Advanced Programming Tutorial
Worksheet 11: Code optimization

Keywords: avoid branches, loop ordering, blocking, cache optimization, optimization flags

Prepare!

Before coming to the tutorial, and so that the time is enough to solve the exercise, revise the slides of the lecture unit "Code optimization techniques" and answer the following questions:

- ☐ What is the importance of profiling?
- ☐ What actions can you take to improve code performance?
- ☐ How can the choice of different data structures impact the performance of the code?
- ☐ What optimizations can the compiler do for you? How can you enable them?
- ☐ How does cache misses affect performance?
- ☐ Download the skeleton for the pandemic simulation. On which section of the code do you expect to spend most of the computation time?
- ☐ Download the skeleton for the matrix operations. What matrix operations are implemented in the code?

Exercise 1: Simulating faster than the virus spreads

We previously started building a code to simulate the spread of a virus, the prototype of which you can find in `PanSimul/main.cpp`. The main task of this code now is to compute distances between people at random locations, to decide which individuals will get infected. We time the loop that computes the distances using the `chrono` library (see section "some new concepts"). Build the code with `cmake . && make` and try the following[1]:

1. Run the program as `./panSimul N`, where `N` is the number of people. Try with different values of `N` (such as $100, 1000, 2000, ...$). What do you observe for the runtime? Take notes of the runtime you observe for different values of `N`. For the rest of the steps, choose a value for `N` that leads to a runtime of a few seconds (e.g. `N = 5000`). SOLUTION:
   The runtime increases quadratically with `N`.

2. Looking only at the main loop, propose three possible optimizations. Without implementing them: which do you think will have the most impact? Take notes of your predictions.
   SOLUTION:
   Some ideas:

   - Remove the `if` branch. It could affect the pipelining, even though the second if-case is rare.

[1] You may want to use `https://quick-bench.com/` to compare implementations, but it is not necessary.

- Change the order of two `for` loops. It should not matter much, since one of the two people will in both cases mostly be in different cache lines. We could gain a bit from better accessing `distances`.
- Replace the expensive `std::pow(..., 2)` with a multiplication. This should have a large difference, since multiplication costs fewer CPU cycles. This is one of the simplest optimizations that the compiler will do if we enable `-O1` or higher.

3. Remove the branch included in the loop (why?). SOLUTION:
   When the CPU sees an `if`, it predicts the most commonly encountered branch. However, every time the prediction is wrong, it needs to empty the pipeline and perform a few computations again. We did not observe a significant runtime decrease. Here we can simply remove the `else` case, as it is setting the same value as initialized.

4. How does the order of traversing the `distances` vector affect the runtime? SOLUTION:
   We get data from the main memory as small chunks, the cache lines. For this reason, accessing neighboring elements of the memory (e.g. `A[0][0], A[0][1], ...`) is often faster, as often the next element is already in the cache. However, in this case, we did not observe a significant difference, due to other operations still dominating the runtime.

5. Can you get rid of any expensive operations? SOLUTION:
   The easiest to replace is the `std::pow` with a cheaper multiplication:
   ```
   std::sqrt(
   (people[i].x() - people[j].x()) * (people[i].x() - people[j].x()) +
   (people[i].y() - people[j].y()) * (people[i].y() - people[j].y())
   );
   ```
   We observed a significant reduction of the runtime.

6. Look back at your data structures, and in particular `distances`: do you see any potential for improvement? SOLUTION:
   `std::vector<std::vector<>>` is not guaranteed to be contiguous in memory, meaning that each of the row vectors could be stored far from the previous one. We can use a simple 1D vector instead: `std::vector<double> distances(n_people*n_people, 0);` and we can access the elements as `distances[i*n_people+j]`. However, we only observed a minimal runtime difference here.

7. Look at the mathematics of what you are actually computing. Can you save some computations? SOLUTION:
   We are computing distances and we know that $distance_{i,j} = distance_{j,i}$. We can avoid computing half the matrix:
   ```
   for (auto i = 0; i < people.size(); i++){
   for (auto j = 0; j < i; j++){
   ```

8. In `CMakeLists.txt`, change the compiler optimization flag from `-O0` to up to `-O3`. Why does the runtime decrease? For a definite answer, try looking into the produced Assembly code, but don't be demotivated if you get lost. You may want to use `https://compiler-explorer.com/` to easily read the Assembly (right-click on the code and select "reveal linked code"). SOLUTION:
   The optimizer automatically performs many of these optimizations already, such as replacing `std::pow` with a multiplication.

9. Take a step back and compile the original code with `-O2`. Does only using the optimizer lead to equivalent performance benefits? SOLUTION:
   The compiler is smart enough to perform many of the low-level optimizations we already applied, but it cannot optimize our algorithm.

Look back at everything you tried so far: what can you conclude? SOLUTION:
Aim for readability and for implementing efficient algorithms, while also letting the compiler optimize your code. In performance-critical parts of the code and in case of doubt, measure the runtime difference for a proposed optimization and look at the Assembly code to verify that the compiler did what you wanted.

Exercise 2: Does it fit in the cache?

In this exercise we will be computing the product of two square matrices using different loop orders. Intuitively, one would write a matrix-matrix product like this:

```
// Base version: row - column - k
// C[rows,columns] = A [rows,k] * B [k,columns]
// Using our own class Matrix for convenience.
void mm_mult_rck(std::size_t size, Matrix<double> &A,
                 Matrix<double> &B, Matrix<double> &C) {
  for (auto row = 0; row < size; ++row) {
    for (auto col = 0; col < size; ++col) {
      double accumulator = 0.0;
      for (auto k = 0; k < size; ++k) {
        accumulator += A(row, k) * B(k, col);
      }
      C(row, col) = accumulator;
    }
  }
}
```

In this case (to which we will refer as "rck"), in the innermost loop we iterate over one column of B. For small matrices, this should not be an issue. But for larger matrices, this will cause a cache miss in every access. However, you see that predicting which order will be faster is complicated.

1. Run the provided code and observe the results. How does the performance change with the size of the matrix?

2. Implement more combinations of the loop orderings, at least (column, k, row) and (row, k, column), simply reordering the `for` loops. Add a call to the provided `testFunction` method for each new implementation in `main()`.

3. Run the code again: do you observe any performance differences between the implementations? Is this what you expected? SOLUTION:
   Look directly at the solution code and run it. Conclusion:

   - For small matrices, we see no difference.
   - For large matrices, cache misses dominate.
   - The fastest version (for large matrices) is the rkc, the slowest is ckr.

4. **Idea:** Use `perf stat` to get more insight on the code. For example, run:

   ```
   perf stat -B -e
   ↪  cache-references,cache-misses,cycles,instructions,branches,faults
   ↪  ./mm_mult
   ```

   You can get a list of other available events to track with `perf list`. See also our starter clues for perf.

Some new concepts

The `chrono` library

We are measuring the performance of our code using the `chrono` standard library. Here is how to measure the time spent in a part of code:

```cpp
#include <chrono>

//...
std::chrono::time_point<std::chrono::system_clock> start;
std::chrono::time_point<std::chrono::system_clock> end;

start = std::chrono::system_clock::now();
//... code measured
end = std::chrono::system_clock::now();

auto elapsed_time =
    std::chrono::duration_cast<std::chrono::milliseconds>(end - start);
std::cout << "Elapsed time:" << elapsed_time.count() << "ms" << std::endl;
```

This technique is useful when we want to measure a specific part of code. For more extensive time profiling, we can use a profiler or other performance tools (see lecture).

General workflow for gprof (not needed in this worksheet):

1. Compile your code with `g++ -pg mycode.cpp -o mycode`
2. Execute your code to generate data (stored in `gmon.out`)
3. Analyze the data with `gprof mycode gmon.out > report.txt`

In any case, make sure to disable the compiler optimizations and get multiple samples to study the performance impact of a code change, avoiding confusion by automatic compiler optimizations.

C++ Core Guidelines

Per.1 Don't optimize without reason

Per.3 Don't optimize something that's not performance critical

Per.5 Don't assume that low-level code is necessarily faster than high-level code

Per.6 Don't make claims about performance without measurements

Per.7 Design to enable optimization

Per.11 Move computation from run time to compile time

Per.19 Access memory predictably

Epilogue

Remember: there are optimizations that the compiler does automatically, and optimizations we can do. Nowadays, compilers are quite smart and they can do a lot by themselves, at least inside one compilation unit. They can often do better than hard-coded, hardware specific optimizations, which may obfuscate the intent of the code. The most important steps we can usually take as programmers are to:

- Choose more efficient algorithms. While an implementation optimization may give us a speed-up of some percentage, choosing a different algorithm may give us a speed-up in orders of magnitude.
- Structuring the data in cache-friendly ways
- Avoid repeatedly reading the same entries from files
- Avoid computations we don't need to do (i.e., pre-computing something we reuse often)
- Avoiding copies we don't need (prefer pass-by-reference over pass-by-value, or prefer `std::string_view` over `std::string`)
- Enabling compiler optimizations