Using C++11 Chrono to time your programs

Document name: "BDiF2015 – TN0004.pdf"
Date/revision: Tue 27th Jan 2015 / Rev A.
Author: Andrew Sheppard, © 2015.

1. Introduction

This technical note is part of the learning materials for the Baruch MFE "Big Data in Finance" (BDiF) course.

The BDiF course puts great emphasis on programs that execute well at scale, meaning that they continue to perform well as the data input grows very large.

But how can you know how well your program is performing both when the data set on which it operates is small and as the data grows to be large? The answer is that you must time your programs; time your I/O, time your algorithms, time your program logic—in fact, time everything!

There are external tools, such as profilers, that can help measure execution times for programs, often in great detail. However, there are very good reasons to build time reporting into your program from the inside, i.e. in your code. Not only does it give you a very selective way to time only the things that you deem important, but can also provide useful information when your program is deployed to a production environment which, as is often the case, does not have any developer tools (such as a profiler) installed.

2. C++11 Chrono

Prior to C++11 there was no easy or portable way to accurately measure the execution time of a piece of program code. It was necessary to use external libraries or machine-specific routines provided by each operating system.

C++11 provides—via the `<chrono>` header file—three clocks:

1. `system_clock`—this is the real-time clock used by the system.
2. `high_resolution_clock`—this is a clock with the shortest tick period possible on the current system.
3. `steady_clock`—this is a monotonic clock that is guaranteed to never be adjusted.

If you want to measure the time taken by a certain piece of code for execution, you should generally use the `steady_clock`, which is a monotonic clock that is

never adjusted by the system. The other two clocks provided by the `<chrono>` header can be occasionally adjusted, so the difference between two consecutive time measurements, one of which follows the other, is not always positive; thereby throwing off the actual order of execution.

3. **Using Chrono to time execution your code**

Here are step-by-step instructions to compiling, running some example programs in C++11 that measure time. In each case, follow these steps:

1. Open a web browser and type in the address http://coliru.stacked-crooked.com/ . This will open the window shown in Figure 1.
2. Cut & paste the code from **Appendix I** into the main panel.
3. Then click on the button labeled "Compile, link and run …". This will run the program and the output will appear in the panel directly below the code.
4. That's it!

You should see output that looks something like this:

```
system_clock
1
1000000000
steady = false

high_resolution_clock
1
1000000000
steady = false

steady_clock
1
1000000000
steady = true
```

Note that `steady_clock` is the only clock on this system that returns `true`.

Now repeat steps 1 through 4 with the code from **Appendix II**. This time you should see some output like this:

```
Estimated value of PI (using 100000000 random samples): 3.14192
calculated in 2003 ms / 2003839040 ns
```

Note that the time in milliseconds (ms) has been truncated, not rounded.

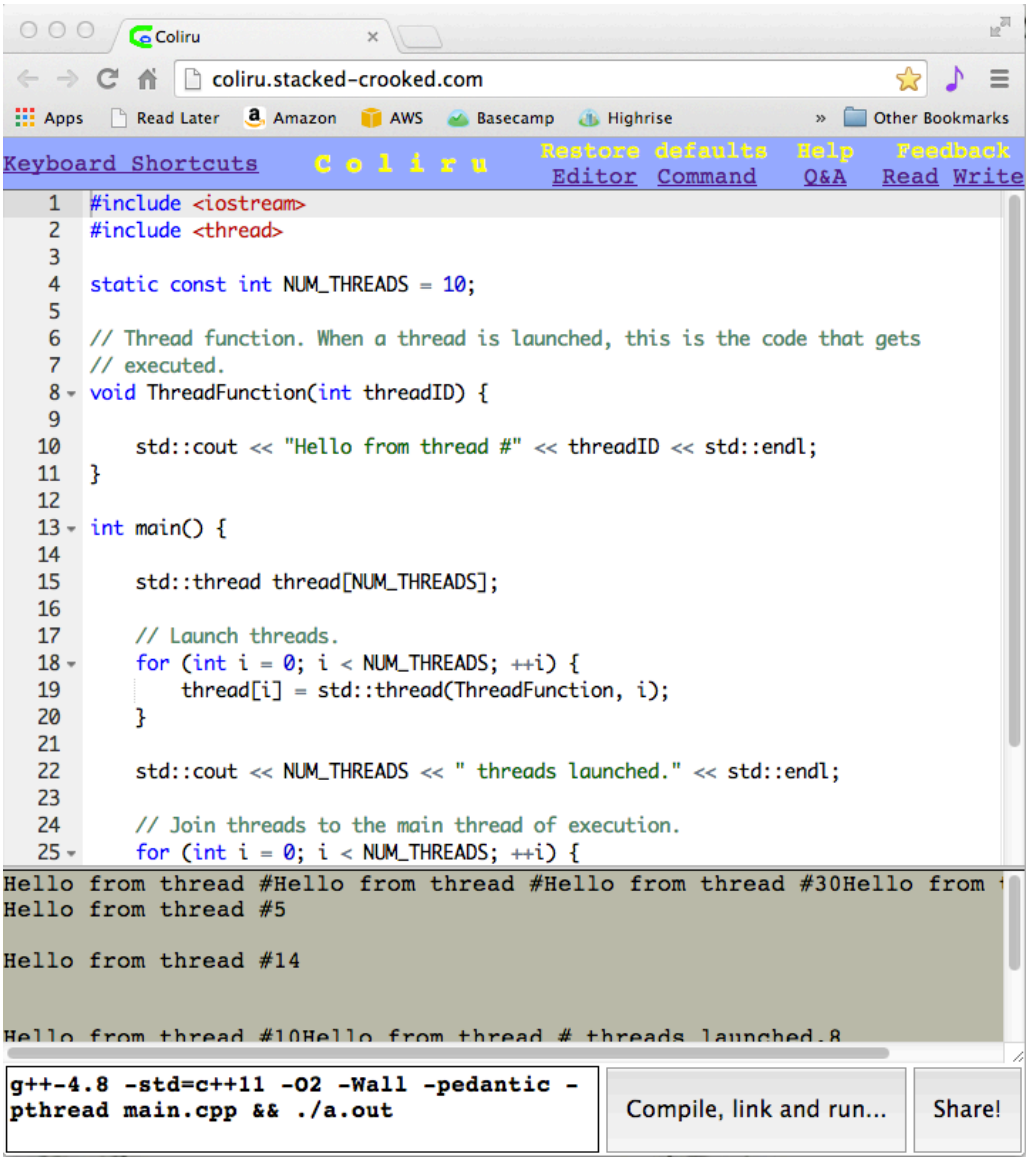

Figure 1. Online C++ compiler "Coliru".

4. **Summary**

Timing your code provides critical feedback on the performance and scalability of your program. You should time not only the overall execution time of your program, but any key parts of your program that are relevant to tracking, and improving, the execution of your program.

Appendix I

Chrono provides functions to check the system clock. The properties of each system's clock vary from machine to machine, so it is often worth testing them. Here's the source code to do that:

_____ SNIP ~ Cut below this line when cutting & pasting _____

```
#include <iostream>
#include <chrono>

using namespace std;

int main(){
    cout << "system_clock" << endl;
    cout << chrono::system_clock::period::num << endl;
    cout << chrono::system_clock::period::den << endl;
    cout << "steady = " << boolalpha << chrono::system_clock::is_steady
<< endl << endl;

    cout << "high_resolution_clock" << endl;
    cout << chrono::high_resolution_clock::period::num << endl;
    cout << chrono::high_resolution_clock::period::den << endl;
    cout << "steady = " << boolalpha <<
chrono::high_resolution_clock::is_steady << endl << endl;

    cout << "steady_clock" << endl;
    cout << chrono::steady_clock::period::num << endl;
    cout << chrono::steady_clock::period::den << endl;
    cout << "steady = " << boolalpha << chrono::steady_clock::is_steady
<< endl << endl;

    return 0;
}
```

_____ SNIP ~ Cut above this line when cutting & pasting _____

Appendix II

To measure the execution time of a piece of code, you can simply sample the clock before and after the code has run. Here's the code to do that:

_____ SNIP ~ Cut below this line when cutting & pasting _____

```
#include <climits>
#include <iostream>
#include <chrono>
#include <random>

// Needed to time things.
#define START_TIMER  std::chrono::system_clock::time_point t0 = \
std::chrono::system_clock::now();
#define END_TIMER    std::chrono::system_clock::time_point t1 = \
std::chrono::system_clock::now();
```

```
#define ELAPSED_TIME_MS \
std::chrono::duration_cast<std::chrono::milliseconds>(t1 - t0).count()
#define ELAPSED_TIME_NS \
std::chrono::duration_cast<std::chrono::nanoseconds>(t1 - t0).count()

const int NUM_SAMPLES = 100000000;

struct Point {

    private:
        std::default_random_engine rng;
        std::uniform_real_distribution<double> uniform;

    public:
        double x;
        double y;

    void next() {
        x = double(rng())/UINT32_MAX;
        y = double(rng())/UINT32_MAX;
    }

    int inside_circle() {
        return (((x-0.5)*(x-0.5)+(y-0.5)*(y-0.5))<0.25) ? 1 : 0;
    }
};

int main()
{
    double pi = 0.0;

    START_TIMER

    Point p;
    int count = 0; // Count of how many darts fall inside/outside.
    for (int n = 0; n<NUM_SAMPLES; n++) {

        // Throw a dart at the unit square!
        p.next();
        count += p.inside_circle();
    }

    // Calculating pi
    pi = 4.0 * count / NUM_SAMPLES;

    END_TIMER

    std::cout << "Estimated value of PI (using " << NUM_SAMPLES;
    std::cout << " random samples): "<< pi;
    std::cout << " calculated in " << ELAPSED_TIME_MS << " ms";
    std::cout << " / " << ELAPSED_TIME_NS << " ns";
    std::cout << std::endl;

      return 0;
}
```

____ SNIP ~ Cut above this line when cutting & pasting ____