ETH zürich

High Performance Computing for Science and Engineering I

P. Koumoutsakos
S. Martin
ETH Zentrum, CLT E 13
CH-8092 Zürich

Fall semester 2021

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

Issued: October 1, 2021
Hand in (optional): October 15, 2021 12:00

Grading: For the exercise submission, you only need to submit 3 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 99.99% of the runtime is parallelizable. You want to run this program on a super computer that has up to $2'000'000$ cores.

What is the maximum speed up you can achieve if you had no limitations of processors n? What speed up can you achieve with 20, 200, 2000, $20'000$, $200'000$ and $2'000'000$ cores?

b) You are executing a code with serial fraction 0.01. However, there are parts of a program that do not benefit from increasing the number of cores n used. Assume that the parallel execution of your code requires a communication operation, which costs a constant amount of time for every core, therefore the corresponding overall time for this operation scales proportionally with the number of cores as $0.01n$. What is the maximum speed up you can achieve with this constraint (and for how many cores)? If you reduce the time spent on this communication operation to $0.001n$, what is the new maximum speedup?

c) Upon inspection of a code, you see that 1000 of the code's operations are parallelizable and 10 are not. Suppose the time to compute one operation (both serial and parallel) is t_1.

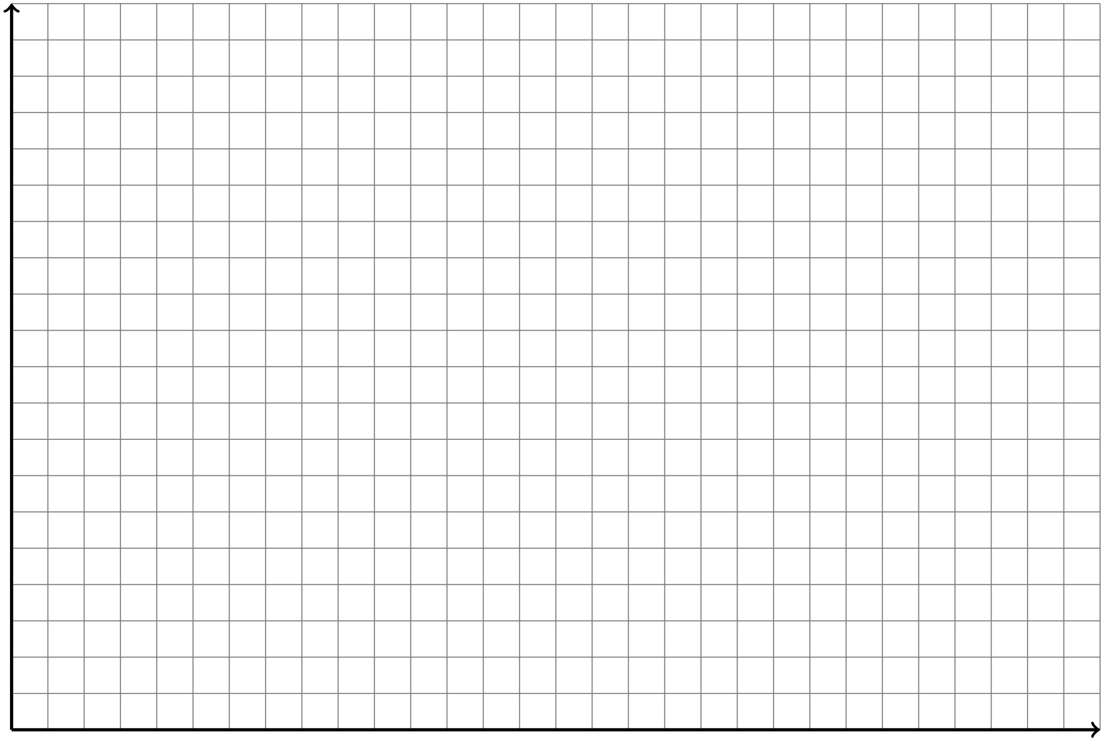
- Compute the time $T(n)$ to execute the code with n cores.
- Using this time $T(n)$, compute the speed-up $S(n)$ for $n = 10$. Verify your answer using Amdahl's law.
- The speed up you get is true only in the case of perfect load balancing. For $n = 10$, re-compute the speed-up assuming one core is assigned 1.5 and 3 times more of the parallel operations.

Question 2: OPTIONAL - Parallel Scaling (Not graded)

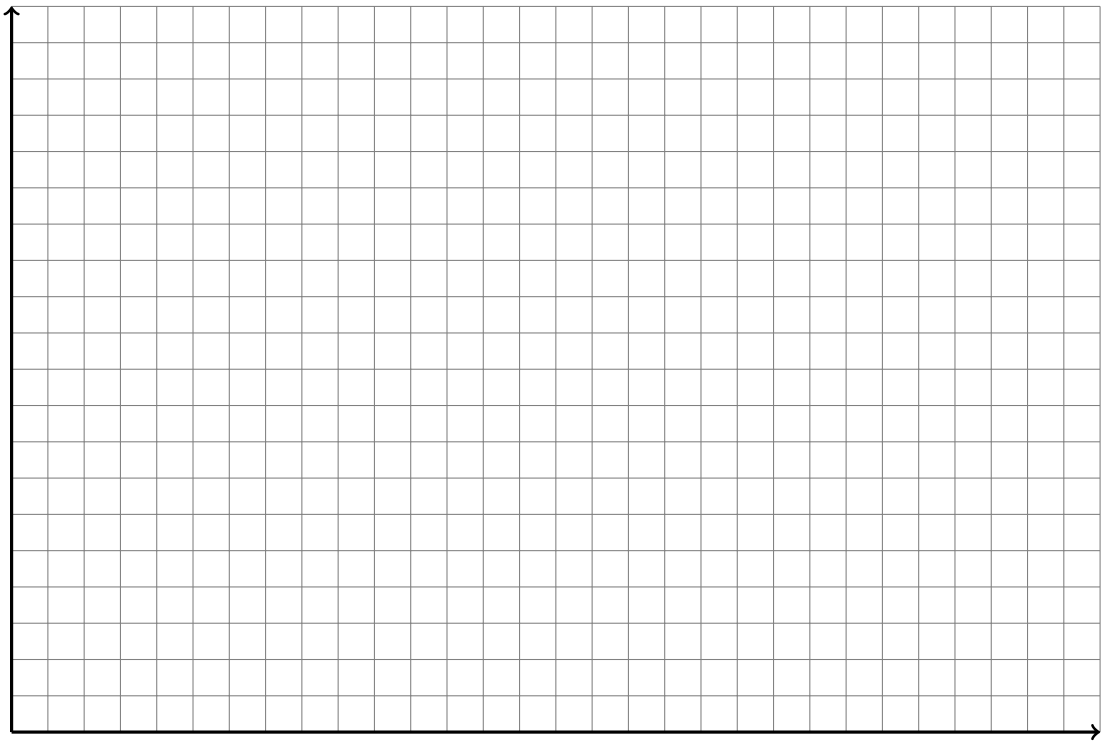
A program simulates N particles. All particles interact with each other and, thus, the number of interactions is proportional to N^2. The runtimes of the program in seconds on P processor cores are in the table:

P\N	500	1000	1500	2000
1	6.00	30.00	72.00	120.00
4	1.50	7.50	18.00	30.00
9	0.75	3.50	9.00	20.00
16	0.50	2.15	6.00	12.00
24	0.40	1.50	4.50	10.00

Plot four points of the strong scaling. Show all steps of the calculations. Label the axes.



Plot four points of the weak scaling using $N = 500$ and $P = 1$ as a reference. Show all steps of the calculations. Label the axes.



Question 3: Roofline Model (30 points)

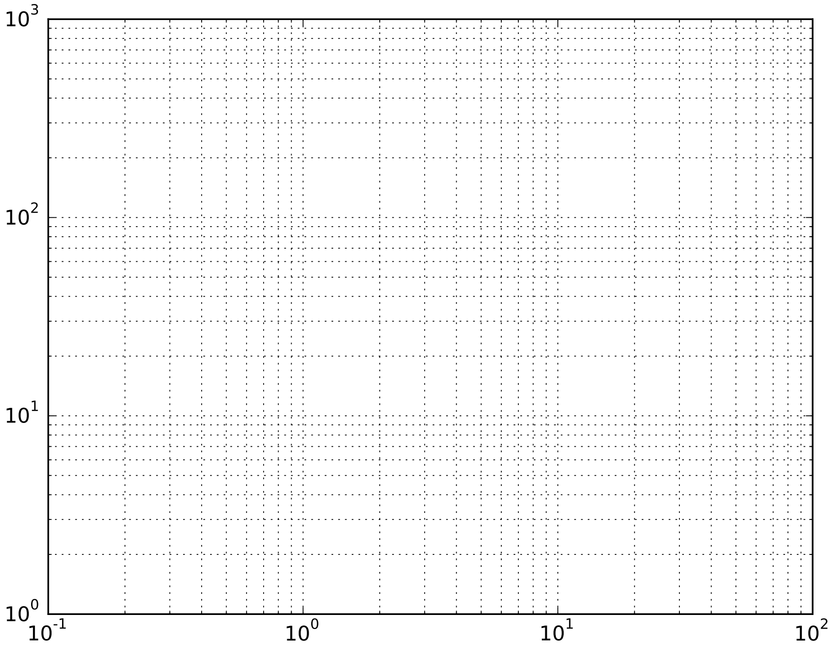
Given the following code:

```
float A[N], B[N], C[N];
...
const int P = 2;
for (int i = 0; i < N; ++i) {
    int j = 0;
    while (j < P) {
        A[i] = B[i] * A[i] + 0.5;
        ++j;
    }
    C[i] = 0.9 * A[i] + C[i];
}
```

a) What is the floating point operational intensity of the code? State all the assumptions you made and show your calculations.

b) For a peak performance of 409.7 GFLOP/s (single precision) and a memory bandwidth of 34 GB/s, find all positive `P` for which the code is memory bound. Assume an infinite cache and state further assumptions you made. Show your calculations.

c) Draw below the roofline corresponding to (b) and label the axes.



Question 4: Linear Algebra Operations (20 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 multiplication of a square matrix $A \in \mathbb{R}^{N \times N}$ with a vector $\boldsymbol{x} \in \mathbb{R}^N$. Notice that we have two options, to store the matrix A row-wise (row major order) or column-wise (column major order). You are provided with a skeleton code and you are asked to implement the product $A\boldsymbol{x}$ storing A with row major order first and then with column major order. You can initialize $A_{i,j} = i + j$ and $x_i = i$.

   Which implementation is faster and why? What do you observe as the dimension of the matrix increases? How do you explain it?

b) We want to compute the transpose of a square matrix $A \in \mathbb{R}^{N \times N}$ The matrix A and its inverse $A^\top$ are stored in row major order in our implementation. One way to do the transposition is the straightforward way, where we loop over all the indices and we set the corresponding (i, j) element of $A^\top$ equal to the (j, i) element of A. As you have seen in the lecture, this is not the most efficient way in terms of cache usage due to compulsory misses. A solution to this problem is to work in blocks.

   First implement the straightforward transposition algorithm in the provided skeleton code. Then, imitate the multiplication algorithm that was explained in classroom and use the provided skeleton code to implement a block version of the transposition algorithm.

   Run your code over increasing matrix dimensions and vary the block size. What do you observe? How do you explain it? In the end, the results are saved in a file. Plot the results with your favourite tool. Analyze the results and draw your conclusion.

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. Finally, implement a second version of the block-multiplication algorithm were the matrix B is stored in column major order. You can initialize $B_{i,j} = 2i + j$.

   Run you code over increasing matrix dimensions and vary the block size. What do you observe? How do you explain it? In the end, the results are saved in a file. Plot the results 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[1].

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[2]:

- 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 1b* 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?

[1] 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.

[2] 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.