful sorting of the ray data by their positions or directions extremely difficult. Because of this property in the global illumination rendering, using the algorithm which exhibits consistent performance for all ray tracing requests is important. This consistency in turn brings simplicity in balancing the load of multiple CPUs.

Another characteristic to highlight is that all ray tracing requests can be processed with almost equal latency. This is because all ray tracings can be computed with the maximum of two data transmissions between machines. If the algorithm required unpredictable number of data transmissions between machines depending on the ray tracing request, the variation in the computation time complicates the load balancing of the entire system. The consistency in latency of the ray tracing computation is thus important from the viewpoint of keeping the implementation simple.

4. Shading engine

Kilauea's ray tracing approach accomplishes simplicity and robustness, though more consideration is needed when the shading computation comes in. Let us consider the case where the reflection ray needs to be traced during the shading computation on some object surface. Conventional sequential ray tracers would handle this by temporarily aborting the shading computation on the surface, trace the reflection ray, and if it hits a surface, a shading computation is executed. Using the computed color from the reflection ray, the aborted shading computation resumes. Usually this mechanism is just implemented as a simple recursive call.

However, this common practice in conventional renderers does not work in Kilauea because shading and ray tracing may be processed in parallel by different threads, or even by different machines. Ray tracing computation takes some time, causing the latency in getting the result. This makes the CPU to become idle during a shading computation which

shoots a reflection ray and waits for its result (Figure 3). One solution is to forbid users from writing a shader which directly uses the results of rays shot from within the shader. However, this breaks one of the guidelines to allow users to write new shaders as in sequential renderers.

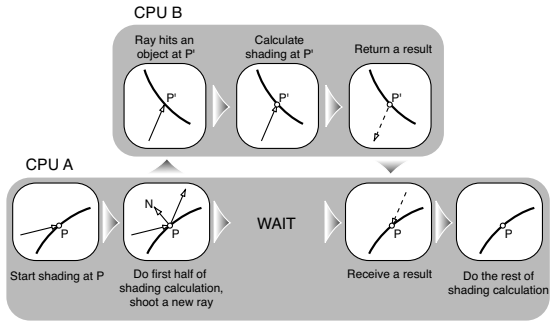


Figure 3: *Problem in parallel ray tracing*

The shading mechanism of Kilauea must be able to handle a long waiting period during the computation, which can be viewed as a complex queue structure. We have devised a parallel shading engine to process such complex queue structure, which allows shaders to freely shoot rays and incorporate their results in the final shading results, while keeping CPUs busy. This engine is called SPOT (Shading Parallel Object Task, pronounced *es-pot*) Engine.

4.1. SPOT Engine mechanism

If a ray needs to be shot during a shading computation, a ray tracing request is placed in the ray tracing engine and this shading computation halts until its result returns. To keep the CPUs busy, the engine starts the next shading computation. The engine keeps the CPUs constantly busy by processing the shading requests in series. While processing other shading requests, the result of the ray shot during the first shading will eventually be computed by the ray tracing engine. Using this result, the shading engine resumes the original shading computation. By computing the shading in this manner, CPUs will never be idle and all computation will be executed in parallel efficiently (Figure 4).

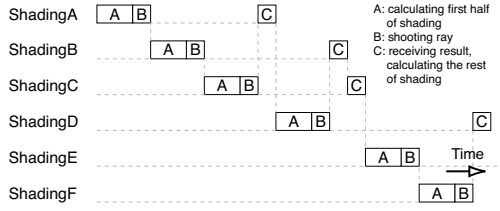


Figure 4: *Optimally scheduling shading to keep CPUs busy*

4.2. Priority rule

The shading mechanism proposed above has a possibility for facing a fatal flaw. As mentioned, SPOT Engine is essentially a huge queue processing system. If a shading computation aborts to wait for a ray tracing result, its state must be saved in the memory. In other words, as the delay for computing the ray tracing result becomes longer, more shading states must be stored in the memory for longer time. This potential inefficiency in memory usage can be critical.

On some occasions, the engine may easily run into the situation where the memory is exhausted by too many intermediate shading states in the system. For example, consider the case where the ray tracing engine processes primary rays in series and some of them hit a mirror object to produce reflection rays. The ray hitting the mirror object will generate a new reflection ray during the surface shading computation of the object, and stores an intermediate shading state in the memory. This shading computation goes into the waiting state until the reflection ray result comes back, and the shading engine proceeds to the next shading computation.

If the newly generated ray is stuck on the end of the input queue of the ray tracing engine, it will never be processed until all preceding ray tracing requests are computed. At this point, existing ray tracing requests in the queue are mostly from the enormous amount of primary rays. Many of these primary rays will end up hitting the mirror object as well, generating even more intermediate shading states. As a result, large portion of the memory will be filled by the shading states. In the worst scenario, all the physical memory may be filled by shading states, eventually resulting in thrashing. The process may even abort due to the memory allocation error in the end.

The memory inefficiency is minimized by prioritizing the processing of ray tracing depending on the generation of the ray. Ray generation is the number starting from zero and incremented every time a ray reflects or refracts. From the queued ray tracing requests, the ray tracing engine gives priority to higher generation rays. This successfully reduces the memory required for the intermediate shading states.

4.3. Shader implementation using SPOT

Kilauea has a special shader description method to automate the decision on where in the shader to go into the waiting state and where to resume the shading after receiving the ray tracing result. The shading computation is broken down into two parts: the part where the described operations can be processed sequentially and the part where the computation must wait for the ray tracing result. The part which can be executed sequentially is called SPOT. SPOTs may have multiple "slots" to accept external data inputs (Figure 5).

An SPOT consists of the definition of the function to represent its operation, and the data structure used by the operation. As seen on Figure 5, an SPOT has a condition of either

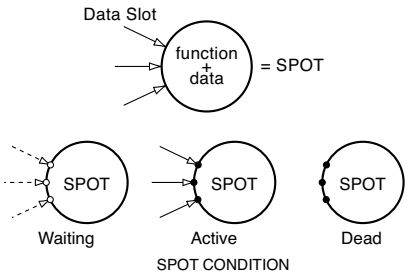


Figure 5: *SPOT*

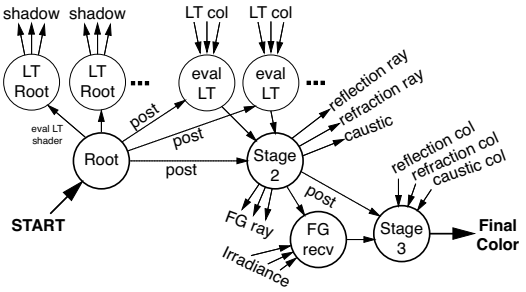


Figure 6: *Shader implementation by SPOT network*

WAIT, ACTIVE, or DEAD. The network of these SPOTs describes all shading computations.

An SPOT is first created with the WAIT condition. When all of its slots are filled, it automatically turns into the ACTIVE condition and the shading engine executes its defined function. The SPOT automatically turns into the DEAD condition after the execution of the function. The garbage collector of the shading engine automatically recycles the memory used by DEAD SPOTs.

A waiting SPOT is equivalent to the intermediate shading state saved in the memory. The mechanism to resume the aborted shading operation by returning the ray tracing result is realized by entering the result to the slot of the appropriate SPOT. If entering the ray tracing result fills the slot, the function defined by this SPOT is executed. Thus, this function becomes the resumed shading operation.

By describing the shading computation as the network of SPOTs, the shading engine can efficiently and automatically abort shading, store the intermediate shading state, and resume shading upon receiving the ray tracing result. However, because writing shaders using SPOT network requires good understanding of C++ coding and highly specific mechanisms of SPOT Engine, high-level shader description method called *shader template* is provided to ease the task. Shader writers can just fill in the necessary parts of the template without any working knowledge of parallel processing. Figure 6 shows the actual shader used in Kilauea described by the network of SPOTs. For more on Kilauea's shading engine, please refer to SIGGRAPH 2001 Practical Parallel Rendering course note[3].

5. Global illumination

5.1. Photon map

Kilauea computes global illumination using an algorithm called photon mapping[1]. Here are several reasons for choosing photon mapping:

1. Goes well with Kilauea's ray tracing engine.
   Because photon mapping is based on ray tracing, Kilauea's ray tracing engine can be used just as it is.
2. Is suitable for parallel processing.
   Each photon in the photon tracing stage is completely independent from each other and therefore photon mapping is inherently suitable for parallel processing.
3. Has an abundant potential for high-quality expression
   Photon mapping can handle both the specular and the diffuse global illumination components.
4. Supports various primitive types
   As long as the primitive can be ray traced, it can be used in photon mapping.

Photon mapping consists of two basic operations: photon tracing and photon lookup. Kilauea's approaches for parallelizing these two operations are explained below.

5.2. Parallel photon tracing

The behavior of the photons emitted from the light sources is computed in photon tracing. Numerous photons emitted from the light sources are completely independent from each other. Thus the photon tracing is computed in parallel. The same distributed ray tracing engine of Kilauea is used for the photon tracing. This results in the robust execution of photon tracing, by the same reasons discussed in the distributed ray tracing.

For the photons stored as the result of photon tracing, Kilauea considers two situations.

First, if the size of the final photon map is small enough to fit in the memory of one machine, the entire photon map data for the scene is constructed in all machines. Some consideration is necessary to create identical photon map on all machines at the end of the photon tracing. The photon data traced and stored on one machine is first stored in its photon map, and then distribute the same photon data to all machines. By executing this process on all machines, the identical photon map data for the entire scene will be generated on all machines eventually (Figure 7).

If the photon map is too large to fit in one machine, the entire data is divided to be shared by a set of several machines. The photons are distributed within the set randomly to avoid inconsistent spatial density. The entire photon map ends up

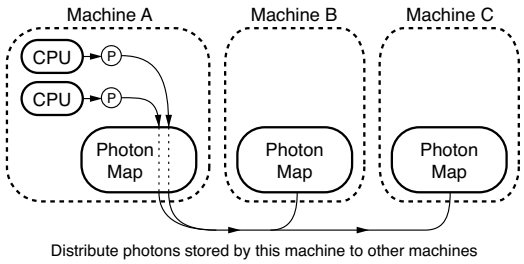


Figure 7: *Photon distribution*

being held by a set of machines. Accordingly, each machine will appear to have the photon map in low density, instead of having the photon map of certain spatial region (Figure 8).

With the same concept used in the case of the photon map fitting in one machine, the photon to be stored is distributed to other sets as well, to end up with the identical photon map on each set of machines.

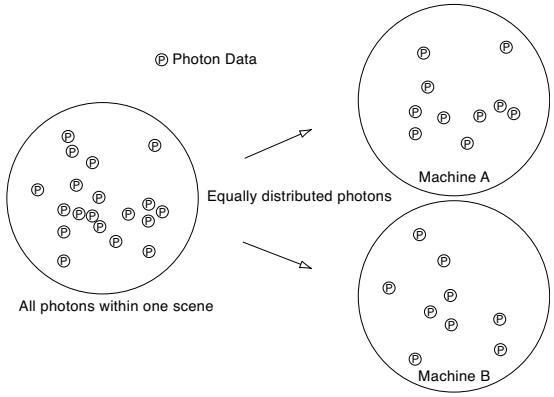


Figure 8: *Multiple photon map distribution*

5.3. Discussion on parallel photon tracing

Because photon tracing is algorithmically suitable for parallel processing, it exhibits excellent parallel performance under Kilauea's implementation. Kilauea's ray tracing engine reliably handles photon tracing even if the scene is too large and is distributed to multiple machines.

From the standpoint of distributed computing in Kilauea, photon storing operation is significantly different from ray tracing. Since photon data are transferred between machines, the size of the transferred data and the time needed for the communication become critical in photon tracing. If 700,000 photons are stored for a certain scene, their total size is approximately 35 MB in Kilauea. Transferring 35 MB of data between machines connected by 100 BaseT network takes about 5 to 6 seconds. Considering that a typical rendering takes over 10 minutes, this transfer time is acceptable. Algorithmic improvements such as packeting and compression may be considered, but with the rapid advancement of network speed, the transfer time will be even reduced to a negligible level without any effort.

5.4. Distributed photon lookup

Photon lookup operation is required in the shading computation. Basically, all machines will need to be involved in photon lookup.

If the generated photon map is small enough and fits in the memory of one machine, identical photon maps each representing the entire scene should have been already created on all machines. Therefore, the lookup operation in this case is extremely simple – just refer to the photon map that the machine has.

If the generated photon map is too large to fit in the memory of one machine and is shared among multiple machines, the lookup operation becomes more complex. The idea is basically the same as the distributed ray tracing. A lookup request is sent to all machines sharing the photon map. On each machine, the received lookup requests are processed by referring to the photon map it has. This stage is executed independently, which in turn means that all machines are used in parallel. All machines send their lookup results back to the original machine, where the results are composed to derive the final answer (Figure 9).

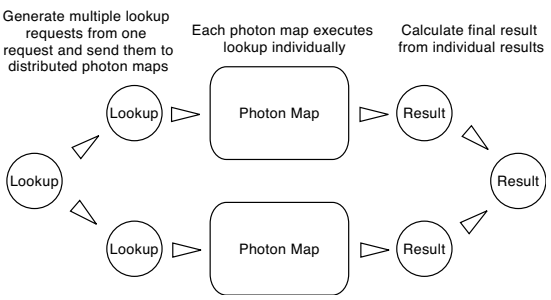


Figure 9: *Parallel photon lookup*

This composition is computed as follows:

1. The lookup in each photon map
$$P_m = \sum_{i=1}^{n/M} P_i$$
where M is the number of photon maps and n is the number of nearest photons to look up.
2. Combined irradiance
$$E_{total} = \frac{M \sum_{m=1}^{M} P_m}{\pi \sum_{m=1}^{M} r_m^2}$$
where r is the lookup radius.

Please refer to SIGGRAPH 2001 photon map course note[4] for more detailed discussion on parallel photon lookup.

5.5. Discussion on distributed photon lookup

Basically the size of the photon map can be thought separately from the scene geometry complexity and size. Instead, its size is determined by the complexity in the sense of global illumination. Considering that the size of one photon is about 50 bytes including the debug information in Kilauea, photon maps are small enough to fit in the memory of one machine in most cases. Except for extremely odd circumstances, there will be no need to share one photon map among several machines. However, Kilauea's parallel algorithm can reliably handle photon lookup even in such rare cases.

5.6. Speeding up final gathering

In the photon mapping, a surface shading computation may shoot final gather rays (Figure 10).

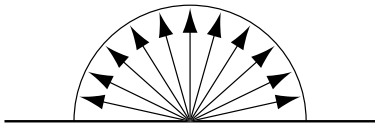


Figure 10: *Final gathering*

These final gather rays are traced to eventually hit other objects, where photon lookups are executed to refer to the photon map. The results from a final gathering are composed to calculate the final gather value at the object surface. This value corresponds to the irradiance value from diffuse surfaces. This final gathering may be necessary during the diffuse surface shading.

The number of final gather rays shot from a point on the object surface may vary from 9 to over 4,000, depending on the spatial complexity of the region. During a global illumination rendering, this final gathering consumes high percentage of the time. To speed up this final gathering, Kilauea has a special rendering pass called final gather estimation which pre-estimates the final gather values at most regions, instead of executing final gathering for every surface shading computation.

Close observation of many global illumination images reveals that the change in final gather values in most regions is very smooth. This in turn means that most final gatherings in the regular rendering are wasted in those smooth regions. A breakthrough in the rendering speed is expected by executing final gathering at the appropriate density depending on the smoothness, and interpolate the resulting final gather values to get the estimate on most regions.

Because the sampling pattern suitable for the final gathering and the actual rendering are different, Kilauea separated them into independent passes for the optimal operation. Kilauea executes the final gather estimation pass to compute the final gather values on object surfaces for all screen pixels, and then the final rendering pass refers to these estimated final gather values from the shading computations. To summarize, Kilauea typically renders a scene with the following three passes:

1. Photon tracing
2. Final gather estimation
3. Final rendering

For more detail on each stage, please refer to SIGGRAPH 2002 course notes about Kilauea[56].

6. Implementations

6.1. Message passing

Kilauea needs to exchange data across multiple machines and MPI is used for this purpose. At the beginning of the development, because all available MPI implementations were not thread safe and frequently caused mysterious errors, we ended up implementing a subset of MPI ourselves. Our MPI implementation achieves thread-safe MPI subset with Pthread implementation.

6.2. Threading

Multi-thread implementation using Pthread was encouraged throughout the development of Kilauea. The number of main engine threads, however, is fixed to the number of CPUs in the machine where the process is running. That is, there will be two engine threads on dual CPU machines and one engine thread on single CPU machines.

The main engine thread processes many tasks by scheduling and switching the tasks. More precisely, ray tracing engine, shading engine, and many procedures within the lookup operation which are the components of Kilauea's pipeline switch the execution frequently in the main engine thread.

This mechanism allows more precise priority control and adjustment of the execution components of Kilauea's pipeline. If each stage is implemented as an individual thread, the priority control must rely on Pthread's scheduler, which complicates the precise priority adjustment.

In Kilauea, the main engine thread exists as one Pthread thread, and various stages composing the rendering pipeline are implemented as individual function calls. The stages of the pipeline are connected by local queues. An execution of a stage takes out an item from its input queue, process it, and write out the result to the output queue. Because the stages are implemented as individual functions connected by input and output queues, the main engine thread can freely change the execution order of the stages, allowing proper and precise execution priority control as needed. As a result, minimizing the local queue size and latency in operations becomes possible.

6.3. Memory management

Kilauea manages heap memory allocation with extra caution. We have implemented our own thread-safe malloc(), and new operator of C++ is overridden to use this malloc(). This self-implemented malloc() is tuned especially for relatively small allocation of memory under 16 KB, which frequently occurs in the message data handling of Kilauea.

Also, this memory manager has its own memory verify function and various schemes to aid memory-related debugging, which are remarkably powerful in providing a robust system by troubleshooting memory problems under a complex multi-threaded environment.

7. Parallel performance results

7.1. Townhouse

Figure 11: *Townhouse: 732,058 triangles, one directional light, one sky light, four area lights in the rooms, 650,000 photons emitted, 885,315 photons stored*

This scene, the townhouse, is small enough to be stored in one machine, and Kilauea will distribute identical copies of scene data to all machines participating in the rendering. Three rendering times were recorded while changing the total number of machines:

A) All: total time from boot up to shut down
B) FG Est: final gather estimation stage
C) Render: final rendering stage

For this experiment, maximum of 15 machines with dual Pentium III 800 Hz, 512 MB of memory, and 100 BaseT Ethernet are used.

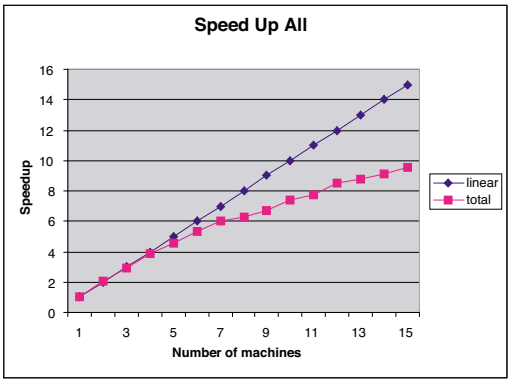


Figure 12: *Townhouse: performance of entire rendering*

Figure 12 is the plot of A), the total rendering time including Kilauea initialization and clean up. Unfortunately, the initialization stages such as booting tasks and reading in the scene data are essentially difficult to take the advantage of parallel processing or parallelization is not thoroughly considered yet. Due to this, the graph in Figure 12 gradually falls off the optimal linear performance as the number of machines increases.

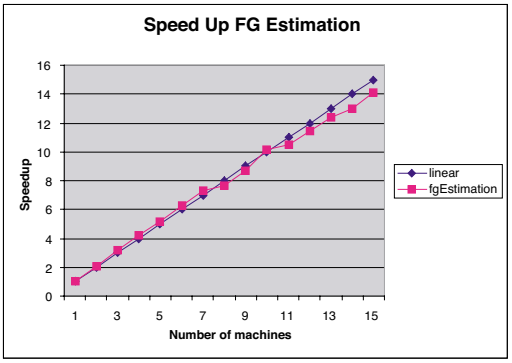


Figure 13: *Townhouse: FG estimation performance*

Figure 13 is the plot of B), the final gather estimation stage. This stage performs parallel computation by dividing up the screen space into rectangular buckets. It achieves adequate parallel performance, though the varying processing time for each tiles starts to dominate and affect the load balancing as the number of machines increases. By decreasing the size of each bucket, better scalability may be attained.

Figure 14 is the plot of C), the final rendering stage.

This stage exhibits a superlinear scalability, as the plots exceed the optimal values. This is speculated to be caused by caching at various levels.

The final gather estimation stage and the final rendering stage both exhibit satisfactory parallel performance, meaning that the more machines invested, the more performance increase achieved correspondingly. These two stages are the most repeated ones during the course of adjusting surface materials and lighting, and their optimal scalability is a huge plus in the production work.

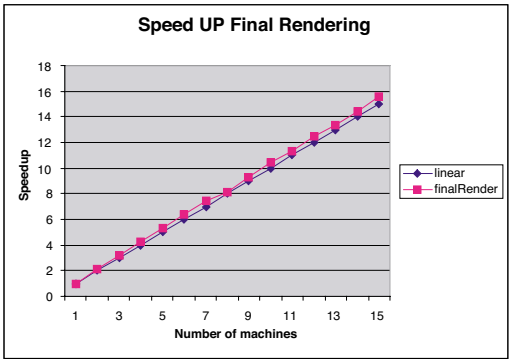


Figure 14: *Townhouse: final rendering performance*

Only 15 machines were available to conduct this experiment, and there is no actual data for using more machines. In theory, the network will be the bottleneck and the performance will saturate at some point as the number of machines increases. The only communication occurring in this experiment is returning of the final sampling results, whose overhead is very small. Therefore, parallel rendering using more than 50 machines should not be a problem at all.

7.2. Fiat 500L × 32

This test intends to show the parallel performance of Kilauea when the scene data is shared among multiple machines. The cars in Figure 15 are exactly identical, but for the purpose of this experiment, they are intentionally duplicated, not instantiated, to make the scene larger. Because one machine cannot hold this much data, it is shared among two machines. Because the minimum number of machines to hold the scene is two, the number of machines will be increased by the multiple of two.

Figure 16 is the plot for the entire rendering, including Kilauea initialization and clean up. Notice that the performance drop is not as drastic compared to the previous case. This is because the final rendering stage is slow and thus dominates the timing results, compared to initialization stages which are not-so-well parallelized.

Figure 17, showing the final gather estimation stage, is now also exhibiting superlinear behavior.

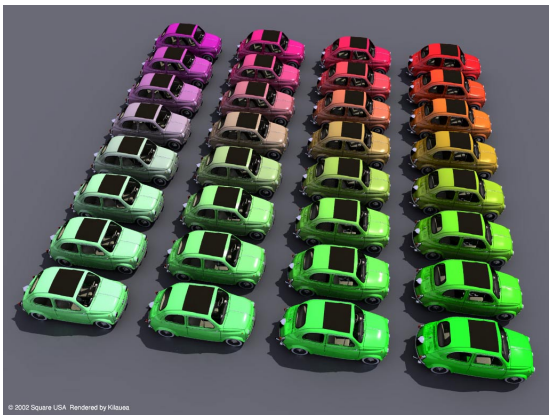


Figure 15: *Fiat 500L × 32: 2,994,752 triangles, one directional light, one sky light, 700,000 photons emitted, 756,492 photons stored*

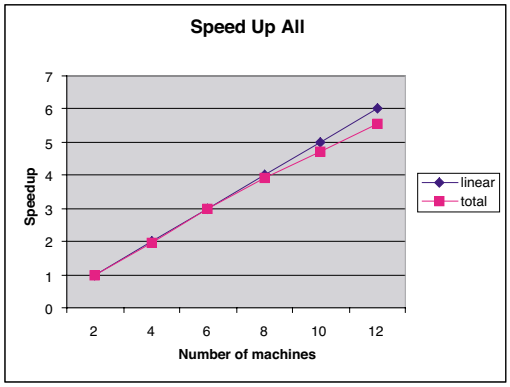


Figure 16: *Fiat 500L × 32: performance of entire rendering*

The scalability of the final rendering stage is also superb, as shown in Figure 18. Overall, the performance of the rendering stage increases as more machines are invested accordingly.

7.3. Discussion on parallel performance

The parallel rendering performance will eventually saturate because the network communication starts to be the bottleneck as the number of participating machines are increased either to speed up the rendering or to store larger scenes. Some of the solutions to remedy the situation are:

1. Suppress network usage
2. Use network with broader bandwidth

One idea to achieve the first solution is to implement communication compression. Currently, the compression is performed on certain data communication only. Applying com-

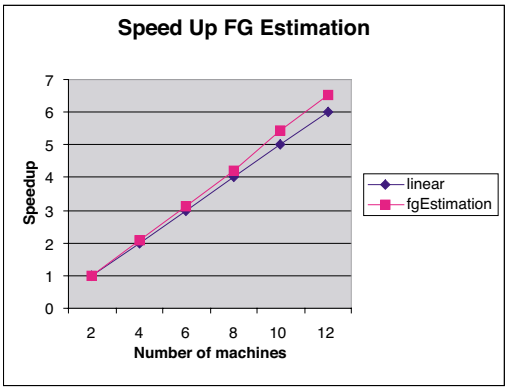


Figure 17: *Fiat 500L × 32: FG estimation performance*

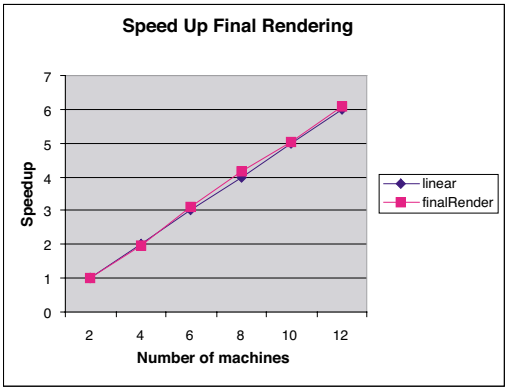


Figure 18: *Fiat 500L × 32: final rendering performance*

pression to all streaming data in Kilauea may somewhat improve the network traffic.

The second solution is to simply make use of the faster network such as Gigabit Ethernet, whose price is rapidly dropping lately. This will push the saturation point even further away.

8. Conclusions

Kilauea succeeded as a fully scalable parallel and distributed system yielding exceptional computational power from the affordable multi-CPU PC cluster. The number of machines to be used can be flexibly modified depending on the demanded speed and scene size.

The use of message-passing based distributed computing for ray tracing and data-flow oriented shading engine accomplished excellent load balancing to keep all CPUs constantly busy, and a general way to describe shading computation in a parallel environment. The latest version fully qualifies for the beta testing as the global illumination renderer in the actual productions.

Various new rendering technologies can be tested by a simple plug-in style development. The ideas on how to parallelize them can also be tested easily. These indicate that Kilauea succeeded in providing a test bed environment for various rendering and parallel processing algorithms.

Further speed-up, optimization, and feature add-ons are sufficiently achievable just by upgrading the current version. A lot of room for the improvements still remains, especially in speeding up the rendering. However, the original two goals to render global illumination and extremely large scenes are achieved using practical methodologies.

Acknowledgements

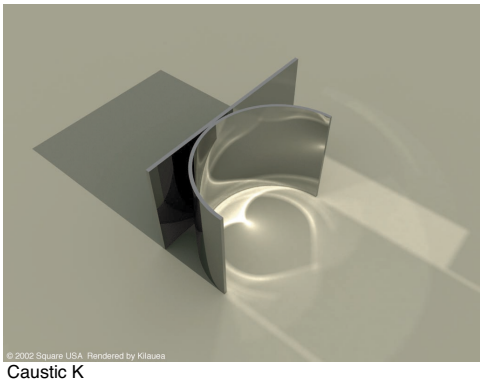
We would like to express our utmost gratitude to Kazuyuki Hashimoto, Kaveh Kardan, and all members of Square USA R&D division for provided us with numerous invaluable advice and ideas over the course of four years in the Kilauea Research Project. Many of the test image renderings would not have been possible without the great assistance of the Square USA movie production team.

We would also like to express our very best gratitude to Alan Chalmers of the University of Bristol for advice on parallel rendering, and Henrik Wann Jensen of Stanford University for advice on global illumination and photon mapping.

Sponza model was provided by the courtesy of Marko Dabrovic. Many test suite models are from De ESPONA 3D Encyclopedia 2000 Edition.

References

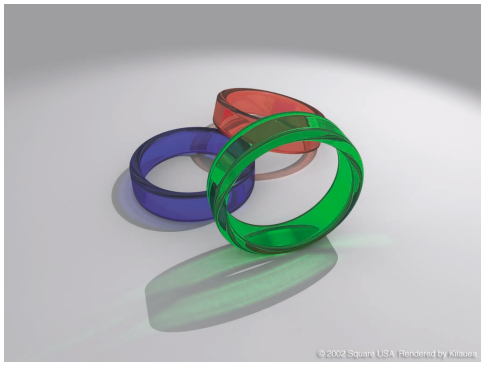
1. Henrik Wann Jensen. *Realistic Image Synthesis Using Photon Mapping*. ISBN: 1-56881-140-7, A K Peters, 2001.
2. Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. “Rendering Complex Scenes with Memory-Coherent Ray Tracing”, *Computer Graphics (SIGGRAPH ’97 Proceedings)*, pp. 101–108 (August 1997).
3. SIGGRAPH 2001 course note. “Parallel Rendering and the Quest for Realism: The ’Kilauea’ Massively Parallel Ray Tracer”.
4. SIGGRAPH 2001 course note. “A Practical Guide to Global Illumination Using Photon Mapping”.
5. SIGGRAPH 2002 course note. “The ’Kilauea’ Massively Parallel Global Illumination Renderer”.
6. SIGGRAPH 2002 course note. “Photon Map in Kilauea”.



Caustic K

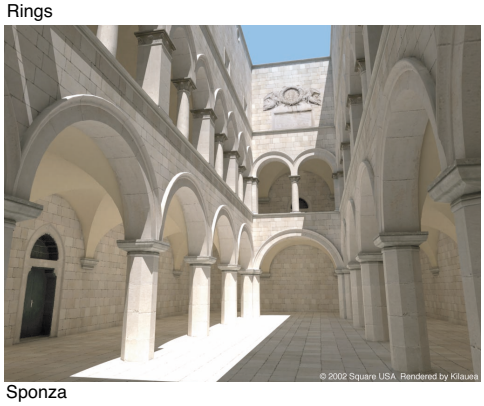
Panzer

FA Loader

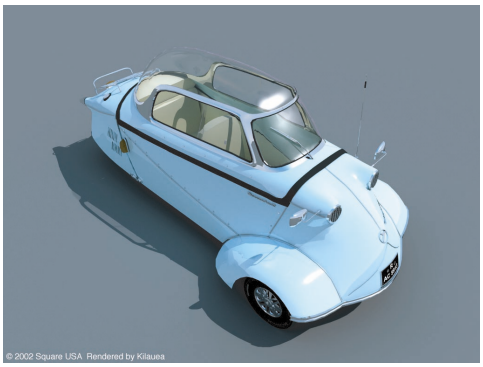


Rings

Fiat500L



Sponza



Messerschmidt


Vase

Images are created by Motohisa Adachi, Tadashi Endo and Tamotsu Maruyama

Figure 19: *Kilauea sample renderings*