lab4 Parallel Applications Report

220010015(C. Harshitha Reddy)
220010032(M. Keerthi)
220010036(N. Soumya)

Contents

1 Introduction

commands can be used to build and run all :
make all
make run

This report explores different methods of implementing parallel applications on a multi-core processor setup. The assignment involves evaluating three

different approaches for parallelizing a baseline image processing application: using processes with pipes, processes with shared memory and semaphores, and threads with atomic operations. Each approach is implemented and studied in terms of execution time, speedup, and ease of implementation.

2 Baseline Application

The baseline application performs image processing sequentially, with no parallelism involved. This part is based on the submission from Laboratory 1, where the image transformation is performed on a single core.

3 Parallel Implementations

We implemented three different parallelization approaches using three cores of a processor. The objective was to process the image in three stages (S1: smoothen, S2: find details, and S3: sharpen) using three different cores, where the data is communicated between cores without waiting for the entire image to be processed.

3.1 Part 2-1: Pipes for Inter-Process Communication

In this approach, S1, S2, and S3 are handled by three different processes, and communication is achieved using pipes. Each process completes its assigned task and sends the resulting image data to the next process through pipes.

Highlights:

- Processes: Three separate processes.
- Communication: Pipes (unidirectional data transfer between processes).
- Tasks:
  - S1 performs image smoothening and passes the data to S2.
  - S2 finds details in the smoothened image and passes them to S3.
  - S3 sharpens the image and writes the final result to a file.

3.2 Part 2-2: Shared Memory with Semaphores

In this implementation, S1, S2, and S3 are performed by three processes communicating through shared memory. Synchronization between the processes is handled using semaphores to ensure that the operations are performed in the correct order.

Highlights:

- Processes: Three separate processes.
- Communication: Shared memory (shared access to image data).
- Synchronization: Semaphores ensure each process completes before the next one starts.
- Tasks:
  - S1 smoothens the image, stores the result in shared memory, and signals S2.
  - S2 finds details in the image, stores the result in shared memory, and signals S3.
  - S3 sharpens the image and writes the final result to a file.

3.3 Part 2-3: Threads with Atomic Operations

This approach uses threads within a single process, where data is communicated through shared memory, and synchronization is managed by atomic operations.

Highlights:

- Threads: Three threads within a single process.
- Synchronization: An atomic flag '$stage\_flag$' is used to control the execution order between the threads.
- Tasks:
  - S1 (Thread 1) smoothens the image and sets the atomic flag.
  - S2 (Thread 2) finds details and updates the atomic flag.
  - S3 (Thread 3) sharpens the image and stores the final result.

4 Proving Correctness of Data Transfer

In all three parallel implementations, the correctness of data transfer between cores (or processes/threads) was verified by ensuring that the pixels were processed and transferred in the correct order. A simple checksum-based validation was used at each stage to confirm that the data remained intact as it moved from S1 to S2 to S3.

5 Performance Analysis

The following experiments were conducted to evaluate the runtime and speedup of each approach:

- Baseline (single-core execution)
- Parallel execution using processes with pipes
- Parallel execution using processes with shared memory
- Parallel execution using threads with atomic operations

Each of the implementations was modified to be compute-intensive by performing the image transformation 1000 times. The file reading and writing steps were limited to once to avoid IO overhead dominating the execution time. The speedup obtained by using three cores was measured and compared across the implementations.

5.1 Results

The table below shows the execution times and speedups observed for each approach:

Approach	Execution Time (s)	Speedup
Baseline (Sequential)	20.0653	1x
Pipes	21.8499	1.089x
Shared Memory	6.01743	0.3x
Threads	11.1813	0.56x

6 Discussion

6.1 Ease of Implementation

Among the three approaches, using threads with atomic operations was the easiest to implement due to the ability to share memory within the same pro-

cess. Synchronization using atomic operations was straightforward compared to semaphores. However, debugging thread-related issues can be challenging.

The approach using shared memory and semaphores, while faster than pipes, was more difficult to implement due to the complexity of synchronization. Pipes, though slower, were relatively easier to set up for inter-process communication.

6.2 Speedup and Performance

As expected, the file reading and writing overhead limited the speedup achievable with three cores. By focusing on computation, we observed moderate speedups with the parallel approaches. The shared memory approach provided the best performance due to lower communication overhead compared to pipes and threads.

7 Conclusion

This laboratory exercise demonstrated different ways of parallelizing an application on a multi-core processor. Shared memory with semaphores provided the best performance and ease of implementation, while pipes and threads had their trade-offs in terms of complexity and speed. The results highlight the importance of choosing the right parallelization strategy based on the application's communication and synchronization needs.