Multithreaded Summation: Using std::lock_guard

This lesson explains the solution for calculating the sum of a vector problem using std::lock_guard in C++.

WE'LL COVER THE FOLLOWING

You may have already guessed that using a shared variable for the summation with four threads is not optimal; the synchronization overhead will outweigh the performance benefit. Let me show you the numbers. The questions I want to answer are still the same.

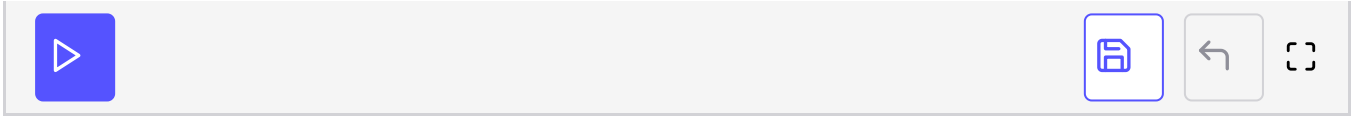
1. What is the difference in performance between the summation using a lock and an atomic?
2. What is the difference in performance between single threaded and multithreaded execution of `std::accumulate`?

The simplest way to make the thread-safe summation is to use a `std::lock_guard`.

Using a `std::lock_guard`

```
// synchronisationWithLock.cpp

#include <chrono>
#include <iostream>
#include <mutex>
#include <random>
#include <thread>
#include <utility>
#include <vector>

constexpr long long size = 100000000;

constexpr long long fir =  25000000;
constexpr long long sec =  50000000;
```

```
constexpr long long thi =  75000000;
constexpr long long fou = 100000000;


std::mutex myMutex;

void sumUp(unsigned long long& sum, const std::vector<int>& val,
           unsigned long long beg, unsigned long long end){
    for (auto it = beg; it < end; ++it){
        std::lock_guard<std::mutex> myLock(myMutex);
        sum += val[it];
    }
}

int main(){

  std::cout << std::endl;

  std::vector<int> randValues;
  randValues.reserve(size);

  std::mt19937 engine;
  std::uniform_int_distribution<> uniformDist(1,10);
  for (long long i = 0 ; i < size ; ++i)
      randValues.push_back(uniformDist(engine));

  unsigned long long sum = 0;
  const auto sta = std::chrono::steady_clock::now();

  std::thread t1(sumUp, std::ref(sum), std::ref(randValues), 0, fir);
  std::thread t2(sumUp, std::ref(sum), std::ref(randValues), fir, sec);
  std::thread t3(sumUp, std::ref(sum), std::ref(randValues), sec, thi);
  std::thread t4(sumUp, std::ref(sum), std::ref(randValues), thi, fou);

  t1.join();
  t2.join();
  t3.join();
  t4.join();

  std::chrono::duration<double> dur= std::chrono::steady_clock::now() - sta;
  std::cout << "Time for addition " << dur.count()
            << " seconds" << std::endl;
  std::cout << "Result: " << sum << std::endl;

  std::cout << std::endl;

}
```



The program is easy to explain. The function `sumUp` (lines 20 - 26) is the work package that each thread executes. `sumUp` gets the summation variable `sum` and the std::vector `val` by reference. Also, `beg` and `end` specify the range of the summation, and the `std::lock_guard` (line 23) is used to protect the shared sum. That being said, each thread (lines 43 - 46) performs a quarter of the summation.

The bottleneck of the program is the shared variable `sum` because it is heavily synchronized by an `std::lock_guard`. With that, one obvious solution comes immediately to mind: *replace the heavyweight lock with a lightweight atomic*.