fractal and fractional

MDPI

Article

Task Partitioning and Scheduling for Fractal Graphics Rendering Based on Web Worker

Jing Li [1], Liwei Mo [2], Ying Wang [3], Keyi He [4] and Hao Guan [2,5,*]

[1] School of Mathematics and Information Science, Guangzhou University, Guangzhou 510006, China; jing_lee@e.gzhu.edu.cn
[2] Institute of Computing Science and Technology, Guangzhou University, Guangzhou 510006, China; moliwei@e.gzhu.edu.cn
[3] School of Cyberspace Security, Guangzhou University of Software, Guangzhou 510990, China; wying@mail.gzus.edu.cn
[4] Department of Business Administration, Hong Kong Shue Yan University, Hong Kong, China; 243007@hksyu.edu.hk
[5] School of Computer Science of Information Technology, Qiannan Normal University for Nationalities, Duyun 558000, China
* Correspondence: guanhao@gzhu.edu.cn

Abstract

As web technologies shift from traditional content delivery toward high-performance computing, Web Worker-based parallelism for compute-intensive tasks such as fractal rendering has become essential for improving web application performance and interactivity. However, due to memory isolation and limited hardware access imposed by the browser sandbox, existing parallel approaches often struggle with excessive communication overhead, severe load imbalance, and straggler blocking when rendering spatially heterogeneous fractal graphics, especially under high-frequency interaction. This paper presents a task partitioning and scheduling scheme for parallel fractal rendering on Web Worker that addresses these challenges. By introducing load prediction for fractal rendering tasks, a communication-aware task partitioning scheme is achieved together with a scheduling strategy that combines interaction-aware adaptive prioritization and stale task cancellation. Experimental results show that the proposed method effectively reduces task skew and communication costs while outperforming baseline methods in term of parallel performance, load balancing, and response latency. Under high-frequency interaction, it also delivers notable improvements in system responsiveness and interaction smoothness, substantially enhancing both rendering efficiency and user experience for complex fractal graphics on the web.

Keywords: web worker; fractal graphics; task partitioning; task scheduling; inverse distance weighting

Check for updates

Academic Editors: Haci Mehmet Baskonus and Riccardo Caponetto

Received: 24 May 2026
Revised: 26 July 2026
Accepted: 2 September 2026
Published: 6 September 2026

Copyright: © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

1. Introduction

With the continued evolution of internet technology and network infrastructure, web applications are transitioning from traditional content presentation platforms to high performance computing environments. Web applications no longer serve merely as document viewers; they have evolved into cross-platform runtime environments that support massive data processing [1], real-time 3D rendering [2–4], scientific visualization [5], and AI inference. Traditional architectures typically offload compute intensive tasks to cloud servers while the web only handles result presentation. However, as user

scale and interaction frequency grow substantially, the pure cloud model reveals increasingly significant limitations in network bandwidth consumption, end-to-end latency, and operational costs. To address these issues, standards bodies and browser vendors have driven the expansion of web computing capabilities. The HTML5 standard provides the foundation for high-performance web applications, while the Web Worker specification overcomes the performance bottleneck of JavaScript's single-threaded model. Browsers can now process compute intensive tasks in parallel through independent Web Workers, fully exploiting the multi-core capabilities of end devices. This not only improves the execution efficiency of complex web applications but also provides essential technical support for building a new generation of apps with desktop-level interactivity.

Although significant progress has been made in exploiting hardware capabilities within browser environments, task partitioning and scheduling remain persistent challenges in developing parallel web applications. While the Web Worker mechanism enables parallel computation at the browser level, practical implementations still face issues such as thread isolation, message passing overhead, and resource contention. Due to JavaScript's memory isolation model, communication between the main thread and Web Workers involves data serialization and deserialization. When the transferred data are large or state synchronization is frequent, the resulting communication overhead can offset the performance gains from parallelism. Even though SharedArrayBuffer enables shared memory and Atomics provides basic data synchronization, the browser environment lacks native support for advanced concurrency primitives such as fine-grained locks and condition variables. Developers must rely on atomic operations or custom synchronization strategies to ensure data consistency, which in turn increases the complexity of designing and implementing parallel web applications.

Generation and rendering of fractal graphics remains a classic computer graphics problem in such complex parallel environments. Fractal graphics serve as an important research subject in computer graphics and scientific visualization, with the Mandelbrot set and Julia set being the most widely studied [6]. Due to their self-similarity and infinite detail properties, fractals are extensively used in natural shape modeling, mathematics education, and graphic art [7]. The rendering of fractal graphics exhibits strong spatial heterogeneity: sampling points near fractal boundaries typically require many iterations to determine convergence, while other regions demand considerably less computation. With conventional fixed grid partitioning, task blocks containing large boundary regions lead to severe load imbalance. Moreover, interactive exploration of fractal graphics in web applications imposes stricter performance requirements. Frequent user operations such as zooming and panning require the system to redraw the viewport within a very short response time.

Fractal rendering is characterized by high computational intensity, uneven load distribution, and strict sensitivity to interaction latency, making it difficult for traditional parallel computing approaches to utilize resources effectively. Conventional task partitioning methods that rely on fixed grids or static task pools perform well under uniform workloads, but struggle to maintain load balance in applications like fractal rendering where complexity varies significantly across space [8]. To model task complexity and reconstruct load distributions, related work in various domains has introduced load prediction mechanisms such as Markov models [9,10], machine learning [11,12], and deep learning [13], thereby informing task partitioning. At the partitioning level, strategies including domain decomposition [14], program dependency graph partitioning [15], and clustering [16] have shown promise in heterogeneous or cluster environments, and dynamic partitioning driven by deep learning is also gaining attention [17]. However, these methods typically depend on low-level hardware support or incur substantial pre-

processing overhead, which limits their direct applicability in the constrained browser sandbox; moreover, they do not fully exploit the spatial coherence and local abruptness inherent in fractal computation. In terms of task scheduling, existing approaches have evolved from FIFO queues [18] to task pools with work-stealing mechanisms [19] and further to adaptive strategies for heterogeneous environments [19,20], continuously improving resource utilization. However, most of these methods assume a relatively stable task set. In high-frequency interaction scenarios such as fractal visualization, continuous user operations rapidly change the viewport, causing a large number of in-flight tasks to become obsolete. Traditional scheduling strategies lack effective mechanisms to handle such stale tasks, leading to wasted computation on invalid tasks and degraded real-time responsiveness in critical regions [21].

This paper studies and designs a Web Worker-based task partitioning and scheduling scheme for fractal rendering. This scheme leverages the spatial distribution characteristics of fractal iterations in combination with rapid probing and historical escape distributions for load prediction to guide task partitioning. By establishing computation and communication cost models, an adaptive recursive partitioning strategy is adopted that performs fine-grained partitioning on high-complexity regions while maintaining larger task blocks for low-complexity regions, thereby reducing load imbalance and communication overhead. On this basis, we introduce an interaction-aware adaptive priority scheduling strategy combined with a soft cancellation mechanism to handle expired tasks, thereby reducing unnecessary computation and improving the rendering response speed for critical viewport regions.

2. Load Prediction

The iterative escape behavior in fractal rendering leads to significant differences in computational load across regions. For instance, areas near fractal boundaries typically incur higher computation costs because a large number of iterations are required to determine whether points in these regions diverge, while other regions that diverge within only a few iterations demand considerably less computation. In interactive fractal exploration, the inherent properties of fractal graphics make load prediction feasible. The self-similarity and local structural stability of fractals create exploitable correlations between consecutive frames, and this correlation is particularly strong when the zoom or pan magnitude is small. Moreover, the escape iteration count of each sampling point serves as a direct measure of computational complexity. Figure 1 shows the Mandelbrot set and its locally magnified details, illustrating the complex boundary structures and typical self-similar characteristics exhibited by this fractal at different scales.

Figure 2 shows the overall process of the fractal computation load prediction method proposed in this paper. First, a coarse-grained complexity rendering map H_{initial} is generated via rapid probing with a low iteration limit to locate high-load boundary regions. Then, by incorporating historical escape time distributions and leveraging a sparse quadtree with inverse distance weighting (IDW) interpolation, fine-grained load prediction is performed on hotspot regions, yielding a more accurate predicted complexity heatmap.

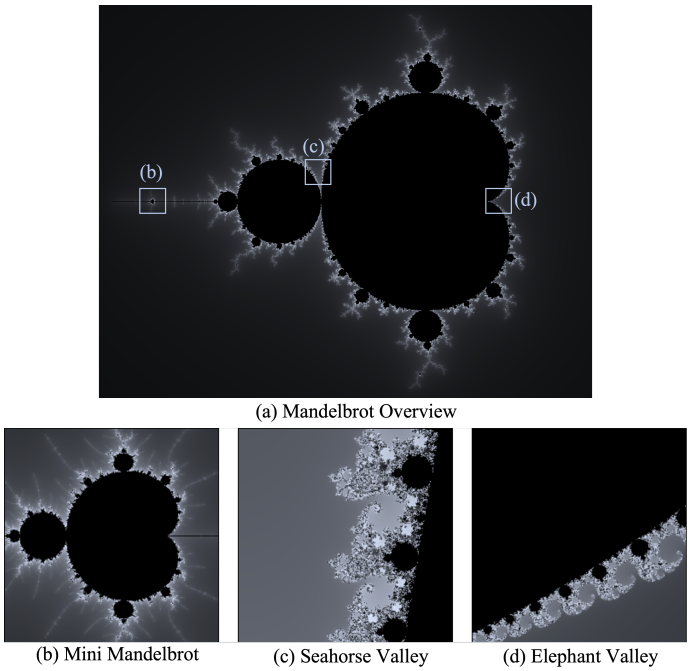


Figure 1. The Mandelbrot set and its locally magnified details.

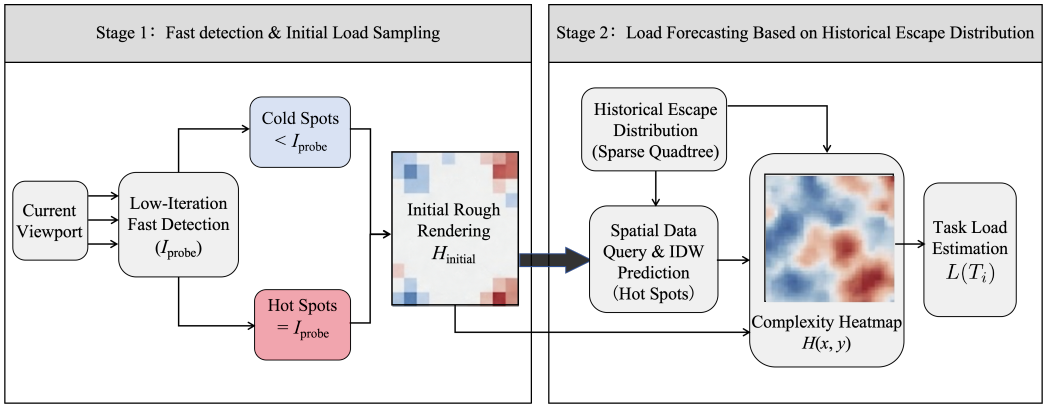


Figure 2. Fractal computational load prediction pipeline.

2.1. Rapid Probing and Initial Load Sampling

An intuitive fact of fractal iteration is that most sample points escape within a small number of iterations, while regions with higher computational complexity tend to concentrate near the boundary. Leveraging this property, a low probing iteration limit I_{probe} is used to traverse the viewport once, where I_{probe} is much smaller than the iteration limit for full rendering. This incurs negligible overhead yet suffices to distinguish between regions with low load and regions with high load spatially. When a sample escapes within very few iterations, its computational load is relatively low, whereas if the sample has not escaped by the probing iteration limit this indicates proximity to the fractal boundary or a region that is hard to converge, implying a relatively high computational load. Representing H_{initial} as a binary load distribution map provides reliable structural information by separating clearly low-load areas from potentially high-load ones, thereby avoiding wasted effort on predicting a large number of low-load regions. Figure 3 shows an example of the binary load distribution of cold and hot spots obtained from the first stage. The viewport is represented as a regular grid of pixel cells. Under the low iteration limit, cells with a value of 0 indicate pixel regions that have already escaped, which are defined as cold spots in this paper. Cells with a value of 1 indicate pixel regions that have not yet escaped,

corresponding to computation areas with potentially high complexity, which are defined as hot spots.

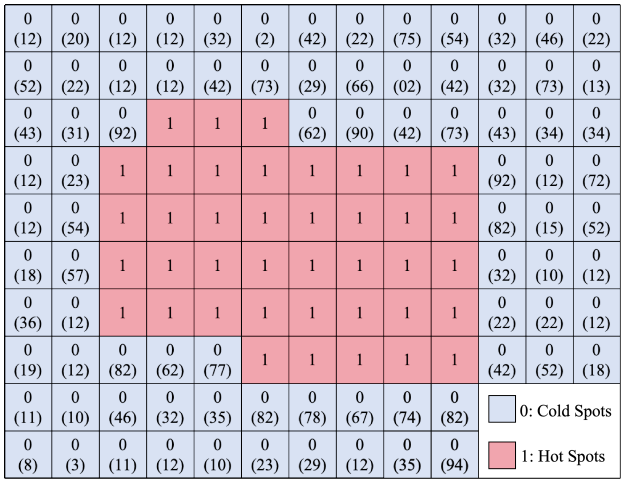


Figure 3. The binary load distribution of cold and hot spots.

2.2. Representation of Historical Escape Distribution Using Quadtree

This paper represents the historical escape distribution as a spatial quadtree. The root node covers the rectangular region R in the complex plane corresponding to the viewport of the previous frame and remains aligned with the current viewport coordinate system, so that coordinate mapping can be performed during interactive panning and zooming. The data structure of any node u in the quadtree is formally defined as a quadruple $\text{Node}_u = \langle B_u, P_u, C_u, N \rangle$, where each component is defined as follows:

1. B_u denotes the rectangular region covered by Node_u, expressed as $B_u = \{(x, y) \mid |x - x_u| \leq w_u,\ |y - y_u| \leq h_u\}$, where (x_u, y_u) is the center coordinate of the region and (w_u, h_u) are the half width and half height, respectively.
2. P_u denotes the set of historical sample points. When Node_u is a leaf node, it stores the set of historical computation samples in the region, $P_u = \{p_1, p_2, \ldots, p_k\}$. Each sample point $p_i = (x_i, y_i, I_i)$ contains its precise position on the complex plane and the corresponding escape iteration count I_i.
3. C_u denotes the child pointers. When a node triggers a split, C_u points to the four child nodes $\{NW, NE, SW, SE\}$ divided by quadrant. If the node is a leaf, this entry is empty.
4. N denotes the capacity threshold, which is used to control the maximum number of samples a leaf node can hold.

In the quadtree data structure, the maximum depth $D_{\max}$ is a key parameter that ensures structural validity and stability. It limits the number of recursive split levels, primarily to control the overall tree size and prevent excessive subdivision under extreme inputs. To support fast decisions in the subsequent query phase, leaf nodes maintain a small amount of summary information in addition to P_u, including the minimum value $I_{\min}$, maximum value $I_{\max}$, and sample count $|P_u|$.

This paper further introduces a sparse storage strategy that explicitly combines step sampling and absolute threshold filtering. This strategy matches the resolution of spatial partitioning to the data density. Hot spot regions where sample points are concentrated trigger deeper subdivision, while cold spot regions remain at shallower levels covering larger areas with fewer nodes, which naturally forms a sparse representation.

Given a step size s, only sample points satisfying $x \bmod s = 0$ and $y \bmod s = 0$ are retained in the candidate set P. To avoid missing local details due to step sampling,

s is treated as a tunable parameter. Based on the binary result from the rapid probing stage, a larger step size is used in cold regions and a smaller step size in hot regions, so that the sample density matches the degree of complexity variation. For each point (x_i, y_i, I_i) in the candidate set, if its historical escape iteration count I_i is below a threshold I_{th}, the point is considered to have low computational load and a limited contribution to characterizing the hot spot structure, and is discarded. If I_i is greater than or equal to I_{th}, the point is retained as a high-value sample and inserted into the quadtree. The threshold I_{th} can be defined in terms of absolute iteration count, for example, within the common range $I_{\text{th}} \in [0.6\, I_{\max}, 0.9\, I_{\max}]$. Combining the two steps above, the dataset that enters the quadtree is expressed as $P' = \{(x_i, y_i, I_i) \mid x \bmod s = 0,\ y \bmod s = 0,\ I_i \geq I_{\text{th}}\}$.

The insertion, update, and splitting process of the quadtree can be summarized as a recursive procedure. After rendering the previous frame, the system obtains a set of new sample points $P = \{p_1, p_2, \ldots, p_k\}$. Step sampling and absolute threshold filtering are first applied to obtain P', which is then inserted into the quadtree as a batch. During insertion, a range check is performed first. If a point lies outside the current node boundary, the procedure returns immediately. If the node is a leaf and not full, the point is added to the point set and the statistics are updated. If the leaf is full and the maximum depth has not been reached, then the node is split and both existing points and new points are redistributed to the child nodes. If the maximum depth is reached, splitting stops and points are appended to the leaf node, with the statistics used to approximate the distribution in that region. Figure 4 illustrates the adaptive subdivision process of the quadtree driven by hot spot regions. As samples are inserted, high-load regions are recursively refined while cold regions remain sparse, forming a spatial storage structure with high resolution in hot spots and low resolution in cold spots.

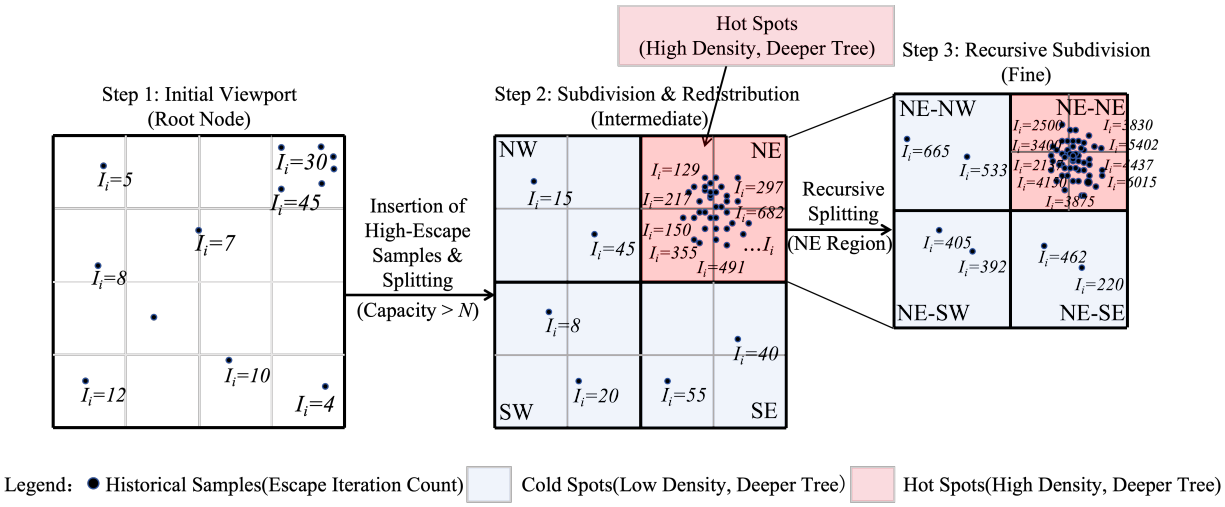


Figure 4. The quadtree spatial representation of historical escape distribution.

Algorithm 1 presents the pseudocode for quadtree insertion and splitting, where `isContains`, `subdivide`, `isLeafNode`, and `updateStats` perform boundary checks, node subdivision, leaf detection, and statistics updates, respectively. Each insertion follows a single root-to-leaf path, giving $O(D)$ time with $D \leq D_{\max}$. When a split occurs, at most N samples in the leaf are redistributed, leading to an additional worst-case cost of $O(N \cdot D_{\max})$. Therefore, inserting n samples costs $O(n \cdot D_{\max})$ on average and is bounded by $O(n \cdot N \cdot D_{\max})$ in the worst case.

Algorithm 1 Insertion and Splitting of Quadtree

Require: Node_u, p (samples to be inserted), $deep$ (depth of Node_u)
Ensure: Node_u

```
procedure INSERT(Node_u, p, deep)
    B_u ← Node_u.B_u, P_u ← Node_u.P_u
    if isContains(B_u, p) is true then
        if isLeafNode(Node_u) is true and (|P_u| < N or deep = D_max) then
            add p to P_u
            updateStats(Node_u)
        else if isLeafNode(Node_u) is true and |P_u| = N and deep < D_max then
            C_u ← subdivide(Node_u)                ▷ Create four child nodes: NW, NE, SW, SE
            add p to P_u
            for p_i in P_u do
                Insert(C_u.NW, p_i, deep + 1)
                Insert(C_u.SW, p_i, deep + 1)
                Insert(C_u.NE, p_i, deep + 1)
                Insert(C_u.SE, p_i, deep + 1)
            end for
            P_u ← {}
            updateStats(Node_u)
        else
            C_u ← Node_u.C_u                       ▷ Node_u already has child nodes
            Insert(C_u.NW, p, deep + 1)
            Insert(C_u.SW, p, deep + 1)
            Insert(C_u.NE, p, deep + 1)
            Insert(C_u.SE, p, deep + 1)
            updateStats(Node_u)
        end if
    end if
end procedure
```

2.3. Computational Load Prediction Based on IDW

This paper employs IDW for local interpolation prediction in hot spot regions. IDW is a deterministic spatial interpolation method in which the fundamental assumption is that attribute values at spatially adjacent locations exhibit strong correlation, i.e., weights decrease as distance increases. For fractal rendering, although the escape iteration count of sample points varies dramatically near boundaries, it still exhibits a certain degree of continuity and local correlation within small-scale neighborhoods. In IDW interpolation prediction, the sample point to be predicted is denoted as $p_{\text{target}}(x, y)$ and its predicted load as $I_{\text{pred}}(x, y)$. The sample points of the historical escape distribution are maintained by the quadtree. To facilitate subsequent derivation, the IDW-related symbols and their meanings are listed in Table 1.

Table 1. IDW-related symbols and their meanings.

Symbol	Meaning
$I_{\text{pred}}(x, y)$	Predicted escape iteration count at the target point
k	Number of neighborhood samples (neighbors participating in interpolation)
λ	Distance decay exponent (weight decay rate control parameter)
d_i	Distance between the target point and the i-th sample point
w_i	Weight assigned to the i-th sample point

The core of IDW lies in estimating the load of $p_{\text{target}}(x, y)$ through weighted aggregation of neighborhood samples, where the neighbor count k determines the scale of historical samples involved in prediction and the distance decay exponent λ controls the rate at which sample influence diminishes with distance. In implementation, the IDW interpola-

tion prediction consists of four steps: neighborhood retrieval, distance computation, weight calculation, and weighted summation.

Step 1: Neighborhood Retrieval. First, the k nearest historical sample points are retrieved from the quadtree for $p_{\text{target}}(x, y)$. The quadtree restricts the search scope to the spatial neighborhood, avoiding linear scans over the entire historical dataset. The retrieved neighborhood set is denoted as $\{p_1, p_2, \ldots, p_k\}$. Since efficient neighborhood retrieval is required for every sample point in hot spot regions, this paper proposes a quadtree-based efficient k-nearest neighbor search algorithm, incorporating branch-and-bound pruning and heuristic search strategies to significantly reduce the search space. Algorithm 2 presents the pseudocode of the quadtree-based efficient k-nearest neighbor search algorithm, where the `minDistanceToB` method computes the minimum bounding box distance from the query point to a node boundary, the `distance` method calculates the distance between two points, and the `sortByDistance` method sorts candidates in ascending order of distance. This heuristic traversal algorithm enables the candidate set to converge to a near-optimal solution as quickly as possible, thereby triggering the pruning condition earlier and effectively eliminating a large number of irrelevant spatial regions.

Algorithm 2 Efficient K-Nearest Neighbor Search Based on Quadtree

Require: Node_u, p_{target}, k, $S_{\max}$ (maximum search distance)
Ensure: C_{res} is the set of the k nearest historical samples to p_{target}

procedure `QUERYKNEAREST`(Node_u, p_{target}, k, $S_{\max}$, $C \leftarrow \{\}$)
  $d_{\min} \leftarrow$ `minDistanceToB`(p_{target}, $\text{Node}_u.B_u$)
  $d_k \leftarrow$ distance of the k-th point in C
  if $|C| \geq k$ and $d_{\min} > d_k$ **then** ▷ Branch and bound pruning
    return
  end if
  if $d_{\min} > S_{\max}$ **then** ▷ Maximum search radius limit
    return
  end if
  if `isLeafNode`(Node_u) is **true then**
    $P_u \leftarrow \text{Node}_u.P_u$
    for p_i in P_u **do**
      $d_p \leftarrow$ `distance`(p_{target}, p_i)
      if $d_p > S_{\max}$ **then**
        continue
      else
        add p_i to C
      end if
    end for
    `SortByDistance`(C)
    if $|C| > 2k$ **then**
      `keep`(C, $2k$) ▷ Only the first $2k$ elements are retained
    end if
  else ▷ Calculate and sort the boundary distances of all child nodes
    $C_u \leftarrow \text{Node}_u.C_u$
    sorted_children $\leftarrow$ `sortByDistance`(C_u)
    for child in sorted_children **do**
      `QueryKNearest`(child, p_{target}, k, $S_{\max}$, C)
    end for
  end if
  $C_{\text{res}} \leftarrow$ the top k elements in C
  return C_{res}
end procedure

With the direct-quadrant insertion of Algorithm 2, building the tree over N samples costs $O(N\,D_{\max})$, and the tree occupies $O(N)$ nodes under the depth cap $D_{\max}$. Average case. Under a μ-regular (bounded local density) distribution, which the exterior potential field satisfies, the branch-and-bound query in Algorithm 2 visits $O(\log N)$ internal nodes and inspects $O(k)$ candidate points, with per-leaf ordering costing $O(k \log k)$; therefore, the expected query complexity is

$$T_{\text{query}} = O(\log N + k \log k) \tag{1}$$

which reduces to $O(k \log N)$ under uniform density. Worst case. For an adversarial degenerate distribution (e.g., all samples collinear or confined to a single leaf), subdivision is capped at $D_{\max}$, the minimum-distance pruning ceases to be effective, and the query degrades to a linear scan $O(N)$. Nonetheless, the depth cap guarantees $O(N)$ space and termination, bounding the worst case rather than leaving it unbounded.

Step 2: Distance Computation. For each neighborhood sample point $p_i = (x_i, y_i, I_i)$, the spatial distance d_i between $p_{\text{target}}(x, y)$ and p_i is computed. This paper adopts the Euclidean distance as the measure of spatial correlation:

$$d_i = \sqrt{(x - x_i)^2 + (y - y_i)^2} \tag{2}$$

The Euclidean distance possesses rotation invariance and isotropy, avoiding bias caused by directional selection. Considering that a large number of predictions are required within hot spot regions, the squared distance

$$d_i^2 = (x - x_i)^2 + (y - y_i)^2 \tag{3}$$

is used in practice to reduce the overhead of square root operations. To ensure numerical stability, when $d_i = 0$, i.e., $p_{\text{target}}(x, y)$ coincides exactly with a historical sample point, $I_{\text{pred}}(x, y) = I_i$ is directly assigned and the prediction terminates.

Step 3: Weight Computation. After obtaining the distances, the sample point weights are computed according to the IDW definition.

$$w_i = \frac{1}{d_i^{\lambda}} \tag{4}$$

The weights decay as distance increases, so closer sample points contribute more to the prediction.

Step 4: Weighted Summation. The load estimate at the target point is obtained via normalized weighted averaging.

$$I_{\text{pred}}(x, y) = \frac{\sum_{i=1}^{k} w_i \cdot I_i}{\sum_{i=1}^{k} w_i} \tag{5}$$

Figure 5 shows a target sample point at the center, with k historical samples retrieved from its neighborhood via the quadtree. These samples are then combined through IDW weighted averaging to obtain the predicted value, where closer samples contribute more significantly.

As shown in Table 2, the three kernels attain comparable prediction accuracy. If the local-smoothness premise of IDW were invalid on oscillatory fractal boundaries, then bilinear interpolation—which relies on the same premise—would fail catastrophically; thus, the fact that all three interpolate effectively confirms that the aggregated fractal load field is locally interpolable. IDW has the lowest mean worker idle time and makespan, whereas

bilinear has a slightly lower mean mean absolute percentage error (MAPE). This shows that the practical benefit of IDW is balanced execution rather than a uniform MAPE advantage at every history density. Moreover, bilinear interpolation requires an axis-aligned regular grid, whereas the historical samples are scattered at arbitrary coordinates in the quadtree after interactive panning and zooming, where bilinear becomes ill-posed and reduces to nearest-neighbor. IDW natively handles such scattered data, which is precisely why it is adopted here.

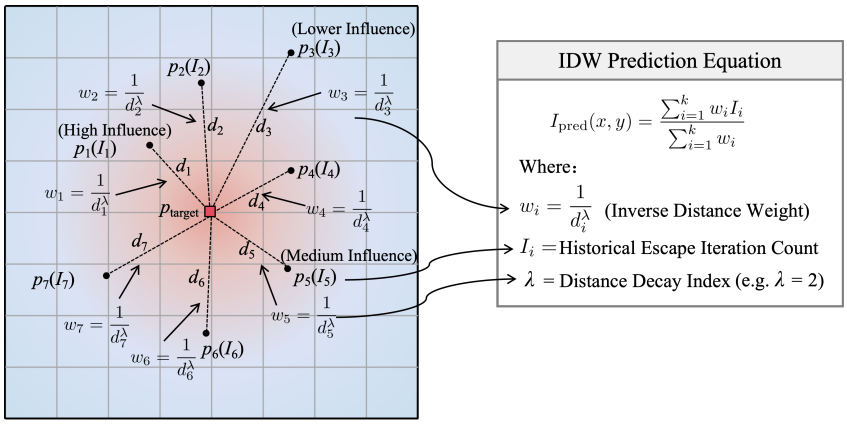


Figure 5. The load prediction process based on IDW. Quadtree-based retrieval of k neighboring historical samples and IDW prediction at the target point, with higher weights assigned to closer samples.

Table 2. Interpolation kernel comparison.

Method	Pred. MAPE	CV_L	Worker Idle	Makespan (ms)
Nearest-Neighbor	27.7%	0.442	3.34%	4929.9
Bilinear	25.8%	0.413	2.29%	4879.6
IDW	27.1%	0.413	1.91%	4857.3

2.4. Construction of Prediction Complexity Heatmap

In implementation, the load prediction results are presented in two different forms. The first is a predicted load feature matrix $M(x,y)$ for parallel task partitioning, where the discretized load values are stored in memory as a two-dimensional array and directly used as input for subsequent task partitioning. The second is a prediction complexity heatmap $H(x,y)$, which maps the discretized load values onto a specific color scheme to form a heatmap, suitable for outputting comparison images during parameter tuning and visualization.

Figure 6 illustrates the visualization and memory layout of the predicted load feature matrix. Here, $M(x,y)$ not only provides an intuitive characterization of the fractal computation complexity distribution but also directly drives parallel task partitioning and scheduling decisions. This paper uses $M(x,y)$ as the basis for the communication-aware task partitioning in the next section, where the predicted load of each task block is defined as the sum of the predicted iteration counts of all sample points within the block. In order for a task block T_i to be partitioned, its total predicted computational load is denoted as $L(T_i)$, defined as

$$L(T_i) = \sum_{(x,y)\in T_i} M(x,y). \tag{6}$$

In addition, $L(T_i)$ serves as the input for task granularity selection and partitioning in the subsequent stage.

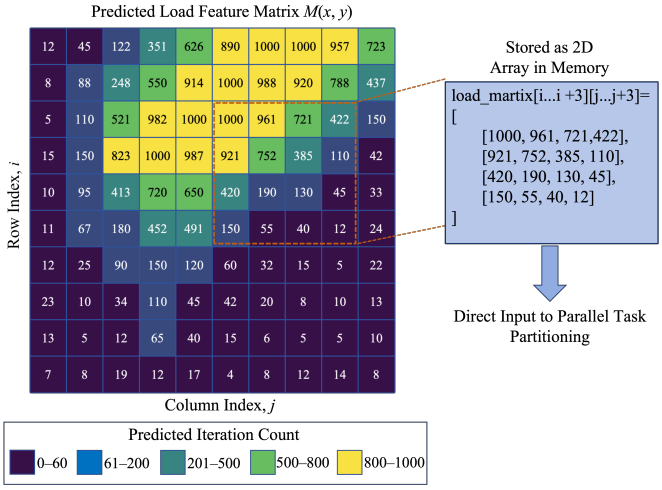
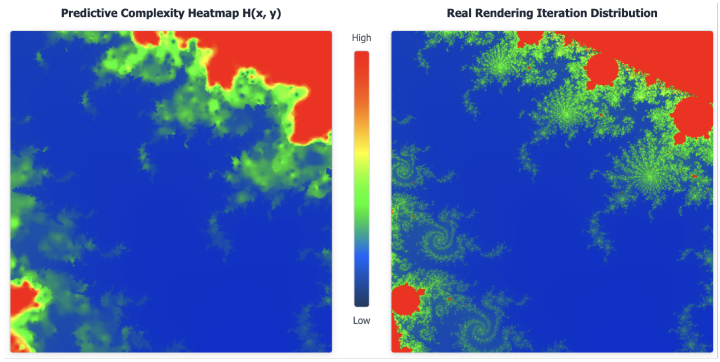
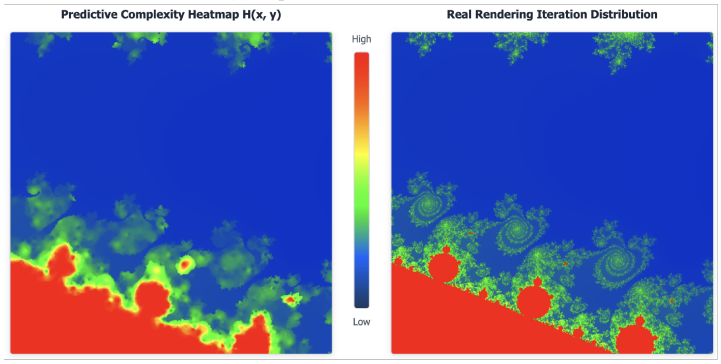


Figure 6. The visualization and memory layout of the predicted load feature matrix.

This paper uses the iteration count as the fundamental measure of complexity. The predicted iteration counts of all pixels are normalized and uniformly mapped to the $[0, 1]$ range. Cool colors represent low load, while warm colors represent high load; darker colors indicate heavier estimated computational load for a given pixel. Figure 7 presents comparison examples of the predicted complexity heatmaps and the actual rendering iteration distributions for two zoomed-in regions of the Mandelbrot set in the web visualization page, corresponding to the Scepter Valley and Triple Spiral regions, respectively. As can be observed, the predicted complexity heatmaps exhibit a high degree of spatial consistency with the actual rendering iteration distributions. The two images show similar distribution trends in terms of the location, extent, and shape of hotspot regions, indicating that the predicted heatmaps can effectively capture the relative load differences and hotspot structures.



(a) Triple Helix of Mandelbrot



(b) Scepter Valley of Mandelbrot

Figure 7. The comparison between $H(x, y)$ and actual iteration distribution.

3. Load Balancing-Oriented Task Partitioning and Scheduling

3.1. Communication-Aware Parallel Task Partitioning

When performing parallel rendering of fractal graphics in the browser, task partitioning is not only influenced by the computational load but also constrained by the communication and scheduling overhead between the main thread and Web Workers. This section introduces a communication-aware cost model that incorporates communication overhead into the task granularity selection process, establishing an analyzable trade-off between computational gain and communication cost in order to determine the optimal task load size.

Let L_{total} denote the total computational load of the viewport to be rendered, and suppose that the system has P workers. The viewport is partitioned into K independent tasks $\{T_1, T_2, \ldots, T_K\}$. The total execution time T_{total} is approximated by three terms:

$$T_{\text{total}} \approx T_{P,\text{ideal}} + T_{\text{comm}} + T_{\text{loss}} \tag{7}$$

where the first term $T_{P,\text{ideal}}$ describes the ideal parallel computation time. Assuming tasks can be evenly partitioned and communication and scheduling overhead are ignored,

$$T_{P,\text{ideal}} = \frac{L_{\text{total}}}{\gamma P} \tag{8}$$

where γ denotes the number of iterations that can be completed per unit time.

The second term T_{comm} describes the interaction cost between the main thread and Web Workers. Let S_i denote the data transfer size of task T_i and C_{transfer} denote the unit data transfer cost coefficient. For fractal rendering tasks, the input description of a single task generally consists of rectangular coordinates and a parameter set. The transfer cost $C_{\text{transfer}} \times S_i$ measured in bytes has limited impact in most scenarios, and can be approximately ignored. A more stable source of overhead comes from the fixed cost C_{fixed} associated with each task distribution and return, which includes message delivery, data cloning, event loop, and scheduling management overhead. Accumulating this process over all tasks yields

$$T_{\text{comm}} \approx \sum_{i=1}^{K} (C_{\text{fixed}} + C_{\text{transfer}} S_i) \approx K \cdot C_{\text{fixed}} \tag{9}$$

To incorporate task granularity into the model, let L_{avg} denote the average load of a single task. There is an approximate relationship $K \approx \frac{L_{\text{total}}}{L_{\text{avg}}}$, which yields

$$T_{\text{comm}}(L_{\text{avg}}) \approx \frac{L_{\text{total}}}{L_{\text{avg}}} \cdot C_{\text{fixed}} \tag{10}$$

The third term T_{loss} describes the imbalance loss caused by discrete scheduling. Even when the average task loads are similar, K and P are still difficult to match exactly in a discrete sense. In the final few scheduling rounds, some workers may finish early and remain idle while waiting. This paper uses an imbalance factor σ ($0 < \sigma < 1$) to describe this average idle rate, and approximates its contribution to the total time as

$$T_{\text{loss}} = \sigma \cdot \frac{L_{\text{avg}}}{\gamma} \tag{11}$$

In this formulation, the larger the task granularity, the longer the single-round execution time and the more the tail waiting is amplified. When tasks are sufficiently fine, the impact

of σ on the total time diminishes but T_{comm} increases. This is precisely why communication cost must be incorporated into task partitioning.

Combining the three terms yields the comprehensive cost function with respect to L_{avg}:

$$f(L_{\text{avg}}) = \frac{L_{\text{total}}}{\gamma P} + \frac{L_{\text{total}}}{L_{\text{avg}}} \cdot C_{\text{fixed}} + \sigma \cdot \frac{L_{\text{avg}}}{\gamma} \tag{12}$$

Taking the derivative of $f(L_{\text{avg}})$ with respect to L_{avg} and setting it to zero gives

$$\frac{df(L_{\text{avg}})}{dL_{\text{avg}}} = -\frac{L_{\text{total}} \cdot C_{\text{fixed}}}{L_{\text{avg}}^2} + \frac{\sigma}{\gamma} = 0 \tag{13}$$

Solving for the optimal average task load L_{opt},

$$L_{\text{opt}} = \sqrt{\frac{L_{\text{total}} \cdot C_{\text{fixed}} \cdot \gamma}{\sigma}} \tag{14}$$

A larger fixed communication cost C_{fixed} leads to a larger L_{opt}, reducing the number of tasks and communication rounds. A larger imbalance factor σ leads to a smaller L_{opt}, mitigating the risk of tail waiting. The value of σ has both theoretical bounds and requires calibration through system measurements. Starting from the worst case of discrete scheduling, only one worker is active in the final round, while the remaining $P - 1$ workers are idle, at which point the average idle rate reaches its upper bound:

$$\sigma_{\max} = \frac{P - 1}{P} \tag{15}$$

This upper bound is primarily used to constrain the reasonable range of the model, preventing parameter values that are inconsistent with the scheduling mechanism. In actual operation, σ is influenced by the number of tasks, task load distribution, and scheduling strategy.

This paper adopts a recursive partitioning strategy in which task partitioning is modeled as an executable spatial subdivision process. This step aims to construct a baseline partition grid, avoiding the invalid traversal overhead caused by recursing from the full canvas scale. Figure 8 illustrates the adaptive recursive task partitioning process based on complexity heatmap backtracking, where the right side of the figure presents the algorithm pseudocode for recursive task partitioning. The rendering canvas is first divided into a set of regular initial task blocks, each covering $M \times M$ heatmap grid cells. For any region R to be processed, the algorithm first obtains $L(R)$ by accumulating the complexity heatmap, then evaluates it against L_{opt} set by the communication cost model:

1. When $L(R) \leq \theta \cdot L_{\text{opt}}$, the region is judged to have a reasonable estimated load and is added as an independent task block to the final scheduling set. Here, θ is a relaxation factor that prevents over-subdivision, achieving a trade-off between load balancing and communication overhead.
2. When the above condition is not satisfied, if $S(R) \leq S_{\min}$, then the region is added as an independent task block to the final scheduling set. Here, $S(R)$ denotes the size of region R, while $S_{\min}$ is the minimum threshold for task block subdivision set by the system to prevent excessive task granularity that would cause a surge in communication overhead.
3. When neither of the above conditions is satisfied, i.e., when $L(R)$ significantly exceeds L_{opt}, the system quad-subdivides region R along its geometric center in both the horizontal and vertical directions, generating four sub-regions, each of which enters the next round of recursive evaluation.

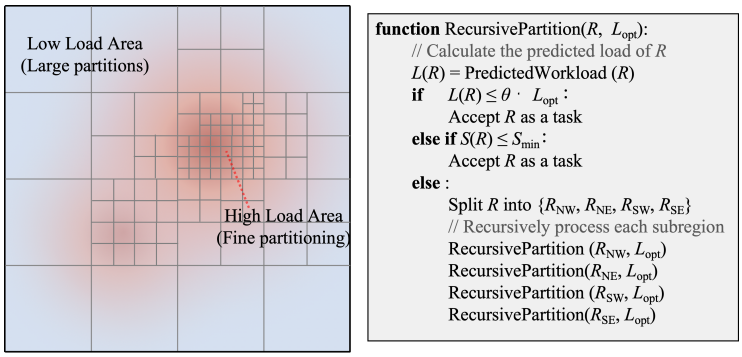


Figure 8. The recursive task partitioning process. The canvas is first divided into regular blocks. For each region, the algorithm estimates load from the complexity heatmap, accepts it if thresholds are met, or otherwise quad-splits and recurses.

The final generated task set consists of rectangular blocks of varying sizes, with the load balance among task blocks determined by $H(x, y)$.

3.2. Interaction-Aware Adaptive Priority Scheduling

Interaction events (viewport panning, zooming, and focus shifting) are captured by the event system and translated into viewport state updates. The scheduler maintains the viewport state $V(t)$ to avoid the overhead of full re-ranking on every individual event. Its awareness follows the basic principles of prioritizing visibility, prioritizing center responsiveness, and prioritizing progressive presentation from outline to detail.

Based on the above principles, this paper represents the priority of task T_i as a score S_i that is modeled using a weighted linear combination. The priority factors include viewport relevance V_i, center deviation D_i, and level weight L_i. Here, V_i indicates whether the task intersects with the current viewport. It takes a value of 1 if the task region lies within the viewport or overlaps with it and a value of 0 otherwise, and is used to push invisible tasks to the end of the queue or treat them as low-priority candidates. In addition, D_i represents the normalized distance from the task center to the viewport focus; the closer the distance, the higher the user perception sensitivity. Let the viewport focus be (x_c, y_c) and the task block center be (x_i, y_i). Then,

$$D_i = \frac{\sqrt{(x_i - x_c)^2 + (y_i - y_c)^2}}{D_{\text{diagonal}}} \tag{16}$$

where D_{diagonal} is the viewport diagonal length, used to constrain D_i within the range $[0, 1]$. The level weight L_i describes the detail level of the task block. This paper uses the recursion depth of the partitioning process as the metric; a shallower level indicates a larger block closer to the global outline, while a deeper level indicates a smaller block oriented toward local details. To reflect the outline-first strategy, $1/L_i$ is used so that shallow-level tasks receive higher priority. Combining the above factors, the task priority score is defined as follows:

$$S_i = \omega_1 \cdot V_i + \omega_2 \cdot (1 - D_i) + \omega_3 \cdot (1/L_i) \tag{17}$$

where ω_1, ω_2, ω_3 are weight coefficients satisfying $\omega_1 + \omega_2 + \omega_3 = 1$. The three weight coefficients control the relative contributions of the three priority factors, enabling the scheduler to make scheduling decisions with different biases. For example, if ω_1 is too small, invisible tasks may still occupy high priority, causing the queue to be clogged with outdated tasks. If ω_2 is too small, the response speed to the central region is reduced. If ω_3 is too large, shallow-level tasks may persistently suppress detail tasks.

3.3. Cancellation Mechanism for Outdated Tasks

The existence of outdated tasks is pervasive, especially during continuous panning, where the viewport center moves continuously and task blocks generated early often leave the viewport before completion. Continuing to execute such tasks introduces additional communication and compositing overhead, eventually leading to task queue congestion.

When a task is identified as outdated, it enters a cancellation or discarding process, as follows:

1. For tasks not yet scheduled for execution, the cancellation process is relatively straightforward. Upon a viewport update, the scheduler filters the pending task queue, marking tasks satisfying $v_i < v_{\text{cur}}$ as invalid and removing them from the queue. If a task belongs to an old version but its region R_i still overlaps with the new viewport, it may be retained and its priority recalculated, allowing it to participate as a potentially usable candidate in subsequent scheduling.
2. For tasks currently executing in a worker, this paper adopts a soft cancellation approach. After a version update, the main thread sends a cancellation signal to the target worker. The worker periodically checks its computation loop to determine whether the current task version still matches that of the main thread. If the task version is detected as outdated, the current task computation is terminated early and a canceled status is returned.

3.4. Scheduler Execution Flow and Strategy

The scheduler takes four inputs: the current viewport state (boundaries, center, zoom level, version v_{cur}), the task set $\{T_1, T_2, \ldots, T_k\}$ (each with region R_i, load $L(T_i)$, level L_i, and version v_i), runtime resource status (worker pool, queue length, scheduling window), and policy parameters (ω_1, ω_2, ω_3, cancellation cycle). Its core functions include task validity management, priority computation with queue re-ranking, and task dispatch.

The scheduling process combines event-driven and cyclic modes. Viewport updates trigger version increments, task pool cleanup, and priority recomputation. Worker result returns trigger consistency checks and worker release. When interaction ceases or the viewport stabilizes, a global re-ranking recalculates priorities for still-valid tasks, progressively refining details. This avoids redundant traversals under high-frequency events while ensuring rapid queue adaptation to viewport changes, prioritizing visible regions with limited workers.

Figure 9 illustrates the complete execution flow of the scheduler. Starting from user interaction, the main thread updates the viewport state and generates a complexity heatmap, followed by task partitioning to obtain the task set $\{T_1, T_2, \ldots, T_k\}$. Upon receiving the task set, the scheduler computes priorities and performs sorting while simultaneously issuing soft cancellation for old-version tasks still in execution and removing outdated tasks from the pending queue. The execution phase employs a worker pool mechanism in which idle workers are acquired from a pre-maintained reusable pool, dispatched with tasks, and returned to the pool upon completion for subsequent reuse. The scheduler then enters a dispatch loop: when a worker becomes idle, it fetches the highest-priority task for execution. Upon result return, version consistency is verified and only results matching the current version proceed to rendering and compositing. If further interaction occurs during rendering, the version update invalidates a portion of the current queue, prompting the scheduler to re-filter and re-rank. This forms a closed loop of interaction update, queue adjustment, continuous dispatch, and incremental rendering. The key to this flow lies in suppressing outdated computation under continuous interaction while progressively refining details after the viewport stabilizes, thereby translating parallel computing resources into perceptible frame update speed.

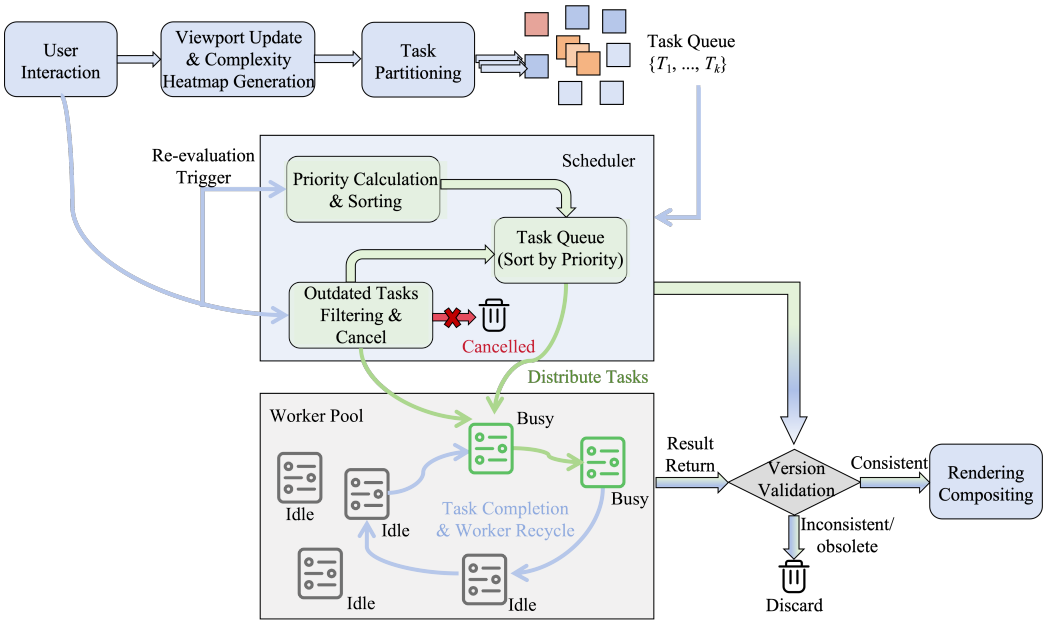


Figure 9. The execution process of the scheduler.

Table 3 summarizes the key modules in the proposed framework, including their key ideas and roles in load prediction, task partitioning, and scheduling.

Table 3. Summary of the proposed framework modules.

Module	Key Idea	Purpose
Rapid probing	Low-iteration escape test with I_{probe}	Fast localization of potential high-load regions; avoids spending prediction effort on cold areas.
Historical escape distribution	Adaptive subdivision with step sampling and threshold filtering; bounded by $D_{\max}$	Stores high-value historical iterations for fast neighborhood queries; naturally sparse in cold regions.
Quadtree-based KNN query	Branch-and-bound search with radius limit $S_{\max}$	Accelerates local neighborhood retrieval for dense hot-spot prediction.
IDW load prediction	Inverse Distance Weighting with decay exponent λ	Predicts local iteration complexity on scattered samples.
Communication-aware load model	Cost trade-off between message overhead and tail idle	Selects target task load L_{opt} to avoid over-partitioning and straggler blocking.
Interaction-aware scheduling	Score based on visibility, center distance and level.	Improves perceived responsiveness under interactions by prioritizing visible and central regions.
Stale-task management	Queue filtering + Worker soft cancellation	Avoids wasted computation on outdated tasks during high-frequency pan/zoom.

4. Experimental Validation and Result Analysis

4.1. Hardware and Software Environment

All experiments were conducted on a unified hardware environment to eliminate variations caused by environmental differences. The hardware setup consisted of an Apple

MacBook Pro (M4) with 16 GB of RAM and 512 GB of storage, Apple Inc., Cupertino, CA, USA, running macOS Sequoia (15.5). The testing environment was Google Chrome (version 142.0+), Safari (20621.2.5.11.8), and Firefox (152.0.6), leveraging the JavaScript execution and Web Worker multi-threading capabilities of each browser's respective engine.

This experiment employs our previously proposed Performance Evaluation for Parallel Web Applications (PEPWA) framework [22]. This tool collects execution times of individual Web Worker subtasks through instrumentation, then aggregates them into six parallel performance metrics: speedup, efficiency, parallel efficiency, load balance, parallel impedance, and worker average performance, thereby presenting computation, communication, and scheduling overheads in a visualized manner.

4.2. Parameter Selection and Configuration

To achieve an optimal trade-off between computational efficiency and prediction accuracy, it is necessary to scientifically configure the core parameters of the quadtree structure, load prediction model, and scheduling strategy. Based on hardware performance constraints, spatial interpolation theory, and preliminary stress test results, this subsection determines the system hyperparameter configuration. Table 4 presents the core parameter settings of the system.

Table 4. System core parameter configuration

Parameter Name	Symbol	Value
Quadtree Maximum Depth	$D_{\max}$	10
Quadtree Split Threshold	N	32
IDW Distance Decay Exponent	λ	2
KNN Neighborhood Sample Count	k	8
Imbalance Factor	σ	Dynamic
Fast Probe Escape Iterations	I_{probe}	50
Fixed Communication Cost (ms)	C_{fixed}	0.01
Iterations Completable per Unit Time (ms)	γ	160,000

4.3. Load Balance Validation of Task Partitioning

Given the computational load imbalance in parallel fractal rendering, this section designs three groups of experiments to compare the load differences among different parallel task partitioning methods. Experiment 1 adopts static partitioning, which divides the viewport into 144 fixed task blocks of 120×120 pixels. Experiment 2 employs a fine-grained dynamic partitioning, reducing the task size to 16×16 pixels to construct 8100 high density queues. Experiment 3 uses the communication-aware load prediction partitioning proposed in this paper. The experiments use a deep region of the Mandelbrot set with extremely high computational complexity as the test scenario, centered at $(-0.74, 0.13)$ with a zoom factor of $20{,}000\times$. This region features high-iteration boundaries and low-iteration backgrounds, exhibiting significant spatial clustering of computational load. Figure 10 compares $H(x, y)$ and the actual rendering iteration distribution for Deep Seahorse Valley of the Mandelbrot set. The results show that $H(x, y)$ is highly consistent with the actual computational complexity in spatial distribution.

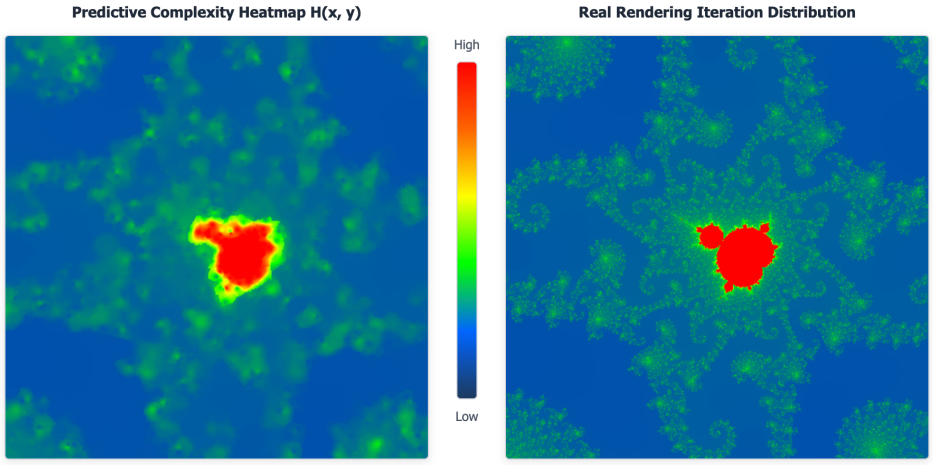


Figure 10. The comparison between $H(x, y)$ and actual iteration distribution of Deep Seahorse Valley.

To provide an interpretable measure of load balance for the task partitioning results of each method, this paper introduces a load dispersion metric on the task set $\{T_1, T_2, \ldots, T_k\}$. Let $L(T_i)$ denote the predicted load of each task block. The coefficient of variation can then be used to describe the relative fluctuation of task loads:

$$\bar{L} = \frac{1}{K} \sum_{i=1}^{K} L(T_i) \tag{18}$$

$$CV_L = \sqrt{\frac{1}{K} \sum_{i=1}^{K} \frac{(L(T_i) - \bar{L})^2}{\bar{L}}} \tag{19}$$

where a smaller CV_L indicates lower task load disparity and consequently a reduced likelihood of inter-thread waiting.

Table 5 presents the partitioning results of the three parallel task partitioning methods when rendering the Deep Seahorse Valley region. The experimental data demonstrate that the proposed load prediction partitioning strategy reasonably controls the total number of tasks while maintaining a low prediction error of 1.5%. It achieves the lowest load coefficient of variation $CV_L = 0.3276$, significantly outperforming both the conventional static and fine-grained dynamic partitioning methods in terms of load balancing.

Table 5. The results of different task partitioning strategies in Deep Seahorse Valley.

Partitioning Method	Task Count	Load Range	Real Load	CV_L	Prediction Error
Static	144	[2,206,419, 144,000,000]	27,530,916	1.7014	–
Fine-grained Dynamic	8100	[20,218, 2,560,000]	485,840	1.8540	–
Load Prediction	1749	[163,952, 8,066,133]	2,266,696	0.3276	1.5%

Figure 11 compares the normalized task load distributions of static and fine-grained partitioning under the same viewport. As shown in Figure 11a, the fixed grid of static partitioning cannot adapt to the varying fractal complexity. The 144 generated tasks exhibit severe load skew with $CV_L = 1.7014$, where a few tasks dominate most computation, causing some workers to block while others remain idle. As shown in Figure 11b, fine-grained partitioning reduces task size to 8100 tasks, yet load imbalance persists, with CV_L even increasing to 1.8540.

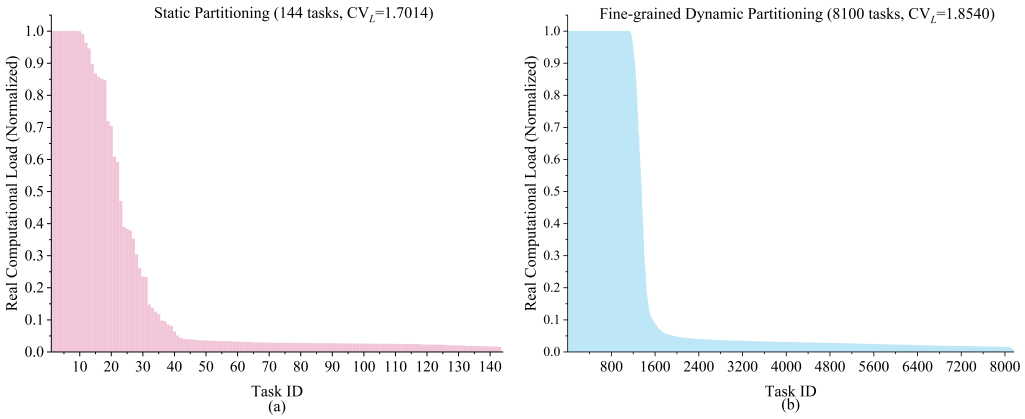


Figure 11. The task load distribution of static and fine-grained dynamic partitioning. Normalized task-load distributions under the same viewport: (**a**) static partitioning (144 tasks, $CV_L = 1.7014$) and (**b**) fine-grained partitioning (8100 tasks, $CV_L = 1.8540$).

As shown in Figure 12, the proposed load prediction and communication-aware partitioning method generates 1749 tasks. The load distribution is notably smoother, with most tasks concentrated around the optimal load $L_{\text{opt}} = 2{,}865{,}090$, and CV_L being reduced to 0.3276. These results validate the effectiveness of the proposed strategy in adaptively adjusting task granularity according to the distribution of computational complexity. In this way, it reduces communication overhead from fine-grained partitioning while avoiding local overload from static partitioning, thereby achieving effective load balancing among workers.

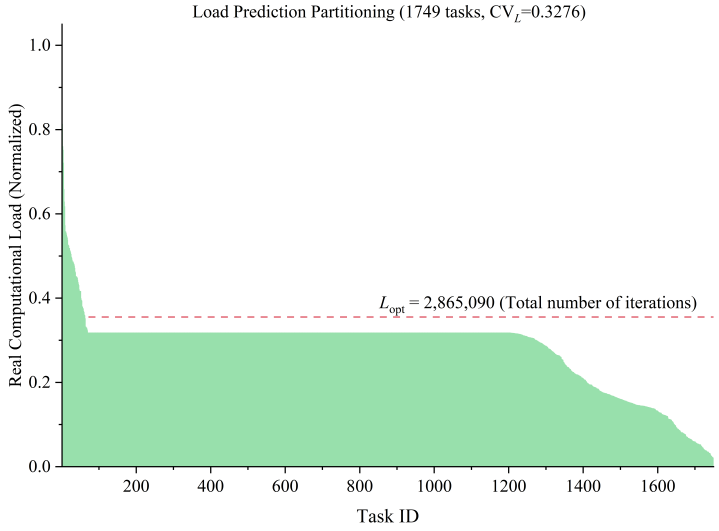


Figure 12. The task load distribution of load prediction partitioning.

4.4. Response Latency Analysis of Interaction-Aware Scheduling

This subsection validates the rendering smoothness of the interaction-aware scheduling strategy at the visual center under task queue congestion caused by frequent drag and zoom operations. The experiment compares the visual fluency of traditional FIFO and adaptive priority scheduling strategies for rendering the Deep Seahorse Valley region during continuous interactions, using center-to-load latency (CTL) as the evaluation metric. A script-driven standardized interaction sequence simulates three rightward translations followed by a rapid zoom-in on the central region within 5 s, with controlled interaction parameters. The reported CTL values are averaged over multiple runs. In the adaptive priority scheduling experiments, ω_1, ω_2, and ω_3 are set to 0.5, 0.3, and 0.2, respectively.

Figure 13 shows the CTL temporal evolution of the two scheduling strategies during continuous interactions. Under FIFO, obsolete tasks accumulate rapidly, causing resource contention; the CTL curve fluctuates severely, peaking above 220 ms, resulting in noticeable stuttering. In contrast, the proposed adaptive priority strategy dynamically reprioritizes tasks and discards obsolete ones, keeping CTL stable below 100 ms throughout and mostly under 50 ms. This confirms the effectiveness of the interaction-aware scheduling and obsolete task cancellation mechanism in ensuring smooth interaction under high-frequency operations.

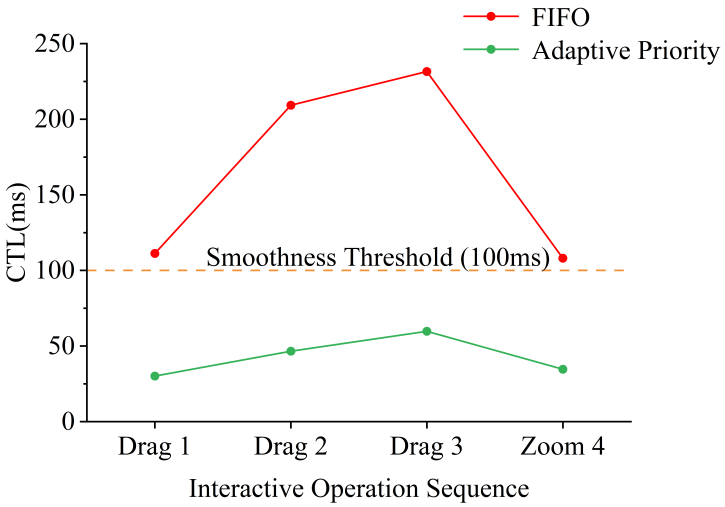


Figure 13. CTL under FIFO and adaptive priority scheduling during continuous interactions.

4.5. Rendering Performance Evaluation of Task Partitioning and Scheduling

To further validate the generality of the proposed task partitioning and scheduling scheme, we extend the original Mandelbrot-based evaluation by adding a Julia set rendering case. Specifically, we evaluate the Julia San Marco configuration with $c = -0.75$, Scale $= 0.24$, and $I_{\max} = 8000$. This extension is designed to verify whether the observed performance trade-offs among static partitioning, fine-grained dynamic partitioning, and the proposed load prediction partitioning persist under a different fractal family and parameter regime, while keeping the experimental protocol consistent with the original Mandelbrot study.

Figure 14 reports the execution-time composition of the Julia workload in Google Chrome, with two subplots corresponding to $P = 4$ and $P = 8$ workers, respectively. The results show a consistent pattern across browsers and worker counts. Static partitioning exhibits substantial worker idle time, indicating straggler blocking due to heterogeneous iteration distribution within the viewport. Fine-grained dynamic partitioning reduces worker idle time but incurs markedly higher communication overhead, which becomes a dominant contributor to total time as the number of tasks increases. In contrast, the proposed load prediction partitioning introduces an explicit load prediction cost, yet systematically reduces communication overhead relative to fine-grained partitioning while also suppressing worker idle time relative to static partitioning. This shifts the overall time composition toward a more balanced profile, yielding a more favorable end-to-end efficiency under both $P = 4$ and $P = 8$.

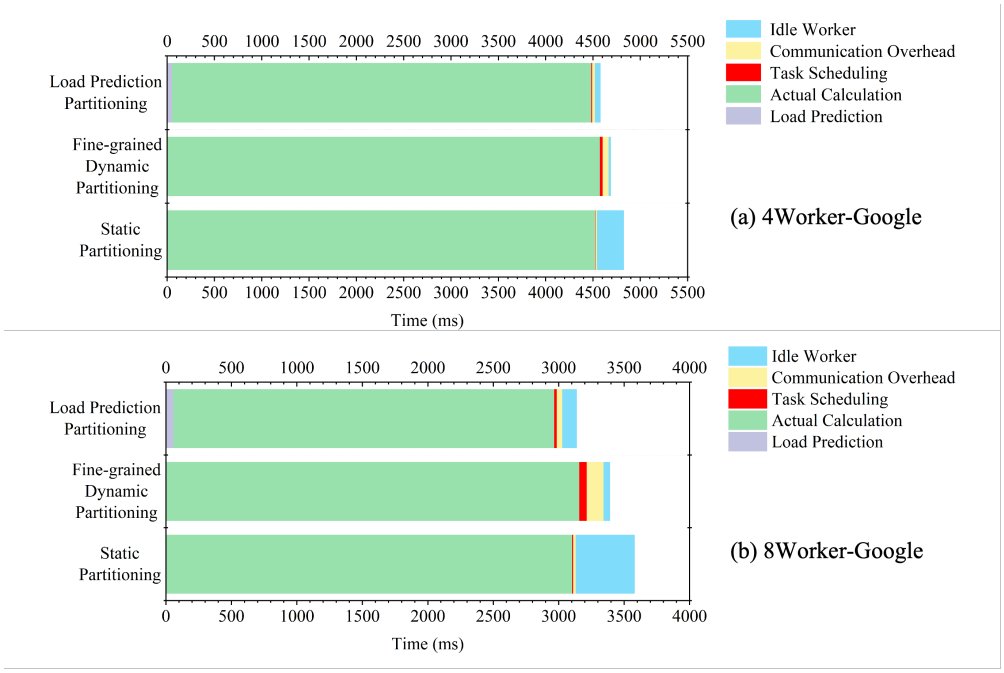


Figure 14. Execution time composition under different partitioning strategies for Julia set in Google.

Figure 15 reports the execution-time composition of the Mandelbrot workload in Google Chrome, with two subplots corresponding to $P = 4$ and $P = 8$ workers, respectively. The Mandelbrot results exhibit the same structural trade-off observed in the Julia experiments. Static partitioning remains limited by long-tail completion and worker idling, while fine-grained dynamic partitioning alleviates idling but amplifies message-passing costs under the Web Worker isolation model, leading to large communication overhead that grows with increased parallelism.

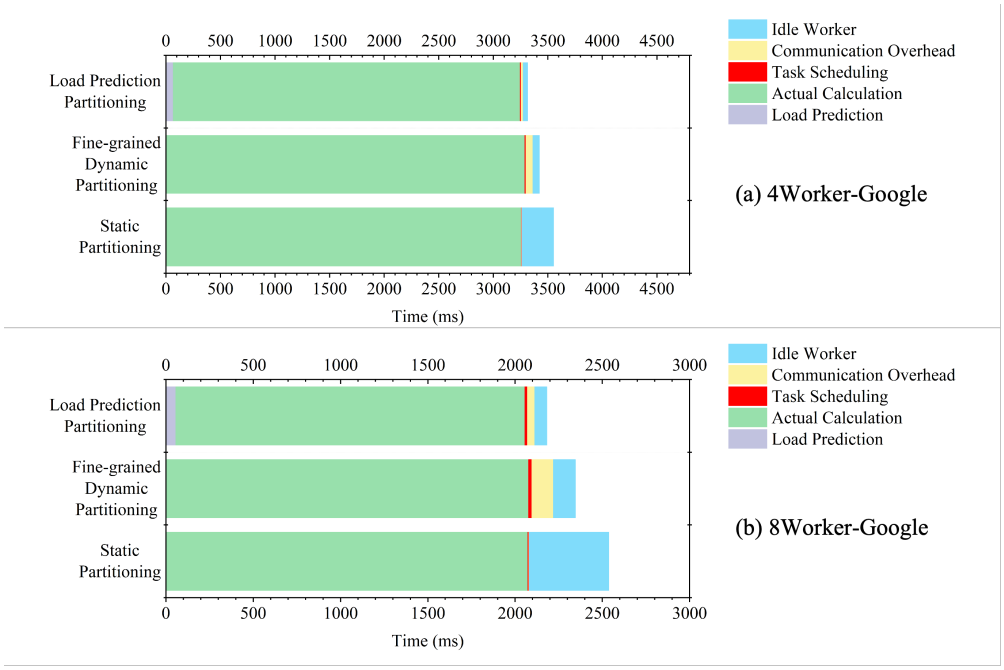


Figure 15. Execution time composition under different partitioning strategies for Mandelbrot set in Google.

The above results were collected under Google Chrome. Identical performance trends can be observed on Safari and Firefox for both fractal sets; full execution time breakdowns for these two browsers are presented in Appendix A (Figures A1–A4).

Taken together, the Julia and Mandelbrot cases provide consistent evidence that the proposed method is effective under realistic browser constraints. Across two fractal families and three browsers, the proposed load prediction partitioning achieves a better end-to-end balance between load balance and communication cost than the two baselines. It mitigates straggler-induced worker idling compared with static partitioning, and suppresses the communication overhead explosion compared with fine-grained dynamic partitioning. Although the proposed method introduces an additional prediction stage, the measured time compositions indicate that this overhead is compensated by the reduction in idle and communication costs, leading to improved overall rendering efficiency under both $P = 4$ and $P = 8$ workers.

Figure 16 uses a multi-dimensional performance radar chart to show the comprehensive performance of the three strategies in a parallel environment, where the radar area directly reflects the overall system performance. The experiment is quantified using PEPWA, a parallel web application performance evaluation framework. The data indicate that the load prediction partitioning method proposed in this paper achieves the best results on multiple core metrics. The proposed method effectively eliminates computational bottlenecks under the condition with four workers, raising the speedup from 3.1 under static partitioning to 3.67, an improvement of 18.4%. With eight workers, our strategy achieves a parallel execution efficiency of 0.75, a load balancing degree of 0.94, and a parallel efficiency of 0.77, significantly reducing worker idle time. Compared with the other two partitioning methods, the proposed method eliminates computational bottlenecks while avoiding the communication overhead caused by a massive number of fine-grained tasks. The notably outward extension of the performance polygon in the radar chart validates the feasibility and superiority of this mechanism in improving browser-side parallel rendering efficiency.

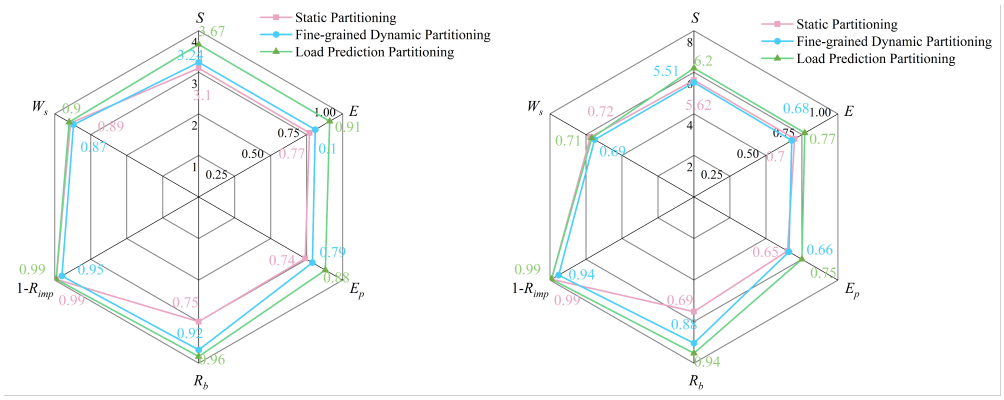


Figure 16. Parallel performance metrics under different partitioning strategies for the Mandelbrot in Google.

Table 6 summarizes the three partitioning strategies compared in Section 4 from the perspective of overhead–balance trade-offs.

Table 6. Strengths and weaknesses of the three task partitioning strategies evaluated in this paper.

Model	Strengths	Weaknesses
Static partitioning	Simple and stable; low scheduling overhead with few fixed tiles.	Severe load imbalance for heterogeneous fractals; long-tail stragglers cause heavy Worker idle.
Fine-grained dynamic partitioning	Alleviates straggler blocking; partially improves resource utilization.	Massive tasks bring high communication and scheduling overhead under Web Worker isolation.
Load prediction partitioning	Adaptive granularity via prediction and communication-aware model; balances load and task count.	Adds prediction overhead and extra parameters; accuracy drops under rapid zooming.

4.6. Ablation of Core Pipeline Components

We conducted a real-browser end-to-end ablation on the Deep Seahorse Valley region at 1920 × 1080, $P = 8$, and I_{max}=15,000. Each condition creates fresh web workers, renders and discards one warm-up frame, and records the subsequent frame. Thus, the reported wall-clock time includes actual worker execution, message transfer, and main-thread dispatch rather than a scheduling simulation. The variants are Full, NoProbe, NoIDW, NoModelAdaptive, and StaticBaseline. The latter uses fixed 120-px uniform tiles, while NoModelAdaptive retains probe, history, and IDW but disables the L_{opt}-based granularity model and recursive subdivision.

Table 7 provides two descriptive observations. First, removing IDW, disabling model-based adaptive subdivision, or replacing the pipeline with uniform static tiling produces larger mean wall times than the full variant in this run. In particular, the mean for the full variant was 22.9% below StaticBaseline and 23.3% below NoIDW. Second, the task count is itself an outcome of the intervention: Full generated 540 tasks, NoProbe generated 1020, and StaticBaseline generated 144.

Table 7. Preliminary ablation test on Deep Seahorse Valley, conducted in Google Chrome (1920 × 1080, $P = 8$, $n = 10$). Values are presented as mean ± standard deviation across independent runs. Idle indicates the average residual time per worker.

Variant	Tasks	Wall Time (ms)	Execution (ms)	Idle (ms)
Full	540	5383.9 ± 137.0	5319.0	6.9
NoProbe	1020	5387.4 ± 469.2	5326.8	18.5
NoIDW	540	7017.1 ± 164.8	6952.6	10.1
NoModelAdaptive	135	7377.6 ± 280.2	7303.7	457.0
StaticBaseline	144	6985.1 ± 107.9	6983.3	260.1

For the Seahorse Valley deep-zoom scenario tested in Google Chrome, the preliminary conclusion is that the complete pipeline outperforms both the uniform static configuration and the two ablated variants with comparable end-to-end workload. The full variant reduces the mean wall time by 22.9% relative to StaticBaseline and by 27.0% relative to NoModelAdaptive; it also has far smaller mean idle residuals (6.9 ms versus 260.1 and 457.0 ms, respectively). The comparison between NoIDW and the full variant is the cleanest available direct contrast, since both generated 540 tasks; in this ablation, removing IDW increased the mean wall time by 30.3%. The NoProbe result instead indicates that probe can alter partition expansion (1020 versus 540 tasks) without yielding a visible difference in mean time during this pilot.

5. Conclusions

This paper addresses the computational load imbalance and long-tail effect caused by uneven task partitioning in browser-based fractal visualization as well as the exacerbated response latency introduced by high-frequency user interactions. The study focuses on load prediction, task partitioning, and scheduling for fractal rendering. First, a load prediction method combining rapid probing with historical escape distributions is proposed to construct the complexity distribution of the fractal rendering task within the current viewport. Second, a computational and communication cost model is built and adaptive recursive partitioning is performed based on the load prediction results. An interaction-aware priority scheduling mechanism and a stale-task soft cancellation strategy are also designed. Finally, taking the Mandelbrot set as the experimental subject, the PEPWA tool is used for experimental validation and analysis. The experimental results show that the proposed method outperforms the compared methods in terms of parallel performance,

load balancing, and response latency and that it significantly improves system response speed and interaction fluidity in high-frequency interaction scenarios.

Although the proposed load prediction and adaptive task partitioning methods alleviate the load imbalance problem in parallel fractal rendering to a certain extent, scale mismatch between the historical escape distribution and the current viewport may occur under rapid zooming operations, which reduces prediction accuracy. Future work will explore the introduction of a confidence evaluation mechanism based on zooming thresholds and an instantaneous compensation strategy to enhance robustness.

Author Contributions: Conceptualization, J.L. and L.M.; methodology, L.M.; software, L.M.; validation, Y.W. and K.H.; formal analysis, L.M.; investigation, J.L.; resources, J.L.; data curation, Y.W.; writing—original draft preparation, L.M.; writing—review and editing, J.L. and K.H.; supervision, H.G.; project administration, H.G.; funding acquisition, H.G. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by the National Natural Science Foundation of China (No. 62407011) and the Guangzhou Academician and Expert Workstation (No. 2024-D003).

Data Availability Statement: No new data were created or analyzed in this study. Data sharing is not applicable to this article.

Conflicts of Interest: The authors declare no conflicts of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

Appendix A. Execution Time Results for Additional Tested Browsers

This appendix provides the complete execution time composition of the three task partitioning strategies on the Julia set and Mandelbrot set in the Safari and Firefox browsers, corresponding to the analysis in Section 4.5. All experimental configurations and parameters are consistent with those in the main text.

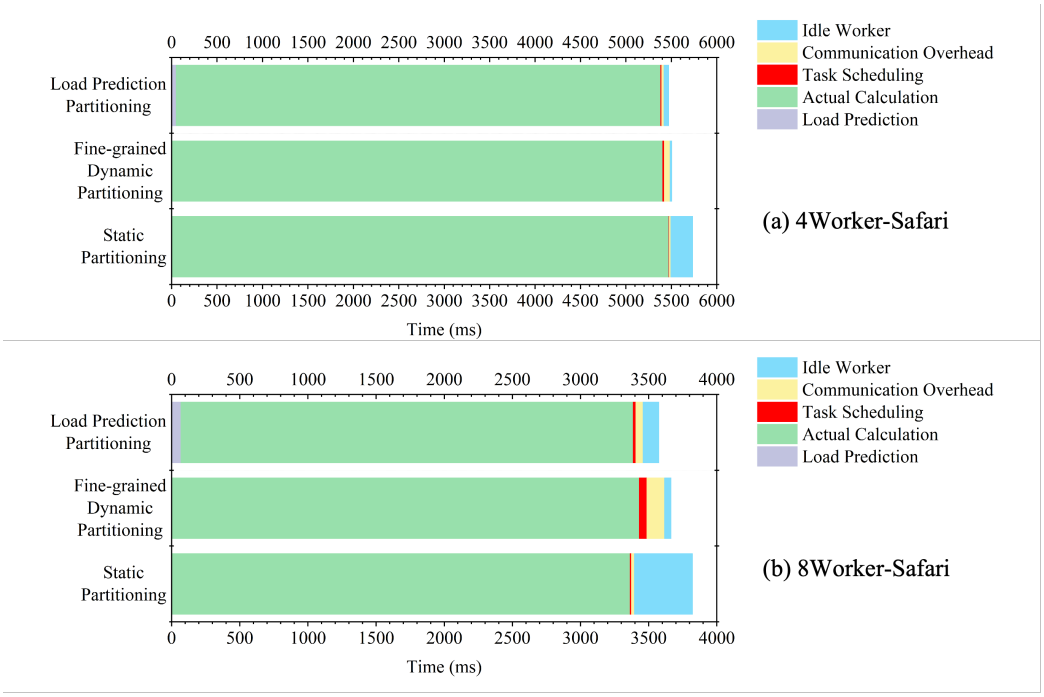


Figure A1. Execution time composition under different partitioning strategies for Julia set in Safari.

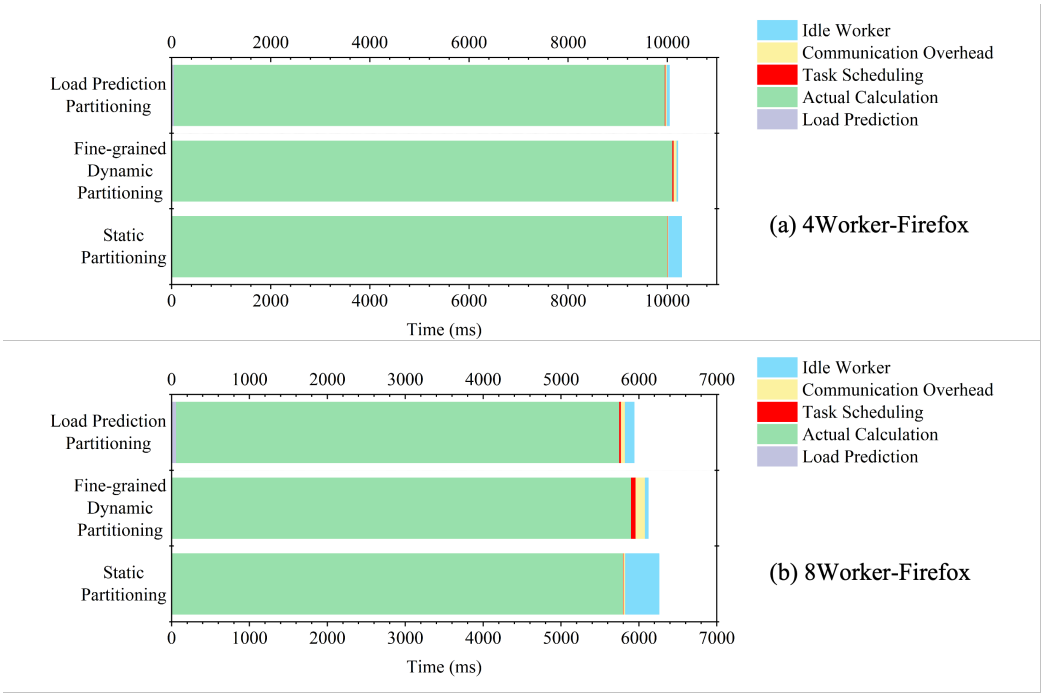


Figure A2. Execution time composition under different partitioning strategies for Julia set in Firefox.

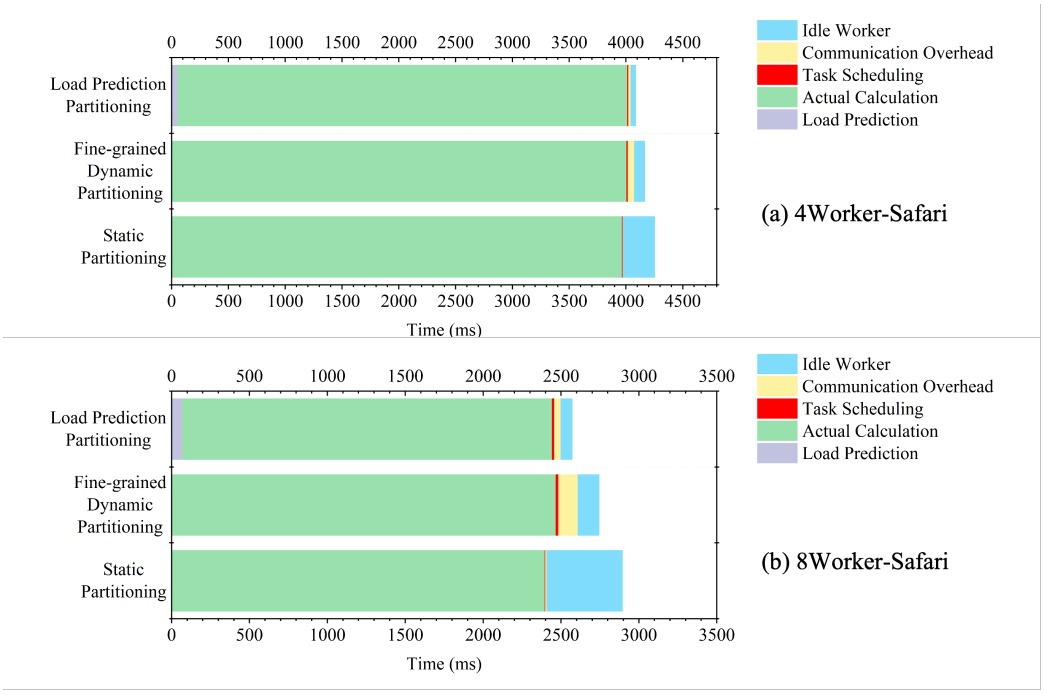


Figure A3. Execution time composition under different partitioning strategies for Mandelbrot set in Safari.

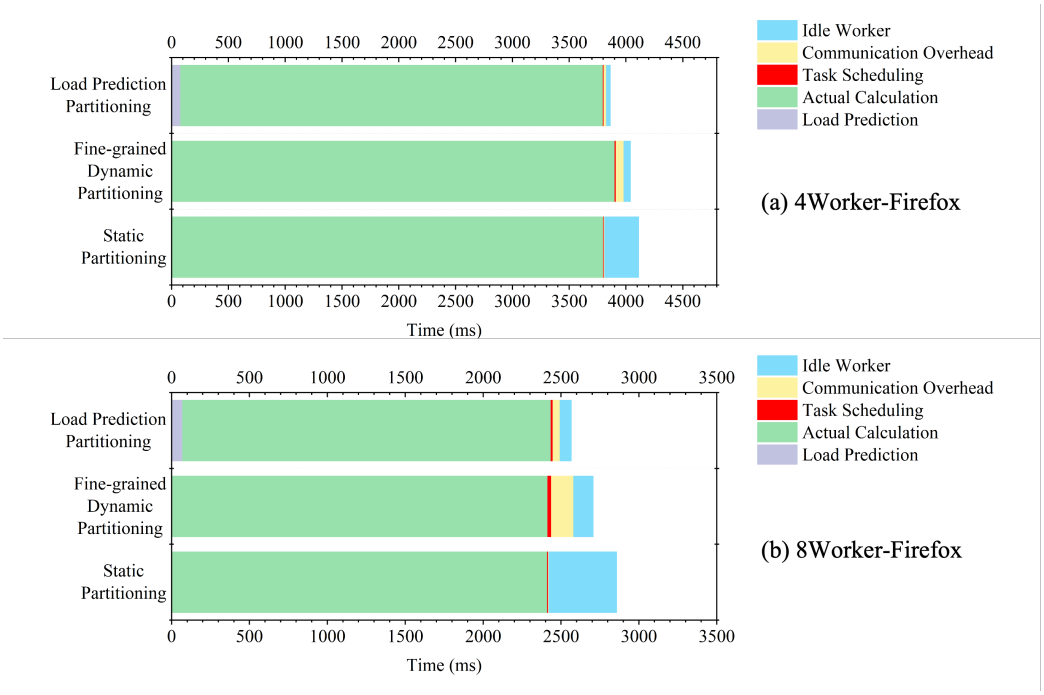


Figure A4. Execution time composition under different partitioning strategies for Mandelbrot set in Firefox.

References

1. Patiny, L. Unlocking the Potential of Browser-Based Scientific Data Analysis: A 20-Year Journey of Expertise. *Chimia* **2025**, *79*, 66–69. [CrossRef]
2. Yu, G.; Liu, C.; Fang, T.; Jia, J.; Lin, E.; He, Y.; Fu, S.; Wang, L.; Wei, L.; Huang, Q. A Survey of Real-Time Rendering on Web3D Application. *Virtual Real. Intell. Hardw.* **2023**, *5*, 379–394. [CrossRef]
3. Boutsi, A.M.; Ioannidis, C.; Verykokou, S. Multi-Resolution 3D Rendering for High-Performance Web AR. *Sensors* **2023**, *23*, 6885. [CrossRef]
4. Xie, C.; Wang, Z.; Chen, H.; Ma, X.; Xing, W.; Zhao, L.; Lin, Z. Real-Time High Realistic Web Display Method of Complex 3D Model. *IEEE Access* **2021**, *9*, 62562–62573. [CrossRef]
5. Franke, L.; Haehn, D. Modern Scientific Visualizations on the Web. *Informatics* **2020**, *7*, 37. [CrossRef]
6. Tomar, A.; Kumar, V.; Rana, U.S.; Sajid, M. Fractals as Julia and Mandelbrot Sets of Complex Cosine Functions via Fixed Point Iterations. *Symmetry* **2023**, *15*, 478. [CrossRef]
7. Lee, M.S. Application of Fractal Geometry to Architectural Design. *Archit. Res.* **2014**, *16*, 175–183. [CrossRef]
8. Boulmier, A.; Abdennadher, N.; Chopard, B. Optimal Load Balancing and Assessment of Existing Load Balancing Criteria. *J. Parallel Distrib. Comput.* **2022**, *169*, 211–225. [CrossRef]
9. Li, Y.; Ji, R.; Su, Y.; Zhang, Y.; Zhang, A.; Li, L. Dynamic Load Balancing for Distributed Large Model Training: A Hybrid Framework of Gray Markov Chain and MDP. *Concurr. Comput. Pract. Exp.* **2026**, *38*, e70456. [CrossRef]
10. Souravlas, S.; Anastasiadou, S.D.; Tantalaki, N.; Katsavounis, S. A Fair, Dynamic Load Balanced Task Distribution Strategy for Heterogeneous Cloud Platforms Based on Markov Process Modeling. *IEEE Access* **2022**, *10*, 26149–26162. [CrossRef]
11. Nangi, P.R.; Obannagari, C.K.R.N. High-Performance Distributed Database Partitioning Using Machine Learning-Driven Workload Forecasting and Query Optimization. *Am. Int. J. Comput. Sci. Technol.* **2024**, *6*, 11–21. [CrossRef]
12. Parizad, A.; Hatziadoniu, C. Deep Learning Algorithms and Parallel Distributed Computing Techniques for High-Resolution Load Forecasting Applying Hyperparameter Optimization. *IEEE Syst. J.* **2021**, *16*, 3758–3769. [CrossRef]
13. Han, J.; Zeng, P. Short-Term Power Load Forecasting Based on Hybrid Feature Extraction and Parallel BiLSTM Network. *Comput. Electr. Eng.* **2024**, *119*, 109631. [CrossRef]
14. Srinivasan, S.G.; Ashok, I.; Jônsson, H.; Kalonji, G.; Zahorjan, J. Dynamic-Domain-Decomposition Parallel Molecular Dynamics. *Comput. Phys. Commun.* **1997**, *102*, 44–58. [CrossRef]
15. Kalyur, S.; Nagaraja, G.S. ParaCite: Auto-Parallelization of a Sequential Program Using the Program Dependence Graph. In *Proceedings of the 2016 International Conference on Computation System and Information Technology for Sustainable Solutions (CSITSS), Bangalore, India, 6–7 October 2016*; IEEE: Piscataway, NJ, USA, 2016; pp. 7–12.

16. Li, Z.; Guan, X.; Wu, H.; Gong, J. A Novel K-Means Clustering Based Task Decomposition Method for Distributed Vector-Based CA Models. *ISPRS Int. J. Geo-Inf.* **2017**, *6*, 93. [CrossRef]
17. Li, Y.; Ge, X.; Lei, B.; Zhang, X.; Wang, W. Joint Task Partitioning and Parallel Scheduling in Device-Assisted Mobile Edge Networks. *IEEE Internet Things J.* **2023**, *11*, 14058–14075. [CrossRef]
18. Pan, D.; Yang, Y. FIFO-Based Multicast Scheduling Algorithm for Virtual Output Queued Packet Switches. *IEEE Trans. Comput.* **2005**, *54*, 1283–1297. [CrossRef]
19. Agrawal, K.; Leiserson, C.E.; He, Y.; Hsu, W.J. Adaptive Work-Stealing with Parallelism Feedback. *ACM Trans. Comput. Syst.* **2008**, *26*, 1–32. [CrossRef]
20. Tang, Z.; Xiao, Z.; Yang, L.; He, K.; Li, K. A Network Load Perception Based Task Scheduler for Parallel Distributed Data Processing Systems. *IEEE Trans. Cloud Comput.* **2021**, *11*, 1352–1364. [CrossRef]
21. Wang, B.; Song, Y.; Cao, J.; Cui, X.; Zhang, L. Improving Task Scheduling with Parallelism Awareness in Heterogeneous Computational Environments. *Future Gener. Comput. Syst.* **2019**, *94*, 419–429. [CrossRef]
22. Mo, L. PEPWA.js: A Performance Evaluation Tool for Parallel Web Applications. In *Proceedings of The 4th International Conference on Computer Applications (CCAT), Chengdu, China, 16 November 2025*; IEEE: Piscataway, NJ, USA, 2025.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.