Konkurentno programiranje - Zadatak, nit, proces, izbor nivoa konkurentnosti

Isidora Ristović i Marijana Urošević
mr16272@alas.matf.bg.ac.rs, mv16186@alas.matf.bg.ac.rs

10. novembar 2018.

1 Uvod

Mnogi domeni problema prirodno navode na korišćenje konkurentnosti na isti način na koji se rekurzija nameće kao najbolja za dizajniranje rešenja nekih problema. Takođe, dosta programa se piše da simulira dešavanja iz realnog sveta. U tim slučajevima, sistem koji se simulira često ima više od jedne zasebne celine koje postoje istovremeno i funkcionišu svaka na svoj način – na primer, avion u letu šalje obaveštenja nadležnoj stanici i u istom vremenskom trenutku mora imati kontrolu nad svojom opremom. Ovakav problem nesmetano može rešiti samo softver koji koristi konkurentnost [1].

Prvi korak u razvoju bilo kakvog softvera je da se dobro razume problem koji želimo da rešimo. Pre pokušaja razvoja konkurentnog rešenja za problem, neophodno je utvrditi da li se program uopšte može rešiti na taj način i, ako može, da li je vremenski isplativo pisati ga (poznata izreka: ,,Možeš provesti čitav životni vek paralelizujući kod i nikad ga ne ubrzati više od 20 puta bez obzira na to koliko procesora imaš!"[2]).Slede dva jednostavna primera:

Primer 1.1 *Izračunati potencijalnu energiju za svaki od nekoliko hiljada nezavisnih struktura molekula i nakon toga naći minimalnu od njih. Ovaj problem se može rešiti paralelno jer svaku od molekularnih struktura možemo izračunati nezavisno. Čak se i minimalna energija može ovako naći [2].*

Primer 1.2 *Računanje Fibonačijevog niza (0,1,1,2,3,5,8,13,21...) korišćenjem formule F(n) = F(n-1) + F(n-2) . Za računanje vrednosti F(n) potrebne su vrednosti F(n-1) i F(n-2), što znači da ih moramo prethodno izračunati, te ovde nijedan deo procesa ne možemo paralelizovati [2].*

2 Zadatak, nit, proces

Zadatak (eng. *task*) je jedinica programa ili skup instrukcija koje izvršava procesor. Zadatak možemo zamisliti kao funkciju (potprogram), ali postoji bitna razlika između funkcije koja se poziva eksplicitno (funkcije sa kakvom smo se upoznali u C-u) i zadatka. Funkcija koja se poziva eksplicitno pauzira delovanje glavnog dela programa i program skače se na njeno izvršavanje, a zadatak se može izvršavati nezavisno od nekog drugog zadatka pa se samim tim, na

kraju zadatka, kontrola ne mora vratiti na mesto njegovog ,,poziva" (koji može biti implicitni). Paralelni programi sadrže više zadataka koji se istovremeno izvršavaju na više procesora [1].

Zadaci mogu imati nekoliko različitih stanja:

1. Nov (eng. *new*) - zadatak je nov kada je tek kreiran, ali nije još započeto njegovo izvršavanje.

2. Spreman (eng. *ready*) - spremni zadaci su oni koji su spremni za izvršavanje, ali ih tajmer[1] još uvek nije dodelio procesoru ili su pre toga bili aktivni i iz nekog razloga su blokirali u toku izvršavanja. Ovi zadaci se čuvaju u strukturi koja se obično naziva red spremnih zadataka (eng. *task ready queue*)

3. Aktivan (eng. *running*) - onaj zadatak koji se trenutno izvršava u procesoru

4. Blokiran (eng. *blocked*) - blokiran zadatak je onaj čije je izvršavanje prekinuto zbog jednog ili više razloga. Najčešće je to zbog greški prilikom ulaznih i izlaznih operacija.

5. Ugašen (eng. *dead*) - ugašen zadatak je onaj koji više nije aktivan ni u jednom smislu. Zadatak se gasi kada se uspešno izvrši ili kada ga program eksplicitno prekida [1].

Ako je bilo kom zadatku potrebno više od, otprilike, desetine sekunde za izračunavanje (što je prag čovekovog opažanja), onda moramo podeliti taj zadatak na delove, između kojih ćemo čuvati stanje i vraćati se na vrh petlje [3].

Zadatke u opštem slučaju delimo na teške i lake, u smislu koliko prostora (memorije) i vremena nose sa sobom. **Teški zadaci** imaju svoj sopstveni adresni prostor dok **laki** dele isti adresni prostor. To je njihova suštinska razlika. Težak zadatak u sebi može obuhvatati više lakih zadataka koji se konkurentno izvršavaju (2.1). Obično, u ovoj[2] terminologiji, teški su procesi, a lake su niti. Ono što je suština je da teškim upravlja operativni sistem, obezbeđujući deljenje procesorskog vremena. Lakše je implementirati lak zadatak, nego težak zadatak. Osim toga, lak zadatak može biti efektivniji od teškog zadatka, jer manje truda je potrebno za upravljanje njihovim izvršavanjem. Stanje teškog zadatka obuhvata: podatke o izvršavanju (stanje izvršavanja koje može biti: spreman, radi, čeka...), vrednosti registara, brojač instrukcija, informacije o upravljanju resursima (informacije o memoriji, datoteke, ulazno-izlazni zahtevi i ostali resursi) [4].

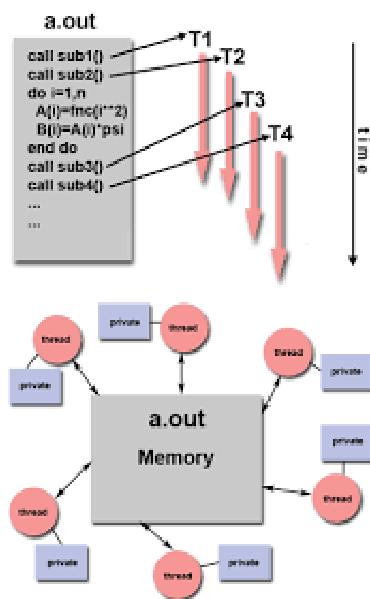
Unutar konkurentnog programa, koristićemo izraz *nit* za aktivnu celinu. Više tih celina se može izvršavati u istom vremenskom periodu. U većini sistema, niti datog programa se izvršavaju u okviru jednog ili više procesa, obezbeđenih od strane operativnog sistema. To znači da se niti izvršavaju nezavisno od glavnog programa, ali i u istom vremenskom periodu sa njegovim izvršavanjem. Nije

[1] aktivan sistemski program koji omogućava deljenje procesorskog vremena među zadacima.

[2] Nažalost, terminologija je nekonzistentna u okviru različitih sistema i literature različitih autora što dodatno otežava snalaženje i razumevanje termina. Nekoliko programskih jezika niti naziva procesima, dok se na primer u Adi oslovljavaju kao niti. Mi ćemo koristiti terminologiju jezika C i C++

bitno šta će se pre izvršiti. Glavni deo programa može biti obavešten o kraju izvršavanja niti, a i ne mora. Niko nikoga ne čeka (za razliku od funkcija u C-u). Korišćenjem više niti obezbeđuje se da brze operacije (prikazivanje teksta npr.) ne čekaju spore operacije. Takođe, kad god neka nit zablokira (čeka određenu poruku ili zbog smetnji ulazno-izlaznih jedinica), implementacija omogućava da se automatski pređe na sledeću nit.

Primer 2.1 *Kada operativni sistem pokrene izvršni fajl (a.out), a.out pribavlja sve neophodne resurse za rad – to spada u težak proces. Izvršni fajl zatim kreira gomilu niti koje operativni sistem izvršava konkurentno. Svaka nit ima lokalne podatke ali i deli čitave resurse koje je pribavio a.out. Ovo smanjuje opšte troškove koje bismo imali dupliranjem resursa koje koriste sve niti (zato je ovo lak proces). Niti dolaze i odlaze, ali a.out ostaje prisutan da bi obezbedio neophodne zajedničke resurse sve dok se aplikacija nije do kraja završila (pogledati sliku 1) [2].*



Slika 1: a.out primer

Operativni sistem igra veoma značajnu ulogu u konkurentnom programiranju. On vrši razmenu izvršavanja zadataka tj. odlučuje kad će koji zadatak doći na red za izvršavanje. Operativni sistem takođe daje ograničenje broja procesa (ne može ih biti beskonačno mnogo), dok sa nitima skoro da nemamo ta ograničenja jer se te granice skoro nikad ne dostižu [4].

3 Promena konteksta

Promena konteksta podrazumeva da je potrebno zapamtiti u memoriji celokupno stanje zadatka koji se završio (prebacujemo iz registara u memoriju sve

vezano za prekinuti zadatak) i na osnovu informacija u memoriji rekonstruisati stanje onog zadatka koji sad treba da se izvrši (dovlačimo iz memorije u registre sve neophodno za tekući zadatak). Promena konteksta je jako skupa (a vrši se i po hiljadama puta u jednoj sekundi) i može značajno da uspori rad programa.

Kod teških zadataka dolazi do promašaja u kešu, što značajno utiče na performanse sistema. Keš memorija je najbrža, a pri smenjivanju procesa, često umesto da dohvatamo podatke iz keša dohvatamo ih iz glavne memorije što je značajno sporije, jer ako imamo nekoliko hiljada puta u jednoj sekundi tih smena i uzmemo u obzir promašaje u kešu onda to govori da je jako važno znati izabrati da li ćemo koristiti teške zadatke ili lake zadatke, to jest znati u kojim situacija se šta koristi.

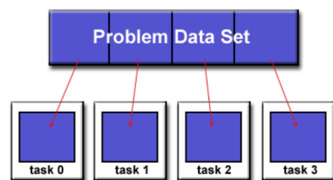
To se vidi ako napišemo jednostavan program koji napravi veliki broj procesa koji nešto rade (ali procesa u smislu teških zadataka) i program koji isto to radi samo ne pravi procese nego niti (odnosno lake zadatke). S jedne strane imamo smenjivanje procesa a sa druge smenjivanje lakih zadataka (Laki su, jer ne vuku sve informacije o memoriji, datoteke, vrednosti registara i ostale resurse sa sobom, već neke delove, koji su samo za njihovo izvršavanje neophodni. Stoga je kod njih smena konteksta mnogo jeftinija.). Onda vidimo, da ako pogrešno izaberemo i problem razdelimo na teške zadatke, dobijamo program koji je neupotrebljiv, kome treba puno vremena, dok u suprotnom ako izabremo lake zadatke, program se značajno brže izvršava.

Koncept lakih zadataka je i nastao baš sa ciljem da se prevaziđu ovi problemi. Odnosno kada nije potrebno da baš sve pamtimo nego nam je, na primer, u okviru iste aplikacije potrebno da zamenimo koji će zadatak da nam uradi posao. Oni imaju zajednički adresni prostor (promenljive, brojače, datoteke ili nešto drugo) i ideja je da se prilikom promene konteksta sve zajedničko zadržava, umesto da te resurse šetamo od registara do memorije,pa opet nazad iz memorije u registre.

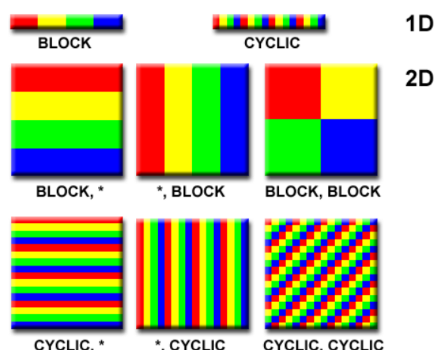
4 Razlika između konkurentnosti u užem smislu i paralelnosti

Konkurentnost u užem smislu podrazumeva sistem izgrađen od skupa komponenata koje se izvršavaju na jednom procesoru. Komponente međusobno interaguju na unapred određen i dobro kontrolisan način da ne bi došlo do sporednih efekata. Kaže se da su ove komponente ili procesi *isprepletani* u procesoru jer se njihovo izvršavanje u njemu smenjuje, a uz to, samo jedan proces može biti aktivan u jednom trenutku.

Paralelnost govori da više od jednog procesora izvršava procese i da oni imaju obezbeđenu međusobnu komunikaciju na neki način. To može biti *multi-core* procesor sa jednom memorijom ili distribuirani sistem. U okviru paralelnog sistema veoma je verovatno da će neki procesori izvršavati određene zadatke (procese) konkurentno u užem smislu [5].



Slika 2: Primer paralelizacije podataka



Slika 3: Načini podele podataka

5 Paralelizacija zadataka ili podataka (izbor nivoa konkurentnosti)

Programer se susreće sa pitanjem šta je paralelizam, kako napraviti paralelnu aplikaciju, šta je nabolje načiniti da bude paralelno i kako najbolje podeliti posao, a odgovor nikad nije jedinstven. Treba sagledati situaciju i mogućnosti i na osnovu različitih informacija i doneti odluku.

5.1 Particionisanje

Jedan od prvih koraka u pravljenju paralelnog programa je razbijanje problema na odvojene ,,komade" posla koji mogu biti raspodeljeni na različite zadatke (niti i procese). Ovo se naziva dekompozicija ili particionisanje. Postoje dva osnovna načina particionisanja računarskog posla među zadacima (nitima i procesima): paralelizacija podataka (eng. *domain decomposition*) i paralelizacija zadataka (eng. *functional decomposition*).

5.1.1 Paralelizacija podataka

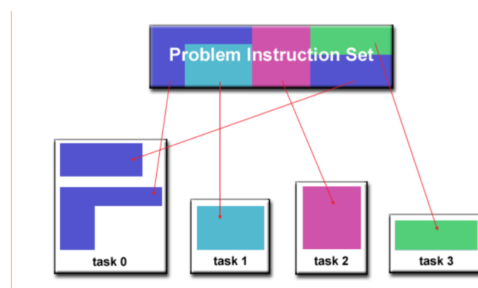
U ovom tipu paralelizacije, problem se dekomponuje po podacima. Što znači da svaki paralelni zadatak (nit ili proces) radi na delu podataka (pogledati sliku 2).

Primer 5.1 *Ukoliko želimo da primenimo neki efekat na sliku, tu sliku možemo podeliti na grupu manjih celina (kvadratića), a onda na svaki od tih delova primeniti željeni efekat. Na taj način nam algoritam dobro skalira jer dodavanjem novih procesorskih resursa možemo smanjvati te kvadratiće i brže odraditi posao (slika je matrica piksela, a mi nad svakim pikselom radimo istu stvar) [4].*

Postoje različiti načini podele podataka (pogledati sliku 3).

5.1.2 Paralelizacija zadataka

U ovom pristupu, fokus je je na računanju koje se izvršava, a ne na podacima nad kojima se računanje vrši. Problem se dekomponuje prema poslu koji treba



Slika 4: Primer paralelizacije zadataka

odraditi. A zatim svaki zadatak (nit ili proces) rešava deo celokupnog posla (pogledati sliku 4).

Paralelizacija zadataka je podesna za probleme koji mogu biti podeljeni na više različitih zadataka. Na primer:

Primer 5.2 *Modelovanje klime - Svaka komponenta modela se može posmatrati kao zaseban zadatak (nit ili proces). Strelice prikazuju razmenu podataka među komponentama tokom računanja: model atmosfere daje podatke o brzini vetra koji se koriste u modelu okeana, model okeana daje podatke o temperaturi površine mora koji se koriste za atmosferski model, i tako dalje.*

Kombinovanje ova dva tipa paralelizacije je često i prirodno [2].

Primer 5.3 *Obrada signala - Skup podataka zvučnog signala prolazi kroz četiri zasebna računarska filtera. Svaki filter je zaseban proces. Prvi segment podataka mora proći kroz prvi filter pre prelaska na drugi filter. Kada stigne do drugog filtera, drugi segment podataka prolazi kroz prvi filter. U trenutku kada je četvrti segment podataka u prvom filteru, sva četiri zadatka su zauzeta.*

5.2 Primer

Za svaki broj iz niza brojeva odrediti koja od narednih svojstva ispunjava:

- broj je prost
- broj je savršen
- broj je deljiv sumom svojih cifara
- broj ima paran broj delilaca
- broj je jednak zbiru kubova svojih cifara

Rešenje: Prva ideja koja nam ,,padne" na pamet, jeste da svako od ovih pet svojstava posmatramo kao zasebnu nit. Što nije naročito dobro rešenje po pitanju skalabilnosti, ukoliko imamo više od 5 procesora, a i ukoliko primetimo da nam svojstva imaju nekih dodirnih tačaka,pa bismo određene poslove vršili više puta (na primer, ako je broj prost,znamo da on sigurno nema paran broj delilaca,kao i da nije savršen,pa zašto bismo to proveravali).

Prirodno želimo da ta računanja koja se ponavljaju izdvojimo u zasebnim nitima. Ali kakvi su uopšte ti brojevi sa kojima radimo? Jesu li integer-i? Može

li se primeniti procesorsko deljenje na njih? Šta ako su stocifreni? Šta ako nam je baš to deljenje skupa operacija? A osim toga ni ovo rešenje nije naročito skalabilno. A ukoliko imamo mašinu sa stotinu ili čak hiljadu procesora, mnogo njih će biti neiskorišćeno.

Za dobro skaliranje na velikom broju procesora potreban je paralelizam podataka. Zato bismo mogli da za svaki broj iz niza računamo svih pet stvari zajedno. Da izdelimo naš niz na onoliko delova koliko imamo procesora i da svaka nit računa ovih pet osobina zajedno. Ovo rešenje je skalabilno. I dobro će funcionisati ukoliko imamo niz sa brojevima koji imaju uniformnu raspodelu po pitanju vremena potrebnog za njihovu obradu.

Ali kakav je naš niz uopšte? Da li njegovi elementi imaju uniformnu raspodelu? Jer ako nemaju, prethodno rešenje nije najbolje. Onda je bolje slati nitima da obrađuju redom broj po broj iz niza. Ali tada se javlja pitanje, kojim redom da uzimaju nove brojeve? Da li nit može da uzme naredni broj ili ga neka druga nit već obrađuje? Niti moraju da komuniciraju. Gubi se vreme na čekanju.

Zato ne bi bilo loše da nitima šaljemo veće delove niza, kako bi smanjili vreme izgubljeno na komunikaciju i čekanje. Da li onda da uzimamo po (na primer) 50 elemenata niza i šaljemo svakoj niti? Da li će ti delovi koje šaljemo, svim nitima zahtevati približno isto vremena za obradu? To sve već zavisi od datog niza. I jasno je da je teško odlučiti šta je najbolje rešenje, jer zavisi od mnogo faktora – od broja procesora, datih podataka [4]...

5.3 Algoritmi za paralelno sortiranje

Najbolji sekvencijalni algortitmi sortiranja (merge sort i quick sort) imaju, kao što znamo, složenost $O(n \log n)$. Pitamo se, da li se paralelizacijom može postići nešto bolje. Operacija koja predstavlja osnovu nekoliko klasičnih sekvencijalnih algoritama sortiranja je uporedi-i-razmeni (eng. *compare-and-exchange*). Funkcioniše tako što se dva broja A i B uporede i ako nisu u ispravnom redosledu, zamene se. U suprotnom ostanu na svom mestu. Predložićemo dva načina njene paralelizacije:

Verzija 1: P1 šalje broj A u P2, koji poredi A i B i vraća nazad u P1 min(A,B).

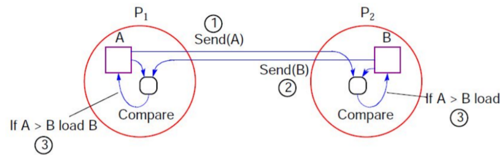
Verzija 2: P1 šalje broj A u P2 i P2 šalje broj B u P1, oba izvrše poređenje i P1 zadrži min(A,B), a P2 zadrži max(A,B) (pogledati sliku 5)

Napomena: Pretpostavili smo da postoji jedna procesorska jedinica za svaki od brojeva, ali u realnom slučaju ćemo imati mnogo više brojeva (n) od procesorskih jedinica (P), što znači da će svakoj procesorskoj jedinici biti dodeljena lista od n/P brojeva.

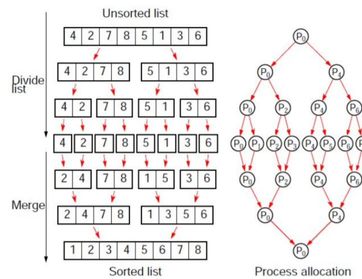
Merge sort je algoritam sortiranja koji radi po principu podeli-pa-vladaj. On deli listu[3] na dve polovine (čija se dužina razlikuje najviše za 1), rekurzivno sortira svaku od njih, i zatim objedinjuje sortirane polovine. Ideja za paralelni merge sort koristi ,,drvenastu strukturu" (pogledati sliku 6) algoritma koja se dobija ako svaku od polovina listi podelimo na još dve liste, pa još dve i tako dok ne ostanu jednočlane liste, a onda ih učešljavamo rekurzivno.

Ako zanemarimo vreme potrebno za komunikaciju između procesorskih jedinica, vidimo da se računanja pojavljuju samo prilikom učešljavanja listi (eng. *mer-*

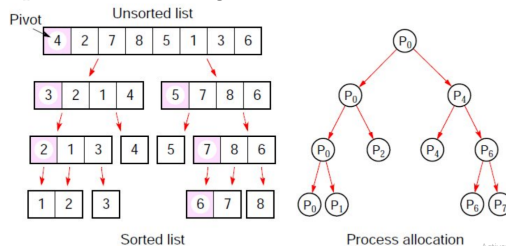
[3]korističemo listu umesto uobičajenih nizova



Slika 5: Compare-and-exchange



Slika 6: Paralelni merge sort



Slika 7: Paralelni quick sort

ging – čak i ovaj proces može biti paralelizovan što je detaljnije objašnjeni u literaturi [6]). U najgorem slučaju nam je potrebno $2s$–1 koraka da učešljamo sve podliste veličine s, a kako ukupno ima $\log n$ učešljavanja, biće

$$\sum_{i=1}^{\log n} (2^i - 1)$$

koraka za dobijanje konačne sortirane liste, što odgovara složenosti O(n).

Quick sort takođe koristi podeli-pa-vladaj princip. Kao i kod merge sort-a, za paralelizaciju je ideja da se iskoristi ,,drvenasta struktura" (pogledati sliku 7) algoritma koja se dobija ako početnu listu podelimo , u odnosu na pogodno odabran pivot, na dve tako da su u jednoj svi brojevi manji od pivota, a u drugoj veći. Taj proces ponavljamo dok ne ostanemo sa jednočlanim listama. Napomena: nailazimo na problem jer u opštem slučaju drvo koje pravimo ne mora biti savršeno balansirano (ovde je od ključne važnosti odabir dobrog pivota).

Ako zanemarimo vreme potrebno za komunikaciju i pretpostavimo da je izbor pivota idealan (tj. da su kreirane podliste jednakih veličina), onda nam je

$$\sum_{i=0}^{\log n} n/2^i \approx 2n$$

koraka za dobijanje sortirane liste, što odgovara složenosti O(n).

Postoji još paralelnih algoritama sortiranja o kojima se više možete informisati ovde: [6], [7].

6 Zaključak

Postoje različiti stavovi u vezi kompleksnosti konkurentnog programiranja.

Na jedan način, konkurentne aplikacije su mnogo složenije od odgovarajućih sekvencijalnih aplikacija. Cena složenosti se vidi u vremenu koje programeru odlazi na sve aspekte razvojnog ciklusa za softver: dizajn, pisanje koda, debagovanje, održavanje koda. S druge strane , pravljenje i dizajniranje konkurentnih i paralelnih sistema postaje značajno jednostavnije ako se koristi odgovarajuća podrška jezika ili odgovarajuće biblioteke (po nekim autorima konkurentno programiranje je jednostavnije od sekvencijalnog, ako se koriste pogodni alati). Koliko je podrška značajna, govori nam poređenje koje kaže da je bez nje konkurentno pisanje programa kao korišćenje asemblera za kreiranje kompleksnih korisničkih interfejsa za pristup bazama podataka [5].

Literatura

[1] Robert W. Sebesta. *Concepts of Programming Languages*. Pearson, 10th edition, 2012.

[2] Lawrence Livermore National Laboratory Blaise Barney. Introduction to parallel computing.

[3] Michael L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[4] Konkurentno programiranje,predmet: Dizajn programskih jezika, Novembar 2018.

[5] Jon Kerridge. *Using Concurrency and Parallelism Effectively – I*. bookboon.com, 2nd edition, 2015.

[6] Ricardo Rocha and Fernando Silva. Parallel sorting algorithms.

[7] Michael Hanke. Parallel sorting.