C++: performance matters (in a nutshell)

JANUSZ MAJCHRZAK

Base problem: Get ***n*** Fibonacci numbers

- **Fibonacci numbers** are the numbers in the following integer sequence, called the **Fibonacci sequence**, and characterized by the fact that every number after the first two is the sum of the two preceding ones: $1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ 55,\ 89,\ 144,\ \ldots$
- The sequence F_n of Fibonacci numbers is defined by the recurrence relation: $F_n = F_{n-1} + F_{n-2}$ with seed values: $F_1 = 1,\ F_2 = 1$
- These numbers are often used in finance or many types of simulations including biology, chemistry or physics.

Common implementation 1: Recursive algorithm

In case of recursive Fibonacci implementation the algorithm complexity is **$O(2^n)$.** Computer has to calculate the same numbers many times due to the recursive function call.
To get *48th* Fibonacci sequence number (*4807526976*) it took **38 seconds** on my machine.

Imagine if we need to get a lot of the numbers in the same time…

```cpp
#include <iostream>

unsigned long long fibonacci(const int& n)
{
    if(n <= 1)
        return static_cast<unsigned long long>(n);
    else
        return fibonacci(n - 2) + fibonacci(n - 1);
}


int main(int argc, const char* argv[]) {
    const auto result = ::fibonacci(48);
    std::cout << result << std::endl;
    return EXIT_SUCCESS;
}
```

Common implementation 2: Iteration algorithm

The previous approach was very simple, but wasn't optimal. We can easily designe better algorithm using an iterative approach.
This time calculation of 48th Fibonacci number took only about **0.09 second**.
The algorithm complexity is **O(n)** now.

This method is clearly way better, but the question remains: What if we need to get a lot of big Fibonacci numbers in the same time?

```cpp
#include <iostream>


int main(int argc, const char* argv[]) {
    unsigned long long f, f0 = 0, f1 = 1;

    for (int i = 0; i <= 48; i++) {
        if (i > 1) {
            f = f0 + f1;
            f0 = f1;
            f1 = f;
        } else
            f = static_cast<unsigned long long>(i);
    }

    std::cout << f << std::endl;

    return EXIT_SUCCESS;
}

```

Problem solution: Use threads

The idea is to calculate values on a different thread. This approach may be useful in some cases, but also has it's down sides.

In some systems adding new threads can be forbidden. (Real time applications for example).

If we can make a new thread, we have to remember that this is not a cheap operation.

It is also worth mention CPU thread pool.

```cpp
/* Problem solution: Use threads */

#include <iostream>
#include <thread>
#include <future>

int main(int argc, const char* argv[]) {
    (void)argc;
    (void)argv;

    auto fibonacci_task = [](const int &number) -> uint64_t{

        // NOTE: iterative algorithm
        uint64_t f = 0, f0 = 0, f1 = 1;
        for (int i = 0; i <= number; i++) {
            if (i > 1) {
                f = f0 + f1;
                f0 = f1;
                f1 = f;
            } else
                f = static_cast<uint64_t>(i);
        }

        return f;
    };

    std::packaged_task<uint64_t(const int&)> task(fibonacci_task);
    std::future<uint64_t> f1 = task.get_future();
    std::thread t(std::move(task), 48);
    t.detach();

    std::future<uint64_t> f2 = std::async(std::launch::async, fibonacci_task, 48);

    std::promise<uint64_t> p;
    std::future<uint64_t> f3 = p.get_future();
    std::thread( [&p, &fibonacci_task]{
        const auto result = fibonacci_task(48);
        p.set_value_at_thread_exit(result); }).detach();

    f1.wait();
    f2.wait();
    f3.wait();

    std::cout << f1.get() << std::endl;
    std::cout << f2.get() << std::endl;
    std::cout << f3.get() << std::endl;

    return EXIT_SUCCESS;
}
```

Problem solution: Load pre computed numbers from external file

The idea is to pre calculate the Fibonacci numbers before the application yet started and load them form file. This method is commonly used for some data especially in the applications that requires computing a real big numbers.
The file could contain a list of numbers placed in very specific order. This requires us to take care if anybody hasn't modified the file. The parser will expect only certain pattern, if this pattern has been changed our parser should also be modified.

But this is also not perfect in our use case. We want to get just a few Fibonacci numbers as quick as possible.

In some applications this method is also not possible. For example the application that I've been developing in previous company had very strict requirements and worked in very controlled environment which is common in case of real time applications. We as developers were not allowed for adding anything to the file system.
Please also remember that I/O operations are also not cheap (OS calls). The CPU has to make a lot of operations in order to gain access and read data form disk.

Problem solution: Use compile time

In this method we will also use already computed Fibonacci numbers, but unlike previous solution, we will not use external file. Instead we want these numbers combined with the program executable file.
This solution is very efficient and works very well for such a small data set.
For storing bigger amount of data using external file may be a better solution.
This method makes accessing the Fibonacci number very fast (the access time is immeasurable).

```cpp
#include <iostream>
#include <thread>
#include <future>
#include <array>

template <std::size_t N, std::size_t NN>
struct Fibonacci_Array{
    enum : unsigned long long{ value = Fibonacci_Array<N-1, NN>::value + Fibonacci_Array<N-2, NN>::value };
    static void add_values(std::array<unsigned long long, NN>& v){
        Fibonacci_Array<N-1, NN>::add_values(v);
        v[N-1] = value; }
};

template <std::size_t N>
struct Fibonacci_Array<0, N>{
    enum : unsigned long long{ value = 0 };
    static void add_values(std::array<unsigned long long, N>& v){
        v[0] = value; }
};

template <std::size_t N>
struct Fibonacci_Array<1, N>{
    enum : unsigned long long{ value = 1 };
    static void add_values(std::array<unsigned long long, N>& v){
        Fibonacci_Array<0, N>::add_values(v);
        v[1] = value; }
};

template <std::size_t N>
struct Array{
    static void fill(std::array<unsigned long long, N>& array){
        Fibonacci_Array<N, N>::add_values(array); };
};

int main(int argc, const char* argv[]) {
    const unsigned long long array_size = 48;
    std::array<unsigned long long, array_size> array;
    Array<array_size>::fill(array);

    std::cout << array[47] << std::endl;

    return EXIT_SUCCESS;
}
```

Bibliography

- https://en.wikipedia.org/wiki/Fibonacci_number
- https://www.geeksforgeeks.org/time-complexity-recursive-fibonacci-program/
- https://en.wikipedia.org/wiki/Automatic_vectorization
- https://en.wikipedia.org/wiki/SIMD
- https://www.it.uu.se/edu/course/homepage/hpb/vt12/lab4.pdf
- https://ccache.samba.org
- https://cppcon.org
- https://isocpp.org
- http://www.stroustrup.com/bs_faq2.html#char
- https://www.rust-lang.org/en-US/
- https://www.ponylang.org
- https://youtu.be/zBkNBP00wJE?t=26m56s

Thank You