Osnove računarstva visokih performansi

Bilješke s predavanja

Andrej Dundović

Križevci, 2021

© 2020 Andrej Dundović.
Ovaj se rad može koristiti, mijenjati i dalje distribuirati po pravilima licence
Creative Commons Imenovanje 4.0:
http://creativecommons.org/licenses/by/4.0/
Inačica dokumenta: git-6969ecc, 2021-06-24

Sadržaj

1 | Uvod

Riječ "računalo" (engl. *computer*) pojavljuje se u engleskom jeziku još u 17. stoljeću, ali u značenju osobe koja vrši kalkulacije. Isto se značenje zadržalo sve do sredine 20. stoljeća, a većina "računala" početkom 20. stoljeća bile su žene kao jeftinija radna snaga u odnosu na muškarce iste razine matematičkog obrazovanja i sposobnosti. Tek nakon Drugog svjetskog rata u definiciji ove riječi učestalo se pojavljuje "stroj", odnosno "mašina". No bilo ono čovjek ili mašina, računalo podrazumijeva izvršavanje matematičkih ili logičkih izračuna, nešto specijalizirano i učinkovito za tu svrhu. Učinkovito.

Za ovaj uvod, ilustrativna je priča o Enigmi. Priča koja se odvija baš u periodu promjene dominantnog značenja riječi s "osobe" na "stroj", a u kojoj je vjerojatno igrala i značajnu ulogu. U Drugom svjetskom ratu, nacistička Njemačka šifrirala je svoju radio komunikaciju sustavom šifriranja zvanim Enigma. Šifriranje predstavlja svojevrstan algoritam kojim se informacija, odnosno čitljiva poruka zamjenjuje šifrom. Jedan od najjednostavnijih primjera je algoritam šifriranja ROT13 gdje se slovo zamjenjuje 13. slovom po redu u abecedi, bazične Latinice. Naprosto šifru tvori translacija svih slova izvorne poruke za isti broj pomaka. Zgodno svojstvo tog algoritma jest da, kad se postupak ponovi u istom smjeru, odnosno drugi put, dobiva se izvorna poruka jer bazična latinička abeceda broji 26 slova. Ovo šifriranje navodno je korišteno još u Rimskom Carstvu.

Dvije tisuće godina kasnije, Enigma je šifrirala poruke sofisticiranije – tako da je svako slovo izvorne poruke bilo zamjenjeno nekim drugim slovom putem sustava rotora. Svaki rotor dodatno je ispremješao slova tako da su se ista slova izvorne poruke preslikavala u različita slova šifre i obrnuto, različita slova izvorne poruke mogla su se preslikati u ista slova šifre, a samo onaj tko je znao početnu poziciju rotora kojom je poruka bila šifrirana mogao ju je jednostavno dešifrirati. Pozicija rotora mijenjala se svaki dan. Onaj tko nije znao poziciju rotora morao je isprobavati različite kombinacije na silu, "sirovom snagom" (engl. *brute-force*) ne bi li dešifrirao poruku.

Poljaci su još prije rata razbijali poruke šifrirane Enigmom, ali su je Nacisti usavršavali tijekom rata, povećavajući kompleksnost i broj mogućih kombinacija što je činilo dešifriranje sve težim i težim, pogotovo da se uspješno provede u što kraćem roku. S 5 rotora i dodatnim mehanizmima preslikavanja slova konačan broj kombinacija došao je do čak 10^{21}, tj. načina na koje je Enigma mogla šifrirati jednu poruku, što je praktički postalo nemoguće za čov-

jeka, odnosno i mnogo njih, da provedu dešifriranje u razumno vrijeme koristeći samo svoju sposobnost računanja.

U pomoć tada uskaću različiti elektromehanički uređaji (Poljaci ih zovu kritptografske bombe) koji ubrzavaju pretraživanje kombinacija. Nakon invazije nacističke Njemačke na Poljsku, Francuzi uspješno izvlače poljski tim matematičara koji su razvijali tehnike razbijanja Enigme. Saveznici uz njihovu pomoć tada razvijaju dalje svoje sustave razbijanja Enigme i svoje Bombe. Dalje priču znate, s Alanom Turingom i njegovom ulogom u razbijanju Enigme. Samo kao kuriozitet, slaba točka Enigme koja je omogućila uspješno razbijanje bila je što nikad slovo nije pretvarala u isto slovo. Taj detalj omogućio je ključnu informaciju u pronalasku efikasnog algoritma razbijanja šifriranih poruka.

Na ovom primjeru vidimo kako nam je, pored činjenice da se nešto može izračunati, bitno i da nešto brzo izračunamo, u smislenom roku. To je na neki način osnovno svojstvo modernih računala - to je alatka koja poboljšava čovjekovu mogućnost računanja. Dakako, to je svojstvo omogućilo i razvoj novih, drugih primjena računala koje nisu vezane isključivo za brzo izračunavanje nečega što bi mogli izračunati i "ručno".

U ovom tečaju mi ćemo se zadržati nad tom temeljnom funkcijom računala – na brzom izračunavanju, a ostale funkcije ostavit ćemo za druge prilike i drugima da se bave. Mogli bi reći da se napori i metode da se nešto brzo izračuna koristeći računalo danas sažimlju u području *računarstvu visokih performansi* (engl. *high performance computing*, HPC). Iako je HPC popularni, marketinški naziv za različite pojmove i tehnike, sve one označavaju nekakvo "veliko računanje". Nerijetko se računarstvo visokih performansi izjednačuje i s pojom superračunarstvom i superračunalima.

Nabrojimo neke aktualne definicije HPC-a:

> Računarstvo visokih performansi, znano još i kao superračunarstvo (engl. *supercomputing*) odnosi se na računske sustave s ekstremno velikom snagom računanja koja mogu riješiti izuzetno kompleksne i zahtjevne probleme. (*Europska komisija, 2021.*)

> Računarstvo visokih performansi u najopćenitijem smislu referira se na gomilanje računalne snage na način koji omogućuje mnogo više performanse nego bi ih netko izvukao iz tipičnog stolnog računala ili radne stanice kako bi riješio velike probleme u znanosti, inženjerstvu ili industriji. (*InsideHPC, 2021.*)

> Superračunalo je računalo s visokim performansama u usporedbi s računalom opće namjene. (*Wikipedia, 2021.*)

> Po definiciji, superaračunala su najbrža i najmoćnija dostupna računala, i u ovom trenu, termin "superračunalo" gotvo uvijek podrazumijeva paralelne strojeve. (*Rubin H. Landau, A Survey of Computational Physics, 2008.*)

Dok su ove prve definicije vrlo općenite, možda čak i antidefinicije, ova zadnja, uz ogradu ("u ovom trenu"), daje natruhu koji je to zapravo mehanizam postizanja visokih performansi u modernim računalima i računalnim sustavima – **paralelizacija**.

No prije nego nastavimo s paralelizacijom, idemo malo ponoviti osnove, a koje će nam trebati i u nastavku tečaja.

Računala provode izračune u instrukcijskim ciklusima (engl. *instruction cycles*) pri čemu *središnja jedinica za obradbu*, CPU (engl. *central processing unit*), odnosno procesor, kako skraćeno kažemo, dohvaća instrukciju iz memorije i izvršava je. U osnovi, što je veća frekvencija ciklusa, to je obrada izračuna brža. Stoga, kako bi smo ubrzali naše izračune trebamo samo povećavati frekvenciju, zar ne? Dok je mnogi niz godina povećanje frekvencije CPU-a i funkcioniralo[1], to je u drugoj polovici dvijetisućitih došlo do svoga kraja. "Besplatan ručak je gotov" (engl. *The Free Lunch Is Over*[2]) napisao je 2004. godine Herb Sutter (poznati stručnjak za C++). Sutter najavljuje da se programeri neće moći pouzdati na puko povećanje brzina procesora kako bi njihovi programi bili brži, već će morati naučiti nove tehnike programiranja kako bi iskoristili nadolazeći hardver i to prvenstveno tzv. višedretvenost (engl. *multithreading*) i višestruki broj jezgara na procesoru.

Godinama je vrijedilo ono što se može sažeti u fenomenu "Ono što je Andy donio, Bill je oduzeo."[3]. Koliko god procesori bili brži, softver je nalazio načina da "utroši" tu dodatnu brzinu... bilo da više toga napravi u isto vrijeme, ili da postane neefikasniji! No *fizika je rekla dosta*. Procesori su došli do $\tilde{3}.5$ GHz, a nakon toga zagrijavanje je postalo preveliko – pogotovo s obzirom na jedinicu površine pa je teško toplinu odvesti s procesora, potrošnja energije prevelika, a tu je bio i problem tzv. "curenja struje" (kvantnomehanički fenomen). Još su "overclockeri" natjeravali procesor posebnim hlađenjem na frekvencije do 4 GHz ili čak 5 GHz (hlađenje tekućim dušikom), a negdje i više (hlađenje tekućim helijem!). Ipak, povećanje frekvencije više nije imalo ni smisla u kontekstu performansi, npr. Intel je tada pisao da spuštanjem frekvencije jedne jezgre za 20% uštedi se polovica energije, a žrtvujući samo 13% performansi.

Od sredine dvijetisućitih nastavlja se razvoj CPU-a u drugim smjerovima. Jedno su intenzivnije optimizacije na razini procesora kao što su predviđanje grananja, izvršavanje više instrukcija u jednom ciklusu ili "optimizacija" kao što je engl. *out-of-order execution* (izvršavanje izvan reda) da bi što manje ciklusa procesora bilo neiskorišteno. Međutim, posljednje više ni nisu optimizacije, već iste mijenjaju način izvršavanja programa! U konačnici, ostalo je donekle povećavanje priručne memorije, odnosno *cache* memorije koja je puno brža (i skuplja!) te bliže samom procesoru (u praksi se fizički nalazi na istoj pločici kao i procesor). Pristup glavnoj memoriji (RAM-u) je "skupo", odnosno sporo! Više o tome bavit ćemo se u prvom dijelu tečaja.

[1] Popularna 486 iz '89 radila je na 50 MHz, a narednih malo više od 10 godina frekvencija se ugrubo udvostručivala svakih 2 godine (tzv. "Mooreov zakon").

[2] http://www.gotw.ca/publications/concurrency-ddj.htm

[3] "Andy giveth, and Bill taketh away.", misli se na Andrewa Grovea u Intelu tada i Billa Gatesa u Microsoftu.

I tada dolazimo do uvođenja više jezgara, a uz to i SMT-a (engl. *Simultaneous multithreading*) – izvršavanja više neovisnih dretvi (engl. *threads*) da bi se smanjili neiskorišteni ciklusi. Sad je možda zgodan trenutak i da se objasni kako radi "istovremeno" izvršavanje više računalnih procesa na jednom procesoru. Jednostavno, tako da je istovremenost samo prividna - jedan se proces izvršava, a ostali čekaju na red i onda se brzo izmjenjuju tako da korisnik dobije dojam "istovremenosti". No što je proces, a što *thread*? Procesi su međusobno izolirani na razini operacijskog sustava jer, primarno, ne dijele isti memorijski prostor, dok ga *threadovi* dijele. Kaže se još da su *threadovi* "lagani procesi". *Threadovi* su tako najmanji nezavisni niz instrukcija. Proces može sadržavati jedan ili više *threadova*, ali također može stvoriti i nove podprocese (npr. tabovi u modernim internetskim preglednicima podijeljeni su u podprocese) koji ne dijele isti memorijski prostor.

Za razliku od višestrukih stvarnih jezgara, SMT (Intel to kod sebe zove *Hyperthreading*) ne omogućuje stvarno paralelno izvršavanje *threadova*, ali daje u praksi 5% to 15% ubrzanje performansi kod višedretvenih programa, a u idealnim situacijama najviše 40%, a što je ipak manje od duplog, koliko mogu ponuditi dvije jezgre. Jasno, ovo uopće ne ubrzava jednodretvene programe.

Nažalost, ovakve tehnike dovele su i do katastrofalnih posljedica, ranjivosti poput Meltdowna i Spectrea. Pogotovo kod Intelovih procesora, makar ni AMD-ovi ni ARM-ovi nisu bili pošteđeni ove druge ranjivosti.

Danas je višejezgrenost procesora uobičajna, čak i na malim uređajima, poput pametnih telefona. Paralelizacija programa tako da se izvršava na više jezgara istovremeno predstavlja glavnu temu ovog tečaja. Treba odmah naglasiti da neki problemi ne mogu baš paralelizirati, npr. u tom kontekstu kaže se da jedna žena iznjedri dijete u 9 mjeseci, ali to ne znači da 9 žena može napraviti isto u mjesec dana.

I da se vratimo na fiziku, tj. već spomenutih ograničenja. Ona su donekle olabavljena jer višejezgreni procesori imaju veću površinu za disipaciju energije i za istu potrošnju energije mogu izvršiti više zadaća na nižem taktu od jedne jezgre iste potrošnje i više frekvencije, pa je tako barem nakratko, omogućeno daljnje povećanje ukupne računske snage procesora.

2 Kratak vodič za programske jezike visoke razine[1]

Većina programera danas piše programe u Pythonu, Rubyju, JavaScriptu ili sličnim jezicima visoke razine u kojima programer obično ne mari o tehničkim detaljima kao što je upravljanju memorijom ili specificiranjem tipova. Tako pisane programe zbog automatiziranog upravljanja memorijom (npr. sakupljanjem smeća, engl. *garbage collection*, GC) nije lako paralelizirati na razini *threadova* (onih dakle koji dijele memoriju), pa se često pribjegava prvo taktikama povećavanja brzine u *single-thread* izvršavanju[2].

Prije nego se krene na programske jezike niže razine i "ručnim" upravljanjem memorijom, logično je prvo pokušati poboljšati performanse svog programa tehnikama koje su još dostupne bez promjene programskog jezika. Ovo poglavlje upravo je namijenjeno za to. Kratak vodič kroz tehnike kako poboljšati performanse svog programa napisanog u Pythonu. Makar je ovo poglavlje konkretno posvećeno Pythonu, što zbog njegove raširenosti, a što zbog autorove upućenosti u isti, slične ideje i tehnike mogu se iskoristiti/primijeniti i za druge programske jezike visoke razine.

Prije svega, treba se prisjetiti mudrih riječi Donalda Knutha:

> Prerana optimizacija je korijen sveg zla. (engl. *Premature optimization is the root of all evil.*)

što je zapravo dio većeg citata:

> The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; premature optimization is the root of all evil (or at least most of it) in programming.

To implicira osnovnu strategiju kod pisanja programa: prvo se treba pobrinuti da program dobro radi ono što treba raditi, pa makar u prvom trenu možda ne i najbrže, a tek je sljedeći korak optimizirati isti.

[1] Na primjeru programskog jezika Python

[2] Jeff Atwood, osnivač the Stack Exchangea, tako i dalje naglašava važnost *single-thread* performansi u slučaju Rubyja, https://blog.codinghorror.com/to-ecc-or-not-to-ecc/

Da se ne bi baš pristupom pokušaja i promašaja pogađalo gdje je usko grlo programa, tj. koji je dio programa najsporiji (makar su neke stvari očite i bez kvantitativne analize), koriste se alati za analizu performansi koda, odnosno različiti profileri i mjerači brzine izvršavanja programa. U slučaju Pythona, to je cProfile.

Vratit ćemo se na profiliranje u nastavku tečaja, a sada ćemo proći popis savjeta kako općenito ubrzati program napisan u Pythonu.

2.1 Biranje efikasnijih struktura

Treba rabiti one strukture koje imaju manji "overhead", ako ono što donosi "overhead" nije bitno za funkcioniranje programa. Očiti je primjer u Pythonu kako su ntorka (engl. *tuple*) ili skup (engl. *set*) efikasnije strukture od liste, pa ako nije nužno potrebna lista da se određene podatke "skupi", bolje je iskoristiti ntorku ili skup.

Primjer usporedbe:

```
$ python -m timeit "x=(1,2,3,4,5,6,7,8,9)"
20000000 loops, best of 5: 10.5 nsec per loop

$ python -m timeit "x=[1,2,3,4,5,6,7,8,9]"
5000000 loops, best of 5: 45.4 nsec per loop
```

gdje `timeit`[3] mjeri vrijeme izvršavanja skripte ili dijela koda.

Kod manipuliranja velikim skupovima brojčanih podataka najbolje je koristiti strukture podatka (i operacije nad njima) koje nudi `numpy`, pogotovo njegova polja (engl. *arrays*).

2.2 Jednostavna paralelizacija

Na gotovo svakom računalu danas na raspolaganju nam je više jezgri, a naš program u Pythonu (ili Rubyju, ili XYZ-u) koristi samo jednu. Kako iskoristiti i ostale? Direktan je pristup paraleliziranje petlji koje ne koriste, a pogotovo ne pišu, po zajedničkoj memoriji. Međutim, kako je već spomenuto, teško je to elegantno ostvariti u slučajevima kad imamo engl. *garbage collection*. No osim što se različiti programski jezici visoke razine tome dovijavaju različitim strategijama, čak i različite implementacije istog programskog jezika nemaju iste pristupe i daju različite rezultate.

U Pythonu tako postoji biblioteka `multiprocessing`[4] koja stvara nove podprocese (podsjećamo, nije isto što i `threading`) koji sadrže zasebne instance interpretera i tako zaobilazi problem GIL-a (engl. *Global Interpreter Lock*), odnosno njegovog ograničenja da se samo jedan *thread* izvrđava u jednom trenutku. Bez GIL-a, barem u zadanom interpreteru, CPythonu, ne

[3] https://docs.python.org/3/library/timeit.html
[4] https://docs.python.org/3/library/multiprocessing.html

bi radio GC. Stvaranje novih podprocesa je "skupo" i time ih treba koristiti nad zadacima čije je vrijeme izvršavanja relativno dugo s obzirom na vrijeme potrebno da se podproces stvori.

Kako iskoristiti `multiprocessing`? Najjednostavnije koristeći posebnu verziju `map`-a (mapiranja/preslikavanja) nad listom podataka:

```
from multiprocessing import Pool
def f(x):
    return x*x
if __name__ == '__main__':
    with Pool(5) as p:
        print(p.map(f, [1, 2, 3]))
```

gdje je dobro s `if`-om ograničiti pokretanje `map`-a na prvo pokrenuti proces, makar će ovaj primjer raditi i bez tog `if`-a.

Međutim, ako se dijelovi koda koji se trebaju izvršavati u zasebnim podprocesima isprepletu, naletit ćemo na kojekakve probleme koje nije lako *debuggirati* i tu treba pristupiti ili prepisivanju dijela programa ili nekim drugim pristupima. Savjet je i za inače, a pogotovo u kontekstu *multiprocessinga* stavljati, tj. definirati varijable isključivo u opseg (engl. *scope*) u kojem su potrebne, ne većim.

2.3 Alternativni interpreteri i anotiranje podataka tipovima

S obzirom na popularnost i rasprostranjenost Pythona kao programskog jezika opće namjene, ne čudi pojava alternativnih interpretera i specijaliziranih prevodioca koji prevode podskup Pythona u strojni kod. Pogotovo od kad je Python postao glavni izbor u "znanosti o podacima" (engl. *data science*), mnogo je resursa uloženo da se Python ubrza. Uostalom, slično kao i s JavaScriptom kao standardnim programskim jezikom svih web preglednika. Google nije imao izbora nego financirati razvoj interpretera, V8, i inače od vrlo sporog skriptnog jezika, doći do performansi koje JS u praksi ima danas[5].

Python ima više implementacija, pored osnovne i referentne, CPython, tu su još IronPython (baziran na .NET-u), Jython (baziran na JVM-u) te PyPy koji je neovisna implementacija koja pokriva i najveći skup mogućnosti osnovne, a k tome je zbog svog dizajna brža. Međutim, teško da će neki kompleksan program napisan za CPython raditi bez modifikacija u PyPyju. Zato u ovom kontekstu preporučam samodostatni paket `numba`[6]. To je, kako opisuju, akcelerator Python funkcija, odnosno specijalizirani prevodioc za Python baziran na LLVM-u koji se koristi kao modul u Pythonu te omogućuje jednostavnim dekoriranjem funkcija njihovo (JIT) prevođenje u strojni kod... i više od toga.

[5] ...ili kako je Yaron Minsky iz zajednice okupljene oko programskog jezika OCaml rekao jednim povodom da je puno "leševa" doktorskih studenata moralo biti utrošeno da bi se ta razina optimizacije postigla, https://vimeo.com/153042584.

[6] http://numba.pydata.org

Primjer dekoriranja funkcije koja po Monte Carlo računa konstantu π:

```
from numba import jit
import random

@jit(nopython=True)
def monte_carlo_pi(nsamples):
    acc = 0
    for i in range(nsamples):
        x = random.random()
        y = random.random()
        if (x ** 2 + y ** 2) < 1.0:
            acc += 1
    return 4.0 * acc / nsamples
```

Pored ovoga, možda valja istaknuti i Cython[7], prevodioc koji može prevest Python, ali i poseban podskup Pythona koji omogućuje anotacije tipova, u efikasniji strojni kod. Primjer:

```
def f(double x):
    return x ** 2 - x

def integrate_f(double a, double b, int N):
    cdef int i
    cdef double s, dx
    s = 0
    dx = (b - a) / N
    for i in range(N):
        s += f(a + i * dx)
    return s * dx
```

2.4 Programski jezici niže razine kao dio Pythona

U konačnici, kada gore predložena *ad-hoc* rješenja nisu zadovoljavajuća, moguće je spore dijelove Python programa prepisati u programski jezik niže razine (kao C/C++/Fortran/Rust) i onda ih učiniti dostupnim u Pythonu. Kod programskih jezika niže razine moguće su optimizacije memorije i instrukcija koje omogućuju kontrolu do najsitnijih detalja kako bi se omogućila što veća brzina pa su tako i mnoge postojeće biblioteke, kao recimo `numpy` ili `scipy`, sazidane tako.

[7] https://cython.org

Tako postoje automatski generatori sučelja prema Pythonu za programe pisane u C ili C++, npr. SWIG[8] ili Pybind11[9], Fortran[10] i sl.

[8] http://www.swig.org
[9] https://github.com/pybind/pybind11
[10] https://www.numfys.net/howto/F2PY/