Performance Analysis in C++

The principle challenge of HPC is *understanding your stack*.

The *first* challenge is *defining correctness*, and asserting on invariants.

The *second* challenge is market competitiveness, which is based principally on usability.

- Adaptive numerical algorithms instead of user parameters
- Progress reporting
- Widely compatible data formats
- Curated algorithms, rather than *choice* of algorithms

The *last* challenge is performance analysis, which is today's topic.

As you optimize code, you will begin to develop intuitions about what C++ code is fast and what is slow

- `std::array` is faster than `std::vector`
- `float` is faster than `double`
- `if` checks break the pipeline
- `std::vector` is faster than `std::list`
- `std::vector::reserve` is faster than `std::vector::push_back`

But tomorrow a smart compiler writer could make all this stuff false.

What does a performance benchmark need to do?

- Call the function repeatedly until statistical confidence is gained about its runtime, and no longer
- Not get optimized out by the compiler
- Test multiple inputs for signs of scaling problems
- Be usable

Use google/benchmark!

The installation is simple:

```
$ git clone https://github.com/google/benchmark.git
$ mkdir build_benchmark; cd build_benchmark
build_benchmark$ cmake -DCMAKE_BUILD_TYPE=Release ../benchmark
build_benchmark$ make -j`nproc`
build_benchmark$ make test
build_benchmark$ sudo make install
```

google/benchmark minimal working example

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

static void BM_Pow(benchmark::State& state)
{
    while (state.KeepRunning())
    {
        auto y = std::pow(1.2, 1.2);
    }
}

BENCHMARK(BM_Pow);

BENCHMARK_MAIN();
```

google/benchmark minimal working example

Build sequence:

```
CXX=clang++

all: run_benchmarks.x run_benchmarks.s

run_benchmarks.x: run_benchmarks.o
    $(CXX) -o $@ $< -lbenchmark -pthread

run_benchmarks.o: run_benchmarks.cpp
    $(CXX) -std=c++14 -O3 -c $< -o $@

run_benchmarks.s: run_benchmarks.cpp
    $(CXX) $(CPPFLAGS) -S -masm=intel $<

clean:
    rm -f *.x *.s *.o
```

google/benchmark minimal working example

Run:

```
./run_benchmarks.x
Run on (4 X 1000 MHz CPU s)
2016-06-23 17:58:41
Benchmark              Time           CPU Iterations
----------------------------------------------------
BM_Pow                 3 ns          3 ns  264837522
```

Stop compiler optimizations

In fact 3 ns seems a bit fast for this operation. The compiler might have (correctly) reasoned that the repeated call to `std::pow` is useless, and optimized it out.

We can generate the assembly of this function via

```
clang++ -std=c++14 -O3 -S -masm=intel run_benchmarks.cpp
```

Stop compiler optimizations

We can see all function calls in the assembly via

```
$ cat run_benchmarks.s | grep 'call' | awk '{print $2}' | xargs c++filt
benchmark::State::KeepRunning()
benchmark::Initialize(int*, char**)
benchmark::RunSpecifiedBenchmarks()
benchmark::State::ResumeTiming()
benchmark::State::PauseTiming()
__assert_fail
operator new(unsigned long)
benchmark::internal::Benchmark::Benchmark(char const*)
benchmark::internal::RegisterBenchmarkInternal(benchmark::internal::Benchmark*)
operator delete(void*)
_Unwind_Resume
```

`std::pow` isn't one of them!

Stop compiler optimizations

- The compiler's goal is to remove all unnecessary operations from your code
- Your goal is to do unnecessary operations to see how long a function call takes

Stop compiler optimizations

This problem is so pervasive that `google/benchmark` has created a function to deal with it: `benchmark::DoNoOptimize`:

```
double y
while (state.KeepRunning()) {
    benchmark::DoNotOptimize(y = std::pow(1.2, 1.2));
}
```

Stop compiler optimizations

The purpose of this is to tell the compiler to *not* optimize out the assignment of `y`.

But `benchmark::DoNotOptimize` can't keep the compiler from evaluating `std::pow(1.2, 1.2)` at compile time.

Stop compiler optimizations

To keep the compiler from evaluating `std::pow(1.2, 1.2)` at compile time, we simply need to ensure that is doesn't *know* what values it needs to evaluate. Here's a solution:

```
std::random_device rd;
std::mt19937 gen(rd());
std::uniform_real_distribution<double> dis(1, 10);
auto s = dis(gen);
auto t = dis(gen);
double y;
while (state.KeepRunning())
{
    benchmark::DoNotOptimize(y = std::pow(s, t));
}
```

Stop compiler optimizations:

However, `benchmark::DoNotOptimize` forces the result to be stored into RAM, which takes time over register storage:

```
template <class Tp>
inline BENCHMARK_ALWAYS_INLINE void DoNotOptimize(Tp const& value) {
    asm volatile("" : "+m" (const_cast<Tp&>(value)));
}
```

Stop compiler optimizations

Even then we might still have to play tricks on the compiler. One of my favorites: Write the result to `/dev/null` outside the loop:

```
double y
while (state.KeepRunning()) {
    benchmark::DoNotOptimize(y = std::pow(s, t));
}
std::ostream cnull(0);
cnull << y;
```

Stop compiler optimizations: Full boilerplate

```
#include <cmath>
#include <ostream>
#include <random>
#include <benchmark/benchmark.h>

static void BM_Pow(benchmark::State& state) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<double> dis(1, 10);
    auto s = dis(gen);
    auto t = dis(gen);
    double y;
    while (state.KeepRunning()) {
        benchmark::DoNotOptimize(y = std::pow(s, t));
    }
    std::ostream cnull(0);
    cnull << y;
}

BENCHMARK(BM_Pow);
BENCHMARK_MAIN();
```

Stop compiler optimizations

Now our timings are more in line with our expectations:

```
Run on (1 X 2300 MHz CPU )
2016-06-24 20:11:40
Benchmark              Time           CPU Iterations
----------------------------------------------------
BM_Pow                80 ns         80 ns    9210526
```

Templated Benchmarks

It's often useful to find out how fast your algorithm is in float, double, and long double precision. Google benchmark supports templates without too much code duplication

Templated Benchmarks

```
template<typename Real>
static void BM_PowTemplate(benchmark::State& state) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<Real> dis(1, 10);
    auto s = dis(gen);
    auto t = dis(gen);
    Real y;
    while (state.KeepRunning()) {
        benchmark::DoNotOptimize(y = std::pow(s, t));
    }
    std::ostream cnull(nullptr);
    cnull << y;
}
BENCHMARK_TEMPLATE(BM_PowTemplate, float);
BENCHMARK_TEMPLATE(BM_PowTemplate, double);
BENCHMARK_TEMPLATE(BM_PowTemplate, long double);
```

Templated Benchmarks

The results are sometimes surprising; for instance double is found to be faster than float:

```
Run on (1 X 2300 MHz CPU )
2016-06-25 00:07:26
Benchmark                                Time           CPU Iterations
----------------------------------------------------------------------
BM_PowTemplate<float>                  136 ns        127 ns    5468750
BM_PowTemplate<double>                  95 ns         94 ns    7000000
BM_PowTemplate<long double>            404 ns        403 ns    1699029
```

View algorithm scaling

Sometimes you need to analysis the scaling properties of your algorithm. Let's try an example with an algorithm with terrible scaling: Recursive Fibonnaci numbers:

```
uint64_t fibr(uint64_t n)
{
  if (n == 0)
    return 0;

  if (n == 1)
    return 1;

  return fibr(n-1)+fibr(n-2);
}
```

View algorithm scaling

Our benchmark code is

```
static void BM_FibRecursive(benchmark::State& state)
{
    uint64_t y;
    while (state.KeepRunning())
    {
      benchmark::DoNotOptimize(y = fib1(state.range_x()));
    }
    std::ostream cnull(nullptr);
    cnull << y;
}
BENCHMARK(BM_FibRecursive)->RangeMultiplier(2)->Range(1, 1<<5);
```

View algorithm scaling

```
BENCHMARK(BM_FibRecursive)->RangeMultiplier(2)->Range(1, 1<<5)
```

will request a benchmark with `state.range_x()` taking values of `[1, 2, 4, 8, 16, 32]`.

View algorithm scaling

```
Run on (1 X 2300 MHz CPU )
2016-06-25 00:38:14
Benchmark                          Time           CPU Iterations
----------------------------------------------------------------
BM_FibRecursive/1                  7 ns           7 ns   83333333
BM_FibRecursive/2                 15 ns          15 ns   72916667
BM_FibRecursive/4                 37 ns          37 ns   17156863
BM_FibRecursive/8                268 ns         268 ns    2868852
BM_FibRecursive/16             13420 ns       13392 ns      64815
BM_FibRecursive/32          24372253 ns    24320000 ns         25
```

View algorithm scaling

Can we empirically determine the asymptotic complexity of the recursive Fibonacci number calculation?

Pretty much . . . !

View algorithm scaling

google/benchmark will try to figure out the algorithmic scaling, if you ask it to. Example:

```
static void BM_FibRecursive(benchmark::State& state)
{
    uint64_t y;
    while (state.KeepRunning())
    {
      benchmark::DoNotOptimize(y = fib1(state.range_x()));
    }
    std::ostream cnull(nullptr);
    cnull << y;
    state.SetComplexityN(state.range_x());
}

BENCHMARK(BM_FibRecursive)->RangeMultiplier(2)->Range(1, 1<<5)->Complexity();
```

The result?

```
BM_FibRecursive/1                    9 ns          9 ns   72916667
BM_FibRecursive/2                   19 ns         19 ns   44871795
BM_FibRecursive/4                   42 ns         43 ns   14112903
BM_FibRecursive/8                  270 ns        268 ns    2611940
BM_FibRecursive/16               11305 ns      11264 ns      54687
BM_FibRecursive/32            27569038 ns   27555556 ns         27
BM_FibRecursive_BigO            828.24 N^3     827.83 N^3
BM_FibRecursive_RMS                 31 %          31 %
```

It erroneously labels the algorithm as cubic; though we can see the fit is not tight.

Passing a lambda to `Complexity()`

If google/benchmark only tries to fit to the most common complexity classes. You are free to specify the asymptotic complexity yourself, and have google/benchmark determine goodness of fit.

The complexity of the recursive Fibonacci algorithm is ϕ^n, where $\phi := (1+\sqrt{5})/2$ is the golden ratio, so let's use ϕ^n as a lambda . . .

Passing a lambda to `Complexity()`

Syntax:

```
BENCHMARK(BM_FibRecursive)
    ->RangeMultiplier(2)
        ->Range(1, 1<<5)
            ->Complexity([](int n) {return std::pow((1+std::sqrt(5))/2, n);});
```

Result:

```
BM_FibRecursive/1               9 ns              9 ns   79545455
BM_FibRecursive/2              19 ns             19 ns   44871795
BM_FibRecursive/4              37 ns             37 ns   20588235
BM_FibRecursive/8             228 ns            228 ns    3125000
BM_FibRecursive/16           9944 ns           9943 ns      60345
BM_FibRecursive/32       23910141 ns       23866667 ns         30
BM_FibRecursive_BigO         4.91 f(N)         4.90 f(N)
BM_FibRecursive_RMS             0 %               0 %
```

(*Note*: All lambdas are denotes as `f(N)`, so you have to know what you wrote in your source code to understand the scaling.)

Other tricks:

Standard complexity classes don't need to be passed as lambdas:

```
Complexity(benchmark::o1);
Complexity(benchmark::oLogN);
Complexity(benchmark::oN);
Complexity(benchmark::oNLogN);
Complexity(benchmark::oNSquared);
Complexity(benchmark::oNCubed);
```

Benchmarking data transfer

To get data about data transfer rates, use `state.SetBytesProcessed`[1]:

```
static void BM_memcpy(benchmark::State& state) {
  char* src = new char[state.range_x()]; char* dst = new char[state.range_x()];
  memset(src, 'x', state.range_x());
  while (state.KeepRunning()) {
    memcpy(dst, src, state.range_x());
  }
  state.SetBytesProcessed(int64_t(state.iterations()) *
                          int64_t(state.range_x()));
  delete[] src; delete[] dst;
}
BENCHMARK(BM_memcpy)->Range(8, 8<<10);
```

[1] Taken directly from the google/benchmark docs.

Benchmarking data transfer

`state.SetBytesProcessed` adds another column to the output:

```
Run on (1 X 2300 MHz CPU )
2016-06-25 19:25:23
Benchmark                Time           CPU Iterations
------------------------------------------------------
BM_memcpy/8              9 ns          9 ns   87500000   883.032MB/s
BM_memcpy/16             9 ns          9 ns   79545455   1.70305GB/s
BM_memcpy/32             9 ns          8 ns   79545455   3.50686GB/s
BM_memcpy/64            11 ns         11 ns   62500000   5.35243GB/s
BM_memcpy/128           13 ns         13 ns   53030303   8.97969GB/s
BM_memcpy/256           16 ns         16 ns   44871795   15.1964GB/s
BM_memcpy/512           19 ns         20 ns   36458333   24.4167GB/s
BM_memcpy/1024          25 ns         25 ns   28225806   38.6756GB/s
BM_memcpy/2k            50 ns         50 ns   10000000    38.147GB/s
BM_memcpy/4k           122 ns        122 ns    5833333   31.2534GB/s
BM_memcpy/8k           220 ns        220 ns    3240741    34.726GB/s
```

Running a subset of your benchmarks:

Each benchmark takes ~1 second to run. If you need to analyze only one of the benchmarks, use the `--benchmark_filter` option:

```
./run_benchmarks.x --benchmark_filter=BM_memcpy/32
Run on (1 X 2300 MHz CPU )
2016-06-25 19:34:24
Benchmark              Time           CPU Iterations
----------------------------------------------------
BM_memcpy/32          11 ns         11 ns   79545455   2.76944GB/s
BM_memcpy/32k       2181 ns       2185 ns     324074   13.9689GB/s
BM_memcpy/32          12 ns         12 ns   54687500   2.46942GB/s
BM_memcpy/32k       1834 ns       1837 ns     357143   16.6145GB/s
```

Error bars

In general you will get a pretty good idea about the standard deviation of the measurements just by running it a few times. However, if you want error bars, just specify the number of times you want your benchmark repeated:

```
BENCHMARK(BM_Pow)->Repetitions(12);
```

Error bars

```
Run on (1 X 2300 MHz CPU )
2016-06-25 19:57:40
Benchmark                          Time           CPU Iterations
----------------------------------------------------------------
BM_Pow/repeats:12                 70 ns         70 ns   10294118
BM_Pow/repeats:12                 73 ns         73 ns   10294118
BM_Pow/repeats:12                 71 ns         71 ns   10294118
BM_Pow/repeats:12                 70 ns         70 ns   10294118
BM_Pow/repeats:12                 71 ns         71 ns   10294118
BM_Pow/repeats:12                 72 ns         72 ns   10294118
BM_Pow/repeats:12                 73 ns         73 ns   10294118
BM_Pow/repeats:12                 71 ns         71 ns   10294118
BM_Pow/repeats:12                 70 ns         70 ns   10294118
BM_Pow/repeats:12                 73 ns         73 ns   10294118
BM_Pow/repeats:12                 71 ns         71 ns   10294118
BM_Pow/repeats:12                 76 ns         75 ns   10294118
BM_Pow/repeats:12_mean            72 ns         72 ns   10294118
BM_Pow/repeats:12_stddev           2 ns          2 ns          0
```