Optimizing Path Tracing Workflows with MPI and Cloud Computing

Andrea Frigatti, Davide Ferrari

September 3, 2024

Abstract

This project focuses on parallelizing a path tracing workflow using the c-ray library and MPI. Two parallelization methods were implemented: tiling and sampling. The development process involved local testing with Docker containers for rapid iterative development and deployment on Google Cloud Platform Compute Engine using Terraform for automation. The study explores the performance of various configurations in distributed rendering, providing insights into the scalability of path tracing across multiple nodes. The project demonstrates the practical application of parallel computing techniques to enhance the efficiency of computationally intensive graphics rendering tasks.

1 Workload

Our project focused on parallelizing a path tracing workflow. Path tracing is an advanced rendering technique used in computer graphics to create highly realistic images by simulating the physical behavior of light. It works by tracing paths of light from the camera through pixels in an image plane and into the scene, accounting for reflections, refractions, and other light interactions with objects and materials.

The computational intensity of path tracing stems from several factors. It requires a high sample count to reduce noise and produce high-quality images, involves recursive calculations for each light bounce, and necessitates complex scene interactions with various materials and geometries. This intensity typically demands the use of GPUs for efficient processing, though our project explores CPU-based parallelization using distributed computing techniques.

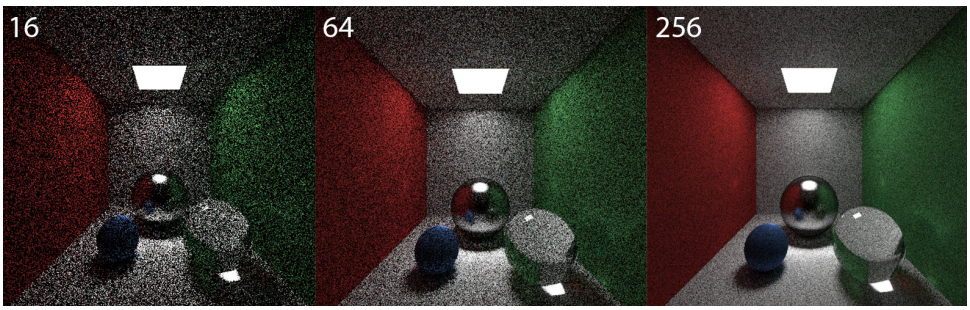


Figure 1: Effect on image clarify on increasing amount of samples in a path tracing render

Path tracing finds widespread use in industries requiring photo-realistic rendering, including film and animation, architectural and product design, video game development, and scientific visualization. Its ability to produce highly accurate and visually compelling images makes it a valuable tool across these diverse fields.

For our project, we employed c-ray, an open-source C library for path tracing. The workload involves rendering scenes defined in JSON files, which specify objects, materials, lights, and camera positions. The computational load is primarily influenced by two key rendering parameters: the number of samples (light paths traced per pixel) and the maximum number of bounces allowed for each light ray. By parallelizing this intensive workload, we

aim to reduce rendering times and enable the processing of more complex scenes or higher sample counts within practical timeframes, pushing the boundaries of what's achievable in realistic image synthesis.

2 Development process

2.1 Library used

We chose the open-source C library c-ray for our path tracing implementation. To ensure consistency across our collaborative development efforts, we fixed our work to a specific commit (84109d4) of the c-ray repository effectively by forking the repository. This decision was crucial as the library is under active development, and it allowed us to make changes to the library source code and to focus on our parallelization efforts without being affected by ongoing changes to the codebase.

2.2 Development environment setup

We established a comprehensive development environment to support our project

a. We created a Makefile that enabled us to compile our program in conjunction with the c-ray library ensuring all dependencies were properly linked and allowing for easy testing and iteration during development.

b. We configured the GNU Debugger (GDB) to facilitate code inspection and troubleshooting, which proved invaluable for understanding the library's behavior and identifying issues in our implementation.

c. We conducted an in-depth analysis of the core structures of the c-ray library, including the renderer, sampler, and output data structures.

2.3 Sequential implementation

We created a C program that utilized the c-ray library to render scenes defined in JSON files, establishing a baseline for our parallelization efforts
This file serves as the entry point for our program, performing the following:

a. Initializes the renderer

b. Loads the scene description and schematic from a JSON file

c. Configures rendering parameters, including image resolution, bounces and sample count

d. Starts the rendering process

e. After rendering, writes the image to file

2.4 Amdahl's law a priori study

To theoretically assess the potential speedup of our parallelized ray tracing implementation, we conducted an a priori analysis using Amdahl's Law. This analysis helps us estimate the maximum performance improvement we can expect when parallelizing our code on a distributed system.

2.4.1 Theoretical background

Amdahl's Law provides a framework to evaluate the theoretical speedup achievable by parallelizing a portion of a computation. It is defined as:

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}}$$

where:

- $S(N)$ is the theoretical speedup using N parallel processors.

- P is the fraction of the total execution time that can be parallelized.
- $1 - P$ is the fraction of the execution time that is inherently sequential.
- N is the number of parallel processors.

2.4.2 Application to our project

In our ray tracing application, the code is divided into three main components:

1. **Render Element Setup:** This involves preparing the scene, initializing variables, and other preliminary tasks. This portion is sequential and cannot be parallelized.
2. **Parallel Render (Tiling and Sampling):** This is the parallelizable part of the code where the image is rendered by dividing it into tiles and performing sampling in parallel across multiple processors.
3. **Image Saving:** Once rendering is complete, the image is saved to disk. This operation is sequential and cannot be parallelized.

Through analysis, we determined that approximately $P = 0.9$ (or 90%) of the total computation time is spent in the parallel render phase (tiling and sampling). The remaining 20% of the code, comprising the render element setup and image saving, is sequential.

2.4.3 Theoretical speedup calculation

Given that $P = 0.9$ and $N = 4$ processors will be used, we can apply Amdahl's Law to calculate the theoretical speedup:

$$S(4) = \frac{1}{(1 - 0.9) + \frac{0.9}{4}} = \frac{1}{0.1 + 0.225} = \frac{1}{0.325} \approx 3.08$$

This result suggests that, under ideal conditions, our parallelized ray tracing implementation can achieve a speedup of 3.1x when using 4 processors. This means that the parallelized code would execute 3.1 times faster compared to a sequential execution.

2.4.4 Discussion

The result highlights the impact of the sequential portion of the code on the overall performance. Even with perfect parallelization of the render phase, the sequential tasks (render element setup and image saving) impose a limit on the maximum achievable speedup. As per Amdahl's Law, increasing the number of processors beyond 4 would yield diminishing returns, given that the sequential part of the code remains a bottleneck.

In practical terms, this theoretical analysis sets expectations for the potential performance gains from parallelization and helps us understand the importance of optimizing not only the parallelizable sections but also minimizing the time spent in sequential operations to maximize overall efficiency.

2.5 Parallelization using MPI

The MPI parallelization strategy is designed to distribute the rendering workload across multiple processes and nodes, leveraging the power of distributed computing to accelerate the rendering process. We explored and implemented two parallelization strategies: tiling mode and sampling mode. These approaches allowed us to distribute the rendering workload across multiple processes, significantly reducing the overall computation time.

In both parallelization strategies, each process maintains its own instance of the renderer. This approach ensures that processes can work independently, minimizing the need for constant communication and synchronization. Each process loads the scene description from a JSON file, setting up identical initial conditions across all processes.

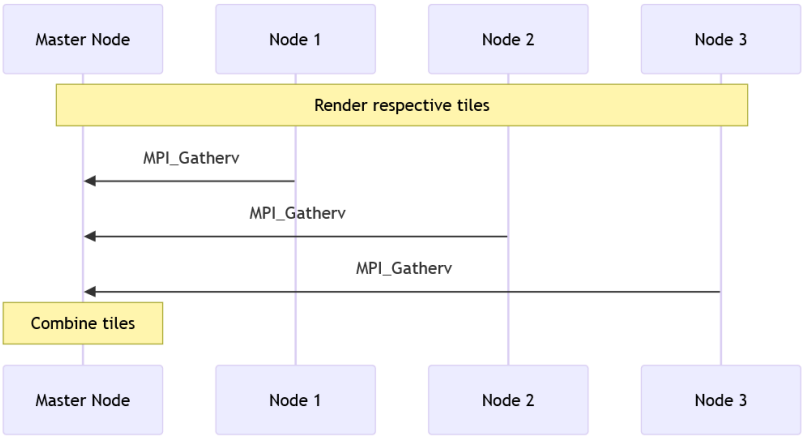
The program begins by initializing the MPI environment, which establishes the communication framework and assigns a unique rank to each process. This rank serves as an identifier, allowing processes to make decisions about their specific tasks within the overall rendering job.

2.5.1 Tiling mode

In tiling mode, the program divides the image into horizontal strips, or tiles, with each process responsible for rendering a complete section of the image. We calculate the height of each tile based on the total image height and the number of available processes. For example, if we have an image that is 1000 pixels tall and 4 processes available, each process would typically be assigned a tile of 250 pixels in height. We implemented logic to handle cases where the image height doesn't divide evenly among the processes, usually assigning any remaining rows to the last process's tile.

This strategy ensures that the workload is evenly distributed if the scene complexity is relatively uniform across the image. However, if certain areas of the scene are more complex and require more computation time, this could lead to some processes finishing their work earlier than others.

After rendering, we use MPI's variable-length gather operation (`MPI_Gatherv`) to collect the tiles from all processes. This operation allows us to efficiently assemble the complete image on the master process, taking into account the potentially different sizes of tiles from each process.



2.5.2 Sampling mode

In sampling mode, each process renders the entire image, but with a reduced number of samples per pixel. The total desired sample count is divided among the available processes. For instance, if we want 1000 samples per pixel and have 10 processes, each process would compute 100 samples for every pixel in the image.

In the initial phase of development of this mode, we ran into an issue that led to an important insight about the nature of parallel path tracing.

Initially, when we distributed the rendering across multiple processes, we observed that the final rendered image was unexpectedly noisy. The image quality was essentially equivalent to what we would have achieved with just a fraction of the total samples – specifically, the number of samples assigned to a single process. This outcome was contrary to our expectation of achieving a clearer, less noisy image by combining the work of multiple processes.

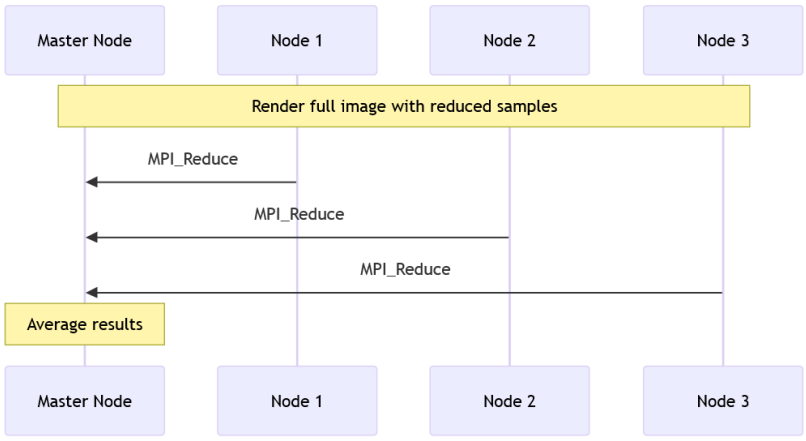
Upon closer investigation, we realized that this issue stemmed from a fundamental oversight in our initial implementation. Each process was using the same random number generator seed, which meant that every process

was effectively rendering exactly the same image, just with fewer samples.

To ensure variety in the sampling and prevent all processes from computing identical samples, we incorporate a unique random seed for each process. This seed is derived from the process rank and the current time, ensuring that each process explores different paths through the scene.

After rendering, we use MPI's reduction operation (`MPI_Reduce`) to combine the results from all processes. This efficiently sums up the sample data for each pixel. The master process then performs a final averaging step, dividing each pixel's accumulated value by the number of processes to produce the final image.

Sampling mode has the advantage of more evenly distributing the workload, especially in scenes where complexity varies significantly across the image. It also allows for easy scaling of sample counts by simply adding more processes. However, it requires more memory per process as each one needs to store the entire image.



2.6 Local cluster environment

To facilitate rapid iterative development and testing of our MPI parallelization methods, we implemented a local Docker-based cluster environment. This approach allowed us to simulate a distributed system on a single machine, significantly speeding up our development and testing cycles.

2.6.1 Docker image

Our Docker setup began with the creation of a Dockerfile to define our base MPI node image. We chose Ubuntu as our base image. The Dockerfile included commands to update the system packages, install essential tools like SSH, git, and basic networking utilities, as well as OpenMPI and its development libraries to support our MPI implementation.

2.6.2 Local cluster using Docker Compose

To manage our multi-container MPI cluster, we utilized Docker Compose. Our compose file defined four MPI nodes to simulate a small cluster. We created a custom bridge network named `'mpi_network'` to allow inter-container communication, mimicking a real cluster environment. Each node's SSH port was mapped to a unique host port, enabling external access for debugging or manual intervention when necessary.

2.6.3 Shared volumes

Using a shared volume we facilitated easy code and data sharing between the host and containers. We would develop and modify our MPI code on the host machine, and it would automatically be available in all containers via the shared volume. We could then run MPI jobs across the containers using `'mpirun'`, specifying the hostnames of our Docker containers. When needed, we could SSH into individual containers for targeted debugging.

2.6.4 Limitation for performance measurement

While our Docker-based setup proved invaluable for rapid development and functional testing, it's crucial to understand that this environment is not suitable for accurate performance measurements or benchmarking of our MPI implementation. The primary reasons for this limitation stem from the fundamental differences between a containerized local environment and a true distributed system. All Docker containers run on the same host machine, sharing resources and introducing contention that wouldn't exist in a real cluster. Additionally, the virtualization layer and simulated network don't accurately represent the characteristics of a high-performance computing environment.

Given these constraints, our Docker setup should be viewed primarily as a development and testing tool, not as a platform for performance evaluation.

2.7 Cloud migration

With the knowledge and experience gained from our local Docker setup, we transitioned our implementation to the Google Cloud Platform (GCP) using Compute Engine. This migration allowed us to test our parallelization strategies on actual distributed systems, providing a more realistic environment for performance evaluation.

We explored various cluster configurations to understand how our parallelized path tracing solution performed under different architectures.
We specifically tested three distinct cluster designs:

a. **Fat Cluster**: We deployed clusters with a small number of powerful Virtual Machines (VMs), each hosting a large number of vCPUs and substantial memory.

b. **Light Cluster**: In contrast to the fat cluster, we also tested configurations with many lightweight VMs, each with a small number of vCPUs.

c. **Intra-Regional vs. Inter-Regional Clusters**: To understand the impact of network latency and data transfer speeds on our rendering performance, we deployed two types of geographical distributions.

By testing these diverse cluster configurations, we gained valuable insights into:

1. The scalability of our solution across different node sizes and quantities
2. The impact of CPU-to-memory ratios on rendering performance
3. The effects of network latency and bandwidth on our distributed rendering tasks
4. The optimal cluster design for different types of rendering jobs (e.g., quick previews vs. high-quality final renders)

2.8 Automated deployment using Terraform

To streamline our testing process and easily iterate through various configurations, we developed Terraform scripts. These scripts automated the deployment of our cluster on GCP, significantly reducing the time and effort required to set up different testing environments, and provided us with an end-to-end solution for deploying and testing configurations.
This approach offered several significant advantages:

a. **Reproducibility**: By codifying our infrastructure setup, we ensured that each deployment was consistent and reproducible. This eliminated discrepancies that could arise from manual setup processes.

b. **Flexibility**: The use of variables in our Terraform script allowed us to easily modify key aspects of our deployment, such as the number of VMs, machine types, and computational parameters.

c. **Time Efficiency**: Automating the deployment process significantly reduced the time and effort required to set up and tear down different cluster configurations. Eliminating costly dead times.

Our Terraform script performs several key functions:

a. **VM Provisioning**: It creates a specified number of Google Compute Engine instances (nodes) based on the `vm-count` variable. Each instance is configured with the specified machine type, zone, and network settings.

b. **SSH Key Management**: The script assigns predefined SSH keys to the VMs for secure access and configures correct access from master to slave nodes by automatically exchanging identification keys.

c. **Initialization**: The script uses Terraform provisioners blocks to upload and execute custom initialization scripts. These scripts set up the necessary environment for our path tracing application from scratch.

d. **MPI Cluster Setup**: On the master node, it sets up the MPI hostfile, configuring the number of slots per node based on the `vm-slots` variable.

e. **Automatic Execution**: After setup, the script automatically executes our path tracing application using MPI, with the number of processes specified by the `vm-np` variable.

f. **Result Retrieval**: Finally, it downloads the rendered output from the master node to the local machine, allowing for immediate inspection of results.

This comprehensive automation allowed us to rapidly deploy, test, and analyze different cluster configurations by simply adjusting variables such as `vm-count`, `vm-type`, `vm-slots`, `vm-np`. For instance, we could easily switch between testing a few powerful machines versus many smaller instances, or adjust the number of MPI processes per node, all without manually reconfiguring the infrastructure.

The ability to quickly deploy, run tests, and tear down environments not only accelerated our development cycle but also enabled us to conduct a more thorough exploration of performance characteristics under various conditions. This approach was instrumental in optimizing our parallelized path tracing implementation and understanding its scalability characteristics across different hardware configurations.

2.9 Cluster Configurations on GCP

To evaluate the performance of our parallelized path tracing solution on distributed systems, we deployed and tested various cluster configurations on Google Cloud Platform (GCP). These configurations were designed to assess scalability, CPU-to-memory ratios, and the impact of network latency and bandwidth on rendering tasks. The following subsections describe the specific cluster designs we used for our experiments.

2.9.1 Fat cluster

The fat cluster configurations consisted of a small number of powerful Virtual Machines (VMs) with a large number of virtual CPUs (vCPUs) and substantial memory. We tested two fat cluster setups:

- **Fat Cluster 1:** This configuration utilized two `c2-standard-4` VMs, each equipped with 4 vCPUs and 16 GB of memory, based on Intel Cascade Lake processors.
- **Fat Cluster 2:** In this setup, we deployed two `c4-highcpu-4` VMs, each providing 4 vCPUs and 8 GB of memory, powered by Intel Emerald Rapids processors.

Both fat clusters were deployed as intra-regional clusters to minimize network latency and ensure consistent performance.

2.9.2 Light cluster

In contrast to the fat clusters, the light cluster configurations involved a larger number of lightweight VMs with fewer vCPUs. The following configurations were tested:

- **Light Cluster 1:** This configuration employed four `e2-highcpu-4` VMs, each with 4 vCPUs and 4 GB of memory. The processors used were based on availability within the GCP region.
- **Light Cluster 2:** This setup included two `e2-highcpu-8` VMs, each providing 8 vCPUs and 8 GB of memory, also using processors based on availability.

Similar to the fat clusters, these light clusters were deployed as intra-regional clusters.

2.9.3 Intra-Regional vs. Inter-Regional clusters

To assess the impact of network latency and data transfer speeds on rendering performance, we compared intra-regional and inter-regional cluster configurations.

- **Intra-Regional Cluster:** For this test, we used the cluster configurations explained in the subsections **Fat Cluster** and **Light Cluster** with all VMs located within the same region (europe-west8-a), ensuring minimal latency.
- **Inter-Regional Cluster:** This configuration involved two `c2-standard-4` VMs, with one VM located in `europe-west4-a` and the other in `us-central1-a`, introducing higher latency due to the geographical distance.

This comparison provided insights into the effects of network latency and bandwidth on distributed rendering tasks, which are critical for optimizing performance in real-world scenarios.

These diverse cluster configurations enabled us to gain valuable insights into the scalability, CPU-to-memory ratio effects, and the impact of network latency on our distributed path tracing solution, allowing us to identify the optimal cluster designs for various types of rendering jobs.

3 Results

Our experiment aimed to evaluate the performance of a parallelized ray tracing library on Google Cloud Platform (GCP) using different cluster configurations. The test was conducted on a rendered image of 250x250 pixels with 250 samples per pixel and 30 bounces. The performance metrics collected include the time taken for tiling and sampling, the total core time, and the speedup compared to a sequential execution.

Node type	Tiling (s)	Sampling (s)	Speedup tiling	Speedup sampling
e2-highcpu-4	17,921311	14,835419	3,90x	4,73x
e2-highcpu-8	17,867890	14,862826	3,93x	4,72x
c4-highcpu-4	14,147418	8,455381	2,33x	3,90x
c2-standard-4	22,916034	12,791110	2,19x	3,93x

Table 1: Average time for parallel modes and relative speedup

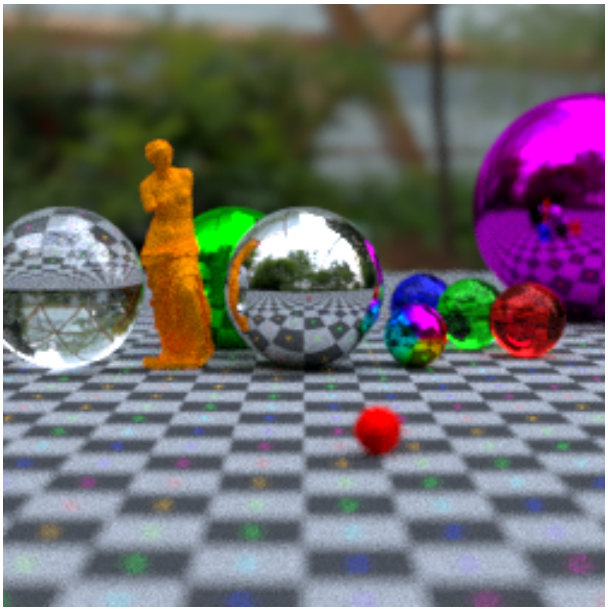


Figure 2: Rendered image from our program, tiling mode, 250px x 250px resolution, 250 samples, 30 bounces

3.1 Fat cluster results

For the fat cluster configurations:

- **c4-highcpu-4 (Intel Emerald Rapis):** The average tiling time was approximately 14.14 seconds, and the average sampling time was around 8.45 seconds. The speedup achieved for tiling was 3.91x, while for sampling it was 4.73x.
- **c2-standard-4 (Intel Cascade Lake):** This configuration showed an average tiling time of about 22.92 seconds and an average sampling time of 12.79 seconds. The speedup ratios were lower compared to c4-highcpu-4, with 2.00x for tiling and 1.65x for sampling.

3.2 Light cluster results

For the light cluster configurations:

- **e2-highcpu-4:** The average tiling time recorded was 17.92 seconds, while the average sampling time was 14.83 seconds. The speedup for tiling was 2.00x, and for sampling, it was 1.65x.
- **e2-highcpu-8:** With more vCPUs, the average tiling time improved to 17.84 seconds, and the sampling time to 14.87 seconds. The speedup values were comparable to those observed in the e2-highcpu-4 configuration.

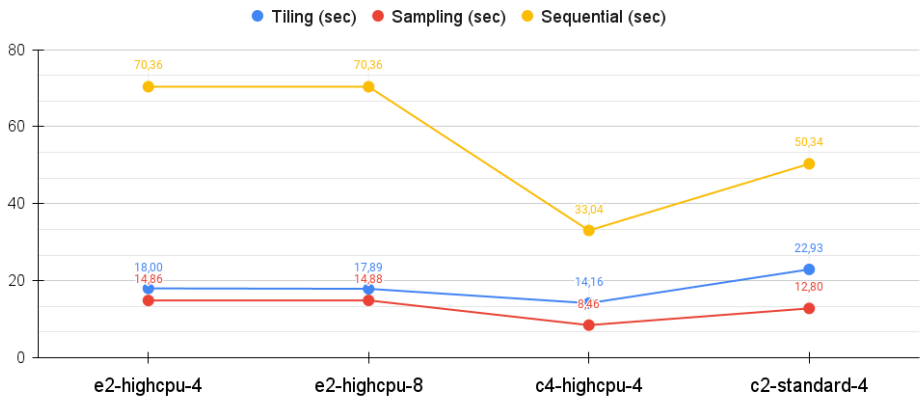


Figure 3: Cluster results for sequential, tiling and sampling

3.3 Inter-Regional cluster results

For the inter-regional cluster (c2-standard-4):

- The average tiling time was 22.92 seconds, and the sampling time was 12.79 seconds. The inter-regional deployment exhibited a higher latency impact, reflected in the longer communication delay between VMs compared to intra-regional configurations.

3.4 Comparison with Amdahl's law prediction

Our a priori study using Amdahl's Law predicted a theoretical speedup of approximately 3.08x for a system with 4 processors, assuming 90% of the code could be parallelized. Comparing this to our experimental results, we observe that the actual speedups varied depending on the cluster configuration and parallelization method. The c4-highcpu-4 configuration achieved the closest match to the theoretical prediction, with a speedup of 3.90x for tiling and 4.73x for sampling. These results slightly exceed the theoretical prediction, which could be attributed to factors not accounted for in the simplified Amdahl's Law model, such as cache effects or improved memory bandwidth utilization in the parallel implementation. Other configurations, like the e2-highcpu-4 and e2-highcpu-8, showed speedups (3.90x and 3.93x for tiling, 4.73x and 4.72x for sampling, respectively) that also aligned well with or exceeded the theoretical prediction. The c2-standard-4 configuration, however, fell short of the predicted speedup (2.19x for tiling and 3.93x for sampling), likely due to the impact of inter-regional latency. Overall, these comparisons demonstrate that our parallelization efforts were successful in achieving, and in some cases surpassing, the theoretical speedup predicted by Amdahl's Law, while also highlighting the impact of hardware and network configurations on real-world performance.

4 Cost analysis

In addition to performance metrics, we analyzed the cost-effectiveness of different VM configurations on Google Cloud Platform (GCP) for rendering tasks. The cost per 1000 rendered images was evaluated for each cluster configuration, taking into account both the tiling and sampling phases of the ray tracing process.

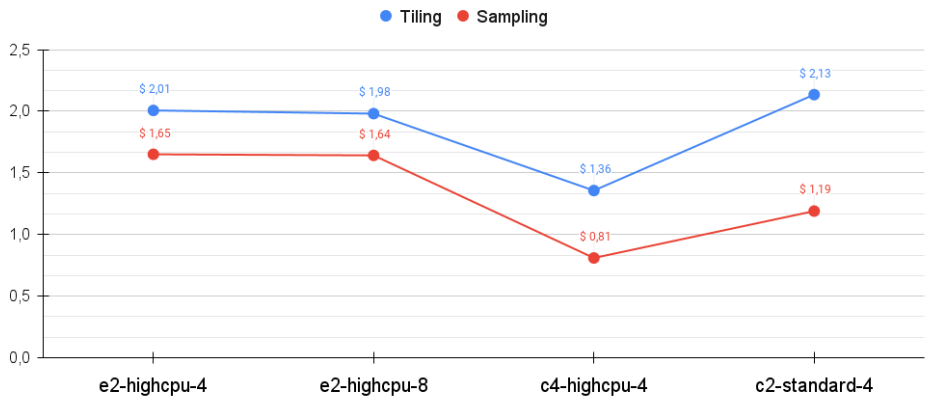


Figure 4: VM costs per 1000 rendered images

4.1 Cost results

The cost analysis revealed the following insights:

- **e2-highcpu-4:** This configuration had a cost of $2.01 for tiling and $1.65 for sampling per 1,000 rendered images. This made it one of the more expensive configurations, particularly for tiling tasks.
- **e2-highcpu-8:** With a cost of $1.98 for tiling and $1.64 for sampling, this configuration offered a slightly reduced cost compared to the e2-highcpu-4, while providing similar performance benefits.
- **c4-highcpu-4:** The cost of $1.36 for tiling and $0.81 for sampling highlighted this configuration as the most cost-effective option among the fat clusters. It provided significant cost savings, especially in the sampling phase.
- **c2-standard-4:** This configuration had the highest cost for tiling at $2.13, but a more moderate cost for sampling at $1.19. The higher tiling cost is likely due to the increased network latency associated with the inter-regional deployment.

4.2 Cost conclusions

The cost analysis provided valuable insights into the trade-offs between performance and expense:

1. **Cost-Performance Balance:** The c4-highcpu-4 configuration emerged as the most cost-efficient option, particularly for tasks where sampling is the primary concern. Its lower costs combined with solid performance make it an attractive choice for high-volume rendering tasks.
2. **High-Cost Configurations:** Both the e2-highcpu-4 and c2-standard-4 configurations were found to be more expensive, with the latter being particularly costly for tiling. These configurations should be reserved for specific use cases where their particular strengths (e.g., higher memory or regional distribution) are necessary.
3. **Optimizing for Task Type:** For tasks that are heavily reliant on sampling, the c4-highcpu-4 configuration offers the best cost savings. In contrast, the e2-highcpu-8 may be preferred when a balance between cost and consistent performance across tiling and sampling is required.
4. **Inter-Regional Considerations:** The elevated cost associated with the c2-standard-4 configuration underscores the impact of network latency on cost. Keeping computations within a single region is crucial for minimizing expenses in latency-sensitive workloads.

These findings will inform future decisions regarding the optimal balance between cost and performance for rendering tasks on GCP, ensuring that resources are allocated effectively based on the specific requirements of each project.

5 Conclusions

The results from our experiments on GCP reveal several key insights:

1. **Scalability and Node Configuration:** Fat clusters with fewer, more powerful VMs (c4-highcpu-4) demonstrated better speedup ratios in both tiling and sampling phases compared to light clusters. The balance between CPU power and memory significantly influences rendering performance.

2. **Impact of Network Latency:** The inter-regional cluster (c2-standard-4) experienced noticeable performance degradation due to increased network latency, which negatively affected the tiling and sampling times. This highlights the importance of deploying clusters within the same region for latency-sensitive tasks.

3. **Cost-Performance Trade-offs:** While the fat clusters provided better performance, the light clusters (e2-highcpu-4 and e2-highcpu-8) offer a cost-effective alternative, particularly for tasks where absolute speed is not critical.

4. **Optimal Cluster Design:** For high-quality final renders, fat clusters are preferred due to their superior speedup and efficiency. For quick previews or less intensive tasks, light clusters provide a balanced approach between cost and performance.

These findings will guide our future decisions in selecting the appropriate cluster configurations for different rendering tasks on GCP.