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 3](#_Toc460461027)

[1.1 Zašto želimo raditi benchmark koda? 3](#_Toc460461028)

[1.2 Postojeće benchmark knjižnice za C++ 4](#_Toc460461029)

[2 Korištene platforme, tehnologije i alati 5](#_Toc460461030)

[3 Proces provođenja Benchmark-a 6](#_Toc460461031)

[3.1 Mjerenje vremena izvršavanja 6](#_Toc460461032)

[3.2 Usporedba dvaju benchmarka 7](#_Toc460461033)

[3.3 Mjerenje utrošene memorije 7](#_Toc460461034)

# 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 .

## 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.

## Postojeće benchmark knjižnice za C++

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

* Hayai library[[1]](#endnote-1)
* Celero library[[2]](#endnote-2)
* Nonius library[[3]](#endnote-3)
* Google Benchmark library[[4]](#endnote-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.

# 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.

# Proces provođenja Benchmark-a

## 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:

* je izmjereno vrijeme
* je pravo vrijeme koje nas zapravo interesira
* je vrijeme dodano zbog šuma kvantizacije (zaokruživanja)
* je vrijeme dodano od strane ostalih šumova
* 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…

## 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
* Pokrenuti benchmark novog koda *n* puta, izmjeriti vrijeme
* Relativno poboljšanje

Diferencijalni:

* Pokrenuti benchmark početnog koda 2*n* puta, izmjeriti vrijeme
* Pokrenuti benchmark novog koda *n* puta, početnog koda *n* puta i izmjeriti vrijeme
* Relativno poboljšanje
* Poništava jedan dio greške koja se može pojaviti u tradicionalnom načinu

## 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.

# 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.

## 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.

## 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.



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.

# Literatura

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

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

1. https://github.com/nickbruun/hayai [↑](#endnote-ref-1)
2. https://github.com/DigitalInBlue/Celero [↑](#endnote-ref-2)
3. https://github.com/rmartinho/nonius [↑](#endnote-ref-3)
4. <https://github.com/google/benchmark> [↑](#endnote-ref-4)