Reference solution to the 2021-2022 exam in HPPS

January, 2022

Context

The reference solution code is in the directory `exam-solution`. This document contains reference answers to the questions posed in section 2 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.

Introduction

I benchmarked on a Ryzen 1700X processor with eight physical cores (16 threads), 256KiB L1d cahce, 4MiB L2 cache, and 16MiB L3 cache.

I generally picked the number of runs to be high for small datasets (e.g. 100 runs to transpose 1024x1024 matrices) and low for large datasets (e.g. 10 runs to multiply 1024x1024 matrices).

I have implemented every function and parallelised as much as I believe reasonable (but `matmul_parallel` is not as parallel as it could be).

The benchmarks are run with `./benchmark`. More complicated benchmarking (e.g. changing `T` or showing scalability) may need code modification, recompilation, or multiple runs changing `OMP_NUM_THREADS` as applicable. Unless otherwise indicated, the parallel versions are run with the maximum number of threads (16).

a)

Using `T=8`.

$n, m =$	1024	2048	4096
`transpose`	$9ms$	$28ms$	$257ms$
`transpose_blocked`	$2ms$	$11ms$	$54ms$
`transpose_parallel`	$2ms$	$14ms$	$73ms$
`transpose_blocked_parallel`	$1ms$	$6ms$	$23ms$

b)

Note that testing different values of `T` requires recompiling everything. All times are in miliseconds.

$n, m =$	1024	2048	4096
`T=1`	$8ms$	$34ms$	$260ms$
`T=2`	$4ms$	$17ms$	$84ms$
`T=4`	$2ms$	$14ms$	$71ms$
`T=8`	$2ms$	$11ms$	$56ms$
`T=16`	$3ms$	$13ms$	$68ms$
`T=32`	$6ms$	$23ms$	$985ms$

Based on these results I picked `T=8` as the best value. I assume (but did not test) this will also be best for `transpose_blocked_parallel`. (**Correction note:** it probably is not, but not checking is a minor flaw.)

The normal `matmul` function has bad locality because the output matrix is accessed column-wise. Instead, `transpose_blocked` accesses memory in chunks corresponding to `T` by `T` blocks of the overall matrix. This means we get better spatial locality, as there is a good chance that when writing the second column of a row of the output, that part is still in cache.

Correction note We don't expect students to do the following level of analysis.

With `T=8`, transposing such a block involves accessing

$$2T^2 \cdot \texttt{sizeof(double)} = 1024B = 1KiB$$

(the 2 factor is because we are both reading and writing such a block). $1KiB$ clearly fits in L1 cache, but so does the $2KiB$ required for `T=16`, so why is `T=18` faster? Perhaps because the L1 cache on this CPU is 8-way set associative, so we get evictions once we need more than 8 lines at a time.

c)

I compute the speedup as the runtime of `transpose_blocked` divided by the runtime of `transpose_blocked_parallel`

First we transpose a large 4096×4096 array (note that this takes $128 MiB$ and so does not fit entirely in cache). This is because it will likely provide plenty of work per thread, even when running with the highest number of threads.

Threads	**Sequential**	**Parallel**	**Speedup**
1	$55ms$	$55ms$	$1.00\times$
2	$55ms$	$39ms$	$1.41\times$
3	$55ms$	$31ms$	$1.77\times$
4	$55ms$	$27ms$	$2.03\times$
5	$55ms$	$25ms$	$2.20\times$
6	$55ms$	$24ms$	$2.29\times$
7	$55ms$	$23ms$	$2.39\times$
8	$55ms$	23	$2.39\times$

We conclude that `transpose_blocked_parallel` does *not* show strong scaling, as the speedup plateaus close to $2.4\times$ (and requires 7 threads to even obtain that).

Now let us scale the work such that we multiply matrices of size $1024 \times t \cdot 1024$, where t is the number of threads used in the parallel case. This keeps the amount of work constant relative to the number of threads.

Threads	**Sequential**	**Parallel**	**Speedup**
1	$2ms$	$2ms$	$1.00\times$
2	$4ms$	$3ms$	$1.33\times$
3	$7ms$	$4ms$	$1.75\times$
4	$9ms$	$4ms$	$2.25\times$
5	$11ms$	$6ms$	$2.17\times$
6	$13ms$	$6ms$	$2.14\times$
7	$15ms$	$7ms$	$2.5\times$
8	$18ms$	$9ms$	$2.00\times$

Interestingly, weak scaling (on these workloads) is even worse than strong scaling; perhaps because of the lower width of the matrix.

d)

$n, m, k =$	256	512	1024
`matmul`	$27ms$	$844ms$	$6903ms$
`matmul_parallel`	$3ms$	$79ms$	$590ms$
`matmul_locality`	$6ms$	$42ms$	$422ms$
`matmul_transpose`	$12ms$	$106ms$	$905ms$
`matmul_locality_parallel`	$1ms$	$5ms$	$44ms$
`matmul_transpose_parallel`	$1ms$	$8ms$	$60ms$

e)

The `matmul` function has poor spatial locality, as we access array `B` with a stride of `k`. This means we likely have many cache misses. In contrast, `matmul_locality` accesses all arrays with unit stride, ensuring perfect spatial locality. Finally, `matmul_transpose` also accesses all arrays with unit stride, made possible by first transposing `B`. Since transposition is asymptotically (and in practice) much faster than matrix multiplication, this preprocessing does not add much to the runtime, but allows a more efficient subsequent memory access pattern.

f)

The raw numbers are in the answer to question (d); here are the computed speedups:

$n, m, k =$	256	512	1024
`matmul_parallel`	8.11	9.3	8.55
`matmul_locality_parallel`	6	7.0	9.5
`matmul_transpose_parallel`	6.0	13.25	15.08

g)

For `matmul_parallel` and `matmul_transpose_parallel`, I decided to parallelise only the outer two loops with `omp pragma parallel for collapse(2)`. This is because the inner loop has a dependency on the accumulator, and while it can be parallelised using a `reduce` clause, it is likely not worth the overhead—the two outer loops provide sufficient iterations for most practical workloads and machines.

I use OpenMP's default static scheduling, because the different iterations should are naturally load-balanced.

h)

`matmul_locality_parallel()` only parallelises the outer (`i`) loop. This is because different iterations of the `j` loop write to the same `i` row of the output matrix, and hence the iterations of the `j` loop are not independent. Hence `matmul_locality_parallel()` is less parallel than `matmul_transpose_parallel`.

For most workloads, the number of iterations in the outer loop (`n`) is going to exceed the number of cores in the machine, and so this difference will not matter.

i)

With t being the number of threads, I am benchmarking with $n = i \cdot 256, m = 1024, k = 1024$. This means the workload scales linearly with the amount of threads, and hence shows *weak scaling.* It also means that the working set does not fit in cache as soon as $t > 1$. I am comparing `matmul_locality_parallel` and `matmul_locality`, average of 10 runs.

Threads	**Sequential**	**Parallel**	**Speedup**
1	$100ms$	$100ms$	1.00×
2	$195ms$	$104ms$	1.88×
3	$295ms$	$105ms$	2.80×
4	$387ms$	$107ms$	3.62×
5	$480ms$	$110ms$	4.36×
6	$589ms$	$112ms$	5.25×
7	$686ms$	$116ms$	5.91×
8	$769ms$	$118ms$	6.52×
12	$1169ms$	$159ms$	7.35×
16	$1536ms$	$176ms$	8.72×

The weak scaling is excellent, with runtime remaining near-constant up to 8 threads even as the workload increases. As we see, this stops after reaching the physical core count (8) of the processor.

To investigate strong scaling, we benchmark with $n = 1024, m = 1024, k = 1024$; the workload for 4 threads above:

Threads	**Sequential**	**Parallel**	**Speedup**
1	$385ms$	$404ms$	$0.95\times$
4	$389ms$	$104ms$	$3.74\times$
8	$376ms$	$58ms$	$6.48\times$
16	$385ms$	$47ms$	$8.19\times$

For this workload, strong scaling remains quite good up to 8 threads, but diminishes significantly after that, lagging slightly behind the speedup achieved when measuring weak scaling.