Centrum Wiskunde & Informatica

C++: Benchmarking, producing data, using external libraries

WISM454 Laboratory Class Scientific Computing, Jan-Willem Buurlage

March 25, 2019

Today

- Today, we discuss producing and analyzing numerical results
- The `<chrono>` library lets you perform benchmarks
- The `<fstream>` library lets you save data to files
- You can make your life easier by writing *helper classes* to perform timings, or that generate data and reports

<chrono>

- This standard library lets you read the current time
- There are three main concepts in the library:
  - Clocks
  - Time points
  - Durations

Simple example

```
#include <chrono>

auto current_time = std::chrono::system_clock::now();
...
auto some_later_time = std::chrono::system_clock::now();
auto delta_t = some_later_time - current_time;
```

Clocks

- The normal clock is `system_clock`. This corresponds to the time set on your operating system.
  - Typically, this is represented as Unix Time: seconds since 01-01-1970. (This will actually be guaranteed from C++20 onwards)
- If you need to make sure to have a clock that is guaranteed to not change (by e.g. the user of the OS), then you can use `steady_clock`.
- If you need to accurately measure short time intervals, you can use `high_resolution_clock`.

Time points

- These represent a certain point in time.
- Use `time_point::time_since_epoch` to obtain a duration since the reference point of the clock.
- Time points can be compared, useful in e.g. parallel code.
- Most of the time not used directly.

```
auto start = std::chrono::steady_clock::now();
std::cout << "Hello World\n";
auto end = std::chrono::steady_clock::now();
assert(end > start);
```

Durations

- Measuring durations is key to benchmarking.
- Obtained by taking the difference between `time_point`.

```
template<class T, class Period = std::ratio<1>>
class duration;
```

- `T` is a type to represent the number of ticks (can be floating point). `Period` says what each tick corresponds to using some compile time rational number. By default, it is simply seconds.
- To convert, you can use aliases for different ratios, e.g. `std::milli`.

```
auto dt = end - start;
std::cout << std::duration<double, std::milli>(dt)
          << " ms\n";
```

Benchmark helper

- Personally, I use a `timer` helper, that you can use to query durations from its construction.

```
auto t = timer();
... // first phase
auto t1 = t.delta();
... // second phase
auto t2 = t.delta();
... // third phase
auto t3 = t.delta();
auto total_time = t.get();
std::cout << t1 << " + " << t2 << " + " << t3
          << " = " << total_time << "\n";
```

- I let it return milliseconds by default (template argument)

Producing data

- Next, we discuss how to produce and store data.
- Typically, your high-performance C++ program will be responsible for generating data. Most likely, this data is analyzed/presented using some other software.
- For example, your C++ program writes its output to a file, that is than analyzed using e.g. MATLAB, Python or some other application of your choice.

<fstream>

- So far, we have written to `stdout`, and if we wanted to store I imagine you either copy pasted, or maybe used:

  `./your_program > some_file.txt`

- Instead of using `std::cout`, you can output directly to a file.

```
#include <fstream>
auto fout = std::ofstream("some_file.txt");
fout << "Hi!\n";
```

File formats

- You will have to choose how to store your data. It is possible to directly dump your data in a binary format, but you can also choose for more readable formats.
- Typical choices are:
  - CSV
  - JSON
  - XML
- What to use depends on the further analysis. There are external libraries that can help you output e.g. JSON.
- (For plots I usually use CSV, or JSON if it is important to keep metadata around.)

Generating tables

- Often, benchmarking data or results are tabular.
- For my own experiments, I have written a 'report' class, that can store and present tabular data, and output to various formats (CSV, LaTeX and Markdown for example).

```
auto report = table("Matrix statistics");
report.set_columns("m", "n", "nnz", "t_1");

for (auto matrix : matrices) {
    auto t = timer();
    // ... computation with matrix
    auto t1 = timer.delta();
    report.row(matrix.m, matrix.n, matrix.nnz, t1);
}

report.save_csv("results.csv");
report.save_latex("results.tex");
```

External libraries

- Just as someone else might use your RNG library for their software, you can use software written by others as well.
- Examples of libraries that are useful for scientific computing:
  - `Eigen` for numerical linear algebra
  - Boost is a popular collection of peer-reviewed libraries
  - CUDA for GPU programming
  - MPI for distributed programming
  - ...

Obtaining an external library

- A library can be installed/used in various ways
  - Downloading the source.
  - Through a package manager (e.g. your Linux distribution).
  - Including it as a (git) submodule.
- Each method has advantages and disadvantages.

Headers and libraries

- Libraries are interfaced with in two ways:
  - Header files are provided so you can call functions that are part of the library.
  - A compiled version of the library is linked against by your code, so that your program is capable of executing library functions.
- Header-only libraries contain all the implementations in the header, you compile their code as part of your own program. This is common for heavily templated libraries.
- In today's tutorial code, you can find an example of how to use an external JSON library.