- Solution

This lesson gives the solution to the exercise from the previous lesson.

> **WE'LL COVER THE FOLLOWING** ⌃
>
> - Solution
>   - Explanation

Solution

The calculation of the scalar product can be spread across four asynchronous function calls.

> 💡 The code might take more time than usual.

```cpp
// dotProductAsync.cpp

#include <chrono>
#include <iostream>
#include <future>
#include <numeric>
#include <random>
#include <thread>
#include <vector>

static const int NUM = 100000000;

long long getDotProduct(std::vector<int>& v, std::vector<int>& w){

  auto future1 = std::async([&]{return std::inner_product(&v[0], &v[v.size()/4], &w[0], 0LL);
  auto future2 = std::async([&]{return std::inner_product(&v[v.size()/4], &v[v.size()/2], &w[
  auto future3 = std::async([&]{return std::inner_product(&v[v.size()/2], &v[v.size()*3/4], &
  auto future4 = std::async([&]{return std::inner_product(&v[v.size()*3/4], &v[v.size()], &w[

  return future1.get() + future2.get() + future3.get() + future4.get();
}

int main(){
```

```
std::cout << std::endl;

  // get NUM random numbers from 0 .. 100

  std::random_device seed;

  // generator
  std::mt19937 engine(seed());

  // distribution
  std::uniform_int_distribution<int> dist(0, 100);

  // fill the vectors
  std::vector<int> v, w;
  v.reserve(NUM);
  w.reserve(NUM);
  for (int i=0; i< NUM; ++i){
    v.push_back(dist(engine));
    w.push_back(dist(engine));
  }

  // measure the execution time
  std::chrono::system_clock::time_point start = std::chrono::system_clock::now();
  std::cout << "getDotProduct(v, w): " << getDotProduct(v, w) << std::endl;
  std::chrono::duration<double> dur  = std::chrono::system_clock::now() - start;
  std::cout << "Parallel Execution: "<< dur.count() << std::endl;

  std::cout << std::endl;

}
```

[▷]   [💾] [↩] [⛶]

Explanation

The program uses the functionality of the random and time libraries. Both libraries are part of C++11. The two vectors, `v` and `w`, are created and filled with random numbers in lines 29 - 43. Each of the vectors gets (lines 41 - 43) a hundred million elements. `dist(engine)` in line 42 and 43 generates the random numbers, which are uniformly distributed on the range from 0 to 100. The current calculation of the scalar product takes place in the function `getDotProduct` (lines 13 - 21). `std::async`, internally, uses the Standard Template Library algorithm `std::inner_product`. The return statement sums up the results of the futures.

In the next lesson, we'll discuss `std::packaged_task` which is used to perform a concurrent computation in C++.