Reference solution to the 2022-2023 exam in HPPS

Troels Henriksen (athas@sigkill.dk)

January, 2023

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.

Introduction

I benchmarked on a Ryzen 1700X processor with eight physical cores (16 threads, although I don't use them), 256KiB L1d cahce, 4MiB L2 cache, and 16MiB L3 cache.

I use ten runs for every benchmark. Empirically this seems enough to get stable results, without being so high as to be annoying.

I have implemented every function and parallelised as much as I believe reasonable (e.g. I do not parallelise multiple levels of nested loops).[1] I have picked my datasets where there the outer loop(s) contain enough iterations to create sufficient threads for the processor (i.e. 8 in my case).

[1] **Correction note:** this is perfectly fine, especially if they explain why.

The benchmarks are run with `./benchmark.sh`[2]. I benchmark with 1, 2, 4, and 8 threads.[3] The random sparse matrices I use all have a nonuniform nonzero density ranging from 0 to 0.1. I have not systematically investigated the impact of varying densities. [4] I have picked data sizes that are large enough to avoid fitting entirely in the cache.

When reporting **weak scaling** I report speedup in throughput over using a single thread, where throughput is computed as the data size divided by the runtime. I increase the data size linearly with the thread count (details in the specific questions).

When reporting **strong scaling** I fix the dataset to the one used in the 4-thread case for weak scaling, and report speedup in latency relative to using a single thread.

Generally speaking, scaling is sublinear. This is to be expected since these problems are memory-bound. Once we add enough threads that we saturate the memory bus, adding more threads gives no more speedup (even if there are still cores idle).

1a

The loop traverses two vectors with a stride of 1, thus exhibiting ideal spatial locality. There is no temporal locality in memory accesses, but various scalar variables are accessed repeatedly.

1b

The iterations of the loop are not fully independent, but I have parallelised the loop as an OpenMP reduction.

[2] **Correction note:** you do not need to look at such auxiliary scripts, unless the reported results are truly mysterious and you think maybe they made measurement errors.

[3] **Correction note:** it's fine that they don't show for all possible thread counts—it's too time consuming.

[4] **Correction note:** we don't expect students to do this either.

1c

Threads	Weak scaling	Strong scaling
1	1.02	1.03
2	1.73	1.93
4	2.33	2.42
8	2.40	2.44

My workload is k-vectors, where $k = t \cdot 8 \cdot 1024^2$ and t is the number of threads (except for weak scaling where $t = 4$, as noted above). I will use this notation and convention in the remaining sections as well. The weak and strong scaling is identical, which probably reflects that at $t = 4$ the memory bus is fully utilised.

2a

My implementation traverses the array row-by-row, which has good spatial locality. Further, the vector is repeatedly traversed for every row of the array, which is an example of temporal locality. Whether this temporal locality is advantageous depends on the size of the vector—if it is too large, the initial elements will be evicted from the cache by the time we get around to traversing it again.

2b

Every iteration of the outer loop is independent, so I parallelised it with OpenMP. The inner loop could be parallelised as an OpenMP reduction, but in most cases the outer loop will have a sufficient number of iterations.

2c

Threads	Weak scaling	Strong scaling
1	1.00	1.00
2	1.97	1.99
4	3.77	3.74
8	4.58	4.41

My workload is $n \times m$-matrices multiplied with m-vectors, where $n = 8096, m = t \cdot 1024$. The weak and strong scaling is identical, which probably reflects that at $t = 4$ the memory bus is fully utilised. The speedup is better than for `mul-vv`, which perhaps reflects the impact of temporal locality in the vector accesses, as the vector is small enough to fit in L3 cache.[5]

3a

I have carefully written this function to ensure that the array input array and output vector is accessed with a stride of 1, yielding ideal spatial locality. The input vector has the same element accessed repeatedly in the inner loop, providing temporal locality. A more immediate way of writing this loop would traverse the matrix with a stride of m, which would be very poor spatial locality.[6]

3b

The iterations of the outer loop are not independent as they all modify the output vector. However, the inner loop can be parallelised. The two loops can be transposed, but then locality would suffer. Here we have a tradeoff between parallel grain size and spatial locality[7].

3c

Threads	Weak scaling	Strong scaling
1	1.00	0.95
2	0.80	0.47
4	1.21	0.39
8	1.64	0.29

[5] **Correction note:**: it is enough for students to provide plausible hypotheses—we do not expect them to investigate in detail.

[6] **Correction note:** it is OK (but not perfect) for students to write a cache-inefficient implementation *if* they recognise that it is indeed inefficient.

[7] **Correction note:** it is possible that some students figure out a solution that solves both these problems, but it requires tricks that lie beyond what we have taught them, and so we do not expect it.

My workload is $n \times m$-matrices multiplied with n-vectors, where $n = 8096, m = t \cdot 1024$. The speedup is bad (particularly for strong scaling), which is probably because only the inner loop is parallelised, which has much less work per iteration. This results in the overhead of parallelisation outweighing the advantage of parallel execution.

4a

The row_{start} array is accessed with a stride of 1 (and only in the outer loop, which is less important). The v and v_{col} arrays are accessed with a stride of 1 in the innermost loop. However, the input vector is accessed with an unpredictable stride, as elements from the v_{col} array determine which element of the vector is read. It is likely that the accesses exhibit neither spatial or temporal locality, dependent on the precise pattern of sparsity of the matrix.

4b

The iterations of the outer loop are independent (as with `mul-mv`), so I parallelised them with OpenMP.

4c

Threads	**Weak scaling**	**Strong scaling**
1	1.00	1.00
2	1.21	1.30
4	1.92	2.04
8	2.55	2.53

My workload is $n \times m$-matrices multiplied with m-vectors, where $n = 32 \cdot 1024, m = t \cdot 4 \cdot 1024$, with a sparsity ranging from 0 to 0.1.

5a

In the innermost loop, the v and v_{col} arrays are accessed with a stride of 1 (good spatial locality) and an element of the input vector is accessed

repeatedly (good temporal locality). However, the output vector is accessed unpredictably based on the values of the v_{col} array, similar to the input vector for `mul-mv`. This is likely to result in poor locality.

5b

Similar to `mul-mTv`, the iterations of the outer loop are not independent, so I parallelised only the innermost loop.

5c

Threads	**Weak scaling**	**Strong scaling**
1	1.00	1.00
2	0.83	0.50
4	1.50	0.48
8	1.92	0.35

My workload is $n \times m$-matrices multiplied with n-vectors, where $n = 32 \cdot 1024, m = t \cdot 4 \cdot 1024$, with a sparsity ranging from 0 to 0.1. As with `mul-mTv`, the speedup is bad, and probably for the same reason: the parallelism is too fine-grained.