

Seminarski rad u okviru kursa
Tehničko i naučno pisanje
Matematički fakultet

13. novembar 2022.

Ovde ide tekst ovde ide tekst ovde ide tekst ovde ide tekst ovde ide
tekst ovde ide tekst ovde ide tekst ovde ide tekst ovde ide tekst ovde ide
tekst ovde ide tekst ovde ide tekst ovde ide tekst ovde ide tekst

1	Uvod	2
2	Istorija	2
2.1	Istorija paralelne obrade	2
2.2	Istorija paralelnog programiranja	2
3	Princip rada paralelnog programiranja	3
3.1	Osnove paralelne obrade	3
3.2	Koraci paralelizacije programa	4
3.3	Modeli paralelnog programiranja	4
4	Upotrebe	5
4.1	Osnovne primene	5
4.2	Prednosti i mane	5
5	Primeri	6
5.1	Primer 1 - Preuzimanje sadržaja u pozadini	6
5.2	Primer 2 - Broj prostih brojeva u listi	7
5.3	Poređenje performansi	8
6	Zaključak	8
	Literatura	8

1 Uvod

Ovde ide tekst ovde ide tekst ovde ide tekst ovde ide tekst ovde ide tekst ovde ide tekst ovde ide tekst ovde ide tekst ovde ide tekst ovde ide tekst ovde ide tekst ovde ide tekst

2 Istorija

Da bi se pričalo o istoriji paralelnog programiranja, potrebno je prvo pričati o istoriji paralelne obrade.

2.1 Istorija paralelne obrade

Koreni paralelne obrade nalaze se u 1950-im godinama. Dva zaposlena iz IBM (en. International Business Machines Corporation), Džon Kok i Daniel Slotnik, prvi put diskutuju paralelizam u radu koji su zajedno objavili 1958 [1]. Tokom narednih godina, razne firme počinju rad na razvoju mašina sposobnih za paralelnu obradu. 1962, korporacija Burroughs proizvodi *D825 Modular Data Processing System* — računar razvijen za rad u vojnom okruženju [2]. Zbog njegove namene, on radi po principu SMP (eng. symmetric multiprocessing ili shared-memory multiprocessing) - dva ili više procesora su povezana na zajedničku memoriju, i kontroliše ih isti operativni sistem. U slučaju D825, koriste se 4 centralne procesorske jedinice, koje imaju pristup do 16 modula memorije [2]. D825 se smatra prvim pravim multiprocesorskim računarom [3].

Na Spring Joint konferenciji Američke federacije društva za obrađivanje podataka (eng. American Federation of Information Processing Societies, AFIPS), održane 1967, Daniel Slotnik i Džin Amdal debatovali su o paralelnoj obradi [4]. U ovoj debati nastaje Amdalov zakon ili Amdalov argument, koji opisuje granicu efikasnosti paralelne obrade. Ovaj zakon će se često koristiti da bi se predvidjela efikasnost i vreme izvršavanja paralelizovanog programa.

Priča o računarskim sistemima sa raspodeljenom memorijom (eng. distributed memory) započinje 1960-ih, ali se prvi takav računar pojavljuje tek 1983[5]. *The Cosmic Cube* od instituta *Caltech* sadrži 64 procesora koji istovremeno rade na jednom problemu, i međusobno komuniciraju tako što jedan drugome šalju poruke[6].

2.2 Istorija paralelnog programiranja

Uz razvoj paralelne obrade, počeo je da se pojavljuje veliki broj interfejsa koji su se do 1990-ih ujedinili u nekoliko standarda. 1992. održana je radionica na temu standarda za razmenu poruka u okruženju sa raspodeljenom memorijom (eng. Standards for Message-Passing in a Distributed Memory Environment) [7], na kojoj je započet razvoj standarda koji će se 1994. objaviti pod nazivom MPI (eng. Message Passing Interface). Naknadno su objavljeni standardi MPI 2.0 (1996), MPI 3.0 (2012) i MPI 4.0 (2021)[8].

Sa druge strane, za programiranje u SMP okruženju, OpenMP ARB (Architecture Review Board) objavljuje prve specifikacije interfejsa OpenMP 1.0 za Fortran u 1997, a sledeće godine i standard za C/C++. Nakon toga, objavljene su i verzije OpenMP 2.0 (2007), 3.0 (2008), 4.0 (2013) i 5.2 (2021), koji se koristi danas[9].

3 Princip rada paralelnog programiranja

Opšte rečeno, paralelno programiranje radi tako što deli program na više zadataka koji se izvršavaju istovremeno. Da bi bolje razumeli celu sliku, moramo prvo objasniti paralelnu obradu.

3.1 Osnove paralelne obrade

Paralelna obrada predstavlja sva izračunavanja na računaru koja se izvršavaju paralelno, tj. istovremeno [13]. Postoje više formi paralelne obrade, od kojih su najvažnije prikazane u narednoj tabeli:

Naziv	Opis	Mane
Nivo bita (eng. bit-level)	Zasniva se na proširenju veličine procesorske reči (osnovna jedinica podataka koju koristi određeni dizajn procesora). Koristi se u slučaju kada je broj bitova podatka kojeg je potrebno obraditi veći nego što procesor podržava za jednu obradu.	Trošak vremena i hardvera, tehnike eksploatacije ne ostvaraju pun potencijal
Nivo naredbe (eng. instruction level)	Predstavlja istovremeno izvršavanje niza naredbi. Daje prosečan broj naredbi koje se izvršavaju pri svakom koraku paralelne obrade [14].	Ograničeno od zavisnosti kontrole i podataka
Paralelizam podataka (eng. data parallelism)	Deli podatke na više procesora tako da se nad njima vrši isti zadatak istovremeno [15].	Mogu nastati problemi tokom nepravilne manipulacije podacima, ograničeno je sa količinom podataka koji mogu da protiču iz memorije.
Paralelizam zadataka (eng. task parallelism)	U kontrastu sa paralelizmom podataka, ovde se dele zadaci koji se izvršavaju nad jednim podatkom istovremeno [15].	Ograničeno troškovima komunikacije i sinhronizacije

Gore navedene mane i činjenica da se paralelizam često koristi zajedno sa sekvencijalnim programiranjem postavljaju pitanje, koliko je potencijalno ubrzanje izvršavanja zadataka ako primenimo paralelizam? Ovime se bavi ranije pomenut Amdalov zakon, koji meri efikasnost izvršavanja paralelnog programa. On kaže da celokupan dobitak zavisi od količine vremena prilikom kojeg se zapravo koristi paralelnost:

$$S_{max} = \frac{1}{(1-p) + \frac{p}{s}}$$

S_{max} je maksimalna potencijalna efikasnost, p procenat sistema koji je poboljššan, a s deo vremena u kojem radi poboljšani deo sistema. Kada bi se ova formula primenila na moderne računare, naizgled bi paralelnost bila nedovoljno efikasna, ali to nije slučaj. Ovo je primetio Džon Gustafson i ponovo ispitao Amdalov zakon i došao do svog zaključka, koji prikazuje efikasnost paralelizma tako što upoređuje rad istog zadatka na mašini sa jednim jezgrom:

$$S_{max} = N + (1 - N) * s$$

Gde je S_{max} maksimalna potencijalna efikasnost, N broj procesora koji obrađuje podatke, a s vreme za koje se zadatak obradi sekvencijalno.

3.2 Koraci paralelizacije programa

Sada kada znamo principe paralelne obrade, možemo komentarisati o koracima koji vode do kreiranja paralelnog programa. To su: dekompozicija, dodela, orkestracija i mapiranje.

Dekompozicija prvo razdvaja obradu na više zadataka koji se kasnije dele među procesima. Traži paralelizam i koliko može da ga iskoristi. Cilj je kreirati dovoljno zadataka da bi ih procesori istovremeno izvršavali, pri čemu zadaci mogu nastajati dinamički i njihov broj može da varira.

Kod dodele se određuje mehanizam koji deli zadatke iz prethodnog koraka među procesima. Cilj je da balansira rad, a i smanji troškove komunikacije i menadžmenta. Zajedno sa dekompozicijom ova dva koraka su poznata kao partitionisanje, tj. raspodela više različitih zadataka na više procesa.

Orkestracija smanjuje trošak komunikacije i sinhronizacije, imenuje podatke, čuva strukture podataka i njihove lokalne reference, organizuje rad, pravi raspored izvršavanja zadataka tako da ne dođe do preopterećenja ili manjka zadataka koji se trenutno izvršavaju.

Nakon ovog koraka, već imamo paralelni program, ali mapiranje je tu da reguliše koji procesi će biti izvršavani na istom procesoru, kao i koji procesi će se izvršavati na nekom specifičnom procesoru generalno.

3.3 Modeli paralelnog programiranja

Termin komunikacija se često pominje tokom koraka paralelizacije, a on se zapravo odnosi na razmenjivanje podataka između procesa koji se trenutno izvršavaju [16]. Na osnovu ovoga nastalo je nekoliko modela, od kojih su najvažniji:

- Model deljene memorije (eng. shared memory model) - Kod ovog modela svi procesi imaju pristup jednom glavnom delu programa koji sadrži sve podatke. Procesi mogu asinhrono da čitaju i pišu u ovoj međusobno deljenoj memoriji. Ipak, zbog toga mogu nastati problemi kao što je data race, o čemu će biti reči kasnije.
- Model prenošenja poruke (eng. message passing model) - U ovom modelu, procesi koji se paralelno izvršavaju šalju jedni drugima podatke preko poruka, koje mogu biti asinhrono (primaćalac dobija poruku i da nije spreman na to) i sinhrono (obe strane su u trenutku slobodne da pošalju i prime poruku).
- Partitionisani globalni adresni prostor (eng. partitioned global address space) - Model koji je hibrid prethodna dva, gde se koristi deljena memorija, međutim ovaj put je ona partitionisana tako da svaki proces ima pristup samo određenim tačkama u memoriji, umesto celokupnom prostoru. Podaci se razmenjuju pisanjem i čitanjem iz datog dela memorije.

4 Upotrebe

Sada kada znamo šta je paralelno programiranje možemo da pričamo o tome gde i kada se koristi

4.1 Osnovne primene

Paralelno programiranje koristimo kada imamo velike količine podataka, kompleksne račune ili velike simulacije. Iako paralelno programiranje može biti vremenski intenzivniji napor za programere da kreiraju efikasne paralelne algoritme i kod, ono sveukupno štedi vreme tako što koristi pun potencijal procesora.

Neke od oblasti u kojima paralelno programiranje može biti korisno su:

- primenjena fizika[10]
- napredna grafika[11]
- elektrotehnika[10]
- finansijsko i ekonomsko modeliranje[10]
- molekulsko modeliranje[10]
- nacionalna bezbednost i nuklearno oružije[10]
- istraživanje nafte i gasa[10]
- veštačka inteligencija[11]
- kvantna mehanika[10] i još mnoge druge

4.2 Prednosti i mane

Prednosti paralelnog programiranja su:

- **Brzina** - Možemo postići bolje performanse jer su zadaci raspoređeni u thread-ovima koji rade paralelno.[12]
- **Poboljšan GUI** - Pošto zadaci obavljaju neblokirajući I/O, to znači da je GUI thread uvek slobodan da prihvati korisničke unose. Ovo dovodi do boljeg odziva.[12]
- **Istovremeno i paralelno pojavljivanje zadataka koji se pokreću paralelno** - Možemo istovremeno pokrenuti različite logike programiranja.[12]
- **Bolje korišćenje keš memorije korišćenjem resursa i bolje korišćenje CPU resursa** - Zadaci se mogu izvršavati na različitim jezgrima, čime se obezbeđuje maksimalna propusnost.[12]

Mane:

- **Promena konteksta** - Svaki thread radi na delu vremena koji mu je dodeljen. Kada vremenski odlomak istekne, dešava se promena konteksta, što takođe troši resurse.[12]

- **Nepredvidljivost** - pošto se paralelno programiranje oslanja na CPU jezgra, možemo dobiti različite rezultate na različitim konfigurisanim mašinama. Uvek treba da razumemo da je paralelno programiranje relativan koncept i da nešto što je funkcionisalo za druge može, ali ne mora, raditi za nas.[12]
- **Teško za programiranje** - paralelne programe može biti teško napisati u poređenju sa sinhronim verzijama.[12]
- **Kompleksno otklanjanje grešaka i testiranje** - Korišćenje više thread-ova pomaže da dobijete više od jednog procesora. Ali onda ovi thread-ovi treba da sinhronizuju svoj rad u zajedničkoj memoriji. Ovo može biti teško ispraviti - a još teže bez problema sa istovremenošću[12]. Evo dve uobičajene vrste problema sa više thread-ova koje može biti teško pronaći samo testiranjem i otklanjanjem grešaka:
 1. **Data Race** - Data race nastaje kada dva ili više thread-ova pristupaju deljenim podacima i pokušavaju da ih modifikuju u isto vreme bez odgovarajuće sinhronizacije. Ova vrsta greške može dovesti do kvarova ili oštećenja memorije.[11]
 2. **Deadlock** - Deadlock se javlja kada je više thread-ova blokirano dok se takmiče za resurse. Jedan thread je zaglavljen čekajući drugi thread, koji je zaglavljen čekajući prvi. Ova vrsta greške može dovesti do zaglavljivanja programa.[11]

5 Primeri

Danas većina programskih jezika podržava paralelno programiranje, ali, radi jednostavnosti i čitljivosti, koristićemo programski jezik Python. Uz Python standardnu biblioteku dolazi i modul *multiprocessing*, koji omogućava paralelno pokretanje više procesa.

5.1 Primer 1 - Preuzimanje sadržaja u pozadini

Jedna od najčešćih upotreba paralelnog programiranja je izvršavanje neke operacije u pozadini, dok glavni program nesmetano nastavlja sa radom.

U ovom primeru prikazan je program koji kreira novi proces, i pomoću njega preuzima prvih 20 cifara broja pi sa određene internet adrese i ispisuje ih. Za to vreme glavni proces nastavlja normalno sa radom.

```
p = multiprocessing.Process(target=download_pi)
p.start()
while (True):
    print("Glavni proces...")
    time.sleep(0.5)
```

U prvoj liniji koda kreira se proces *p* - objekat klase *Process*. Kao argument šaljemo naziv procedure koju proces treba da izvrši. I konačno ga pokrećemo metodom *start()*. Program neće čekati da se izvršavanje okonča, nego će nastaviti normalnim tokom. U naredne 3 linije nalazi se petlja koja će da ispisuje određeni tekst na svakih pola sekunde. Ovaj deo postavljen je kako bi prikazao da se procesi izvršavaju paralelno.

Definicija procedure *download_pi* koju proces izvršava:

```
def download_pi():
    req = requests.get("https://api.pi.delivery/v1/
```

```

pi?start=0&numberOfDigits=20")
req = req.json();
print("Prvih 20 cifara broja pi:", req["content"])

```

Rezultat pokretanja celog programa:

```

Glavni proces...
Glavni proces...
Prvih 20 cifara broja pi: 31415926535897932384
Glavni proces...
...

```

Vidimo da je paralelizacija uspela, jer bi se u suprotnom prvo završilo izvršavanje procedure `download_pi`, pa tek onda program nastavio s radom.

5.2 Primer 2 - Broj prostih brojeva u listi

Nekada je za jedan problem potrebno pokrenuti više procesa. To su obično izuzetno zahtevne komputacije, kao što je, na primer, kriptografija ili kompresija velike količine podataka, ili kompilacija koda. Iako možemo pokrenuti više procesa pojedinačno (slično prethodnom primeru), multiprocessing pruža mogućnost korišćenja klase `Pool`.

`Pool` predstavlja skup procesa koji će izvršavati isti zadatak. Prilikom inicijalizacije, potrebno je proslediti samo broj procesa, koji ne može biti veći od broja jezgara na procesoru.

U ovom primeru prikazan je program koji izračunava broj prostih brojeva u nekoj listi brojeva.

```

pool = multiprocessing.Pool(processes=4)
nums_list = generate_list(15000, 9, 12)
nums_list = chunks(nums_list, 4)
print(sum(pool.map(count_primes, nums_list)))

```

U prvoj liniji kreiramo `pool`, koji će sadržati 4 procesa. Dalje, generišemo listu `nums_list` (pomoću procedure `generate_list`), koja će sadržati 15000 nasumičnih brojeva u intervalu $[10^9, 10^{12}]$.

S obzirom da se, pri kreiranju procesa, oni "odvajaju" od glavne memorije programa, potrebno je proslediti im podatke na kojima će vršiti komputacije. Da bi program bio što efikasniji, treba podeliti podatke na približno jednake delove. Delova bi trebalo biti onoliko koliko i procesa. U ovom slučaju, koristimo proceduru `chunks`, koja će listu podeliti na 4 približno jednaka dela, i vratiti novu listu koja sadrži te 4 podliste.

U poslednjoj liniji koristimo metodu `map()`, i prosleđujemo joj proceduru `count_primes` i listu `nums_list`. Ova metoda će u svakom procesu (koji se nalazi unutar skupa `pool`) pozvati proceduru `count_primes`, dok će kao argument proslediti jedan od elemenata liste `nums_list`, odnosno jedan od one 4 podliste. Nakon što svaki od procesa okonča izvršavanje, upisaće rezultat koji dobije u novu listu, koja će imati isto onoliko elemenata koliko i `nums_list`, odnosno 4. Procedura `sum` sabraće sve elemente novodobijene liste, a `print()` će dobijeni zbir ispisati.

Procedura `count_primes` definisana je na sledeći način:

```

def count_primes(lst):
    res = 0
    for x in lst:
        if is_prime(x):

```

```

        res += 1
    return res

```

Za argument prima listu brojeva. Zatim prolazi kroz datu listu, i za svaki broj proverava da li je prost pomoću procedure `is_prime`. Ako je broj prost, promenljiva `res`, koja predstavlja broj prostih brojeva u listi, povećava svoju vrednost za 1.

Ovako je definisana procedura `is_prime`:

```

def is_prime(n):
    if n == 1 or n == 2:
        return True
    for i in range(2, math.ceil(math.sqrt(n))):
        if n % i == 0:
            return False
    return True

```

Jedini parametar je broj n koji se testira. Zatim, za svaki ceo broj i u intervalu $[2, \lceil \sqrt{n} \rceil]$, proveramo da li je ostatak pri deljenju n sa i jednak nuli. Ako jeste, to znači da n nije prost broj, i procedura vraća vrednost *False*. Ako procedura dođe do kraja, a ne vrati *False*, to znači da delilac nije pronađen, te da je broj n prost broj.

5.3 Poređenje performansi

Sada ćemo testirati efikasnost paralelnog programiranja. Za to ćemo iskoristiti poslednji primer - brojanje prostih brojeva u listi. Upoređićemo vreme izvršavanja sa 1, 2, 4, 8 i 12 paralelnih procesa.

Da bi merili vreme izvršavanja, potrebna je samo jedna mala izmena u kodu. Pomoću metode `time.time()`, sačuvaćemo vremenski trenutak pre početka izvršavanja glavnog zadatka. Zatim to isto uraditi i tačno nakon izvršenja. Razlika ta dva vremenska trenutna predstavlja vreme izvršavanja zadatka.

```

start = time.time()
print(sum(pool.map(count_primes, nums_list)))
print(time.time() - start)

```

Za 5000 nasumičnih brojeva u intervalu $[10^9, 10^{12}]$, i 4 procesa koja izvršavaju zadatak, rezultat je sledeći:

```

203
6.540586948394775

```

Vreme je prikazano u sekundama.

Ovde ću ubaciti sliku sutra ili prekosutra. I još nešto malo treba da dodam, zaboravio sam šta.

6 Zaključak

Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak.

Literatura

- [1] G. V. Wilson. *The History of the Development of Parallel Computing*. CS Dept. NSF-Supported Education Infrastructure Project, 1994. online at: <https://ei.cs.vt.edu/history/Parallel.html>
- [2] J. P. Anderson, S. A. Hoffman, J. Shifman, and R. J. Williams. *D825 — A Multiple-Computer System For Command & Control*. Burroughs Corporation, Pennsylvania, 1962.
- [3] P. H. Enslow Jr. *Multiprocessor Organization — A Survey*. School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Georgia, USA, 1977.
- [4] Proceedings of the April 18-20, 1967, spring joint computer conference. Association for Computing Machinery, New York, NY, USA, 1967.
- [5] G. Ostrouchov. *Parallel Computing on a Hypercube: An Overview of the Architecture and Some Applications*. Oak Ridge National Laboratory, USA, 1987.
- [6] C. L. Seitz. *The Cosmic Cube*. California Institute of Technology, California, USA, 1985.
- [7] D. W. Walker. *Standards For Message-Passing in a Distributed Memory Environment*. Mathematical Sciences Section Oak Ridge National Laboratory, Tennessee, USA, 1992.
- [8] *MPI forum*. <https://www.mpi-forum.org/docs/>, 13. 11. 2022.
- [9] *The OpenMP API specification for parallel programming*. <https://www.openmp.org/specifications/>, 14. 11. 2022.
- [10] *TotalView: What Is Parallel Programming?* <https://totalview.io/blog/what-is-parallel-programming>, 17. 11. 2022.
- [11] *Perforce: What Is Parallel Programming and Multithreading (Multithreaded Programming)?* <https://www.perforce.com/blog/qac/multithreading-parallel-programming-c-cpp>, 17. 11. 2022.
- [12] Shakti Tanwar. *Hands-On Parallel Programming with C# 8 and .NET Core 3: Build solid enterprise software using task parallelism and multithreading*, 2019.
- [13] Dr. Daniel C. Hyde. *Introduction to the Principles of Parallel Computation*. Bucknell University, Lewisburg, 1995.
- [14] Bernard Goossens, Philippe Langlois, David Parelo and Eric Petit. *PerPI: A Tool to Measure Instruction Level Parallelism*. DALI Research Team, University of Perpignan Via Domitia, France
- [15] ZDNet, James Reinders. *Understanding task and data parallelism*. <https://www.zdnet.com/article/understanding-task-and-data-parallelism-3039289129/>
- [16] Christoph Kessler and Jörg Keller. *Models for Parallel Computing: Review and Perspectives*. Dept. of Computer Science (IDA) Linköping university, Sweden, Dept. of Mathematics and Computer Science, Fernuniversität Hagen, Germany, December 2007.