Exercise 1

As we learned in last week's lab, the speed up $S(p)$ is defined as the fraction of serial execution time t_s of the best sequential algorithm and parallel execution time t_p. In this exercise, the execution times are not given in terms of seconds, but rather asymptotic complexity using O notation is provided.

Of course, we could argue that asymptotic behavior might not be the best measure. However, algorithms are commonly assessed using asymptotic measures. Before implementing both algorithms (the sequential and the parallel) for benchmarking, this analysis could be used as a reference.

a)

In this task, you were asked to compute the theoretical speedup comparing the parallel algorithm using one processor vs. the parallel algorithm using p processors. This ratio is also known as relative speedup.

$$S(p) = \frac{t_s}{t_p} \approx \frac{O(n^3 + n)}{O\left(\frac{n^3}{p} + n\right)} \approx \frac{C_1(n^3 + n)}{C_2\left(\frac{n^3}{p} + n\right)} = C\frac{p(n^3 + n)}{n^3 + np} = C\frac{p(n^2 + 1)}{n^2 + p}$$

Furthermore, we analyze the asymptotic behavior (assuming a cluster with infinitely many cores p).

$$\lim_{p\to\infty} S(p) \approx \lim_{p\to\infty} C\frac{n^2 + 1}{\frac{n^2}{p} + 1} = C(n^2 + 1) = O(n^2)$$

b)

What is the theoretical speedup comparing the sequential algorithm vs. the parallel algorithm?

$$S(p) = \frac{t_s}{t_p} = \frac{O(n^{2.8})}{O\left(\frac{n^3}{p} + n\right)} = C\frac{n^{2.8}}{\frac{n^3}{p} + n} = C\frac{pn^{1.8}}{n^2 + p}$$

Again, we also analyze the asymptotic behavior (assuming a cluster with infinitely many cores p).

$$\lim_{p\to\infty} S(p) \approx \lim_{p\to\infty} C\frac{n^{1.8}}{\frac{n^2}{p} + 1} = Cn^{1.8} = O(n^{1.8})$$

c)

Given a fixed input size n. How many processors p are required such that the parallel algorithm is faster than the sequential? Using an other formulation: How many processors p are needed such that the execution time t_p of the parallel algorithm is smaller than the execution time t_s of the sequential algorithm?

$$t_p < t_s \Rightarrow O(n^3 + n) < O(n^{2.8}) \Leftrightarrow C_1\left(\frac{n^3}{p} + n\right) < C_2 n^{2.8} \overset{p\geq 0}{\Longleftrightarrow} p > \frac{n^2}{Cn^{1.8} - 1}$$

Note that we assumed $Cn^{1.8} > 1$.

Exercise 2

a)

The first snippet has a high locality of data since each iteration accesses only consecutive elements. The initial matrix access causes a cache miss, and s bytes are loaded into the cache. This cache line holds $\left\lfloor \frac{s}{4} \right\rfloor$ elements since the matrix elements are of type `int32_t` (`sizeof(int32_t) = 4`). Therefore, the subsequent $\left\lfloor \frac{s}{4} \right\rfloor - 1$ elements can be loaded from the cache. The access of the $\frac{s}{4}$ element again produces a cache miss. Note that for this theoretical analysis, the cache architecture is not relevant.

$$f_1 : \mathbb{N}_+ \times \mathbb{N}_+ \to \mathbb{N}_+ : (s; n) \mapsto \left\lceil \frac{8n^2}{s} \right\rceil$$

Conversely, the second snippet has a low locality of data since each iteration of the inner loop skips n-elements. Each matrix access causes a cache miss because we assumed a large matrix ($n >> \frac{s}{4}$).

$$f_2 : \mathbb{N}_+ \times \mathbb{N}_+ \to \mathbb{N}_+ : (s; n) \mapsto 2n^2$$

c)

Cachegrind simulates how a program interacts with the cache. On LCC2, it uses an 8-way associative level 1 data cache (D1 cache) with a cache line size of 64 bytes and a cache size of 32kB. The two extracts below show the results of the simulation for both snippets with a matrix size of $1,000$. Using our functions, we can evaluate the expected number of D1 cache read misses:

$$f_1(64; 1,000) = \left\lceil \frac{8 \cdot 1,000^2}{64} \right\rceil = 125,000$$

$$f_2(64; 1,000) = 2 \cdot 1,000^2 = 2,000,000$$

The tool reports the same number of cache misses (see: line `c[i][j] = a[i][j] * b[i][j];` column D1mr (D1 cache read misses))

```
--------------------------------------------------------------------------------
-- User-annotated source: /ex2.c -> Snippet 1
--------------------------------------------------------------------------------
        Ir I1mr ILmr        Dr    D1mr    DLmr        Dw   D1mw   DLmw

         .    .    .         .       .       .         .      .      .  #include <stdint.h>
         .    .    .         .       .       .         .      .      .  #include <stdlib.h>
         .    .    .         .       .       .         .      .      .  #define N 1000
         .    .    .         .       .       .         .      .      .
         3    0    0         0       0       0         1      0      0  int main() {
         .    .    .         .       .       .         .      .      .   int32_t a[N][N];
         .    .    .         .       .       .         .      .      .   int32_t b[N][N];
         .    .    .         .       .       .         .      .      .   int32_t c[N][N];
     3,004    2    2     2,001       0       0         1      0      0   for (int i = 0; i < N; ++i) {
 3,004,000    0    0 2,001,000       0       0     1,000      0      0   for (int j = 0; j < N; ++j) {
22,000,000    0    0 8,000,000 125,000 125,000 1,000,000 62,500 62,500   c[i][j] = a[i][j] * b[i][j];
         .    .    .         .       .       .         .      .      .   }
         .    .    .         .       .       .         .      .      .   }
         1    0    0         0       0       0         0      0      0   return EXIT_SUCCESS;
         2    0    0         2       0       0         0      0      0  }
```

```
--------------------------------------------------------------------------------
-- User-annotated source: ex2.c -> Snippet 2
--------------------------------------------------------------------------------
       Ir I1mr ILmr        Dr    D1mr   DLmr      Dw    D1mw   DLmw
        .    .    .         .       .      .       .       .      .  #include <stdint.h>
        .    .    .         .       .      .       .       .      .  #include <stdlib.h>
        .    .    .         .       .      .       .       .      .  #define N 1000
        .    .    .         .       .      .       .       .      .
        3    0    0         0       0      0       1       0      0  int main() {
        .    .    .         .       .      .       .       .      .   int32_t a[N][N];
        .    .    .         .       .      .       .       .      .   int32_t b[N][N];
        .    .    .         .       .      .       .       .      .   int32_t c[N][N];
    3,004    2    2     2,001       0      0       1       0      0   for (int j = 0; j < N; ++j) {
3,004,000    0    0 2,001,000       0      0   1,000       0      0   for (int i = 0; i < N; ++i) {
22,000,000   0    0 8,000,000 2,000,000 125,973 1,000,000 1,000,000 63,000   c[i][j] = a[i][j] * b[i][j];
        .    .    .         .       .      .       .       .      .   }
        .    .    .         .       .      .       .       .      .   }
        1    0    0         0       0      0       0       0      0   return EXIT_SUCCESS;
        2    0    0         2       0      0       0       0      0  }
```

However, if we compare the simulation from cachegrind with the real data from the performance counters measured by perf, the results vary significantly (especially for the first snippet). This indicates that there are other optimizations used in the real world. For example, processors can use data cache prefetching in which data is fetched before it is actually required. The lesson is clear: Caches are complicated. Use appropriate tools (e.g., perf) to find performance bottlenecks in your code.

```
Performance counter stats for './ex2' (7 runs) -> Snippet 1:
 13.301.586      L1-dcache-loads:u                                           ( +- 10,12% )  (43,16%)
     13.409      L1-dcache-load-misses:u   # 0,10% of all L1-dcache hits    ( +-  9,25% )  (43,20%)
```

```
Performance counter stats for './ex2' (7 runs) -> Snippet 2:
 11.437.446      L1-dcache-loads:u                                           ( +-  0,89% )  (44,16%)
    995.966      L1-dcache-load-misses:u   # 8,71% of all L1-dcache hits    ( +- 16,71% )  (40,30%)
```