1) Discuss the race conditions of your solution, and where did you need to use locks to avoid them.

The main shared resource in the program is the global histogram array. If all threads directly executed “global_histogram[pixel]++” then two threads could read the same value, increment the value and return it losing one increment and producing incorrect histogram values.
But since we use local histograms each thread creates its own private array.

The other race condition could happen if all threads shared the same file descriptor, since the file pointer would be shared and read() operations would interfere but since we open the file independently inside each thread seeking the assigned offset we ensure the right parts of the file are readed by each thread.

The only moment where a lock is needed is where shared memory is modified, in the merge step. The mutex ensures that only one thread at a time updates the global histogram, so no simultaneous writes occur and therefore no lost updates happen.

2) In your solution, how many threads can simultaneously read from the file? And update the histogram?

All threads can read simultaneously because:

- Each thread opens the file independently.
- Each thread has its own file descriptor.
- Each thread seeks to a different offset.

The operating system allows concurrent reads.

During pixel processing:

- All threads update their local histograms simultaneously.
- No locks are required.

During merging:

- Only one thread at a time holds the mutex.
- Other threads must wait briefly.

3) Report the needed execution time as a function of the number of threads. Comment your results.

Execution time decreases as the number of threads increases, speedup being nearly linear until reaching the number of physical CPU cores. After that, performance improvement saturates.

Based on the time command output, the "real" (wall-clock) execution time for processing heart.pgm is as follows:

| **Number of Threads** | **Execution Time (Seconds)** |
|---|---|
| 1 Thread | 4.151s |
| 2 Threads | 3.205s |
| 3 Threads | 2.311s |
| 4 Threads | 2.192s |
| **5 Threads** | **1.941s (Fastest)** |
| 6 Threads | 2.074s |
| 7 Threads | 2.033s |
| 8 Threads | 2.165s |

The program gets much faster as we increase from 1 to 5 threads, cutting the total time by more than half. This proves that splitting the work among multiple helpers speeds things up. However, adding more than 5 threads actually makes it slightly slower again. This happens because the computer starts spending too much effort just organizing all those extra threads, and they end up creating a traffic jam trying to read from the hard drive all at the exact same time.