Optimizing Path Tracing Workflows with MPI and Cloud Computing

Advanced Computer Architecture
Parallel Programming Project

Davide Ferrari
Andrea Frigatti
COMPUTER SCIENCE AND MULTIMEDIA

UNIVERSITÀ
DI PAVIA

Project overview

Objective:
This project focuses on **parallelizing a path tracing workload** using the **c-ray library** and **MPI**.

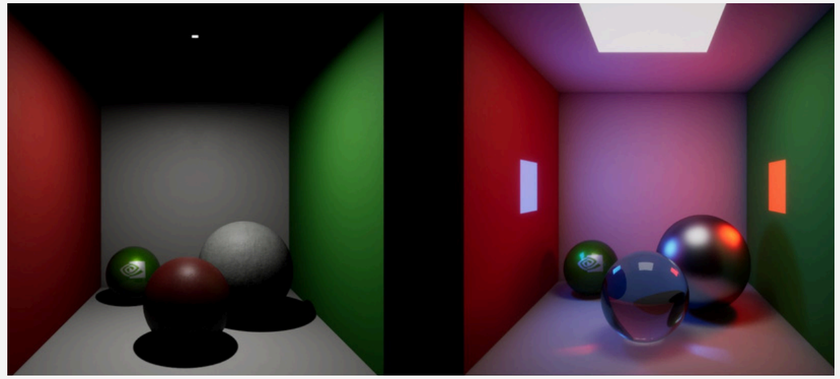
Implementation:
The development process involved **local testing** with **Docker containers** for rapid iterative development and **deployment** on **GCP Compute Engine** also using **Terraform** for automation. The study explores the performance of various configurations in distributed rendering, providing insights into the scalability of path tracing across multiple nodes.

c-ray library

c-ray is an **open-source C library** for path tracing that allows rendering scenes defined in JSON files, which specify objects, materials, lights, and camera positions.

Core components:

- Renderer
- Sampler
- Workers
- Proprietary data structures



Path tracing is a **rendering technique** used in computer graphics to create **highly realistic images** by simulating the **physical behavior of light** by tracing light paths 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.

Path tracing is **highly computationally intensive** and typically relies on GPUs for efficient processing. However, our project explores CPU-based parallelization using distributed computing techniques.

Developement process

- We **forked the c-ray repository** to fix the version of the library and make changes to its source code, as it’s still in active developement
- Setup the compilation and debugging process with **Makefile** and **GDB**
- Developed an initial **baseline** program that executes a **sequential render**
- Implemented **two parallelism techniques** (tiling and sampling) using **MPI**, this involved changes in c-ray source code.
- **Tested** the parallel configuration using a **Docker local cluster**
- **Deployed** the solution on **Google Cloud Platform Compute Engine**
- **Automated** the deployment process using **Terraform** for more efficient data collection and analysis

Amdahl's law a priori study

Theoretical framework

- S(N): Theoretical speedup with N processors
- P: Fraction of parallelizable execution time
- 1 - P: Fraction of sequential execution time

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

Application to our project:

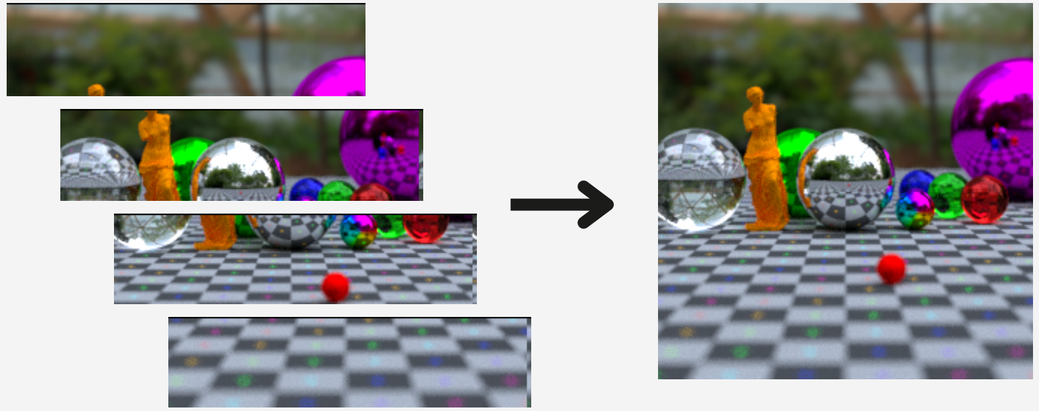
- Estimated P = 0.9 (90% parallelizable)
- Assuming N = 4 processors

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

Implications

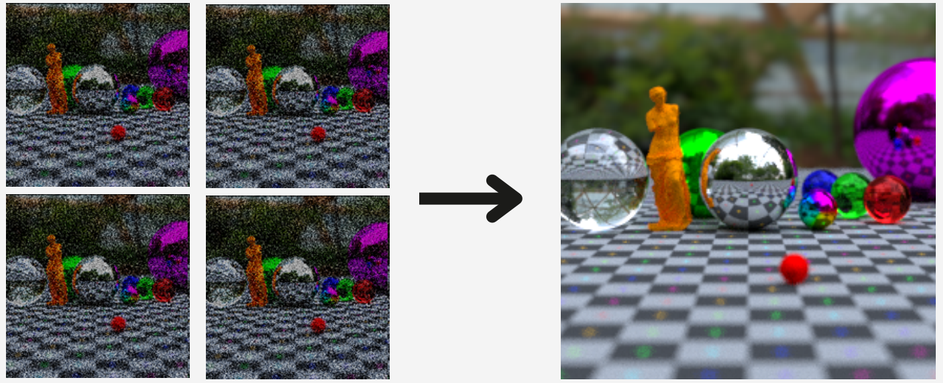
- Maximum expected speedup: 3.08x
- Highlights impact of sequential portions
- Sets realistic performance expectations

Parallelization techniques



Tiling mode

- Image is divided into **horizontal strips**, or tiles, with each process responsible for rendering a complete section of the image
- The **height of each tile** is based on the total image height and the **number of available processes**
- A logic to handle cases is implemented where the **image height doesn’t divide evenly** among the processes, usually assigning any remaining rows to the **last process’s tile**



Sampling mode

- Each process renders the **entire image**, but with a **reduced number of samples**
- The total desired **sample count** is **divided among the available processes**
- A **unique random seed** is incorporated 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**

Docker local cluster

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

- **Dockerfile**: includes 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
- **Docker Compose**: defines four MPI nodes to simulate a small cluster and creates a custom bridge network named 'mpi_network' to allow inter-container communication
- **Shared Volumes**: facilitates code and data sharing between host and containers allowing our MPI code being compiled on the host machine and automatically being available in all containers via the shared volume

```yaml
services:
  mpi_node_1:
    image: ubuntu-mpi
    container_name: mpi-node-1
    hostname: mpi-node-1
    networks:
      - mpi_network
    ports:
      - "8022:22"
    volumes:
      - ./shared_volume:/home/ubuntu/data
```

Cloud migration

Transition to real distributed systems

- Moved from local Docker setup to **GCP Compute Engine**
- Enabled testing on actual distributed systems for **realistic performance evaluation**

Cluster configurations explored

- **Fat Cluster:** Small number of powerful VMs with many vCPUs
- **Light Cluster:** Many lightweight VMs with fewer vCPUs
- **Inter Regional:** Tested impact of geographical distribution

Key insights gained

- Impact of node size, quantity and geolocation in scalability
- Effects of network latency and bandwidth on distributed rendering
- Optimal cluster design for various rendering job types

Tests conducted using c2-standard-4, c4-highcpu-4 (fat cluster); e2-highcpu-4, e2-highcpu-8 (light cluster); c2-standard-4 (inter regional)

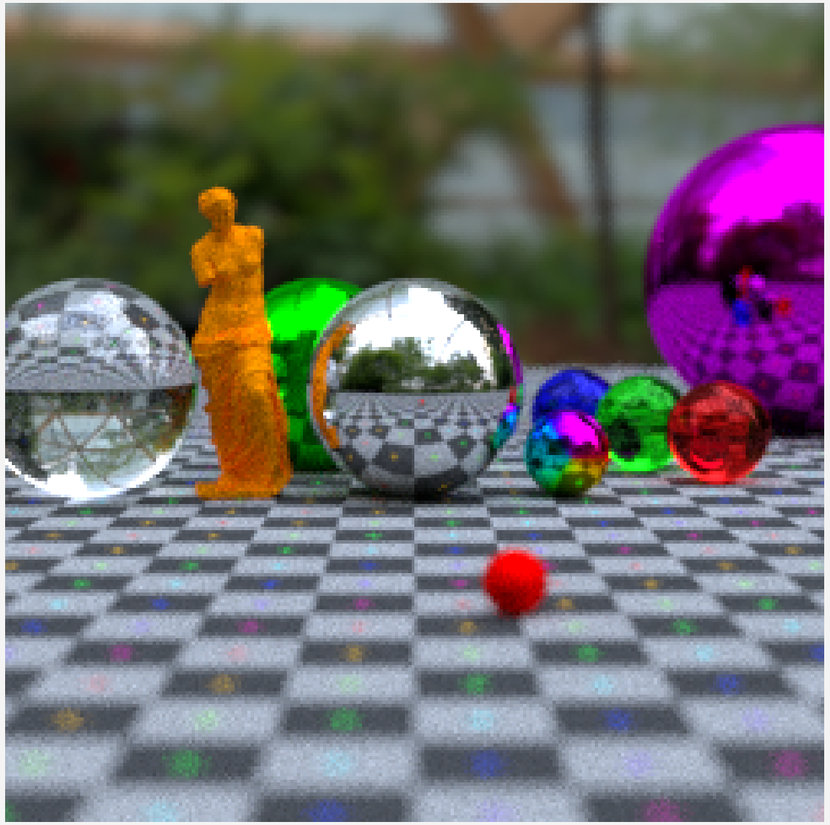
Terraform

Terraform is an open-source **Infrastructure as Code** (IaC) tool
It allows to define and manage cloud infrastructure using a **declarative language**, giving us **consistency** and **reproducibility** in deployment operations

Our Terraform script:

- **VM provisioning**: Creates a specified number of node instances
- **SSH key management**: Correctly configures master and slaves with SSH keys
- **Environment initialization**: Updates packages and compiles source codes
- **Automatic execution**: Runs rendering benchmarks automatically
- **Result retrieval**: Downloads benchmark data (results and rendered image)

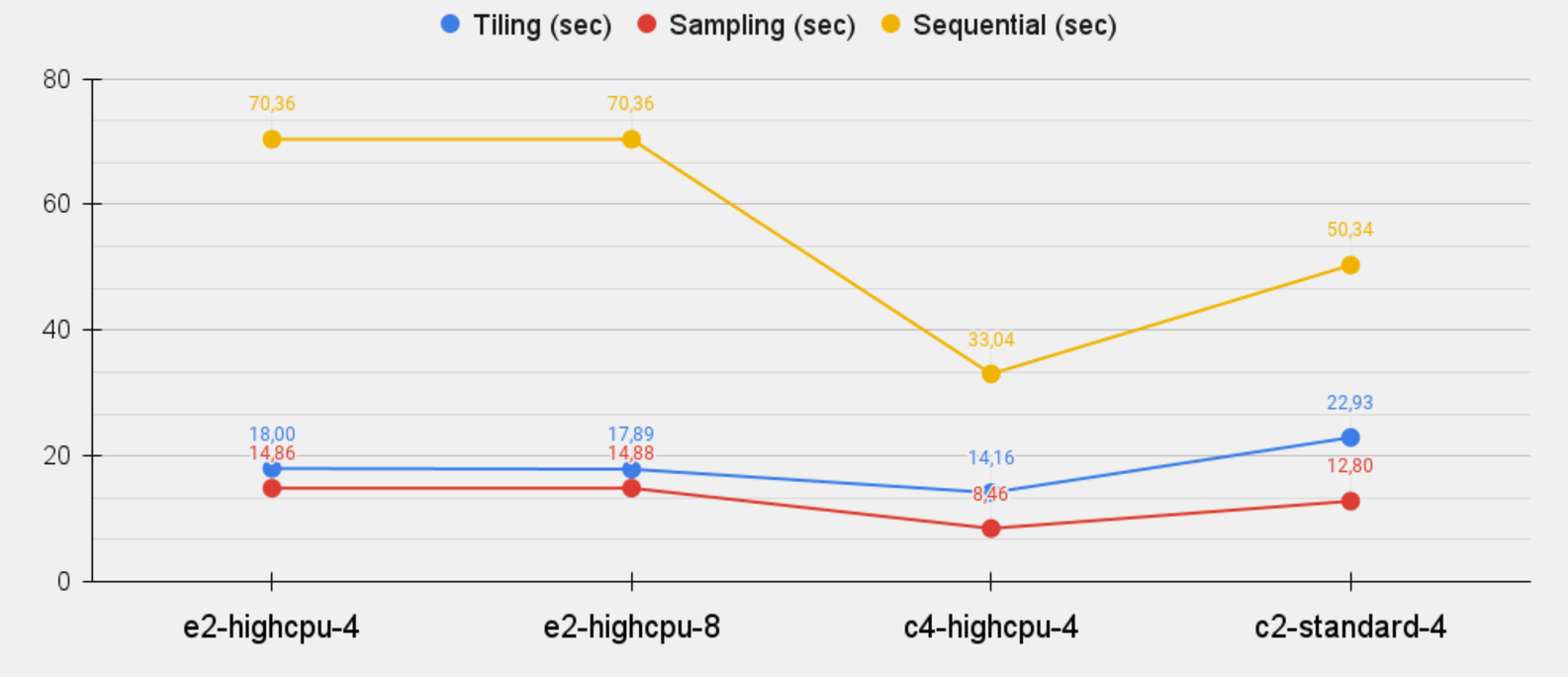
Results



250px x 250px resolution, 250 samples, 30 bounces

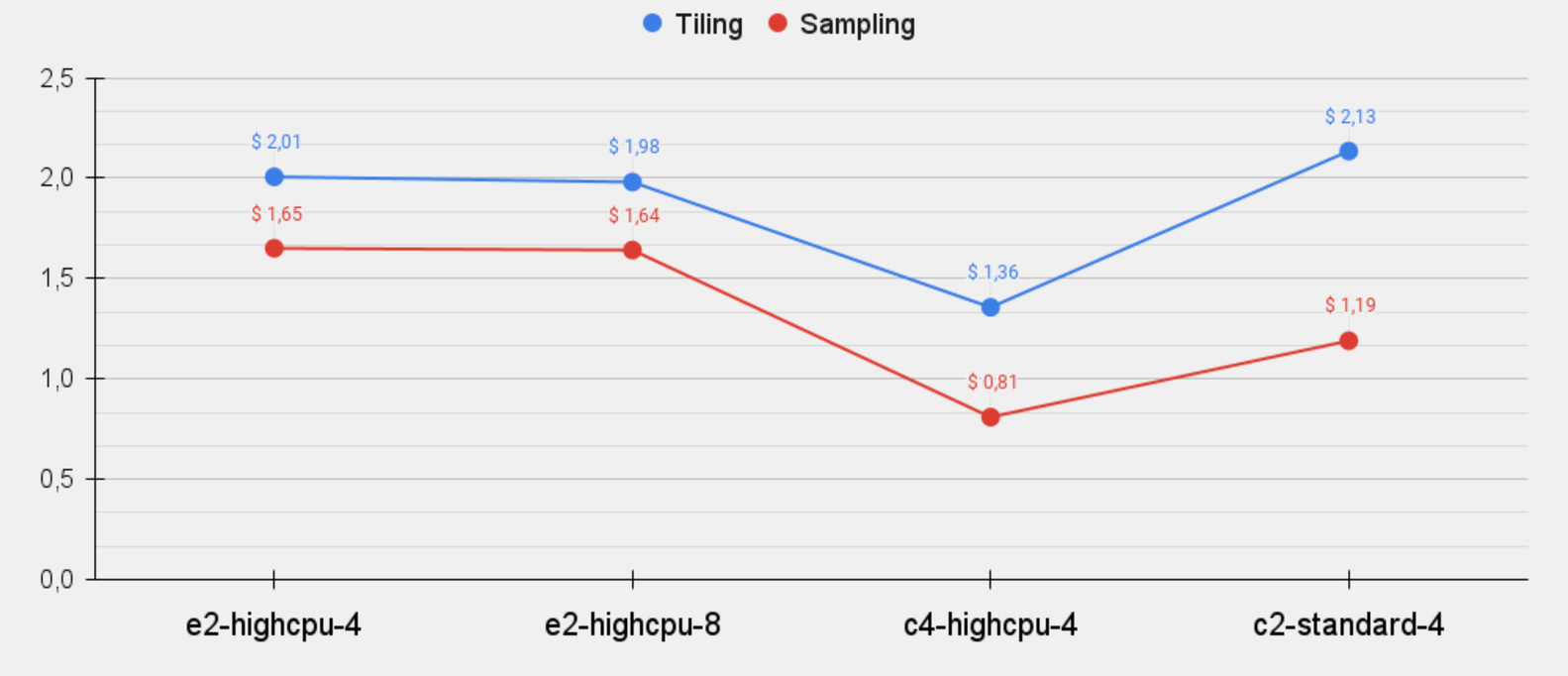
Results

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



Cost analysis

We conducted a comprehensive **price comparison** of different VM configurations on Google Cloud Platform (GCP) for our rendering tasks and identified c4-highcore-4, the highest performance configuration to also be the cheapest



Price comparison made per 1000 rendered images

Conclusions

1. **Scalability & Node Configuration**: Fat clusters (fewer, powerful VMs) outperform light clusters in both tiling and sampling

2. **Network Latency**: Inter-regional clusters suffer from performance degradation due to latency, highlighting the importance of intra-regional deployments for sensitive tasks

3. **Cost-Performance Trade-offs**: Fat clusters offer better speed and performance, but light clusters are more cost-effective for non-intensive tasks

4. **Optimal Cluster Design**: Fat clusters are ideal for high-quality final renders, while light clusters work well for previews and less intensive tasks

UNIVERSITÀ
DI PAVIA

Thanks for your attention!