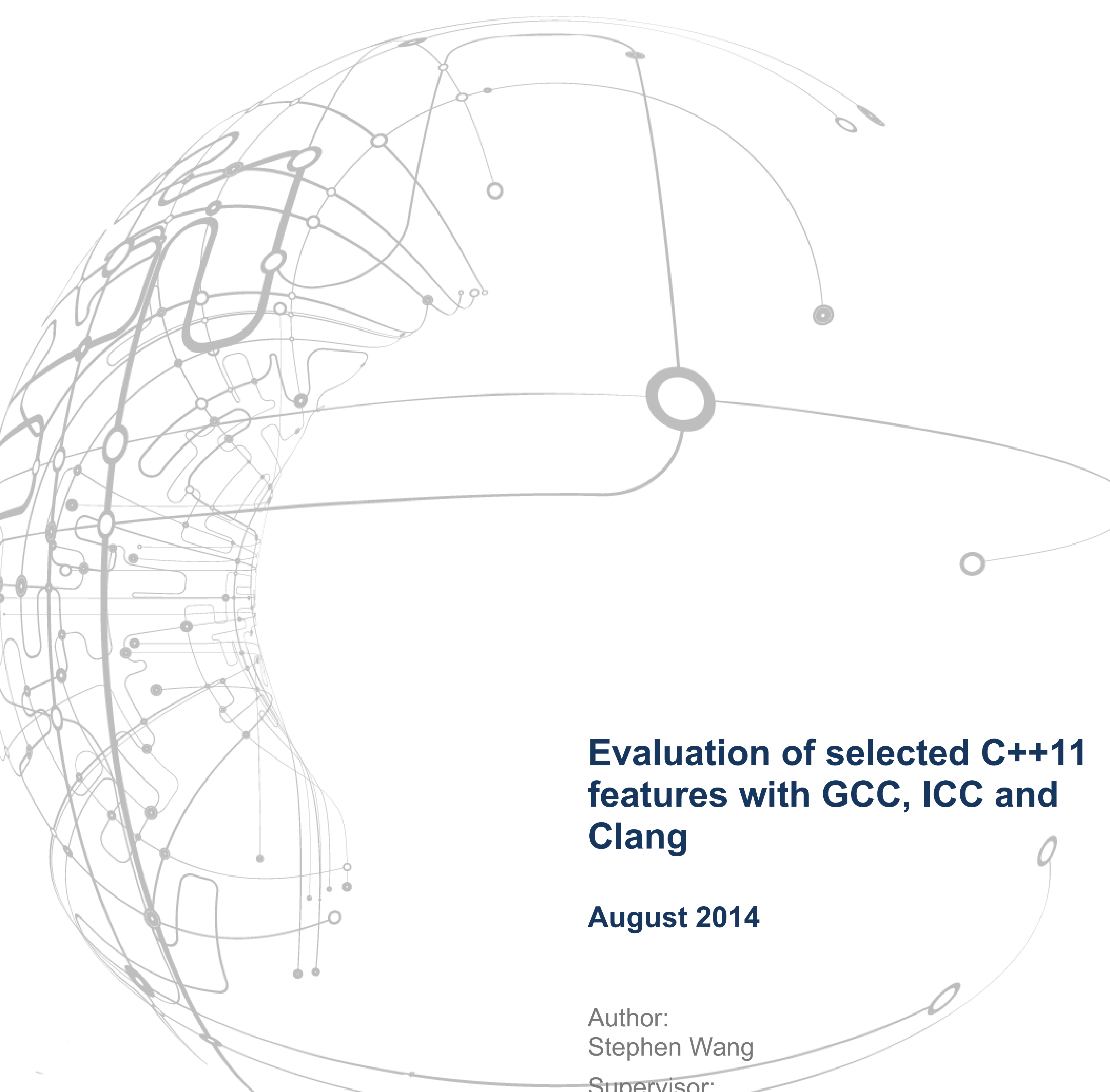

Evaluation of selected C++11 features with GCC, ICC and Clang

August 2014

Author:
Stephen Wang

Supervisor:
Pawel Szostek

CERN openlab Summer Student Report 2014

CERNopenlab

Project Specification

The project concerns various C++11 features - their performance and reliability. The report summarizes the tesults from four micro-benchmarks designed for this project and run with three different compilers (GCC, ICC, Clang) and tries to make an evaluation based on the results.

Abstract

As C++11 gained almost full support by compilers, it is interesting to see whether we can leverage some of the features to improve performance and reliability of C++ code. This work is focused on four selected problems: time measurement techniques, for-loops efficiency, asychronuous tasks and parallel mode of STL algorithms. For each of them a micro-benchmark is made. All the benchmarks are fully automatized to generate results from running binaries compiled by three compilers: GCC, ICC and Clang with -O2, -O3 and -Ofast options. In order to evaluate vectorization and multithreading, profiling tools such as perf and Intel Vtune are used.

Table of Contents

Project Specification....................2

Abstract....................2

Introduction....................3

Platform....................4

Time measurement techniques....................4

Evaluation of various types of for loops....................8

Evaluation of std::async....................13

Evaluation of STL algorithms in parallel mode....................14

Conclusions....................16

Acknowledgement....................17

References....................17

Introduction

C++ was developed by Bjarne Stroustrup starting in 1979 at Bell Labs as an enhancement to the C language and originally named C with Classes. It was renamed C++ in 1983. C++ standard committee takes charge of updating the standard. The previous standard is often referred to as C++98 or C++03, but actually the differences between C++98 and C++03 are not so much.

Nowadays more languages are rising such as Python, which is more flexible and easier to use. More and more people find C++ hard to write and think that its performance does not increase anymore. New features appeared in the new standard of C++, which can make C++ users feel more convenient in development.

C++11 is the most recent version of the standard of the C++ programming language, which was approved by ISO on 12 August. "Surprisingly, C++11 feels like a new language: The pieces just fit together better than they used to and I find a higher-level style of programming more natural than before and as efficient as ever." said by Stroustrup. C++11 actually mainly improves C++ by language usability, mulithreading and other stuff. Lambda expression, auto keyword and ranged for-loop are designed to improve language usability. And it is the first time for C++ to include its own multithreading library so that developer can get out of the trouble like pthread, which makes users not necessary to take care on low-level threading management.

In this report, we will focus on four different topics, out of which each one is related to the C++11 standard. Instead of looking at the code reposponsible for each of them, we adopted a black-box approach. We will be trying to estimate performance of lanugage features by running micro-benchmarks targeting selected areas.

The investigated topics are :

- time measurement methods reliability,
- for-loop efficiency,
- std::async mechanism,
- STL algorithm performance in parallel mode.

Four micro-benchmarks are made for each of the topics, as well as scripts to automatize the whole process. This is supposed to make all the results easily reproducible with different compilers, optimization options, containers and even STL algorithms. Perf and Intel VTune Amplifier are used to understand the behaviour of different code variants.

Benchmarks are compiled with GCC 4.9.0, ICC 15.0.0 and Clang+LLVM 3.4 run with –O2, -O3, -Ofast. We would like to note here that we were fully aware that those flags enable different optimization options when run with each of the compilers. Nevertheless, we didn't try to balance them, as that would be very laborious and not always possible.

Platform

The platform used in the experiments is a dual-socket Intel Jefferson Pass equipped with E5-2695v2 2.4GHz “Ivy Bridge” CPUs with TurboBoost and HyperThreading enabled and 64 GB of DDR3 1666MHz DIMMs. The operating system is SLC6.5 with a 3.11 kernel used instead of the stock one.

Time measurement techniques

Time measurement is the most basic problem to be considered in the evaluation. We tried to asses four different time measurement techniques, which are:

- gettimeofday,
- omp_get_wtime,
- RDTSCP-based
- std::chrono.

The idea behind the benchmarks in this part is to estimate the overhead of the measurement and its precision. Below is a summary of evaluated functions:

Name	Source	Resolution
gettimeofday	sys/time.h	microseconds
omp_get_wtime	omp.h (OpenMP)	nanoseconds
RDTSCP-based	x86 assembly	machine cycles
std::chrono	C++11 standard library	nanoseconds

Table 1: Tested time measurement techniques

RDTSCP is actually an assembly instruction which can access the timestamp register (TSC). The value returned by the function is a 64-bit integer and its resolution are machine cycles. There are many caveats related to code serialization that need to be considered when using this technique. Therefore, we decide to use directly the code presented in [1].

Sample of a measurement using RDTSCP:

```
asm volatile ("CPUID\n\t"
                    "RDTSCP\n\t"
                    "mov %%edx, %0\n\t"
                    "mov %%eax, %1\n\t": "=r" (cycles_high), "=r"
                    (cycles_low):: "%rax", "%rbx", "%rcx", "%rdx");
// put your function here
asm volatile("CPUID\n\t"
                    "RDTSCP\n\t"
                    "mov %%edx, %0\n\t"
```

```
                         "mov %%eax, %1\n\t": "=r" (cycles_high1), "=r"
                         (cycles_low1):: "%rax", "%rbx", "%rcx", "%rdx");
start = ( ((uint64_t)cycles_high << 32) | cycles_low );
end = ( ((uint64_t)cycles_high1 << 32) | cycles_low1 );
uint64_t elapsed = (end - start) / CPU_NOMINAL_FREQ;
```

std::chrono is the new feature in C++11 which aims time measurement. It offers class std::chrono::steady_clock representing a monotonic clock, which means that the time of this clock cannot decrease as physical time moves forward, even if the system clock gets readjusted. According to the specification, we can expect a resolution of nanoseconds.

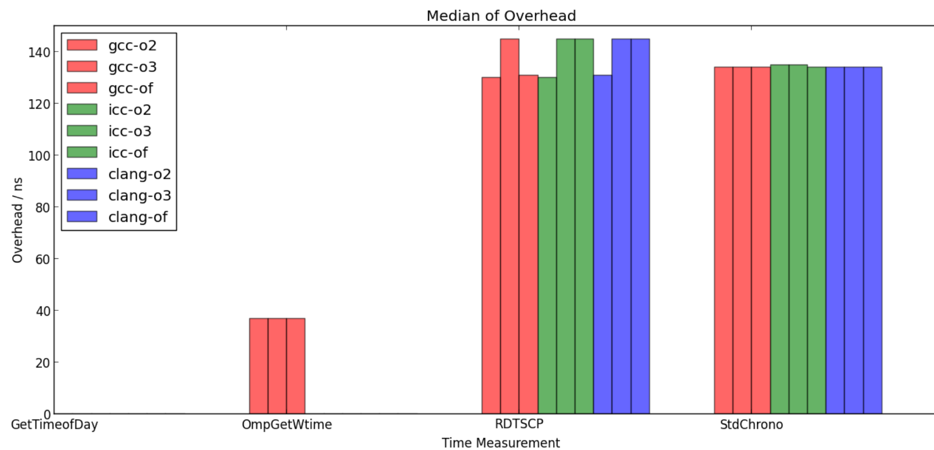

Figure 1: Time measurement median overhead

Overhead is one of the most important factors affecting the realibility of time measurement of small, unrepeated pieces of code. In this experiment we ran a loop whose task was to measure the time that is taken by the function call. As next, we calculated maximun, minimuln, median and variance of all the measurements.

Sample of overhead benchmark:

```
std::vector<int> overhead;
for (i=0; i<OVERHEAD_LOOP_SIZE; i++) {
  auto start = std::chrono::steady_clock::now();
  auto end = std::chrono::steady_clock::now();
   int elapsed = std::chrono::duration_cast<std::chrono::nanoseconds>(end
- start).count();
  overhead.push_back(elapsed);
}
```

The figure shows the maximun overhead of four tested methods

- omp_get_wtime
- gettimeofday
- RDTSCP-based
- std::chrono

Since precision of gettimeofday can not access nanosecond, its median is equal to zero. Surprisingly, omp_get_wtime only can achieve nanosecond resolution onlt when compiled by GCC. The overhead of RDTSCP and std::chrono are similar and the stability achieved by std::chorono is even better than RDTSCP's.

Then real-time benchmark is aimed to check what is the correspondance between the measured time and real-time. To this end, we need an accurate timer to control the real-time consumed. Since sleep is not an accurate enough function, we used select function, which guarantees micro second precision. The time spent by a proces in inactive mode depends on the operating system policy and current system usage.

The code used for making a process sleep :

```
void microseconds_sleep(unsigned long uSec){
    struct timeval tv;
    tv.tv_sec=uSec/1000000;
    tv.tv_usec=uSec%1000000;
    int err;
    do{
        err=select(0,NULL,NULL,NULL,&tv);
    }while(err<0 && errno==EINTR);
}
```

Sample code of real-time benchmark:

```
for (i=10; i<CYCLES_LOOP_SIZE; i++) {
  auto start = std::chrono::steady_clock::now();
  microseconds_sleep(i);
  auto end = std::chrono::steady_clock::now();
   int elapsed = std::chrono::duration_cast<std::chrono::nanoseconds>(end
- start).count();
  cycles_elapsed.insert(std::pair<int, int>(i, elapsed));
}
```

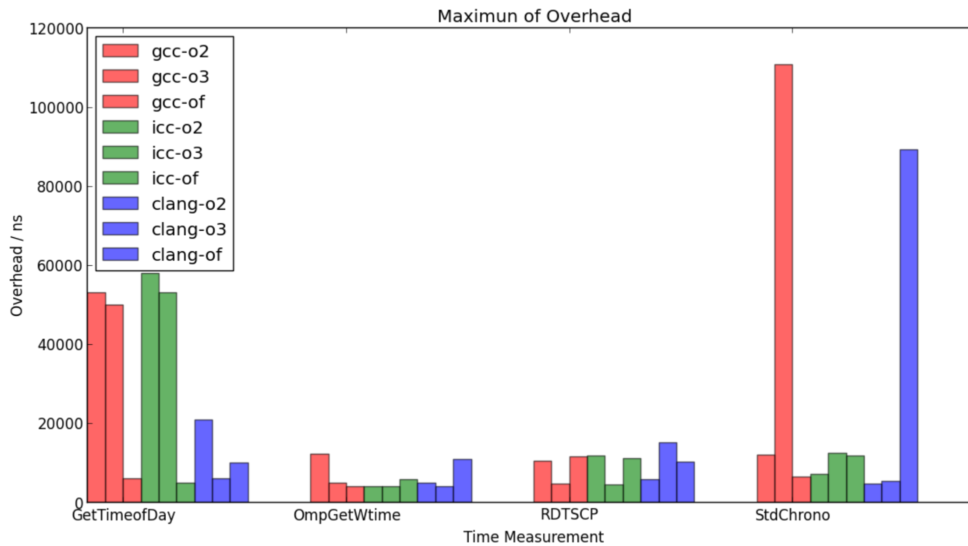

Figure 2: Time measurement maximum overhead

This measurement has been done with the sleeping time ranging from 10 micro seconds to 1 seconds. A plot of The real-time benchmark works from 10 micro-second to 1000 micro-second and the results are shown here.

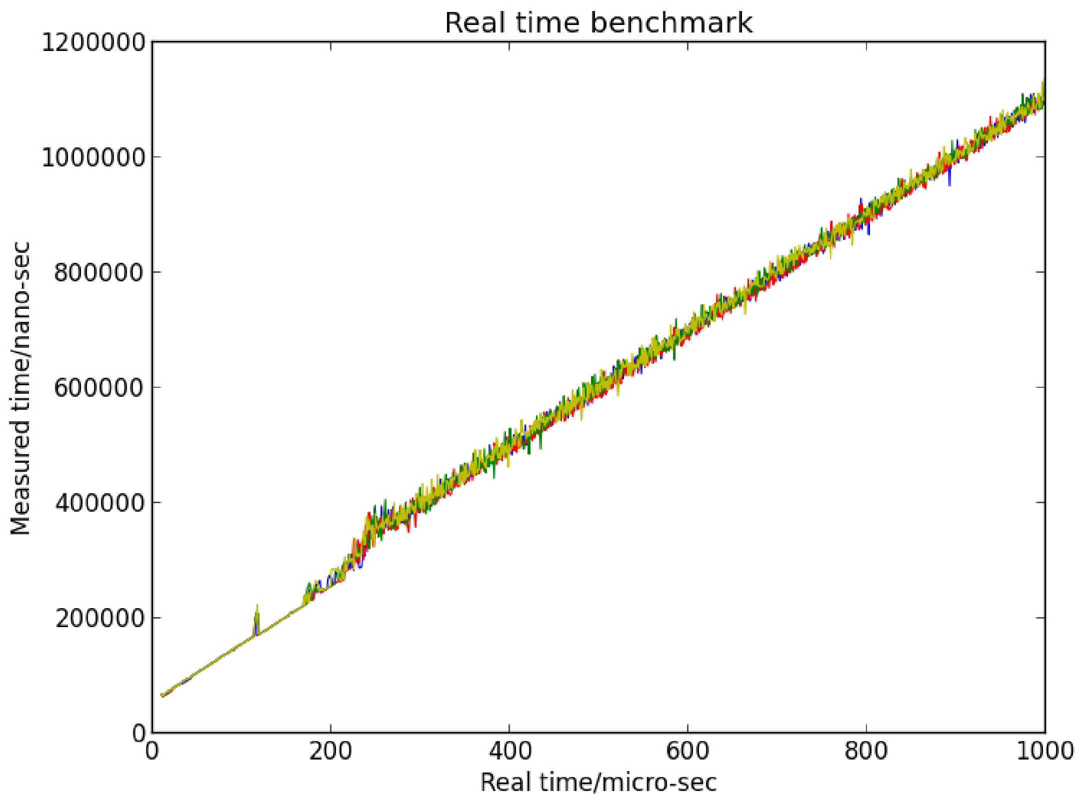

Figure 3: Measured time vs. sleep time

The results are almost the same for these four different time measurement techinques.

Looking at our tests we can say that std::chrono might be less reliable than other methods. Maximum measurement overhead obtained in our experiments was much lower in case of omp_get_wtime and RDTSCP. Thus, in our opinion when doing very fine-grained measurements of small pieces of code without use of loops it is prefereable to use different methods than std::chrono.

Evaluation of various types of for loops

For-loop is a crucial element of programming languagues, concentrating heavy computation in a single block of code. C++11 introduces a range-based variant, which makes the code simpler and

more elegant. In our experiment we wanted to investigate if this and other loop flavours have different performance. To this end, we were iterating over containers of different sizes and types. Different. This was combined with various compilers and optimization options..

The table below contains the data that we gathered in our experiment. We chose following naming convention:

- for_index accesses a container with an integer index,
- for_iterator uses STL's iterators
- for_each uses STL's for_each algorithm
- C++11_range uses C++11's range-based looping

Lambda allows a function to be implemented in the place where it is used. It can also be encapsulated as an object and be passed as a function parameter or used as a class member. Its main idea is very similar to lambdas Python. It helps ‘for_each’ function to be more elegant.

Finally, let’s see what range based for-loop is, which actually is based on iterator and helps user to get out of the complex iterator initialization. We will check whether any optimization is implemented by compilers on it.

Sample of range based for-loop :

```
std::vector<int> vec;
for(auto i : vec){
    std::cout << i << std::endl;
}
```

In micro-benchmark of for-loop effeciency. accumulation is used as workload inside of the iteration. And one note is the result of accumulation should be a temporary variable, since all workloads for these four benchmarks are the same and compilers will optimize the operation to actually ignore the calculation. Sample of micro-benchmark with workload:

```
double for_range(void) {
double sum=0;
for (auto i : v) {
    sum+=i;
}
temp = sum;
return sum;
}
```

Usually for-loop effeciency will be affected a lot by differenet compilers and optimization options. So all the results are listed here to show the big picture of for-loop effeciency.

Container	Syntax	Size	gcc -O2	gcc -O3	gcc- Of
std::vector	for_index	100	95	96	96
	for_iterator	100	81	29	29
	for_each	100	94	29	29

	C++11_range	100	91	29	29
	for_index	10000	**8481**	**8497**	**8500**
	for_iterator	10000	6567	1864	1866
	for_each	10000	6577	1857	1857
	C++11_range	10000	6563	1859	1871
std::array	for_index	100	81	24	24
	for_iterator	100	82	24	24
	for_each	100	81	24	24
	C++11_range	100	82	35	34
	for_index	10000	6500	1521	1540
	for_iterator	10000	6499	1517	1543
	for_each	10000	6496	1528	1539
	C++11_range	10000	6483	2316	2314
C_array	for_index	100	81	24	24
	for_iterator	100	82	34	34
	for_each	100	81	34	37
	C++11_range	100	85	34	34
	for_index	10000	6499	2309	2300
	for_iterator	10000	6500	2304	2313
	for_each	10000	6497	2311	2303
	C++11_range	10000	6492	2320	2314
std::set	for_iterator	100	832	637	641
	for_each	100	628	639	658
	C++11_range	100	635	638	642
	for_iterator	10000	**102082**	91793	93256
	for_each	10000	91750	94276	94978
	C++11_range	10000	91913	91736	92821
std::list	for_iterator	100	180	179	179
	for_each	100	178	181	185
	C++11_range	100	180	179	179
	for_iterator	10000	23702	23594	23556
	for_each	10000	23699	23635	23604
	C++11_range	10000	23716	23603	23559

Container	Syntax	Size	gcc -O2	gcc -O3	gcc -Of
std::vector	for_index	100	26	26	26
	for_iterator	100	20	20	20
	for_each	100	25	25	25
	C++11_range	100	20	20	20
	for_index	10000	1627	1614	1627
	for_iterator	10000	1306	1310	1270
	for_each	10000	1285	1269	1273
	C++11_range	10000	1299	1264	1274
std::array	for_index	100	24	24	24

	for_iterator	100	82	82	82
	for_each	100	53	57	67
	C++11_range	100	54	54	55
	for_index	10000	1252	1326	1301
	for_iterator	10000	**6508**	**6505**	**6498**
	for_each	10000	1270	1248	1199
	C++11_range	10000	1221	1256	1233
C_array	for_index	100	35	34	34
	for_iterator	100	67	67	67
	for_each	100	67	67	67
	C++11_range	100	95	95	97
	for_index	10000	1261	1238	1233
	for_iterator	10000	1239	1225	1205
	for_each	10000	1227	1237	1200
	C++11_range	10000	1235	1198	1199
std::set	for_iterator	100	638	639	638
	for_each	100	678	677	676
	C++11_range	100	628	635	638
	for_iterator	10000	90677	91802	91313
	for_each	10000	91657	93469	91659
	C++11_range	10000	90974	91969	91432
std::list	for_iterator	100	179	179	179
	for_each	100	181	181	181
	C++11_range	100	179	179	179
	for_iterator	10000	23489	23555	23652
	for_each	10000	23494	23559	23651
	C++11_range	10000	23485	23555	23644

Container	Syntax	Size	clang -O2	clang -O3	clang -Of
std::vector	for_index	100	94	94	94
	for_iterator	100	24	24	24
	for_each	100	23	23	23
	C++11_range	100	24	24	24
	for_index	10000	**8489**	**8497**	**8495**
	for_iterator	10000	1261	1263	1271
	for_each	10000	1254	1266	1260
	C++11_range	10000	1257	1276	1255
std::array	for_index	100	25	27	27
	for_iterator	100	34	34	34
	for_each	100	24	27	27
	C++11_range	100	34	37	34
	for_index	10000	1275	1306	1293
	for_iterator	10000	1238	1250	1285
	for_each	10000	1250	1281	1294

	C++11_range	10000	1247	1242	1306
C_array	for_index	100	24	24	24
	for_iterator	100	24	24	24
	for_each	100	24	24	24
	C++11_range	100	24	24	24
	for_index	10000	**8383**	**8379**	**8376**
	for_iterator	10000	1264	1239	1282
	for_each	10000	1256	1250	1284
	C++11_range	10000	1243	1234	1288
std::set	for_iterator	100	**748**	654	641
	for_each	100	**652**	743	746
	C++11_range	100	**652**	644	655
	for_iterator	10000	95843	92702	92106
	for_each	10000	93253	95038	95505
	C++11_range	10000	93193	95432	94004
std::list	for_iterator	100	179	183	185
	for_each	100	179	181	181
	C++11_range	100	179	181	181
	for_iterator	10000	**23700**	**23743**	**23596**
	for_each	10000	**23705**	**23755**	**23589**
	C++11_range	10000	**23688**	**23751**	**23585**

Table 1: Execution times of iteration over containers. Various iteration methods were chosen over a set of conainers such as std::vector, std::array, std::set, std::list and plain C array. We ran the same tests on containers keeping 100 and 10000 integers. Execution time in the array is given in CPU cycles.

This comparison doesn't yield any obvious winner. Direct indexing is the slowest variant in many cases, though. However, when having a closer look at the data, one can see a couple of interesting results:

- using iterator with std::array anf gcc is 5x slower than for_each or based-ranged loop,
- increasing the problem size by a factor of 100x makes iteration per element either faster or slower, depending on the container
- iterating over std::vector is almost as fast as over C array

A related issue is the compiler auto-vectorization for different iteration schemes. We compiled simple loops with -O3 enabled with three compilers. Subsequently, we ran the binaries and checked with perf the number of vector instructions. This allowed us to reason about auto-vectorization in each case, what is shown below.

	C array	std::array	std::list	std::set	std::vector
GCC	Vectorized	Vectorized	Not vectorized	Not vectorized	Vectorized
ICC	Vectorized	Vectorized	Not vectorized	Not vectorized	Vectorized
clang	Vectorized	Not vectorized	Not vectorized	Not vectorized	Vectorized

Table 1: Vectorization of for-loop flavours for different containers and compilers with -O3 flag.

Evaluation of std::async

In C++11 standard std::async is a template function that spawns threads asynchronously. It returns a std::future that will eventually hold the result of that function call. It accepts two execution policies: deffered (lazy evaluation) and async (asyncronous evaluation). In our experiment we used only the latter. We tried to see when a new thread is created and whether threads are reused. For this purpose we employed Intel VTune Amplifier.

Sample benchmark:

```
int len = std::distance(v.begin(), v.end());
int distance = len / thread_num;
double sum = 0;
auto handle1 = std::async(std::launch::async, fsum, v.begin(), v.begin()
+distance);
auto handle2 = std::async(std::launch::async, fsum, v.begin()+distance,
v.begin()+2*distance);
auto handle3 = std::async(std::launch::async, fsum, v.begin()+2*distance,
v.begin()+3*distance);
auto handle4 = std::async(std::launch::async, fsum, v.begin()+3*distance,
v.end());
sum = handle1.get()+handle2.get()+handle3.get()+handle4.get();
return sum;
```

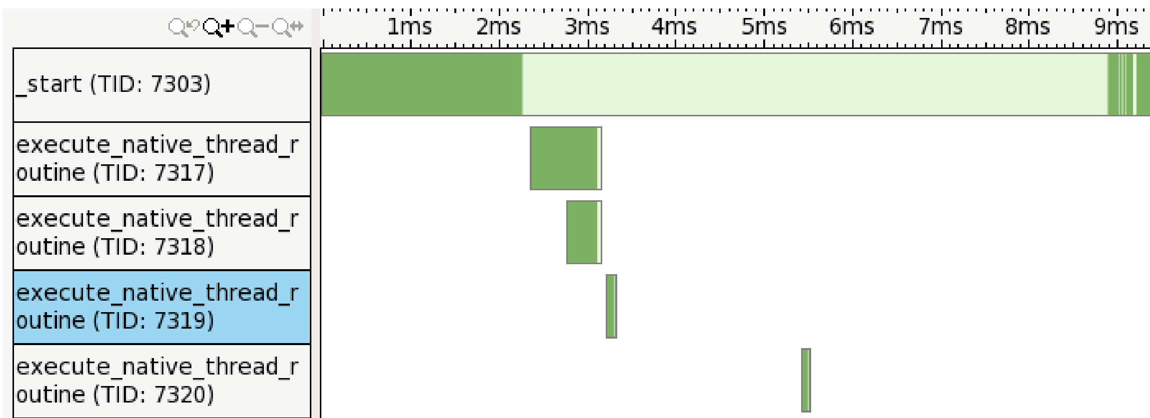

Figure 4: Threads spawning with std::async

In the figure we can see that every call to std::async causes a new thread to be spawned immediately and there is no thread reuse. The computation starts after a thread is created.

Evaluation of STL algorithms in parallel mode

libstdc++'s parallel mode is a an experimental feature of C++ standard library which allows running chosen STL algorithms using all available cores. It comes very handy when we want to run standard operations on standard containers with levereging computing capacity of our hardware. With our benchmarks we can see that even though multi-threading is used, achieved performance is worse than expected.

These parallel mode constructs can be invoked by explicit source declaration or by compiling existing sources with a specific compiler flag.

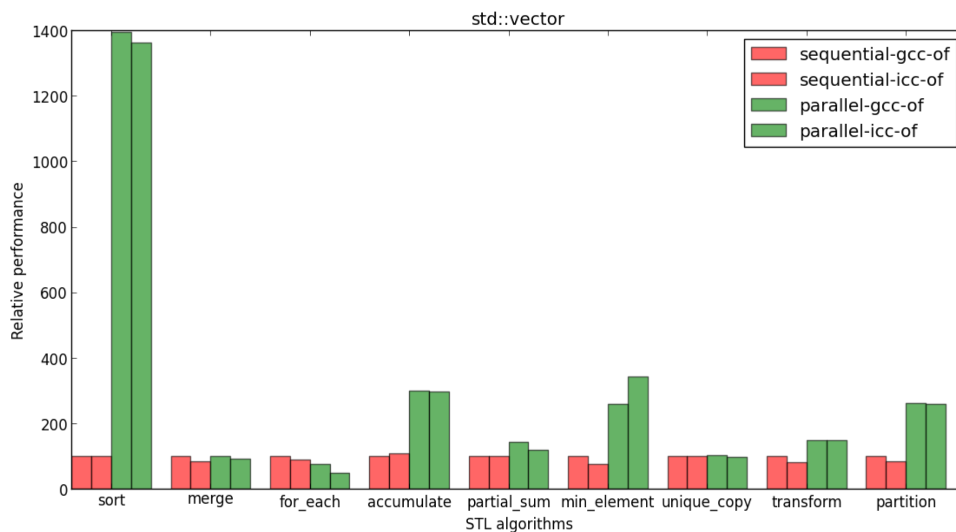

Figure 5: Relative speed-up for STL algorithms for std::vector. Sequential execution of the code compiled with gcc serves as a baseline

Out evaluation costited in comparing the excecution times of STL algorithms in sequential mode and parallel mode. One need to note that not all algorithms are implemented in parallel mode - the guide (https://gcc.gnu.org/onlinedocs/libstdc++/manual/parallel_mode.html) explains which algorithms can be employed.

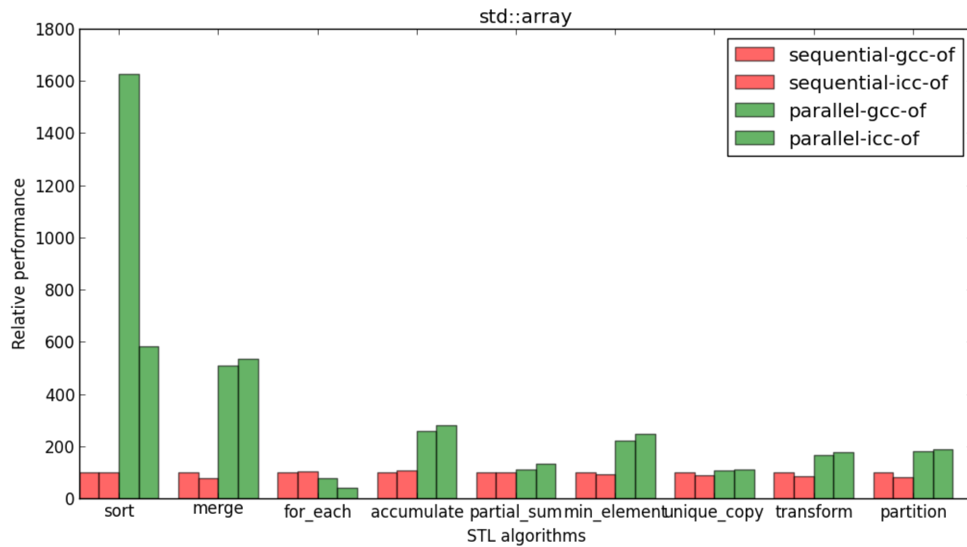

Figure 6: Relative speed-up for STL algorithms for std::array. Sequential execution of the code compiled with gcc serves as a baseline

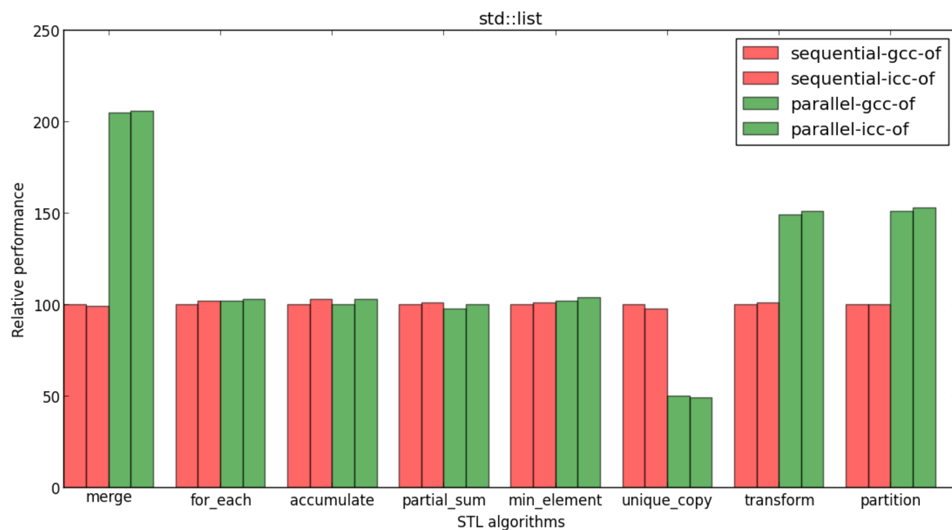

Figure 7: Relative speed-up for STL algorithms for std::list. Sequential execution of the code compiled with gcc serves as a baseline

One obvious conclusion we can get from these results are std::sort is the best accelerated by parallel mode. And std::vector and std::array are two containers which benefit most from the parallel mode, which is related to the data structures that they represent and their implementation.

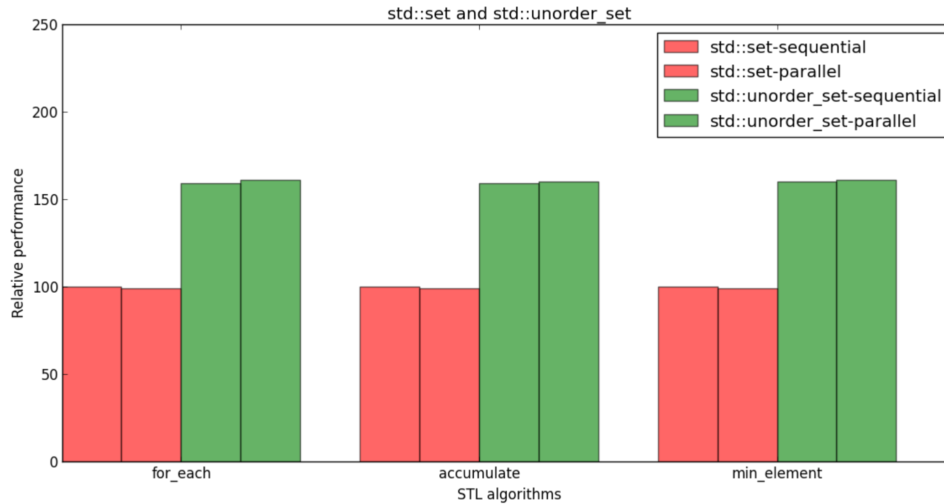

Figure 8: Relative speed-up for STL algorithms for std::set and std::unordered_set. Execution times are gathered from a binary compiled with gcc. Serial execution server as a baseline.

Conclusions

In this project several features from C++11 have been evaluated with GCC, ICC and LLVM. From the results we can see that the ease of use does not result in better code performance. Nevertheless, C++11 includes a handful of new features whose effectivness still might be investigated. Some of them are: tuples, move semantics, efficiency of different random number generators, uniqe_ptr performance, influence of -std=C++11 flag on the C++03 code efficiency. All the benchmarks and results from the report can be found on https://github.com/wangyichao/CPP11_Benchmarks**.**

Acknowledgement

Thanks to my supervisor Pawel Szostek. I learnt a lot from him, which is even more than the project.

Thanks a lot for CERN openlab. It is an amazing platform and gives teenages from all over the world chance to work together. I think it is the bect summer for me.

References

1. Intel Corporation. How to Benchmark Code Execution Times on Intel® IA-32 and IA-64Instruction Set Architectures[EB/OL]. [September 2010]. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf#page=31&zoom=auto,139,694.

2. Bjarne Stroustrup. C++11 - the new ISO C++ standard[EB/OL]. [September 5, 2014]. http://www.stroustrup.com/C++11FAQ.html#std-async.

3. Orion Lawlor. Dr. Lawlor's Quick and Dirty C++11 Benchmark Results[EB/OL]. []. https://www.cs.uaf.edu/~olawlor/2012/c++11/.

4. Alex Allain. Lambda Functions in C++11 - the Definitive Guide[EB/OL]. []. http://www.cprogramming.com/c++11/c++11-lambda-closures.html.