Benedikt Heuser
Atharva Gangal

Part 1

The non-optimized version of the code produced the following run times for 100M samples and 100 buckets:

threads	time
1	0.6568s
2	0.6575s
4	0.676s
8	0.6564s

Table 1: Non-optimized

The run time increases with the number of buckets likely due to a rise in the number of cache misses with larger histograms. We optimized the following aspects of the code:

1. We reduced true sharing of *histogram* by using a two dimensional array and a second merge loop as seen in the lecture.
2. We privatized the random number generator to reduce true sharing.
3. We padded our integers to reduce false sharing as seen in the lecture with a struct *PadInt*

threads	time
1	0.4159s
2	0.2282s
4	0.1464s
8	0.1516s

Table 2: Optimized

With this implementation the run times are significantly lower for two, four and eight threads.

Part 2

2.1

While we tried out multiple methods to optimize memory sharing and performance, based on our testing, the lowest execution time was obtained on simple parallelization of the outer loop (with variable i). Hence, the only optimization we applied was a simple omp parallel for loop. We present all the optimizations we tried, with the results, in the next part of the question.
The data can be split into various ways to preserve locality, for example row-wise or column-wise. However, it is best to follow a 'tiling' based approach, as it makes the most of the data brought into the cache on the first (compulsory) cache miss. Making tiles based on cache line size, and splitting the tiles equally among threads is most effective.
There is possibility of unequal load if the tiles are not able to split evenly amongst the threads due to sizing constraints.

2.2

Method tried	Execution Time	Potential Reason for behaviour
Simple parallelization with omp parallel for, split by rows	6.868s	Trivial improvement over single threaded implementation, baseline for comparison
Tiling approach: split grid into 8x8 tiles, distribute evenly for locality	7.929s	Internal for-loops adds overhead per iteration
Thread creation optimization: Create parallel section outside loop and put barrier for data correctness	6.867s	Barrier adds timing delays
Loop unrolling: Unroll loop along columns and perform consecutive row operations in same iteration	7.871s	For-loop overhead negligible compared to loop body
Dynamic scheduling: Add dynamic scheduling to potentially improve thread performance	7.204s	Overhead of dynamic scheduling higher compared to benefits of potential load-balancing, if any

Table 3: Brief summary of optimizations

Execution times above are for 10000 size, 100 iterations and 4 threads. Additionally, there was little scope for true/false sharing or loop fission/fusion improvements.

2.3