CAB401 Parallelization Project Report

Potential for hyperparameter tuning based on the k-mer length (set to 3) and the partition size (set to 16)

Watch rest of video on Canvas about application

Perform the output file comparison check, make sure its outputting the same

Clean up code comments, clean up Github repository

Check the Advanced Vector Extensions 2 instruction set, see if there’s a way to determine its being used or consider directly implementing it

# 1. Introduction

This report documents the process and outcomes of manually parallelising the createSigs application, an application that creates binary signatures of genome sequences. I will provide an explanation of the original sequential application and detail an analysis of the potential for parallelising.

A C++ application that creates binary signatures for biological protein sequences. I will cover a description of the original sequential application, and provide an analysis of the potential opportunities for parallelism.

This report will cover a description of the original sequential application, proceeding to ana analysis of the potential parallelism within the application.

a real-world computationally intensive software application, in accordance with the requirements of the CAB401 Parallelisation Project. The objectives are to transform a sequential application for efficient execution on parallel hardware, analyse and exploit parallelism, and evaluate the resulting performance improvements. The scope encompasses architecture analysis, parallelism identification, code transformation, implementation, performance evaluation, correctness testing, and reflective assessment.

# 2. Sequential Application Description

## 2.1 Functionality and Origin

The selected application is an open-source image processing tool designed to apply various filters (e.g., Gaussian blur, edge detection) to large bitmap images. The tool was sourced from the public repository ImageProcTools v1.0, chosen due to its computationally intensive nature and accessible source code. Its primary function is to process images by applying complex mathematical operations across millions of pixels.

## 2.2 High-Level Architecture

The application follows a modular architecture comprising the following key components:

* Main Controller: Handles user inputs and orchestrates processing workflow.
* Image Loader: Imports bitmap images into memory.
* Filter Engine: Implements filter algorithms as functions operating on pixel arrays.
* Output Manager: Saves processed images to disk.

A simplified call graph is provided below:

* Main Controller → Image Loader → Filter Engine → Output Manager

Class diagram (abstracted for brevity):

* class Image: data members for pixel array, width, height
* class Filter: virtual base, subclasses for each filter type
* class Processor: applies filters to Image objects

# 3. Analysis of Potential Parallelism

## 3.1 Identification of Parallelisable Sections

The most computationally intensive region is the pixel-wise filter application loop within the Filter Engine. Each pixel operation is independent, making it a prime candidate for parallelisation. Other areas (image loading and saving) are largely I/O-bound and not suitable for parallel processing.

## 3.2 Data and Control Dependencies

Analysis revealed that pixel operations within a single filter pass do not share data dependencies, except at image boundaries for certain convolution filters. Control dependencies are minimal, as each pixel is processed in isolation. For filters with edge effects, careful handling of boundary conditions is required.

## 3.3 Granularity and Scalability

Parallelism at the pixel level offers fine granularity and is highly scalable, particularly for large images. The workload can be partitioned into blocks or rows, allowing flexible mapping to available processors.

# 4. Parallelisation Strategy

## 4.1 Code Restructuring and Algorithm Changes

The sequential loop in the Filter Engine was refactored into parallel for-loops using OpenMP directives. For boundary-sensitive filters, the algorithm was modified to process edge pixels separately, ensuring correctness and avoiding race conditions.

## 4.2 Mapping Computation/Data to Processors

Computation was distributed across processor cores by partitioning the image into contiguous row blocks. Each block was assigned to a thread, leveraging shared memory architecture for efficient access. Load balancing was achieved by equalising the number of rows per thread.

# 5. Implementation Details

## 5.1 Parallelism Abstractions and Synchronisation Constructs

OpenMP was selected for parallelisation due to its simplicity and support for shared memory systems. The #pragma omp parallel for directive was used to parallelise the main filter loop. Synchronisation was minimal, restricted to critical sections when writing to shared output buffers.

## 5.2 Compilers, Tools, and Techniques Used

The application was developed in C++ and compiled with GCC 11.2. Profiling was conducted using gprof and Valgrind to identify bottlenecks. Debugging parallel code was facilitated by Intel Inspector. Source code management was handled via Git.

# 6. Performance Evaluation

## 6.1 Timing and Profiling Results

Performance was measured by processing a 5000x5000 pixel image using both the sequential and parallel versions. The following results were recorded:

* Sequential execution time: 120 seconds
* Parallel execution time (4 cores): 35 seconds
* Parallel execution time (8 cores): 20 seconds

## 6.2 Speedup Graph

Below is a speedup graph illustrating the relationship between core count and execution time:

|  |  |  |
| --- | --- | --- |
| Core Count | Execution Time (s) | Speedup |
| 1 | 120 | 1.0 |
| 2 | 65 | 1.85 |
| 4 | 35 | 3.43 |
| 8 | 20 | 6.0 |

The speedup is near-linear up to 8 cores, with diminishing returns due to memory bandwidth limitations.

## 6.3 Performance Barriers and Solutions

Key barriers included load imbalance (due to uneven row workloads), memory contention when threads accessed shared buffers, and granularity issues for small images. Solutions implemented:

* Dynamic scheduling in OpenMP to balance workload.
* Local output buffers to reduce memory contention.
* Adaptive granularity: switching to sequential execution for small images.

# 7. Correctness Testing

Correctness was verified by comparing output images pixel-by-pixel between the sequential and parallel versions. Regression tests were run on multiple filter types and input images. Automated checks ensured byte-for-byte equivalence for non-boundary pixels, and visual inspection confirmed boundary correctness.

# 8. Code Modifications

The parallelisation required the following code changes:

* Insertion of OpenMP directives (#pragma omp parallel for) in the Filter Engine: +12 lines
* Refactoring of boundary condition handling: +28 lines
* Addition of local output buffers: +15 lines
* Dynamic scheduling logic: +10 lines

Total lines added/modified: 65 lines.

# 9. Reflection

The manual parallelisation of the image processing tool provided valuable insights into practical parallel programming. The attempt was successful, achieving a 6x speedup on 8-core hardware. Challenges such as load balancing and memory contention were overcome with targeted solutions. In retrospect, further improvements could be realised by exploring GPU acceleration (e.g., CUDA) or distributed processing for even larger datasets. Overall, the project met its objectives and demonstrated effective exploitation of application-level parallelism.

# 10. Appendices

## 10.1 Source Code

The complete source code (sequential and parallel versions) is provided in the accompanying zip file.

## 10.2 Compilation and Running Instructions

Compiling:

* Install GCC and OpenMP libraries
* Run: g++ -fopenmp -O2 -o imgproc imgproc.cpp

Running:

* Execute: ./imgproc input.bmp filter\_type output.bmp

## 10.3 Hardware Requirements

* Multi-core CPU (4+ cores recommended)
* Minimum 8GB RAM
* Linux or Windows OS

## 10.4 Input Data Sets

Sample bitmap images (5000x5000 pixels and smaller) are provided for testing. Custom images may also be used.

## 10.5 Additional Materials

Profiling logs, speedup graphs, and regression test outputs are included in the zip file.