Parallel Google Maps Tile Stitching with PARCS.NET

Pavel Shpagin

February 10, 2026

1 Introduction

This report presents a parallel implementation for downloading Google Maps satellite imagery tiles and stitching them into a high-resolution mosaic using **PARCS.NET** [1,2]. The task mirrors the Python PARCS solution used in `report.tex`, but replaces Pyro4-based workers with PARCS.NET *points* (remote executors) and C# modules.

The workload is naturally parallel: each map tile is fetched independently from the Google Maps Static API. The master node performs lightweight coordinate generation and a final (partially sequential) mosaic assembly step.

2 Algorithm Overview

Given a geographic center point (lat, lon) and requested region size $(width_m, height_m)$ in meters, the system builds a grid of tiles at a fixed spatial resolution of 100 meters per tile:

$$n_{cols} = \left\lfloor \frac{width_m}{100} \right\rfloor \tag{1}$$

$$n_{rows} = \left\lfloor \frac{height_m}{100} \right\rfloor \tag{2}$$

Tile center coordinates are computed using the Web Mercator projection. With zoom level z and world size in pixels $W = 256 \cdot 2^z$, forward mapping is:

$$x = \frac{lon + 180}{360} \cdot W \tag{3}$$

$$y = \left(0.5 - \frac{\ln\left(\frac{1+\sin(lat)}{1-\sin(lat)}\right)}{4\pi}\right) \cdot W \tag{4}$$

The reverse mapping converts pixel coordinates back to (lat, lon) to generate centers for each tile request.

Tiles are requested from Google Maps Static API at zoom $z = 19$ with parameters:

- `size = 640x640`

- `scale = 2` (higher pixel density, same geographic coverage)
- `maptype = satellite`
- `format = jpg`

Each tile is preprocessed by cropping 40 pixels from the bottom to remove the watermark area, then re-encoding as JPEG (quality 45–60) to keep memory and output sizes manageable.

3 PARCS.NET Parallelization Strategy

PARCS.NET provides a master–worker execution model:

- The **master module** derives from `Parcs.MainModule` and is executed locally.
- The master creates p **points** via `ModuleInfo.CreatePoint()` and starts a worker class on each point via `ExecuteClass()`.
- Master and worker communicate through **channels** (`IChannel`), exchanging serializable objects.

Work distribution uses **round-robin assignment**: point i receives tiles with indices $\{i, i+p, i+2p, \ldots\}$. This balances load under variable HTTP latency.

4 Experimental Setup

Two datasets are used:

- **Small**: 400m $\times$ 400m $\Rightarrow$ 16 tiles
- **Medium**: 1200m $\times$ 1200m $\Rightarrow$ 144 tiles

Execution time is measured from the start of dispatch to completion of mosaic generation. Speedup is defined as:

$$S_p = \frac{T_1}{T_p}$$

5 Results

5.1 Single-Region Experiments

Initial experiments with a single PARCS.NET cluster revealed that Google Maps API rate-limiting prevents linear speedup when all daemons share a single external IP (via Cloud NAT). However, when each daemon has an **independent external IP**, real parallel speedup is achieved.

5.2 Multi-Region Federated Architecture

To overcome per-IP rate limits, we deploy clusters across **multiple GCP regions**, each with independent external IPs. The workload is **split across regions** (federated), with each region downloading a subset of tiles in parallel:

```
# Deploy 3 clusters (us-east1, us-west1, europe-west1)
.\gcp\run_multi_region_experiments.ps1

# Run federated split experiment
.\gcp\run_federated_split.ps1 -InputFile tests\medium_district.txt
```

Table 1: Federated multi-region results (medium dataset, 144 tiles)

Configuration	Download Time	Speedup
1 region, 1 point (baseline)	65.0s	1.0×
1 region, 3 points (3 IPs)	28.0s	2.3×
3 regions, 9 points (9 IPs, federated)	**12.5s**	**5.2**×

The federated architecture splits 144 tiles across 3 regions (48 tiles each). Each region uses 3 daemons with independent external IPs, achieving true parallel downloads without API throttling.

Table 2: Per-region download times in federated mode

Region	Tiles	Download Time
europe-west1	48	11.4s
us-west1	48	11.7s
us-east1	48	12.5s

Wall-clock time is determined by the slowest region (12.5s), yielding the **5.2× speedup** over the single-point baseline.

6 Optimizations

Key implementation optimizations:

- **Multi-region federation**: splits tiles across GCP regions to leverage independent external IPs and avoid API throttling.
- **Independent external IPs**: each daemon VM has its own external IP, preventing shared-NAT rate limiting.
- **Round-robin distribution**: reduces stragglers when HTTP latency varies.

- **Connection reuse**: workers reuse HTTP connections (keep-alive) to reduce handshake overhead.
- **Minimal throttling**: workers use no per-request delay, allowing true parallel downloads.
- **Master-side preprocessing**: workers return raw JPEG bytes; the master performs crop and re-encode before stitching.

7 Containerization / Deployment Notes

HostServer and Daemons are deployed as **Linux containers** on Google Compute Engine using upstream PARCS.NET images:

- `andriikhavro/parcshostserver:dotnetcore-2.1`
- `andriikhavro/parcsdaemon:dotnetcore-2.1`

The cluster is **headless** (no web UI). Modules are launched via CLI from within the same VPC.

On GCP Container-Optimized OS, user home directories are mounted with `noexec`, so the C# runner is executed inside a small Docker image built on the host VM (see `gcp/parcsnet_runner.Dockerfile`).

8 Conclusion

The PARCS.NET implementation achieves **5.2× speedup** on the medium dataset (144 tiles) using a multi-region federated architecture with 9 parallel workers across 3 GCP regions. This demonstrates that the Google Maps tile download problem is highly parallelizable when API rate-limiting is mitigated through independent external IPs.

The key insight is that single-region deployments hit API throttling limits regardless of worker count, but distributing workers across regions with independent IPs enables true parallel scaling. The provided scripts automate cluster deployment and federated execution.

9 References

1. Parcs.NET repository. `https://github.com/AndriyKhavro/Parcs.NET`
2. Project repository. `https://github.com/PavelShpagin/gcp-project`
3. Google Maps Platform Documentation. `https://developers.google.com/maps/documentation`