Thread Local Summation: Using an Atomic Variable with Relaxed Semantic

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

Using an Atomic Variable with Relaxed Semantic

We can do better. I will use relaxed semantic now instead of the default memory model. That’s well defined because the only guarantee we need is that all summations take place and are atomic. With relaxed semantic the summations can even take place out of order.

```
// localVariableAtomicRelaxed.cpp

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

...
```

Below is a running example of this code:

```
// localVariableAtomicRelaxed.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){
      unsigned int long long tmpSum{};

     for (auto i = beg; i < end; ++i){
        tmpSum += val[i];
    }
    sum.fetch_add(tmpSum, 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;

}
```

As expected, it doesn't make any difference whether I use a `std::lock_guard` or an atomic with sequential consistency or relaxed semantic.

Thread-local data is a special kind of local data. Its lifetime is bound to the scope of the thread, not to the scope of the function, such as for the variable `tmpSum` in this example.

We'll examine thread local summation using thread local data afterwards.