Single Threaded Summation: Addition with std::accumulate

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

`std::accumulate` is the right way to calculate the sum of a vector. For the sake of simplicity, I will only show the application of `std::accumulate`.

```
// calculateWithStd.cpp

...

const unsigned long long sum = std::accumulate(randValues.begin(),
                                               randValues.end(), 0);

...
```

Here's the above code in action:

```
//calculateWithStd.cpp

#include <chrono>
#include <iostream>
#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));

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

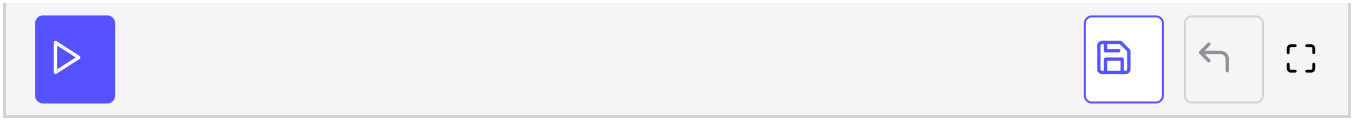
```
  const unsigned long long sum = std::accumulate(randValues.begin(),
                                                 randValues.end(), 0);
   const std::chrono::duration<double> dur =

         std::chrono::steady_clock::now() - sta;

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

   std::cout << std::endl;

}
```



On Linux, the performance of `std::accumulate` is roughly the same as the performance of the range-based for loop. However, using `std::accumulate` on Windows makes a big difference, as its performance is much better than Linux.

Now, let me run two additional single threaded scenarios: one with a lock and the other with an atomic. Why? We get the performance numbers indicating how expensive the protection is - by a lock or an atomic - when there is no contention.