Performance Comparison of Parallel Image Processing Methods

Group 8 - CS22BT043, MC22BT004, CS22BT006,CS22BT004

October 21, 2024

1 Introduction

This report presents a comparison between three different methods for parallelizing an image processing application across three cores:

1. Processes with Pipes
2. Processes with Shared Memory and Semaphores
3. Threads with Atomic Operation

The aim of this study is to evaluate the performance (in terms of run-time and speed-up) of each method and discuss the challenges encountered during the implementation.

2 Methodology

Each method was implemented such that each core performed a specific task:

- **Core 1 (S1)**: Smoothening
- **Core 2 (S2)**: Detailing
- **Core 3 (S3)**: Sharpening and File Writing

The image was processed 1000 times to simulate a compute-intensive scenario. Communication and synchronization between the stages were implemented differently for each method, and the performance was measured.

3 Time Comparison

The following table shows the actual run-time for each method based on the experiment:

Method	Execution Time (1.ppm) (in seconds)
Processes with Pipes	4.37377
Processes with Shared Memory (Atomic Operations)	7.65179
Threads with Semaphores	4.2543

Table 1: Run-time Comparison of Parallel Methods

4 Problems Faced and Solutions

Throughout the implementation, we encountered several challenges:

Problem Faced	How It Was Solved
Process Synchronization (Pipes): Ensuring data was transferred in the correct order between processes. How much data one can write into the pipe?	Wrote only width*3 into the pipe one at a time. Using fcntl to increase the pipe size for correctness of the program in all cases(.ppm files).
Shared Memory (Atomic Operations): Race conditions when multiple processes accessed shared memory simultaneously. What data type to use while writing in the shared memory? Ensuring the correct flow of the input and output in the program. Placement of the semaphores and ensuring the flow of S1 - S2 - S3.	Employed semaphores operations to ensure safe updates to the shared memory, preventing conflicts. Employed array data structure to pass the flattened out image into the shared memory. Ensured that after reading from the shared memory, it was reset and that it didn't use it again without another process updating it.
Thread Synchronization (Semaphores): In thread - Initially I was executing the thread t1,t2,t3 1000 times but the program was killed forcefully by OS after around 640 iterations. However in different laptop the number was different due to different RAM size . Mine was 8 GB so it was working well till 640 iterations. The issue with this was that in each thread I was creating a new memory for smoothened, details and sharpened images inside S1, S2, S3 respectively. Due to this each time space was being allocate in head due to this after 640 iterations entire RAM and swap space was filled and no space was left for heap. Due to this the program got crashed or forcefully killed by OS.	Since t1, t2, t3 are thread of same process they share the global variables so declare them as global variable. So instead of allocation space in each thread I allocated them all space inside main itself. Also I changes the return type of S1, S2, S3 now instead of return a pointer of image they simply override to the same memory space again and again.

Table 2: Problems Faced and How They Were Solved

5 Conclusion

The study shows that using threads with semaphores offers the best performance improvement in terms of speed-up and execution time. Processes with shared memory also perform well but are more complex to manage. Processes with pipes, while simple, suffer from slower execution times due to the overhead of inter-process communication.