Multithreaded Summation: Using fetch_add Method with Relaxed Semantic

This lesson explains the solution for calculating the sum of a vector problem using the fetch_add method with relaxed semantic in C++.

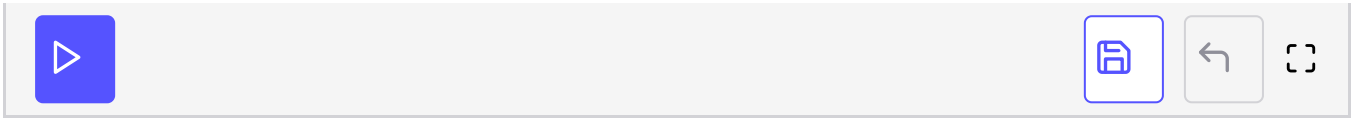
The modification of the source code is minimal. I have only changed the summation expression to `sum.fetch_add(val[it], std::memory_order_relaxed)`.

```
// synchronisationWithFetchAddRelaxed.cpp

...

void sumUp(std::atomic<unsigned long long>& sum, const std::vector<int>& val,
           unsigned long long beg, unsigned long long end){
    for (auto it = beg; it < end; ++it){
        sum.fetch_add(val[it], std::memory_order_relaxed);
    }
}

...
```

Below is the running example of this code:

```
// synchronisationWithFetchAddRelaxed.cpp

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

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;
std::atomic<unsigned long long> sum = {};
```

```
void sumUp(std::atomic<unsigned long long>& sum, const std::vector<int>& val,
           unsigned long long beg, unsigned long long end){
    for (auto it = beg; it < end; ++it){

        sum.fetch_add(val[it], std::memory_order_relaxed);
    }
}

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));


  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 default behavior for atomics is sequential consistency. This statement is true for the addition and assignment of an atomic, and of course for the `fetch_add` method, but we can optimize even more. I adjust the memory model in the summation expression to the relaxed semantic: `sum.fetch_add(val[it], std::memory_order_relaxed)`. The relaxed semantic is the weakest memory model and, therefore, the endpoint of my optimization.

The relaxed semantic is fine in this use-case because we have two guarantees: each addition with `fetch_add` will take place in an atomic fashion, and the threads synchronize with the join calls. Because of the weakest memory

model, we have the best performance.