Assignment 2 Group 11

Part 1

Parallelization

In the naive parallelization, we parallelize the sampling for loop and use a critical section to protect the increment of buckets in histogram. From the result, the execution time significantly increases with the number of threads, due to true and false sharing. True sharing happens when two threads update the same bucket, and false sharing happens when two threads update different buckets, but residing in the same cache block which gets invalidated regardless.

Optimization

To resolve true sharing, we let each thread update their own temporary histogram array and merge the results later. We created a 2D temporary histogram array, using thread ID to index rows and number of bucket to index columns. Each thread modifies a different row, eliminating true sharing.

When adding results up, we divide the buckets among threads. Each thread updates a different bucket and adds up results from all threads. This eliminates true sharing, and thus critical sections, completely.

To deal with the false sharing problem, we created a struct called PaddingHist. It contains a counter and an array of 15 integers, making a total size of 64 bytes that allows it to fit an entire cache block.

Performance

# of Threads	Naive (seconds)	Optimized (seconds)
1	1.80	0.67
2	9.30	0.34
4	25.52	0.18
8	31.06	0.09

If histogram size means the number of buckets, then increasing the number of buckets in the unoptimized algorithm would result in lower rate of both true sharing and false sharing, because it would be less likely for two threads to update the same bucket or different buckets in the same cache block. But in practice, the performance stays the same. This is because the critical section is more coarse grained and only allows one thread to update any bucket at a given time.

If histogram size means the number of samples, then execution time increases linearly with the number of samples.

Part 2

1.

Parallelization

We parallelized the for loop iterating the rows. This simple parallelization turns out to be optimal, implicitly containing efficient memory access patterns.

Out of the three for loops, we cannot parallelize the iteration one, because the next iteration would depend on results from the previous iteration (According to suggestion from TA this is possible, and would theoretically reduce the number of fork/joins and improve performance, but might be optimized already by modern compilers). Parallelizing the loop over columns would not be efficient either, because it breaks locality for cache access, and would result in more cache misses and harm performance.

Memory optimizations

Running cachegrind on the program indicates a cache miss rate of only 2.9%(overall, excluding writing to file) or 5%(array read/write) in single thread scenario. It leaves little room for memory optimizations for this algorithm.

Blocking for locality does not really apply to this program, because there are no pure vertical access patterns (with sensible parallelization) like in matrix multiplications. Loop unrolling does not yield observable performance improvement either, probably due to the low cache miss rate and compiler optimizations.

Following another suggestion from TA, we also tried removing the if statement that decides if a point is in the center, and re-assigning the center values later. This saves a lot of if statements but may add a few cache misses per iteration. This did not contribute to any observable speedup either, probably due to very low miss rate of branch prediction.

Therefore, we did not include any memory optimization in the final algorithm. We focus instead on different scheduling methods.

Scheduling

Optimal performance is achieved when we don't specify the scheduling method. Judging by execution time, the default scheduling method seems to be static scheduling, despite gcc documentation says otherwise. We sometimes observe faster execution compared to specifying static scheduling for unknown reasons.

Static scheduling not only has the least scheduling overhead, but also happens to access memory in a way that results in optimal locality. In static scheduling, each thread gets assigned a consecutive chunk of the loop, which by default is of the size ceil(#iter / #threads). In this algorithm, computations need to access neighboring rows. It is thus very efficient for threads to work on a chunk of consecutive rows because neighboring rows will more likely be kept in cache for faster access. If we use dynamic scheduling that allocates one iteration at a time, for example, each thread would need to fetch three rows, which might not be in cache, for computation of just one row. Guided scheduling has less overhead than dynamic scheduling, but is still slightly slower than static scheduling.

We did not identify significant work imbalance among threads, even with static scheduling. We measured execution time of each thread, and found that each thread finished work at roughly the same time.

2. As we focus mainly on scheduling methods, we try to isolate the effects of cache locality. Results are presented in q3.

With static scheduling, each thread gets assigned a chunk of size ceil(#iter / #threads) by default. This happens to result in continuous chunks and thus optical locality. We try to eliminate this locality by specifying a chunk size of 1. With 16 threads, for example, thread 0 would get row 0, 16, 32… and thread 1 would get row 1, 17, 33… and so on. We can see locality contributes to a speedup of a little over one second..

We can also examine the cost of scheduling overhead by comparing (static,1) with dynamic scheduling, which by default also assigns 1 chunk at a time. We can see that the execution time is longer by another second or so.

Guided scheduling also assigns chunks of size (#iter / #threads) at first, and thus we can see very close performance to static scheduling. The execution time is only a bit longer when using 16 threads, possibly due to the overhead of assigning the remaining chunks.

3.

#Threads	Baseline	Static	Static,1	Dynamic	Guided
1	25.47	25.36	25.40	25.29	25.35
2	-	12.94	14.60	15.74	13.12
4	-	6.68	8.14	8.82	6.64
8	-	3.58	4.75	5.31	3.57
16	-	2.45	3.75	4.62	3.02