std::async

std::async gets a callable as a work package. In this example it's a function, a function object or a lambda function.

// async.cpp

#include <future>

#include <iostream>

#include <string>

std::string helloFunction(const std::string& s){

return "Hello C++11 from " + s + ".";

}

class HelloFunctionObject{

public:

std::string operator()(const std::string& s) const {

return "Hello C++11 from " + s + ".";

}

};

int main(){

std::cout << std::endl;

// future with function

auto futureFunction= std::async(helloFunction,"function");

// future with function object

HelloFunctionObject helloFunctionObject;

auto futureFunctionObject= std::async(helloFunctionObject,"function object");

// future with lambda function

auto futureLambda= std::async([](const std::string& s ){return "Hello C++11 from " + s + ".";},"lambda function");

std::cout << futureFunction.get() << "\n"

<< futureFunctionObject.get() << "\n"

<< futureLambda.get() << std::endl;

std::cout << std::endl;

}

## Eager or lazy evaluation

Eager or lazy evaluation are two orthogonal strategies, to calculate the result of an expression. In case of [eager evaluation](https://en.wikipedia.org/wiki/Eager_evaluation), the expression will immediately be evaluated, in case of [lazy evaluation](https://en.wikipedia.org/wiki/Lazy_evaluation), the expression will only be evaluated, if needed. Often lazy evaluation is called call-by-need. With lazy evaluation you save time and compute power, because there is no evaluation on suspicion. An expression can be a mathematical calculation, a function or a std::async call.

By default, std::async executed immediately its work package. The C++ runtime decides, if the calculation happens in the same or a new thread. With the flag std::launch::async std::async will run it's work package in a new thread. In opposite to that, the flag std::launch::deferred expresses, that std::async runs in the same thread. The execution is in this case lazy. That implies, that the eager evaluations starts immediately, but the lazy evaluation with the policy std::launch::deferred starts, when the future asks for the value with its get call.

The program shows that different behaviour.

// asyncLazy.cpp

#include <chrono>

#include <future>

#include <iostream>

int main(){

std::cout << std::endl;

auto begin= std::chrono::system\_clock::now();

auto asyncLazy=std::async(std::launch::deferred,[]{ return std::chrono::system\_clock::now();});

auto asyncEager=std::async( std::launch::async,[]{ return std::chrono::system\_clock::now();});

std::this\_thread::sleep\_for(std::chrono::seconds(1));

auto lazyStart= asyncLazy.get() - begin;

auto eagerStart= asyncEager.get() - begin;

auto lazyDuration= std::chrono::duration<double>(lazyStart).count();

auto eagerDuration= std::chrono::duration<double>(eagerStart).count();

std::cout << "asyncLazy evaluated after : " << lazyDuration << " seconds." << std::endl;

std::cout << "asyncEager evaluated after: " << eagerDuration << " seconds." << std::endl;

std::cout << std::endl;

}

## A bigger compute job

std::async is quite convenient, to put a bigger compute job on more shoulders. So, the calculation of the scalar product is done in the program with four asynchronous function calls.

// dotProductAsync.cpp

#include <chrono>

#include <iostream>

#include <future>

#include <random>

#include <vector>

#include <numeric>

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[v.size()/4],0LL);});

auto future3= std::async([&]{return std::inner\_product(&v[v.size()/2],&v[v.size()\*3/4],&w[v.size()/2],0LL);});

auto future4= std::async([&]{return std::inner\_product(&v[v.size()\*3/4],&v[v.size()],&w[v.size()\*3/4],0LL);});

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;

}

The program uses the functionality of the random and time library. Both libraries are part of C++11. The two vectors v and w are created and filled with random number in the lines 27 - 43.  Each of the vector gets (line 40 - 43) hundred million elements. dist(engine) in line 41 and 42 generated the random numbers, which are uniform distributed on the range from 0 to 100. The current calculation of the scalar product takes place in the function getDotProduct (line 12 - 20). std::async uses internally the standard template library algorithm [std::inner\_product](http://en.cppreference.com/w/cpp/algorithm/inner_product). The return statement sums up the results of the futures.

But now the question is. How fast is the program, if I executed it on one core? A small modification of the function getDotProduct and we know the truth.

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

return std::inner\_product(v.begin(),v.end(),w.begin(),0LL);

}

### Optimization

But, if I compile the program with maximal optimization level O3 with my GCC, the performance difference is nearly gone. The parallel execution is about 10 percent faster.