Retour d'expérience sur des tests de performances via Google Benchmark

Virgile Dubos [1]

December 12, 2024

[1] POEMS, CNRS, Inria, ENSTA Paris, Institut Polytechnique de Paris, 91120 Palaiseau, France.

Outline

1. Benchmarking in a nutshell
2. Before you get started
3. Google Benchmark: basic examples
4. Google Benchmark: advance features
5. Sources

Benchmarking in a nutshell

- Measurement of code "performance" (execution time, memory usage, throughput, energy consumption, ...).

Benchmarking in a nutshell

- Measurement of code "performance" (execution time, memory usage, throughput, energy consumption, ...).
- Comparison between "equivalent" code:
  - Models (PDEs, empirical/mathematical approximations, ...).
  - Algorithms (direct/iterative solvers, preconditioners, sorting algorithms, ...).
  - Systems (parallel computations, hardware, compilers, ...).
  - Parameters (problem size, number of threads, expected error, ...).
  - Concepts (design patterns, data structures, programming paradigms, ...).

Benchmarking in a nutshell

- Measurement of code "performance" (execution time, memory usage, throughput, energy consumption, ...).
- Comparison between "equivalent" code:
  - Models (PDEs, empirical/mathematical approximations, ...).
  - Algorithms (direct/iterative solvers, preconditioners, sorting algorithms, ...).
  - Systems (parallel computations, hardware, compilers, ...).
  - Parameters (problem size, number of threads, expected error, ...).
  - Concepts (design patterns, data structures, programming paradigms, ...).
- Extent of benchmarking:
  - Microbenchmark $\sim$ unit tests: very specific sections of code (sorting algorithms, matrix multiplication, vector addition, ...), can be easily skewed (beware !!!).
  - Macrobenchmark $\sim$ integration tests: broader sequences, impactful from the user point of view (PDEs solvers, softwares, ...).

Benchmarking in a nutshell

- Measurement of code "performance" (execution time, memory usage, throughput, energy consumption, ...).
- Comparison between "equivalent" code:
  - Models (PDEs, empirical/mathematical approximations, ...).
  - Algorithms (direct/iterative solvers, preconditioners, sorting algorithms, ...).
  - Systems (parallel computations, hardware, compilers, ...).
  - Parameters (problem size, number of threads, expected error, ...).
  - Concepts (design patterns, data structures, programming paradigms, ...).
- Extent of benchmarking:
  - Microbenchmark $\sim$ unit tests: very specific sections of code (sorting algorithms, matrix multiplication, vector addition, ...), can be easily skewed (beware !!!).
  - Macrobenchmark $\sim$ integration tests: broader sequences, impactful from the user point of view (PDEs solvers, softwares, ...).
- Benchmarking $\neq$ profiling (identifying performance bottlenecks), same purpose (optimizing code) complementary approaches.

Benchmarking in a nutshell

- Measurement of code "performance" (execution time, memory usage, throughput, energy consumption, ...).
- Comparison between "equivalent" code:
  - Models (PDEs, empirical/mathematical approximations, ...).
  - Algorithms (direct/iterative solvers, preconditioners, sorting algorithms, ...).
  - Systems (parallel computations, hardware, compilers, ...).
  - Parameters (problem size, number of threads, expected error, ...).
  - Concepts (design patterns, data structures, programming paradigms, ...).
- Extent of benchmarking:
  - Microbenchmark $\sim$ unit tests: very specific sections of code (sorting algorithms, matrix multiplication, vector addition, ...), can be easily skewed (beware !!!).
  - Macrobenchmark $\sim$ integration tests: broader sequences, impactful from the user point of view (PDEs solvers, softwares, ...).
- Benchmarking $\neq$ profiling (identifying performance bottlenecks), same purpose (optimizing code) complementary approaches.
- Benefits: evaluating changes/differences, identifying performance regressions, ensuring compatibility/requirements, debugging.

Before you get started / Conclusion

- Measurement "by hand" in the code.
  - Advantages: easy to implement and debug.
  - Disadvantages: lots of manual work, "uncertainty principle".

Before you get started / Conclusion

- Measurement "by hand" in the code.
  - Advantages: easy to implement and debug.
  - Disadvantages: lots of manual work, "uncertainty principle".
- Benchmarking tools: JMH (Java), BenchmarkDotNet (.NET), **Google Benchmark** (C), ...
  - Advantages: a lot less of manual work, nice looking output.
  - Disadvantages: harder to use and debug, not easy to understand what happen under the hood.

Before you get started / Conclusion

- Measurement "by hand" in the code.
  - Advantages: easy to implement and debug.
  - Disadvantages: lots of manual work, "uncertainty principle".
- Benchmarking tools: JMH (Java), BenchmarkDotNet (.NET), **Google Benchmark** (C), ...
  - Advantages: a lot less of manual work, nice looking output.
  - Disadvantages: harder to use and debug, not easy to understand what happen under the hood.
- Personal recommendation: microbenchmarks with tools, macrobenchmarks from scratch.

Before you get started / Conclusion

- Measurement "by hand" in the code.
  - Advantages: easy to implement and debug.
  - Disadvantages: lots of manual work, "uncertainty principle".
- Benchmarking tools: JMH (Java), BenchmarkDotNet (.NET), **Google Benchmark** (C), ...
  - Advantages: a lot less of manual work, nice looking output.
  - Disadvantages: harder to use and debug, not easy to understand what happen under the hood.
- Personal recommendation: microbenchmarks with tools, macrobenchmarks from scratch.
- Tips for designing benchmarks:
  - Benchmark the lowest level of abstraction possible: easier to maintain, scope of benchmark is better defined.
  - Occasionally benchmark the entire program.
  - Best programming practices: really know what/how you should measure.

Before you get started / Conclusion

- Measurement "by hand" in the code.
  - Advantages: easy to implement and debug.
  - Disadvantages: lots of manual work, "uncertainty principle".
- Benchmarking tools: JMH (Java), BenchmarkDotNet (.NET), **Google Benchmark** (C), ...
  - Advantages: a lot less of manual work, nice looking output.
  - Disadvantages: harder to use and debug, not easy to understand what happen under the hood.
- Personal recommendation: microbenchmarks with tools, macrobenchmarks from scratch.
- Tips for designing benchmarks:
  - Benchmark the lowest level of abstraction possible: easier to maintain, scope of benchmark is better defined.
  - Occasionally benchmark the entire program.
  - Best programming practices: really know what/how you should measure.
- Tips for running benchmarks:
  - Reduce noise by preparing the environment: very OS dependent.
  - Repeat benchmarks: recognize/reduce noise.
  - Beware of branch prediction and other compiler optimizations.

Before you get started / Conclusion

- Measurement "by hand" in the code.
  - Advantages: easy to implement and debug.
  - Disadvantages: lots of manual work, "uncertainty principle".
- Benchmarking tools: JMH (Java), BenchmarkDotNet (.NET), **Google Benchmark** (C), ...
  - Advantages: a lot less of manual work, nice looking output.
  - Disadvantages: harder to use and debug, not easy to understand what happen under the hood.
- Personal recommendation: microbenchmarks with tools, macrobenchmarks from scratch.
- Tips for designing benchmarks:
  - Benchmark the lowest level of abstraction possible: easier to maintain, scope of benchmark is better defined.
  - Occasionally benchmark the entire program.
  - Best programming practices: really know what/how you should measure.
- Tips for running benchmarks:
  - Reduce noise by preparing the environment: very OS dependent.
  - Repeat benchmarks: recognize/reduce noise.
  - Beware of branch prediction and other compiler optimizations.
- Bottom line: make sure you get stable and relevant results.

Focus on reducing noise

- Use a high resolution timer.
- Disable as many processes or services as possible on the target system.
- Reserve CPUs/GPUs, always the same ones if possible.
- Disable frequency scaling, turbo boost and address space randomization (randomize the starting address of the stack).

Bash script on Linux

```bash
# Prepare machine for benchmarking
sudo cpupower frequency-set --governor performance > /dev/null
sudo bash -c "echo␣1␣>␣/sys/devices/system/cpu/intel_pstate/
    no_turbo"
sudo bash -c "echo␣0␣>␣/proc/sys/kernel/randomize_va_space"

# Script
./MyBench #--benchmark_enable_random_interleaving=true

# Restore machine settings
sudo cpupower frequency-set --governor powersave > /dev/null
sudo bash -c "echo␣0␣>␣/sys/devices/system/cpu/intel_pstate/
    no_turbo"
sudo bash -c "echo␣2␣>␣/proc/sys/kernel/randomize_va_space"
```

Focus on reducing noise

- Use a high resolution timer.
- Disable as many processes or services as possible on the target system.
- Reserve CPUs/GPUs, always the same ones if possible.
- Disable frequency scaling, turbo boost and address space randomization (randomize the starting address of the stack).

Bash script on Linux

```
# Prepare machine for benchmarking
sudo cpupower frequency-set --governor performance > /dev/null
sudo bash -c "echo␣1␣>␣/sys/devices/system/cpu/intel_pstate/
    no_turbo"
sudo bash -c "echo␣0␣>␣/proc/sys/kernel/randomize_va_space"

# Script
./MyBench #--benchmark_enable_random_interleaving=true

# Restore machine settings
sudo cpupower frequency-set --governor powersave > /dev/null
sudo bash -c "echo␣0␣>␣/sys/devices/system/cpu/intel_pstate/
    no_turbo"
sudo bash -c "echo␣2␣>␣/proc/sys/kernel/randomize_va_space"
```

Google Benchmark

- A library to benchmark C++ code snippets, similar to unit tests.
- Provides performance metrics (CPU time, wall time, and memory usage).
- Supports a wide range of benchmarking scenarios, from simple function benchmarks to complex, parameterized tests.

Basic example.cpp

```cpp
#include <benchmark/benchmark.h>

static void BM_SomeFunction(benchmark::State& state) {
  // Perform setup here
  for (auto _ : state) {
    // This code gets timed
    SomeFunction();
  }
}
// Register the function as a benchmark
BENCHMARK(BM_SomeFunction);
// Run the benchmark
BENCHMARK_MAIN();
```

Benchmark with one argument

Basic example_v2.cpp

```
#include <benchmark/benchmark.h>
static void BM_SomeFunction(benchmark::State& state) {
  for (auto _ : state) {
    SomeFunction(state.range(0)); // This code gets timed
    state.PauseTiming();
    std::cout<<state.range(0)<<std::endl; // This code doesn't
    state.ResumeTiming();
  }
}
// Register the function as a benchmark
BENCHMARK(BM_SomeFunction)->Arg(10);
BENCHMARK(BM_SomeFunction)->Arg(10)->Arg(20);
BENCHMARK(BM_SomeFunction)->RangeMultiplier(2)
                          ->Range(1 << 10, 1 << 20)
                          ->Name("ToTo");
BENCHMARK(BM_SomeFunction)->DenseRange(0, 1024, 128);
BENCHMARK(BM_SomeFunction)->Arg(10)->Arg(20)
                          ->Threads(16)->Threads(32);
// Run the benchmark
BENCHMARK_MAIN();
```

Benchmark with multiple arguments

Basic example_v3.cpp

```cpp
#include <benchmark/benchmark.h>

static void BM_SomeFunction(benchmark::State& state) {
  // Perform setup here
  for (auto _ : state) {
    // This code gets timed
    SomeFunction(state.range(0), state.range(1));
  }
}
// Register the function as a benchmark
BENCHMARK(BM_SomeFunction)->Args({1<<10, 128});
BENCHMARK(BM_SomeFunction)->Args({1<<10, 128})->Args({1<<20, 256});
BENCHMARK(BM_SomeFunction)
->RangeMultiplier(2)
->Ranges({{min_N, max_N}, {min_eps, max_eps}})
BENCHMARK(BM_SomeFunction)->ArgsProduct({{1<<10, 3<<10}, {60, 80}})
->ArgNames({"N", "ExpEps"})

// Run the benchmark
BENCHMARK_MAIN();
```

GBm example output

Console output

```
Benchmark               Time     CPU      Iter     Bytes/s          Items/s
---------------------------------------------------------------------------------
BM_SetInsert/1024/1     28928    29349    23853    133.097kiB/s     33.274k  items/s
BM_SetInsert/1024/8     32065    32913    21375    949.487kiB/s     237.372k items/s
BM_SetInsert/1024/10    33157    33648    21431    1.13369MiB/s     290.225k items/s
```

Json file

```
  "context": {...},
  "benchmarks": [
    {
      "name": "BM_SetInsert/1024/1",
      "iterations": 23853,
      "real_time": 28928,
      "cpu_time": 29349,
      "bytes_per_second": 133097,
      "items_per_second": 33274
    }, {...}
  ]
```

GBm example output

Console output

```
Benchmark               Time    CPU     Iter    Bytes/s        Items/s
--------------------------------------------------------------------------------
BM_SetInsert/1024/1     28928   29349   23853   133.097kiB/s   33.274k  items/s
BM_SetInsert/1024/8     32065   32913   21375   949.487kiB/s   237.372k items/s
BM_SetInsert/1024/10    33157   33648   21431   1.13369MiB/s   290.225k items/s
```

Json file

```
  "context": {...},
  "benchmarks": [
    {
      "name": "BM_SetInsert/1024/1",
      "iterations": 23853,
      "real_time": 28928,
      "cpu_time": 29349,
      "bytes_per_second": 133097,
      "items_per_second": 33274
    }, {...}
  ]
```

Calculating Asymptotic Complexity (Big O)

The following code will calculate the coefficient for the high-order term in the running time and the normalized root-mean square error of string comparison.

string_compare.cpp

```
static void BM_StringCompare(benchmark::State& state) {
  std::string s1(state.range(0), ’-’);
  std::string s2(state.range(0), ’-’);
  for (auto _ : state) {
    auto comparison_result = s1.compare(s2);
  }
  state.SetComplexityN(state.range(0));
}
BENCHMARK(BM_StringCompare)
    ->RangeMultiplier(2)->Range(1<<10, 1<<18)->Complexity(benchmark
        ::oN);
BENCHMARK(BM_StringCompare)
    ->RangeMultiplier(2)->Range(1<<10, 1<<18)->Complexity();
```

Fixture tests

fixture_tests.cpp

```cpp
class MyFixture : public benchmark::Fixture {
public:
  void SetUp(::benchmark::State& state) {...}
  void TearDown(::benchmark::State& state) {...}
};

// Defines and registers ‘FooTest‘ using the class ‘MyFixture‘.
BENCHMARK_F(MyFixture, FooTest)(benchmark::State& st) {
   for (auto _ : st) {...}
}

// Only defines ‘BarTest‘ using the class ‘MyFixture‘.
BENCHMARK_DEFINE_F(MyFixture, BarTest)(benchmark::State& st) {
   for (auto _ : st) {...}
}

// ‘BarTest‘ is NOT registered.
BENCHMARK_REGISTER_F(MyFixture, BarTest)->Threads(2);
// ‘BarTest‘ is now registered.
```

Custom Counters

UserCountersExample.cpp

```cpp
static void UserCountersExample1(benchmark::State& state) {
  double numFoos = 0, numBars = 0, numBazs = 0;
  for (auto _ : state) {
    // ... count Foo,Bar,Baz events
  }
  state.counters["Foo"] = numFoos;
  state.counters["Bar"] = numBars;
  state.counters["Baz"] = numBazs;
}
```

Console output

```
Benchmark               Time    CPU     Iter    UserCounters
--------------------------------------------------------------------------------
BM_SetInsert/1024/1     28928   29349   23853   Bar=16 Bat=40 Baz=24 Foo=8
BM_SetInsert/1024/8     32065   32913   21375   Bar=2  Bat=5  Baz=3  Foo=102
BM_SetInsert/1024/10    33157   33648   21431   Bar=12 Bat=25 Baz=31 Foo=12
```

Preventing Optimization

- `DoNotOptimize(<expr>)` and `ClobberMemory()` prevent a value or expression from being optimized away by the compiler.
- `DoNotOptimize(<expr>)` forces the result of `<expr>` to be stored in either memory or a register, does not prevent optimizations on `<expr>`.
- `ClobberMemory()` forces the compiler to perform all pending writes to global memory.

PreventingOptimization.cpp

```
static void BM_vector_push_back(benchmark::State& state) {
  for (auto _ : state) {
    std::vector<int> v;
    v.reserve(1);
    auto data = v.data();           // Allow v.data() to be
        clobbered. Pass as non-const
    benchmark::DoNotOptimize(data); // lvalue to avoid undesired
        compiler optimizations
    v.push_back(42);
    benchmark::ClobberMemory(); // Force 42 to be written to memory
  }
}
```

Multithreaded Benchmarks

None of the threads will start until all have reached the start of the benchmark loop, and all will have finished before any thread exits the benchmark loop.

MultithreadedBenchmarks.cpp

```cpp
static void BM_MultiThreaded(benchmark::State& state) {
  if (state.thread_index() == 0) {
    // Setup code here.
  }
  for (auto _ : state) {
    // Run the test as normal.
  }
  if (state.thread_index() == 0) {
    // Teardown code here.
  }
}
BENCHMARK(BM_MultiThreaded)->Threads(2);
```

Setup / Teardown

Global setup/teardown specific to each "benchmark".

SetupTeardown.cpp

```cpp
#include <benchmark/benchmark.h>

static void BM_SomeFunction(benchmark::State& state) {
    ...
}

static void DoSetup(const benchmark::State& state) {
    ...
}

static void DoTeardown(const benchmark::State& state) {
    ...
}

BENCHMARK(BM_SomeFunction)->Arg(1)->Arg(3)->Threads(16)->Threads
    (32)->Setup(DoSetup)->Teardown(DoTeardown);
```

Usefull runtime options and tools

- Running a Subset of Benchmarks : `--benchmark_filter=<regex>`
- Random interleaving : `--benchmark_enable_random_interleaving=true`
- Time unit : `--benchmark_time_unit=<unit>`
- Warmup time : `--benchmark_min_warmup_time=<value>`
- Benchmark repetitions : `--benchmark_repetitions=<value>`
- Minimum benchmark runtime : `--benchmark_min_time=<value>s`
- Compare two benchmarks :

  ```
  compare.py benchmarks <benchmark_baseline>
   <benchmark_contender> [benchmark options]...
  ```
- Compare two different filters of one benchmark :

  ```
  compare.py filters <benchmark> <filter_baseline>
   <filter_contender> [benchmark options]...
  ```
- Compare filter one from benchmark one to filter two from benchmark two :

  ```
  compare.py filters <benchmark_baseline> <filter_baseline>
  <benchmark_contender> <filter_contender> [benchmark options]...
  ```

Conclusion

- Measurement "by hand" in the code.
    - Advantages: easy to implement and debug.
    - Disadvantages: lots of manual work, "uncertainty principle".

Conclusion

- Measurement "by hand" in the code.
  - Advantages: easy to implement and debug.
  - Disadvantages: lots of manual work, "uncertainty principle".
- Benchmarking tools: JMH (Java), BenchmarkDotNet (.NET), **Google Benchmark** (C), ...
  - Advantages: a lot less of manual work, nice looking output.
  - Disadvantages: harder to use and debug, not easy to understand what happen under the hood.

Conclusion

- Measurement "by hand" in the code.
  - Advantages: easy to implement and debug.
  - Disadvantages: lots of manual work, "uncertainty principle".
- Benchmarking tools: JMH (Java), BenchmarkDotNet (.NET), **Google Benchmark** (C), ...
  - Advantages: a lot less of manual work, nice looking output.
  - Disadvantages: harder to use and debug, not easy to understand what happen under the hood.
- Personal recommendation: microbenchmarks with tools, macrobenchmarks from scratch.

Conclusion

- Measurement "by hand" in the code.
  - Advantages: easy to implement and debug.
  - Disadvantages: lots of manual work, "uncertainty principle".
- Benchmarking tools: JMH (Java), BenchmarkDotNet (.NET), **Google Benchmark** (C), ...
  - Advantages: a lot less of manual work, nice looking output.
  - Disadvantages: harder to use and debug, not easy to understand what happen under the hood.
- Personal recommendation: microbenchmarks with tools, macrobenchmarks from scratch.
- Tips for designing benchmarks:
  - Benchmark the lowest level of abstraction possible: easier to maintain, scope of benchmark is better defined.
  - Occasionally benchmark the entire program.
  - Best programming practices: really know what/how you should measure.

Conclusion

- Measurement "by hand" in the code.
  - Advantages: easy to implement and debug.
  - Disadvantages: lots of manual work, "uncertainty principle".
- Benchmarking tools: JMH (Java), BenchmarkDotNet (.NET), **Google Benchmark** (C), ...
  - Advantages: a lot less of manual work, nice looking output.
  - Disadvantages: harder to use and debug, not easy to understand what happen under the hood.
- Personal recommendation: microbenchmarks with tools, macrobenchmarks from scratch.
- Tips for designing benchmarks:
  - Benchmark the lowest level of abstraction possible: easier to maintain, scope of benchmark is better defined.
  - Occasionally benchmark the entire program.
  - Best programming practices: really know what/how you should measure.
- Tips for running benchmarks:
  - Reduce noise by preparing the environment: very OS dependent.
  - Repeat benchmarks: recognize/reduce noise.
  - Beware of branch prediction and other compiler optimizations.

Conclusion

- Measurement "by hand" in the code.
  - Advantages: easy to implement and debug.
  - Disadvantages: lots of manual work, "uncertainty principle".
- Benchmarking tools: JMH (Java), BenchmarkDotNet (.NET), **Google Benchmark** (C), ...
  - Advantages: a lot less of manual work, nice looking output.
  - Disadvantages: harder to use and debug, not easy to understand what happen under the hood.
- Personal recommendation: microbenchmarks with tools, macrobenchmarks from scratch.
- Tips for designing benchmarks:
  - Benchmark the lowest level of abstraction possible: easier to maintain, scope of benchmark is better defined.
  - Occasionally benchmark the entire program.
  - Best programming practices: really know what/how you should measure.
- Tips for running benchmarks:
  - Reduce noise by preparing the environment: very OS dependent.
  - Repeat benchmarks: recognize/reduce noise.
  - Beware of branch prediction and other compiler optimizations.
- Bottom line: make sure you get stable and relevant results.

Sources

- Google Benchmark official user guide: https://github.com/google/benchmark/blob/main/docs/user_guide.md
- Benchmarking tips: https://llvm.org/docs/Benchmarking.html
- Optimizations for C++ multi-threaded programming: https://medium.com/distributed-knowledge/optimizations-for-c-multi-threaded-programs-33284dee5e9c
- Google Benchmark basic guide: https://ccfd.github.io/courses/hpc_lab01.html
- How to benchmark C++ code with Google Benchmark: https://bencher.dev/learn/benchmarking/cpp/google-benchmark/