Single Threaded Summation: Protection with Atomics

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

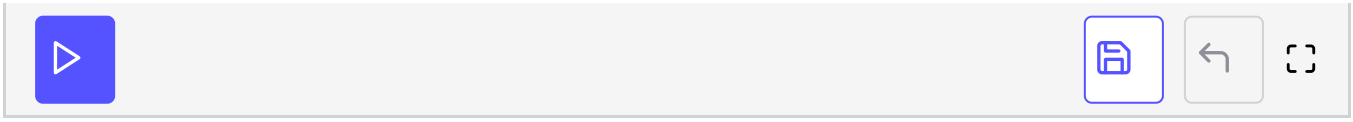
Accordingly, I have the same questions for atomics that I had for locks.

1. How expensive is the synchronization of an atomic?
2. How fast can an atomic be if there is no contention?

I have an additional question: what is the performance difference of an atomic compared to a lock?

```
// calculateWithAtomic.cpp

#include <atomic>
#include <chrono>
#include <iostream>
#include <numeric>
#include <random>
#include <vector>

constexpr long long size = 100000000;

int main(){

  std::cout << std::endl;

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

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

  std::atomic<unsigned long long> sum = {};
  std::cout << std::boolalpha << "sum.is_lock_free(): "
            << sum.is_lock_free() << std::endl;
  std::cout << std::endl;

  auto sta = std::chrono::steady_clock::now();

  for (auto i: randValues) sum += i;

  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;

  sum = 0;
  sta = std::chrono::steady_clock::now();

  for (auto i: randValues) sum.fetch_add(i);

  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;

}
```



First, I check line 28 to see if the atomic `sum` has a lock. That is crucial because, otherwise, there would be no difference between using locks and atomics. On all mainstream platforms I know, atomics are lock-free. Second, I calculate the sum in two ways. I use the `+= operator` in line 33, and the `fetch_add` method in line 45. In the single-threaded case, both variants have comparable performance. However, for `fetch_add` I can explicitly specify the memory model. More about that point in the next subsection.

I want to stress three points:

1. Atomics are 12 - 50 times slower on Linux and Windows than `std::accumulate` without synchronization.
2. Atomics are 2 - 3 times faster on Linux and Windows than locks.
3. `std::accumulate` seems to be highly optimized on Windows.