Project 3: Parallel System for Medical Image Processing

Sonia Sharapova

December 2024

1 Project Overview

Medical image processing plays a crucial role in modern healthcare, particularly in the analysis of sequential imaging data. This report examines a parallel processing system designed to analyze DICOM (Digital Imaging and Communications in Medicine) image sequences using optical flow analysis. The system implements two distinct parallel processing approaches: pipeline parallelism and work stealing. Each approach offers unique advantages in handling the computational challenges of medical image processing.
The primary objectives of the system are to process DICOM image sequences, detect motion between frames using optical flow analysis, and generate visualizations of the detected motion patterns. This analysis focuses on the implementation details, benefits, and challenges of both parallel processing approaches.

The core functionality includes:

- 1. Loading and preprocessing DICOM files.
- 2. Feature detection using Shi-Tomasi corner detection
- 3. Optical flow computation using the Lucas-Kanade method
- 4. Visualization of the motion analysis as animated GIFs

Medical images are used as input, and gifs with the optical motion vectors overlayed on top of them are the output.

2 System Overview

Core Functionality

The system processes DICOM images through several key stages. Initially, it loads and preprocesses DICOM files, performing necessary normalization and histogram equalization. It then applies Shi-Tomasi corner detection to identify trackable features in each frame. The system uses these features to compute optical flow using the Lucas-Kanade method, tracking movement between consecutive frames. Finally, it generates animated GIF visualizations to display the detected motion patterns.

3 Code Overview

Project Structure

Packages:

- cmd: Contains main functions that runs from command line. Includes single use and full benchmarking.
- dicom: DICOM handling package.
- flow: Optical flow package.
- viz: Visualization.

```
proj3/
├── benchmark/
│   └── benchmark.go           # Benchmarking performances
├── cmd/
│   └── benchmark/
│       └── main.go            # to run all
│   └── process-dicom/
│       └── main.go            # single mode run
├── dicom/
│   └── preprocessing.go       # File loading and image preprocessing
├── flow/
│   ├── features.go            # Feature detection
│   └── optical_flow.go        # Optical flow computation
├── proc/                      # Program execution modes
│   ├── pipeline.go
│   ├── sequential.go
│   ├── types.go
│   └── workstealing/
│       ├── deque.go
│       └── processor.go
├── scripts/
│   └── plot_results.py        # Plots benchmarking results on a plot
└── viz/
    └── visualization.go       # Generates gifs as output
```

4 Dataset

To detect motion of the contracting heart, I used CineMRI data: a series of images that capture the heart's motion during a cardiac cycle. These images are taken by repeatedly imaging the heart at a single location during the cardiac cycle. Playing the frames together shows movement. I obtained my data from the AMRG Cardiac Atlas which provides a complete set of labeled MRI images of a patient's heart.

AMRG Cardiac Atlas and can be downloaded here.

For more manageable runs, I made a custom subset containing a smaller portion of these files, which can be found in the directory /smallerDataset.

Input: Read in DICOM data
Output: Produces gifs with motion vectors overlayed.

5 Usage

Independent Runs: Sequential:

$ go run ./cmdprocess-dicom/main.go -input {INPUT_FOLDER} -output ./output/sequential/ -mode sequential

Pipeline:

$ go run ./cmd/process-dicom/main.go -input {INPUT_FOLDER} -output ./output/pipeline/ -mode pipeline -workers {WORKERS} -buffer 10

Work Stealing:
$ go run ./cmd/process-dicom/main.go -input {INPUT_FOLDER} -output ./output/workstealing/ -mode workstealing -workers {WORKERS}

Run All (Benchmarking)
$ go run ./cmd/benchmark/main.go -input {INPUT_FOLDER} -maxworkers {MAX_WORKERS}

6 Sequential Implementation (Baseline)

The sequential version processes each stage linearly:
LoadFiles → Preprocess → DetectFeatures → ComputeFlow → Visualize → SaveGIF

In the sequential mode, the program processes each folder of DICOM files one at a time, following a linear sequence of operations. Each folder's files are preprocessed, features are detected, optical flow is computed between consecutive frames, and visualization frames are generated with results are then saved as a GIF. This implementation is straightforward but limited by the sequential nature of its execution, which utilizes only a single thread. While it is simple and free from synchronization overhead, it has longer processing times for large datasets.

7 Pipeline Implementation

The pipeline implementation divides processing into four concurrent stages, each handling a specific aspect of the workflow and processing data independently and concurrently. This design allows multiple images to be processed simultaneously at different stages, improving overall throughput.

The stages are:

Stage 1: Folder Reader Stage

- Scans input directories for DICOM files
- Groups files by folder
- Creates output directory structure
- Feeds folder data to the next stage through a buffered channel

Stage 2: Image Processor Stage

- Performs DICOM preprocessing (normalization, histogram equalization)
- Detects features using Shi-Tomasi corner detection
- Employs worker pool pattern for parallel image processing
- Maintains image sequence order for optical flow analysis

Stage 3: Optical Flow Computer Stage

- Computes motion between consecutive frames
- Tracks feature points across frame pairs
- Filters and validates tracked points

- Generates visualization data

Stage 4: GIF Writer Stage
- Collects processed frames
- Assembles frames into animations
- Writes output GIF files

7.1 Pipeline Implementation Details

The pipeline uses Go's channels for inter-stage communication, implementing bounded buffering to manage memory usage and provide natural backpressure. Each stage operates independently but coordinates through these channels, allowing for efficient resource utilization.
Resource management is handled through worker pools, particularly in the computationally intensive image processing stage. Error handling is implemented across all stages, ensuring validation and proper resource cleanup.

7.2 Benefits and Challenges

The pipeline architecture is appropriate and helpful for this program due to its natural alignment with the sequential nature of image processing. It provides data locality, as each stage maintains its own working set, reducing memory contention and improving cache utilization. The buffered channels between stages help absorb variations in processing speed, providing natural load balancing. This implementation does however encounter challenges in state management, particularly in maintaining frame order for optical flow analysis and managing cleanup of intermediate results.

8 Workstealing

The work-stealing model dynamically distributes tasks among worker threads using lock-free deques. Tasks, such as processing batches of frames, are initially distributed among workers, and if a worker finishes its tasks, it attempts to "steal" tasks from others' deques (ensuring better load balancing). This approach maximizes resource utilization, especially when some tasks require more time than others, by dynamically redistributing work as

needed.

The implementation includes:

- The initial task distribution
- The work stealing process
- Lock-Free operations

For this program, workstealing is beneficial because the workload (processing DICOM frames) can be naturally divided into independent tasks, and processing times may vary between frames. The implementation uses double-ended queues (deques) for each worker, where tasks consist of batches of DICOM frames that need to be processed. Workers primarily process tasks from the bottom of their own deques, maintaining good cache locality, while idle workers become "thieves" that attempt to steal work from the top of other workers' queues. This approach provides natural load balancing without requiring central coordination through lock-free operations using atomic primitives, ensuring thread safety while minimizing synchronization overhead.

9 General Implementation Challenges:

OpenCV's Mat objects required meticulous memory management to avoid issues such as memory leaks and double-free errors, particularly in the pipeline implementation. Additionally, the data dependencies inherent in optical flow computation, which relies on consecutive frames, imposed limitations on parallelization. Load balancing presented another challenge, as the computational costs varied significantly across stages: preprocessing accounted for 10.4% of the total time, visualization for 26.7%, while feature detection and optical flow were relatively quick, consuming only $1 - 2\%$. Finally, the pipeline implementation demanded careful synchronization between stages to prevent data races and ensure smooth execution.

I originally wrote my implementation on my local device and faced unexpected hurdles when trying to install OpenCV on the Linux Cluster. All libraries had to be installed manually and some errors were unresolved.

10 Hotspots and Bottlenecks

Hotspots (parallelizable sections):
Parallelizable sections, or hotspots, included frame preprocessing, which accounted for 10.4% of the sequential execution time, visualization, which took 26.7%, and the independent processing of frames within batches.

Bottlenecks (sequential sections):
Bottlenecks arose in sequential sections, such as initial file loading and organization, final GIF compilation, and the dependencies between consecutive frames required for optical flow computation. These bottlenecks highlighted areas where further optimization was constrained by inherent data dependencies.

11 Results:

Speedup Results The experimental results show:

Execution Time for Processing in Parallel (in miliseconds)			
Workers	Sequential	Pipeline	WorkStealing
seq	301435	-	-
2	-	105947	199751
4	-	97083	183366
6	-	93632	179097
8	-	94499	174736

Pipeline Implementation: - 2 workers: 2.84x speedup
- 4 workers: 3.10x speedup
- 6 workers: 3.21x speedup
- 8 workers: 3.18x speedup

Work-Stealing Implementation: - 2 workers: 1.50x speedup
- 4 workers: 1.64x speedup
- 6 workers: 1.68x speedup
- 8 workers: 1.72x speedup

Pipeline reached a maximum speedup of 3.22x at 6 workers and work-stealing reached a maximum speedup of 1.73x with 8 workers.

This speedup graph illustrates the performance of Pipeline and Work-Stealing. When compared to ideal speedup, their performance doesn't show very significant improvement..

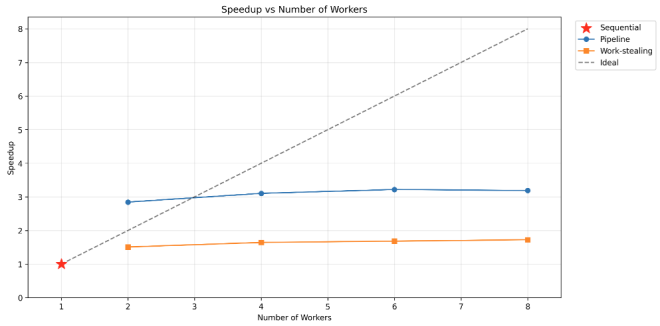

Figure 1: Speedup graph for parallelization against achievable speedup.

The results highlight the performance of sequential, pipeline, and work-stealing modes, with sequential execution serving as the baseline. While both pipeline and work-stealing implementations improved performance over the sequential baseline, the pipeline mode demonstrated far superior speedups, reaching over 3x improvement at its peak. Work-stealing, while more flexible, was hindered by synchronization and task management overheads, achieving only modest speedups. Both implementations showed diminishing returns as the number of workers increased. Pipeline Mode exhibited a slight drop in performance at 8 workers, likely due to increased overhead and workload imbalance across pipeline stages, while Work-Stealing showed gradual improvements with more workers but at a significantly slower rate. This suggests that synchronization and task distribution overheads outweighed the benefits of additional workers. These findings demonstrate the effectiveness of parallelization while highlighting the scalability limits imposed by sequential bottlenecks and synchronization overhead.

12 Conclusion

This project explored sequential, pipeline, and work-stealing parallelization strategies for processing computational tasks. The results demonstrated that parallelization significantly improves performance compared to sequential execution, with the pipeline implementation achieving the highest speedup of 3.22x at 6 workers. This approach capitalized on its structured flow, efficiently overlapping computational stages to maximize resource uti-

lization. However, it experienced diminishing returns beyond 6 workers due to increased synchronization overhead and stage imbalances.

The work-stealing implementation provided a more flexible approach, dynamically redistributing tasks among workers to improve load balancing. Despite this, it achieved only a maximum speedup of 1.73x at 8 workers, constrained by the overhead of task management and synchronization. The modest gains suggest that the work-stealing model may be less suitable for this workload's characteristics, such as small task granularity and data dependencies.

Overall, the pipeline model emerged as the more effective strategy for this application, striking a balance between simplicity and parallel efficiency. The results underscore the importance of tailoring parallelization strategies to specific workloads and the trade-offs between dynamic flexibility and structured predictability. Future efforts could explore hybrid approaches combining pipeline and work-stealing methods, as well as optimizing task granularity and reducing synchronization costs to further enhance scalability and performance.

It is important to consider that these methods were run on a subset of the full available data, so the results could differ if executing on the entire set. Although the program was run on the linux server, it was not run on the peanut cluster which would significantly improve parallelization.

13 References

DICOM Data: AMRG Cardiac Atlas

Paper on Cine MRI Data: National Library of Medicine

Motion Estimation with Optical Flow: Optical Flow Information

Optical Flow in OpenCV: OpenCV

Optical flow with Go: Go Implementation

Feature Detection Using Shi-Tomasi Corner Detection: OpenCV Informational

Feature Detection Implementation: gocv library

Go Image Library: Image Documentation

Pipeline with Go: Go Documentation

Pipeline Approach Example: Pipeline file management

Implementing Work Stealing: Task Scheduler Implementation

Work Stealing Task Distribution: Using Round-Robin

+ Lecture notes and examples from class and golang