Temporal Coherence-Based Distributed Ray Tracing of Massive Scenes

Xiang Xu, Lu Wang, Arsène Pérard-Gayot, Richard Membarth, Cuiyu Li, Chenglei Yang, and Philipp Slusallek

Abstract—Distributed ray tracing algorithms are widely used when rendering massive scenes, where data utilization and load balancing are the keys to improving performance. One essential observation is that rays are temporally coherent, which indicates that temporal information can be used to improve computational efficiency. In this paper, we use temporal coherence to optimize the performance of distributed ray tracing. First, we propose a temporal coherence-based scheduling algorithm to guide the task/data assignment and scheduling. Then, we propose a virtual portal structure to predict the radiance of rays based on the previous frame, and send the rays with low radiance to a precomputed simplified model for further tracing, which can dramatically reduce the traversal complexity and the overhead of network data transmission. The approach was validated on scenes of sizes up to 355 GB. Our algorithm can achieve a speedup of up to 81% compared to previous algorithms, with a very small mean squared error.

Index Terms—Computer graphics, ray tracing, distributed graphics

1 Introduction

Monte Carlo ray tracing is the most commonly used algorithm for rendering with global illumination. With the increasing demand for high-quality rendering, the geometric model's complexity also increases, making distributed and out-of-core rendering more important. A typical solution to render massive scenes that cannot fit in the main memory is dividing the scene into parts. Each part is called a *domain* containing an independent acceleration structure (e.g., BVH), and a *virtual portal* is placed between every two adjacent domains to record rays that may pass through a domain to the other. For the ray that intersects with the virtual portal and is ready to enter the next domain, the computing node chooses to fetch the domain data or pass the ray task according to different scheduling algorithms.

For distributed ray tracing, some dynamic scheduling algorithms [1], [2] perform ray transfer and domain fetching according to the current state of each distributed node to improve data utilization and balance the workload among nodes. However, those algorithms incur additional synchronization overhead and reduce parallelism. Other algorithms, like the domain-space decomposition algorithm [3], which does not synchronize information between nodes during rendering and allows communication and computation to execute asynchronously, does in fact improve computational efficiency but cannot guarantee load balancing.

In this paper, considering rays with good temporal coherence between frames, we propose a temporal coherence-based distributed ray tracing method that uses temporal coherence in ray tracing to improve the load balance and data utilization of the domain-space decomposition algorithm. Furthermore, we observe that multiple reflections of a ray increase the amount of computation and communication in distributed rendering but mainly add low-frequency radiance contribution. We use a simplified model to trace those rays with low radiance, which can not only decrease computation, but also avoid ray transmission between nodes. Our contributions are summarized as follows:

- An efficient temporal coherence-based scheduling algorithm, including a domain assignment algorithm and a runtime scheduling algorithm. Before rendering, our domain assignment algorithm assigns do-

- *Xiang Xu is with the Shandong Key Laboratory of Blockchain Finance, Shandong University of Finance and Economics, Jinan, Shandong 250101, China. E-mail: xuxiang8420@outlook.com.*
- *Lu Wang and Chenglei Yang are with the School of Software, Shandong University, Jinan, Shandong 250101, China. E-mail: {luwang_hcivr, chl_yang}sdu.edu.cn.*
- *Arsène Pérard-Gayot is with Wētā Digital, PO Box 15208, Miramar Wellington 6243, New Zealand. E-mail: aperardgayot@wetafx.co.nz.*
- *Richard Membarth is with the Technische Hochschule Ingolstadt (THI), Research Institute AImotion Bavaria, 85049 Ingolstadt, Bayern and the German Research Center for Artificial Intelligence (DFKI), Saarland Informatics Campus, 66123 Saarbrücken, Saarland, Germany. E-mail: richard.membarth@thi.de.*
- *Cuiyu Li was formerly with the Advanced Computing East China Sub-center, Suzhou, JiangSu 215300, China. E-mail: lxyystn@126.com.*
- *Philipp Slusallek is with the German Research Center for Artificial Intelligence (DFKI) and Saarland University, Saarland Informatics Campus, 66123 Saarbrücken, Saarland, Germany. E-mail: philipp.slusallek@dfki.de.*

Manuscript received 24 February 2022; revised 27 October 2022; accepted 28 October 2022. Date of publication 7 November 2022.
This work has been partially supported by the National Key R&D Program of China (2020YFB1709203), the National Natural Science Foundation of China (62272275, 61872223, 62007021, 62202268, 62002200), the Shandong Provincial Natural Science Foundation of China (ZR2020LZH016), the Shandong Provincial Science and Technology Support Program of Youth Innovation Team in Colleges (2021KJ069), as well as by the Federal Ministry of Education and Research (BMBF) as part of the HorME and Metacca projects. The calculation is supported by Advanced Computing East China Sub-center. (Corresponding author: Lu Wang.)
Recommended for acceptance by K. Moreland.
This article has supplementary downloadable material available at https://doi.org/10.1109/TVCG.2022.3219982, provided by the authors.
Digital Object Identifier no. 10.1109/TVCG.2022.3219982

mains to all nodes according to the ray transmission information among domains in the previous frame. At runtime, we propose a scheduling algorithm that estimates each domain's pre-loading prior based on the node's current situation and information of the previous frame.
- A new virtual portal structure to record the radiance of rays passing through domains in the previous frame. In the current frame, by the recorded radiance in our virtual portal, we predict the radiance of the ray intersects with the virtual portal and send the low radiance rays to a pre-loaded simplified model in the current node.
- An asynchronous distributed path tracing framework, which can process domain loading, rendering, and communication asynchronously. This distributed ray tracing architecture can achieve a more balanced workload and higher data utilization.

We tested our algorithms in scenes up to 355 Gbytes (including BVH). Compared with the dynamic scheduling algorithm [1] and the asynchronous domain-space decomposition algorithm [3], our algorithm is faster thanks to the temporal coherence-based scheduling strategy and thanks to the use of a simplified model for ray tracing.

2 Related Work

Out-of-core rendering and distributed rendering are the two main methods for rendering massive scenes. The efficiency of both of them is highly reliant on the task and data assignment and scheduling algorithm.

2.1 Out-of-Core Ray Tracing

Fetching domain data from storage is costly, so some research has focused on improving the data access efficiency for out-of-core rendering. Pharr et al. [4] proposed an out-of-core ray tracing method based on ray batching, which improves ray coherence in domain loading. Budge et al. [5] proposed a software system that enables hybrid CPU/GPU out-of-core rendering based on priority-based heuristics. Eisenacher et al. [6] reduced the loading costs by deferring shading of all shading points.

Using simplified models to assist computation is an important technique in massive scene rendering. Simplified models have a low memory footprint and computational overhead, but there is a significant loss of quality if they are applied directly to rendering. Yoon et al. [7] proposed a model following a LOD structure, which is implied in the acceleration structure and bounds the errors by an error metric. Moon et al. [8] obtain approximate shading points by intersecting the simplified model, and using it as a measure to reorder rays when rendering the original scene. Stoll et al. [9] proposed a multiresolution method by interpolating between discrete LODs for each ray. They compute their LODs by choosing proper tessellation levels for subdivision meshes.

2.2 Distributed Ray Tracing

Distributed ray tracing algorithms divide the rendering tasks and scene data and assign them to distributed nodes. The way to assign scene data and rendering tasks determines the efficiency of distributed ray tracing. These algorithms can be categorized into three classes: image-space decomposition, domain decomposition, and dynamic scheduling strategies.

Image-space decomposition strategies assign a portion of image samples to each node, and each node fetches the corresponding scene data from the network or local storage [10], [11], [12], [13], [14], [15], [16], [17]. For instance, Wald et al. [14], [15] divide the screen into multiple tiles and assign them to different nodes. They copy the top-level KD-tree to all nodes and store the bottom part in storage. At runtime, each node performs ray tracing in the top-level KD-tree and accordingly loads the corresponding subtree for further rendering. DeMarle et al. [12] combine the image-space decomposition strategy with distributed shared memory. However, their results rely on a preprocessing step to distribute the initial data, which is costly for massive scenes. Park et al. [17] introduced a speculative ray-scheduling method, which trades the redundant computation for data utilization to improve overall efficiency. Image-space decomposition methods can easily achieve load balancing between nodes [11], [18], but require frequent domain loading, leading to low data utilization.

Domain-space decomposition strategies are usually used when sufficient hardware resources are available. With this strategy, the rays are sent to the node containing the required domain data over network [3], [19], [20], [21], [22], [23]. Kobayashi et al. [22] decompose the scene into regular subspaces, and assign them to different processes. Kato et al. [23] evenly allocate the scene geometries to different nodes. Each node separately intersects all rays and gathers the results in a master node to get the nearest intersection. In this method, each node has the same amount of data and ray task, which naturally leads to a good workload balance, but the computation utilization is low. Abram et al. [3] proposed an asynchronous architecture for domain decomposition to execute ray transmission and render in parallel. Wald et al. [24] and Zellmann et al. [25] proposed decomposition algorithms for scenes with massive instances. They combined the object space and spatial space decomposition strategies to reduce the replication of proxy geometry between nodes and balance the data and computation of each node. Compared with image-space decomposition, the domain-space decomposition strategy has good data utilization and hardly requires data loading. Still, it is challenging to balance the workload (except for Kato et al. [23]).

The *dynamic scheduling method* mixes the above two methods and determines whether the process should perform image-space processing or domain scheduling based on various pieces of information during rendering [1], [2], [26], [27], [28]. Reinhard et al. [27] group coherent rays and load the relevant data locally. They send incoherent rays to other processors to find the required data. Navrátil et al. [1], [28] proposed a variety of dynamic scheduling algorithms suitable for different scenes. Son et al. [2] proposed a timeline scheduling for CPU/GPU heterogeneous clusters by a device connection graph and timing model. They predict data transmission and processing time to improve computation and bandwidth utilization.

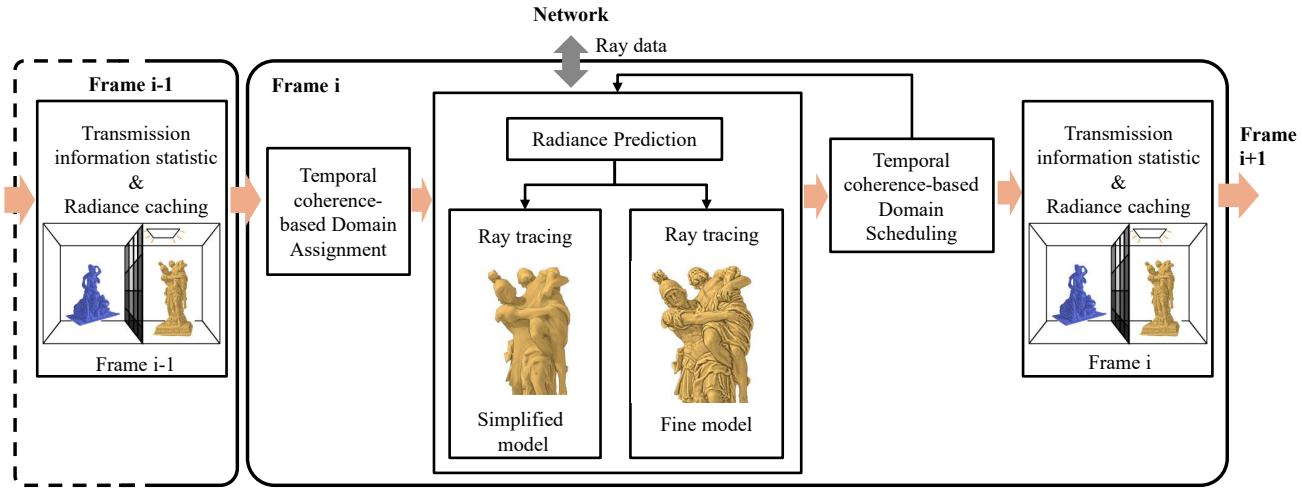

Fig. 1. The framework of our temporal coherence-based distributed ray tracing.

2.3 Spatial Directional Radiance Caching

Ray transmission in scenes is complicated. A spatial directional radiance caching structure is widely used to record the ray transmission. The precise radiance cache, also known as the radiance field, can be used directly as a rendering result, but it requires extensive precomputation to refine the cache [29], [30], [31]. The coarser cache can be constructed quickly and is commonly used to guide the ray sampling [32], [33]. It can be constructed at runtime and keep refining frame to frame. In this paper, we use a Holodeck-like [29] radiance caching structure, placing virtual radiance portals (or virtual radiance screens) around the composited distributed domains. Instead of using the cached radiance for rendering, we use it to judge whether to intersect rays with a simplified or a fine model to reduce the ray data transmission between distributed nodes.

3 Distributed Ray Tracing Framework

For our distributed ray tracing architecture, we used a hybrid ray tracing framework that combines the domain-space decomposition and out-of-core rendering. Before rendering, a group of domains is assigned to a node, and each node uses a cost-benefit formula to choose a domain to load or a cached domain to render at runtime.

For most scenes, the ray transmission in consecutive frames is very similar. This property is typically referred to as *temporal coherence*. We propose a temporal coherence-based distributed algorithm, which uses the information from the previous frame to optimize the rendering performance of the current frame. The rendering pipeline is shown in Fig. 1. Our algorithm has three main parts:

1) The ray transmission and radiance information is recorded when rendering the previous frame and further used as input data for the current frame (Section 4).
2) A temporal coherence-based domain assignment and a domain scheduling algorithm are proposed for the current frame by using the ray transmission data of the previous frame to improve the load balance and resource utilization (Section 5).
3) The cached radiance information on the virtual portal in the previous frame is used to predict the radiance of the ray intersected with the virtual portal in the current frame. A low memory-cost simplified model is used to render the predicted low radiance ray (Section 6).

Our method allows communication and computation to execute asynchronously. We realize the asynchronous execution by three kinds of thread:

- **Management thread** is responsible for communicating with other distributed nodes and managing other threads.
- **Domain preloading thread** is used to preload the uncached domain.
- **Rendering thread** process the exact rendering task.

Each node keeps ray queues for each domain to enable asynchronous ray data transfer between the management and rendering threads.

4 Temporal Information Collection

The temporal information we used between frames includes the ray transmission information between domains and the radiance of the rays passing through the domains. The ray transmission information can be easily recorded in the form of statistics, and, for radiance information, we propose a cache made of a virtual portal structure. That information will be recorded in each frame and further be used in the next frame (see Section 5 and Section 6, respectively) to improve the efficiency of distributed ray tracing.

4.1 Transmission Information Statistics for Domains

We measure statistics on ray transmission for each domain, at runtime, and for each frame. We take the domain A in Fig. 2 as an example. The recorded information includes:

- For domain A's neighbors, domain B and domain C, record the numbers of rays sent from domain A to each of them as $s(A, B)$ and $s(A, C)$, respectively.
- The total number of rays sent from domain A to other domains, recorded as $s(A)$.

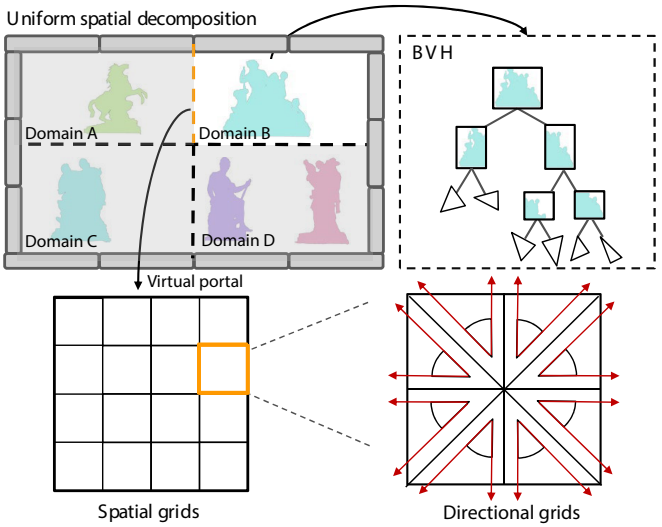

Fig. 2. Domain and virtual portal structure. A scene is uniformly divided into many domains, each domain has an individual BVH for the geometry inside the domain, and a 4D virtual portal in spatial and directional space is used to record the radiance of rays passing through the portal.

- The total number of rays received by domain A from other domains, recorded as $r(A)$.
- The proportion of $s(A)$ to $r(A)$, recorded as $R(A)$.
- The proportion of $s(A, B)$ to $r(A)$, recorded as $R(A, B)$.
- The data loading times of domain A.

4.2 Virtual Portal with Radiance Caching

Virtual Portal Structure. In our distributed ray tracing algorithm, we place virtual portals on each of the six faces of a domain's bounding box to determine whether rays need to be sent to other domains (see Fig. 2). Beyond that, we propose a four-dimensional (4D) virtual portal structure to record the radiance of rays that intersect with a virtual portal and pass through a domain, which records spatial and directional information (2D each, 4D total). We divide each virtual portal into uniform 2D grids (32×32) in spatial space to represent the position of the intersection between the ray and virtual portal at the virtual portal's local coordinate system. Since we only need to record rays that start tracing inside the domain for a virtual portal, the direction of a ray intersecting the virtual portal is always from inside to outside the domain, which means we can represent this direction in the virtual portal's coordinate system in a half-sphere. In practice, for each spatial grid, we separate it into eight angle areas, each representing a 2D direction. The entire virtual portal only takes several MB of memory, and the overhead time of using our 4D virtual portal structure is also small.

Radiance Recording on Virtual Portal. We record the ray's radiance at the closest virtual portal it intersected with (see Fig. 3). When a ray intersects a virtual portal for the first time, we calculate its virtual portal coordinates and record it onto the ray's structure, and keep tracing the ray until it intersects geometry in a domain. After finishing the light source visibility test (shadow ray intersection) and shading, we record the radiance of this ray path to the proper coordinates on the virtual portal. Once the current frame has been finished, the recorded virtual portal is gathered and broadcast to each node and will be used to predict the radiance of rays in the next frame. To resolve potential conflicts, when recording radiance in our virtual portal, we only keep the maximum radiance on the coordinate if there is already a radiance value.

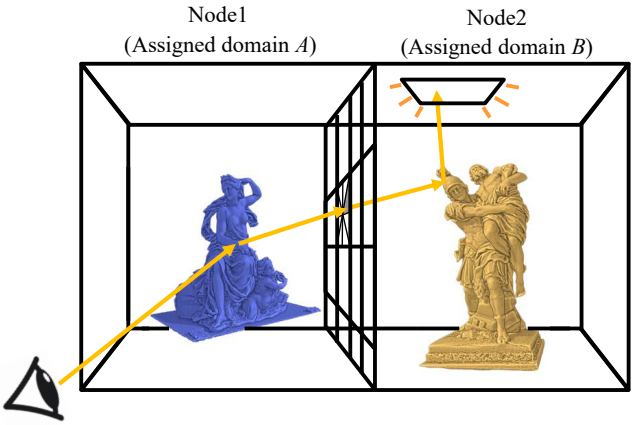

Fig. 3. Radiance recording on virtual portal. Domain A and domain 2 are assigned to node 1 and node 2, respectively. A ray is passing through the virtual portal from domain 1 to domain 2. Then, its radiance is recorded on the corresponding virtual portal's coordinates.

4.3 Node Information

As the distances between each node and the respective network storage system have wide variations, the data loading speeds are different for each node. In the first frame, when a node is loading a domain, we record the storage size and loading time of the domain and further calculate their ratio as the data loading speed S_{PROC} of the node.

After one frame is completed, node 0 gathers all the recorded information from other nodes and broadcasts the results to all nodes.

5 Temporal Coherence-based Domain Assignment and Scheduling

We use a domain assignment algorithm to assign the domains to each node, and a runtime domain scheduling algorithm to select the next domain to pre-load during rendering. By using the recorded temporal information in the previous frame, our temporal coherence-based domain assignment and scheduling algorithms can improve data utilization and balance the workload between each node.

5.1 Domain Assignment

When rendering with n nodes, we divide the screen space evenly into n tiles and group the domains into an equal number of n domain groups. We pack a screen tile and a domain group as an *AssignmentUnit* structure (see Algorithm 1 lines 1–4). Each node will be assigned an AssignmentUnit, in which the screen tile is the node's ray generating task and the domain group is its processing data. For a given node, domains inside its domain group are called its *local domains*. A node can only load its local domains for ray tracing at runtime.

For massive scene ray tracing, transmitting rays between nodes is faster than out-of-core domain loading. Hence, we hope to minimize the ray transmission between domains inside an AssignmentUnit, to reduce the domain loading times of each node. Our assignment algorithm consists of three steps (see Algorithm 1):

Step 1: Select initial domains for each AssignmentUnit (lines 9–15). We assign the n screen tiles to the n AssignmentUnits in turn (lines 9–10). To use the screen space coherence, we project the bounding box of each domain to screen space (lines 11–12). And for each tile, we select the domain with the largest projection area with it, and set the domain as the initial domain for this tile's AssignmentUnit (lines 13–15).

If the viewpoint is inside a model, the domain containing this model has the largest projection area for all tiles, which will be set as initial domain for all AssignmentUnits. Further more, for other domains, we sort them according to the number of rays sending from them in previous frame (i.e. $s(A)$ for domain A), and choose the first n domains, and assign them to n AssignmentUnits in turn.

Step 2: Group remaining domains (lines 18–21). We add a remaining domain to the AssignmentUnit with the fewest domain numbers in a loop until all domains have been assigned. Each time, we choose an AssignmentUnit U with the fewest domains, then find a remaining domain with lowest ray transmission number with all existing domains in U according to the information recorded in previous frame and add the domain to the domain group for U (lines 18–21).

Step 3: Assign AssignmentUnits to nodes (lines 24–27). We sort all AssignmentUnits by the data size of domains in each AssignmentUnit, and sort all nodes by the data loading speed (lines 24–25). Then we assign the AssignmentUnit with a larger data volume to the node with the faster loading speed (lines 26–27).

Algorithm 1 Assign Domains

```
1:  AssignmentUnit {
2:    List⟨Domain⟩ domainGroups;
3:    Tile tile;
4:  };
5:  AssignDomains:
6:    AssignmentUnit units[n];
7:
8:    #Step 1: Select initial domains.
9:    for each T_i in all screenTiles;
10:     units[i].tile = T_i;
11:   for each D_i in allDomains;
12:     ProjectDomainToImageSpace (D_i);
13:   for each U_i in units;
14:     D = MaxProjectedAreaDomain (U_i.tile);
15:     U_i.domainGroups.insert (D);
16:
17:   #Step 2: Group the remaining domains.
18:   while hasUnAssignedDomain
19:     U = FindLeastDomain(units);
20:     D = LeastTransmission (U, UnAssignedDomain);
21:     U.domainGroups.insert (D);
22:
23:   #Step 3: Assign AssignmentUnits to nodes.
24:   sort (nodes, maxLoadingSpeed);
25:   sort (units, maxStorageUsed);
26:   for each N_i in allNode;
27:     N_i.initialisation (units[i]);
```

5.2 Domain Scheduling

At runtime, for each node, we use a temporal coherent-based domain scheduling algorithm to choose a candidate domain from the domain group (including multiple local domains) assigned to it and pre-load the candidate domain as the next active domain for further rendering.

More precisely, when a node is rendering an active domain, its management thread will calculate the priority of other local domains according to a cost-benefit function and select the domain with the highest priority as the next active domain. If the next active domain is not cached, the domain pre-loading thread will load this domain when the active domain is being rendered. Meanwhile, if the domain cache is full, the domain with the lowest priority will be selected to unload.

Our cost-benefit function calculates the domain's priority by predicting its computation time and domain loading time. For a node n and a candidate local domain d, we use $Q_{PRED}(d)$ to denote the predicted quantity of rays that may be processed when domain d is loaded. $S_{PROC}(d)$ denotes the ray processing speed of the domain d in the previous frame, $T_{LOAD}(n,d)$ denotes the time of node n to load domain d, and $Dep^{'}(d)$ denotes the normalized distance between the viewpoint and domain d. The priority of domain d, $P(n,d)$, is calculated as follows:

$$P(n,d) = (1 - Dep^{'}(d)) \cdot \left(\frac{Q_{PRED}(d)}{S_{PROC}(d)} - T_{LOAD}(n,d) \right) \tag{1}$$

The larger the difference of $Q_{PRED}(d)/S_{PROC}(d)$ and $T_{LOAD}(n,d)$, means that domain d is more computationally intensive, while its loading overhead is relatively small, therefore this domain has a higher priority to be loaded. When rays are transmitted in a scene, domains that are closer to the viewpoint tend to be requested earlier, which is why we use $1 - Dep^{'}(d)$ as a term to schedule the closer domain to viewpoint. The parameters $T_{LOAD}(n,d)$ and $Q_{PRED}(d)$ in Eq. 1 are calculated according to Eq. 2 and Eq. 3, respectively.

In Eq. 2, $V(d)$ denotes the data size of domain d and $S_{LOAD}(n)$ denotes the loading speed of node n. If the domain is already cached, its loading cost will be zero. Otherwise, we use $V(d)$ and $S_{LOAD}(n)$ to calculate the domain loading time.

$$T_{LOAD}(n,d) = \begin{cases} 0 & \text{if domain loaded} \\ \dfrac{V(d)}{S_{LOAD}(n)} & \text{otherwise} \end{cases} \tag{2}$$

Considering the next active domain is selected while the active domain is being rendered, we cannot get the exact quantity of rays that need to be rendered when a domain d is loaded to be rendered: Because the active domain may generate additional rays for domain d, the other nodes may also send rays to domain d when processing the active domain.

Still, we can predict this quantity as $Q_{PRED}(d)$ based on the current ray transmission status and the ray transmission information in the previous frame (see Section 4.1). For the currently active domain act and the candidate domain d, the $Q_{PRED}(d)$ calculation is defined as follow:

$$\begin{aligned} Q_{PRED}(d) = & Q_{CUR}(d) \\ & + Q_{CUR}(act)R(act)R(act,d) \\ & + Q_{POT}(d) \end{aligned} \tag{3}$$

The variables act and d represent the currently active domains and the candidate domain, respectively. We divide $Q_{PRED}(d)$ into three parts, corresponding to the three terms summed in Eq. 3. The first term $Q_{CUR}(d)$ represents the current ray quantity of domain d. For the second term, we use $Q_{CUR}(act)R(act)R(act,d)$, to approximate the ray quantity generated from the active domain act for domain d after domain act finishes rendering the current ray $Q_{CUR}(act)$. In this expression, the terms $R(act)$ and $R(act,d)$ are recorded at the previous frame, see subsection 4.1. Finally, the third term $Q_{POT}(d)$ represents the received ray quantity from other nodes' (non-local) domain when domain d is selected as the active domain and being rendered, and is defined in the following equation:

$$Q_{POT}(d) = \frac{Q_{PRE}(d) - Q_{CUR\_RECV}(d)}{N_{PRE}(d) - N_{CUR}(d)} \tag{4}$$

In this formula, $Q_{PRE}(d)$ represents the rays sent to domain d from the non-local domains in the previous frame, and $Q_{CUR\_RECV}(d)$ represents the number of rays that the domain d has received from them in the current frame. We use $Q_{PRE}(d) - Q_{CUR\_RECV}(d)$ to approximate the remaining rays that may be received from other nodes. $N_{PRE}(d)$ is the total loading times of domain d in the previous frame, and $N_{CUR}(d)$ counts the number of times the domain d has been loaded in the current frame. We use this formula to approximate the average rays received by domain d in each of the remaining loading operations, and use it as the ray quantity received from non-local domains if domain d is rendered as the active domain.

6 Temporal Coherence-based Simplified Model Tracing Algorithm

So far, we have finished our main scheduling algorithm for distributed ray tracing. Furthermore, we proposed an optimized ray tracing algorithm to reduce the overhead of ray computation and transmission by using a simplified model.

When tracing rays, an increased number of ray bounces brings more computation overhead and makes rays less coherent. Especially for the massive scene distributed ray tracing, incoherent rays will lead to more ray transmission between nodes and more domain loading times, which may significantly reduce the rendering efficiency. Thus, we store precomputed simplified models of the whole scene in each node. For those unimportant rays (with low radiance) passing through its neighboring domains, we prefer to trace them with simplified models in the current node to reduce the data transfer overhead.

6.1 Tracing with Simplified Models

The camera rays (depth of 0) contribute the most to the radiance, so we use the original fine model to trace. For rays with depths ≥ 2, which provide almost exclusively low-frequency radiance contributions, we trace them all using the simplified model. For rays at a depth of 1, we use our virtual portal data structure to predict ray radiance and trace those rays with low radiance with the simplified model.

As simplified models are resident in memory for all nodes, we don't need to transfer data for further tracing. At the same time, tracing with a simplified model can also simplify the intersection computation. Since there is little to no energy loss when tracing perfect reflection and refraction rays, we render all such rays with the original fine model.

6.2 Radiance Prediction

From the recorded radiance on our virtual portal in the previous frame (Section. 4.2), we can predict radiance of a ray when it intersects with the virtual portal. Then, according to the predicted radiance, we decide whether to send the ray to the simplified model or not.

When a ray intersects the virtual portal, we get its coordinates in our 4D virtual portal grid according to the intersection position and the ray's direction, thus getting the recorded radiance at this coordinate in the previous frame and using it as the predicted radiance of the current ray. If the predicted radiance of the ray is greater than a threshold, we trace it with the original fine model; otherwise, we render it with the simplified model (Fig. 4). We derived the threshold experimentally and found that a value of 0.5 was optimal (see section 7.3.2).

The radiance distribution and the resolution of our virtual portal impact the performance and accuracy of radiance prediction, which we will analyze separately in subsection 7.3.3.

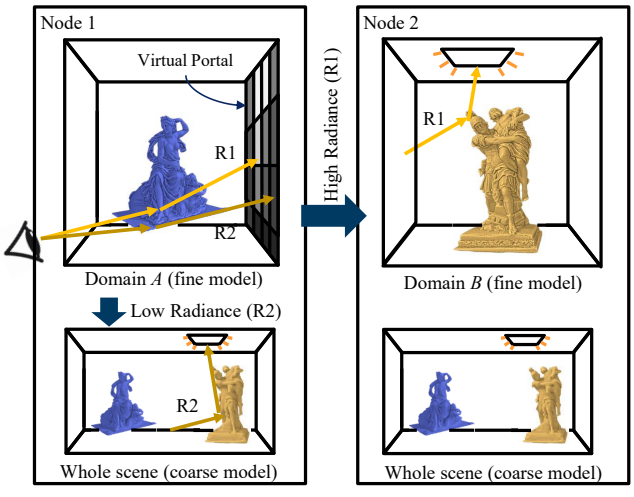

Fig. 4. Radiance prediction. Node 1 and node 2 are assigned to domain 1 and domain 2, respectively, and a simplified scene model is always kept in the memory of each node. For the predicted high radiance ray (e.g., R1), we send it to Node 2 to intersect with the fine model of domain 2. While, for the ray R2 that was predicted as low radiance, we use the local kept simplified model to render.

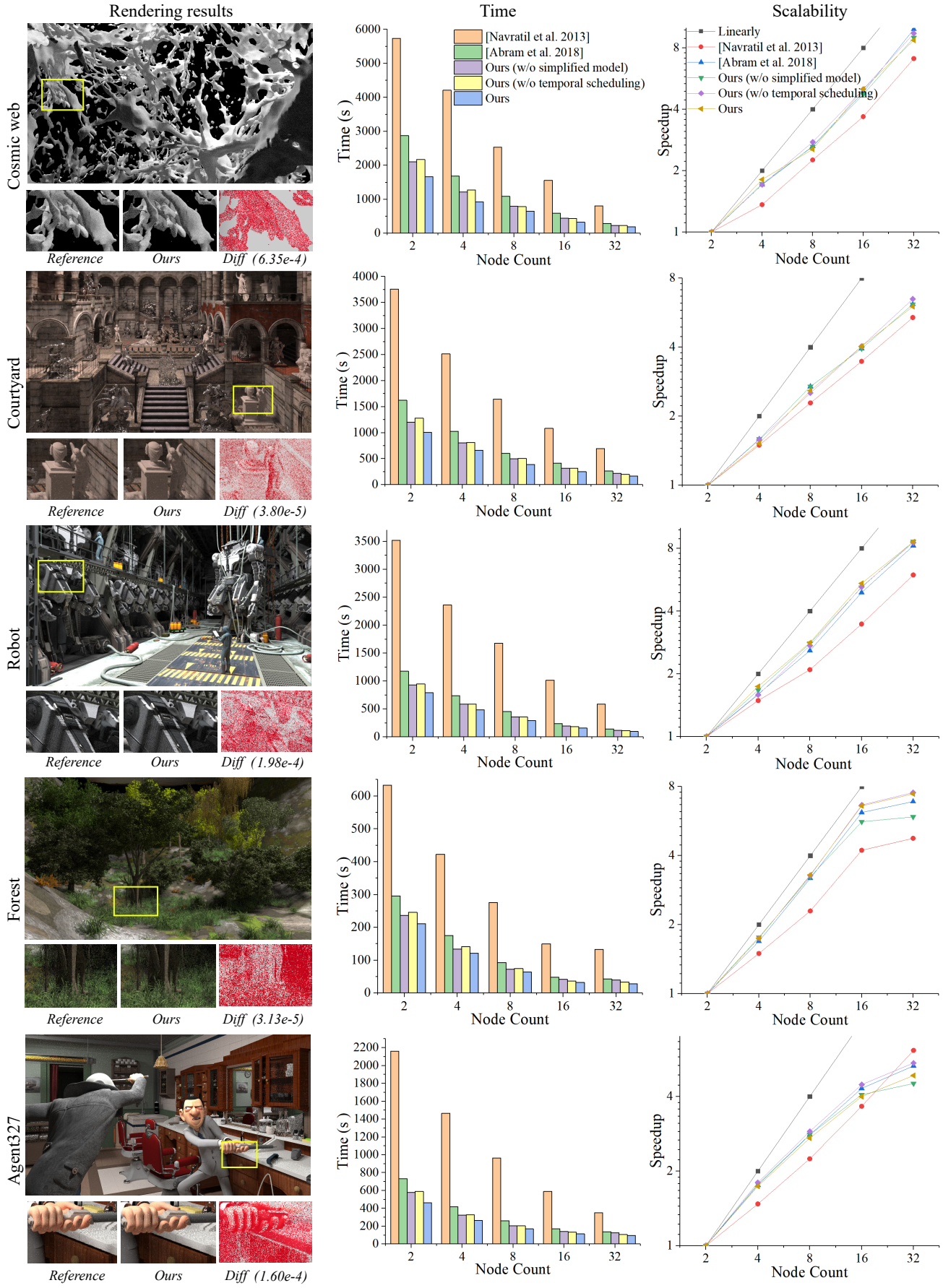

Fig. 5. Rendering results and performance of our five test scenes. The first column shows our rendering results and difference with reference. And the second and third columns show the time and scalability comparison among ours, Navrátil et al. [1] and Abram et al. [3], for different nodes, respectively.

7 RESULTS

We test our algorithm on the Sugon advanced computing cluster, each computing node has a 32-core 2.0 GHz x86 Hygon Dhyana processor with 128 GB main memory, each distributed node and storage system is connected by 200 GB/s HDR InfiniBand network. Our path tracing implementation is based on the Rodent [34] path tracer with a BVH with branching factors eight as the accelerated structure and supports multiple importance sampling, area lights, next event estimation (NEE) [35], and physically-based materials. We use MPI [36] for message passing between nodes. Table 1 shows the statistics of our five test scenes. Fig. 5 shows the rendering results and performance. For the Agent327 scene [37], we also provides the results with 1024 spp in Fig. 9 and Fig. 10.

TABLE 1
Statistics for each test scene. Model memory includes the scene information and the BVH. For each scene we use the same rendering parameters, 1920×1080 resolution and the max ray path depth is eight.

	Tris.	Original Model Mem.	Simplified model Mem/Rate.	SPP
Cosmic Web	1437M	354.5G	163M/0.05%	64
Courtyard	923M	209.3G	149M/0.07%	128
Robot	378M	86.5G	173M/0.19%	32
Forest	90M	18.6G	204M/1.05%	32
Agent327	56M	11.3G	161M/1.39%	128/1024

7.1 Performance Comparison

We measure the impact of our temporal coherence-based scheduling, and our ray tracing algorithm based on a simplified model, and compare it with the dynamic scheduling algorithm proposed by Navrátil et al. [1] and the domain-space decomposition algorithm proposed by Abram et al. [3]. To be fair, we modified all the compared algorithms so that each node only allows loading of the assigned domains and use the current number of rays in the domain as its priority at runtime for domain scheduling. For the method of Navrátil et al. [1], we assign the spatially adjacent domains to each node. We also use the same spatial-based algorithm to distribute domain data for Abram et al. [3]. The method of Abram et al. [3] is also used as the baseline for our algorithm evaluation.

Rendering Time. We compared the rendering time of these five methods, where each node caches four domains (Fig. 5). We can see that Abram et al. [3] is about 2 to 3 times faster than Navrátil et al. [1]. This is because, in Abram et al. [3]'s asynchronous domain-space decomposition algorithm, the ray transmission overhead can be hidden by computation. However, Navrátil et al. [1] has to synchronize nodes frequently, increasing ray data transmission and reducing parallelism between nodes. Compared with Abram et al. [3], our temporal coherence-based scheduling (w/o simplified model in Fig. 5) can take the ray transmission information into account in the domain assignment and domain scheduling phase, and consequently achieves up to 39% (average 26%) speedup. Moreover, using a simplified model for ray tracing (w/o temporal scheduling in Fig. 5) greatly reduces the amount of communication and calculation caused by the rays' multiple bounces, increasing the speed by up to 38% (average 27%). When we use the two together, the speedup can be up to 81% (average 56%).

Scalability. As shown in Fig. 5, Navrátil et al. [1] is lower in parallel efficiency than other algorithms. The parallel efficiency of our scheduling algorithm is similar to Abram et al. [3]. With respect to communication between nodes, our simplified model tracing algorithm is better than the others because the simplified model is maintained in each node's memory, reducing the communication overhead caused by adding more nodes. Additionally, the parallel efficiency is highly affected by the task size. When using 16 nodes for the Forest, each node only needs to cache four domains to load the entire scene, and thus there will be no domain loading overhead. With that setup, the rendering workload is also insufficient, leading to a significant bottleneck. This point also can be seen from Fig. 6.

Fig. 6 shows the average time of three parts of our algorithm at each node count. When there are few nodes, each node has enough work, the communication, data-loading, and rendering can be better parallelized, and the time of each node is close to the total time. However, when the number of nodes increases, the total time does not show the same reduction as the average time of each thread because it becomes harder to balance the workload with the insufficient work.

Domain Cache Rate. Different domain cache rates have a great impact on rendering speed. A larger domain cache rate means fewer domain scheduling operations, especially when the cache size is 100%, domain scheduling is no longer needed. Fig. 7 shows the results of the five algorithms with different maximum domain cache sizes. As the domain cache rate increases, the rendering time shows a similar trend to Fig. 5. In this figure, Navrátil et al. [1] is still slower than the others, and our algorithm is still faster than other algorithms even at 100% cache rate.

7.2 Quality Evaluation

In terms of rendering quality, Abram et al. [3] and Navrátil et al. [1] are both lossless, while the use of the simplified model in our algorithm can cause some quality loss. We use Abram et al. [3] as the reference for quality evaluation.

Static Scenes. The difference between our algorithm and the reference for static scenes without animation is shown in Fig. 5. The MSE in the darker area is larger because, in that area, we use the simplified model at smaller ray depths. There is a trade-off between quality loss and performance when choosing to use the simplified model during ray tracing, which we will discuss in subsection 7.3.2.

Dynamic Scenes. Fig. 8, Fig. 9, and Fig. 10 show our rendering quality for moving viewpoint, dynamic geometry, and dynamic light sources, respectively, where we only record the radiance information in the first frame. We can see that our algorithm can maintain a low MSE even in dynamic scenes.

7.3 Analysis of Our Algorithm

Our algorithm contains two parts: temporal coherence-based scheduling and ray tracing using a simplified model, providing 26% and 27% speedups, respectively. This subsection will experiment and analyse these two parts separately, and present some other important parameters.

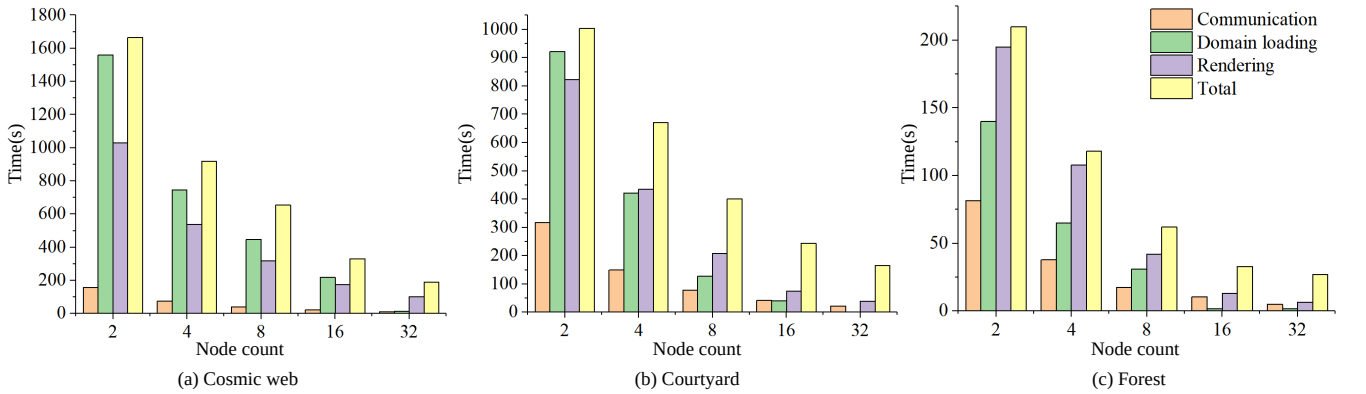

Fig. 6. Execution time of the three parts and the total time of our algorithm when we use both temporal coherence-based scheduling and simplified models for ray tracing (8 nodes). The X-axis is the number of nodes, and the Y-axis is the average time of one frame.

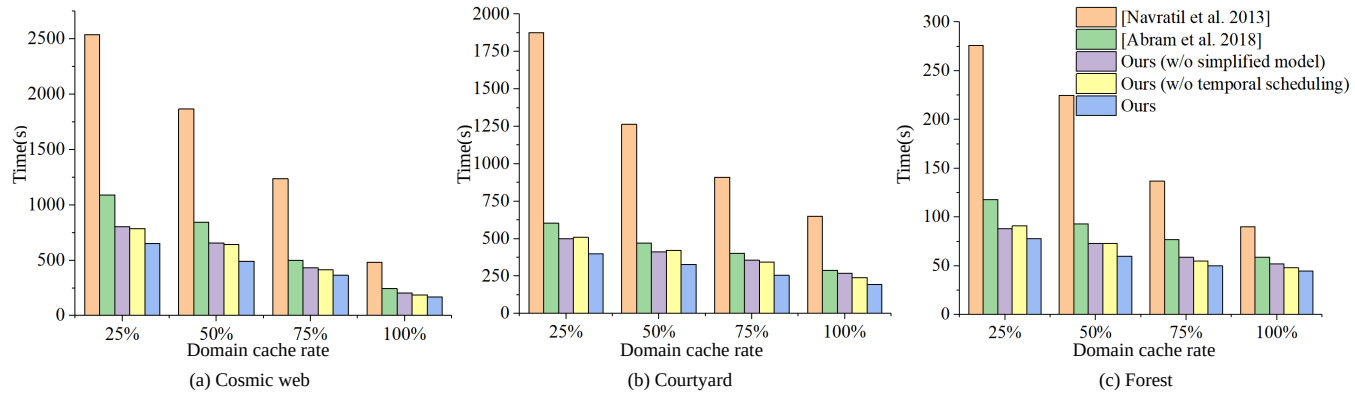

Fig. 7. Performance comparison with different domain cache rates (8 nodes). The X-axis is the domain cache size, and the Y-axis is the average rendering time of one frame.

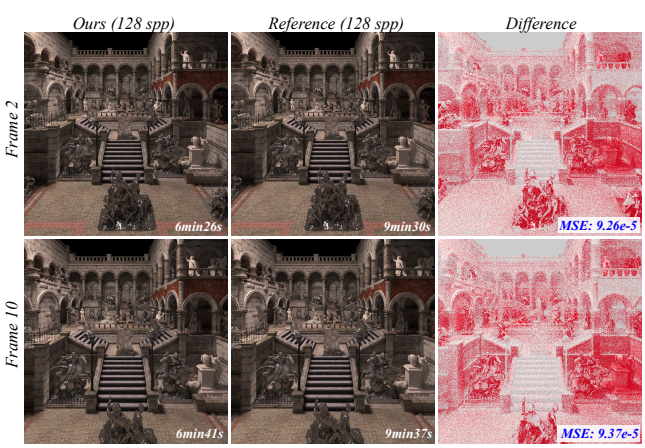

Fig. 8. Rendering results with moving viewpoint (8 nodes).

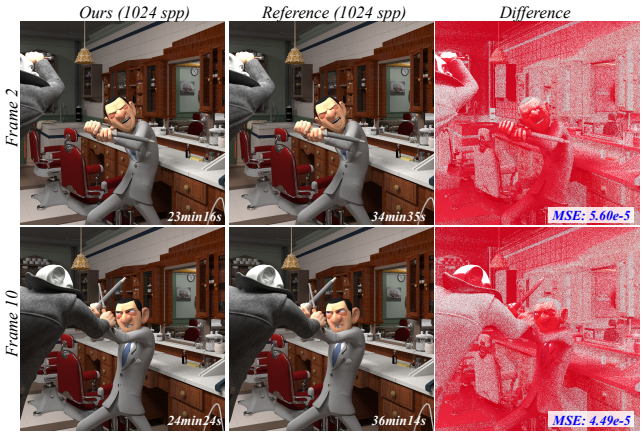

Fig. 9. Rendering results with dynamic geometry (8 nodes).

7.3.1 *Temporal Coherence-Based Scheduling*

Our temporal coherence-based scheduling algorithm includes domain assignment and runtime domain scheduling. Table 2 shows the rendering time with different domain assignment algorithms and different domain scheduling cost-benefit formulas. We compare our domain assignment algorithm with sequential and spatial-based assignment, where sequential assignment has a more balanced load while spatial-based assignment has less communication between nodes. As shown in Table 2, spatial-based assignment is faster than sequential assignment for most scenes, except the Courtyard and Robot, due to their highly unbalanced distribution in spatial space. Our temporal coherence-based domain assignment strategy is 7.9% faster than sequential assignment and 6.4% faster than spatial assignment.

When we add domain scheduling to our domain assignment algorithm, we also add domain pre-loading, which improved the rendering speed by 9.2%. We use the strategy that only considers the current ray quantity of a domain as the baseline algorithm. Compared with the baseline, when we take the current rays of the domain d and the rays generated by domain act into account, the speed can be improved by 5.9%. And, our complete algorithm can give

TABLE 2
Statistics for temporal coherence-based scheduling (8 nodes, cache 4 domains). The meaning of symbols is explained in Section 5.2.

	$Cosmic\,web$	$Courtyard$	$Robot$	$Forest$	$Agent327$
Domain assignment					
Sequential assignment	1143s	604s	456s	93s	262s
Spatial-based assignment	1091s	605s	466s	92s	254s
Temporal coherence-based assignment (ours)	951s	573s	443s	88s	248s
+ Domain Scheduling with different Cost-benefit formula					
$Q_{CUR}(d)$	864s	548s	394s	82s	222s
Spray scheduling [17]	837s	528s	373s	80s	210s
$Q_{CUR}(d) + Q_{CUR}(act)R(act)R(act,d)$	833s	513s	375s	75s	212s
$(1 - Dep^{'}(d)) \cdot \left(\frac{Q_{PRED}(d)}{S_{PROC}(d)} - T_{LOAD}(n,d)\right)$ (ours)	802s	493s	361s	73s	206s

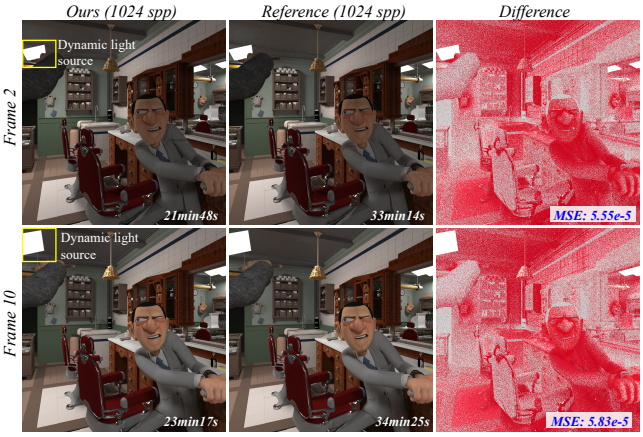

Fig. 10. Rendering results with dynamic light source (8 nodes).

a 9.6% speed up to the baseline algorithm. We also compared Park et al. [17]'s scheduling strategy, which consists in scoring domain priority by intersecting rays with the domain bounding box. This strategy can evaluate the ray transmission well in synchronous distributed architectures, but in our asynchronous distributed architecture, the ray in each node is constantly updated; moreover, the ray bounding box intersection test will result in additional overhead. Even if we ignore the additional overhead, our scheduling algorithm is 5.8% faster than Park et al. [17].

7.3.2 Simplified Model and Ray Tracing Algorithm

The use of simplified models during ray tracing can significantly improve the overall rendering speed, and only incurs a small additional memory footprint. We use the edge collapse method to simplify each object in a scene, except for some objects with too few faces, like the Courtyard architecture. The Forest scene has too many simple objects that cannot be simplified, leaves and grasses, and we simplify this scene by randomly removing some of them. In our test, the simplification rate from 0.05% to 1.39% (as shown in Table 1), the most drastically simplified scene, Cosmic web, has the largest MSE, while the Forest has the least MSE for its least simplified. However, many other scene settings can also affect the MSE, like the viewpoint and the light source distribution. Besides, in some scenes, many un-simplified simple models occupied a large range (architecture of Courtyard and Robot), resulting in their small MSE.

Besides the simplification quality, the condition which determines if we can use the simplified model is also a crucial factor that affects the rendering speed and quality. Table 3 shows the rendering time and MSE for different conditions. This table shows that if we send all rays with a depth greater than 0 to the simplified model, the rendering speed will be greatly improved, but it will also bring a significant MSE. If we only send the rays with a depth greater than two to the simplified model, the MSE is smaller, but the speed is slow. When we add the radiance contribution condition and the ray depth condition, we achieve a smaller MSE and maintain the speed increase. Therefore, we send the rays whose depth is greater than 0 and radiance is less than 0.5 and the rays with a depth greater than 1 to the simplified model.

7.3.3 Virtual Portal Analysis

The radiance prediction efficiency of our virtual portal is affected by its resolution and the radiance distribution of scenes. In addition, since the virtual portals are placed between domains, a different domain decomposition size can also lead to different radiance prediction accuracy.

Radiance Distribution. In the radiance caching stage, we only keep the highest value, so the distribution of radiance of each scene will affect prediction accuracy, as shown in Table 4. For scenes with low radiance and discrete high radiance distribution, such as Courtyard, the radiance prediction accuracy is 84.3%, and 85.7% of rays with a depth of 1 use the simplified model. In contrast, for scenes with high radiance and a concentrated distribution, such as Agent327, the prediction accuracy is higher (98.3%), but only 16.5% of the rays are rendered using the simplified model.

Virtual Portal Resolution. A higher resolution of our 4D virtual portal can record the radiance information in more detail. When using a high resolution, high radiance rays can be better identified, which leads to higher prediction accuracy and more rays rendered using the simplified model, while still producing an image with lower MSE, as shown in Table 4. Since high virtual portal resolutions consume more memory and require more computing and communication than smaller ones, we chose resolution 32×32 to balance overheads and benefits.

7.3.4 Parameter Analysis

The temporal information update interval and the domain decomposition size are two important parameters affecting the rendering speed and quality.

Temporal Information Update Interval. When recording radiance in our virtual portal, we need to trace all the

TABLE 3
The rendering time and image MSE when using the simplified model under different ray depths and radiance. D represents the ray depths of rays sent to the simplified model, and R represents the radiance of rays sent to the simplified model.

		Origin model	$D>2$	$D>1$	$D>0$ & $R<0.3$	$D>0$ & $R<0.5$	$D>0$
$Cosmic\,web$	Time(s)	801	724	693	675	654	611
	MSE	0	2.07e-4	5.94e-4	6.27e-4	6.35e-4	2.50e-3
$Courtyard$	Time(s)	495s	447	416	397	393	354
	MSE	0	2.55e-5	3.e-5	3.77e-5	3.80e-5	1.09e-4
$Robot$	Time(s)	361	339	319	298	294	226
	MSE	0	1.36e-4	1.87e-4	1.97e-4	1.99e-4	7.19e-4
$Forest$	Time(s)	75	72	69	65	63	58
	MSE	0	8.15e-6	2.43e-5	2.93e-5	3.13e-5	2.738e-4
$Agent327$	Time(s)	205	196	178	172	170	136
	MSE	0	1.33e-4	1.55e-4	1.59e-4	1.60e-4	1.17e-3

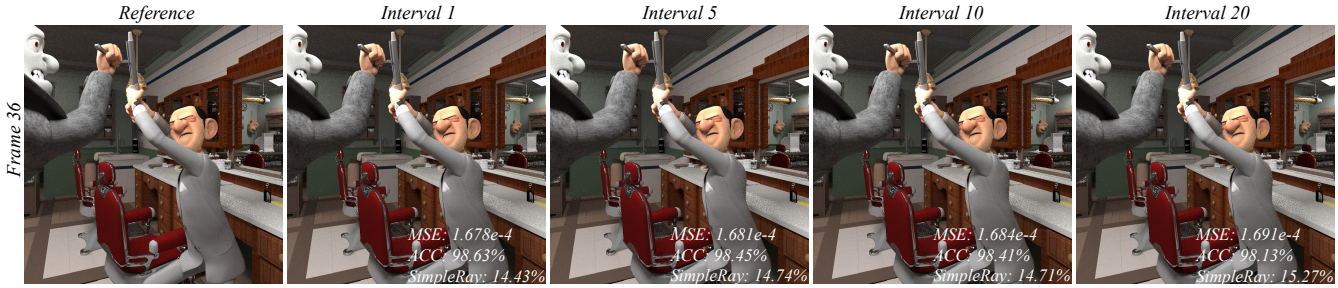

Fig. 11. Rendering results of frame 36 in different number of frame intervals for a dynamic scene (8 nodes), where interval 5 means the radiance of frame 31 is cached. The $SimpleRay$ denotes the ratio of rays rendered by simplified model to all rays transmitted between domains (depth of 1).

TABLE 4
Virtual portal resolution. Statistics at different 2D spatial space resolutions, including rendering time, MSE, prediction accuracy, and the ratio of rays predicted to be low to all rays transmitted between domains (depth of 1).

	8×8	16×16	32×32	64×64	128×128
Courtyard					
$Time(s)$	379	386	395	407	414
MSE	3.812e-5	3.808e-5	3.807e-5	3.796e-5	3.791e-5
ACC	81.7%	83.1%	84.3%	86.6%	86.9%
$Simple/total$	84.4%	84.9%	85.7%	86.5%	86.7%
Agent327					
$Time(s)$	162	166	170	182	192
MSE	1.615e-4	1.611e-4	1.604e-4	1.602e-4	1.597e-04
ACC	97.4%	97.7%	98.3%	98.5%	98.6%
$Simple/total$	16.0%	16.2%	16.5%	16.8%	17.1%

rays at depth 1 by the original model, which will reduce the rendering speed. Thus, we update our virtual portal every several frames. Fig. 11 shows the MSE of our algorithm at different frame intervals in a dynamic scene. The MSE is connected to both the accuracy and the number of rays sent to the simplified model, where the ray size sent to the simplified model is mainly related to the radiance distribution of the cached frame and the current frame, and the accuracy is determined by the similarity between the two frames. We chose to update our virtual portal every ten frames to balance the radiance recording overhead and the rendering quality.

Domain Decomposition Size. Table 5 shows the effect of domain decomposition size on our virtual portal. In our simplified model ray tracing algorithm, we only judge the ray to be sent to the simplified model when it passes through the virtual portal. Large domain division will cache more radiance information, leading to high accuracy. However, it will make more rays pass through the virtual portal, making rays more likely to send to the simplified model, resulting in a high MSE. Moreover, the domain scheduling algorithm works better when there are more domains, further improving the rendering speed. We chose to use more domain decomposition to improve the efficiency of our algorithms.

TABLE 5
Domain decomposition. Statistics at different domain decomposition size, including rendering time, MSE, prediction accuracy, and the ratio of rays predicted to be low to all rays transmitted between domains (depth of 1).

	16	32	64	128
Courtyard				
$Time(s)$	426	413	404	395
MSE	3.61e-05	3.75e-5	3.78e-5	3.81e-05
ACC	77.8%	80.1%	81.7%	84.3%
$Simple/total$	77.5%	80.2%	82.4%	85.7%
Agent327				
$Time(s)$	201	192	186	170
MSE	1.51e-4	1.54e-4	1.58e-4	1.60e-4
ACC	95.6%	96.3%	96.8%	98.3%
$Simple/total$	14.5%	15.4%	15.8%	16.5%

8 LIMITATION

Our simplified model tracing method sends all perfectly reflected and refracted rays to the fine model to trace because they have no energy loss, which makes our method inefficient for scenes with many perfectly reflected and refracted rays. In the Sibenik scene (Fig. 12), many objects are mirror or glass material, so we can only send very few rays to the simplified model. In this scene, our simplified model tracing method can provide a 5.3% speedup compared to Abram et al. [3], while for the scenes in Fig. 5, it can provide an average 27% speedup.

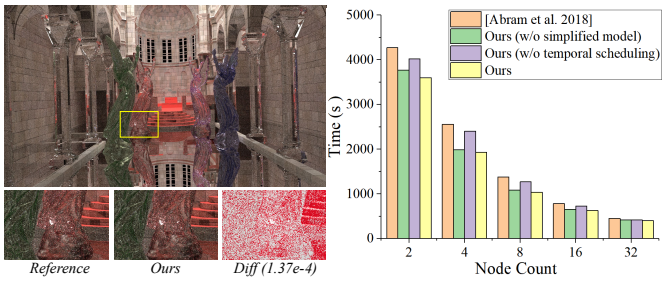

Fig. 12. Sibenik scene rendering with 1920x1080 resolution, 512 SPP. The scene is 8.4GB, and its simplified model is 76MB. The two Lucys in front use mirror material, and the other two use glass material.

In addition, we use a uniform grid to divide the scene into domains and assign domains for each node according to the number of domains regardless of the different memory footprints of each domain, resulting in an unbalanced data distribution among nodes. In domain scheduling, considering only the number of domains also leads to low memory usage.

9 Conclusions

We used the temporal coherence in ray tracing to accelerate the efficiency of domain-space decomposition distributed ray tracing. We proposed a temporal coherence-based domain assignment and scheduling algorithm, that uses the recorded ray transmission information in the previous frame to improve the computation utilization and load balance. Moreover, we send some of the rays passing through domains to a simplified model, according to the radiance cached at our 4D virtual portal structure in the previous frame, to reduce the computation and communication overhead. Finally, compared to the previous domain-space decomposition algorithm, our temporal coherence-based distributed ray tracing algorithm can achieve up to 81% speedup with a quality loss below e^{-4}.

As mentioned in section 8, some limitations and issues still require further research. We would like to propose a new virtual portal structure to record the complex ray transmission between domains and adapt to different domain decomposition methods. And combining some other conditions (etc., frustum) with the ray's radiance to guide the simplified model usage is also worth studying.

Acknowledgments

We thank the reviewers for their valuable comments.

References

[1] P. A. Navrátil, H. Childs, D. S. Fussell, and C. Lin, "Exploring the spectrum of dynamic scheduling algorithms for scalable distributed-memory ray tracing," IEEE Transactions on Visualization and Computer Graphics, vol. 20, no. 6, pp. 893–906, 2013.

[2] M. Son and S.-E. Yoon, "Timeline scheduling for out-of-core ray batching," in Proceedings of High-Performance Graphics, 2017, pp. 11:1–11:10.

[3] G. Abram, P. Navrátil, P. Grossett, D. Rogers, and J. Ahrens, "Galaxy: Asynchronous ray tracing for large high-fidelity visualization," in 2018 IEEE 8th Symposium on Large Data Analysis and Visualization (LDAV). IEEE, 2018, pp. 72–76.

[4] M. Pharr, C. E. Kolb, R. Gershbein, and P. Hanrahan, "Rendering complex scenes with memory-coherent ray tracing," in Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH). ACM, 1997, pp. 101–108.

[5] B. Budge, T. Bernardin, J. A. Stuart, S. Sengupta, K. I. Joy, and J. D. Owens, "Out-of-core data management for path tracing on hybrid resources," Computer Graphics Forum, vol. 28, no. 2, pp. 385–396, 2009.

[6] C. Eisenacher, G. Nichols, A. Selle, and B. Burley, "Sorted deferred shading for production path tracing," Computer Graphics Forum, vol. 32, no. 4, pp. 125–132, 2013.

[7] S.-E. Yoon, C. Lauterbach, and D. Manocha, "R-LODs: Fast LOD-based ray tracing of massive models," The Visual Computer, vol. 22, no. 9, pp. 772–784, 2006.

[8] B. Moon, Y. Byun, T.-J. Kim, P. Claudio, H.-S. Kim, Y.-J. Ban, S. W. Nam, and S.-E. Yoon, "Cache-oblivious ray reordering," ACM Transactions on Graphics (TOG), vol. 29, no. 3, pp. 1–10, 2010.

[9] P. Djeu, W. Hunt, R. Wang, I. Elhassan, G. Stoll, and W. R. Mark, "Razor: An architecture for dynamic multiresolution ray tracing," ACM Transactions on Graphics, vol. 30, no. 5, oct 2011.

[10] D. Badouel, K. Bouatouch, and T. Priol, "Distributing data and control for ray tracing in parallel," IEEE computer graphics and applications, vol. 14, no. 4, pp. 69–77, 1994.

[11] T. Plachetka, "Tuning of algorithms for independent task placement in the context of demand-driven parallel ray tracing," in Proceedings of the 5th Eurographics conference on Parallel Graphics and Visualization, 2004, pp. 101–109.

[12] D. E. DeMarle, C. P. Gribble, S. Boulos, and S. G. Parker, "Memory sharing for interactive ray tracing on clusters," Parallel Computing, vol. 31, no. 2, pp. 221–242, 2005.

[13] D. E. DeMarle, C. P. Gribble, and S. G. Parker, "Memory-savvy distributed interactive ray tracing," in Eurographics Workshop on Parallel Graphics and Visualization, 2004, pp. 93–100.

[14] I. Wald, P. Slusallek, and C. Benthin, "Interactive distributed ray tracing of highly complex models," in Eurographics Workshop on Rendering Techniques. Springer, 2001, pp. 277–288.

[15] D. E. DeMarle, S. Parker, M. Hartner, C. Gribble, and C. Hansen, "Distributed interactive ray tracing for large volume visualization," in IEEE Symposium on Parallel and Large-Data Visualization and Graphics (PVG). IEEE, 2003, pp. 87–94.

[16] A. Keller, C. Wächter, M. Raab, D. Seibert, D. van Antwerpen, J. Korndörfer, and L. Kettner, "The iray light transport simulation and rendering system," ACM SIGGRAPH 2017 Talks, pp. 1–2, 2017.

[17] H. Park, D. Fussell, and P. Navrátil, "SpRay: Speculative ray scheduling for large data visualization," in 8th Symposium on Large Data Analysis and Visualization (LDAV). IEEE, 2018, pp. 77–86.

[18] T. Plachetka, "Event-driven message passing and parallel simulation of global illumination," 2003.

[19] M. Larsen, C. Harrison, J. Kress, D. Pugmire, J. S. Meredith, and H. Childs, "Performance modeling of in situ rendering," in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC). IEEE, 2016, pp. 276–287.

[20] M. Howison, E. Bethel, and H. Childs, "MPI-hybrid parallelism for volume rendering on large, multi-core systems," in Eurographics Symposium on Parallel Graphics and Visualization (EGPGV), 2010, pp. 1–10.

[21] C. H. Howison M, Bethel EW, "Hybrid parallelism for volume rendering on large, multi, and many-core systems," IEEE Transactions on Visualization and Computer Graphics, vol. 18, no. 1, pp. 17–29, 2011.

[22] H. Kobayashi, S. Nishimura, H. Kubota, T. Nakamura, and Y. Shigei, "Load balancing strategies for a parallel ray-tracing system based on constant subdivision," The Visual Computer, vol. 4, no. 4, pp. 197–209, 1988.

[23] T. Kato and J. Saito, "Kilauea–parallel global illumination renderer," in Eurographics Workshop on Parallel Graphics and Visualization, 2002, pp. 7–17.

[24] I. Wald and S. G. Parker, "Data parallel path tracing in object space," ArXiv, vol. abs/2204.10170, 2022.

[25] S. Zellmann, N. Morrical, I. Wald, and V. Pascucci, "Finding Efficient Spatial Distributions for Massively Instanced 3-d Models," in Eurographics Symposium on Parallel Graphics and Visualization,

S. Frey, J. Huang, and F. Sadlo, Eds. The Eurographics Association, 2020.
[26] S. A. Green and D. J. Paddon, "A highly flexible multiprocessor solution for ray tracing," Visual Computer, vol. 6, no. 2, pp. 62–73, 1990.
[27] E. Reinhard, A. Chalmers, and F. W. Jansen, "Hybrid scheduling for parallel rendering using coherent ray tasks," in IEEE Symposium on Parallel Visualization and Graphics. IEEE, 1999, pp. 21–28.
[28] P. A. Navrátil, D. S. Fussell, C. Lin, and H. Childs, "Dynamic scheduling for large-scale distributed-memory ray tracing," in Eurographics Symposium on Parallel Graphics and Visualization (EGPGV). The Eurographics Association, 2012, pp. 61–70.
[29] G. W. Larson, "The holodeck: A parallel ray-caching rendering system," 1998.
[30] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng, "Nerf: Representing scenes as neural radiance fields for view synthesis," in ECCV, 2020.
[31] J. Zhu, Y. Bai, Z. Xu, S. Bako, E. Velázquez-Armendáriz, L. Wang, P. Sen, M. Hašan, and L.-Q. Yan, "Neural complex luminaires: Representation and rendering," ACM Transactions on Graphics, vol. 40, no. 4, jul 2021.
[32] V. Gassenbauer, J. Křivánek, and K. Bouatouch, "Spatial directional radiance caching," in Proceedings of the Twentieth Eurographics Conference on Rendering, ser. EGSR'09. Eurographics Association, 2009, p. 1189–1198.
[33] T. Müller, M. Gross, and J. Novák, "Practical Path Guiding for Efficient Light-transport Simulation," Computer Graphics Forum, 2017.
[34] A. Perard-Gayot, R. Membarth, R. Leissa, S. Hack, and P. Slusallek, "Rodent: Generating renderers without writing a generator," ACM Transactions on Graphics, vol. 38, no. 4CD, pp. 40.1–40.12, 2019.
[35] P. Matt, J. Wenzel, and H. Greg, Physically Based Rendering, 3rd ed. Morgan Kaufmann, 2016.
[36] W. D. Gropp, E. L. Lusk, and A. Skjellum, Using MPI - portable parallel programming with the message-parsing interface. Using MPI - portable parallel programming with the message-parsing interface, 1994.
[37] B. Studio, "Agent 327—blender cloud," https://cloud.blender.org/films/agent-327, 2020.

Arsène Pérard-Gayot is a rendering researcher at Weta Digital, New Zealand. He holds a computer science masters' degree from Ensimag, France, and a PhD in computer graphics from Saarland University, Germany.

His research interests are centered around compilation and programming language design, computer graphics, parallel programming and high-performance computing.

Richard Membarth is a professor for system on a chip and AI for edge computing at the Technische Hochschule Ingolstadt (THI), Germany and affiliated professor at the German Research Center for Artificial Intelligence (DFKI), Germany. He holds a diploma degree and a PhD in Computer Science from the Friedrich-Alexander University Erlangen-Nürnberg, Germany as well as a postgraduate diploma in Computer and Information Sciences from the Auckland University of Technologies, New Zealand.

His research interests include parallel computer architectures and programming models with a focus on automatic code generation for a variety of architectures ranging from embedded systems to HPC installations for applications from image processing, computer graphics, scientific computing, and deep learning.

Cuiyu Li received the PhD degree in computational chemistry from East China Normal University, China, in 2020. She is now a Postdoctoral Fellow at the Zhijiang Laboratory, China. Her research interest is parallel computing and machine learning.

Xiang Xu received the PhD degree in software engineering from Shandong University, China, in 2021. He is a lecturer in Shandong Key Laboratory of Blockchain Finance, Shandong University of Finance and Economics, China. His research interests include photorealistic rendering, high-performance computing and machine learning.

Chenglei Yang received the PhD degree in computer software and theory from Shandong University, China, in 2004. He is now a professor in School of Software, Shandong University, China. His research interest is computational geometry, human-computer interaction and virtual reality.

Lu Wang received the PhD degree in computer science and technology at Shandong University, China, in 2009. She is a professor in School of Software, Shandong University, China. Her research interests include photorealistic rendering, real-time rendering, material appearance modeling, and high-performance rendering.

Philipp Slusallek is professor for computer graphics at Saarland University, Germany and scientific director and member of the executive board at the German Research Center for Artificial Intelligence (DFKI), where he heads the research area on Agents and Simulated Reality. His research interest are centered around the intersection of real-time and realistic graphics, artificial intelligence, high-performance computing, and novel programming and compiler techniques.