

Reference solution to the 2025-2026 exam in HPPS

Troels Henriksen (athas@sigkill.dk)

January, 2026

Context

The reference solution code is in the directory `exam-solution`. This document contains reference answers to the questions posed in 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 AMD Ryzen 9950X CPU with 16 cores, 48KiB L1d cache per core, 1MiB L2 cache per core, and 64MiB L3 cache.¹ When benchmarking, I perform only a single run for every workload, but I take care to ensure that the workloads take a reasonably long time to run, in order to minimise the impact of noise.² The benchmarking is automated in the script `benchmark.sh`.³

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.

²**Correction note:** this is acceptable; we do not require statistical analysis of results.

³**Correction note:** you do not need to look at these scripts unless something looks seriously mysterious in the results.

I have validated the correctness of parallel implementations by comparison to results produced by the handed-out reference implementation. This is done by manual inspection, as the parallelisation slightly changes the numerical results (more on that in the questions below).

I have verified the absence of memory leaks (or other memory errors) through Address Sanitizer. Based on this, I believe that my implementation is correct. I do not have any automatic testing.⁴

1a

Instead of a single random number generation state used across all pixels, I initialise a distinct random number generator for each pixel, using a different seed. This does change the behaviour of the program, as now different random numbers are generated. Since the random numbers picked in the original program were also essentially arbitrary, there is no reason to think this makes the rendered images worse.⁵

I also interchanged the loops to put the outermost sampling loop innermost, to move sequential work inside the threads.

After this, I parallelised the resulting loop nest with `#pragma omp parallel for collapse(2)`.

1b

My workload is generating a 100×100 image of the `rgbbox` scene with 10 samples per pixel. This workload is chosen because it is big enough for each thread to have a substantial amount of work, yet small enough that with enough threads, it might not show perfect strong scaling. I use the `rgbbox` scene because every pixel has roughly the same amount of work. I measure only the rendering time reported by the program, not startup or IO.⁶

When benchmarking weak scaling, I compute speedup against an initial single-threaded run.

⁴**Correction note:** This is acceptable, although risky.

⁵**Correction note:** partial credit if they do not make this observation.

⁶**Correction note:** it is tolerable if they measure full program run-time, but they must make it explicit what they measure.

When benchmarking weak scaling, I multiply the height of the image by the thread count, and compare speedup against a single-threaded run doing the same work.

The results are seen below. There seems to be no significant difference.⁷

Threads	Weak scaling		Strong scaling	
	Runtime	Speedup	Runtime	Speedup
1	0.858s	1.0	0.857s	1.01
2	0.439s	1.95	0.840s	1.95
4	0.233s	3.67	0.846s	3.81
8	0.121s	7.09	0.938s	6.81
16	0.070s	12.25	1.076s	12.01

1c

Since the program repeatedly iterates sequentially across all objects in the scene, it exhibits good spatial locality *if* said objects are stored adjacent in memory, which is somewhat likely.⁸

1d

My parallelisation statically divides the loop iterations (i.e., regions of the image) among the used threads. It is possible that some regions finish before others, resulting in some threads idling while others work—this is an example of poor load balancing.⁹ One solution is to switch OpenMP to *dynamic* scheduling instead of static.

2a

My implementation stores first the `lookfrom` and `lookat` vectors, then the number of materials, spheres, xy/xz/yz rectangles. After this, the the actual

⁷**Correction note:** it is of course possible to construct workloads with poor strong scaling, but this is not required—all we need is that the students demonstrate how to measure these things.

⁸**Correction note:** partial credit if they do not make this assumption clear.

⁹**Correction note:** it is important that the students use this term or an equivalent one.

materials are stored, followed by the spheres, xy-rectangles, xz-rectangles, and yz-rectangles. Each object is stored in a binary format, corresponding to their in-memory representation. The exception is the `struct material` pointer for objects, which is instead stored as an index into the sequence of materials in the file. Because I know how many of each object is expected, I do not need to store the type tags. Further, I deduplicate materials when storing the file, such that a given material is stored only once, even if it is referenced by multiple objects. This means the size of the scene file is proportional to the memory usage of the scene.

2b

Due to the deduplication, the ordering of materials is determined by the order in which they are used by the objects. Since all objects are stored along others of their type, the order of objects is not maintained.

3a

My approach is the same as in question 1b, except that I use a 500×500 image as the starting point. I measure the sum of BVH construction time and rendering time.¹⁰ There is no significant difference in the scalability compared to that which was observed in 1b, except that perhaps the strong scaling is slightly worse, my measurements are not statistically robust enough to rule that out as a fluke.

Threads	Weak scaling		Strong scaling	
	Runtime	Speedup	Runtime	Speedup
1	3.677s	0.99	3.666s	1.00
2	1.867s	1.95	3.231s	1.96
4	1.023s	3.56	3.026s	3.79
8	0.542s	6.71	3.325s	6.56
16	0.293s	12.42	3.889s	11.02

¹⁰**Correction note:** this is important!

3b

It is expected that `ray_bvh` exhibits worse spatial locality than `ray`, as a tree structure has worse intrinsic locality, and the whole point of the traversal is to skip over objects. Note that the amount of memory we access is likely to be lower, it just happens to be more widely scattered.¹¹

3c

It is unlikely to find a scene where *rendering* is slower for `ray_bvh` than for `ray`. However, when also taking the cost to construct the BVH into account, it is possible to construct (contrived) scenes where the cost to construct the BVH exceeds the rendering—for example, rendering a 1×1 image of `rgbbox` with `ray` takes 90 microseconds on my machine, while `ray_bvh` takes a combined 132 microseconds to construct the BVH and perform the rendering.

3d

Constructing the BVH is not a single parallel loop, but a recursive divide-and-conquer algorithm that is awkward to parallelise with OpenMP, because it involves nested parallelism. Consequently I have not parallelised it in my implementation. However, here is a procedure for doing so:

At the top level, after sorting the objects, instead of splitting them in two parts, split them in t parts of equal size, where t is the number of threads, and let each thread sequentially construct a sub-BVH per part. Finally, sequentially stitch together these sub-BVHs using approximately t additional nodes.

3e

For most reasonable scenes, it can be expected that construction of the BVH takes up *much* less time than the subsequent rendering, and so by Amdahl’s Law, the potential gain of parallelising it is minor.

¹¹**Correction note:** the important thing here is that “good spatial locality” is not equivalent to “good performance” in all cases.