SVEUČILIŠTE U SPLITU

PRIRODOSLOVNO-MATEMATIČKI FAKULTET

POSLIJEDIPLOMSKI SVEUČILIŠNI STUDIJ: ISTRAŽIVANJE U EDUKACIJI U PODRUČJU PRIRODNIH I TEHNIČKIH ZNANOSTI, SMJER INFORMATIKA

Seminarski rad

MODELI ISTOVREMENOG PROGRAMIRANJA

Kolegij: Distribuirani informacijski sustavi

Profesor: red. prof. dr. sc. Maja Štula

Student: Marin Aglić Čuvić

Split, ožujak 2017.

Sadržaj

1 Uvod

Moore-ov zakon (1965) tvrdi da će se broj tranzistora na procesoru jednake veličine svako dvije godine otprilike poduplati [1]. Ovo je moguće samo dok veličinu tranzistora budemo mogli smanjivati što je postalo problematično posljednjih godina. S obzirom na ograničenja materijala sve je teže držati korak sa Moore-ovim zakonom te se mogu pronaći izvori koji tvrde da mu je došao kraj. No, Intel i dalje radi na tome da veličinu tranzistora dodatno smanji. Tako da danas u računalima nalazimo tranzistore od svega 14 nm (2010. godine su bili 32 nm), a postoje naznake da bismo 2018. godine mogli vidjeti i tranzistore velike svega 10 nm [2]. S obzirom na plan Intel-a da nakon procesora sa 10 nm tranzistorima predstavi dva poboljšanja tih tranzistora prije sljedećeg smanjenja, može se pretpostaviti da će se broj tranzistora i dalje povećavati, ali možda nešto sporije nego što je predviđeno Moore-ovim zakonom.

Smanjenjem veličine tranzistora i povećanjem njihova broja u procesoru se mogu poboljšati opće performanse procesora, ali se javlja drugi problem. Naime, po pravilu Dennardovog skaliranja smanjenjem veličine tranzistora, smanjuje se i potreban napon što dovodi do toga da, u idealnim uvjetima, procesor jednake veličine pri istom utrošku energije može raditi na većoj frekvenciji [3, 4]. No, Dennardov zakon je zanemario i) pod-granično "curenje" (engl. *sub-threshold leakage*) električne energije, koje se smanjenjem tranzistora povećalo, i 2) granični napon izmjene stanja tranzistora kojem su se skalirani (smanjeni) naponi približili [3, 5]. Pod-granično curenje je gubitak energije koji se događa dok je tranzistor isključen uslijed difuzijskog protoka električne energije [6]. S obzirom na gubitak struje i poteškoće daljnjeg skaliranja graničnog napona, pri većim frekvencijama se poveća utrošak energije po jedinici površine čipa što ograničava brzinu takta procesora. Dodatno, pri većem taktu procesora se stvara veća količina topline koja može oštetiti procesor što znači da trebaju efikasniji načini odvoda topline.

Iz prethodne rasprave se može zaključiti da su Dennardov i Moore-ov zakon povezani. Smanjenjem tranzistora bi se i povećala frekvencija rada procesora. Stoga su se programeri mogli osloniti na to da će se performanse njihovih aplikacija poboljšati dolaskom nove generacije procesora. S obzirom na to da je dosegnuta granica brzine takta, došao je kraj Dennardovom zakonu [3]. Stoga je zaključeno da se bolje performanse mogu ostvariti dodavanjem većeg broja jezgri na procesor. Sekvencijalnim programiranjem, programeri mogu ostvariti minimalnu iskoristivost ovih jezgri dok veći dio procesora ostaje neiskorišten (barem od strane aplikacije programera, npr. operacijski sustav može i dalje može koristiti te ostale jezgre). Cilj istovremenog i paralelnog programiranja je postići što veću iskoristivost dostupnih jezgri procesora, odnosno, što bolje iskoristiti dostupne resurse. Aplikacije koje mogu iskoristiti ove resurse uspješno ćemo zvati *istovremene aplikacije.* Postoje dvije metodologije [7] i nekoliko modela istovremenog i paralelnog programiranja koje programer može iskoristiti za izradu ovakvih aplikacija. Ove paradigme programiranja sa sobom donose nove poteškoće i izazove.

2 Programski jezik Scala

U ovom seminaru se za demonstracije primjera pogrešaka i primjene različitih metoda istovremenog programiranja koristi programski jezik Scala, što stoji za "*scalable language*" [8]. Scala je hibridni (objektno-orijentirani i funkcionalni) programski jezik koji se pokreće na JVM-u (*Java Virtual Machine*). Štoviše, Scala i Java su potpuno interoperabilne, tj. metode jednog jezika se mogu koristiti u drugom.

Scala je dizajnirana na način da bude proširiva [8]. Naime, omogućuje programeru da proširi postojeće tipove podataka. Sve što programer mora napraviti jest deklarirati novu klasu i deklarirati implicitnu

metodu koja će napraviti konverziju iz zadanog tipa u novi. Potom, svaki put kada se u programskog kodu pojavi taj tip podatka, Scala napravi automatsku konverziju u novo definirani. Taj podatak i dalje ima sve metode koje je originalno imao uz metode koje je programer dodao. Štoviše, Scala omogućuje deklariranje vlastitih kontrolnih struktura i operatora na način da se njihovo korištenje čini kao da su dio jezika.

Osnovni razlozi korištenja Scale:

1) Autor seminara poznaje programski jezik;
2) Hibridni programski jezik koji podržava sve najvažnije aspekte funkcionalnog programiranja (C# ih podržava tek od verzije 7.0 [9]);
3) Podržava sve metode istovremenog programiranja koje ćemo demonstrirati;
4) Podržava Reactore (engl. *Reactors*)[10] koji su relativno nov programski model, kojeg spominjemo u poglavlju 8.1.

Scala je dizajnirana na način da programer može izostaviti elemente sintakse koji su tipični za jezike slične C-u. Tako programer može i) izostaviti ";" na kraju svakog reda; ii) izostaviti ključnu riječ "return" u metodi; iii) pozvati metodu bez pisanja znaka "." i iv) ukoliko metoda ili konstruktor ne primaju parametre, programer ih može pozvati bez pisanja praznih zagrada. Ova pojednostavljenja sintakse se mogu pronaći u našim primjerima. Na primjer, poziv `.par` u Dodatak B je poziv metode, a ne pristup svojstvu kao što se možda čini. Štoviše, primjer koji demonstrira nekoliko pojednostavljenja sintakse je pozive metode `to` u izrazu `1 to 100`, a koju smo mogli pozvati i na način `1.to(100)`. Uz pojednostavljenja sintakse, u ovom izrazu vidimo i proširenje postojećeg tipa `int`, kojim se deklarira nova metoda `to`.

3 Procesi i niti

Proces je dinamička instanca programa koja se izvršava izolirano i nezavisno, a kojoj operacijski sustav dodjeljuje resurse poput memorije, pristup datotekama, sigurnosni kontekst i jedinstveni identifikator [8, 9]. Ukoliko je potrebno, procesi mogu međusobno komunicirati. Neki od načina da se ta komunikacija ostvari su: soketi, semafori, dijeljena memorija i datoteke.

Operacijski sustavi omogućuju pokretanje većeg broja procesa istovremeno, pri čemu to mogu biti i instance istog programa. Faktori koji su doveli do razvoja ovakvih operacijskih sustava su [11]:

- **Iskorištavanje resursa** – određeni programi moraju čekati na nekakav događaj ili resurs da se oslobodi. S obzirom da ne mogu odrađivati nikakav koristan posao dok čekaju, onda je efikasnije prepustiti izvršavanje nekom drugom programu;
- **Poštenje** – u računalu obično nekoliko programa ima jednaka prava na korištenje računalnih resursa. Umjesto da se pusti jedan program da se izvrši u potpunosti prije nego drugi dobije pravo na resurs, preferira se reguliranje pristupa resursu korištenjem vremenskog intervala. Nakon što jedan proces iskoristi svoje vrijeme, drugi dobije pravo pristupa;
- **Pogodnost** – različiti programi mogu koordinirati svoje radnje, što znači da umjesto jednog složenog programa možemo imati više jednostavnijih koji će se međusobno koordinirati. Ovakve jednostavnije programe može biti lakše realizirati te su mogu biti poželjniji.

Iz istih ovih razloga su razvijene niti. To su entiteti koji omogućuju da unutar programa postoji više tokova izvršavanja. Pri tome proces i niti dijele pristup datotekama i memoriju koju je operacijski sustav dodijelio procesu. Svaka nit ima svoj programski brojač, stog i lokalne varijable. Svaki proces ima

najmanje jednu nit koju zovemo *primarna nit*, a iz bilo koje niti može kreirati dodatne. Većina modernih operacijskih sustava tretira niti kao osnovnu jedinicu koja koristi računalne resurse i na kojoj se temelji raspored (zakazivanje, engl. *scheduling*) korištenja resursa [11].

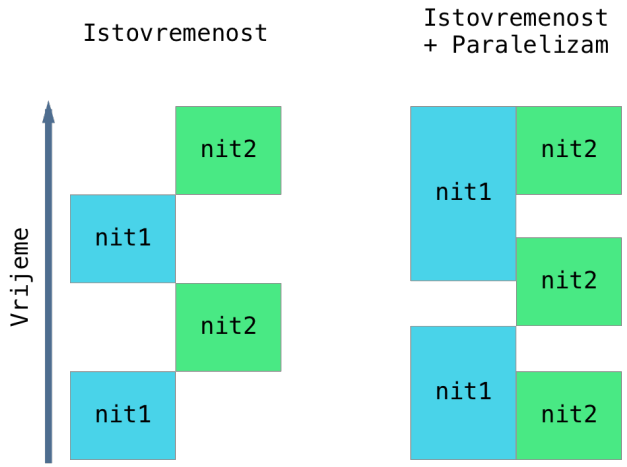
4 Istovremeno ili paralelno izvršavanje?

Koja je razlika između istovremenog i paralelnog izvršavanja? Nekome se može činiti da se ovi pojmovi odnose na istu pojavu, odnosno na istu sposobnost neke aplikacije, ali to nije točno [7] te ćemo ih u ovome seminarskom radu razlikovati. Pojmove "istovremeno" i "paralelno" definiramo na sljedeći način [8, 9]:

- Paralelno – situacija u kojoj se barem dvije niti izvršavaju u istom trenutku. Da bi paralelizam bio moguć, procesor mora imati barem dvije jezgre.
- Istovremeno – odnosi se na situaciju u kojoj izvršavanje barem dviju niti napreduje.

S obzirom na definicije istovremenog i paralelnog izvršavanja, možemo zaključiti da je paralelno izvršavanje ujedno i istovremeno, ali istovremeno nije paralelno. Štoviše, istovremeno izvršavanje je moguće i na jednoprocesorskom računalu. Procesor jednostavno mora osigurati da se zamijeni nit koja se trenutno izvršava s nekom od onih koje čekaju na izvršavanje. S obzirom na brzinu mijenjanja niti na procesoru, korisniku se može činiti da program paralelno izvršava više zadataka.

Razliku između istovremenog i paralelnog izvršavanja možemo ilustrirati Slika 1.



Slika 1. Usporedba istovremenog i paralelnog izvršavanja niti

5 Istovremeno programiranje

Istovremeno programiranje se odnosi na organizaciju programa kao skup operacija, ili zadataka (engl. *task*), koje se mogu izvršiti u vremenskim razdobljima koja se preklapaju i koje je potrebno na neki način koordinirati [15]. Ukoliko se izvršavanje dviju ili više operacija vremenski preklapa one će biti pokrenute na različitim nitima.

Istovremenost je ključna za izradu sustava s adekvatnim odzivom (engl. *responsive system*) [16]. Tipičan primjer je da se osigura odziv grafičkog sučelja dok u pozadini program odrađuje nekakav složen zadatak ili komunicira sa jednim ili više eksternih izvora. Na nekoj razini, ovo osigurava modularnost aplikacije jer nit na kojoj je pokrenuto grafičko sučelje, s kojim je korisnik u interakciji, je

različita od niti za komunikaciju s npr. bazom podataka [17]. Ukoliko hardware to podržava, različite operacije u istovremenom sustavu se mogu izvršavati paralelno što može poboljšati performanse programa. S obzirom na raširenost višejezgrenih procesora, ovo je možda osnovni razlog povećanja popularnosti istovremenog programiranja [15].

Istovremenost omogućuje izradu sustava otpornih na pogreške – ukoliko se dogodi pogreška u jednoj operaciji, drugu operaciju se može obavijestiti o tome i sustav može samog sebe zaliječiti. Ono što je važno jest da su operacije adekvatno izolirane kako pogreška ne bi utjecala na ostatak aplikacije [16]. Samo zacjeljenje sustava je temelj danas popularne filozofije programiranja *"Let it crash"* o kojoj ćemo u kasnijem poglavlju reći nešto detaljnije.

U konačnici, istovremenost u nekim slučajevima može pojednostavniti implementaciju i održivost programa. Naime, neke programe je prikladnije razdvojiti u manje nezavisne operacije koje se mogu istovremeno pokrenuti [15].

5.1 Izazovi istovremenog programiranja

Sekvencijalni programi se izvršavaju na jednoj niti, što znači da se u bilo kojem trenutku može izvršiti najviše jedna instrukcija. Stoga, za bilo koji ulaz u program, uvijek ćemo dobiti isti rezultat (ako zanemarimo neodređenosti poput generatora slučajnih brojeva). Takav tip programa spada u skupinu *determinističkih* programskih modela.

Zanimljivo je da i paralelno programiranje spada u istu skupinu. Naime, dok je cilj istovremenog programiranja napraviti aplikaciju otpornu na pogreške s adekvatnim odzivnim vremenom, cilj paralelnog programiranja je isključivo poboljšati performanse aplikacije [11, 12]. Stoga se nastoji izbjeći bilo kakav nedeterminizam jer nepotrebno komplicira izvršavanje, razvoj i održavanje programa. Uostalom, kod nekih zadataka, npr. ako želimo paralelno inkrementirati sve elemente niza (jer je niz jako velik), se neće odjednom pojaviti nedeterminizam.

Istovremeno programiranje, s druge strane, spada u skup *nedeterminističkih* programskih modela kod kojih konačni rezultat izvršavanja aplikacije ne mora svaki put biti isti. Naime, u slučaju nedeterminističkih programa konačni rezultat može ovisiti o nekom od aspekata izvršavanja [17]. Tako rezultat istovremenog programa može ovisiti o točnom trenutku događaja [16], pri čemu taj događaj može imati eksterni izvor na koji program nema utjecaja.

Nedeterminizam koji je prisutan u istovremenim aplikacijama otežava samo razumijevanje njegovog izvršavanja, dizajn programa i testiranje [7]. Naime, uslijed velikog broja niti postaje teško pratiti stanje u kojem se aplikacija nalazi. Sam nedeterminizam može dovesti do pogrešaka u kodu (engl. *bugs*) koje nastaju uslijed korištenja većeg broja niti, što znači da ni paralelni sustavi nisu imuni na njih. Testiranje istovremenih sustava je otežano iz razloga što se i pogreška može nedeterministički manifestirati – jednom u X slučajeva, gdje X može biti jako veliki broj. Pogreške koje se mogu pojaviti u istovremenim sustavima su:

- *Deadlock* (zastoj)
- *Starvation* (izgladnjivanje)
- *Race condition*
- *Data race*

Izvor ovih pogrešaka je često nekakvo stanje ili resurs kojeg više niti koristi, dijeli među sobom. Ove pogreške su tipične za programiranje s nitima [15]. Pod "programiranje s nitima" se misli na korištenje konstrukata niske razine za izradu istovremenih aplikacija. Treba naglasiti da ni ostale metode istovremenog programiranja nisu imune na ove pogreške.

5.1.1 Race condition i *data race*

Race condition je situacija u kojoj točnost programa ovisi o vremenu ili redoslijedu izvršavanja niti [11]. *Race condition* se često pomiješa sa sličnim terminom *data race*. *Data race* nastupa onda kada barem jedna nit zapisuje u varijablu, a više njih čita vrijednost varijable pri čemu čitanje i pisanje vrijednosti nisu sortirani po *dogodilo-se-prije* (engl. *happens-before*) odnosu [11]. Svrha *dogodilo-se-prije* odnosa je da osigura da niti vide međusobne zapise u memoriju. Iako možda zvuči zbunjujuće, cilj ovog odnosa nije uspostava vremenskog poretka. Zapravo, kada kažemo da se pisanje u memoriju A dogodilo prije čitanja iz memorije B, garantira se da je efekt pisanja u memoriju A vidljiv tom konkretnom čitanju B [15]. Kao što je navedeno u [15]: "Zadatak programera je da osigura da je zapisivanje u varijablu (memorijsku lokaciju) u *dogodilo-se-prije* odnosu sa svakim čitanjem iz varijable koje bi trebalo pročitati zapisanu vrijednost".

Ova dva termina se podosta preklapaju te u praksi možemo imati *race condition* koji je *posljedica data racea*, a i *data race* može dovesti do *race conditiona* [18]. Isto tako, možemo imati i jedno bez drugog.

Za demonstraciju *race conditiona* koristimo jednostavan primjer (preuzet iz [11]) implementiran u programskom jeziku Scala. Dodatak A sadrži cijeli kod primjera.

U ovom primjeru želimo imati isključivo jednu instancu `ExpensiveEntity` objekta koju pri tome želimo instancirati onda kada nam ona zatreba. Jednu instancu bismo mogli željeti možda zato što je objekt resursno zahtijevan pa da uštedimo na resursima ili se možda radi o objektu koji učita i sadrži postavke aplikacije pa želimo imati istu instancu na razini cijele aplikacije. S tim ciljem u metodi `getInstance` prvo provjeravamo postoji li instanca te ukoliko ne postoji, stvaramo ju i pridružimo polju `instance`. U suprotnom, vraćamo referencu na već stvorenu vrijednost.

Race condition je posljedica toga što metodi `getInstance` istovremeno pristupaju obje niti koje stvaramo. Obje niti će provjeriti je li vrijednost polja `instance` jednaka `null`, a rezultat provjere ovisi o trenutku u kojem te dvije niti istu izvrše. Vrlo često će rezultat provjere za obje niti biti istina te će obje niti nastaviti s instanciranjem "skupog" objekta, onog istog za kojeg smo htjeli samo jednu instancu. To se može vidjeti na Slika 2 na kojoj je prikazan rezultat jednog izvršavanja.

```
[info] Running main.Main_RC
Current thread id: 1056935
Current thread id: 1056936
do x and y reference the same object: false
[success] Total time: 5 s, completed Apr 11, 2017 9:48:16 PM
```

Slika 2. Rezultat izvršavanja - posljedica race conditiona

Usporedi li se kod primjera s ispisom poviše, očito je da su obje niti dobile istu vrijednost provjere te instancirale svaka po jedan "skupi" objekt.

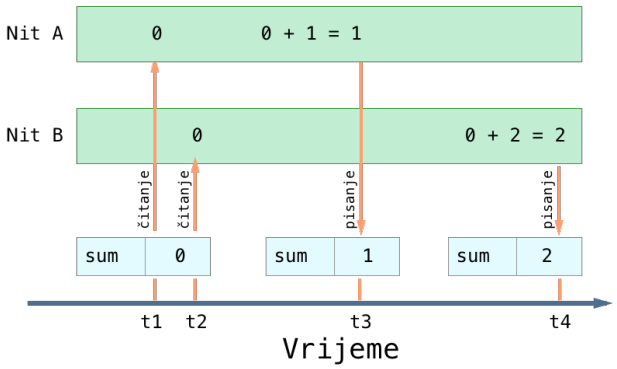
Kako bismo demonstrirali *data race* pretpostavimo da želimo paralelno zbrojiti prvih 100 prirodnih brojeva. Kod za ovaj primjer je prikazan u dodatku B. Sam *data race* je uzrokovan korištenjem varijable

`sum` u paralelnom kontekstu pri čemu više niti pristupa varijabli istovremeno i nastoji novu vrijednost istovremeno zapisati. Ovaj primjer ujedno sadrži i *race condition* jer konačan rezultat ovisi o vremenu ili redoslijedu izvršavanja pojedine niti [8, 15]. Na Slika 3 su prikazani rezultati dobiveni višestrukim pokretanjem primjera.

```
> run
[info] Running main.Main_DR
sum is: 4625
should be: 5050
[success] Total time: 0 s, completed Apr 12, 2017 11:26:04 AM
> run
[info] Running main.Main_DR
sum is: 5050
should be: 5050
[success] Total time: 0 s, completed Apr 12, 2017 11:26:05 AM
> run
[info] Running main.Main_DR
sum is: 4254
should be: 5050
[success] Total time: 0 s, completed Apr 12, 2017 11:26:06 AM
> run
[info] Running main.Main_DR
sum is: 4977
should be: 5050
[success] Total time: 0 s, completed Apr 12, 2017 11:26:14 AM
```

Slika 3. Posljedica data racea na izvršavanje

Pa kako to da inkrementiranje varijable iz više niti uzrokuje *data race*? Razlog je u tome što se inkrementiranje varijable na razini procesora zapravo sastoji od 3 instrukcije: 1) čitanje vrijednosti varijable, 2) inkrementiranje vrijednosti i 3) spremanje nove vrijednosti u registar. Problem nastaje kada jedna nit pročita vrijednost iz registra prije nego je druga stigla novu vrijednost u njega zapisati. Nit u toj situaciji koristi staru vrijednost varijable. Grafički je to prikazano na Slika 4.



Slika 4. Data race - nit B čita staru vrijednost varijable sum. Točan rezultat bi bio 3

Ovaj problem se jednostavno mogao riješiti koristeći *atomarni* (engl. *atomic*) inkrement. Atomarne operacije su one čije se izvršavanje čini trenutnim iz perspektive drugih niti. Atomarnost garantira izoliranost od drugih niti izvršavanja dok se operacija ne izvrši u potpunosti.

S obzirom da smo već rekli da se u praksi *data race* i *race condition* mogu pojaviti skupa ili zasebno, zainteresiranog čitatelja upućujemo na [18]. Autor problemom transfera određenog iznosa s jednog

na drugi bankovni račun nudi primjere rješenja problema koji sadrže 1) *data race* i *race condition*, 2) samo *race condition*, 3) nijedan problem i 4) samo *data race*.

5.1.2 Deadlock

Često se za rješavanje pogrešaka tipa *race condition* i *data race* koristi pristup međusobnog isključivanja [15]. Ovim pristupom se želi osigurati da u bilo kojem trenutku isključivo jedna nit ima pristup sekciji koda osjetljivoj na navedene pogreške, koja se zove *kritična sekcija*. Postavljeno ograničenje se postiže korištenjem sinkronizacijskog mehanizma *lock*. Da bi nit dobila pristup kritičnoj sekciji, mora dobiti vlasništvo nad *lockom*. Ukoliko je *lock* u vlasništvu druge nit, onda prva mora čekati da se *lock* oslobodi prije nego može nastaviti s izvršavanjem. U programskom jeziku Java, resurs se zaključava korištenjem ključne riječi `synchronized`, a u C#-u korištenjem ključne riječi `lock`. Neispravno, nepažljivo zaključavanje resursa može dovesti do *deadlocka*.

Deadlock je situacija u kojoj dvije niti ne mogu nastaviti s izvršavanjem jer je svakoj potreban resurs kojeg je zaključala ona druga pa jedna drugu vječno čekaju da ona druga nit taj resurs oslobodi [11]. Programski kod primjera je dostupan u Dodatak C.

U metodi `doComplexWork` zaključavamo pristup primljenim argumentima i to prvo prvom argumentu koji je primljen, a zatim drugom. Kada kreiramo niti, šaljemo blok koda koji će one izvršiti s time da kod niti t1 kao prvi argument šaljemo instancu a `Component` klase, a kao drugi instancu b, a kod niti t2 obrnuto. Ono što se dogodi kada obje niti pozovu metodu doComplexWork jest da nit t1 zaključa pristup objektu a, a nit t2 zaključa pristup objektu b (istovremeno). S obzirom da t1 treba pristup objektu b da bi nastavila s izvršavanjem, čeka da t2 istoga otključa te time dozvoli pristup. S druge strane, t2 treba dobiti pristup objektu a da bi nit nastavila s izvršavanjem, ali mora čekati da t1 otključa objekt a, tj. dozvoli pristup. Vidimo da je situacija takva da t1 čeka na t2 da otključa svoj objekta i t2 čeka na t1 da otključa svoj objekt. Ove dvije niti će se međusobno vječno čekati i nijedna neće moći nastaviti s izvršavanjem, a to je *deadlock*.

5.1.3 Izgladnjivanje (*starvation*)

Izgladnjivanje nastupa kada nit ne može dobiti pristup resursu koji joj je potreban da bi nastavila sa svojim izvršavanjem, a najčešće je to vrijeme izvršavanja na procesoru [11]. Jedan od uzroka izgladnjivanja može biti nepravilno korištenje lockova pri čemu neka od niti ne prepušta drugim nitima da koriste određeni resurs. Izgladnjivanje može nastupiti i mijenjanjem prioriteta niti.

5.2 A distribuirani sustavi?

Sva tri pojma, istovremeno, paralelno i distribuirano, se donekle preklapaju. U ovom seminarskom radu paralelne i distribuirane sustave smatramo istovremenim sustavima. Za neke je razlika između paralelnog i distribuiranog sustava u cilju koji se želi postići [19]. Pri tome je cilj paralelnih sustava poboljšati performanse, a distribuiranih osigurati otpornost sustava na pogreške.

Paralelne i distribuirane sustave možemo razlikovati i po sljedećem kriteriju:

- U paralelnim sustavima svi procesori imaju pristup zajedničkoj memoriji za razmjenu informacija među procesima [20].
- U distribuiranim sustavima, svaki procesor ima svoju privatnu memoriju, a razmjena informacija se odvija razmjenom poruka [15]. Distribuiranim sustavom možemo smatrati i skupinu procesa koji su pokrenuti na istom računalu, a komuniciraju razmjenom poruka.

U distribuiranim sustavima, svako računalo mora pretpostaviti da može doći do pogreške u bilo kojem drugom računalu u nekom trenutku te pružiti sigurnosno jamstvo, odnosno, na odgovarajući način reagirati na pogrešku.

6 Tipovi istovremenog programiranja

Općenito postoje dva tipa istovremenog programiranja: dijeljenom memorijom (engl. *shared-state*) i razmjenom poruka (engl. *message passing*) [7]. Razlikuju se u načinu na koji je ostvarena komunikacija među nitima.

6.1 Istovremeno programiranje zajedničkom memorijom

Kod ovog načina istovremenog programiranja, komunikacija među nitima (ili procesima) se ostvaruje čitanjem i zapisivanjem u dijeljenu memoriju [18, 19].

Izrada istovremenih sustava temeljenih na ovom komunikacijskom modelu se sastoji u izradi niti koje međusobno komuniciraju koristeći zajedničke varijable, pristup kojima se mora sinkronizirati uz pomoć nekog od konstrukata za sinkronizaciju [22]. Varijable kojima pristup ima više različitih niti se zovu dijeljene varijable [11]. Ako definiramo stanje aplikacije kao skup stanja (vrijednosti) svih njegovih varijabli, onda je skup svih dijeljenih varijabli se naziva promjenjivo dijeljeno stanje (engl. *mutable shared-state*). Da bi istovremeni program ispravno radio, svi dijelovi programskog koda moraju ispravno funkcionirati onda kad više niti istovremeno izvršava isti komad programskog koda.

Poteškoće koje smo naveli u poglavlju 5.1 proizlaze iz loše sinkronizacije pristupa dijeljenom stanju. Nažalost, nekada ispravna sinkronizacija može dovesti do smanjenja paralelizma u istovremenom sustavu. O ovome ćemo malo više reći u kasnijem poglavlju.

Model komuniciranja s dijeljenom memorijom je tipičan za istovremene programe koji se izvršavaju lokalno na jednom računalu.

6.2 Istovremeno programiranje razmjenom poruka

Kod ovog modela istovremenog programiranja, procesi ili niti komuniciraju razmjenom poruka (engl. *message-passing*). Internet i druge širokopojasne računalne mreže su primjeri sustava koji se temelje na razmjeni poruka. Nezavisne jedinice (čvorovi) tih sustava komuniciraju koristeći TCP (*Transmission Control Protocol*) ili UDP (*User Datagram Protocol*) protokol pri čemu su poruke spremljene u paketima koji se dolaskom na odredište interpretiraju [21].

Razmjena poruka može biti asinkrona ili sinkrona [23]. U slučaju sinkrone razmjene, proces ili nit su blokirani dok neka od operacija (slanja ili primanja poruke) ne završi. Na primjer, u slučaju poziva udaljene procedure (engl. *Remote Procedure Call* (*RPC*)), proces koji šalje poruku je blokiran (čeka s izvršavanjem) dok ne primi odgovor [24]. Kod asinkrone razmjene poruka, proces ili nit samo inicira operaciju, ali ne čeka da se ona dovrši.

Zbog popularnosti primjene, spominjemo ovdje i *Message-Passing Interface* (MPI). MPI se odnosi na specifikaciju biblioteke i njenih sučelja (engl. *interface*) koju je donijela skupina akademskih institucija i kompanija iz industrije [25]. Same implementacije specifikacije su npr. OpenMPI [26] i Intel MPI [27]. MPI je postala "primarna metoda komunikacije među čvorovima" u *High-Performance Computing* (HPC) zajednici [21]. Još jedan razlog zbog kojeg ga spominjemo u ovom seminaru je taj što podržava način sinkronog slanja poruke drukčiji od onog kod RPC-a. Naime, MPI prilikom sinkronog slanja može

čekati da se cijela poruka proslijedi u *buffer* primatelja ili u buffer sustava. Slanje poruke u buffer sustava, iz kojega će kasnije primatelj iščitati poruku, je sporije, ali pošiljatelj može nastaviti s izvršavanjem odmah nakon transfera poruka jer ne ovisi o primatelju [28]. MPI podržava i asinkronu komunikaciju.

U ostatku seminara ćemo za demonstraciju ovog modela istovremenog programiranja koristiti Akka biblioteku.

7 Metode istovremenog programiranja

U ovom poglavlju su demonstrirane tri različite metode istovremenog programiranja:

1) Klasično programiranje s nitima;
2) Funkcionalno (istovremeno) programiranje;
3) Programiranje s actorima.

Demonstraciju različitih metoda istovremenog programiranja provodimo koristeći dva primjera koji su implementirani u sve tri promatrane metode. Dodatno, gdje to ima smisla, implementacije uspoređujemo s obzirom na brzinu izvršavanja. Važno je samo naglasiti da, zbog već ranije spomenute nedeterminističke prirode višenitnih aplikacija te samog operacijskog sustava koji dodjeljuje različitim nitima pristup procesoru [11], brzina izvršavanja nije svaki put ista. Stoga ćemo svaki put prezentirati prosječnu brzinu dobivenu većim brojem izvršavanja dijela programskog koda koji nas zanima.

Svi izrađeni primjeri (njih ukupno 6) su pokrenuti na MacBook Pro laptopu iz 2013-te godine. Laptop koristi macOS Sierra operacijski sustav te sadrži:

1) Intel i7 procesor sa radnim taktom od 2.3 GHz i 4 fizičke jezgre (8 virtualnih);
2) 16 GB DDR3 radne memorije sa radnim taktom od 1600 MHz.

Važno je napomenut da zbog nedeterminizma svaki put kada pokrenemo program, moguće je dobiti različitu prosječnu brzinu izvršavanja. Stoga, dobivena prosječna brzina predstavlja procjenu prosječne brzine implementacije.

7.1 Odabrani primjeri

Primjeri koje ćemo promotriti su:

1) Istovremeno/paralelno množenje matrica
2) Obrazac proizvođača-više potrošača (engl. *Producer-multiple consumer pattern*)

Prvi primjer je odabran jer predstavlja klasičan problem koji se pojavljuje u paralelnom programiranju.

Drugi primjer je preuzet iz rada u kojem su autori uspoređivali programiranje višenitnih aplikacija koristeći imperativnu i funkcionalnu programsku paradigmu [29]. Između ostalog, autori su usporedili performanse programa koji su u konačnici sadržavali različite udjele koda napisanih navedenim paradigmama.

Cijeli programski kod za primjere se može pronaći na Github repozitoriju[1] seminara. U dodatcima samog seminara nije prikazan programski kod pomoćnih objekata i metoda.

7.1.1 Problemi i napomene mjerenja

Dodatno, želimo implementacije usporediti po broju stvorenih niti tijekom izvršavanja primjera, ali to u nekim slučajevima može biti problematično za realizirati. Jedan takav slučaj jest kada koristimo klasično programiranje s nitima gdje sami kreiramo svaku nit u programu.

Pa gdje bi mogao nastati problem? Problem bi mogao nastati onda kada želimo dobiti točan broj stvorenih niti tijekom izvršavanja programa. Naime, sve niti koje izvršavaju istovremeni dio programskog koda sebe trebaju registrirati kod `MeasurementHelpers` objekta s ciljem da kasnije možemo dobiti broj različitih niti koji se koristio. Problem je što u nekim slučajevima ne dobijemo točan broj stvorenih niti. Naime, ideja je bila spremiti ID-eve stvorenih niti u listu te prebrojati samo one različite, ali Java Virtual Machine (JVM) nanovo koristi ID-eve niti koje su gotove s izvršavanjem [30]. Uzmimo za primjer množenje dviju matrica 300x300 i implementaciju prezentiranu u poglavlju 7.2.1. Za svaki redak matrice stvorimo po jednu nit, ali ono što bi se potencijalno moglo dogoditi jest da prva nit završi sa izvršavanjem prije nego se, npr. 295-ta stigne stvoriti i pokrenuti. Zbog toga 295-ta nit i prva nit potencijalno mogu imati isti ID ili hash kod. S obzirom da se i hash kod može ponoviti, ni spremanje samih referenci na niti nije riješilo problem.

Iako se ova situacija možda i neće dogoditi tijekom testiranja implementacija u ovom seminaru, smatramo da je važno imati na umu ovaj nedostatak liste stvorenih niti sadržane u `MeasurementHelpers` objektu tijekom čitanja ovog seminarskog rada. Stoga, ukoliko sami stvaramo, npr. 300 niti, i dobijemo manje od 300 jedinstvenih elemenata u navedenom objektu, znamo da dobiveni broj ne predstavlja jedinstveno stvorene niti.

Zašto sve ovo i čemu nam onda ovo služi? Ovo sve radimo samo da bismo demonstrirali kako različiti razvojni okviri mogu nanovo koristiti već postojeće niti umjesto da stvaraju nove. Iako ne možemo sa sigurnošću reći da je broj jedinstvenih elemenata prikupljenih u nizu `MeasurementHelpers` objekta ujedno i točan broj stvorenih niti, ipak ćemo moći prikazati razliku u veličini broja korištenih niti ovisno o metodi istovremenog programiranja koja se koristi.

Izrazito je važno dodatno naglasiti da rezultat brojanja broja korištenih niti može ovisiti i o tome u kojem dijelu programskog koda taj broj brojimo. Ovo je slučaj kod implementacija primjera 2 – obrazac proizvođač-više potrošača koji je implementiran uz pomoć *Futurea* (poglavlje 7.3.7) i actora (poglavlje 7.5.2). Kod primjera sa *futurom*, u slučaju kada je brojanje broja korištenih niti bilo na početku start metode najčešći prosječan broj korištenih niti je bio oko 10, a prilikom jednog brojanja unutar *blocking* naredbe (opisane u poglavlju 7.3.5), dobiven je prosječan broj od 25 niti. Ovaj rezultat nismo uspjeli reproducirati. To upućuje na još jednu neugodnost prilikom mjerenja kod istovremenih sustava, a to je nedeterminizam koji je spominjan u ranijem poglavlju. Stoga sva ova mjerenja treba uzeti u obzir kao procjenu reda veličina.

Kod oba primjera implementirana na klasičan način s nitima, gore navedeni problem ne postoji jer sami kontroliramo koje se niti koriste. Dodatno, mislimo da ovaj problem ne postoji kod primjera 1 jer kod njega samo raspodijelimo posao među nitima, futurima i actorima, koji zatim vrate rezultat. Primjer 2 je različit utoliko što entiteti sami nastoje dobiti ili dobivaju elemente dok proizvođač ne završi sa stvaranjem novih. U slučaju kada je futureu zadan zadatak, onda će se taj zadatak poslati jednoj niti koja će ga izvršiti do kraja. Kod actora, jednu poruku će izvršiti jedna nit, ali različite poruke mogu izvršiti različite niti [31].

Za skupljanje broja različitih niti se koristi niz lista. Svakom entitetu koji se stvori se pridijeli identifikator koji određuje poziciju u nizu kojoj može pristupiti (na neki način dijelimo memoriju da izbjegnemo *data race*). Pod tim identifikatorom se nalazi lista id-eva različitih niti koje su izvršavali navedeni entitet. S obzirom da se radi o listi, brojanje broja različitih niti može imati utjecaj na brzinu izvršavanja programa ukoliko veliki broj id-eva treba dodati u listu.

Unatoč utjecaju na brzinu, u slučaju implementacija primjera 2 (obrazac proizvođač-više potrošača) identifikatori niti se dodaju u kolekciju svaki put kada potrošač dobije element. Ovo je isto za implementacije sa futurima i actorima kod kojih bismo potencijalno mogli dobiti različit broj niti.

7.2 Klasično programiranje s nitima

Klasičnog programiranja sa nitima smo se praktički već dotakli. Svi problemi koje smo prije spomenuli u poglavlju 5.1 (i njegovim potpoglavljima) su tipični problemi koji se javljaju u tom obliku programiranja. Te probleme smo objasnili ranije zato što se oni mogu pojaviti u različitim metodama istovremenog programiranja. U konačnici, ispravnost istovremenog programa je odgovornost programera, a metoda koja se koristi samo alat koji programeru može pomoći da lakše (ili teže) dođe do cilja. Metoda kao takva nije imuna na pogreške koje programer radi, a koje mogu dovesti npr. do *race conditiona*.

Ipak, korištenje niti ima i neke svoje prednosti [32]:

- Niska razina apstrakcija omogućuje širok raspon primjena te veću kontrolu nad samom niti koja izvršava nekakav programski kod;
- Mogu biti vrlo efikasne u ispravnoj implementaciji;
- Jednostavno se mogu implementirati koristeći postojeće imperativne objektno-orijentirane programske jezike.

Iako niska razina apstrakcije ima neke prednosti, isto tako otežava programiranje samim nitima jer je potrebno eksplicitno sinkronizirati pristup zajedničkim resursima korištenjem *lockova*. Programiranje s nitima i lockovima se smatra teškim [32] – tako da ovo računamo kao nedostatak ove metode istovremenog programiranja. Lockove koristimo onda kada imamo zajedničku memoriju koju je potrebno zaštiti od istovremenog pristupa različitih niti. Poteškoćama programiranja s nitima pridonosi nedeterministička priroda niti te potreba za sinkronizacijom pristupa zajedničkim resursima, o čemu smo već pisali u poglavlju 5.1 i potpoglavljima.

7.2.1 Primjer 1 – implementacija

Implementacija prvog odabranog primjera je (dijelom) dana u Dodatak D. Ostatak implementacije je dostupan na mreži.

Istovremeni izvršavanje se provodi po redu, tj. za svaki red matrice se kreira jedna nit koja množi elemente tog reda sa stupcima druge matrice. Sam algoritam se nalazi u metodi matrixMultiply u Dodatak D. Rezultat metode je nova matrica koju čije vrijednosti će biti spremljene u niz lista. Niz instanciramo na onoliko elemenata koliko će biti redataka. Kada izvrši množenje dodijeljenog retka sa svim stupcima, parcijalni rezultat će biti novi red čija će se referenca spremiti u odgovarajući element niza. Na ovaj način izbjegavamo *race condition*, jer da smo koristili listu koja dinamički dodaje elemente, riskiramo da, npr. nit koja je zadužena za četvrti red prva završi te se prva doda u listu, zbog čega bi se pomiješali retci nove matrice.

Kako bismo znali koliko je množenje matrica zapravo trajalo, glavna nit mora znati kada su sve niti koje je stvorila završile s množenjem svoga retka. Stoga svaku stvorenu nit spremamo u listu, a potom, nakon što je za svaki redak nit stvorena i pokrenuta, svakoj niti iz te liste pozivamo metodu `join`. Pozivom metode `join`, nit koja je pozvala metodu je blokirana, tj. čeka da nit nad kojom je pozvana metoda izvrši do kraja.

Matrice su veličina 300x300. Program pokrećemo 1 put, ali cijeli proces kreiranja matrica, instanciranja niti i istovremenog množenja matrica se ponavlja 100 puta. Ovdje iznosimo samo prosječne vrijednosti izvršavanja. Tako je prosječna brzina izvršavanja 4039.63 milisekundi, odnosno 4.03963 sekunde. U kasnije poglavlju (7.3.6) ćemo vidjeti da je ovako sporo izvršavanje programa vrlo vjerojatno uzrokovano trostrukom petljom u implementaciji.

S obzirom na to da sami stvaramo jednu nit po retku matrice, znamo da će biti stvoreno i pokrenuto 300 niti.

7.2.1.1 Cijena stvaranja niti

Stvaranje niti nije jeftino, svaka stvorena nit se mora registrirati s operacijskim sustavom te se za nju mora alocirati stog (memorija). Pogledati ćemo implementaciju množenja matrica koja koristi jednu nit. Sam algoritam stvaranja i množenja matrica je identičan, osim što se za množenje svakog pojedinog retka koristi jedina nit koju imamo na raspolaganju. S obzirom da je programski kod skoro identičan, isti nije uključena među dodatke ovog seminara, ali se može pronaći na github repozitoriju.

Pokretanjem implementacije sa jednom niti na 10 ponavljanja, dobivena prosječna brzina je 41.2658 sekundi pri čemu su se množile dvije matrice 300x300. Iako smo odabrali mnogo manji broj ponavljanja, očito je da je množenje matrica samo s jednom niti u prosjeku višestruko sporije nego ekvivalentna višenitna implementacija.

Za sljedeću usporedbu, postavljeno je da se stvore matrice 100x100 i da se proces stvaranja i množenja matrica ponovi 100 puta. Prosječna brzina višenitne implementacije je ispala 105.98 milisekundi, a implementacije s jednom niti 303.81 milisekundu.

Ukoliko smanjimo veličine matrica na 20x20, dakle mnogo manje matrice u odnosu na prethodna pokretanja, te ponovno pokrenemo proces sa 100 ponavljanja, uočavamo da je implementacija koja koristi jednu nit postala brža. Naime, prosječno vrijeme izvršavanja je 1,54 milisekundu za višenitnu implementaciju te 0,65 milisekundi za implementaciju s jednom niti.

Što se dogodilo? S obzirom da radimo sa mnogo manjim matricama, mnogo manje operacija je potrebno izvršiti da bi se one pomnožile pri čemu cijena stvaranja niti postaje vidljiva. Cijena stvaranja niti uključuje vrijeme potrebno da se nit stvori i ugasi, pri čemu operacijski sustav mora alocirati memoriju za novu nit [11]. Stoga je važno imati na umu da ponekad stvaranje i korištenje niti može učiniti aplikaciju sporijom nego što bi ona bila da smo jednostavno koristili jednu ili manji broj niti.

7.2.2 Primjer 2 – implementacija

Implementacija ovog primjera je dana u Dodatak E. Programski kod je u ovom primjeru, zbog jednostavnosti, podijeljen u klase `producer` i `consumer`. Producer klasa predstavlja proizvođača, a consumer potrošača. Zadatak proizvođača je da stvori određen broj elemenata te ih spremi u zajednički *queue* (red) iz kojega će onda potrošači te elemente izvlačiti. Potrošači mogu iz reda uzeti proizvoljan broj elemenata – natječu se za pristup za redu. Svaki potrošač i proizvođač su pokrenuti na

vlastitoj niti. Broj elemenata koji će proizvođač stvoriti je definiran u objektu `Configuration` koji sadrži konfiguracije za veći broj primjera.

U ovom primjeru vidimo potrebu za sinkronizacijom pristupa dijeljenoj strukturi podataka (u ovom slučaju red). Sinkronizacija se vrši na dva mjesta:

1. Proizvođač koji stvara resurs mora sinkronizirati pristup prije dodavanja novog elementa u red;
2. Potrošač mora stalno provjeravati postoji li element u redu te čekati da se doda ukoliko je red prazan i proizvođač nije proizveo posljednji element.

U Tablica 1 su dane prosječne brzine izvršavanja programa izražene u milisekundama, a dobivene su temeljem pokretanja zadane konfiguracije 100 ili 1000 puta. Po zadanoj konfiguraciji, proizvođač je morao stvoriti 10 000 elemenata, za koje su se potrošači (niti) natjecali. Tek kada su svi elementi stvoreni i preuzeti iz reda, program je gotov.

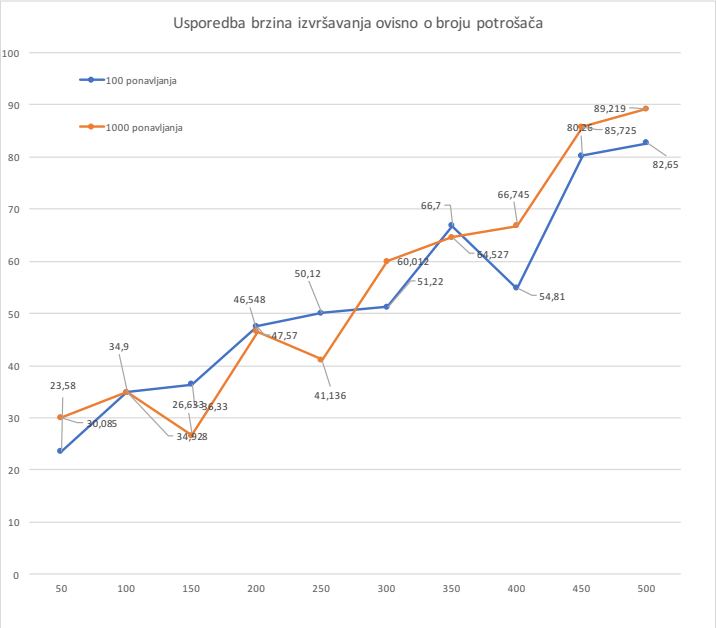
	Prosječna brzina izvršavanja u milisekundama (ovisno o broju ponavljanja)	
Broj potrošača	100 ponavljanja	1000 ponavljanja
50	23,58	30,085
100	34,9	34,928
150	36,33	26,633
200	47,57	46,548
250	50,12	41,136
300	51,22	60,012
350	66,7	64,527
400	54,81	66,745
450	80,26	85,725
500	82,65	89,219

Tablica 1 Brzina izvršavanja obrasca proizvođač-više potrošača koristeći niti

Iz tablice možemo vidjeti sljedeće:

1. Slične prosječne brzine izvršavanja za 100 i 1000 ponavljanja;
2. Povećanjem broja potrošača (niti), usporava se izvršavanje programa.

Rezultati su prikazani i grafički Slika 5.



Slika 5 Promjena brzine izvršavanja ovisno o broju potrošača – implementacija s nitima (vrijednosti u milisekundama). X-os je broj niti, a Y-os vrijeme u milisekundama.

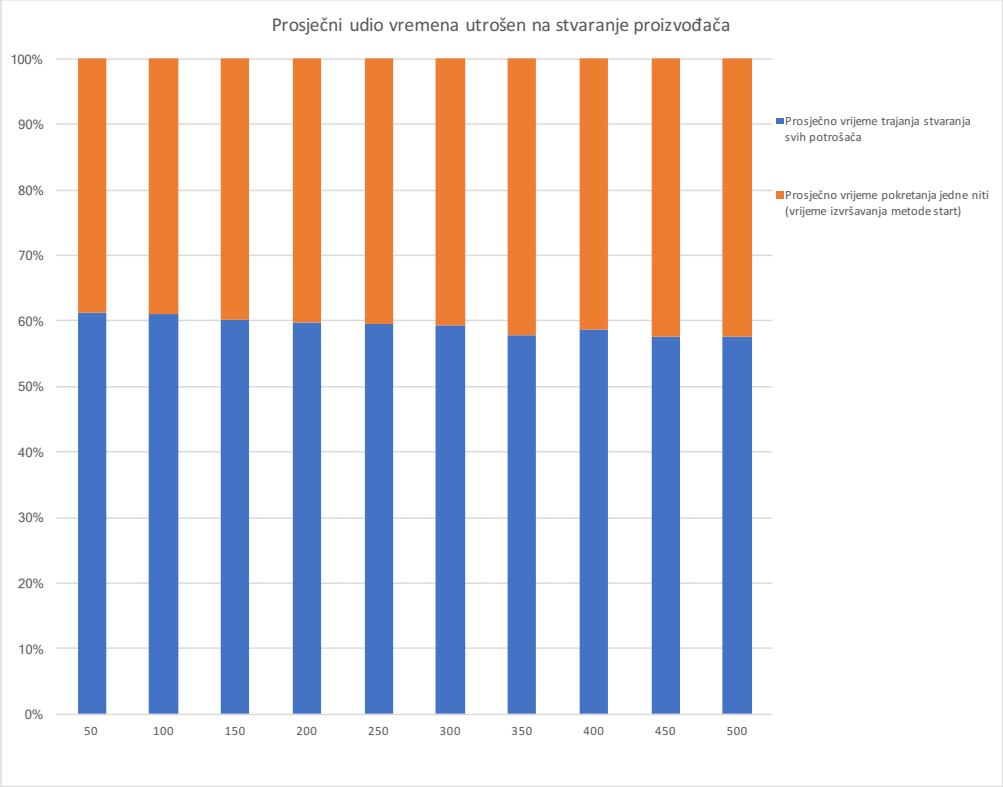
Jedan od razloga što se implementacija usporila je taj što glavna nit mora stvoriti veći broj potrošača. Kako bi se vidio odnos vremena potrebnog da se stvori veći broj potrošača i izvrši program, odlučili smo malo izmijeniti programski kod. Pri tome smo mjerili prosječnu brzinu stvaranja svih potrošača po pokretanju implementacije i prosječno trajanje poziva metode `start` ukupno za sva izvršavanja. Poziv metode start se odvija unutar metode za stvaranje potrošača na način da se ista pozove odmah nakon što je instanca potrošača stvorena. Novi rezultati su prikazani u Tablica 2 Mjerenje izvršavanja, stvaranja potrošača i pokretanja niti. Grafičkim prikazom na Slika 6 vidi se da se veći dio vremena stvaranja potrošača zapravo potroši na pokretanje niti.

S obzirom na to da sve niti pristupaju istom redu, potrebno je taj pristup sinkronizirati. Pri tome nit koja nastoji uzeti element iz reda mora, u slučaju da je red prazan, čekati neku drugu nit da doda element u red. Nit koja dodaje elemente mora neku od niti obavijestiti da red više nije prazan. Obavijest se vrši pozivom metode `notify`, čime će jedna nit dobiti obavijest da može pristupiti dijeljenom resursu. Po završetku odrađenog posla, `producer` mora obavijestiti sve niti da se novi elementi više neće stvarati kako iste ne bi zapele u beskonačnoj petlji.

Broj potrošača	Trajanje izvršavanja (ms)	Trajanje stvaranja potrošača (ms)	Trajanje poziva metode start (ms)
50	23,58	2,93	1,86
100	34,9	5,86	3,74
150	36,33	8,44	5,6
200	47,57	10,47	7,07

250	50,12	13,03	8,83
300	51,22	16,03	11,02
350	66,7	19	13,86
400	54,81	22,26	15,66
450	80,26	26,45	19,5
500	82,65	29,63	21,78

Tablica 2 Mjerenje izvršavanja, stvaranja potrošača i pokretanja niti (100 ponavljanja)



Slika 6 Odnos vremena trajanja stvaranja potrošača i poziva metode start

Sinkronizacija pristupa stvara promet na sabirnici kojom se pristupa zajedničkoj memoriji [11]. Ova sabirnica je zajednička za sve procesore, a ima ograničenu protočnost prometa. Stoga, ukoliko se niti moraju natjecati za pristup toj sabirnici, sve niti koje koriste sinkronizaciju će imati slabije performanse. Upravo zbog toga u implementaciji se koristi `notify` umjesto `notifyAll`, koji obavijesti sve niti o prepuštanju resursa. Korištenjem `notifyAll`, cjelokupni kod bi bio značajno sporiji. Npr. korištenje `notifyAll` svaki put kada proizvođač stvori element, prosječno vrijeme izvršavanja za 500 potrošača sa 100 ponavljanja iznosi 113,7 milisekundi (u odnosu na 92,610 iz tablice).

Sinkronizacija pristupa uzrokuje *context switching* (izmjenu konteksta), koja ima negativan utjecaj na performanse [11]. Operacijski sustav svako određeni period mora prekinuti izvršavanje jedne niti i dati priliku drugoj da nastavi sa vlastitim izvršavanjem. Ovakvo obustavljanje izvršavanja uključuje spremanje stanja izvršavanja, kasnije vraćanje tog stanja kako bi nit nastavila s izvršavanjem te trošenje procesorskog vremena na planiranje (engl. *scheduling*) izvršavanja, umjesto na samo izvršavanje. Ovaj

postupak ima značajan utjecaj na performanse aplikacije. Ukoliko se nit mora obustaviti zato što čeka na nekakav resurs ili zato što nije uspjela dobiti lock, dvije dodatne izmjene konteksta se dogode: operacijski sustav mora maknuti s procesora nit prije nego ona iskoristi svo procesorsko vrijeme koje joj je dodijeljeno i mora tu istu niti vratiti na procesor, dodijeliti joj još vremena, onda kada resurs ili lock postane dostupan.

U primjeru 1 smo u principu implementirali paralelno množenje matrica i izbjegli potrebu za sinkronizacijom pristupa. U ovom primjeru to nije bilo moguće jer niti imaju zajedničku kolekciju. To dovodi do toga da se iste natječu za pristup resursu. Važno je naglasiti da nekontrolirano stvaranje novih niti može ozbiljno ugroziti performanse programa. Stoga, metode koje ćemo promotriti u nastavku seminara nastoje nanovo koristiti postojeće niti programa.

7.3 Funkcionalno (istovremeno) programiranje

Funkcionalno programiranje je programska paradigma, odnosno stil programiranja, u kojem se izračuni temelje na primjeni čistih funkcija (engl. *pure function*) na argumente te izbjegava mijenjanje stanja programa [31, 32, 33]. Funkcionalni programski jezik je onaj koji podržava i potiče programiranje u funkcionalnom stilu.

7.3.1 Čiste funkcije, nuspojave i referentna transparentnost

Za funkciju se kaže da je čista ukoliko ne radi ništa drugo osim što za nekakav argument A vrati rezultat B [33]. Ukoliko funkcija nema ovo svojstvo, onda kažemo da ima nuspojavu (engl. *side effect*). Nuspojavom se smatra:

- Izmjena vrijednosti varijable
- *In place* modifikacija strukture podataka
- Postavljanje polja objekta
- Bacanje iznimke ili zaustavljanje uslijed pogreške
- Ispis na konzolu ili čitanje unosa korisnika
- Čitanje iz ili zapisivanje u datoteku
- Crtanje po ekranu

Možemo primijetiti da se dio stavki odnosi na sučelje i interakciju korisnika s programom. Očekivano je da se čitatelj onda zapita na koji način bi onda korisnik trebao unijeti nekakvu informaciju u program ili na koji način bi program trebao ispisati rezultat korisniku ili što napraviti ako želimo imati spremljene logove događaja u programu. Naravno da velika većina programa mora biti u nekakvoj interakciji s okolnim svijetom, inače je beskorisna. U tom slučaju programeri koji programiraju funkcionalnim stilom koriste apstrakciju koja se temelji na matematičkom pojmu *monad* [35]. Ona omogućuje programeru da bilo kakvu interakciju s vanjskim svijetom prikaže kao očekivani rezultat funkcije i time zadrži "čistoću" funkcija. Monadi su priča za sebe i van fokusa ovog seminarskog rada da bismo ih detaljnije proučavali. Smatraju se jednom od težih koncepata funkcionalnog programiranja.

Uz čiste funkcije se veže i pojam referentne transparentnosti (engl. *referential transparency*), koji se ne odnosi samo na funkcije već je svojstvo bilo kakvog izraza koji se evaluira [33]. Ovo svojstvo nam omogućuje da bilo kakav izraz u programu sa danim argumentima zamijenimo rezultatom koji bismo dobili evaluacijom tog izraza, a da pri tom ne promijenimo značenje programa. Npr. ako imamo u programu izraz 2 + 3, taj izraz uvijek možemo zamijeniti rezultatom 5 i konačni rezultat izvršavanja

programa se neće promijeniti. Uočite da nuspojave narušavaju ovo svojstvo jer program osim što vraća nekakav rezultat, obavi još nekakav posao "sa strane".

Referentna transparentnost je svojstvo koje je programerima poželjno jer im omogućuje jednostavan i prirodan način razmišljanja o izvršavanju programa koji se zove *model zamjene* (engl. *substitution model*) [33]. Kada su izračuni referentno transparentni, možemo zamisliti da se izvršavanje odvija na isti način na koji bismo rješavali matematičku jednadžbu. Prvo svaki dio izraza proširimo te umjesto naziva varijabli ubacimo pridijeljene vrijednosti. Dobiveni izraz svedemo na najjednostavniji mogući oblik. U svakom koraku dobiveni termin zamijenimo s ekvivalentnim.

7.3.2 Nepromjenjivost

U funkcionalnom programiranju se često koriste nepromjenjivi (engl. *immutable*) podatci [36]. Umjesto mijenjanja postojećih vrijednosti, stvara se modificirana kopija, a original ostaje nepromijenjen.

```
scala> val a = List(1, 2, 3, 4, 5)
a: List[Int] = List(1, 2, 3, 4, 5)

scala> val b = 6 :: a
b: List[Int] = List(6, 1, 2, 3, 4, 5)

scala> b.tail
res11: List[Int] = List(1, 2, 3, 4, 5)

scala> b.tail.eq(a)
res12: Boolean = true

scala> val c = List(1, 2, 3, 4, 5)
c: List[Int] = List(1, 2, 3, 4, 5)

scala> c.eq(a)
res13: Boolean = false
```

Slika 7. Dodavanje elementa u nepromjenjivu listu

U slučaju struktura podataka, npr. liste, ako programer dodaje novi element, ta operacija će kao rezultat vratiti novu instancu korištene strukture, pri čemu će se koristiti svi nepromijenjeni postojeći podatci stare. Dakle, postojeći elementi se neće kopirati u novu instancu, već će ih ista referencirati. Ovo se zove dijeljenje podataka (engl. *data sharing*) [33]. Ovo je prikazano slikom 6. Naime, na slici je vidljivo da se stvara početna lista sa 5 elemenata koja se referencira varijablom *a*. Zatim se dodavanjem elementa u postojeću listu zapravo stvara nova koja se referencira varijablom *b*. Želimo pokazati da *b* za dio svojih podataka koristi istu instancu u memoriji kao varijabla *a* pa ćemo usporediti odgovarajuće reference. Naime, uz pomoć svojstva `tail` možemo dobiti sve elemente liste bez prvog elementa, a uz pomoć metode `eq` možemo usporediti instance po referenci. S obzirom da je rezultat usporedbe *istina*, to pokazuje da je sa `b.tail` uistinu referencirana ista instanca kao i sa varijablom *a*, odnosno da nije došlo do kopiranja podataka stvaranjem nove liste. Isto tako, iz slike se vidi da je u slučaju varijable c stvorena potpuno nova lista.

Kod izrade vlastitih nepromjenjivih objekata, programer mora biti pažljiv da koristi strukture koje su same po sebi nepromjenjive. Korištenje promjenjivih struktura unutar takvog objekta će dovesti do suptilne promjenjivosti. Na primjer, neka takav jedan objekt ima svojstvo u kojem se nalazi promjenjiva lista elemenata. Ukoliko je svojstvo javno dostupno, netko drugi će u tu listu moći dodavati i izbacivati elemente, zbog čega objekt u konačnici neće biti nepromjenjiv. U tom slučaju, programer bi trebao koristiti tip podatka *nepromjenjivu listu*.

7.3.3 Funkcije višeg rada

Kada se govori o funkcionalnom programiranju, neizostavno je spomenuti funkcije višeg reda (engl. *higher-order functions*) [35]. S obzirom da su funkcije osnova funkcionalnih jezika, sa njima možemo sve što i sa varijablama u imperativnim jezicima. Dakle, funkcije možemo spremiti u varijable te ih pozvati iz varijabli po potrebi, poslati u neku drugu funkciju pa i vratiti kao rezultat neke funkcije. Funkcije koje kao parametre primaju druge funkcije ili vraćaju funkcije kao rezultat se zovu *funkcije višeg reda*.

7.3.4 Funkcionalno istovremeno programiranje

Prednost korištenja funkcionalnog programiranja u izradi istovremenih sustava je u korištenju principa referentne transparentnosti i nepromjenjivosti, koje smo objasnili u poglavljima 7.3.1 i 7.3.2.

Funkcije koje zadovoljavaju svojstvo referentne transparentnosti je jednostavno za paralelizirati [32]. Naime, budući da funkcija samo mapira primljene argumente u nekakav rezultat, onda nije važno kojim redoslijedom se one izvršavaju. S obzirom da je istovremenost po prirodi nedeterministička, onda ovo svojstvo pojednostavljuje istovremeno programiranje. Naravno, u slučaju da je rezultat jedne funkcije argument u druge, onda se one moraju sekvencijalno izvršiti.

S obzirom da se kod nepromjenjivih objekata stanje ne može mijenjati, interferencija niti ne može dovesti objekt u oštećeno ili nekonzistentno stanje [37]. Ovo svojstvo se smatra posebice korisnim kod programiranja istovremenih sustava. Ipak, programer mora paziti kada radi s varijablama, jer iako je nekakav objekt nepromjenjiv, varijabla koja ga referencira ne mora biti.

Dodatna prednost korištenja funkcionalnog programiranja je u čitljivosti programskog koda [32]. Naime, funkcionalno programiranje spada u skupinu deklarativnih stilova programiranja koji stavljaju fokus na "što" je potrebno napraviti, a ne na "kako" to napraviti.

S druge strane, i) velik broj programera ne zna funkcionalno programiranje; ii) s obzirom na to da se izvršavanje zadataka delegira biblioteci ili razvojnom okruženju, koje onda zakazuje izvršavanje zadatka, programer ima manje kontrole; iii) smatra se dobrom abastrakcijom za paralelizam, ali ne i za istovremenost [32].

7.3.5 Dobivanje vrijednosti iz “budućnosti“

U dosadašnjim poglavljima smo se najviše fokusirali na klasično programiranje sa nitima te smo vidjeli negativan utjecaj sinkronizacije na performanse, cijenu stvaranja niti te poteškoće koje se često javljaju u obliku *race conditiona*, *deadlocka* i *data racea*. Ove pogreške, kao i pad u performansama su često uzrokovani blokiranjem niti. Štoviše, vidjeli smo da i stvaranje velikog broja niti može dovesti do usporavanja cjelokupnog izvršavanja aplikacije. Uz to, klasično programiranje s nitima i razumijevanje takvih programa se smatra zahtjevnim [32].

Umjesto da se u programu niti blokiraju, izvršavanje nekakvog zadatka se može zakazati za onda kada resursi postanu dostupni [15]. Ovo je odlika stila programiranja koje se zove *asinkrono programiranje*. Kod asinkronog programiranje jedna nit zakaže izvršavanje nekakvog zadatka koje će se odvijati neovisno od njenog vlastitog izvršavanja. Izvršavanje zakazanog zadatka će se često odvijati na nekoj drugoj niti, ali kod korištenja određenih alata, isti mogu procijeniti da je optimalnije zadatak izvršiti na istoj niti.

Jedan od načina asinkronog programiranja u Scali, je korištenje *futurea*. Future je poseban tip podatka koji će u nekom trenutku u budućnosti sadržavati vrijednost koja je rezultat izvršavanja zakazanog zadatka [15]. S obzirom da će rezultat biti dostupan u nekom trenutku u budućnosti, pitanje koje se postavlja jest kako doći do rezultata? Jedan od načina je da se čeka, odnosno blokira nit dok rezultat ne postane dostupan, ali ovo se nastoji izbjeći. Preporučeni način je da se registrira funkcija koja će se izvršiti onda kada rezultat postane dostupan. Dodatno, rezultat futurea je moguće mapirati u novi future. Uz to, Future je jedan od tipova podataka koji spadaju u skupinu *monada*.

U slučaju sekvencijalnog programiranja, pozivom metode izvršavanje programa se prebaci u tijelo metode, a pozivatelj "čeka" dok se ne vrati rezultat. U slučaju da metoda vraća rezultat tipa *Future[T]*, onda on predstavlja izračun koji će se izvršiti asinkrono [8]. Često će se taj izračun izvršiti na niti različitoj od one koja je inicirala stvaranje futurea. Oznaka T u *Future[T]* označuje tip podatka kojeg asinkroni izračun vraća.

Da bi se u programskom jeziku Scala future uopće mogao koristiti, potrebno mu je pridijeliti *execution context* koji pruža strategiju za asinkrono izvršavanje funkcija [8]. U Scali se na jednostavan način u program može uvesti implicitni *execution context*. Ovaj *execution context* interno koristi *thread pool* za asinkrono izvršavanje zadataka.

Thread pool se može shvatiti kao skup niti koje se nalaze u stanju pripravnosti te se pokrenu onda kada je nekakav zadatak potrebno istovremeno izvršiti [15]. Niti koje se nalaze u *thread poolu* se mogu koristiti više puta za različite zadatke. Zadani *thread pool* kojeg implicitni *execution context* koristi se zove `ForkJoinPool`, koji inicijalno stvori onoliko niti koliko procesor ima jezgri [38]. Broj inicijalno stvorenih niti se može postaviti izmjenom konfiguracije *thread poola*. U prisutnosti operacija koje blokiraju izvršavanje, ovaj thread pool se može obavijestiti o blokiranju (naredba *blocking*) kako bi stvorio dodatne niti. Ipak, važno je napomenuti da `ForkJoinPool` nije namijenjen za dugotrajna blokiranja koja mogu dovesti do stvaranja ogromnog broja niti.

7.3.6 Primjer 1 – implementacija

Programski kod za ovaj primjer se nalazi u Dodatak F. Iz njega je vidljivo da su korišteni funkcionalni principi – programsko kod je izrađen od većeg broja manjih "čistih" funkcija. U kontrastu sa programskim kodom iz Dodatak D, ovaj je mnogo kompaktniji i čitljiviji (programer mora samo poznavati sintaksu i korištene funkcije).

Kao i u poglavlju 7.2.1, implementacija prvog primjera klasičnim korištenjem niti, i u ovom slučaju implementaciju pokrećemo sa 100 ponavljanja pri čemu se množe dvije matrice 300x300. U ovom slučaju, umjesto da stvaramo po jednu novu nit za svaki redak matrice, za svaki redak zakažemo asinkrono izvršavanje koristeći future. Jedino što nije baš u skladu s principom korištenja futurea je blokiranje uz pomoć `Await.result` koje se koristi kako bismo bili sigurni da su sva izvršavanja gotova prije nego dobijemo konačno vrijeme trajanja.

Sa 100 ponavljanja, programski kod koji koristi funkcionalne principe programiranje i Future se izvrši u prosjeku za 281.98 milisekundi. Implementacija iz poglavlja 7.2.1 je koristila 300 niti. U kontrastu s tim, ova implementacija sa Futurima je koristila svega 8 različitih niti u prosjeku pri čemu se ne računa glavna nit.

Osim razlike u korištenju metode istovremenog programiranja, postoji razlika u implementaciji, pri čemu je klasična implementacija programirana imperativnim stilom, a ova sa future-ima funkcionalnim. Stoga smo napravili još jednu verziju klasične implementacije sa nitima koja će i dalje stvoriti po jednu nit za svaki redak, ali je većinski korišten stil funkcionalnog programiranja (Dodatak G).

Implementacija i dalje koristi 300 različitih niti,, ali je ovaj put brzina izvršavanja bila 275.62 milisekunde. S obzirom na dobiveni rezultat, odlučili smo povećati broj redaka na 800 i smanjili broj ponavljanja na 20. Funkcionalna implementacija sa nitima je u prosjeku trajala 5705.95 milisekundi, a ona sa future-ima 5607.05 milisekundi. Ovo ukazuje na to da su u slučaju ovog primjera na matrici 800x800 implementacije otprilike jednako brze. Pretpostavljamo da je postignuto ubrzanje kod funkcionalne implementacije rezultat uklanjanja trostruko ugniježđene petlje prisutne u imperativnoj implementaciji korištenoj u poglavlju 7.2.1. Naime, u funkcionalnoj implementaciji ona nije potrebna jer se matrica kojom množimo prvo transponira. S obzirom na korištene metode istovremenog programiranja, niti i future, zaključujemo da ne postoji velika razlika u brzinama izvršavanja za primjer množenja matrica. S druge strane, postoji izrazita razlika u broju korištenih niti.

U slučaju implementacije sa future-ima, zanimljivo je primijetiti način na koji se stvara nova matrica. Naime, za svaki redak se registrira asinkrona operacija koristeći future, a rezultat svake te asinkrone operacije će biti po jedan redak nove matrice. Zatim listu future-a pretvaramo u jedan future kojemu je rezultat nova matrica. Ni u jednom trenutku tijekom izrade nove matrice ne moramo brinuti o pozicijama novih redaka – hoće li npr. treći redak biti gotov prije drugog pa se sukladno tome naći u drugom (pogrešnom) retku u novoj matrici. Zahvaljujući svojstvima funkcionalnog programiranja i future-a, redoslijed kojim će biti poredani rezultati asinkronih operacija, odnosno retci nove matrice, je isti onaj kojim smo zabilježili asinkrone operacije kada smo mapirali svaki redak u future.

7.3.7 Primjer 2 – implementacija

U ovom primjeru ponovno promatramo obrazac proizvođača-više potrošača. S obzirom na to proizvođač u zajedničku memoriju sprema elemente koje proizvede, a potrošač ih vadi i koristi, nije bilo načina da izbjegnemo korištenje sinkronizacijskih naredbi – zaključavanja pristupa resursu i slanja obavijesti kada isti postane dostupan. I dalje se umjesto klasičnih niti koristi future. Dodatno, potrebno je obavijestiti thread pool o blokirajućim operacijama koristeći naredbu *blocking* kako bi se izbjegao *deadlock* uzrokovan nedostatkom aktivnih niti. U ovoj implementaciji, naredba se koristi samo u situaciji kada potrošač mora čekati da nekakav element postane dostupan.

S druge strane, da bi se signaliziralo drugim potrošačima da je proizvođač stvorio sve zadane elemente, koristi se *promise*. Promise je tip podatka koji služi da se future kompletira. Svaki promise je povezan sa točno jednim future i omogućuje programeru da kompletira izvršavanje futurea sa uspjehom ili neuspjehom.

Zašto nije bilo moguće izbjeći korištenje sinkronizacijskih naredbi? Proizvođač i svi potrošači koriste isti red za dodavanje, odnosno uzimanje elemenata. Stoga je potrebno da svi oni imaju referencu na zajednički red, što znači da ta referenca mora biti poslana u nekom trenutku u njihove instance. Ta referenca može biti zadržana u svojstvu objekta koji će se samo jednom instancirati. Da je korišten nepromjenjivi red, onda bi trebalo koristiti promjenjivo svojstvo kako bi se svaki put kada se stvori novi red, isti mogao referencirati. Problem koji nastaje jest: proizvođač mora referencirati element sa novim elementom, a potrošač bez nekog elementa kojeg je upravo uzeo – tko će prije?! Dolazi se do *race*

conditiona! Očito je onda potrebno koristiti sinkronizacijske naredbe. S druge strane, da se koristio promjenjivi red, onda se mogla koristiti nepromjenjiva referenca na taj red – budući da se red može mijenjati, nije potrebno referencirati novu instancu. Ali, zbog interferencije niti, postoji mogućnost da se bez sinkronizacije taj red pronađe u nekonzistentnom stanju - u C#-u ovakva situacija dovodi do iznimki u programu. Stoga je opet potrebno koristiti sinkronizacijske naredbe (kao u ekvivalentnom primjeru s nitima). Zbog jednostavnosti, programski kod je promijenjen samo toliko da se mogu koristiti futuri. Odnosno, stvaranjem potrošača i proizvođača ne stvara se nova nit, ali se pozivom metode *start* nad instancom zakaže asinkrona operacija. Programski kod za ovaj primjer je dostupan u Dodatak H.

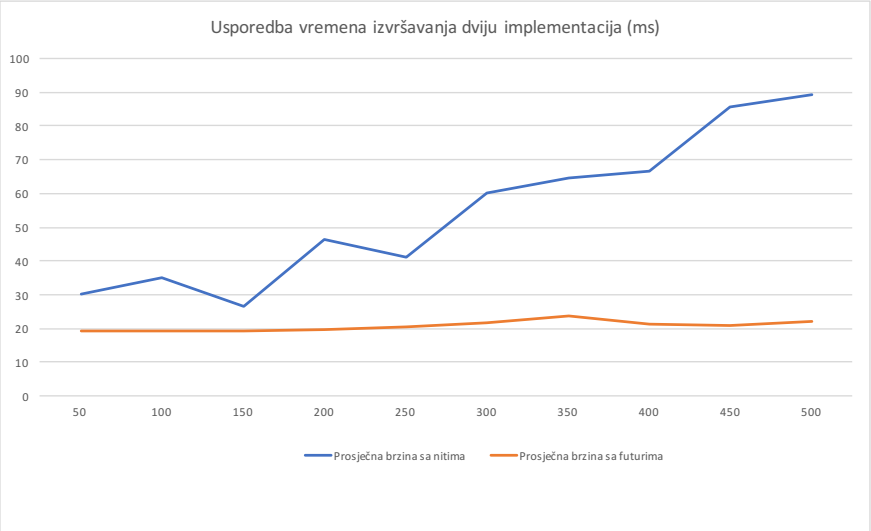
Kao i prije, u poglavlju 7.2.2, izvršavanje se mjeri na način da proizvođač mora u svim slučajevima stvoriti 10 000 elemenata, a broj potrošača se mijenja. Mjerenje je ovaj put provedeno samo sa 1000 ponavljanja, ali je u tablicu uključen i prosječan broja stvorenih niti. U ovoj implementaciji stvaranjem proizvođača ili potrošača ne stvaramo mi novu nit, već se to prepušta thread poolu. Rezultati izvršavanja se nalaze u Tablica 3.

Važno je naglasiti da se broj korištenih niti broji unutar naredbe *blocking*. Broj se također odnosi samo na niti koje su izvršavale entitete tipa potrošač.

	Prosječna brzina izvršavanja u milisekundama (ovisno o broju potrošača)	
Broj potrošača	1000 ponavljanja	Prosječni broj korištenih niti
50	19,198	10,779
100	19,452	11,029
150	19,281	10,981
200	19,600	10,868
250	20,322	10,964
300	21,722	10,864
350	23,828	11,096
400	21,271	10,906
450	21,036	10,906
500	22,047	10,897

Tablica 3 Brzina izvršavanja obrasca proizvođač-više potrošača koristeći future

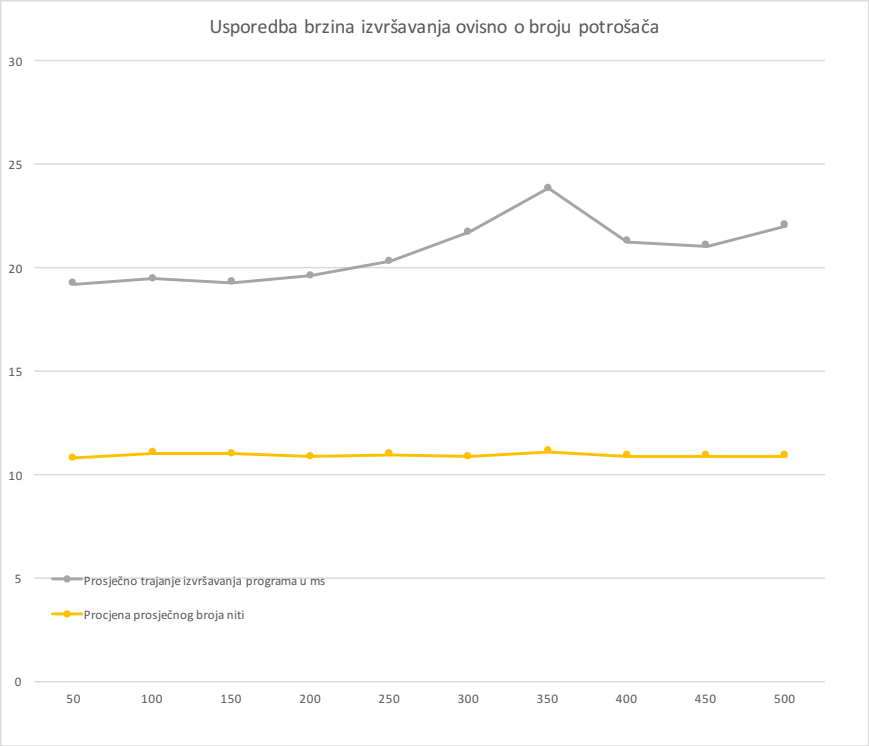
Vidljivo je da postoji ubrzanje u odnosu na klasičnu implementaciju s nitima te je prosječan broj korištenih niti znatno manji. Slika 8 prikazuje odnos prosječnih brzina sa 1000 pokretanja dviju implementacija primjera 2.



Slika 8 Usporedba dviju implementacija (nit i future)

Iz slike poviše zanimljivo je vidjeti da se povećanjem broja potrošača praktički ne mijenja trajanje izvršavanja programa. Ovo upućuje na to da to što glavna nit mora stvoriti veći broj instanci potrošača nema značajan utjecaj na brzinu izvršavanja programa. Prisjetimo se da smo u poglavlju 7.2.2 zaključili da u prosjeku 60% vremena potrebnog da se izvrši metoda `startConsumers` koja stvara i pokreće potrošače otpada na samo pokretanje niti.

Slika 9 prikazuje promjenu u trajanju izvršavanja i prosječnom broju niti u odnosu na povećanje broja potrošača. Iz slike se može zaključiti da je prosječan broj korištenih niti jako sličan za sva pokretanja.



Slika 9 Brzina izvršavanja i prosječan broj niti s obzirom na broj potrošača

7.4 Actor model

Posljednji model istovremenog programiranja koji razmatramo je *actor model*. Riječ je o modelu koji formalizira istovremene izračune otvorenih distribuiranih sustava [39]. Programiranje s actorima je ujedno jedina metoda istovremenog programiranja koju razmatramo, a koja se temelji na modelu razmjene poruka.

Sustavi izrađeni u ovom modelu se sastoje od računalnih agenata koji se zovu *actori* [40]. Actori međusobno komuniciraju slanjem poruka. Ova komunikacija je asinkrona i čini osnovu actor modela. Nakon što je poslao poruku drugom actoru, actor ne mora čekati na odgovor, kao što bi to bio slučaj u sinkronom sustavu, već može nastaviti s obavljanjem nekakvog drugog posla ili obradom poruke primljene od nekog trećeg actora. U modelu svaki actor ima jedinstveni identitet koji se koristi za komunikaciju i vlastiti poštanski pretinac (engl. *mailbox*) u kojem prima poruke.

Poštanski pretinac služi kao spremnik za pohranu u kojeg drugi actori spremaju poruke koje šalju, a actor koji je vlasnik pretinca može iz istog izvući poruke i obraditi ih, odnosno provesti niz akcija u skladu s primljenom porukom [40]. Ukoliko više actora pošalje poruku jednom actoru, poruke se spremaju u poštanski pretinac redoslijedom kojim su primljene. Actor po FIFO (*First-In-First-Out*) principu obrađuje poruke sinkrono jednu po jednu.

Svaki actor posjeduje lokalnu memoriju odvojenu od ostalih actora, odnosno ostatka sustava. Ova lokalna memorija se zove stanje actora i u potpunisti je enkapsulirano unutar actora. Enkapsulacija sprječava actore da jedan drugome zapisuju u dodijeljenu memoriju. Zbog toga jedan actor ne može pristupiti unutarnjem stanju drugog, već može slanjem poruke utjecati na *ponašanje* drugog actora ili zatražiti nekakve vrijednosti iz njegovog internog stanja [40]. Ponašanjem actora smatramo slijed akcija koje će provesti primitkom poruke. Programer koji programira actor sustav mora biti pažljiv da ne bi slučajno “razbio“ ovu enkapsulaciju.

Primitkom poruke, actore može provesti jednu od tri akcije [39]:

- Poslati poruku nekom drugom actoru čiji identitet, odnosno lokacija, su mu poznati
- Stvoriti novog actora s određenim ponašanjem
- Promijeniti svoje ponašanje.

Ova svojstva omogućuju actor modelu da modelira otvorene sustave [39]. Dodavanje novih komponenti se modelira stvaranjem novog actora, a promjenom ponašanja se modelira zamjena komponenti. Veze među actorima i dinamička svojstva sustava se mogu modelirati razmjenom lokacija actora koristeći poruke, što u principu omogućuje izmjenu povezanosti actora, odnosno arhitekture sustava..

Actor model pretpostavlja [39]:

- Poruka će se sigurno isporučiti (stići do cilja) i;
- Actor koji je beskonačno spreman da obradi poruku će u nekom trenutku tu poruku i obraditi.

Dakle, u actor modelu se Istovremenost postiže asinkronom razmjenom poruka. S aspekta actor modela, ponašanja unutar pojedinog actora se odvijaju u potpunosti sinkrono. Naravno, programer koji implementira actor sustav može prekršiti ovu pretpostavku. Ipak sinkrono izvršavanje poruka pojednostavljuje razmišljanje i razumijevanje istovremenog izvršavanja sustava.

Unutarnje stanje različitih actore može ovisiti o sadržaju poruka koje primaju. Pri tome različiti actori mogu pokušati držati referencu na dobivenu poruku i mijenjati njene vrijednosti. Ukoliko bi se ovo dogodilo, moglo bi dovesti do data *racea* ili *racea conditiona*. Štoviše, to bi značilo da jedan actor može pristupiti unutarnjem stanju drugog actora, što bi srušilo pretpostavku o enkapsulaciji unutarnjeg stanja actora. Stoga je važno da su poruke koje actori razmjenjuju nepromjenjive (poglavlje 7.3.2).

S obzirom na to da se unutarnjem stanju actora ne može pristupiti izvana, neki kažu da su oni više objektno orijentiranih od objekata, odnosno da su actori ono što su objekti trebali biti [32]. Štoviše, sam Alan Kay, kao jedan od začetnika objektno-orijentiranog programiranja (OOP) je kazao da je najvažnija ideja OOPa zapravo razmjena poruka [30, 39].

7.5 Let it Crash

Kod klasičnog programiranja sa nitima, često programer programira na defenzivan način tako što pokušava predvidjeti pogreške koje bi se mogle dogoditi i dovesti do pucanja programa. Kada programiramo s actorima, filozofija programiranja koja se primjenjuje se zove *let it crash* [7].

Po tom principu programiranja, umjesto da se programer pokuša obraniti od pogrešaka, pretpostavi da će se one dogoditi i dozvoli da se dogode, ali na jednom izoliranom dijelu aplikacije. Kada govorimo o razvojnom okviru Akka, kojega ćemo koristiti za implementaciju primjera, actori unutar actor sustava su organizirani u hijerarhijsku (stablastu) strukturu [42, 43]. Pri tome je svaki otac automatski zadužen da nadgleda svoju djecu. U slučaju da se u nekom od djece dogodi pogreška, otac može odabrati koju od akcija i strategija nadzora će primijeniti. Akcije koje može primijeniti: 1) dozvoli djetetu da nastavi s izvršavanjem i sačuvaj akumulirano stanje; 2) ponovno pokreni dijete pri čemu se akumulirano stanje briše; 3) u potpunosti zaustavi dijete; i 4) eskaliraj pogrešku na vlastitog roditelja. Strategija određuje hoće li se odabrana akcija primijeniti samo na dijete u kojem se dogodila pogreška ili svu djecu roditelja.

Ovakav način organizacije actora skupa sa strategijama oporavka omogućuje actor sustav da samog sebe zaliječi u slučaju da se dogodi pogreška. Zadatak programera je onda da dijelove programskog koda u kojima postoji rizik da se dogodi pogreška, izdvoji u posebne actore. Na taj način programer izolira rizične funkcije od ostatka sustava. To su, na primjer, dijelovi koji ovise o eksternim izvorima podataka ili drugim komponentama za koje programer ne može garantirati da će uvijek biti dostupne.

7.5.1 Primjer 1 – implementacija

Implementacija primjera 1 koristeći actor model je dana u Dodatak I.

Za implementaciju se koriste 2 tipa actora, *distributer i multiplier,* pri čemu prvi tip (porukom) primi dvije matrice koje je potrebno pomnožiti te za svaki redak prve matrice stvori po jednog actora drugog tipa (*multiplier*). Zadatak multiplier actora je pomnožiti dobiveni redak sa transponiranom drugom matricom te rezultat poslati natrag distributeru. Distributer skuplja djelomične rezultate multiplier actora i slaže ih u konačni rezultat. Jednom kada actori više nisu potrebni, kada su obavili zadani im zadatak, oni se gase slanjem `PoisonPill` poruke.

Za ispitivanje se ponovno množe dvije matrice dimenzija 300x300 sa 100 ponavljanja pri čemu actor sustav stvaramo samo jednom. Prosječna brzina izvršavanja je 287.17 milisekundi pri čemu su u prosjeku korištene 24 različite niti. U ovaj broj nije uračunata nit na kojoj je distributer actor pokrenut. Brzina izvršavanja je dosta slična brzini implementacija sa nitima i sa futurima.

7.5.2 Primjer 2 – implementacija

Budući da enkapsulacija actora sprječava pristup unutarnjem stanju actora, obrazac primjera 2 ćemo implementirati nešto drugačije nego u slučaju prethodna dva primjera. Naime, umjesto da potrošači dobavljaju vrijednosti iz zajedničkog reda, proizvođač će jednome od potrošača poslati stvorenu vrijednost. Ovo odgovara implementaciji obrasca jedan proizvođač i jedan potrošač prezentiranoj u [29].

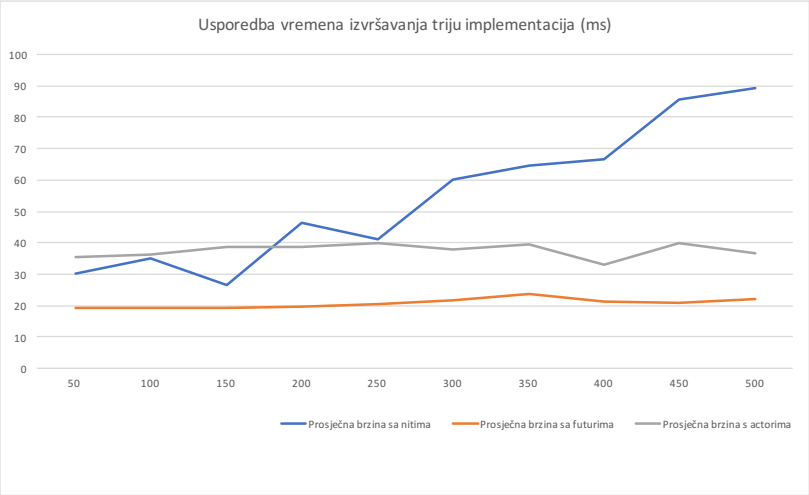
Za implementaciju ovoga koristi se *router*. Router je poseban actor u akka razvojnom okviru koji stvara određen broj actora kojima zatim prosljeđuje poruke po nekom pravilu [44]. Budući da želimo imati kontrolu nad stvaranjem potrošača, koristit ćemo *grupu*. Grupa se u ovom slučaju odnosi na routera koji ne stvara sam actore koji obrađuju poruku, već mu dodjeljujemo putanje unaprijed stvorenih actora. Router odabire actora kojemu će prosljediti poruku po *round-robin* pravilu. Po tom pravilu router svakom od actora prosljeđuje poruke otprilike jednakom frekvencijom.

Implementacija je dana u Dodatak J. U svrhu mjerenja program pokrećemo sa 1000 ponavljanja i promjenjivim brojem potrošača. Rezultati su prikazani u tablici .

	Prosječna brzina izvršavanja u milisekundama (ovisno o broju ponavljanja)	
Broj potrošača	1000 ponavljanja	Prosječni broj korištenih niti
50	35,295	23,6
100	36,136	23,603
150	38,66	23,74
200	38,736	23,64
250	39,916	23,695
300	37,996	23,482
350	39,55	23,657
400	32,939	23,421
450	39,785	23,585
500	36,781	23,76

Tablica 4 Brzina izvršavanja obrasca proizvođač-više potrošača koristeći actore

Važno je još jednom naglasiti da se novi id niti dodaje u kolekciju svaki put kada neki od potrošača actora primi poruku. Ovo utječe na brzinu izvršavanja pa je sam program nešto sporiji, ali dobivamo precizniji prosječan broj korištenih niti u aplikaciji (samo za potrošače jer se broje samo niti na kojima su isti bili pokrenuti).



Slika 10 Usporedba vremena trajanja za sve tri implementacije

8 Budući trendovi?

Budući da se većina programskih paradigmi i jezika pojavilo prije 30-40 godine, prije pojave višejezgrenih i višeprocesorskih računala, 2011-te godine je održan panel na kojem je glavno pitanje bilo: je li potrebno odrediti novi skup programskih jezika s novim paradigmama koje će biti primjerenije programiranju sustava koji će moći iskoristiti snagu višeprocesorskih i višejezgrenih računala ili je dovoljno postojeće programske jezike proširiti s novim mogućnostima [45]. Na panelu je bilo devet sudionika. Neki od njih smatraju da je potrebno dizajnirati nove programske jezike koji će omogućiti jednostavnije programiranje i izražavanje u paralelnom stilu. Drugi smatraju da su postojeći programski jezici adekvatni, da ih je potrebno samo proširiti s primjerenim apstrakcijama i domenskim jezicima (*Domein Specific Language – DSL*). Dvojica sudionika spominju razmjenu poruka kao najprimjereniji model programiranja ovakvih sustava.

8.1 *Reactive isolates*?

Reactive isolates su model istovremenog programiranja čiji je cilj riješiti probleme actor modela [10]. Actori imaju 3 problema: i) "u osnovnom actor modelu, actori ne mogu imati više od jednog ulaza poruka"; ii) osnovni actor model ne može očekivati nekakvu kombinaciju poruka – tj. primiti prvu poruku i čekat da vidi koja će druga poruka stići, već odmah po primitku prve poruke, istu obradi; i iii) metoda za primanje poruke (u Akka razvojnom okviru) nije *građanin prvog reda* koji se može poslati u funkciju ili vratiti iz funkcije.

Model *reactive isolates* se sastoji od tri osnovne apstrakcije [10]:

- *Isolates* – određuje istovremeni izračun i povezano stanje programa;
- Kanala (engl. *channels*);
- Streamovi događaja (engl. *event streams*).

U ovom modelu poruke se nazivaju *događaji* (engl. *event*).

Komunikacija se dijeli na propagaciju događaja među isolatima, koja se temelji na kanalima, i propagaciju unutar isolata koja se temelji na streamovima događaja [10]. Bilo koji isolat može poslati poruku putem bilo kojeg kanala, ali samo vlasnik kanala, koji može biti maksimalno jedan, može procesirati događaj (poruku). Streamovi događaja se ne mogu dijeliti među isolatima, već služe za

propagaciju događaja unutar isolata. Po jedan stream događaja je vezan uz svaki kanal. Oni se mogu slagati u kompozicije koristeći deklarativne kombinatore.

Ovdje smo samo ukratko odlučili upoznati čitatelja sa postojanjem ovog modela. Model je relativno nov te će biti zanimljivo vidjeti hoće li u nekom trenutku dobiti na popularnosti. Ukoliko čitatelja zanimaju dodatni detalji vezani uz ovaj mode, usporedba sa actor modelom te kako on izgleda u primjerima s programskim kodom, upućujemo ga na [10].

9 Zaključak

Seminarski rad smo započeli s opisom aktualnog problema koji je nastao jer su tranzistori postali premali. To dovodi do usporenja razvoja procesora i nemogućnosti daljnjeg povećanja procesorskog takta. S obzirom na to, programeri se više ne mogu osloniti na novu generaciju procesora da bi se njihovi programi mogli izvršavati brže, već moraju naučiti kako iskoristiti potencijal koji pruža pojava procesora s više jezgri.

Programiranje na više jezgri se može smatrati programiranjem raspodijeljenog sustava sa zajedničkom memorijom, gdje su niti raspodijeljene jedinice. Objasnili smo razliku između istovremenog i paralelnog programiranja i izvršavanja te ih doveli u vezu sa raspodijeljenim sustavima. Za raspodijeljene sustave smo rekli da također spadaju u istovremene sustave, ali se najčešće odnose na takve sustave čije su komponente geografski razdvojene.

Nedeterminizam uzrokovan istovremenim programiranjem sa sobom dovodi do novih poteškoća, tipova problema s kojima se ne susrećemo kod običnog sekvencijalnog programiranja. Problemi koje smo opisali su *deadlock*, *race condition* i *data race*, koji su rezultat nedeterminističkog izvršavanja niti, a koji se iz istog razloga mogu pojaviti samo ponekad. Ovi problemi dodatno otežavaju programiranje sustava koji će koristiti više procesorskih jezgri jer se nekada ne mogu riješiti nego uvođenjem sinkronizacije, koja zauzvrat uzrokuje usko grlo u višetnitnom programu. Zbog toga program ne može adekvatno iskoristiti dostupne jezgre procesora.

Potom smo objasnili tipove istovremenog programiranja s obzirom na način komunikacije među nitima – zajedničkom memorijom i razmjenom poruka.

Opisali smo i predstavili tri metode istovremenog programiranja koje smo usporedili po brzinama izvršavanja na temelju dvaju primjera. Metode uključuju klasičnog programiranje s nitima, funkcionalno istovremeno programiranje koristeći *future* i actor model. Za svaku od metoda smo iznijeli njihove prednosti i mane. Dodatno, smo ih usporedili i po procjeni broja korištenih niti za izvršavanje. Naglasili smo da procjena broja niti kod primjera obrasca proizvođač-više potrošača ovisi i na kojem se mjestu u kodu u kolekciju sprema identifikator niti koja u tom trenutku izvršava taj dio programskog koda. Na mjerenja utječe i nedeterministička priroda niti pa ponekad prosječan broj korištenih niti, a i prosječno vrijeme izvršavanja može varirati. Stoga se ove brzine moraju uzeti u obzir kao informirana procjena.

Ipak, u slučaju dva promatrana primjera i prezentiranih načina implementacija, mjerenja su pokazala da se daleko najveći broj niti koristi onda kada ih sami stvaramo u klasičnom načinu programiranja sa nitima. S druge strane, druge dvije metode nastoje optimizirati broj korištenih niti pa je taj broj drastično manji. Pri tome je najmanje niti procijenjeno za funkcionalno istovremeno programiranje sa futurima. Za primjer 2 smo vidjeli i da su brzine izvršavanja praktički neovisne o broju korištenih

potrošača u slučaju actora i futura. S druge strane, vidjeli smo da se brzina izvršavanja smanjuje u slučaju klasične implementacije sa nitima. Podsjetimo se da u klasičnoj implementaciji s nitima svaki potrošač odgovara jednoj niti, dok kod drugih implementacija to nije tako. Stoga se može zaključiti da nekontrolirano stvaranje niti može dovesti do sporijeg izvršavanja programa.

Od promatranih metoda istovremenog programiranja, metoda programiranja klasičnim nitima i funkcionalno istovremeno programiranje su metode koje koriste zajedničku memoriju. Actor model je jedina od promatranih metoda u kojoj se komunikacija ostvaruje razmjenom poruka. Stoga je ovaj model adekvatan za izradu distribuiranih sustava. Štoviše, uklanja potrebu za defenzivnim programiranjem i daje sustavu mogućnost samo oporavka vlastitih komponenti.

Svaka od metoda ima svoje prednosti i mane, potrebno je odabrati onu koja najbolje odgovara danom problemu. Ukoliko je programeru potrebna maksimalna kontrola nad nitima ili da ima niti koje će biti posvećene isključivo jednom zadatku, onda je klasično programiranje s nitima najbolji izbor. U slučaju izrade distribuiranih sustava, actor model predstavlja najbolju prezentiranu opciju. U slučaju paralelnog programiranja kojemu je cilj ubrzanje izvršenja nekakvog zadatke, onda bi programiranje sa futurima moglo biti najbolje.

Programski kod za sve primjere se može pronaći na GitHubu na adresi:

https://github.com/mrn-aglic/DistributedSystems_SeminarExamples

Dodatak A

Primjer za demonstraciju *race conditiona* naveden u poglavlju 5.1.1.

```
object Main_RC {

  def main(args: Array[String]): Unit = {

    var x: ExpensiveEntity = null
    var y: ExpensiveEntity = null

    val t1 = thread { x = Singleton.getInstance() }
    val t2 = thread { y = Singleton.getInstance() }

    t1.start()
    t2.start()

    t1.join()
    t2.join()

    println(x eq y)
  }

  def thread(body: => Unit): Thread = new Thread{

    override def run(): Unit = body
  }
}

class ExpensiveEntity {

  val content = "some expensive content"
}

object Singleton {

  var instance: ExpensiveEntity = null

  def getInstance(): ExpensiveEntity = {

    if (instance == null) {

      println(s"Current thread id: ${Thread.currentThread().getId}")

      instance = new ExpensiveEntity
    }

    instance
  }
}
```

Dodatak B

Primjer za demonstraciju *data racea* naveden u poglavlju 5.1.1.

```
object Main_DR {

  def main(args: Array[String]): Unit = {

    val xs = (1 to 100).par

    var sum = 0

    xs.foreach(x => {

      sum = sum + x
    })

    println(s"sum is: $sum")
    println(s"should be: ${(1 to 100).sum}")
  }
}
```

Dodatak C

Primjer za demonstraciju *deadlocka* naveden u poglavlju 5.1.2.

```
object Main_Deadlock {

  import DeadlockExample._

  def main(args: Array[String]): Unit = {

    val a = new Component(1)
    val b = new Component(2)

    var t1 = thread { doComplexWork(a, b) }
    var t2 = thread { doComplexWork(b, a) }

    t1.join()
    t2.join()

    println("Work is done!")
  }

  def thread(body: => Unit): Thread = {

    val t = new Thread{

      override def run(): Unit = body
    }

    t.start()

    t
  }
}

class Component(id: Int) {

  def doWork(): Unit = println(s"$id doing work")
}

object DeadlockExample {

  def doComplexWork(a: Component, b: Component): Unit = a.synchronized {

    a.doWork()

    b.synchronized {

      b.doWork()
    }
  }
}
```

Dodatak D

Istovremena implementacija množenja matrica uz pomoć niti.

```
package comparison_examples.threadImplementations

import scala.concurrent.Future
import scala.util.Random

/**
  * Created by Marin on 28/04/2017.
  */
object MatrixMultiplication {

   import common.MeasurementHelpers
   import common.Configuration
   import common.ThreadHelpers.thread

   val r = new Random()

   // pomoćna metoda za stvaranje matrica
    def createMatrix(row: Int, col: Int): List[List[Int]] =
       (1 to row).map(x => (1 to col).map(_ => r.nextInt(10)).toList).toList

    def main(args: Array[String]): Unit = {

       val n: Double = Configuration.runTimes

       val runNTimes = MeasurementHelpers.runNTimes(n.toInt) _

       val results = runNTimes {

          val nrow = Configuration.numberOfRows
          val ncol = Configuration.numberOfCols

          // stvorimo prvu matricu
          val firstMatrix = createMatrix(nrow, ncol)
          // stvorimo drugu matricu
          val secondMatrix = createMatrix(nrow, ncol)

          // posao koji se želi odraditi: Množenje matrica
          matrixMultiply(nrow, ncol, firstMatrix, secondMatrix)
       }

       // ispišemo prosječnu brzinu izvršavanja
      println(s"Average duration: ${results.map(x => x._2).sum / n}")
       // ispišemo prosječan broj niti

       println(s"Average number of distinct threads: ${results.map(x =>
x._3.length).sum / n}")

       println()
    }
```

```
   def matrixMultiply(nrow: Int, ncol: Int, firstMatrix: List[List[Int]],
secondMatrix: List[List[Int]]): Unit = {

      val result = Array.ofDim[List[Int]](nrow)

      var threads = List[Thread]()

      val array = Array.ofDim[Long](nrow)
      MeasurementHelpers.setNumThreads(array)

      for(i <- 0 until nrow){

         val row = firstMatrix(i)

         // stvorimo po jednu nit za svaku red prve matrice
         val t = thread {

            MeasurementHelpers.addCurrentThread(i)

            var newRow = List[Int]()

            for (j <- 0 until nrow) {

               var sum = 0

               for(z <- 0 until ncol) {

                  sum += row(z) * secondMatrix(z)(j)
               }

               newRow = newRow :+ sum
            }

            result.update(i, newRow)
         }

         threads = t :: threads
      }

      threads.foreach(_.join())
   }
}
```

Dodatak E

Višenitna implementacija obrasca *proizvođač-više potrošača* koristeći niti.

```
package comparison_examples.threadImplementations

import common.MeasurementHelpers._
import common._

import scala.collection.mutable

case class Item(value: Int)

object ProducerConsumer {

   private val sharedQueue = mutable.Queue[Item]()

   def main(args: Array[String]): Unit = {

      val n: Double = if(args.nonEmpty) args(0).toDouble else
Configuration.runTimes.toDouble
      val numConsumers = if(args.length == 2) args(1).toInt else
Configuration.numberOfConsumers

      val funRunNTimes = MeasurementHelpers.runNTimes(n.toInt) _

      val results = funRunNTimes {

         Helper.isFinished = false

         setNumThreads(Array.ofDim[Long](numConsumers))

         val producer = new Producer(sharedQueue)
         val cs = startConsumers(numConsumers, List[Consumer](), sharedQueue)

         cs.foreach(_.join())
         producer.join()

         println(s"Total items consumed: ${cs.map(x =>
x.getObtainedItems.length).sum}")
      }

      println(s"Run times: $n")
      println(s"Average duration: ${results.map(x => x._2).sum / n} milliseconds")

      println()
   }

   def startConsumers(max: Int, result: List[Consumer], sharedQueue:
mutable.Queue[Item]): List[Consumer] =
      if (max == 0) result
      else {
         val consumer = new Consumer(max - 1, sharedQueue)
         consumer.start()

         startConsumers(max - 1, consumer :: result, sharedQueue)
      }
}
object Helper {
   var isFinished = false
}
```

```
class Producer(sharedQueue: mutable.Queue[Item]) extends Thread {

   override def run(): Unit = {

      val n = Configuration.workToProduce

      for (i <- 1 to n) {

         val item = Item(i)

         // pristup dijeljenom resursu mora biti sinkroniziran
         sharedQueue.synchronized {

            sharedQueue.enqueue(item)
            sharedQueue.notify()
         }
      }
      sharedQueue.synchronized {

         Helper.isFinished = true
         sharedQueue.notifyAll()
      }
   }

   start()
}
class Consumer(index: Int, sharedQueue: mutable.Queue[Item]) extends Thread {

   private var obtainedItems = List[Item]()

   def getObtainedItems: List[Item] = obtainedItems

   override def run(): Unit = {

      addCurrentThread(index)

      while (sharedQueue.nonEmpty || !Helper.isFinished) {

         obtainedItems = getItem match {

            case None => obtainedItems
            case Some(item) => item :: obtainedItems
         }
      }
   }

   def getItem: Option[Item] =
   // pristup dijeljenom resurse (redu) mora biti sinkroniziran
      sharedQueue.synchronized {

         while (sharedQueue.isEmpty && !Helper.isFinished) {
            sharedQueue.wait()
         }

         if (sharedQueue.nonEmpty)
            Some(sharedQueue.dequeue())
         else
            None
      }
   def printObtainedItems(): Unit =
      obtainedItems.foreach(x => print(s"\t ${x.value}"))
}
```

Dodatak F

Primjer sa matricama implementiran funkcionalnom programskom paradigmom koristeći *future*.

```
package comparison_examples.futuresimplementation

import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.duration._
import scala.concurrent.{Await, Future}
import scala.util.Random

object MatrixMultiplication {

   import common.MeasurementHelpers._
   import common.{Configuration, MeasurementHelpers}

   val r = new Random()

   def main(args: Array[String]): Unit = {

      val n: Double = Configuration.runTimes

      val runNTimes = MeasurementHelpers.runNTimes(n.toInt) _

      val results = runNTimes {

         val nrow = Configuration.numberOfRows
         val ncol = Configuration.numberOfCols

         setNumThreads(Array.ofDim[Long](nrow))

         // stvorimo prvu matricu
         val firstMatrix = createMatrix(nrow, ncol)
         // stvorimo drugu matricu
         val secondMatrix = createMatrix(nrow, ncol)

         val transposedMatrix = transpose(secondMatrix)

         val f = matrixMultiply(nrow, ncol, firstMatrix, transposedMatrix)

         Await.result(f, Duration.Inf)
      }

      // ispišemo prosječnu brzinu izvršavanja
      println(s"Average duration: ${results.map(x => x._2).sum / n}")
      // ispišemo prosječan broj niti
      println(s"Average number of distinct threads: ${results.map(x =>
x._3.length).sum / n}")
      println()
   }

   def matrixMultiply(nrow: Int, ncol: Int, firstMatrix: List[List[Int]],
secondMatrix: List[List[Int]]): Future[List[List[Int]]] = {

      val seqOfFutures = firstMatrix.zipWithIndex.map(x => {

         multiplyRowByMatrix(x._2, x._1, secondMatrix)
      })
      val allFutures = Future.sequence(seqOfFutures)

      allFutures
   }
```

```
   def multiplyRowByMatrix(index: Int, row: List[Int], secondMatrix:
List[List[Int]]): Future[List[Int]] = Future {

      addCurrentThread(index)

      secondMatrix.map(x => multiplyRows(row, x))
   }

   def multiplyRows(r1: List[Int], r2: List[Int]): Int =
      r1.zip(r2).map { case (x, y) => x * y }.sum

   // pomoćna metoda za stvaranje matrica
   def createMatrix(row: Int, col: Int): List[List[Int]] =
      (1 to row).map(x => (1 to col).map(_ => r.nextInt(10)).toList).toList

   def transpose(m: List[List[Int]]): List[List[Int]] =
      if (m.head.isEmpty) Nil
      else m.map(_.head) :: transpose(m.map(_.tail))

}
```

Dodatak G

Implementacija istovremenog množenja matrica koristeći klasično programiranje s nitima i funkcionalni stil programiranja.

```
package comparison_examples.threadImplementations

import scala.concurrent.Future
import scala.util.Random

/**
  * Created by Marin on 28/04/2017.
  */
object MatrixMultiplication_Functional {

   import common.MeasurementHelpers
   import common.Configuration
   import common.ThreadHelpers.thread

   val r = new Random()

   // pomoćna metoda za stvaranje matrica
   def createMatrix(row: Int, col: Int): List[List[Int]] =
      (1 to row).map(x => (1 to col).map(_ => r.nextInt(10)).toList).toList

   def main(args: Array[String]): Unit = {

      val n: Double = Configuration.runTimes

      val runNTimes = MeasurementHelpers.runNTimes(n.toInt) _

      val results = runNTimes {

         val nrow = Configuration.numberOfRows
         val ncol = Configuration.numberOfCols

         // stvorimo prvu matricu
         val firstMatrix = createMatrix(nrow, ncol)
         // stvorimo drugu matricu
         val secondMatrix = createMatrix(nrow, ncol)

         val transposedMatrix = transpose(secondMatrix)

         // posao koji se želi odraditi: Množenje matrica
         matrixMultiply(nrow, ncol, firstMatrix, transposedMatrix)
      }

      // ispišemo prosječnu brzinu izvršavanja
      println(s"Average duration: ${results.map(x => x._2).sum / n}")
      // ispišemo prosječan broj niti
      println(s"Average number of distinct threads: ${results.map(x =>
x._3.length).sum / n}")

      println()
   }
```

```
   def matrixMultiply(nrow: Int, ncol: Int, firstMatrix: List[List[Int]],
secondMatrix: List[List[Int]]): List[List[Int]] = {

      var result = Array.ofDim[List[Int]](nrow)

      MeasurementHelpers.setNumThreads(Array.ofDim[Long](nrow))

      val threads = firstMatrix.zipWithIndex.map { case(x, index) => thread {

         MeasurementHelpers.addCurrentThread(index)

         result.update(index, multiplyRowByMatrix(x, secondMatrix))
      }}

      threads.foreach(_.join())

      result.toList
   }

   def multiplyRowByMatrix(row: List[Int], secondMatrix: List[List[Int]]):
List[Int] =  {

      secondMatrix.map(x => multiplyRows(row, x))
   }

   def multiplyRows(r1: List[Int], r2: List[Int]): Int =
      r1.zip(r2).map { case (x, y) => x * y }.sum

   def transpose(m: List[List[Int]]): List[List[Int]] =
      if(m.head.isEmpty) Nil
      else m.map(_.head) :: transpose(m.map(_.tail))
}
```

Dodatak H

Korištena implementacija obrasca proizvođač-više potrošača koristeći *future*.

```
package comparison_examples.futuresimplementation
import common.MeasurementHelpers._
import common._
import scala.collection.mutable
import scala.concurrent.blocking
import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.duration.Duration
import scala.concurrent.{Await, Future, Promise}

case class Item(value: Int)

object ProducerConsumer {

   private val sharedQueue = mutable.Queue[Item]()

   def main(args: Array[String]): Unit = {

      val n: Double = if (args.nonEmpty) args(0).toDouble else
Configuration.runTimes.toDouble
      val numConsumers = if (args.length == 2) args(1).toInt else
Configuration.numberOfConsumers

      val funRunNTimes = MeasurementHelpers.runNTimes(n.toInt) _

      val results = funRunNTimes {
         val p = Promise[Boolean]()
         setNumThreads(Array.ofDim[Long](numConsumers))

         val producer = new Producer(p, sharedQueue)
         val cs = startConsumers(numConsumers, List[Consumer](), sharedQueue, p)

         val fp = producer.start()
         val fs = cs.map(x => x.start())

         val f = Future.sequence(fs)
         Await.result(fp, Duration.Inf)
         Await.result(f, Duration.Inf)

         val allElementsObtained = cs.flatMap(_.getObtainedItems)

         println(allElementsObtained.length)
      }
      println(s"Run times: $n")
      println(s"Average number of distinct threads: ${results.map(x =>
x._3.length).sum / n}")
      println(s"Average duration: ${results.map(x => x._2).sum / n} milliseconds")

      println()
   }

   def startConsumers(max: Int, result: List[Consumer], sharedQueue:
mutable.Queue[Item], p: Promise[Boolean]): List[Consumer] =
      if (max == 0) result
      else {
         val consumer = new Consumer(max - 1, p, sharedQueue)
         startConsumers(max - 1, consumer :: result, sharedQueue, p)
      }
}
```

```
class Producer(p: Promise[Boolean], sharedQueue: mutable.Queue[Item]) {

   def start(): Future[Unit] = Future {

      val n = Configuration.workToProduce

      for (i <- 1 to n) {
         val item = Item(i)
         // pristup dijeljenom resursu mora biti sinkroniziran
         sharedQueue.synchronized {

            sharedQueue.enqueue(item)
            sharedQueue notify()
         }
      }

      p success true

      sharedQueue.synchronized {

         sharedQueue.notifyAll()
      }
   }
}
class Consumer(index:Int, p: Promise[Boolean], sharedQueue: mutable.Queue[Item]) {

   private var obtainedItems = List[Item]()

   def getObtainedItems: List[Item] = obtainedItems

   def start(): Future[Unit] = Future {

      while (sharedQueue.nonEmpty || !p.isCompleted) {

         obtainedItems = getItem match {

            case None => obtainedItems
            case Some(item) => item :: obtainedItems
         }
      }
   }
   def getItem: Option[Item] = blocking {

      sharedQueue.synchronized {

         while (sharedQueue.isEmpty && !p.isCompleted) {

            sharedQueue.wait()
         }
         addCurrentThread(index)

         val result = if (sharedQueue.nonEmpty)
            Some(sharedQueue.dequeue())
         else
            None

         result
      }
   }
   def printObtainedItems(): Unit =
      obtainedItems.foreach(x => print(s"\t ${x.value}"))
}
```

Dodatak I

Implementacija množenja matrica koristeći razvojni okvir Akka i actor model.

```
package comparison_examples.actorsimplementation

import akka.actor.{Actor, ActorRef, ActorSystem, PoisonPill, Props}
import akka.pattern.ask
import akka.util.Timeout
import common.MeasurementHelpers
import common.Types.{Matrix, RowNumber}

import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.Await
import scala.concurrent.duration._
import scala.util.{Failure, Random, Success}

case class Matrices(m1: Matrix[Int], m2: Matrix[Int])
case class MultiplyRowsByMatrix(i: RowNumber, row: List[Int], matrix: Matrix[Int])
case class RowsMultiplicationResult(i: RowNumber, values: List[Int])
case class Result(matrix: Matrix[Int])
object PropsConfigurations {
   def matrixProps(): Props = Props[Distributer]
   def multiplierProps(): Props = Props[Multiplier]
}
object MatrixMultiplication {

   import PropsConfigurations._
   import common.{Configuration, MeasurementHelpers}
   val r = new Random()

   def main(args: Array[String]): Unit = {

      implicit val timeout = Timeout(10 seconds)

      var system = ActorSystem.create("matrix-multiplication")

      val n: Double = Configuration.runTimes

      val runNTimes = MeasurementHelpers.runNTimes(n.toInt) _

      val results = runNTimes {

         val matrixActor = system.actorOf(matrixProps())
         val nrow = Configuration.numberOfRows
         val ncol = Configuration.numberOfCols

         MeasurementHelpers.setNumThreads(Array.ofDim[Long](nrow))

         // stvorimo prvu matricu
         val firstMatrix = createMatrix(nrow, ncol)
         // stvorimo drugu matricu
         val secondMatrix = createMatrix(nrow, ncol)

         val f = matrixActor ? Matrices(firstMatrix, secondMatrix)

         f.onComplete {

            case Success(x) => matrixActor ! PoisonPill
            case Failure(er) =>
         }

         Await.result(f, Duration.Inf)
      }
```

```
      println(s"Average duration: ${results.map(x => x._2).sum / n}")
      println(s"Average number of distinct threads: ${results.map(x =>
x._3.length).sum / n}")
      println()
      system.terminate()
   }
   // pomoćna metoda za stvaranje matrica
   def createMatrix(row: Int, col: Int): List[List[Int]] =
      (1 to row).map(x => (1 to col).map(_ => r.nextInt(10)).toList).toList

}
class Distributer extends Actor {

   import PropsConfigurations.multiplierProps

   var matrix: Array[List[Int]] = _
   var numberOfReceivedResponses = 0
   var respondToTopLevel: ActorRef = _

   def transpose(m: Matrix[Int]): Matrix[Int] =
      if (m.head.isEmpty) Nil
      else m.map(_.head) :: transpose(m.map(_.tail))

   def receive: Receive = {
      case Matrices(m1, m2) =>

         respondToTopLevel = sender
         val transposed = transpose(m2)
         matrix = Array.ofDim[List[Int]](m1.length)

         m1.zipWithIndex.foreach { case (row, i) =>
            val child = context.actorOf(multiplierProps())
            child ! MultiplyRowsByMatrix(i, row, transposed)
         }
      case RowsMultiplicationResult(i, row) =>

         matrix(i) = row
         numberOfReceivedResponses = numberOfReceivedResponses + 1

         sender ! PoisonPill

         if (numberOfReceivedResponses == matrix.length) {
            respondToTopLevel ! Result(matrix.toList)
         }
   }
}
class Multiplier extends Actor {
   def receive: Receive = {

      case MultiplyRowsByMatrix(i, row, matrix) =>

         MeasurementHelpers.addCurrentThread(i)

         val result = matrix.map(x => multiplyRows(row, x))

         sender ! RowsMultiplicationResult(i, result)
   }

   def multiplyRows(r1: List[Int], r2: List[Int]): Int =
      r1.zip(r2).map { case (x, y) => x * y }.sum
}
```

Dodatak J

Implementacija obrasca proizvođač-više potrošača koristeći *actore*.

```
package comparison_examples.actorsimplementation

/**
  * Created by Marin on 19/05/2017.
  */

import akka.actor.{Actor, ActorRef, ActorSystem, PoisonPill, Props, Terminated}
import akka.pattern.ask
import akka.routing.{Broadcast, RoundRobinGroup, TailChoppingGroup}
import akka.util.Timeout
import common._

import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.duration._
import scala.concurrent.{Await, Future}
import scala.util.{Failure, Success}


object ProducerActor {

   def apply(): Props = Props(new ProducerActor())

   case class Start(max: Int, pool: ActorRef)

   case class Produce(index: Int)
}

object ConsumerActor {

   import Common._

   def apply(index: Int): Props = Props(new ConsumerActor(index))

   case class ObtainedItems(items: List[Item])

   case object SendObtained
}

object RouterProps {

   def apply(paths: List[String]): Props = RoundRobinGroup(paths).props()
}

object Common {

   case class Item(value: Int)

   case object Finished

}
```

```
object ProducerConsumer {
   import ConsumerActor._
   import ProducerActor._
   def main(args: Array[String]): Unit = {

      implicit val timeout = Timeout(10 seconds)

      val n: Double = if (args.nonEmpty) args(0).toDouble else
Configuration.runTimes.toDouble
      val numConsumers = if (args.length == 2) args(1).toInt else
Configuration.numberOfConsumers

      val funRunNTimes = MeasurementHelpers.runNTimes(n.toInt) _

      val system = ActorSystem("producer-consumer")

      MeasurementHelpers.setNumThreads(Array.ofDim[List[Long]](numConsumers))

      val results = funRunNTimes {

         val producer = system.actorOf(ProducerActor())
         val consumers = for (i <- 0 until numConsumers) yield
system.actorOf(ConsumerActor(i))

         val paths = consumers.map(_.path.toString).toList
         val router = system.actorOf(RouterProps(paths))
         val f = producer ? Start(Configuration.workToProduce, router)

         val obtainedF = f flatMap (_ => {

            val z = consumers.map(x => (x ?
SendObtained).mapTo[ObtainedItems].map(x => x.items))

            Future.sequence(z).map(x => x.flatten)
         })

         obtainedF onComplete {

            case Success(x) =>

               println(s"Number of all elements obtained: ${x.length}")

               router ! Broadcast(PoisonPill)
            case Failure(err) => println(s"An error occurred: $err")
         }

         Await.result(obtainedF, Duration.Inf)

         router ! PoisonPill
      }


      println(s"Run times: $n")
      println(s"Average number of distinct threads: ${results.map(x =>
x._3.length).sum / n}")
      println(s"Average duration: ${results.map(x => x._2).sum / n} milliseconds")
      println()

      system.terminate().onComplete {

         case Success(_) => println("Shutdown complete")
         case Failure(err) => println(err)
      }
   }
}
```

```
class ProducerActor extends Actor {

   import Common._
   import ProducerActor._

   var respondToTopLevel: ActorRef = _

   def receive: Receive = {

      case Start(max, pool) =>

         println("starting...")

         respondToTopLevel = sender()
         context.become(produceWork(max, pool))

         self ! Produce(0)
   }

   def produceWork(max: Int, pool: ActorRef): Receive = {

      case Produce(index) if index < max =>

         val item = Item(index)

         pool ! item

         self ! Produce(index + 1)

      case Produce(index) =>

         respondToTopLevel ! Finished

         self ! PoisonPill
   }
}

class ConsumerActor(index: Int) extends Actor {

   import Common._
   import ConsumerActor._

   private var obtainedItems = List[Item]()

   def receive: Receive = {

      case msg@Item(x) =>

         MeasurementHelpers.addCurrentThread(index)

         obtainedItems = msg :: obtainedItems

      case SendObtained =>

         sender() ! ObtainedItems(obtainedItems)
   }
}
```

Literatura

[1] G. E. Moore, "Cramming more components onto integrated circuits (Reprinted from Electronics, pg 114-117, April 19, 1965)," *Proc. Ieee*, vol. 86, no. 1, pp. 82–85, 1965.

[2] R. Courtland, "Intel Finds Moore's Law's Next Step at 10 Nanometers," *IEEE Spectrum*, 2016. [Online]. Available: http://spectrum.ieee.org/semiconductors/devices/intel-finds-moores-laws-next-step-at-10-nanometers. [Accessed: 01-Apr-2017].

[3] M. Bohr, "A 30 Year Retrospective on Dennard's MOSFET Scaling Paper," *IEEE Solid-State Circuits Newsl.*, vol. 12, no. 1, pp. 11–13, 2007.

[4] K. Sankaranarayanan, "Thermal Modeling and Management of Microprocessors," 2009.

[5] N. S. Kim *et al.*, "Leakage Current," *Computer (Long. Beach. Calif).*, vol. 36, no. 12, pp. 68–75, 2003.

[6] R. Singh, D. Sahu, N. K. Shukla, P. Bhatnagar, Geetanjali, and A. Goel, "Analysis of the Effect of Temperature Variations on Sub-threshold Leakage Current in P3 and P4 SRAM Cells at Deep Sub-micron CMOS Technology," *Int. J. Comput. Appl.*, vol. 35, no. 5, pp. 8–13, 2011.

[7] D. Wyatt, *Akka Concurrency*. 2013.

[8] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala*, 3rd ed. Artima Press, 2016.

[9] "What's new in C# 7.0," 2016. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/articles/csharp/whats-new/csharp-7. [Accessed: 27-Apr-2017].

[10] A. Prokopec and M. Odersky, "Isolates, Channels, and Event Streams for Composable Distributed Programming," in *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, 2015, pp. 171–182.

[11] Brian Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea, *Java Concurrency in Practice*, 1st ed. Addison-Wesley Professional, 2006.

[12] "About Processes and Threads." [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/ms681917(v=vs.85).aspx. [Accessed: 08-Apr-2017].

[13] "Parallelism vs. Concurrency," 2014. [Online]. Available: https://wiki.haskell.org/Parallelism_vs._Concurrency. [Accessed: 01-Apr-2017].

[14] "Defining Multithreading Terms," 2010. [Online]. Available: http://docs.oracle.com/cd/E19455-01/806-5257/6je9h032b/index.html. [Accessed: 01-Apr-2017].

[15] A. Prokopec, *Learning Concurrent Programming in Scala*, 1st ed. 2014.

[16] P. Butcher, *Seven Concurrency Models in Seven Weeks*. Pragmatic Bookshelf, 2014.

[17] S. Marlow, *Parallel and Concurrent Programming in Haskell*, 1st ed. O'Reilly Media, 2013.

[18] J. Regehr, "Race Condition vs. Data Race," 2011. [Online]. Available: http://blog.regehr.org/archives/490. [Accessed: 10-Apr-2017].

[19] S. Ghosh, *Distributed Systems: An Algorithmic Approach*, 1st ed. Chapman and Hall/CRC, 2006.

[20] C. H. Papadimitriou, *Computational Complexity*, 1st ed. Pearson, 1993.

[21] S. K. Prasad, A. Gupta, A. L. Rosenberg, A. Sussman, and C. Weems, *Topics in parallel and distributed computing : introducing concurrency in undergraduate courses*. 2015.

[22] H. El-Rewini and M. Abd-El-Barr, *Advanced Computer Architecture and Parallel Processing*. 2005.

[23] P. Dewan, "Class notes," 2006. [Online]. Available: http://www.cs.unc.edu/~dewan/242/s07/notes/ipc/node9.html. [Accessed: 26-Apr-2017].

[24] R. Thurlow, "RPC: Remote Procedure Call Specification Version 2," 2009. [Online]. Available: https://tools.ietf.org/html/rfc5531. [Accessed: 26-Apr-2017].

[25] M. P. I. Forum, "MPI: A Message-Passing Interface Standard Version 3.0." 2012.

[26] "A High Performance Message Passing Library," 2017. [Online]. Available: https://www.open-mpi.org. [Accessed: 26-Apr-2017].

[27] "Intel MPI Library." .

[28] "MPI: A message-passing interface standard," 2012.

[29] V. Pankratius, F. Schmidt, and G. Garret, "Combining Functional and Imperative Programming for Multicore Software An Empirical Study Evaluating Scala and Java," in *34th International Conference on Software Engineering*, 2012, pp. 123–133.

[30] "No Title." [Online]. Available: http://docs.oracle.com/javase/6/docs/api/java/lang/Thread.html#getId(). [Accessed: 15-May-2017].

[31] "Akka and the Java Memory Model." [Online]. Available: http://doc.akka.io/docs/akka/2.3.9/general/jmm.html. [Accessed: 23-May-2017].

[32] M. Fusco, "Comparing Different Concurrency Models on the JVM," 2014. .

[33] P. Chiusano and R. Bjarnason, *Functional Programming in Scala*. Manning Publications, 2014.

[34] M. Fogus, *Functional JavaScript: Introducing Functional Programming with Underscore.js*, 1st ed. O'Reilly Media, 2013.

[35] G. Hutton, *Programming in Haskell*, 1st ed. Cambridge University Press, 2007.

[36] "Functional Programming," 2014. [Online]. Available: https://wiki.haskell.org/Functional_programming#Features_of_functional_languages. [Accessed: 04-May-2017].

[37] "Immutable Objects." .

[38] P. Haller, A. Prokopec, H. Miller, V. Klang, R. Kuhn, and V. Jovanovic, "Futures and Promises." [Online]. Available: http://docs.scala-lang.org/overviews/core/futures.html. [Accessed: 05-May-2017].

[39] C. A. Varela, *Programming Distributed Computing Systems*, 1st ed. MIT Press, 2013.

[40] S. Miriyala, G. Agha, and Y. Sami, "Visualizing actor programs using predicate transition nets," *Journal of Visual Languages and Computing*, vol. 3, no. 2. pp. 195–220, 1992.

[41] "Dr. Alan Kay on the Meaning of 'Object-Oriented Programming,'" 2003. [Online]. Available:

http://userpage.fu-berlin.de/~ram/pub/pub_jf47ht81Ht/doc_kay_oop_en. [Accessed: 14-May-2017].

[42] "Akka." [Online]. Available: http://akka.io. [Accessed: 18-May-2017].

[43] "Supervision and Monitoring." [Online]. Available: http://doc.akka.io/docs/akka/current/general/supervision.html. [Accessed: 23-May-2017].

[44] "Akka Routing." [Online]. Available: http://doc.akka.io/docs/akka/current/scala/routing.html. [Accessed: 22-May-2017].

[45] S. T. . Taft *et al.*, "Multicore, manycore, and cloud computing: Is a new programming language paradigm required?," *Proc. ACM Int. Conf. companion Object oriented Program. Syst. Lang. Appl. companion*, pp. 165–169, 2011.