Parallelizing a Mosaic Image Generator

David Dabreo and Robert Pianezza
Ontario Tech University – Massively Parallel Programming (CSCI 4060)

April 15, 2025

Abstract

This project looks at how to make a mosaic image generator run faster by using OpenMP to add parallel processing in C++. The original version worked, but it was slow when handling large images. We looked for parts of the program that could run concurrently on threads and used OpenMP to speed them up. By doing this, we reduced the time it takes to build the mosaic. Our results show that the program gets faster when more threads are used, proving that parallel programming is helpful for image-related tasks.

1 Introduction

Mosaic-style images are formed through reconstructing a source image using a collection of tile images based on color similarity. This can be a compute-intensive task, as the resolution of the source image and the number of tile images increase, the time required to create the mosaic image grows quickly.

This project aims to improve the performance of a mosaic image generator by introducing parallelism using OpenMP in C++. The original sequential version processed pixel blocks one at a time, creating an opportunity for parallel execution. By using multiple threads to handle image sections together, we aim to reduce runtime while preserving output quality.

The C++ program was built based on a Python program originally implemented by `Rob Dawson` on GitHub (`https://github.com/codebox/mosaic`).

2 Program Design

The mosaic generator reconstructs a target image by replacing each region with the closest matching tile image based on pixel wise absolute difference. The program operates in three main stages:

1. **Tile Preparation:** Tile images are loaded from a specified directory, cropped to squares, and resized into two versions: a large tile for the final mosaic and a small tile used for fast similarity comparison.

2. **Target Image Processing:** The input image is loaded and enlarged to enhance the mosaic detail. The target image is then divided into non-overlapping blocks that match the dimensions of the tiles.

3. **Mosaic Generation:** For each block in the target image, the best matching tile is identified using pixel-wise color similarity. This tile replaces the corresponding block in the output mosaic image.

3 Sequential Implementation

The sequential implementation processes each image block independently in a nested loop, iterating over the enlarged target image by tile-sized steps. For each region:

1. The region is resized to a smaller resolution.
2. A linear search is performed across all preprocessed tile images to find the tile with the minimum pixel-wise difference.
3. The best matching tile is copied into the mosaic image at the corresponding location.

This approach can get computationally expensive, especially as the number of tiles or the resolution of the target image increases.

4 Parallelized Implementation

We parallelized two main components of the mosaic generator using OpenMP:

- **Tile Preparation:** When loading and preparing the image tiles, we parallelized the loop that resizes each image into both small and large versions. This step is independent across tiles, making it ideal for parallel execution.
- **Mosaic Generation:** While building the mosaic, we parallelized the nested loop that traverses the target image in tile-sized blocks. For each block, the program computes the best-matching tile based on color similarity and inserts the corresponding image. This process was independent from other block and parallelizing allowed multiple blocks to be processed simultaneously.

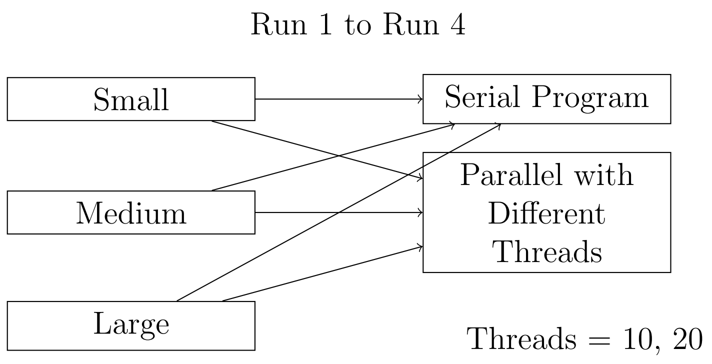
5 Analysis

To compare the performance of the serial and parallel programs, we designed a test that would provide different insights. We selected images based on their resolution and put them in one of three categories:

Category	**Resolution (px)**	**Images**
Small	256×256	bear.jpg, pokemon.jpg
Medium	1000×1000	car.jpg, tinfoil.jpg
Large	6000×4000 and 3381×4898	mountain.jpg, shoe.jpg

Table 1: Image categories by resolution

Two images were selected for each category, making a total of six images. Both the serial and parallel versions of the program were run on each image, with four separate runs for each program. The parallel program was tested again with a different number of threads. This setup allowed us to observe how the program's performance changed with image size and the number of threads used in the parallel version.



5.1 Analyzing Average Run Times

We will now look at how both programs performed with each image on average.

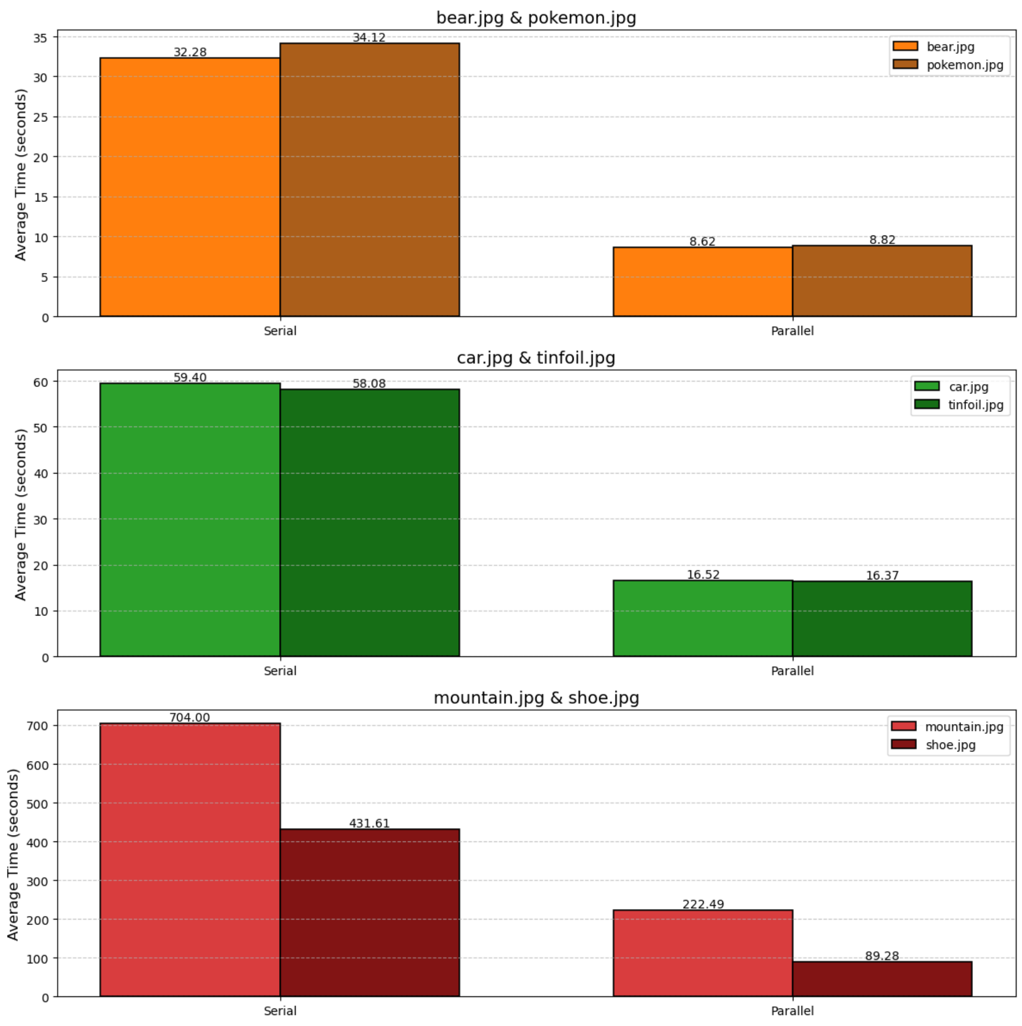


Figure 1: Average run times for serial and parallel (20 threads) programs

The above graphs were generated with the parallel program using 20 threads. For a given image size, the serial programs take approximately the same time (around 33 seconds for small images), and so do the parallel programs (around 8.5 seconds for small images).

The only outliers are the large images; this is likely due to one of the images being 6000 x 4000 pixels and the other being 3381 × 4898 pixels, resulting in the notable time difference. The parallel program is clearly beneficial and seems to have a speed-up of well over 3 times.

5.2 Indvidual Run Analysis

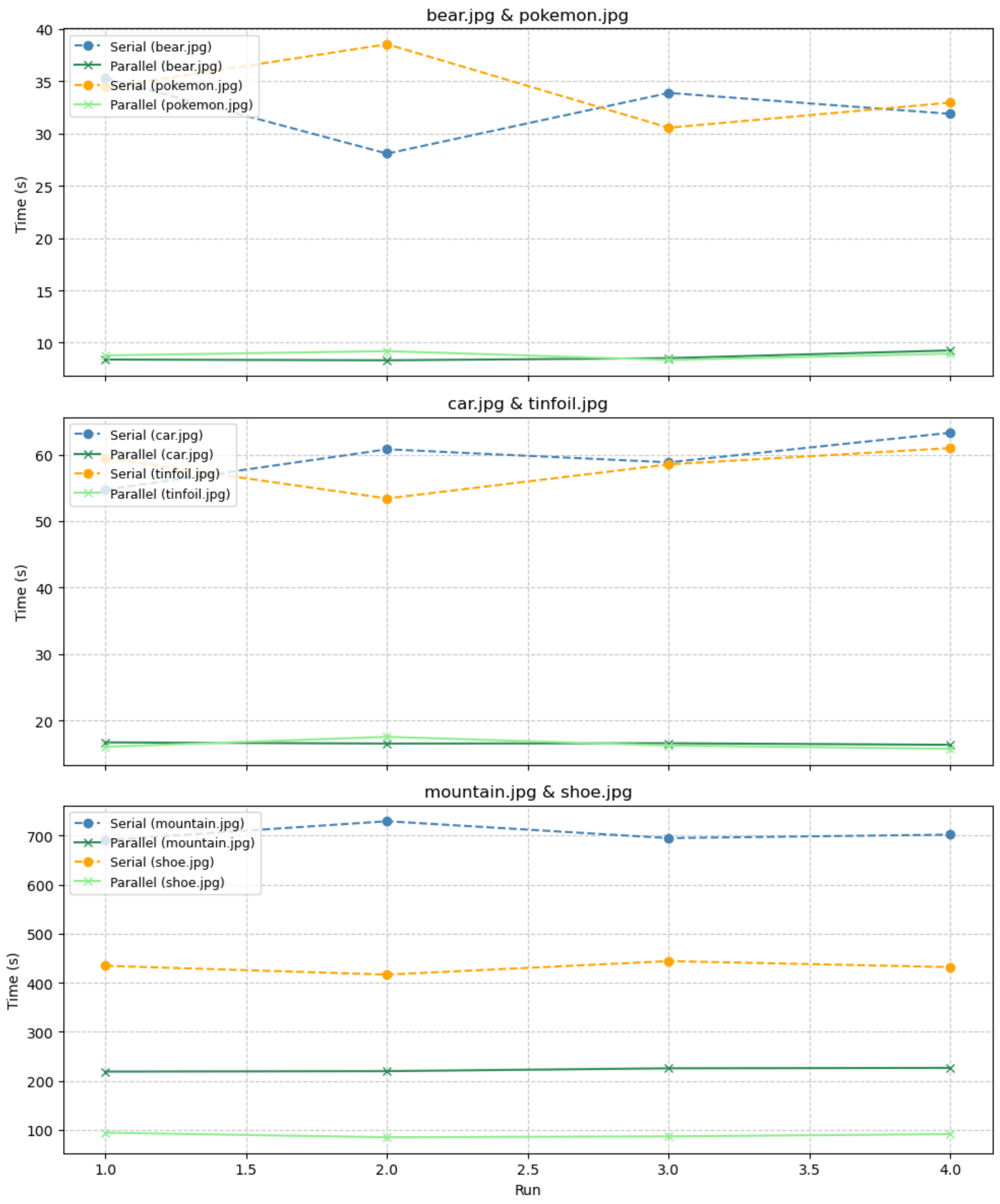


Figure 2: Individual run times for serial and parallel (20 threads) programs

The serial runs for each image show some variations; this is due to changes in at the system level, such as background processes and scheduling during the runs, but not enough to consider them outliers.

The parallel runs show much less variation, indicating that they can give more consistent performance.

This is unusual because the expectation would be that parallel execution would be more affected by system level factors. It could be that the longer runtime and reliance on a single execution thread make serial runs more sensitive to small environmental differences, while shorter, distributed workloads in parallel runs help "average out" those fluctuations.

5.3 Speed Up Comparison

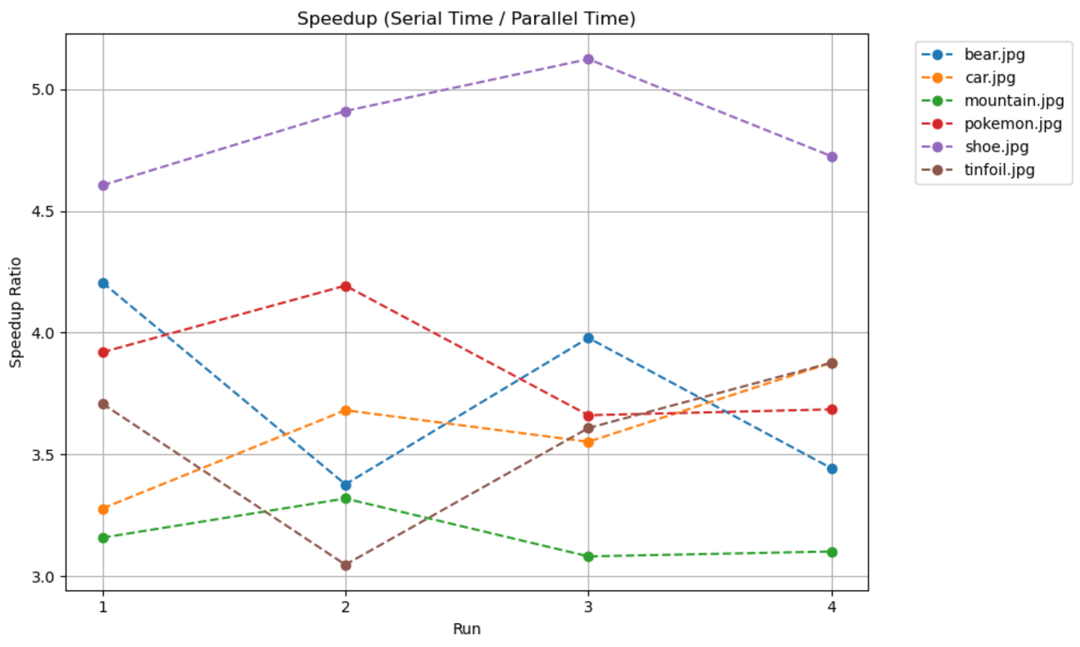


Figure 3: Speed up for each image (Parallel execution of 20 threads)

There is a fair amount of variation here, the shoe image had the greatest speed up while the mountain was the least, even though they were both in the category of large images.

The bear, car, and pokemon images had significant changes in speed up across runs. This change is primarily due to the variance we see in the sequential processing times.

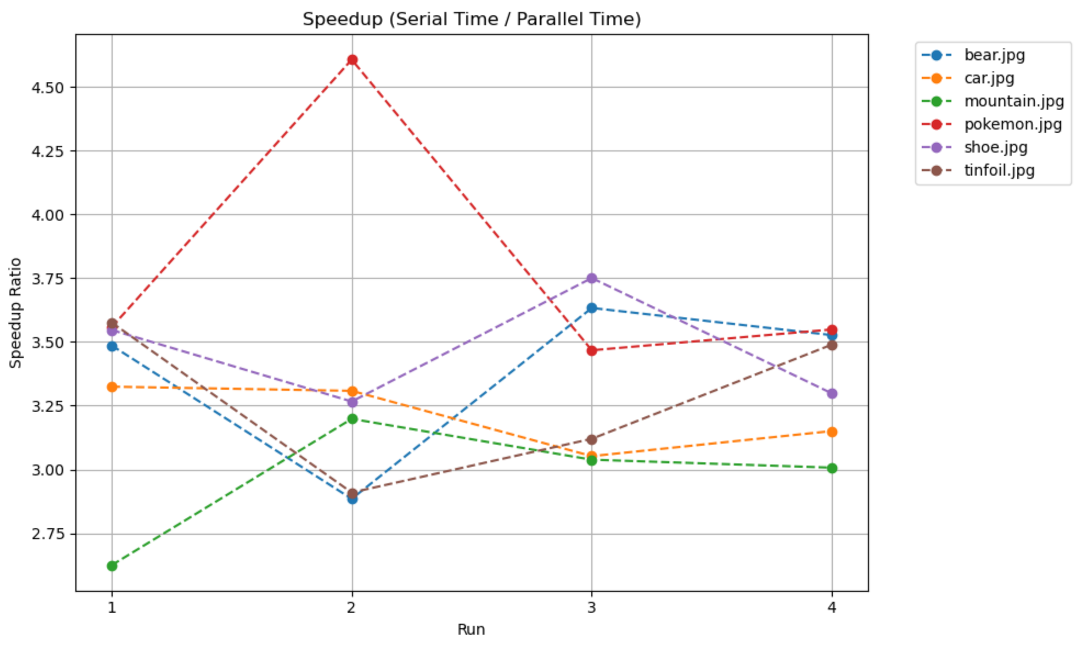


Figure 4: Speed up for each image (Parallel execution of 10 threads)

Overall, the speedup here is lower than the program running on 20 threads. This makes sense since having fewer threads limits the amount of computations that can happen in parallel. On the other hand, the speed up is fairly close to the 20 thread runs, which could be due to the processor reaching some limit.

Let us look at the averages of the speed up in each image.

Image	**Speedup (10 Threads)**	**Speedup (20 Threads)**
bear.jpg	3.38x	3.75x
pokemon.jpg	3.79x	3.86x
car.jpg	3.21x	3.60x
tinfoil.jpg	3.27x	3.56x
mountain.jpg	2.97x	3.17x
shoe.jpg	3.47x	4.84x

Table 2: Average speedup per image across runs using 10 and 20 threads

The speed up clearly decreases as the images sizes get larger, indicating that the fixed number of threads becomes less impactful as the processing needs increase. We would benefit from having more threads available.

The shoe image is clearly an outlier here. While other images showed only a slight speedup from 10 to 20 threads, the shoe image saw a much larger improvement. Our initial hypothesis was that the lower color complexity of the image, relative to the mountain image, accounted for the observed difference in speedup. To evaluate this, we conducted an experiment using two additional 4K images with similarly low color complexity. However, the resulting speedups of 2.676× and 3.478× did not support our hypothesis.

5.4 Results

Summary of results:

Image	Threads	Avg Time (s)	Avg Speedup
bear.jpg	1	32.28	1.00
	10	9.56	3.38
	20	8.62	3.75
pokemon.jpg	1	34.12	1.00
	10	9.04	3.79
	20	8.82	3.86
car.jpg	1	59.4	1.00
	10	18.55	3.21
	20	16.52	3.60
tinfoil.jpg	1	58.08	1.00
	10	17.80	3.27
	20	16.37	3.56
mountain.jpg	1	704.0	1.00
	10	238.27	2.97
	20	222.49	3.17
shoe.jpg	1	431.61	1.00
	10	124.81	3.47
	20	89.28	4.84

Table 3: Average time and speedup for each image with different thread counts

6 Conclusion

One of the main challenges was identifying parts of the code that could be safely parallelized. Although the image block matching operations were inherently parallel, care had to be taken to ensure that no two threads modified the same region of memory.

Even though the tasks could run separately, adding more threads did not keep improving the speed by a large margin after a point. This is probably because the threads started competing for access to memory or cache, which slowed things down.

Finally, we observed that as image size increased, the average speedup decreased. Despite this, the serial execution times were consistently around three times longer than their parallel counterparts, highlighting the efficiency gains achieved through parallelization.

References

- OpenMP Documentation: `https://www.openmp.org/specifications/`
- Python Mosaic Script: `https://github.com/codebox/mosaic`