C++ Benchmarking

Platformski nezavisno programiranje

Mentor: doc. dr. sc. Jerko Škifić

Sveučilište u Rijeci, Tehnički fakultet

smjer računarstvo

Nino Križanec, 0069055799

Rijeka, 30.08.2016

Sadržaj

1 Uvod

Što je to Benchmarking? Benchmarking je proces prikupljanja, analize, obrade i zaključivanja na temelju dobivenih podataka. Benchmarking kompjuterskog koda nešto je drugačija disciplina pošto ju nije lagano kvantificirati kao neke druge stvari.

U ovome ćemo se izvješću baviti benchmarkom C++ koda, preciznije neovisno o platformi ćemo izraditi alat koji će biti u mogućnosti mjeriti performanse dijelova C++ koda.

Benchmarking se u kodu najčešće izvodi kako bi poboljšali performanse, to naravno uključuje potrošnju CPU vremena, potrošnju memorije, bandwitha i naravno s uvođenjem .

1.1 Zašto želimo raditi benchmark koda?

Kako bismo poboljšali naš kod, tj povećali njegovu efikasnost, moramo znati koliko je on zapravo efikasan, te mijenjaju li naše promjene koda njegovu efikasnost na bolje ili lošije. Efikasnost programa možemo kvantificirati jednostavno, vremenom i resursima utrošenim da bi program izvršio zadatak.

U modernom svijetu važnost efikasnog koda prenosi se ne samo na potrošnju električne energije u data centrima, super računalima pa i običnim stolnim računalima, danas je svijet okružen mobilnom tehnologijom koja najčešće crpi električnu energiju iz baterija. Sve se svodi na to da benchmarkom i profiliranjem našeg koda možemo napraviti kod efikasnijim i smanjiti vrijeme koje procesor mora raditi, te tako produljiti i trajanje baterija kod mobilnih uređaja.

Sada, posebno, potrebno je raditi benchmark koda svakog jezika, ali posebno je važno u C++ jeziku iz razloga što nam C++ dopušta da kontroliramo performanse i direktno utječemo na kompajliranje i izvršavanje koda.

Efikasan program je onaj program koji učini najmanje posla da izvrši zadatak.

1.2 Postojeće benchmark knjižnice za C++

Postoje mnoge gotove i takoreći usavršene knjižnice za benchmark, neke od njih su:

- Hayai library[1]
- Celero library[2]
- Nonius library[3]
- Google Benchmark library[4]

Google-ova knjižnica jedna je od najmoćnijih ali isto tako i kompliciranijih ako se želi ući u dubinu. Od ovih su knjižnica preuzete neke generalne ideje i smjernice pri pisanju vlastitog Cross-Platform Benchmark rješenja.

Pri rješavanju ovog zadatka, puno inspiracije je preuzeto iz Hayai knjižnice koja je izuzetno jednostavna za korištenje, a u isto vrijeme dovoljno snažna za napredniji benchmark.

2 Korištene platforme, tehnologije i alati

Pri implementaciji i testiranju korišten je isključivo C++ programski jezik. C++ standardne knjižnice te najpopularnija „Boost“ knjižnica sadrže funkcije koje rade na svim kompajlerima i svim C++ podržanim platformama što je bio glavni kriterij odabira jezika s obzirom da je cilj ovog projekta napraviti platformski nezavisni benchmark alat.

Za verzioniranje programskog koda i izvještaja korišten je Git (GitHub) sustav.

Pošto radimo platformski nezavisni program moramo koristiti alat koji će moći pronaći kompajler, kompajlirati i linkati sve potrebne zavisnosti na svim C++ platformama (Windows, Linux, Mac). Za naše potrebe koristiti ćemo CMake koji je osmišljen tako da on sam može raditi samo sa C++ kompajlerom. Namijenjen je aplikacijama koje koristi velik broj knjižnica, a radi sa nativnim OS build okruženjima kao što je make. Uz sve ove pogodnosti, CMake može raditi i cross-compile što je također značajno za platformski nezavisni alat. Cross-compile neće biti korišten u ovom projektu.

Kao razvojnu okolinu koristili smo KDeveleop 4 na Linux (Lubuntu) OS-u. Za potrebe testiranja korišten je Microsoft Visual Studo na Windows OS-u.

3 Proces provođenja Benchmark-a

3.1 Mjerenje vremena izvršavanja

Mjerenje vremena izvršavanja osnovna je zadaća benchmark alata. Kako su računala vrlo kompleksna, rade na više slojeva te postoji velik broj varijabli koje mogu utjecati na to da naše očitanje ne bude točno ili dovoljno točno.

Vrijeme možemo podijeliti u više kategorija: procesorsko vrijeme korisnika (User CPU Time), procesorsko vrijeme sustava (System CPU Time) i realno proteklo vrijeme (Elapsed real time / Wall time).

- User CPU Time je vrijeme koje procesor provede izvršavajući instrukcije našeg koda.
- System CPU Time je vrijeme koje procesor provede izvršavajući sistemske instrukcije na kernelu, kao npr fork.
- Elasped real time je stvarno vrijeme koje je proteklo dok se naš kod izvršio.

Sada kada znamo koja sve vremena možemo dobiti, možemo vidjeti da postoji još varijabli koje će utjecati na naš benchmark, izmjereno vrijeme:

$$t_m = t + t_q + t_n + t_o$$

- t_m je izmjereno vrijeme
- t je pravo vrijeme koje nas zapravo interesira
- t_q je vrijeme dodano zbog šuma kvantizacije (zaokruživanja)
- t_n je vrijeme dodano od strane ostalih šumova
- t_o je overhead, dodan od strane mjerenja, ponavljanja i pozivanja funkcija

Sva se ova vremena sumiraju, tj. ne mogu biti negativna. Nastavno na prošlu činjenicu kako bismo točno odredili vrijeme potrebno je napraviti više mjerenja nad istim kodom. Što veći broj mjerenja napravimo veća je šansa da ćemo dobiti točnije rješenje. Uz veliki broj mjerenja, Također pravilno je zaključiti da će s većim brojem mjerenja doći do normalne (Gaussove) distribucije.

Jednom kada imamo velik broj očitanja potrebno je iz svih rezultata izvući najtočnije moguće vrijeme, što se u većini slučajeva radi uz pomoć minimum, tj uz pomoć mode operatora vrijeme će težiti prema minimumu. Postoji nekoliko iznimaka u kojima bismo trebali uzeti u obzir i najgore slučajeve, kao u slučaju mrežnog rada, live locking-a, dead locking, itd…

3.2 Usporedba dvaju benchmarka

Kako bismo mogli uspoređivati potrebno je napraviti osnovnu točku (bazno očitanje) benchmark koda kako bismo mogli znati što se događa kada mijenjamo i optimiziramo kod.

Za primjer ćemo uzeti proteklo procesorsko vrijeme. Postoji više načina usporedbe baznog očitanja i novog (poboljšanog).

Tradicionalni:

- Pokrenuti benchmark početnog koda *n* puta, izmjeriti vrijeme t_a
- Pokrenuti benchmark novog koda *n* puta, izmjeriti vrijeme t_b
- Relativno poboljšanje $r = \frac{t_a}{t_b}$

Diferencijalni:

- Pokrenuti benchmark početnog koda $2n$ puta, izmjeriti vrijeme t_{2a}
- Pokrenuti benchmark novog koda *n* puta, početnog koda *n* puta i izmjeriti vrijeme t_{a+b}
- Relativno poboljšanje $r = \frac{t_{2a}}{2t_{a+b} - t_{2a}}$
- Poništava jedan dio greške koja se može pojaviti u tradicionalnom načinu

3.3 Mjerenje utrošene memorije

Utrošak memorije također utječe na efikasnost koda, ako procesor mora raditi dulje zato jer radi sa nepotrebno velikim tipom podataka znači da možemo optimizirati naš kod.

Neke od stvari koje možemo mjeriti su objekti (alokacije/dealokacije), memorija(ukupna potrošnja, po objektu, itd.) Također možemo mjeriti broj kopiranja i micanja argumenata.

Alati koje možemo korisiti su googleperftools/TCMalloc, MemTrack i ostali.

4 Implementacija

Implementaciju sam započeo definiranjem CMakeLists.txt datoteke koju korisi CMake kako bi znao buildati zadani program.

U CMakeLists.txt dodana je Boost knjižnica funkcija koja će nam pomoći implemetacijom nekih gotovih funkcija i knjižnica koje su podržane na svim platformama.

Tijekom kompilacije i dodavanja drugih funkcionalnosti Boost knjižnice u projekt došlo je do komplikacija sa instalacijom boost knjižnice i linkanjem pojedinih djelova knjižnice kao što su boost/accumulators i boost/function knjižnice.

4.1 Prvotni tok benchmarka

Benchmark algoritam sam zamislio na temelju činjenica koje su napisane u prethodnom poglavlju te sam se pokušao što više držati njih kao vodilja. Moja prva zamisao algoritma s kojim ću izmjeriti vrijeme je bila ovo:

1. Korisnik unosi kod koji želi izmjeriti te opcije za benchmark
2. Korisnik može unijeti više različitih benchmark-a koji se skupa pokreću run metodom
3. Algoritam izvršava kod nekoliko puta kako bi se što više približilo normalnoj razdiobi
4. Tijekom svakog pojedinačnog pokretanja kod se izvrši nekoliko puta (više iteracija)
5. Algoritam pokreće precizni timer (mikro sekunde) prije svake iteracije
6. Algoritam zaustavlja precizni timer nakon svake iteracije
7. Razlika između dvoje timera se sprema kao uzorak u akumulator
8. Akumulator uzima najmanju vrijednost te ju spremu u drugi akumulator koji nam služi da možemo uzeti median iz više pokretanja
9. Ispisuju se statistike koda

Ispočetka je bilo teško ostvariti sve funkcionalnosti no uz nekoliko iteracija dobio sam funkcionalan koji je obavljao posao.

4.2 Boost Accumulators

Tijekom izrade projekta bilo je potrebno iz prikupljenih vremena izvršavanja izvuči najbitnije statističke podatke. Za to sam se odlučio koristiti boost::accumulators frameworkom.

Accumulators framework je poprilično moćan alat za inkrementalne kalkulacije, poprilično je jednostavan za korištenje i najbitnije od svega boost knjižnica optimizira kod prema tipu podataka i statistikama koje su nam potrebne što ga čini brzim.

Princip na kojem accumulators radi je jednostavan. Potrebno je inicijalizirati accumulator_set za tip podataka i statistike koje želimo izračunati. Prvi parametar je tip dok iza njega slijede tag parametri kojima označavamo željene statistike.

```
accumulator_set < double, stats<tag::mean, tag::min, tag::max > >acc_all;
```

Kod 1accumulator_set metoda

U našem slučaju, ovo je akumulator koji izračunava srednju vrijednost, minimum i maximum na double varijablama, a konkretno služi za pronalazak navednih statističkih parametara nakon odrađenog dovoljnog broja pokretanja koda.

Dodavanje uzoraka u akumulator vrši se jednostavnim pokretanjem naredbe acc(podatak koji se sprema). Extrakcija matematičkih statistika je također trivijalna te sve što treba učiniti je pozvati željenu metodu i proslijediti joj ime akumulatora kao npr. max(acc_all). U nižem primjeru koda može se vidjeti ispis nekih od statistika unutar mojeg alata.

```
std::cout << "| Average time: " << mean(acc_all) << " us." << std::endl;
std::cout << "| Fastest time: " << min(acc_all) << " us." << std::endl;
std::cout << "| Slowest time: " << max(acc_all) << " us." << std::endl;
```

Kod 2Jednostavnost ispisa statističkih podataka

4.3 Korištenje implementacije s glavnim programom

Za korištenje ove implementacije potrebno je kompajlirati executable pomoću CMake-a, a nužno je imati i boost knjižnice.

Korištenje je vrlo jednostavno. Kod na kojem želimo napraviti benchmark staviti ćemo u funkciju, kao što je to napravljeno ovdje:

```
void testFunction()
                        {
                          for(int i=0;i<500;i++)
                          {
                            std::cout << ""; //Just a test code


                          }
                        }
```

Kod 3Primjer testne funkcije za simuliranje instrukcija na CPU

Nakon što smo napravili testnu funkciju možemo instancirati benchmark klasu koja u ovom slučaju ima defaultni konstruktor pa ne prima nikakve parametre.

struct benchmark_test
- title : string - iterations : int - runs : int - functiontocall void(*)()

Tablica 1UML struct-a koji pohranjuje test

Podatci svih benchmark testova spremaju se u strukturu koja se vidi na tablici 1. Benchmark klasa sadrži listu strukturu pa je zato moguće izvršiti i jako veliki broj testova od jednom ako je to potrebno.

Benchmark Class
- benchmark_test : struct - benchmarks_list : list<benchmark_test>
+ add(string title, int iterations, int runs, void (*function)() + run()

Tablica 2UML Benchmark klase

Kao što se može vidjeti na tablici 2 postoje add() i run() metode koje naravno odgovoraju dodavanju novog benchmarka kojeg želimo provoditi i pokretanju svih dodanih benchmark-a.

Ovo je primjer dodavanja novog benchmark testa za neku željenu klasu. Napomena, add metoda može biti pozvana i samo sa parametrima (string title, void (*function)()) u kojem slučaju se uzimaju default vrijednosti od 10 pokretanja i 1000 iteracija u svakom.

```
int main (){
                benchmark testBenchmark;
                testBenchmark.add("Prvi test",100, 10, testFunction);
                testBenchmark.add("Drugi test",100, 10, testFunction2);
                testBenchmark.run();

                return 0;
              }
```

Kod 4Primjer main metode

U kodu 4 možemo vidjeti da je korištenje poprilično jednostavno, pa sam se odlučio cijeli benchmarking alat pokušati napraviti kao statičnu knjižnicu kako bi se lako mogao integrirati u bilo koji program u kojem je potreban benchmark.

4.4 Prebacivanje koda u statičnu knjižnicu

Prebacivanje u statičnu knjižnicu naišlo je na neke zapreke, a većinom u linkanju sa CMake-om, no na kraju sam uspješno povezao .h i .cpp file i knjižnica dobro funkcionira. Važno je napomenuti da knjižnica također zahtjeva Boost pa je bilo važno da se na moju knjižnicu poveže i boost knjižnica.

Nakon što je veći dio posla uspješno odrađen u CMake-u, promijenjeno je nekoliko sitnica na .cpp i .h datoteci i knjižnica je u punom pogonu.

Korištenje knjižnice ne razlikuje se od korištenja u poglavlju 4.3, potrebno je napraviti include knjižnice u direktoriju benchlib : `#include<bench.h>.` Te se nakon toga instancira benchmark objekt te je proces isti kao i prije.

4.5 Rezultati implementacije

Nakon uspješnog prebacivanja u statičku knjižnicu benchmark je pokrenut na testnoj datoteci benchtest.cpp koja se nalazi u vršnom direktoriju repozitorija, a povezana je na knjižnicu koja se nalazi u direktoriju benchlib.

Nakon uspješnog pokretanja benchtest.cpp u konzoli se dobije ispis ovakvog oblika:

```
_____________________________________
Running 2 benchmark/s.
Benchmark Name: Prvi test
10 runs and 1000 iterations to perform:
Benchmark Running
-------------------------------------
Average time: 53.8353 us.
Fastest time: 53.64 us.
Slowest time: 53.919 us.
Average performance: 18575.2 iterations/s.
Fastest performance: 18642.8 iterations/s.
Slowest performance: 18546.3 iterations/s.
-------------------------------------

Benchmark Name: Drugi test
10 runs and 1000 iterations to perform:
Benchmark Running
-------------------------------------
Average time: 159.243 us.
Fastest time: 159.243 us.
Slowest time: 159.243 us.
Average performance: 6279.71 iterations/s.
Fastest performance: 6279.71 iterations/s.
Slowest performance: 6279.71 iterations/s.
-------------------------------------
```

Kod 5Ispis benchmark alata nakon 2 testa

Benchmark alat je u mogućnosti mjeriti:

- Prosječno vrijeme
- Najbrže vrijeme
- Najsporije vrijeme
- Prosječni broj iteracija po sekundi
- Najveći broj iteracija po sekundi
- Najmanji broj iteracija po sekundi

Također je vrijedno napomenuti da se vremena prikupljaju na sljedeć način:

1. Prikupljeno se vrijeme od pojedine iteracije akumulira, te se na kraju pokretanja uzima minimalna vrijednost koja bi statistički trebala biti najbliža pravoj vrijednosti izvođenja.
2. Nakon što se sva pokretanja odrade, uzima se prosjek, minimalna i maksimalna vrijednost od svih minimuma koji su prikupljeni.

Mijenjanjem parametara iteracija i broja pokretanja može se značajno utjecati na preciznost. Također u nekim se trenutcima zna dogoditi da jedan od testova prikaže sve iste vrijednosti, na žalost nisam uspio ukloniti ovu anomaliju, ali mišljenja sam da je možda posljedica optimizacije ili nekakvog cache mehanizma.

5 Zaključak

Mišljenja sam da je ovo bio jedan od zanimljivijih projekata kojima sam pristupio, iako moram priznati da mi platformski nezavisno programiranje nije baš privlačno, naučio sam se drugačijem razmišljanju i većem oprezu pri kodiranju.

Benchmarking koda, a najviše optimizacije brzine koje se zapravo događaju u kompajleru su me zapravo ugodno iznenadili i volio bi se i u budućnosti baviti s time. Mislim da svijet u kojem trenutno živimo očajno vapi za brzinom, Internet stvari i ostale stvari su nemilosrdne prema procesorima i potrebno je optimizirati njihov rad do maksimuma kako bismo mogli napredovati.

Posljednje što sam zaključio je da iako za vrijeme našeg studija nismo imali puno doticaja sa C++ jezikom te osjetim kako mi fali profinjenosti, no i dalje mi se izuzetno sviđa zbog gotovo neograničene kontrole koju pruža nad vlastitim kodom.

6 Literatura

http://jogojapan.github.io/blog/2012/11/25/measuring-cpu-time/

http://www.boost.org/doc/libs/1_61_0/doc/html/accumulators/user_s_guide.html

https://www.youtube.com/watch?v=vrfYLlR8X8k

http://rubylearning.com/blog/2013/06/19/how-do-i-benchmark-ruby-code/

http://www.bfilipek.com/2016/01/micro-benchmarking-libraries-for-c.html

https://en.wikipedia.org/wiki/CMake

https://bruun.co/2012/02/07/easy-cpp-benchmarking

https://github.com/CppCon/CppCon2015/blob/master/Presentations/Benchmarking%20C%2B%2B%20Code/Benchmarking%20C%2B%2B%20Code%20-%20Bryce%20Adelstein%20Lelbach%20-%20CppCon%202015.pdf

https://msdn.microsoft.com/en-us/library/ms235627.aspx

https://github.com/ijuresa/Platformski-Nezavisno-Programiranje

[1] https://github.com/nickbruun/hayai

[2] https://github.com/DigitalInBlue/Celero

[3] https://github.com/rmartinho/nonius

[4] https://github.com/google/benchmark