3-execution

June 3, 2024

1 Profiling execution

1.1 Globally, with GNU `time` command

- As seen previously, it is easy to get a global execution analysis with GNU time.
- If you want to focus on a given subset of a bigger code, you can try to setup a small demonstration program.

1.1.1 Exercice

With GNU time or hyperfine, compare the execution time of those implementations:

[1]:
```
%%file tmp.fibo1.cpp

#include <iostream>

constexpr int fibonacci( int n ) {
  if (n>1) return fibonacci(n-1) + fibonacci(n-2) ;
  else return n ;
}

int main() {
  constexpr int res { fibonacci(36) } ;
  std::cout<<res<<std::endl ;
  return 0 ;
}
```

```
Overwriting tmp.fibo1.cpp
```

[2]:
```
!rm -f tmp.fibo1.exe
```

[2]:
```
!g++ -O2 -std=c++17 tmp.fibo1.cpp -o tmp.fibo1.exe
```

[3]:
```
!./tmp.fibo1.exe
```

```
14930352
```

[5]:
```
%%file tmp.fibo2.cpp

#include <iostream>
```

```
int fibonacci( int n ) {
  if (n>1) return fibonacci(n-1) + fibonacci(n-2) ;
  else return n ;
}

int main() {
  int res { fibonacci(36) } ;
  std::cout<<res<<std::endl ;
  return 0 ;
}
```

```
Overwriting tmp.fibo2.cpp
```

[4]:
```
!rm -f tmp.fibo2.exe
```

[4]:
```
!g++ -O2 -std=c++17 tmp.fibo2.cpp -o tmp.fibo2.exe
```

[5]:
```
!./tmp.fibo2.exe
```

```
14930352
```

1.2 Comparing alternative implementations

If you do not care about the absolute compilation time, but want to compare two (or more) alternative small implementations, you can try QuickBench.

Basically, in the QuickBench main window, for each code to be compared, add a section like this…

[ ]:
```
static void CodeVersion1(benchmark::State& state) {

  // code to be executed once
  // ...

  for (auto _ : state) {

    // code to be measured repeatedly
    // ...

    // output variable should be protected so that
    //  the compiler does not optimize them out
    benchmark::DoNotOptimize(res) ;

  }

}
BENCHMARK(CodeVersion1);
```

1.2.1 Exercice

Compare with QuickBench the two previous implementations of fibonacci, from exercice 1.

1.3 Internally, with `std::chrono`

When trying to speed-up a big real world application, one needs to known the execution time of the subparts of the application, so to focus his efforts where it is worth. **It is highly advised to learn how to use a profiling tool such as `perf`.**

Meanwhile, some internal information can be collected thanks to `std::chrono`. One of the interesting aspects of `std::chrono` is the work done on time units. Below, we measure the execution time of `generate` and displays it in milliseconds.

[17]:
```
%%file tmp.chrono.cpp

#include <iostream>
#include <cstdlib>
#include <cassert>
#include <valarray>
#include <chrono>

std::valarray<double> generate( int size )
 {
  using namespace std::chrono ;
  auto t1 { steady_clock::now() } ;

  std::valarray<double> data(size) ;
  for ( double & value : data ) {
    value = std::rand()/(RAND_MAX+1.) ;
  }

  auto t2 { steady_clock::now() } ;
  auto dt { duration_cast<microseconds>(t2-t1).count() } ;
  std::cout<<"(generate: "<<dt<<" us)"<<std::endl ;

  return data ;
 }

double analyse( std::valarray<double> const & data, int power )
 {
  double res = 0 ;
  for ( double value : data ) {
    double prod = 1 ;
    for ( int j=0 ; j<power ; ++j ) {
      prod *= value ;
    }
    res += prod ;
   }
  return res ;
 }

int main( int argc, char * argv[] ) {
  assert(argc==3) ;
  int size {atoi(argv[1])} ;
  int power {atoi(argv[2])} ;

  auto data = generate(size) ;
  std::cout << analyse(data,power) << std::endl ;
}
```

```
Overwriting tmp.chrono.cpp
```

[15]:
```
!rm -f tmp.chrono.exe && g++ -std=c++17 -I./ tmp.chrono.cpp -o tmp.chrono.exe
```

[16]:
```
!./tmp.chrono.exe 1024 100000
```

```
(generate: 19 us)
0.525744
```

2 Questions ?

3 Exercise

The code below defines some kind of “high-order function” `time()`, which takes as input another function `f`, and a set of arguments to be used for a call to `f`. 1. You are asked to complete the definition of `time`, with `chrono` features, so to compute, display and compare the execution time of `analyse1` and `analyse2`. 2. Try to write a Python script which will run the program 10 times, and compare the mean execution time of `generate`, `analyse1` and `analyse2`… 3. …And/or try with QuickBench.

[ ]:
```
%%file tmp.chrono.cpp

#include <valarray>
#include <cstdlib>
#include <cassert>
#include <iostream>
#include <string_view>
#include <chrono>

template< typename Fonction, typename... ArgTypes >
auto time( std::string_view title, Fonction f, ArgTypes... args )
 {
  // COMPLETE HERE
  auto res {f(args...)} ;
  // COMPLETE HERE
  std::cout<<"("<<title<<" time: ?? us)"<<std::endl ;
  return res ;
 }

std::valarray<double> generate( int size )
 {
  std::valarray<double> data(size) ;
  for ( double & value : data ) {
    value = std::rand()/(RAND_MAX+1.) ;
  }
  return data ;
 }

double analyse1( std::valarray<double> const & data, int power )
 {
  double res = 0 ;
  for ( double value : data ) {
    double prod = 1 ;
    for ( int j=0 ; j<power ; ++j ) {
      prod *= value ;
    }
    res += prod ;
   }
  return res ;
 }

double analyse2( std::valarray<double> const & data, int power )
 {
  std::valarray<double> values(1.,data.size()) ;
  for ( int j=0 ; j<power ; ++j ) {
    values *= data ;
  }
  double res = 0 ;
  for ( double value : values ) {
    res += value ;
  }
  return res ;
 }

int main( int argc, char * argv[] ) {
  assert(argc==3) ;
  int size {atoi(argv[1])} ;
  int power {atoi(argv[2])} ;

  auto datas = time("gen",generate,size) ;
  auto res1 = time("ana1",analyse1,datas,power) ;
  auto res2 = time("ana2",analyse2,datas,power) ;
  std::cout << res1 << " " << res2 << std::endl ;
```

```
}
```

[ ]:
```
!rm -f tmp.chrono.exe && g++ -O3 -std=c++17 -I./ tmp.chrono.cpp -o tmp.chrono.
↪exe
```

[ ]:
```
!./tmp.chrono.exe 1024 100000
```

4 Resources

- Chrono
- QuickBench
- Google Benchmark
- Perf.
- TP Perf