Reference solution to the 2023-2024 exam in HPPS

Troels Henriksen (athas@sigkill.dk)

January, 2024

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 7 PRO 4750U processor with eight physical cores (16 threads, although I don't use them), 256KiB L1d cahce, 4MiB L2 cache, and 8MiB L3 cache.[1]

When benchmarking the N-body simulations, I use ten time steps. Empirically this seems enough to get stable results, without being so slow 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).[2]

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

The benchmarks are run with `./benchmark.sh`[3]. I benchmark with 1, 2, 4, and 8 threads.[4]

When reporting **weak scaling** for `nbody` I report speedup in throughput over using a single thread, where throughput is computed as the work size divided by the runtime. Since the work for a given n is n^2, I use $10000\sqrt{t}$ particles when benchmarking with t threads. This ensures that the amount of work increases linearily with the number of threads.

When reporting **strong scaling** I fix the dataset to the one used in the 1-thread case for weak scaling, and report speedup in latency relative to using a single thread.

Generally speaking, scaling is linear but not perfect. This is slightly surprising, since for 10000 particles the `nbody` working set is approximately

$$10000 \cdot \cdot 7 \cdot 8B = 0.53MiB$$

which easily fits in the L2 cache, and so this program is not memory bound.

1a

The following things can go wrong:[5]

- The input file cannot be opened.
- Memory cannot be allocated.
- A read from the file may fail.

1b

The following things can go wrong:

[3] **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.

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

[5] **Correction note:** students may also mention things that are a violation of the function preconditions (e.g. passing invalid pointers) or exotic things like hardware failure or the computer losing power. We don't penalise them for this.

- The output file cannot be opened.
- A write to the file may fail.

1c

My implementation reads the number of particles n from the file, allocates an array of n `struct particle` objects (each $7 \cdot 8 = 54$ bytes), then reads each of the n particles incrementally. The peak memory usage is thus $54 \cdot n$ bytes.[6]

2a

The function exhibits temporal locality in that it repeatedly traverses all particles. If n is not too large (which it isn't for my datasets), this makes it likely that all particles remain resident in the cache.[7]

The function also exhibits good spatial locality when traversing the particles, as these are iterated in serial order.

However, the locality is not perfect, as we do not use all elements of the particles in the innermost loop. In particular, we do not use the velocity component, meaning there is a 24-byte stride between the components we do use (the position and mass).

2b

The outermost loop is a sequential time step loop. It cannot be parallelised.

The two middle loops have completely independent iterations and have been parallelised with OpenMP.

The innermost loop is a kind of reduction, and could be parallelised with some difficulty. However, it is enclosed in a parallel loop with n iterations, and since n is largely (much) larger than the number of cores on the CPU, there would be no advantage to this.

[6] **Correction note:** some students may also talk about memory allocated by the IO subsystem. This is fine, but the question is phrased carefully to not require consideration of this.

[7] **Correction note:** some students may also talk about the locality of scalar variables. This is harmless, but not required.

2c

For strong scaling I am using $n = 10000$, and for weak scaling $n = 10000\sqrt{t}$ as discussed in the introduction.

Threads	Weak scaling	Strong scaling
1	1.00	1.00
2	1.69	1.73
4	3.12	2.75
8	4.26	3.56

3a

The temporal and spatial locality of traversing the octree is somewhat poor, as there is no guarantee nodes in a sibling or parent/child relationship are close to each other in memory. This is because each node is `malloc()`ed separately. For particles with positions close to each other, the traversals of octree in `bh_accel()` will be somewhat similar, which could aid temporal locality, but there is no guarantee (or even likelihood) that particles that are adjacent in 3D space are adjacent in the particle array.

However, with the low values of n I use for benchmarking, it is likely that the tree fits mostly in cache, so the practical performance impact is likely minor.

3b

I have parallelised two loops, corresponding to the loops I also parallelised in the naive N-body simulation.

Construction of the octree is completely sequential, and parallelising it requires a very different approach.

There is one loop that could also be parallelised without too much trouble: the one that computes the minimum and maximum coordinates of particles. This is a kind of fmin/fmax reduction, but OpenMP does not by default support fmin/fmax reductions. It can be parallelised with manual use of atomic updates, but I decided not to do that.[8]

[8] **Correction note:** this is fine, but they should note it.

3c

The following measurements are with 8 threads and $\theta = 0.5$.

n	**Speedup**
500	0.81
1000	1.28
5000	4.46
10000	6.93

It is clear that the speedup improves as n increases.

3d

With $n = 10000$.

Threads	**Strong scaling**
1	1.00
2	1.49
4	2.80
8	3.91

3e

When measuring weak scaling, we must increase the amount of work to be proportional with the number of threads we are using. However, it is difficult to quantify how much work `nbody-bh` does for a given dataset, since it depends on how the points are clustered. We cannot simply linearily increase the number of particles. Measuring strong scaling is straightforward, since the workload is fixed.

3f

For $\theta = 0$, `nbody-bh` will traverse every particle, just like `nbody`. With $n = 10000$, `nbody-bh` runs in $8.03s$ and `nbody` in $1.53s$. The reason `nbody-bh` is slower is that it has to pay the cost of constructing (and destructing) the octree, and the actual traversal of the tree is much less efficient than a simple `for` loop.

4a

While the code does not contain any optimisations, I have noticed that in `bh_accel()`, we are computing the distance `d` from `p` to the centre of mass possibly twice: once to determine whether we need to recurse into the children, and (if we do not recurse) in `force()`. This could be optimised by inlining the definition of `force()` and reusing the already computed distance.

4b

We could eliminate the corner coordinate and length fields, by instead computing these as needed when we traverse the tree. This would make the tree more compact, but perhaps the additional recomputation would make the overall program slower.

4c

In the worst case, the octree needs $n \log n$ nodes. Since n is known, we could pre-allocate all nodes as a single big array. This means we need fewer calls to `malloc()` (and saves associated metadata), and also lets us nodes in memory in some order that is advantageous for locality when we traverse it - for example by using a space filling curve.[9]

[9] **Correction note:** We don't really expect anyone to talk about space filling curves, and it's good if they just mention the idea of bulk-allocating the nodes.