LAB ASSIGNMENT-5(cs23bt013,cs23bt063)

Part-1: Serial execution(lab0)

In sequential execution, program executes all three stages one after the other (S1,S2,S3) using a single core.
Each stage processes the entire image completely before passing its output to the next stage. That is,

1. Stage S1 reads the input image and produces a fully smoothened version.

2. Once S1 finishes processing all pixels, the smoothened image is passed to S2, which computes the details.

3. Finally, S3 uses the output of S2 to produce the sharpened image and writes it to the output file.

Execution time:

The total execution time is measured as the sum of the times taken by all three stages (S1+S2+S3). This version serves as the baseline for evaluating speedup and efficiency in the parallel implementations in other part (part2).

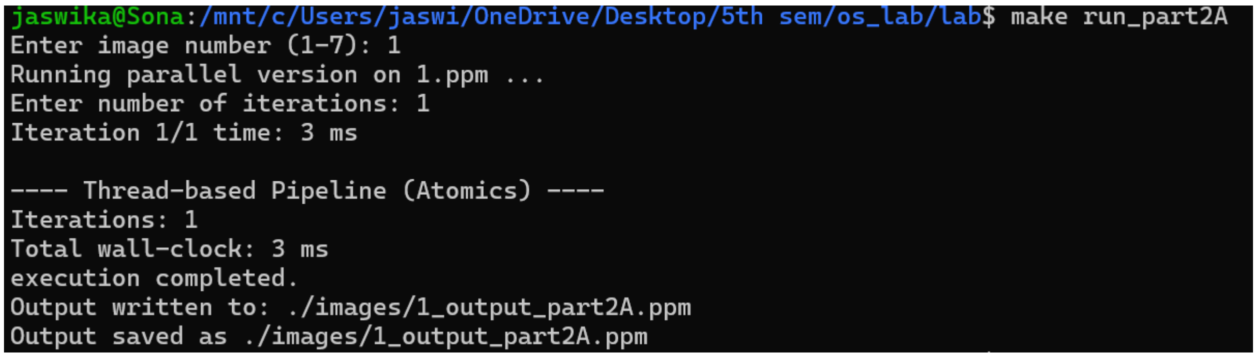
image before:

image after sharpening:

PART-2
1.Multi threaded with atomics:

Each stage runs in its own thread. Shared memory buffers are used for communication and atomic variables synchronize pixel readiness between threads.

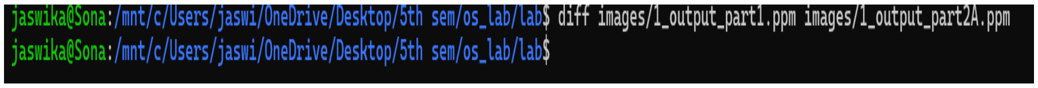
1-iteration:



```
jaswika@Sona:/mnt/c/Users/jaswi/OneDrive/Desktop/5th sem/os_lab/lab$ make run_part2A
Enter image number (1-7): 1
Running parallel version on 1.ppm ...
Enter number of iterations: 1
Iteration 1/1 time: 3 ms

---- Thread-based Pipeline (Atomics) ----
Iterations: 1
Total wall-clock: 3 ms
execution completed.
Output written to: ./images/1_output_part2A.ppm
Output saved as ./images/1_output_part2A.ppm
```

Pixel order correctness:
diff method: diff is a **Linux utility** that compares **two files line by line (or byte by byte)** and shows their differences.
If there is no output if diff command runs then diff conforms that
1.each pixel value in this result matches with the sequential version
2. Pixels are read or written in the same exact order as in the sequential version.
3. All data successfully transferred across threads/processes/pipes.



```
jaswika@Sona:/mnt/c/Users/jaswi/OneDrive/Desktop/5th sem/os_lab/lab$ diff images/1_output_part1.ppm images/1_output_part2A.ppm
jaswika@Sona:/mnt/c/Users/jaswi/OneDrive/Desktop/5th sem/os_lab/lab$
```

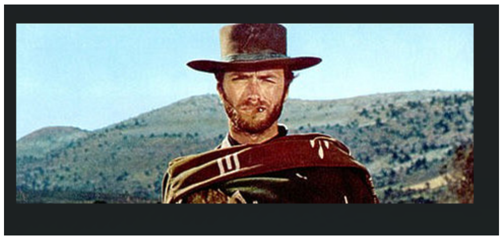
Advantages of including this:
1. All three stages can execute concurrently.
2. Efficient use of shared memory, there is no use of pipes or shared memory setup.
3. No ipc overhead.

Difficulties faced:

1. Harder to debug: race conditions or incorrect atomic ordering can lead to partial/inconsistent images
2. Requires precise synchronization s1-index,s2-index and s3-index must be updated correctly, otherwise S2 or S3 threads can stall or read incomplete data.
3. CPU utilization must be balanced — if one stage (say S1) takes longer, the others may idle waiting.

Input image :

sharpened image using atomics:



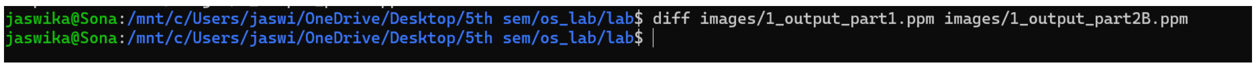
CONCLUSION:
The atomic-based multi-threaded pipeline achieved **partial parallel speed-up** while maintaining perfect correctness.
However, due to fine-grained synchronization at the pixel level and per-thread dependencies, the performance improvement was moderate.
Debugging was moderately difficult compared to the sequential version but easier than the multi-process implementations.

2.Multi-process communication via pipes:

To parallelize the image sharpening process by assigning each stage (S1, S2, and S3) to separate processes that communicate using **UNIX pipes**.
Each stage runs concurrently on a different CPU core, passing pixel data in order through the pipeline.

A **pipe** is a **one-way communication channel** between processes. Whatever one process **writes** into the pipe, the other process can **read** from it (in the same order).

Pixel order correctness:



```
jaswika@Sona:/mnt/c/Users/jaswi/OneDrive/Desktop/5th sem/os_lab/lab$ diff images/1_output_part1.ppm images/1_output_part2B.ppm
jaswika@Sona:/mnt/c/Users/jaswi/OneDrive/Desktop/5th sem/os_lab/lab$ 
```

SPEED-UP:

The speed-up is limited because **file reading and writing** dominate total execution time.
However, when the experiment was modified to run each transformation **1000 times** (making it compute-intensive), the multi-process version achieved noticeable overlap across stages.

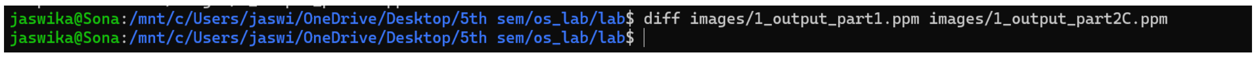
Input image

Sharpened image using pipes:

CONCLUSION:
The pipe-based multi-process approach successfully parallelized the three stages of the image sharpening task.
It allowed concurrent execution on different CPU cores while maintaining data integrity through FIFO pipes.
Though speed-up was modest due to I/O overhead, the design proved effective for stream-based parallel processing.

3.Multi-process communicating via shared memory and synchronisation using semaphores
To parallelize the image-sharpening application by using three independent processes for stages S1, S2, and S3, which communicate through a shared memory segment and are synchronized using semaphores to ensure correct ordering and safe access to data.

pixel-correctness:



```
jaswika@Sona:/mnt/c/Users/jaswi/OneDrive/Desktop/5th sem/os_lab/lab$ diff images/1_output_part1.ppm images/1_output_part2C.ppm
jaswika@Sona:/mnt/c/Users/jaswi/OneDrive/Desktop/5th sem/os_lab/lab$
```

SPEED-UP:
This method performed **better than the pipe version** because data transfer occurs in shared memory (no kernel copying).
However, synchronization using semaphores adds some overhead, and performance improvement still depends on the compute-to-I/O ratio.
Advantages:

1. Shared memory allows direct access-no kernel-space copying like in pipes.
2. Data is exchanged without serialization or intermediate buffers.
3. Semaphores give precise control over when each process can read/write.
4. Efficient for image or matrix operations where large chunks of data are shared.

Difficulties:

1. Managing multiple semaphores correctly is tricky and prone to deadlocks.
2. 3.Errors in semaphore usage can cause processes to block indefinitely.
3. If the program crashes, semaphores and shared memory must be manually removed.

4. Works only on systems supporting POSIX/UNIX shared memory; not trivial on Windows.

Input image:

sharpened image after using semaphores:

PART-3:

In this part, we extend the pipeline across **two computers** using **socket communication**. Each computer handles a subset of the three stages (S1, S2, and S3). The communication between the computers happens over a TCP connection identified by an IP address and a port number.

Configuration A: S1 and S2 on Computer A, S3 on Computer B
->Computer A performs Stage 1 (S1) and Stage 2 (S2) sequentially
->Once S2 finishes processing, its output is sent to Computer B via sockets
->Computer B performs Stage 3 (S3) and writes the final image to disk.

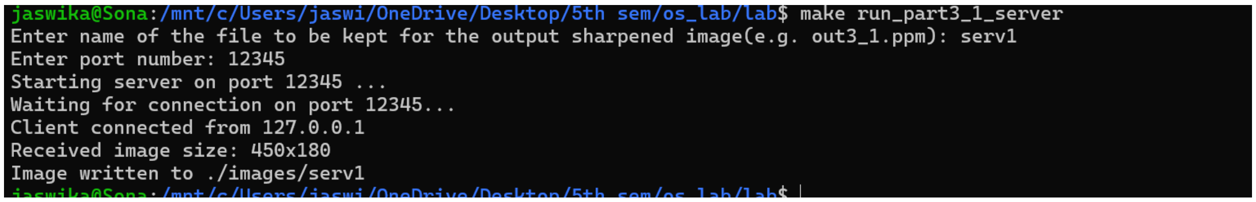
Observations:
->This configuration reduces the data transmission overhead because
->intermediate data (after S2) is smaller than raw image data.
Computer A performs most of the computation, which may cause imbalance if S1 and S2 are computationally heavy.
->S3 executes independently and completes faster once data is received.

Configuration B: S1 on Computer A, S2 and S3 on Computer B
->Computer A performs Stage 1 (S1) and sends the processed image data directly to Computer B using sockets
->Computer B performs Stages 2 (S2) and 3 (S3) in sequence and writes the final result to disk.

Observations:
->This configuration balances the computational load better between the two systems.
->Since S1 completes quickly, Computer A can act as a data source, while Computer B performs the heavier computations (S2 + S3).
->However, transmitting the entire S1 output (the smoothened image) over the network incurs more data transfer time compared to Configuration A.

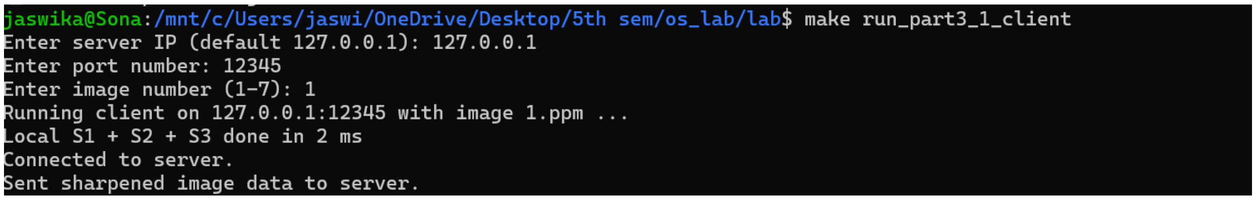
Final Conclusion:
Configuration B (S1->S2+S3) achieved better performance due to balanced workload, despite higher data transfer.

Server-1:



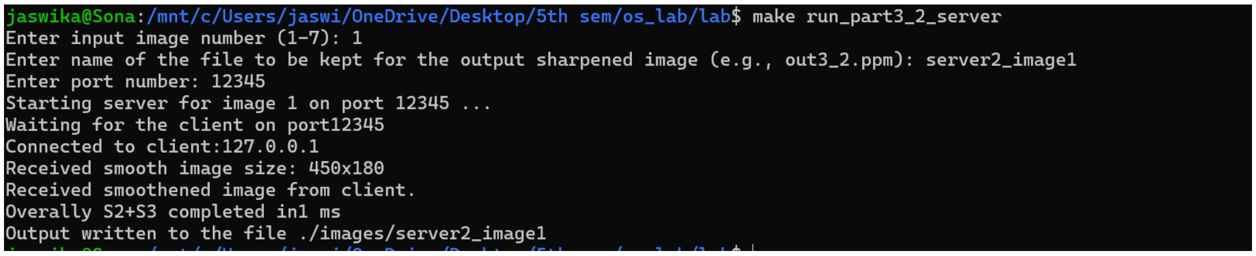
```
jaswika@Sona:/mnt/c/Users/jaswi/OneDrive/Desktop/5th sem/os_lab/lab$ make run_part3_1_server
Enter name of the file to be kept for the output sharpened image(e.g. out3_1.ppm): serv1
Enter port number: 12345
Starting server on port 12345 ...
Waiting for connection on port 12345...
Client connected from 127.0.0.1
Received image size: 450x180
Image written to ./images/serv1
jaswika@Sona:/mnt/c/Users/jaswi/OneDrive/Desktop/5th sem/os_lab/lab$
```

Client-1:



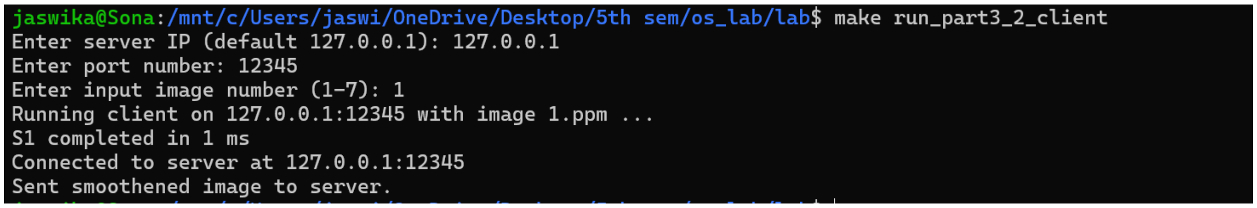
```
jaswika@Sona:/mnt/c/Users/jaswi/OneDrive/Desktop/5th sem/os_lab/lab$ make run_part3_1_client
Enter server IP (default 127.0.0.1): 127.0.0.1
Enter port number: 12345
Enter image number (1-7): 1
Running client on 127.0.0.1:12345 with image 1.ppm ...
Local S1 + S2 + S3 done in 2 ms
Connected to server.
Sent sharpened image data to server.
```

Server-2:



```
jaswika@Sona:/mnt/c/Users/jaswi/OneDrive/Desktop/5th sem/os_lab/lab$ make run_part3_2_server
Enter input image number (1-7): 1
Enter name of the file to be kept for the output sharpened image (e.g., out3_2.ppm): server2_image1
Enter port number: 12345
Starting server for image 1 on port 12345 ...
Waiting for the client on port12345
Connected to client:127.0.0.1
Received smooth image size: 450x180
Received smoothened image from client.
Overally S2+S3 completed in1 ms
Output written to the file ./images/server2_image1
```

Client-2:



```
jaswika@Sona:/mnt/c/Users/jaswi/OneDrive/Desktop/5th sem/os_lab/lab$ make run_part3_2_client
Enter server IP (default 127.0.0.1): 127.0.0.1
Enter port number: 12345
Enter input image number (1-7): 1
Running client on 127.0.0.1:12345 with image 1.ppm ...
S1 completed in 1 ms
Connected to server at 127.0.0.1:12345
Sent smoothened image to server.
```