Task Introduction

The task is to apply image effects on a series of images using 2D image convolutions. The project implements three versions of image editor that apply convolution effects on given images. The sequential version processes images one at a time without parallelism. Each image is fully loaded, effects are applied in-order using sequential convolution operations, and results are saved before moving to the next image. This version serves as the baseline for performance comparisons. The BSP version processes an individual image by splitting it into slices. This version has each goroutine apply the same effect on their own slices, wait for all slices to be completed between effects, and move on to the next effect instruction together. Finally, the BSP+Work-Stealing version allows the task (processing a series of images) to be split into smaller tasks (), which are placed in a work queue such that a thread will steal work from other threads when idle.

Instruction

The test runs each image combination of `mode, [number of threads], and data_dir` five times, and outputs the results into text files at `benchmark/results.`

```
Generating testing plots: /proj3/benchmark$: sbatch benchmark-proj3.sh

Usage: go run editor.go data_dir mode [number of threads]

    data_dir = The data directory to use to load the images

    mode     = (s) run sequentially

               (bsp) process slices of each image in parallel

               (bspsteal) bsp + work-stealing algorithm

    [number of threads] = Runs the parallel version of the program with the
specified number of threads
```

Data Source

Inside the `proj3` directory, the dataset directory should be downloaded and placed at the same level as subdirectories `editor` and `png`. Data can be downloaded: here

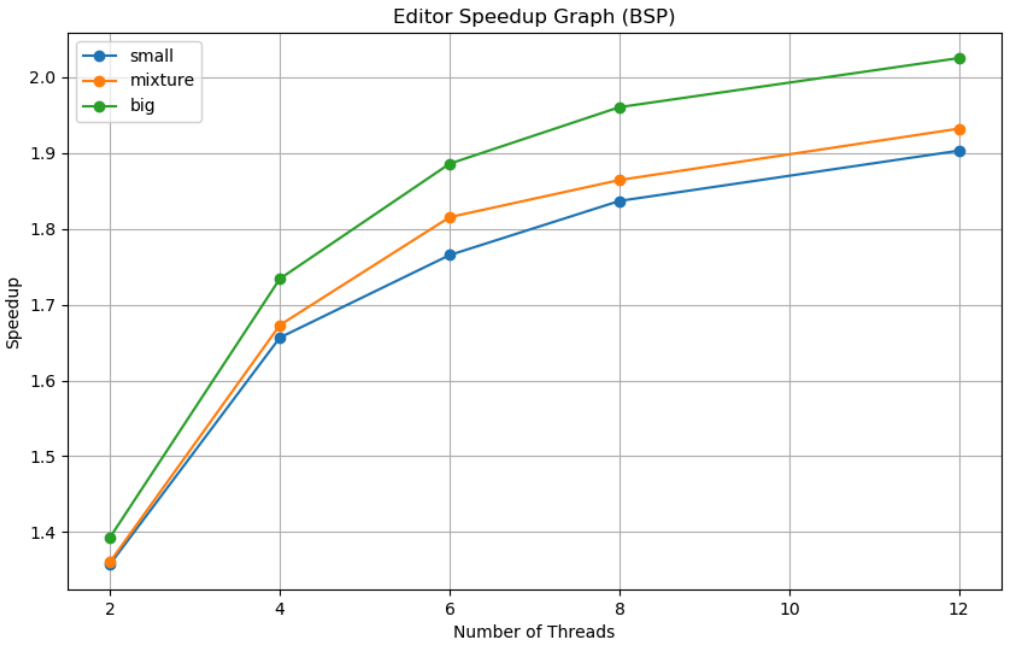
Sequential Hotspots

The main hotspot in the sequential program is the convolution operation, which requires multiple nested loops and kernel calculations for each pixel. File I/O operations (reading/writing PNG files) create sequential bottlenecks since loading and writing large image files create latency.

Parallel Implementations

1. **Bulk Synchronous Parallel (BSP):**
   The BSP pattern is implemented using phase barriers to coordinate parallel execution of image effects where each effect (e.g., blur, edge detection) represents a

superstep. First, each image is divided into horizontal slices, with each goroutine processing a slice (e.g., `BSPConvolution()` in effects.go splits images into `numThreads` slices). Within `BSPConvolution()`, a reusable `Barrier` struct is used to ensure the main thread and finished workers wait for all spawned sub-workers to complete their slice processing before advancing to the next effect. Additionally, after synchronization, `SwapBuffers()` exchanges input/output buffers for subsequent effects, preserving data consistency.



Design Rationale

This implementation offers advantages, particularly in terms of dependency management and predictable latency. Since specific convolutions require neighboring pixels, processing slices without waiting between effects could cause data races. Barriers ensure that no worker starts the next effect until all workers have finished the current one. Additionally, barriers bound the worst-case latency per effect.

Trade-off and Limitation

However, using barriers also introduces performance tradeoffs. Synchronization overhead grows with thread count due to: more threads → higher contention on the barrier's mutex/cond variables. On the other hand, the limitation of BSP-based design is that `SwapBuffers()` forces all threads to synchronize between effects, which is also a sequential bottleneck.

2. **BSP + Work-Stealing using Deque:**

Compared to the pure BSP pattern, in this version image tasks (`ImageTask`) are initially distributed round-robin to worker deques. After the `RunBSPSteal()` starts the program, whenever a worker's deque is empty, it steals tasks from others' heads, ensuring high throughput under uneven workloads. Here is the structure of the deque that enables work-stealing mechanism:

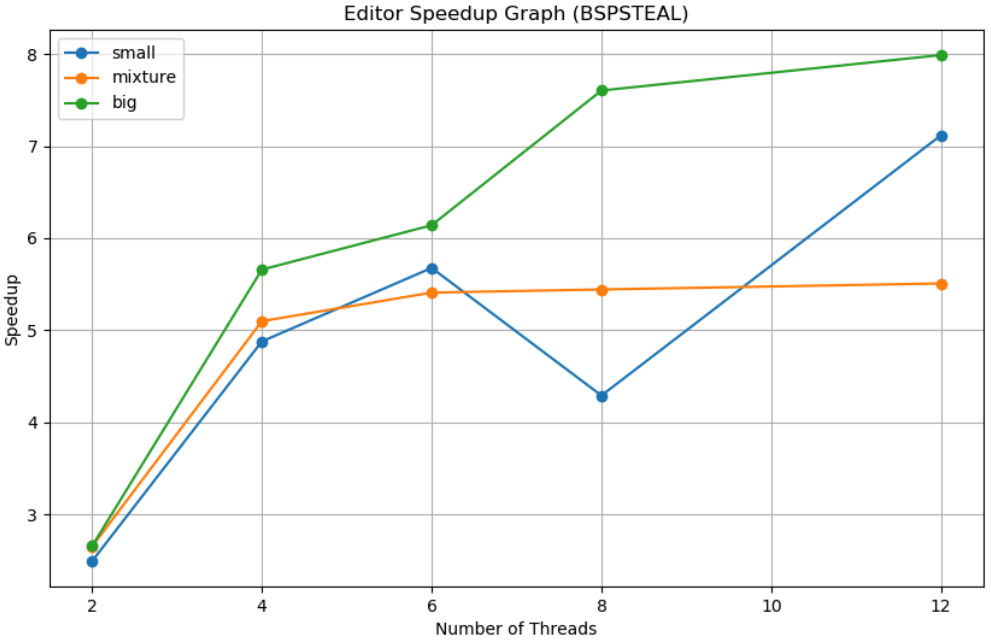
- Deque: A linked list of nodes with atomic operations on `head/tail` pointers.
- Operations:
    - Push/Pop (LIFO): Owner threads add/remove tasks at the `tail` using `CompareAndSwap` for thread safety.
    - Steal (FIFO): Idle threads steal tasks from the `head`, minimizing contention via atomic pointer swaps.

Design Rationale

In terms of the work-stealing design rationale, the mechanism provides excellent load balancing. In our context, ImageTasks vary in size and complexity (e.g., the length of effects to apply). In this sense, work-stealing prevents thread starvation, especially when processing a mix of large and small images. On the other hand, the design implements per-image task granularity to balance parallelism efficiency and synchronization overhead. Instead of dividing individual images into slices for stealing, which risks excessive fragmentation and cache thrashing, the system treats each image as an atomic task. This coarse-grained approach allows workers to process full images sequentially using BSP for intra-image parallelism, preserving spatial locality in pixel data while attempting to minimize deque contention since processing a full image keeps related pixel data close in memory. With M images and T threads, maximum steals ≈ M-T (vs M×T for per-slice stealing).

Trade-off and Potential Risks

The trade-off here is that while per-image stealing reduces synchronization overhead, if one or a few threads are assigned disproportionately large or complex tasks compared to the rest, this can lead to underutilization despite work-stealing for mixed workloads, as those threads become bottlenecks. However, this design prioritizes simplicity over perfect load balancing and does not overly concern itself with such extreme cases, as ImageTasks are distributed in a round-robin manner. By stealing entire images rather than slices, workers avoid fine-grained synchronization and maintain predictable memory access patterns, which is critical for convolution-heavy effects that introduce pixel dependency.


Editor Speedup Graph (BSPSTEAL)
small
mixture
big
Speedup
Number of Threads
8
7
6
5
4
3
2
4
6
8
10
12