Thread Local Summation: Using Local Variable

This lesson explains the solution for calculating the sum of a vector problem using a local variable in C++.

WE'LL COVER THE FOLLOWING

Let's combine the two previous strategies for adding the numbers. I will use four threads and minimize the synchronization between the threads.

There are different ways to minimize the synchronization: local variables, thread-local data, and tasks

Using a Local Variable

Since each thread can use a local summation variable, it can do its job without synchronization; synchronization is only necessary to sum up the local variables. The summation of the local variables is a critical section that must be protected. This can be done in various ways. A quick remark: since only four additions take place, it doesn't matter from a performance perspective which synchronization I use. Anyway, I will use an `std::lock_guard` - an atomic with sequential consistency and relaxed semantic - for the summation.

`std::lock_guard`

```
// localVariable.cpp

#include <mutex>
#include <chrono>
```

```
#include <iostream>
#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){
    unsigned long long tmpSum{};
    for (auto i = beg; i < end; ++i){
        tmpSum += val[i];
    }
    std::lock_guard<std::mutex> lockGuard(myMutex);
    sum += tmpSum;
}

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{};
  const auto sta = std::chrono::system_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();

  const std::chrono::duration<double> dur=
        std::chrono::system_clock::now() - sta;


  std::cout << "Time for addition " << dur.count()
            << " seconds" << std::endl;
  std::cout << "Result: " << sum << std::endl;

  std::cout << std::endl;

}
```

Explanation:

Lines 26 and 27 are the interesting lines; these are the lines where the local summation result `tmpSum` is added to the global summation variable `sum`.

In the next two variations using a local variable, only the function `sumUp` will change; therefore, I will only display the function. For the entire program, please refer to the source files.

In the next lesson, we'll throw some light on thread local summation using an atomic variable with sequential consistency.