Reference solution to the 2024-2025 exam in HPPS

Troels Henriksen (`athas@sigkill.dk`)

January, 2025

Context

The reference solution code is in the directory `exam-solution`. This document contains reference answers to the questions posed in section 3 of the exam text *relative to the reference solution code*. It is possible that a student submits differing-but-correct code, and hence also provides differing-but-correct answers. However, given the quite fixed task, it is unlikely that any major divergence is going to be correct. The text contains **correction notes** in footnotes as guidelines to graders and future students.

Introduction

I benchmarked on an Intel Xeon Silver 4114 CPU with 10 cores, 32KiB L1d cache per core, 1MiB L2 cache per core, and 13.8MiB L3 cache.[1] All run-time measurements are by taking the average of ten runs.

I have validated the correctness of my sequential programs by manual inspection on small datasets. Correctness of parallel implementations is by comparison to sequential results. I have verified the absence of memory leaks (or other memory errors) through Address Sanitizer. Based on this,

[1]**Correction note:** Most students will use a laptop CPU, which is generally not a good platform for benchmarking, but we are not grading students on which hardware they have access to.

I believe that my implementation is correct. I do not have any automatic testing.[2]

There is a part of the histograms that I did not fully parallelise, namely in `histogram_parallel_samples()`, where the per-thread local histograms are combined. I assume that in most workloads, n is much greater than k, and so sequential runtime of combining the histograms is immaterial.[3]

1a

It is challenging to allocate memory for a structure whose size we do not know in advance - in this case neither the array of lines, nor the size of each line, is known. I use the `getline()` library function to read a single line, which takes care of allocating an array of sufficient size. For the array of lines, I allocate a small initial array, keep track of its size, then `realloc()` it as necessary to a larger size (by doubling) when necessary.

1b

A file that does not end in a newline character is not a valid text file, and will trigger a failing assertion.[4]

1c

1. File cannot be opened.
2. Memory allocation fails.
3. Reading from the file fails.
4. The file has invalid contents (as above).

[2]**Correction note:** This is acceptable, although risky.

[3]**Correction note:** This is OK, but the student must mention it.

[4]**Correction note:** It is OK if students handle this failure in a less explosive way, but it must be detected and remarked upon here.

Threads	$k = 100$		$k = 1000$		$k = 10000$	
	Speedup		**Speedup**		**Speedup**	
	`bins`	**`samples`**	**`bins`**	**`samples`**	**`bins`**	**`samples`**
1	$0.57\times$	$1.02\times$	$0.56\times$	$1.01\times$	$0.58\times$	$1.00\times$
2	$0.36\times$	$1.61\times$	$0.55\times$	$1.30\times$	$0.61\times$	$1.30\times$
4	$0.35\times$	$1.95\times$	$0.48\times$	$2.20\times$	$0.70\times$	$1.97\times$
8	$0.29\times$	$2.88\times$	$0.62\times$	$2.95\times$	$0.73\times$	$2.82\times$

Table 1: Strong scalability of counting histograms for $n = 10^7$ and different values of k.

Threads	**Speedup**	
	`bins`	`samples`
1	$0.48\times$	$0.58\times$
2	$0.36\times$	$1.16\times$
4	$0.37\times$	$1.50\times$
8	$0.27\times$	$2.77\times$

Table 2: Weak scalability of counting histograms for $n = p \cdot 10^7, k = 100$, where p is the number of threads.

2a

Reading `samples` exhibits good spatial locality, but no temporal locality, as each element is read once in order. The locality for the writes to `H` depends on the input—we can easily imagine a `samples` where all elements are identical, resulting in perfect temporal locality for the updates of `H`, or one where we jump between bins far from each other.

2b

The strong scalability is shown in table 3. I chose a fairly large n to ensure each thread had enough work to do.[5] I use different values of k to see how it influences the performance. We see that `histogram_parallel_bins()` performs particularly poorly. This is because `histogram_parallel_bins()`

[5]**Correction note:** It is fine to pick a workload so large that we will observe good strong scalability–the task here is not to find an adversarial input, just to show that you understand the process.

essentially duplicates the work for each of p threads, meaning the $O(n)$ algorithm becomes $O(p \cdot n)$. Relative performance does improve slightly for larger k, which is possibly because `histogram_parallel_bins()` does not have the sequential bottleneck mentioned in the introduction.

Scalability is overall poor, which is likely because this is a memory-bound problem, and using p threads does not make memory p times faster.

The weak scalability is shown in table 4. For simplicity, I use only a fixed k, but increase n proportionally with the number of threads.

2c

This can happen due to the input-dependent locality discussed in 2a. The fastest input would involve only updating a single bin, and the slowest input would maximise the distance from each update to the next.[6]

3a

Locality is again data-dependent, and in the limit, can be arbitrarily bad. However, distance histograms perform $O(n^2)$ work given an input of size $O(n \cdot k)$, so assuming that k is much smaller than n, distance histograms will generally exhibit better locality, simply by having a smaller working set.

3b

I used the same approach as `histogram_parallel_samples()`, including the sequential part, as the results above show that it clearly performs the best.

3c

Strong scaling is shown on table 3. I used a fairly large n to provide enough work for all threads. Scalability is much better than with counting his-

[6]**Correction note:** It is OK to show this experimentally, but the question does not ask for it. A clear answer is sufficient.

	$k = 100$	$k = 1000$	$k = 10000$
Threads	**Speedup**	**Speedup**	**Speedup**
1	$0.98\times$	$1.00\times$	$1.01\times$
2	$1.86\times$	$1.76\times$	$1.80\times$
4	$3.48\times$	$3.32\times$	$3.46\times$
8	$6.44\times$	$6.06\times$	$6.39\times$

Table 3: Strong scalability of distance histograms for $n = 10^5$ and different values of k, with s picked such that all updates in-bounds.

Threads	**Speedup**
1	$0.98\times$
2	$1.71\times$
4	$3.20\times$
8	$5.85\times$

Table 4: Weak scalability of distance histograms for $n = p \cdot 2 \cdot 10^3, k = 100$, where p is the number of threads, and s picked such that all updates are in-bounds

tograms, which is likely because distance histograms are not as effected by memory access speed.

Weak scaling is shown on table 4. Here one challenge is ensuring a fixed workload relative to the number of threads (p), as the computational complexity is $O(n^2)$. My solution is to multiply a base workload with $\sqrt{p}$.[7]

4a

If we allowed the strings in mapping files to contain arbitrary bytes, then they might also contain bytes with the numeric value zero (NUL). As NUL bytes are used by C to indicate end-of-string, this means we would be unable to use standard C string functions to operate on arbitrary byte sequences, and we would instead need to manually track their size, and use more explicit memory manipulation functions.

It would of course also make debugging more difficult when we cannot simply print strings to the console.

[7]**Correction note:** This is the main tricky part, and students must address it.