HPPS-EXAM

Exam nr 40

01-2025

Introduction

After implementing the exam problem, as stated under tasks I am generally satisfied with my solution.

There were some ambiguity in regards to counting distances between points in dist-histogram, weather the formulation under 2.2 "all the distances between different points in a collection of points" refer to a solution where we one count the distance from point A to point B, or also include the distnace from B to A, hence double counting. In the presented solution I double count the points as this was the method that resulted in outputs matching the reference outputs under "A Examples". I also failed to obtain perfect scaling in dist-histogram.c of the parallel implementation. I suspect the meassuring of the workload upscaling might be off, or that since the problem scales quadratically weak scaling is unobtainable if I scale after filesize. I have choosen to scale based on file size, but am not fully sure what the correct way of scaling is in this scenario.

My solution produces the example outputs given the example inputs, and is to my knowledge conceputally correct and of a high quality in regards to application of course relevant methods. None of the provided test cases fail. No memory leaks were detected with valgrind.

For benchmarking I use an Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz processor, with 4 cores, with hyper threading 8 threads. My cache sizes are; L1: 256 KiB (data cache per core), L2: 1 MiB, and L3 6 MiB (6144 KB, shared across all cores).

1. Answers regarding util.c

1.a)

For both read_ints() and read_points(), we get file formats that contain a header with information on the size of the file, we can use this info for allocation. This is not the case for read_lines() where it is challenging that lines files can contain any nr of lines of any size, since in c we allocate memory for this before reading in the information with fgets. Hence our function need to be able to reallocate memroy if (1) the lenght of a line exceed the preallocated buffer lenght or (2) if the nr of lines exeeds the preallocated nr of lines.

- (1) The basic strucutre of the funciton is a while loop that run until we have read all lines and return our lines array. Each itteration we attempt to read a line. In that attempt we use anoter while loop, that continues to read in an array of chars: fgets(line + current_length, line_capacity - current_length, file) == NULL. Where fgets read in line_capacity nr chars to line. In the case when this fails and we need to reallocate we start were we left off in the previous attempt at current_lenght and read in the rest of the chars util line_capacity - current_lenght. But again this may fail again for which we reallocate line_capacity, doubling it each time, and again start at our new current_lenght.

- Our reallocation startegy in the case (2) when we exceed our rows is a bit more straight forward, we just check if our current row counter *n is larger than our preallocation, in which case we use realloc to doulbe the lines_capcacity size (nr rows allocation). We run this check every time we read in a new line.

1.b)

We can jsut look at the definition given to us:

"A lines file is a file comprising zero or more lines, each of which is terminated by a newline character ('\n', ASCII 10). That is, a lines file is a standard (Unix) text file."

From this definition follows that there are 2 types of valid lines files:

- 0, lines - empty

- 0<lines - file must end with newline character ('\n', ASCII 10)

While our read_lines() function can read any file tecnically only a file ending with the end of line character is a valid lines file.

1.c)

The function read_lines can fail in the following ways:

- File Opening Failure: If fopen fails to open the file (e.g., due to permissions or an invalid path), the function prints an error and returns NULL.
- Memory Allocation Failure: If malloc, realloc, or strdup fails to allocate memory for buffers or line storage, the function prints an error, cleans up, and returns NULL.
- Read Errors: In the unlikely event of an I/O error during fgets, subsequent reads may not succeed. The code checks fgets for NULL, and if it never succeeds, no lines will be returned.
- Unbounded Resource Use: Extremely long lines or files with a very large number of lines may exhaust memory or slow processing. One could implement a max lenght for a line, or open it, read the last line and see if it ends appropriately, but i have choosen not to handle this since I dont want to impose on the user how they define their lines files. In reality any super long file is a problem with finite hardware, but I here trust the user to provide the right hardware and judgement for it's usecase.

---

2. Answers regarding histogram.c

2. The following questions pertain to histogram.c. a) Do the memory accessed performed by histogram_sequential() exhibit good spatial or temporal locality? Why or why not? b) Measure and show the weak and strong scaling of histogram_parallel_bins() and histogram_parallel_samples() as you vary the number of threads. Explain how you chose the datasets and why one function may or may not scale better than the other. c) Is it plausible that two inputs, both of size n and for some fixed k, but containing different values, can differ in how long it takes to compute their histograms, and if so, what might such inputs look like, and what is the reason one is slower than the other?

2.a)

In histogram sequential, we are sequentially accessing integers from the samples dataset of size n. We then update the counter for our bins contained in H, as we discover samples within our range for H, [0, k-1].

- Sptaial locality: Our implementation shows good spatial locality as we traverse the samples array sequentially, hence accessing memory slots next to each other. Unless we have a dataset larger than 6 MiB, which is the case here, all our points will fit in our cache. When we update H however

we may reach for memory slots for any of Hs k elements at any time. This will not be a problem here since H (array with k bins, holding our histogram) likely is quite small in comparison to n (size of our samples dataset for our integers), but strictly speaking this operation does not show good spatial locality. However overall spatial locality is very good.

- Temporal locality: When reading in the data sequentially we do not access it afterwards, in regards to temporal locality we are left with analysing the H[sample]++ operation updating our bins. This operation for a uniform random dataset is random, hence the updates we do to slots in H also becomes random. In the case wehre H has a large k nr elements this leads to poor temporal locality. If we are lucky and have a clustered dataset so that we mainly access a small nr of bins over and over agin this will lead to better temporal locality, as of the nature of the dataset, not the algorithm.

2.b)

Weak Scaling

I meassure speedup in throughput over using one single thread vs using many. $throughput = worksize/runtime$

The size of ints dataset I can fit in to L3 is approximately: $\frac{(6144)\cdot 1000}{4} - 1 \approx 1,535,000$. I choose sto scale out of cache here since benchmark is running quite fast. I choose n = 1 000 000 ints per core and a moderate distance s of 7, and a decently large range to wrap around k=86. In weak scaling we will scale the dataset with the threads we run. We choose our nr bins to 100. Our implementations loop through the data once having $O(n)$ time complexity, we scale the datasets as $1000,000 \cdot t$ to keep per thread workload constant:

- 1 thread - 1 000 000 ints
- 2 thread - 2 000 000 ints
- 4 thread - 4 000 000 ints
- 8 thread - 8 000 000 ints

Having some distance s between points will increase the relative cost of the update operation, while increasing the number of points will increase the relative cost of sequentially traversing the data as n. After trying a few different benchmarks I think this is a good representation of the problem. And since we have to reach out of L3 cache for these datasets, this give a good approximation for large non chashe bound problems as we scale out of its size.

Strong scaling

Now holding the problem size constant I choose to run the initial size 1000 000, k=86, s=7, bins=100 assigned to one thread for comparison. I meassure speedup in latency relative to using a single thread. $speedup = \frac{latency(1thread)}{latency(t-threads)}$

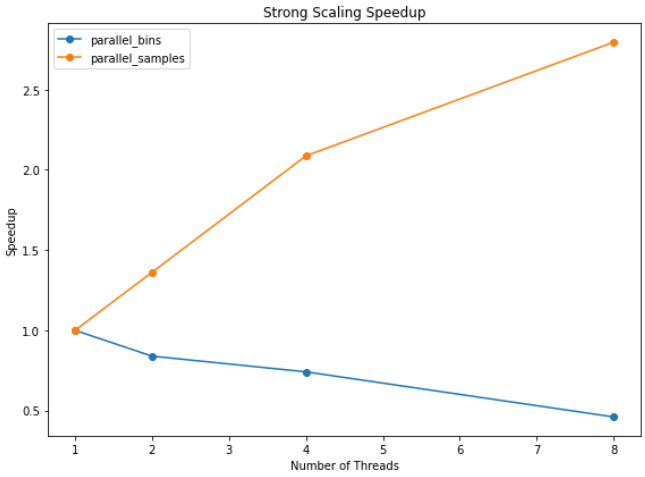

Figure 1:

Results

Results 1 000 000: latency strong, througput weak**:**

Threads	Strong : bins	Strong : samples	Weak : bins	Weak : samples
1	1.000	1.000	1.000000	1.000000
2	1.005	0.965	1.550994	2.535270
4	0.878	0.815	1.825070	2.260870
8	0.900	0.385	1.845363	1.993881

We see very fast runtimes overall and no scaling for strong:bins, while strong samples got slower. The runtimes indicate that the workload might be too small for the overhead of paralellisation with higher cores beeing worth it. We can try increasing the workload by scaling to 10 000 000 samples instead.

Threads	Strong : bins	Strong : samples	Weak : bins	Weak : samples
1	1.000000	1.000000	1.000000	1.000000
2	0.839330	1.361278	0.892088	1.240943
4	0.741460	2.086846	0.736635	1.772111
8	0.460535	2.795653	0.490647	2.955364

Why one bins may scale worse than samples

In dist histogram we divide the final histogram H into some ranges from a start bin to a end bin for each thread. We then loop though our samples array and if we encounter a value that fits into a given range for some thread we update that bin. This way only one thread is responsible for updating some range in H. However this turns out quite ineffective since we still end up looping though our whole samples array with every thread which arguably is the part of the

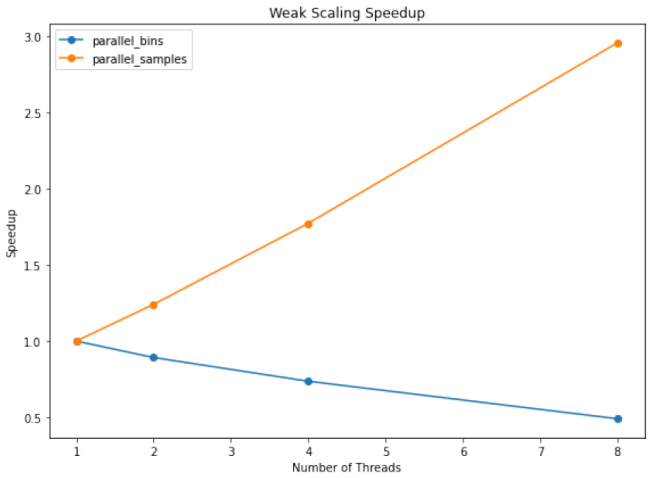

Figure 2:

problem that probably would scale the most. While keeping a separate thread for updating some range of bins would in theory make the update operation faster, we need to keep in mind that the nr bins we want to show in a histogram likely is not that many compared to n, moreover this paralellisation comes with overhead that likely just slows execution down for many cores. We see this reflected in fig1 diagram.

It makes much more sense to do as we do in paralell samples and have each thread be responsible for some range in samples. Hence we effectively divide the problem up, in similar workloads to spread across the threads. This make sense with the default constant sheduling for our paraelell region. In the end we combine each threads partaial solution for its given range into one histogram. In this case since we divide the problem up with more cores it makes sense that we see significant speedup as is confirmed by our result.

2.c)

For histogram computations, balanced value distributions lead to better performance, while skewed distributions result in imbalanced workloads and higher contention, which slow down execution.

- Uniform vs. Skewed Distributions. A uniform distribution may spread writes across the entire range $[0, k-1]$, causing more cache misses when updating H. A highly skewed distribution (e.g., many samples falling into just a few bins) repeatedly accesses a small subset of bins, increasing temporal locality and often speeding up updates.

- Parallel Load Balancing. For parallel implementations, a distribution that heavily concentrates values in certain bins (for parallel_bins) or certain

slices of the samples array (for parallel_samples) can lead to uneven thread workloads or more frequent updates to particular cache lines, impacting runtime.

Overall, memory-access patterns and load balancing can vary significantly depending on the data's value distribution, so two equal-sized inputs can indeed exhibit different performance for histogram computation.

---

3. Answers regarding dist-histogram.c.

3. The following questions pertain to dist-histogram.c. 12 a) Do distance histograms exhibit worse or better locality than counting histograms? b) Howdid you parallelise histogram_parallel(), andwhy? c) Measure and show the weak and strong scaling of histogram_parallel() as you vary the number of threads. Explain how you chose the datasets.

3.a)

Distance histograms generally exhibit worse locality than counting histograms because:

- All-pairs computation: For n points, we need a nested loop of size $O(n2)$, resulting in a more scattered read pattern of the point coordinates in memory (as each point is compared with every other).
- Potentially random bin updates: The distance calculations can produce bin indices that vary widely, leading to less predictable updates of the histogram array H.
- Larger working set: If n is large, repeatedly accessing different pairs of points can exceed typical cache capacities, causing frequent cache misses.

By contrast, a counting histogram scans each sample exactly once and updates a bin based on the sample's direct value, which is typically a simpler, more sequential operation. Consequently, counting histograms can achieve better (though not always ideal) spatial and temporal locality compared to the all-pairs distance approach.

3.b)

Process:

My first instinct was to use a similar strategy to paralell_samples(), witch thread based chunking splitting the problem up through a paralell region, creating a histogram for each thread and then combining the histograms in the end.

However i realised i can algorithmically half the problem by instead of having a full nested loop to find every distance between points A and B, only check for points we have not previously checked in A, so for the nested loop i start at i+1. I then double count each distance for the range [0,k-1] to account for both the A to B and B to A distance.

With this discovery my previous paralellisation was still in theory just as good as before, but loosing the benefits of my new optimisation. I would have lost potential in the sense that the last range where i+1 is high, would have to wait for the thread doing the first. In other words the load inbalance would be quite big. I tried a dynamic scheduling apporach and actaully got a better result.

However i realised as i was doing this that i knew more or less how unequal the workload was, and that i just could adjust the ranges in my original implementation to account for this. In theory this solution is great for large uniform datasets. I practise I only managed to implement a version dynamically calcuating the customsied ranges, which in itself took some looping. I also came pretty far with a more static calculation but did not managed to make this work in code. The dynamic calculaton of the adjusted ranges seemed to do similarly to my dynamic apporach for 8 threads where as for lower thread counts my dynamic approach seemed faster. In the end I think that if I could get the calcualtion to work, without looping and checking things in the code and then running a static paraellel region would be best for high thread machines and large uniform datasets.

However with the overhead of doing those calclation potentially beeing similar to my dynamic solution i choose to go ahead with that one as with the dynamic region it would generally also be more flexible for different kind of datasets and workloads.

Solution:

I am creating thread local histograms and combining them jsut like in our histogram_samples() strategy, but but here with the dynamic region not to split up the problem but to avoid threads updating the same bins. My solution is similar to that one but inside the paralell region i have another dynamic region where we dynamically take chunks of 10 from our points list and process. Here the reason for creating thread local histograms becomes hence not that we are splitting the problem in terms of range, but that we avoid that svereal cores constest on updating the same bins in H, here every thread has their own histogram. This avoids race conditions. We then combine the locla histograms. This step was not paralellised as it might lead to race conditions, however its possible that we could in some conveluted way. However since this step for a large n and smaller nr bins, probably is negilabe i choose to let it stand as is.

3.c)

Weak scaling

The problem we are solving is approximately $O(n^2)$, since we have a nested loop. Hence I want to scale my workload with the base worload for one thread $base \cdot \sqrt{t}$, as I scale. I start with 100 000 points. This scaling is right in terms of algorithm complexity and this is what I will go with here but it might be impossible to archieve weak scaling defined like this since our problem size increases exponentially with every point that is added and all new relationships that creates. For T threads we have : T=1 : n= 100 000, T=2 : n = 141,000, T = 4 : n = 200,000, T = 4 : n = 200,000. We repeat 10 times, and run with 100 bins, spacing is 1.4.

Strong scaling

Keeping the problem size constant with 100 000 points generated in the same way as for the base case in weak scaling.

Results
We see sublinear strong scaling which is expected, and unfortunately we dont have perfect weak scaling. I tried a myriad of algorithms and with similar results, making me think that this is a problem with the way I conduct the weak scaling test, that the problem scales faster than our solution. If I had more time I would try to in better detail calculate the workload as I scale the number of points and relate it to my algorithm complexity and the way I scale my test datasets during the benchmark.

Threads	Strong	Weak
1	1.000000	1.000000
2	1.836572	1.308437
4	3.097994	1.535818
8	3.994142	1.403299

---

4. Answers regarding dist-histogram.c.

4. The following questions pertain to strings2mapping.c and strings2ints.c. a) Why might it be more difficult to write C programs that use mapping files, if we did not assume that strings are human-readable ASCII strings, but could contain arbitrary bytes?

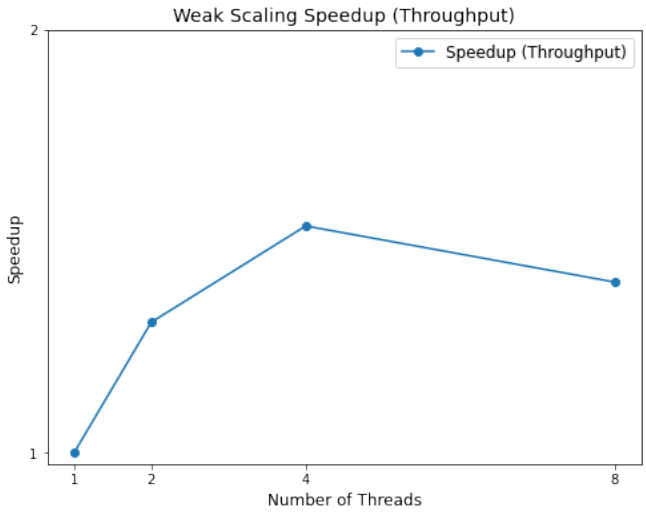

Figure 3:

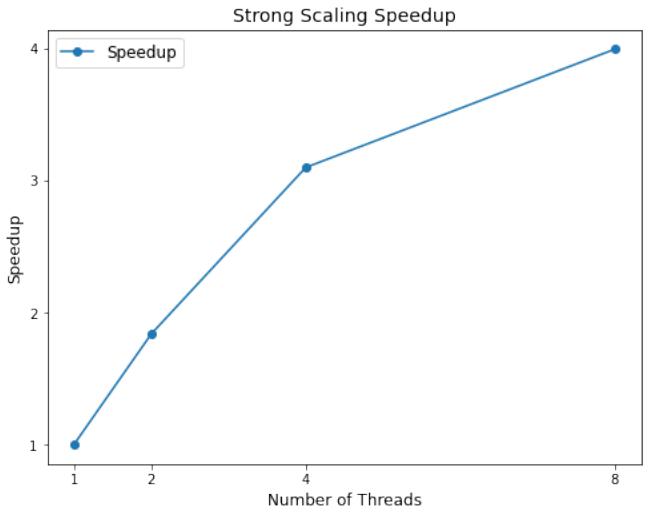

Figure 4:

4.a)

If the file format allowed arbitrary byte sequences as "strings" (rather than strictly humanreadable ASCII), we would face several complications:

- Null Terminators and String Lengths. Typical C string functions (strcmp, strdup, etc.) rely on a null terminator ('\0') to mark the end of a string. Arbitrary data could contain null bytes in the middle, rendering these standard functions unusable or potentially insecure
- Binary vs. Textual Operations. Reading and writing such data would have to be done strictly in binary mode, taking care to preserve all bytes. We could not safely treat them as typical text strings, so functions like fgets would not be adequate.
- Length-based Parsing. We would need to track the exact length of each byte sequence and store/compare them carefully, likely using memcmp instead of strcmp. This introduces extra complexity in memory management (e.g., always storing length and pointer).
- User Ambiguity. Debugging or verifying the files by inspection becomes more difficult, since the raw bytes may not be human-readable, requiring binary-hex dumps to interpret them

Thus, restricting strings to human-readable ASCII simplifies parsing, debugging, and usage of standard string-handling functions in C.