J. H. Walther
S. Martin
ETH Zentrum, CLT
CH-8092 Zürich

Fall semester 2022

Set 1 -Amdahl's Law, Roofline Model, Cache

Issued: September 28, 2022
Hand in (optional): October 12, 2022 08:00

Grading: For the exercise submission, you only need to submit 4 out of 5 questions, i.e. the ones not marked with "OPTIONAL".

Question 1: Amdahl's law (30 points)

a) Suppose you have a program where 99% of the runtime is parallelizable. Your boss gives you a computer which has a CPU with 64 cores.

- He wants you to achieve a speedup of 50. Is this possible? Justify.
- What is the maximum speed up you can achieve if you had no limitations of CPU cores n?

b) Your boss doesn't want to spend too much money on upgrading your Computer.

- What is the minimum amount of cores needed to have at least a speedup of 50?

c) You somehow improved your program and now achieve a parallelization fraction of 99.5%.

- How many cores are now needed to reach a speed up of 50?

d) Given the same parallelizable fraction from the previous question $p = 99.5\%$, you just found out that your code does not follow Amdahl's law. Part of your code, the CPU cores do have to communicate with each other, which costs a constant amount of time for every core. Therefore the corresponding time for this operation scales proportionally with the number of cores as $0.001(n-1)$[1].

- What is the maximum speed up you can achieve and for how many cores?

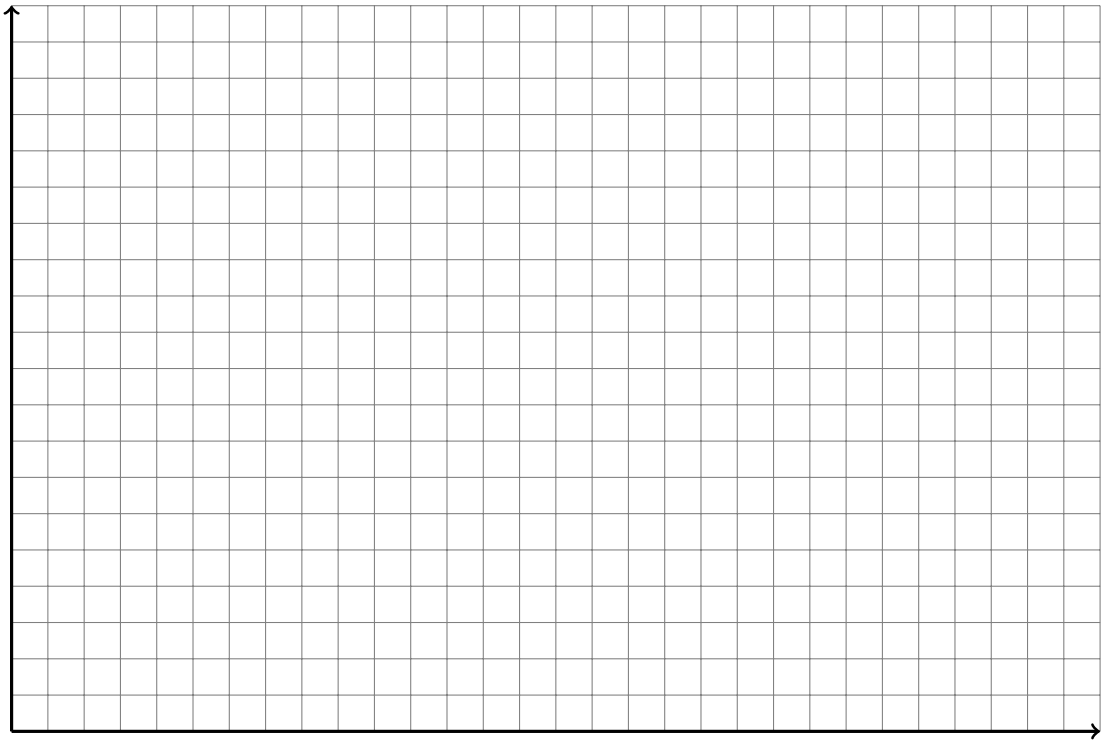
[1] Note that $(n-1)$ because if you only have 1 core, there is no communication.

Question 2: Parallel Scaling (20 points)

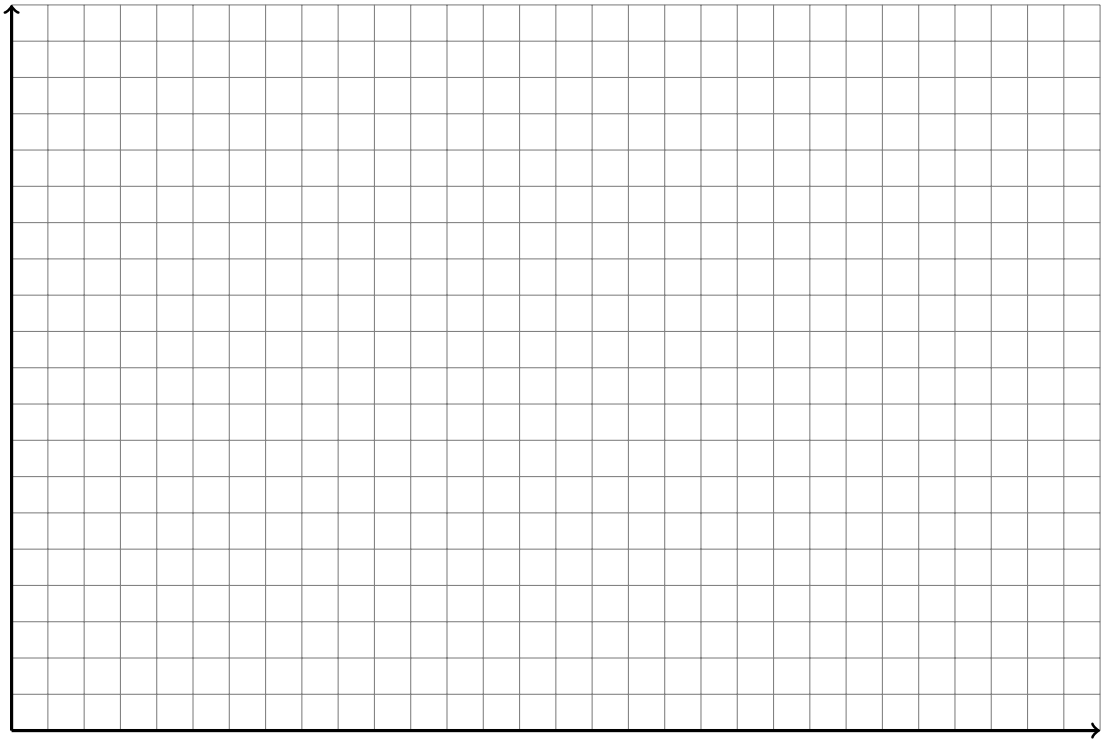
In this exercise, we want to draw the plots of strong and weak scaling. You are given runtime measurements of a simulation with different array sizes N and for different numbers of CPU cores P. For the weak scaling Efficiency, the runtime of the system is increasing linearly by N. You can draw the graphs directly on paper or use any plotting software (MATLAB, Matplotlib, ...).

P\N	1000	2000	4000	8000
1	100	300	600	1300
2	70	165	310	680
4	39	95	170	330
8	27	55	90	170

- Plot the points of the strong scaling speed up, as well as the ideal line.
- Show all steps of the calculations.
- Don't forget to Label the axes.



- Plot four points of the weak scaling Efficiency using $N = 1000$ and $P = 1$ as a reference.
- Show all steps of the calculations. Also, draw the Ideal line and label the axes.



Question 3: Roofline Model (30 points)

a) Given the following code:

```
1    double a[N], b[N], c[N];
2    ...
3    for (int i = 0; i < N; ++i) {
4
5        c[i] = a[i] * a[i] * a[i] + b[i] - c[i] + 1.0;
6    }
```

- What is the operational intensity of the code, assuming that we have infinite cache size and cache is cold (empty)?
- State all the assumptions you made and show your calculations.

b) Given the following code:

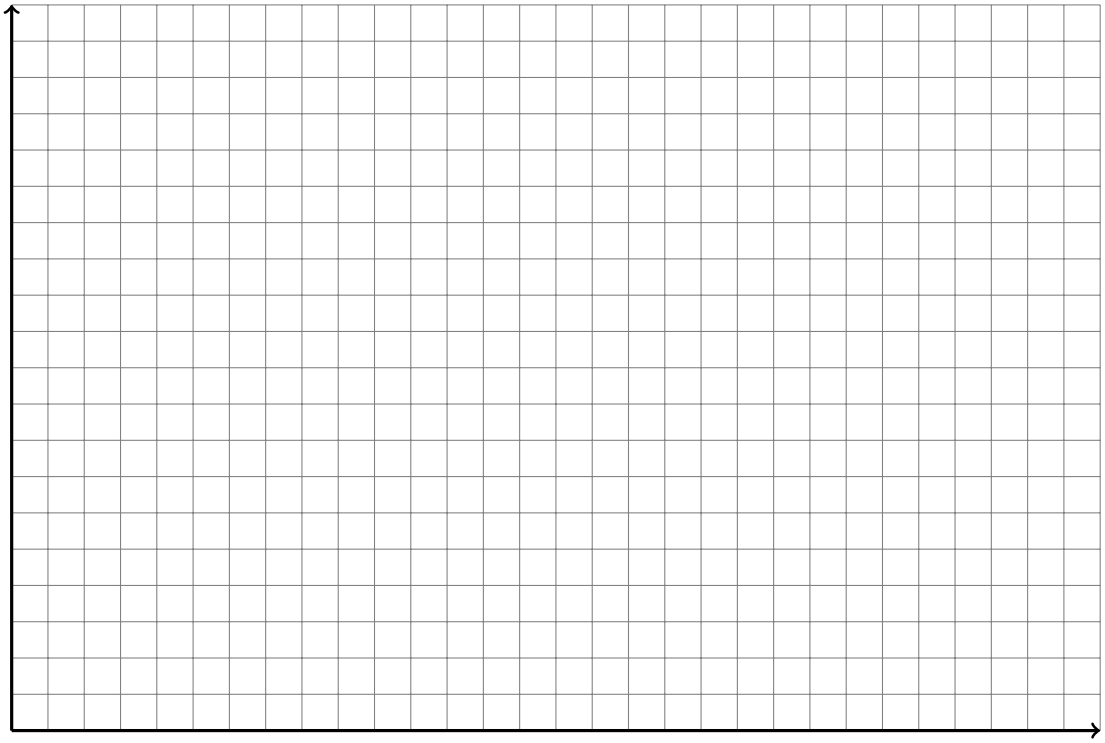
```
1    float a[N], C[N*N];
2    ...
3    for (int i = 0; i < N; ++i) {
4        for(int j = 0; j < N; ++j){
5            a[i] = a[i] + C[i*N+j] * 2.0;
6        }
7        C[i*N+i] = C[i*N+i]/(a[i] + 1.0);
8    }
```

- What is the operational intensity of the code, assuming that we have infinite cache size and cold cache?
- State all the assumptions you made and show your calculations.

c) You are given an AMD R9 5950x which has 16 cores. 3.4GHz clock speed. As each core has 2 FMA[2] units and 2 floating point add/sub units. The CPU support AVX2 instructions. For memory, it has a 32KB L1 data cache, 512KB L2 cache and 32MB L3 cache. The measured memory peak bandwidth of this CPU is $\beta = 6.5 bytes/cycle$.

- State the peak performance and peak bandwidth of this CPU for multi-core and with AVX2 vectorization.
- Draw the roofline model of this CPU down below for non-vectorization and only for single-core performance.
- Don't forget to label the axis.
- State the ridge point.

[2] FMA = Fused Multiply Add floating point operation, which does a*b+c in one cycle.



d) You measured the performance of a) being 0.5 flops/cycle and from b) being 0.6 flops/cycle.

- Draw those points inside your roofline model.
- Are they memory bound or compute bound?
- How do they perform?

Question 4: Linear Algebra Operations (30 points)

In this exercise we will implement some basic linear algebra operations. We study the effects of storing and using data in different ways on the efficiency of the code.

a) We want to implement the element-wise matrix-matrix multiplication (also known as the Hadamard product). Implement it ones in row-wise and ones in column-wise. You are provided with a skeleton code and only have to fill in the TODO parts.

- State the time difference between T(col-wise)/T(row-wise) for different sizes of N and for different compiler flags.
- What do you observe?
- Play around with the compiler optimization flags to see if something changes with the runtime of the code.

b) In this exercise we will compare our implementation of matrix-matrix multiplication with LAPACK subroutine function dgemm(). You are provided with a skeleton code and you only have to fill in the TODO parts.

c) We want to compute the product of two square matrices $A, B \in \mathbb{R}^{N \times N}$ The matrices A, B are stored in row major order in our implementation. One way to do the multiplication is the straightforward way,

$$C_{i,j} = \sum_k A_{i,k} B_{k,j} \, .$$

As discussed in the previous subquestion, you have already seen the reasons this is not the most efficient way of implementing the product.

First, implement the straightforward multiplication algorithm in the provided skeleton code. Then implement the block-multiplication algorithm that was explained in the classroom. The block sizes are given. Finally, implement a second version of the block-multiplication algorithm where the matrix B_col is stored in column-major order. Initialize all the matrices with random numbers (not too big numbers). Note that B_row & B_col have to have the same values but in their respective major order stored, because you want to get the same matrix-matrix multiplication results for all implementations.

- Run your code over increasing matrix dimensions and vary the block size.
- What do you observe? How do you explain it?
- How much faster is the Lapack function?
- In the end, the results are saved in a file. Plot the results with the given python script or with your favourite tool.
- Analyze the results and draw your conclusion. Which one is the most efficient algorithm? Explain in terms of cache usage.

Question 5: OPTIONAL - Cache size and cache speed (Not graded)

This exercise shows how the performance of a program can be affected by the size of the data it operates with. Depending on whether the data fits into L1, L2, L3 cache, or not at all, we expect different memory access time. Furthermore, in which order we access the elements will also have an effect.

We will demonstrate this by traversing a linked list in the form of a permutation of size N, for different values of N. In other words, we will have an array of integers $a_0, \ldots, a_{N-1}$, where each a_i is a unique value from 0 to $N-1$. We start with the index $k = 0$, and then repeat M times the operation $k \leftarrow a_k$, for some $M \gg N$. This way we minimize memory-unrelated operations and measure virtually only the memory access time[3].

a) Before writing and running the code, check the sizes of L1, L2 and L3 cache by running the following command on an Euler compute node:

`grep . /sys/devices/system/cpu/cpu0/cache/index*/*`

The output will contain information from four different `index*` folders, each of which represents one cache level. Two are L1 caches (one for data, one for program code), one L2 and one L3 (both unified data and code). Extract the following information[4]:

- total size (property `size`),
- cache line size (property `coherency_line_size`).

b) You are provided with a skeleton code for sampling the execution time for different values of N. The code already selects the values of N and outputs the results.

Fill out the `TODO` sections marked with *Question 5b* with the code for linked list traversal and time measurement. Use the provided `sattolo` function to generate a random one-cycle permutation. This function guarantees that the permutation is such that all of the N elements are visited. Compile the code with `make`, run with `bsub ... make run` and plot the results with `make plot`.

What do you observe, do the drops in performance match the cache sizes? Are the transitions smooth or sharp, why?

c) Instead of jumping randomly through memory, initialize the array a such that k goes repeatedly as $0, 1, 2, \ldots, N-1, 0, 1, 2 \ldots$ Compare the performance of this (mostly) continuous access to the random access from the previous subquestions.

How would you explain the result?

d) The previous subquestion was somewhat unfair. We would load a single cache line (of 64 bytes), and then do several jumps for free, because the data is already there. Implement the third variant of the permutation, where k jumps by 64 bytes (how many elements is that?). If that would cause k to go above $N-1$, take the modulo N. It does not matter if not all elements are visited this way, we still do force the CPU to load the whole array from memory, as we read from every cache line.

Compare the results with the previous two cases. What limits the performance for very large N in this case, and what in the case of a random permutation?

[3] To be precise, we measure latency of reading a_k from memory (or cache), plus latencies of CPU instructions themselves. Thus, this will only give as an approximation of the memory and cache latencies.

[4] Depending on the node type your program assigned to, you will get different values. Optionally, you might also want to run `lscpu` (in the same run), to get the exact CPU model name.