Distribuirano računanje pomoću programskog jezika Python

Barišić, Martina

Master's thesis / Diplomski rad

2018

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Science / Sveučilište u Zagrebu, Prirodoslovno-matematički fakultet**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:217:657008

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-04-19**

Repository / Repozitorij:

Repository of the Faculty of Science - University of Zagreb

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Martina Barišić

DISTRIBUIRANO RAČUNANJE POMOĆU PROGRAMSKOG JEZIKA PYTHON

Diplomski rad

Voditelj rada:
prof. dr. sc. Robert Manger

Zagreb, rujan, 2018.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Sadržaj

Sadržaj **iii**

Uvod **1**

1 Općenito o paralelnim i distribuiranim sustavima **2**
1.1 Osnovni pojmovi. Razlike između sustava 2
1.2 Amdahlov zakon 4
1.3 Asinkrono programiranje 6
1.4 Višedretvenost 19
1.5 Višestruki procesi 22
1.6 Višeprocesorski redovi 26

2 Paketi za izgradnju distribuiranih aplikacija **28**
2.1 Celery 28
2.2 Uspostavljanje okruženja 29
2.3 Python-RQ 34
2.4 Pyro 35

3 Studijski primjeri **39**
3.1 Razmjena valuta 39
3.2 Distribuirano sortiranje 42

Zaključak **46**

Bibliografija **48**

Uvod

Ovaj rad opisuje načine realizacije distribuiranog računanja pomoću programskog jezika Python (verzija 3.5).

U prvom poglavlju ukratko će se ponoviti pojmovi paralelnog i distribuiranog sustava te razlike među njima. Proći će se kroz Amdahlov zakon, objasniti činjenica zašto je on bitan te što znači asinkrono programiranje. U drugom dijelu prvog poglavlja opisat će se višedretvenost, višestruki procesi i višestruki redovi te koja je njihova uloga u distribuiranom sustavu.

U drugom poglavlju, opisat će se Pythonov alat Celery te njegove alternative. Pokazat će se kako se uspostavlja okruženje za rad na nekoliko jednostavnih primjera te što je sve potrebno da bi aplikacija korektno funkcionirala.

Zadnje poglavlje donosi dva studijska primjera distribuiranih aplikacija u kojima se primjenjuje Celery. Osim što će se pokazati kako aplikacije funkcioniraju, na drugom studijskom primjeru proučit će se i vrijeme izvođenja takve aplikacije.

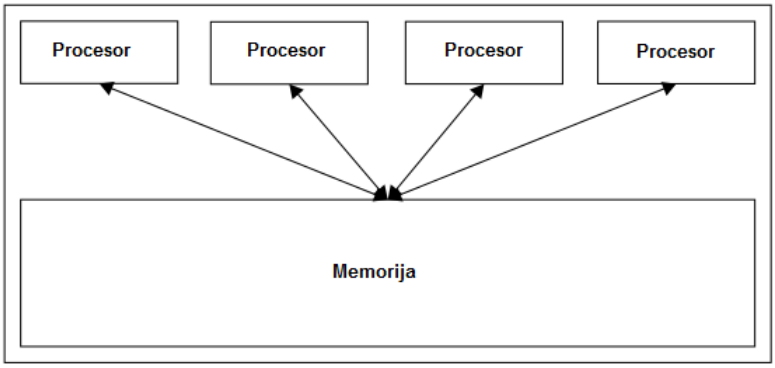
Poglavlje 1

Općenito o paralelnim i distribuiranim sustavima

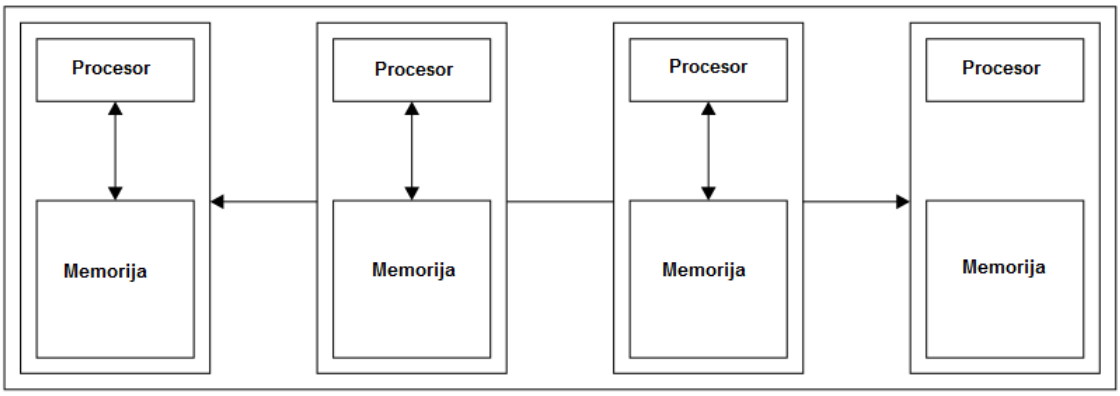
1.1 Osnovni pojmovi. Razlike između sustava

Na početku ovog poglavlja navedeni su neki osnovni pojmovi vezani uz distribirane sustave.

Najprije su definirani pojmovi paralelnog i distribuiranog računalnog sustava. Paralelni sustav sastoji se od više procesora koji komuniciraju preko zajedničke memorije. Distribuirani sustav sastoji se od više procesora koji komuniciraju razmjenom poruka preko komunikacijske mreže, a svaki procesor ima vlastitu memoriju. Procesori u distribuiranom sustavu ne mogu pristupiti memorijama ostalih procesora. Građa paralelnog sustava prikazana je slikom 1.1, a građa distribuiranog sustava slikom 1.2.



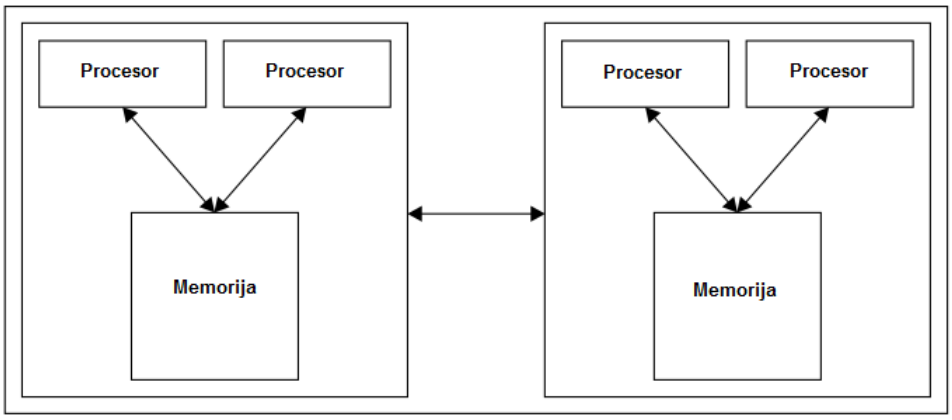
Slika 1.1: Paralelni sustav



Slika 1.2: Distribuirani sustav

Osnovna razlika između ove dvije vrste sustava svodi se na postojanje ili nepostojanje zajedničke memorije. No, ta razlika je samo na logičkoj razini. U fizičkom sustavu u kojem procesori imaju zajedničku memoriju, moguće je simulirati poruke pisanjem u zajedničku memoriju i čitanjem iz nje. U fizičkom sustavu u kojem su procesori povezani u mrežu, moguće je simulirati zajedničku memoriju tako da jedan procesor postane poslužitelj zajedničkih podataka.

Računala koja se danas koriste su ”hibridi” paralelnih i distribuiranih sustava. Komunikacija se vrši preko mreže, kao u distribuiranom sustavu. Međutim, svako računalo ima više procesora i/ili jezgri procesora koji pristupaju zajedničkoj memoriji. Sljedeća slika ilustrira takvu hibridnu arhitekturu koja se dijeli između pojedinih računala:



Slika 1.3: Hibridni sustav

Svaki od ovih sustava ima svoje prednosti i mane. U paralelnom sustavu, dijeljenje podataka kroz različite dretve jedne izvršne datoteke je poprilično brzo, čak i brže od korištenja mreže. No, pri izgradnji programa potrebno je obratiti pozornost da dretve ne "pregaze" jedna drugu ako se promijene neke od varijabli korištenih u programu.

Distribuirani sustav teži tome da bude vrlo skalabilan i jeftin za sastavljanje. Na primjer, ukoliko je potrebno još snage, moguće je dodati još jedan procesor. Još jedna prednost distribuiranog sustava je što svaki procesor vrlo jednostavno može pristupiti svojoj memoriji. Nedostaci ovog sustava uključuju činjenicu da programeri trebaju implementirati vlastitu strategiju za premještanje podataka. Također, nije moguće sve algoritme jednostavno implementirati u distribuirani sustav.

1.2 Amdahlov zakon

Amdahlov zakon kaže da je moguće paralelizirati/distribuirati izračunavanje koliko god je potrebno, pritom poboljšavajući izvedbu dodavajući nove računalne resurse. Međutim, kod ne može biti brži od kombinacije sekvencijalnih (neparalelnih) dijelova na jednom procesoru.

Formulacija je sljedeća. Dani algoritam je djelomično paralelni te djelomično serijski. Sa P označimo paralelni dio, a sa S serijski, takvi da je $S + P = 100\%$. Nadalje, neka je $T(n)$ vrijeme izvođenja algoritma kada koristi n procesora. Imamo sljedeću relaciju:

$$T(n) \geq S * T(1) + \frac{P * T(1)}{n}$$

Dakle, vrijeme izvršavanja algoritma na n procesa je jednako ili veće vremenu izvršavanja serijskog dijela na jednom procesu zajedno s vremenom izvršavanja paralelnog dijela na jednom procesu podijeljenog s ukupnim brojem procesora.

Kako se povećava broj procesora, n, drugi dio u gore navedenoj jednadžbi postaje sve manji i manji, skoro zanemariv. Stoga slijedi

$$T(\infty) \approx S * T(1)$$

Vrijeme izvršavanja algoritma s velikim brojem procesora je približno jednako vremenu izvršavanja serijskog dijela pomoću jednog procesora.

No, zašto je Amdahlov zakon tako bitan? Primjetno je da se vrlo često algoritam ne može paralelizirati. Razlozi su brojni: možda će se morati kopirati podatke i/ili kod na neku lokaciju gdje će im razni procesori moći pristupati. Možda će se morati podijeliti podatke u manje komade i poslati ih dalje putem mreže. Možda će se morati skupljati rezultate svih istodobnih zadataka i izvršiti daljnju obradu na njima, i tako dalje. Bez obzira na razlog, ako nije moguće u potpunosti paralelizirati algoritam, na kraju će vrijeme izvršavanja koda

ovisiti o izvođenju serijskog dijela. Osim toga, već i prije nego se to dogodi, uvidjet će se da je to lošije od očekivane brzine. Algoritmi koji su potpuno paralelni obično se nazivaju neugodno (sramotno) paralelni algoritmi.

Primjer 1.2.1. Pretpostavlja se da se algoritam vrti 100 sekundi na jednom procesoru. Također, pretpostavlja se da je moguće paralelizirati 99% algoritma. Kod se može izvršavati brže, kako se povećava broj procesora, što je i očekivano. Pogledajmo sljedeće izračune:

$$T(1) = 100s$$

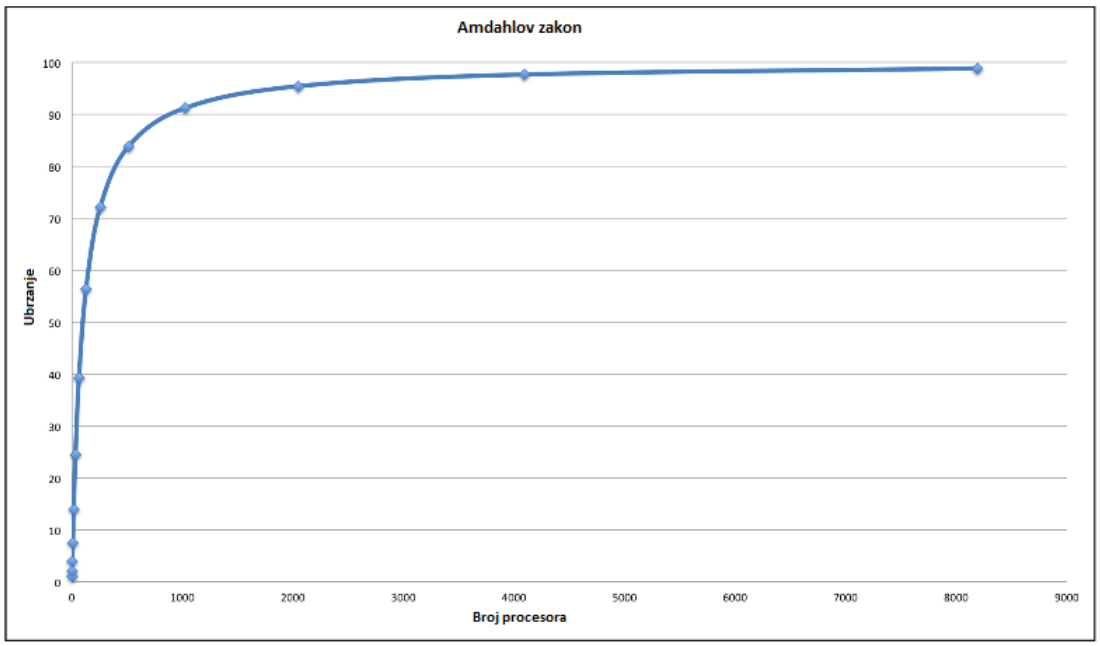
$$T(10) \approx 0,01 * 100s + \frac{0,99 * 100s}{10} = 10,9s \Rightarrow 9,2x \quad kraće \quad vrijeme$$

$$T(100) \approx 1s + 0,99s = 1,99s \Rightarrow 50,2x \quad kraće \quad vrijeme$$

$$T(1000) \approx 1s + 0,099s = 1,099s \Rightarrow 91x \quad kraće \quad vrijeme$$

Vidljivo je da se ubrzanje povećava s povećanjem procesora. Ubrzanje je najveće na početku, kada je postignuto 9,2 puta kraće vrijeme koristeći 10 procesora. Ali, ubrzanje je manje kada se koristi 100 procesora i postiže 50 puta kraće vrijeme te kada se koristi 1000 procesora i postiže 91 puta kraće vrijeme.

Sljedeća slika pokazuje očekivano najbolje ubrzanje za isti algoritam (izračunato koristeći skoro 10 000 procesora). Nije bitno koliko procesora se koristi jer je nemoguće dobiti ubrzanje veće od 100 puta. To znači da će se najbrži kod izvršiti za jednu sekundu, koliko iznosi vrijeme izvršavanja serijskog dijela na pojedinom procesoru, baš kako je Amdahlov zakon predvidio:



Slika 1.4: Predikcija Amdahlovog zakona [1]

Amdahlov zakon ukazuje na dvije stvari: koliko ubrzanje je razumno očekivati u najboljem slučaju te kada je potrebno prestati unaprjeđivati hardver zbog sve manjih koristi.

Zanimljivo je zapažanje da se Amdahlov zakon jednako primjenjuje na distribuiranim i paralelno-distribuiranim sustavima. U tim slučajevima, n se odnosi na ukupan broj procesora među računalima. Još jedan aspekt Amdahlovog zakona je činjenica da sustavi koje koristimo mogu postati jači i da će distribuirani algoritmi trajati (vremenski) sve kraće, ukoliko je moguće iskoristiti dodatne cikluse.

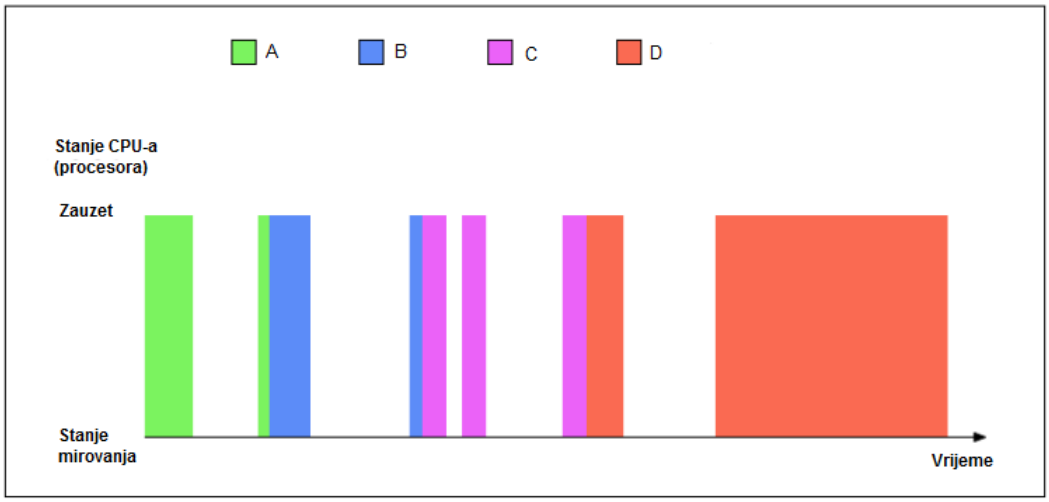
Kada vrijeme izvršavanja aplikacija postane razumno kratko, slijedi rješavanje većih problema. Ovaj aspekt algoritamske evolucije (širenje veličine problema do trena kada je prihvatljiva izvodljivost dosegnuta) zove se **Gustafsonov zakon**.

1.3 Asinkrono programiranje

Poznato je kako se algoritmi i programi mogu strukturirati tako da rade na lokalnom računalu ili na jednom ili više računala na mreži. Čak i kada se kod pokreće na lokalnom računalu, moguće je koristiti više dretvi i/ili više procesa tako da varijabilni (pojedini, različiti) dijelovi mogu raditi istovremeno na više različitih CPU-a (procesora).

Takav stil programiranja naziva se **asinkrono programiranje**, koji u specifičnim slučajevima može dovesti do impresivnih performansi. Posebno će se promotriti izvršavanje pojedinog procesa/dretve.

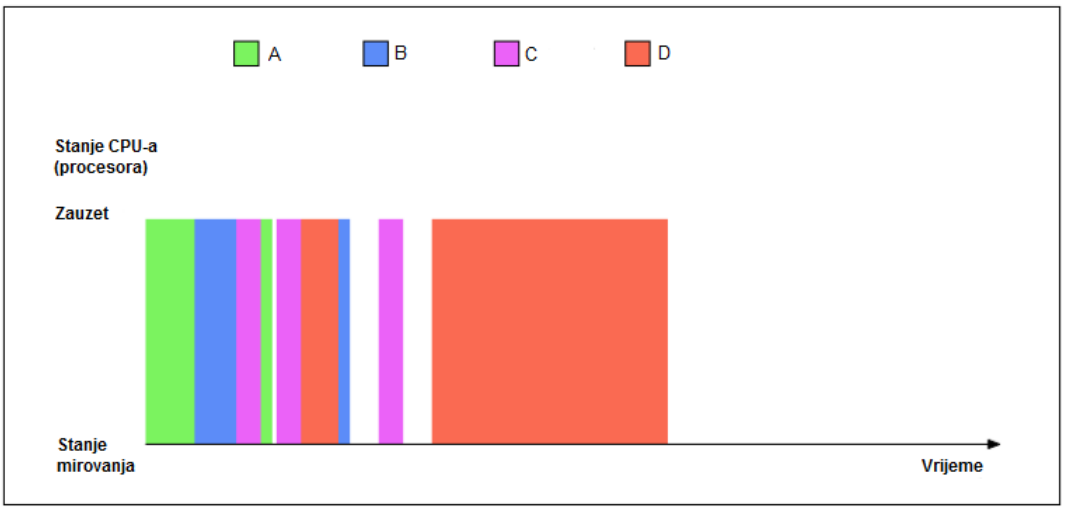
Pogledajmo kako ovi zadatci koriste CPU u tipičnom, općenitom primjeru gdje je dio softvera sastavljen od četiri zadatka: A, B, C i D (trenutno nije bitno što ovi zadatci izvršavaju). Pretpostavka je da svaki od ova 4 zadatka izvršava neke izračune te svaki ima ulazne i izlazne podatke. Najintuitivnije je da se zadatci pozivaju sekvencijalno. Iduća slika pokazuje iskorištavanje CPU-a jednostavne aplikacije s četiri zadatka:



Slika 1.5: Korištenje CPU-a [1]

Dok svaki zadatak izvršava vlastite operacije ulaznih i izlaznih podataka, CPU je u stanju mirovanja i čeka da zadatak ponovno pokrene izračunavanje. Time se CPU ostavlja neiskorištenim relativno veliku količinu vremena. Ključ razmatranja ovdje je velika razlika u brzini pri kojoj se razmjenjuju podatci iz različitih komponenti, kao što su diskovi, RAM i mreža u CPU-u. Posljedica ove ogromne razlike je sljedeća: bilo koji kod koji radi sa značajnim ulaznim i izlaznim podacima ima rizik zadržavanja CPU-a neiskorištenim velik dio vremena izvršavanja (kao što je prikazano na prethodnoj slici). Idealno je organizirati zadatke na način da jedan dio čeka ulazne ili izlazne podatke, (što se naziva *blokiranje*) i tada je suspendiran, a drugi dio zadatka preuzme CPU. O tome se i radi u asinkronom programiranju.

Iduća slika prikazuje reorganizaciju koncepta četiri zadatka koristeći **asinkrono programiranje**:



Slika 1.6: Korištenje CPU-a u asinkronom programiranju [1]

I ovdje su zadaci pozivani sekvencijalno, ali umjesto blokiranja svakog rezerviranog zadatka, oni odustaju od CPU-a kada ga ne trebaju. Iako se CPU i dalje nađe u stanju mirovanja, ukupno vrijeme izvršavanja programa je sada znatno brže. Iako je očito, vrijedi napomenuti da višedretveno programiranje dozvoljava istu učinkovitost izvršavanja zadataka paralelno u različitim dretvama. Međutim, postoji razlika: kada se koristi višedretvenost, operacijski sustav odlučuje koje dretve su aktivne i kada su smijenjene. U asinkronom programiranju, svaki zadatak može odlučiti kada odustati od CPU-a i tako obustaviti njegovo izvršavanje. Osim toga, asinkrono programiranje ne dostiže pravu istovremenost izvršavanja zadataka, tj. i dalje postoji zadatak koji radi u bilo kojem trenutku, što uklanja većinu uvjeta izvođenja koda. Kao rješenje, moguće je miješati paradigme i koristiti dretve i/ili procese s asinkronim tehnikama u sklopu pojedine dretve/procesa. Bitno je napomenuti da asinkrono programiranje dolazi do izražaja kada se bavi ulaznim i izlaznim podacima, a ne s procesorski intenzivnim zadacima.

U Pythonu se koriste **korutine** koje su u mogućnosti zaustaviti izvršavanje funkcije u određenom trenutku, kao što će se vidjeti u nastavku ovog potpoglavlja. Za razumijevanje korutina, potrebno je razumjeti generatore, a za to treba shvatiti iteratore.

Većina programera je upoznata s konceptom iteriranja kroz neku vrstu kolekcija (npr. stringovi, liste, itd.)

```
>>> for i in range(3):
...     print(i)
...
0
1
2
```

Slika 1.7: Primjer iteriranja [1]

Razlog zašto je moguće iterirati kroz sve vrste objekata, a ne samo kroz liste ili stringove je **iteracijski protokol (protokol za iteraciju)**. Iteracijski protokol definira standardno sučelje za iteraciju: objekt koji implementira `__iter__` i `__next__` je iterator i, kao što samo ime sugerira, može se iznova iterirati, kao što je moguće vidjeti na idućem dijelu koda:

```
class MyIterator(object):

        def __init__(self, xs):
                self.xs = xs

        def __iter__(self):
                return self

        def __next__(self):
                if self.xs:
                        return self.xs.pop(0)
                else:
                        raise StopIteration

for i in MyIterator([0, 1, 2]):
        print(i)
```

Pokretanje ovog koda daje sljedeći izlaz:

```
0
1
2
```

Za prikazivanje funkcionalnosti protokola, potrebno je odmotati petlju, što prikazuje sljedeći dio koda:

```
itrtr = MyIterator([3, 4, 5, 6])
print(next(itrtr))
print(next(itrtr))
```

```
print(next(itrtr))
print(next(itrtr))
print(next(itrtr))
```

Izlaz je sljedeći:

```
3
4
5
6
Traceback (most recent call last):
File "iteration.py", line 1, in <module>
import MyIterator
File "/home/martina/Desktop/diplomski/MyIterator.py", line 23,
in <module>
print(next(itrtr))
File "/home/martina/Desktop/diplomski/MyIterator.py", line 13,
in __next__
raise StopIteration
StopIteration
```

Instancira se `MyIterator` te se redom, kako dobiva vrijednosti, poziva metoda `next()` više puta. Kada sekvenca dođe do kraja, poziv `next()` izbaci iznimku `StopIteration`. Slično je i s petljama u Pythonu: poziva se `next()` na iteratoru i hvata se iznimka `StopIteration`.
Generator se poziva da generira niz rezultata, umjesto da se vraćaju rezultati. To se postiže tako da se umjesto `return` stavlja `yield` kao u idućem kodu:

```
def mygenerator(n):
        while n:
                n -= 1
                yield n

if __name__ == '__main__':
        for i in mygenerator(3):
                print(i)
```

Kada se ova naredba izvrši, dobije se sljedeći izlaz:

```
2
1
0
```

Primjetno je da `yield` čini `mygenerator` generatorom, a ne jednostavnom funkcijom. Zanimljivo ponašanje prethodnog koda je da pozivanje funkcije `generator` ne

pokreće generaciju niza, već samo stvara objekt:

```
>>> from generators import mygenerator
>>> mygenerator(5)
<generator object mygenerator at 0x7f5478f40f68>
```

Da bi se aktivirao `generator`, potrebno je pozvati metodu `next()` na njega:

```
>>> g=mygenerator(2)
>>> next(g)
1
>>> next(g)
0
>>> next(g)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
StopIteration
```

Svaki poziv metode `next()` stvara vrijednost iz generirane sekvence sve dok sekvenca ne bude prazna, a to je kada se dobije iznimka `StopIteration`. Takvo ponašanje je primjetno i kada se koriste iteratori. U osnovi, generatori su jednostavan način za zapisivanje iteratora, bez potrebe za definiranjem klasa s njihovim metodama `__iter__` i `__next__`. Potrebno je napomenuti da su generatori jednokratne operacije, tj. nije moguće iterirati kroz generiranu sekvencu više od jednom. Ukoliko je potrebno više iteracija, ponovno se pozove funkciju `generator`.

Izraz `yield`, korišten u funkciji `generator` za stvaranje niza vrijednosti, može se koristiti na desnoj strani zadatka za proizvodnju vrijednosti.

To dopušta stvaranje korutina. Korutina je jednostavna vrsta funkcije koja može obustaviti i nastaviti izvršavanje programa na dobro definiranim mjestima u kodu (preko `yield` izraza). Treba imati na umu da, iako su korutine implementirane kao poboljšani generatori, one konceptualno nisu generatori. Razlog tome je što korutine nisu povezane s iteracijama. Još jedna razlika je u tome što generatori proizvode vrijednosti, a korutine ih troše.

Kreirajmo neke korutine i pogledajmo kako ih možemo koristiti. Imamo tri glavna konstruktora u korutinama:

- `yield()`: koristi se za zaustavljanje izvršenja korutina
- `send()`: koristi se za prosljeđivanje podataka korutini
- `close()`: koristi se za prekidanje rada korutine

Idući kod pokazuje kako se konstruktori mogu koristiti u jednostavnoj korutini:

Listing 1.1: coroutines.py [1]

```
def complain_about(substring):
        print('Please talk to me!')
        try:
                while True:
                        text = (yield)
                        if substring in text:
                                print('Oh no: I found a %s again!'
                                                % (substring))
        except GeneratorExit:
                print('Ok, ok: I am quitting.')
```

Vidljivo je da korutina ima samo jednu funkciju koja ima jedan argument, i to string. Nakon ispisivanja poruke ulazi u beskonačnu petlju ograđenu `try except` blokom. Dakle, iz petlje je moguće izaći samo preko iznimke. Primjetno je da se koristi iznimka `GeneratorExit`. Kada se ta iznimka uhvati, očisti se memorija i program završi. Korutina radi na sljedeći način [1]:

```
>>> from coroutines import complain_about
>>> c = complain_about('Ruby')
>>> next(c)
Please talk to me!
>>> c.send('Test data')
>>> c.send('Some more random text')
>>> c.send('Test data with Ruby somewhere in it')
Oh no:  I found a Ruby again!
>>> c.send('Stop complaining about Ruby or else!')
Oh no:  I found a Ruby again!
>>> c.close()
Ok, ok:  I am quitting.
```

Izvršavanje `complain_about('Ruby')` stvara korutinu. Za korištenje novokreirane korutine, na nju je potrebno pozvati metodu `next()` kao i kod generatora. Primjetno je da se tek nakon pozivanja metode `next()` na ekran ispisuje poruka `Please talk to me!`

U ovom trenutku, korutina je došla do linije koda `text = (yield)`, gdje se obustavlja izvršavanje. Kontrola ide nazad do interpretera, stoga je moguće poslati podatke samoj korutini. Ovo se izvršava pomoću metode `send()`:

```
>>> c.send('Test data')
```

```
>>> c.send('Some more random text')
>>> c.send('Test data with Ruby somewhere in it')
Oh no:  I found a Ruby again!
```

Svaki poziv metode `send()` pomiče kod do sljedećeg `yield`-a. U ovom slučaju, na sljedeću iteraciju `while` petlje i nazad do `text = (yield)`. U ovom trenutku kontrola ide nazad do interpretera.

Korutinu je moguće zaustaviti pozivom metode `close()`, što rezultira bacanjem iznimke `GeneratorExit`. Korutini jedino preostaje uhvatiti tu iznimku, očistiti i izaći:

```
>>> c.close()
Ok, ok:  I am quitting.
```

Zakomentiranjem `try...except` bloka, nije moguće dobiti iznimku `GeneratorExit`, no korutina će opet stati [1]:

Listing 1.2: coroutines2.py [1]

```
@coroutine
def complain_about2(substring):
        print('Please talk to me!')
        while True:
                text = (yield)
                if substring in text:
                        print('Oh no: I found a %s again!'
                                % (substring))
```

```
>>> from coroutines2 import complain_about2
>>> c = complain_about2('Ruby')
>>> next(c)
Please talk to me!
>>> c.close()
>>> c.send('This will crash')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
StopIteration
>>> next(c)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
StopIteration
```

Primjetno je da zatvaranjem korutine objekt ostaje, ali nije koristan. Nije mu moguće slati podatke ili ga koristiti pozivanjem metode `next()`.

Dosta programera ne koristi metodu `next()`, već radije koristi dekorator, da se izbjegnu dodatni pozivi:

Listing 1.3: coroutines3.py [1]

```
def coroutine(fn):
        def wrapper(*args, **kwargs):
                c = fn(*args, **kwargs)
                next(c)
                return c
        return wrapper

@coroutine
def complain_about2(substring):
        print('Please talk to me!')
        while True:
                text = (yield)
                if substring in text:
                        print('Oh no: I found a %s again!'
                                % (substring))
```

```
>>> from coroutines3 import complain_about2
>>> c = complain_about2('JavaScript')
Please talk to me!
>>> c.send('Test data with JavaScript somewhere in it')
Oh no:  I found a JavaScript again!
>>> c.close()
```

Korutine mogu biti organizirane u kompleksnim hijerarhijama, gdje jedna korutina šalje i prima podatke od više drugih korutina. Korisne su u mrežnom i sistemskom programiranju, gdje se mogu vrlo učinkovito iskoristiti za reimplementaciju većine Unix alata u Pythonu.

Primjer 1.3.1 (Asinkroni primjer)**.** Pogledajmo jednostavan primjer asinkronog programiranja. Radi se u Linux okruženju, jer se koristi `grep` naredba za postizanje rezultata. Uzmimo neku online knjigu u .txt formatu. Recimo, The Project Gutenberg EBook of Pride and Prejudice, by Jane Austen:
`http://www.gutenberg.org/files/1342/1342-0.txt`. Nju lako možemo skinuti:

```
$ curl -sO http://www.gutenberg.org/files/1342/1342-0.txt
$ wc 1342-0.txt
```

Sada se koristi naredba `grep` za dobivanje točnog broja riječi *Elizabeth* u skinutom tekstu:

```
$ time (grep -io Elizabeth 1342-0.txt | wc -l)
635

real 0m0.011s
user 0m0.004s
sys 0m0.004s
```

Napravimo istu stvar u Pythonu, koristeći korutine:

Listing 1.4: grep.py [1]

```
def coroutine(fn):
        def wrapper(*args, **kwargs):
                c = fn(*args, **kwargs)
                next(c)
                return c
        return wrapper

def cat(f, case_insensitive, child):
        if case_insensitive:
                line_processor = lambda l: l.lower()
        else:
                line_processor = lambda l: l

        for line in f:
                child.send(line_processor(line))


@coroutine
def grep(substring, case_insensitive, child):
        if case_insensitive:
                substring = substring.lower()
        while True:
                text = (yield)
                child.send(text.count(substring))


@coroutine
def count(substring):
        n = 0
        try:
                while True:
                        n += (yield)
        except GeneratorExit:
                print(substring, n)

if __name__ == '__main__':
```

```
        import argparse

        parser = argparse.ArgumentParser()
        parser.add_argument('-i', action='store_true',
                                                dest='case_insensitive')

        parser.add_argument('pattern', type=str)
        parser.add_argument('infile', type=argparse.FileType('r'))

        args = parser.parse_args()


        cat(args.infile, args.case_insensitive,
                grep(args.pattern, args.case_insensitive,
                        count(args.pattern)))
```

Pogledajmo koje rezultate ćemo dobiti:

```
$ time python3.5 grep.py -i Elizabeth 1342-0.txt
Elizabeth 635

real 0m0.112s
user 0m0.096s
sys 0m0.016s
```

Vidljivo je da je kod u Pythonu koji koristi korutinu kompetitivan sa Unix-ovom `grep` naredbom. Naravno, `grep` naredba je značajno bolja od koda u Pythonu, no rezultat koji smo dobili je poprilično impresivan.

U kodu se kreće od reimplementacije korutine dekoratora kojeg smo vidjeli ranije. Nakon toga, problem se rastavlja na tri zasebna koraka:

- čitanje teksta red po red (`cat` funkcija)
- brojanje učestalosti `substring`-a u svakom retku (`grep` korutina)
- zbrajanje svih brojeva u ukupnu sumu (`count` korutina)

U `main`-u se rasčlanjuju opcije komandne linije, izlaz funkcije `cat` se prosljeđuje korutini `grep`, a izlaz korutine `grep` se prosljeđuje korutini `count`, kao što rade i Unix alati. Ovo ulančavanje je vrlo jednostavno. Korutina koja primi podatke prosljeđuje se kao argument prema funkciji ili korutini koja stvara podatke. Zatim se unutar te strukture podataka pozove korutina `send`. Naredba `grep` je prva korutina. U njoj se ulazi u beskonačnu petlju gdje se nastavlja primanje podataka `(text = (yield))`, brojanje koliko puta smo

našli na zadani `substring` u tekstu i slanje tog broja idućoj korutini (u našem slučaju `count`): `child.send(text.count(substring))`. Korutina `count` sadrži trenutni broj, `n`, od brojeva koje prima od `grep (n += (yield))`. Zatim hvata iznimku `GeneratorExit` poslanu svakoj korutini, koje su zatvorene, da znaju kada treba ispisati traženi substring i `n`.

Zanimljivo je korutine organizirati u kompleksnije grafove. Recimo, ako se želi naći točan broj više riječi u zadanom tekstu. Ovaj kod pokazuje jedan od načina kako je jedna korutina zadužena za emitiranje svog unosa na proizvoljan broj drugih korutina (svoju djecu):

Listing 1.5: mgrep.py [1]

```
def coroutine(fn):
        def wrapper(*args, **kwargs):
                c = fn(*args, **kwargs)
                next(c)
                return c
        return wrapper

def cat(f, case_insensitive, child):
        if case_insensitive:
                line_processor = lambda l: l.lower()
        else:
                line_processor = lambda l: l

        for line in f:
                child.send(line_processor(line))

@coroutine
def grep(substring, case_insensitive, child):
        if case_insensitive:
                substring = substring.lower()
        while True:
                text = (yield)
                child.send(text.count(substring))

@coroutine
def count(substring):
        n = 0
        try:
                while True:
                        n += (yield)
        except GeneratorExit:
                print(substring, n)

@coroutine
def fanout(children):
```

```
        while True:
                data = (yield)
                for child in children:
                        child.send(data)

if __name__ == '__main__':
        import argparse

        parser = argparse.ArgumentParser()
        parser.add_argument('-i', action='store_true',
                                                dest='case_insensitive')
        parser.add_argument('patterns', type=str, nargs='+', )
        parser.add_argument('infile', type=argparse.FileType('r'))

        args = parser.parse_args()
        cat(args.infile, args.case_insensitive,
                fanout([grep(p, args.case_insensitive,
                                        count(p)) for p in args.
                                           patterns]))
```

Kod je jako sličan prethodnom, no postoje razlike. Primjetno je da je definiran prijenosnik: `fanout`. On uzima listu korutina kao ulaz te ulazi u beskonačnu petlju i čeka podatke. Nakon što primi te podatke (`data = (yield)`), pošalje ih ostalim korutinama (`for child in children: child.send(data)`). Ostalo je isto kao u prošlom kodu. Vidljivo je da su performanse dobre, čak i kada se šalje neki niz riječi:

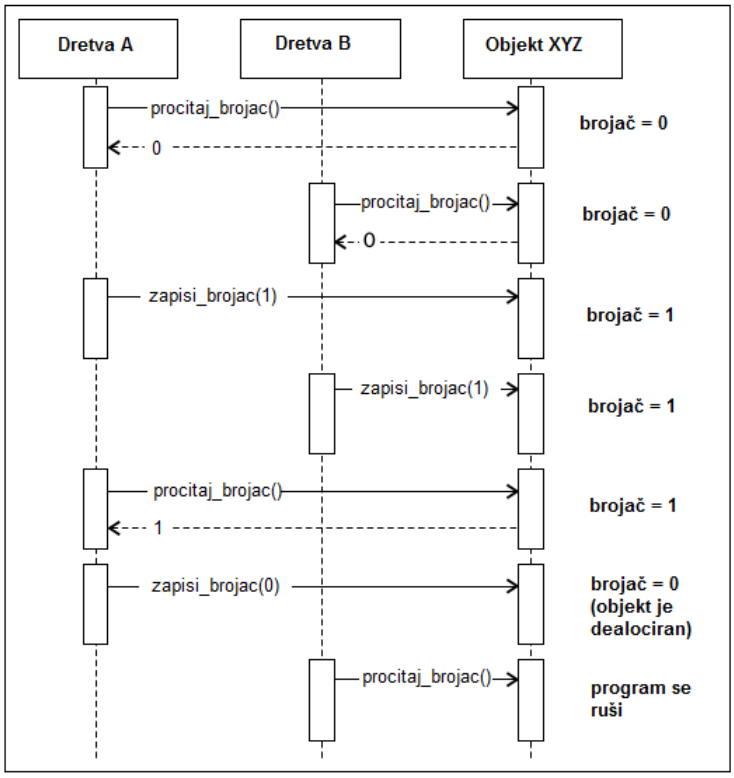
```
$ time python3.5 mgrep.py -i Elizabeth Darcy Jane 1342-0.txt
Elizabeth 635
Darcy 418
Jane 295

real 0m0.290s
user 0m0.252s
sys 0m0.020s
```

Python ima podršku za asinkrono programiranje još od verzije 1.5.2, kada su uvedeni moduli `asynchat` i `asyncore`. Verzija 2.5 uvodi mogućnost slanja podataka korutinama preko izraza `yield`, što omogućava jednostavnije pisanje asinkronog koda. U verziji 3.4 uvedena je biblioteka **asyncio**, a u verziji 3.5 su predstavljene prave korutine preko `async def` i `await`.

1.4 Višedretvenost

Python već dugo nudi podršku za dretve (još od verzije 1.4). Također, nudi robusno sučelje na visokoj razini kao **POSIX** na Linux-u, za dretve. Na jednoprocesorskim sustavima korištenje više dretvi ne bi dalo pravo konkurentno izvođenje, budući da se samo jedna dretva može izvršiti u bilo kojem trenutku. Samo na višeprocesorskom sustavu dretve se mogu izvršavati paralelno.

S obzirom da su ovakve dretve *pokreni i zaboravi*, mogu se napraviti da su pozadninski procesi (eng. daemons), što znači da ih glavni Python program neće čekati da prestanu s radom prije izlaska. Glavna poteškoća u korištenju dretvi za izvršavanje paralelnih akcija je da se ne može utvrditi kada će određena dretva čitati ili napisati bilo koji podatak koji je dostupan i drugim dretvama. To može dovesti do onoga što se naziva **uvjeti izvođenja**. To je situacija gdje, s jedne strane točna izvedba sustava ovisi o akcijama koje su izvršene u zadanom redoslijedu, a s druge strane, nije garantirano da će se izvesti u tom redoslijedu, već u redoslijedu kako je zamislio programer. Korutine imaju veliku prednost izbjegavanja uvjeta izvođenja. Jedan primjer uvjeta izvođenja može se vidjeti u algoritmima za brojanje referenci. Interpreter koji sakuplja ostatke (eng. garbage-collected), kao **CPython** radi ovako: svaki objekt ima brojač i bilježi koliko referenci za taj objekt trenutno postoji. Svaki put kada je neki objekt kreiran, odgovarajuća referenca brojača je povećana za 1. Svaki put kada je referenca izbrisana, brojač se smanji za 1. Kada brojač dođe do 0, objekt je dealociran. Pokušaj da se iskoristi dealocirani objekt rezultira greškom (segmentation fault). To znači da je potrebno nekako provesti strogi redoslijed povećanja i smanjenja referentnog brojača. Zamislimo dvije dretve kako dobivaju referencu objekta i nekoliko trenutaka kasnije ju brišu. Ako obje dretve pristupe referentnom brojaču u isto vrijeme, mogu pregaziti njezinu vrijednost:



Slika 1.8: Višedretvenost

Jedan način rješavanja ovakvih problema je korištenje lokota. Thread-safe redovi (`Queue` instanca iz Pythonovog modula `Queue`) su pogodni kao lokoti koje možemo iskoristiti za pristup podacima.

Budući da svaka dretva piše isti izlaz u red, moguće je pratiti taj red kako bi se znalo kada su rezultati spremni i kada je vrijeme za prekid rada. No, svakako treba imati na umu da korištenje lokota za pristupanje podacima i izbjegavanje uvjeta izvođenja može biti skupo, ovisno o aplikaciji.

Pogledajmo sljedeći kod:

Listing 1.6: fib.py [1]

```
from threading import Thread

def fib(n):
        if n <= 2:
                return 1
        elif n == 0:
                return 0
```

```
        elif n < 0:
                raise Exception('fib(n) is undefined for n < 0')
        return fib(n - 1) + fib(n - 2)

if __name__ == '__main__':
        import argparse
        parser = argparse.ArgumentParser()
        parser.add_argument('-n', type=int, default=1)
        parser.add_argument('number', type=int, nargs='?',
                        default=34)
        args = parser.parse_args()

        assert args.n >= 1, 'The number of threads has to be > 1'
        for i in range(args.n):
                t = Thread(target=fib, args=(args.number,))
                t.start()
```

Nakon što se pokrene ovaj kod, traži se korisnika da odredi broj dretvi, te nakon toga da svaka dretva izračuna sumu prvih 30 Fibonaccijevih brojeva (`args.number`). Ovdje nije bitna veličina broja, već performanse:

```
$ time python3.5 fib.py -n 1 30

real 0m1.694s
user 0m1.176s
sys 0m0.016s

$ time python3.5 fib.py -n 2 30

real 0m3.644s
user 0m2.296s
sys 0m0.060s

$ time python3.5 fib.py -n 3 30

real 0m5.878s
user 0m3.408s
sys 0m0.048s

$ time python3.5 fib.py -n 4 30

real 0m7.608s
```

```
user 0m4.552s
sys 0m0.064s
```

Primjetno je sljedeće: što se više dretvi koristi, potrebno je više vremena za izračunavanje sume prvih 30 Fibonaccijevih brojeva. Povećanje broja dretvi (paralelnih izračuna) povećava linearno vrijeme izvršavanja. To se ne čini u redu jer se očekuje da dretve rade paralelno. Takav problem naziva se **Global Interpreter Lock (GIL)**. GIL je globalni lokot koji se koristi uglavnom da referenca ostane očuvana.

Slična situacija se događa i kod korutina. Jedan dio koda može raditi u bilo kojem trenutku. Moguće je dobiti paralelizam koji se očekuje, a to je kada korutina ili dretva čeka ulazne ili izlazne podatke, a druga korutina ili dretva preuzme CPU. Ali, ovo ne radi dobro ukoliko neki zadatak treba CPU na duže vrijeme, kao što je slučaj u primjeru dobivanja sume prvih 30 Fibonaccijevih brojeva.

Paralelni rad na ulaznim i izlaznim podacima doprinosi značajnom poboljšanju aplikacije, bilo da koristimo dretve ili korutine. GUI aplikacije imaju velike prednosti korištenjem dretvi. Jedna može rukovati ažuriranjima, a druga radi u pozadini bez da se "zaledi" korisničko sučelje. Treba biti svjestan učinaka GIL-a u standardnom Python interpreteru i planirati u skladu s tim. Postoji i Python interpreter bez GIL-a, npr. **Jython** [9].

1.5 Višestruki procesi

Način na koji su programeri zaobilazili GIL i njegove utjecaje na procesorski ograđene dretve dovelo je do toga da se koristi više procesa umjesto više dretvi. Ovaj pristup ima svoje mane koje su svedene na to da postoji više instanci Python interpretera sa svim vremenima pokretanja i velikim potrošnjama memorije koje zahtijeva. Ali višestruki procesi imaju svoje prednosti. Imaju vlastiti memorijski prostor i implementiraju *nedjeljivu arhitekturu* (arhitektura u kojoj nema zajedničke memorije), što potiče na razmišljanje o raznim oblicima za pristupanje podacima. To olakšava tranziciju s arhitekture jednog računala na distribuiranu aplikaciju, gdje se ionako koristi više procesa (na različitim strojevima).

Postoje dva modula u standardnoj Pythonovoj biblioteci za implementiranje paralelizma zasnovanog na procesima. Jedan je **multiprocessing**, a drugi je `concurrent.futures`. Modul `concurrent.futures` je izgrađen na temelju višestrukih procesa i modula za dretve (`threading`) te pruža snažno sučelje. Pogledajmo ponovno primjer za Fibonaccijeve brojeve, no sada se, umjesto višedretvenosti, koriste višestruki procesi:

Listing 1.7: mpfib.py [1]

```
import concurrent.futures as cf

def fib(n):
        if n <= 2:
```

```
                return 1
        elif n == 0:
                return 0
        elif n < 0:
                raise Exception('fib(n) is undefined for n < 0')
        return fib(n - 1) + fib(n - 2)

if __name__ == '__main__':
        import argparse
        parser = argparse.ArgumentParser()
        parser.add_argument('-n', type=int, default=1)
        parser.add_argument('number', type=int, nargs='?',
                        default=34)
        args = parser.parse_args()

        assert args.n >= 1, 'The number of threads has to be > 1'
        with cf.ProcessPoolExecutor(max_workers=args.n) as pool:
                results = pool.map(fib, [args.number] * args.n)
```

U odnosu na višedretvenost, nakon što se prime parametri komandne linije, kreira se instanca `ProcessPoolExecutor` i pozove se metoda `map()` da izvrši izračunavanja paralelno. Intuitivno, stvorena je skupina radnih procesa `args.n` i ta se skupina koristi za izvođenje funkcije `fib` na svakom elementu ulazne liste.

```
$ time python3.5 mpfib.py -n 1 34

real 0m2.729s
user 0m2.752s
sys 0m0.016s

$ time python3.5 mpfib.py -n 2 34

real 0m3.347s
user 0m6.516s
sys 0m0.020s

$ time python3.5 mfib.py -n 3 34

real 0m5.809s
user 0m26.976s
sys 0m0.044s

$ time python3.5 mfib.py -n 4 30
```

```
real 0m7.824s
user 0m26.976s
sys 0m0.024s
```

Vidljivo je da je moguće izvršiti više od jednog izračunavanja paralelno do trenutka gdje vremena izvršavanja za `args.n` između 1 i 4 ostaju ista. Pokretanje većeg broja procesa od onih koje nude hardverske jezgre predstavlja značajnu degradaciju performansi:

```
$ time python3.5 mpfib.py -n 8 34

real 0m17.273s
user 1m4.396s
sys 0m0.088s

$ time python3.5 mpfib.py -n 16 34

real 1m24.235s
user 3m21.522s
sys 0m0.116s
```

Pogledajmo zadnje dvije linije koda. Koristi se klasa `ProcessPoolExecutor` iz modula `concurrent.futures`. To je jedna od dviju glavnih klasa u tom modulu. Druga je `ThreadPoolExecutor` koja stvara skup dretvi, umjesto skupa procesa. Obje klase imaju isti API, od kojih izdvajamo tri glavne metode:

- `submit(f, *args, **kwargs)`: koristi se za dodjeljivanje asinkronog poziva `f(*args, **kwargs)` i vraća `Future` instancu
- `map(f, *arglist, timeout=None, chunksize=1)`: ova metoda ekvivalentna je ugrađenoj `map(f, *arglist)`. Ona vraća listu objekata tipa `Future`
- `shutdown(wait=True)`: služi za oslobađanje resursa pomoću `Executor` objekta čim su sve predviđene funkcije gotove. Korištenje `Executor` objekta nakon poziva na ovu metodu javlja iznimku `RuntimeError`

Instanca `Future` čuva mjesto za rezultat asinkronog poziva. Moguće je provjeriti radi li poziv još uvijek, je li uhvaćena iznimka, itd. Poziva se metoda `result()` za pristupanje vrijednosti kada je spremna.

Pogledajmo kako radi klasa `Future` [1]:

```
>>> from mpfib import fib
>>> from concurrent.futures import ProcessPoolExecutor
>>> pool = ProcessPoolExecutor(max_workers=1)
>>> fut = pool.submit(fib, 38)
>>> fut
<Future at 0x101b74128 state=running>
>>> fut.running()
True
>>> fut.done()
False
>>> fut.result(timeout=0)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "/Library/Frameworks/Python.framework/Versions/3.5/lib/
python3.5/concurrent/futures/_base.py", line 407, in result
raise TimeoutError()
concurrent.futures._base.TimeoutError
>>> fut.result(timeout=None)
39088169
>>> fut
<Future at 0x101b74128 state=finished returned int>
>>> fut.done()
True
>>> fut.running()
False
>>> fut.cancelled()
False
>>> fut.exception()
>>>
```

Primjetno je kako se koristi modul `concurrent.futures` za stvaranje skupine radnih procesa i za predaju rada na njima `(pool.submit(fib, 38))`. Metoda `submit` vraća `Future` objekt (`fut`), koji čuva mjesto za rezultat koji još nije dostupan. Pomoću metoda `running(), done() i cancelled()` moguće je provjeriti u kojemu stanju je `fut`. Ukoliko se zatraži rezultat prije nego je spreman (`fut.result(timeout=0)`), dobiva se iznimka `TimeoutError`. To znači da se, ili treba čekati da `Future` objekt bude spreman, ili pitati za rezultat bez vremenskog ograničenja (`fut.result(timeout=None)`), koji blokira dok `Future` objekt nije spreman. Budući da kod radi bez greške, `fut.exception()` vraća `None`.

1.6 Višeprocesorski redovi

Kada se koristi više procesa, javlja se problem kako razmjenjivati podatke između radnih procesa. Modul `multiprocessing` nudi mehanizam da se to riješi koristeći redove. Dakle, radi se o višeprocesnim redovima.

Klasa `multiprocessing.Queue` je nastala po uzoru na klasu `queue.Queue`. U idućem kodu vidljivo je kako se koriste redovi:

Listing 1.8: queues.py [1]

```
import multiprocessing as mp

def fib(n):
        if n <= 2:
                return 1
        elif n == 0:
                return 0
        elif n < 0:
                raise Exception('fib(n) is undefined for n < 0')
        return fib(n - 1) + fib(n - 2)

def worker(inq, outq):
        while True:
                data = inq.get()
                if data is None:
                        return
                fn, arg = data
                outq.put(fn(arg))

if __name__ == '__main__':
        import argparse

        parser = argparse.ArgumentParser()
        parser.add_argument('-n', type=int, default=1)

        parser.add_argument('number', type=int, nargs='?', default=34)
        args = parser.parse_args()

        assert args.n >= 1, 'The number of threads has to be > 1'

        tasks = mp.Queue()
        results = mp.Queue()
        for i in range(args.n):
                tasks.put((fib, args.number))

        for i in range(args.n):
                mp.Process(target=worker, args=(tasks, results)).start()
```

```
        for i in range(args.n):
                print(results.get())

        for i in range(args.n):
                tasks.put(None)
```

I dalje se koristi primjer koji računa sumu Fibonaccijevih brojeva. U ovom slučaju, koristi se arhitektura dvostrukog reda. Jedan red služi za izvršavanje zadatka, a drugi za održavanje rezultata. Kao i prije, koristi se indicirana vrijednost (`None`) u redu da signalizira kada radni proces treba stati. Radni proces je tipa `multiprocessing`. Pogledajmo performanse ovog programa:

```
$ time python3.5 queues.py -n 1 34
5702887

real 0m2.665s
user 0m2.648s
sys 0m0.016s

$ time python3.5 queues.py -n 4 34
5702887
5702887
5702887
5702887

real 0m6.235s
user 0m21.752s
sys 0m0.044s
```

Primjetno je da dodavanje redova ne stvara veliku degradaciju performansi.

Poglavlje 2

Paketi za izgradnju distribuiranih aplikacija

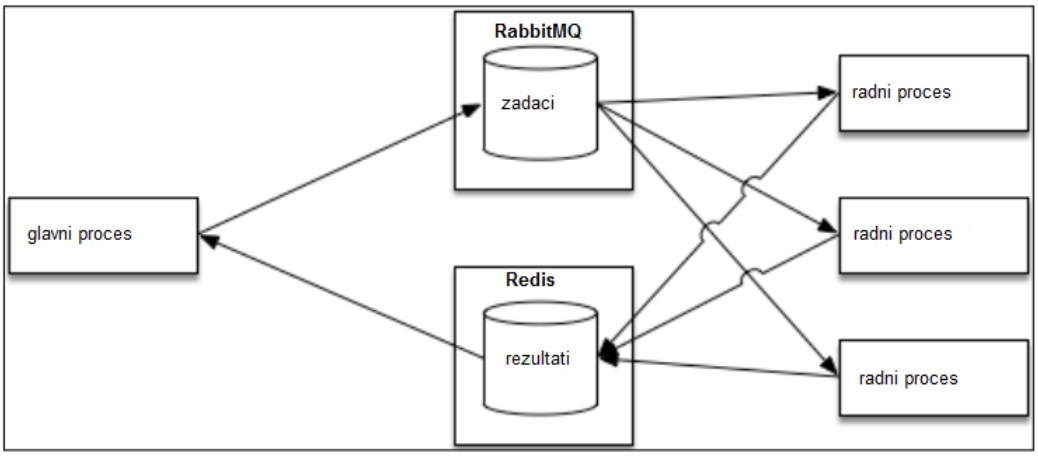
2.1 Celery

U ovom poglavlju detaljnije se razmatra asinkrono programiranje i distribuirano izračunavanje kroz **Celery**[1]. Celery je Pythonov aplikacijski okvir koji se koristi za izgradnju distribuiranih aplikacija. Postoje i alternativni paketi: **Pyro**[6] i **Python-RQ**[5]. Temelje se na arhitekturi "gazda-radnik" sa srednjim slojem koji koristi skup redova za radne zadatke te red ili prostor za pohranu koji sadrži rezultat, a kojeg vrate radni procesi (radnici). Glavni proces (klijent, proizvođač ili gazda) postavlja radne zahtjeve u jedan red i hvata rezultate iz pozadine koje pošalju radni procesi. Radi se o vrlo jednostavnoj i fleksibilnoj arhitekturi. Glavni proces ne treba znati koliko radnih procesa ima, koliko ih je slobodno ili koliko ih obavlja neku zadaću. On samo treba znati gdje se nalaze redovi i kako poslati zahtjev.

Slično se može reći i za radne procese. Oni ne znaju od kuda i od koga dolaze radni zahtjevi te što će se dalje dogoditi s rezultatom kojeg oni generiraju. Trebaju samo znati gdje preuzeti radni zahtjev i gdje spremiti rezultate.

Velika prednost je ta što se broj, tip i morfologija radnih procesa može promijeniti u svakom trenutku bez utjecaja na funkcionalnost sustava. Još jedan aspekt ovako "odvojenog" sustava je taj što, na primjer, "radnici" i "proizvođači" mogu biti napisani u različitim jezicima. Na primjer, Python kod koji generira zadatke koje će izvršavati "radnici" napisani u C-u.

Celery koristi snažne i testirane sustave za redove i pozadinske rezultate. Preporučeni posrednik između glavnog procesa i radnih procesa je **RabbitMQ** [3], a **Redis** [4] koristimo za spremanje rezultata koji se isporučuju nazad glavnom procesu. Iduća slika prikazuje arhitekturu tipične Celery aplikacije koja koristi RabbitMQ i Redis:



Slika 2.1: Celery arhitektura

Svaki proces koji se nalazi na slici u pravokutniku može se nalaziti na različitim računalima. Jednostavnije instalacije uglavnom drže RabbitMQ i Redis na istom poslužiteljskom računalu te imaju jedan ili dva čvora za radnike. Veće instalacije ipak zahtijevaju više računala, a ponekad i više poslužiteljskih računala.

2.2 Uspostavljanje okruženja

Za rad je prvo potrebno uspostaviti okolinu. Budući da se radi o razvoju distribuiranih aplikacija, potrebno je višestrojno okruženje. Ako postoji više računala, ona trebaju imati namještenu mrežu (DNS imena). Ako postoji jedno računalo, tada su rješenje virtualni strojevi na koja se instalira Linux okruženja. Druge opcije su kupnja malih, jeftinih računala kao **Raspberry Pi**, na koja se instalira Linux i koja se spoje na lokalnu mrežu. Treće rješenje može biti spajanje na oblak, npr. na **Amazon EC2** i iskorištavanje nekog od njegovih virtualnih strojeva. U ovom slučaju, treba omogućiti da vatrozid propušta sve mrežne utičnice koje će se stvoriti.

U nastavku se služimo virtualnim strojevima na računalu. Da bi se računalo moglo spajati na virtualne strojeve, u `/etc/hosts` se dodaju IP adrese virtualnih strojeva te njihovi nazivi. Analogno se napravi i na virtualnim strojevima.

```
$ cat /etc/hosts
127.0.0.1 localhost
127.0.1.1 martina-300E4Z-300E5Z-300E7Z
10.10.100.100 ubuntu1
```

```
10.10.100.101 ubuntu2

# The following lines are desirable for IPv6 capable hosts
::1 ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
```

Treba se pobrinuti da IP adrese i nazivi u datotekama `hosts` odgovoraju na strojevima koje odlučimo koristiti.

Kao što je rečeno, koristit će se Celery. Slijede upute kako se instalira. Prvo se podiže virtualno okruženje na svakom računalu (stroju). To se radi pomoću iduće naredbe u terminalu:
`$ pip install virtualenvwrapper`
ili
`$ sudo pip install virtualenvwrapper`, ako prva ne radi.
Sada je potrebno konfigurirati `virtualenvwrapper` tako da se definiraju tri varijable za okruženje:

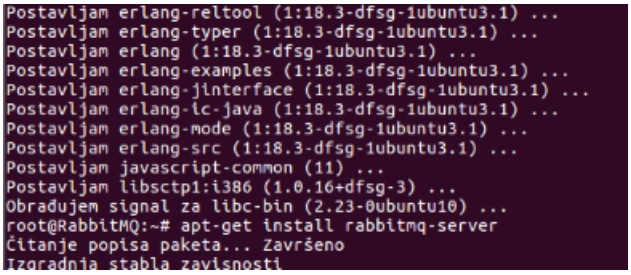
```
$ export WORKON_HOME=$HOME/venvs
$ export PROJECT_HOME=$HOME/workspace
$ source /usr/local/bin/virtualenvwrapper.sh
```

Ovim naredbama definirano je gdje će se nalaziti virtualno okruženje (`$WORKON_HOME`) te ishodišni direktorij (eng. root) (`$PROJECT_HOME`).

Idućom naredbom stvorit će se novo virtualno okruženje (`book`) te će se aktivirati u (`$WORKON_HOME`) direktoriju koristeći Python 3.5.
`$ mkvirtualenv book --python=‘which python3.5‘`
Za svako iduće aktiviranje koristi se komanda `workon`: `$ workon book .`
Naredbom `$ pip install celery` skinemo, otpakiramo i instaliramo Celery na trenutno aktivno virtualno okruženje (u ovom slučaju `book`).

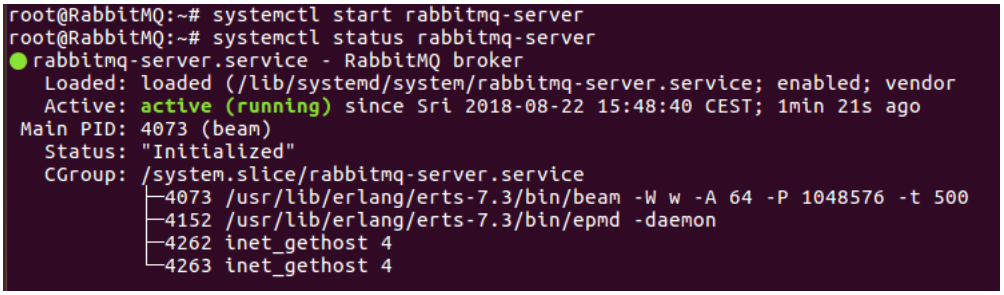
Potrebno je još instalirati i konfigurirati posrednika koji će Celery koristiti za čuvanje radnih redova i dostavljanje poruka radnim procesima. Njega se instalira na jednom virtualnom stroju. Celery podržava brojne ovakve posrednike, ali koristit će se **RabbitMQ**.

Da bi se instalirao RabbitMQ prvo je potrebno instalirati **erlang** [7]. Instaliraju se u terminalu, ali kao root korisnik (naredbom `$ sudo -i` u terminalu možemo raditi kao root korisnik): `$ apt-get install erlang`
te
`$ apt-get install rabbitmq-server`



Slika 2.2: Instalacija RabbitMQ

Nakon što je RabbitMQ instaliran, potrebno je pokrenuti server naredbom
`$ systemctl start rabbitmq-server`
i provjeriti njegov status:
`$ systemctl status rabbitmq-server`



Slika 2.3: Pokretanje i provjera RabbitMQ-a

Pokazat će se još kako se instalira **Redis**, iako neće biti potreban. On se koristi u pozadini za spremanje rezultata koje pošalju radni procesi. Redis bi radio na zasebnom stroju (računalu) kao i RabbitMQ. Instalacija se vrši naredbom
`$ sudo apt-get install redis-server` , a pokreće se sa
`$ sudo redis-server`

Pogledajmo iduću jednostavnu Celery aplikaciju da vidimo kako Celery zapravo radi. U prvom prozoru pokrene se RabbitMQ server, u drugom Redis, ako je instaliran. U trećem prozoru je potrebno aktivirati virtualno okruženje `book` naredbom `workon book` i pozvati `test.py`:

Listing 2.1: test.py

```
import celery
```

```
app = celery.Celery('test',
                broker='amqp://RabbitMQ',
                backend='amqp://RabbitMQ')

@app.task
def echo(message):
        return message
```

Ovaj kod je jednostavan. Uključi se paket `celery` i definira se Celery aplikacija (`app` u kodu) nazvana `test`, dakle istog naziva kao i skripta (ovo je standardna praksa u Celeryu). Aplikacija je konfigurirana da se koristi zadani račun i red za poruke na RabbitMQ kao posredniku. RabbitMQ će se koristiti i za čuvanje rezultata umjesto Redisa. No, ukoliko se želi koristiti Redis, tada se samo zamijeni `backend=amqp://RabbitMQ` s `backend=redis://Redis` (umjesto imena RabbitMQ i Redis mogu se staviti i IP adrese strojeva na kojima se nalaze). U trećem prozoru potrebno je pokrenuti (udaljene) radne procese u terminalu naredbom:

```
$ celery -A test worker --loglevel=info
```

Bitno je pri tom pokretanju nalaziti se u istom direktoriju gdje je skripta `test.py`, tako da Celery može uključiti kod. Naredba `celery` će se pokrenuti i pokrenut će radne procese. Oni će uzeti aplikaciju `app`.

U još jednom terminalu (virtualnom stroju), kopira se skripta `test.py`, aktivira `book` s `workon book` i pokrene Python interpreter u istom direktoriju gdje je i `test.py`:

```
$ python3.5
>>> from test import echo
>>> res=echo('Python rocks!')
>>> print(res)
Python rocks!
```

```
(book) martina@martina-300E4Z-300E5Z-300E7Z ~/venvs/book $ celery -A test worker --loglevel=info
/home/martina/venvs/book/lib/python3.5/site-packages/celery/backends/amqp.py:67: CPendingDeprecationWarning:

    The AMQP result backend is scheduled for deprecation in     version 4.0 and removal in version v5.0.
 Please use RPC backend or a persistent backend.

  alternative='Please use RPC backend or a persistent backend.')

 -------------- celery@martina-300E4Z-300E5Z-300E7Z v4.2.1 (windowlicker)
---- **** -----
--- * ***  * -- Linux-4.4.0-53-generic-x86_64-with-LinuxMint-18.1-serena 2018-09-04 22:37:41
-- * - **** ---
- ** ---------- [config]
- ** ---------- .> app:         test:0x7f9475b6d160
- ** ---------- .> transport:   amqp://guest:**@localhost:5672//
- ** ---------- .> results:     amqp://
- *** --- * --- .> concurrency: 4 (prefork)
-- ******* ---- .> task events: OFF (enable -E to monitor tasks in this worker)
--- ***** -----
 -------------- [queues]
                .> celery           exchange=celery(direct) key=celery


[tasks]
  . test.echo

[2018-09-04 22:37:41,899: INFO/MainProcess] Connected to amqp://guest:**@127.0.0.1:5672//
[2018-09-04 22:37:41,913: INFO/MainProcess] mingle: searching for neighbors
[2018-09-04 22:37:42,947: INFO/MainProcess] mingle: all alone
[2018-09-04 22:37:42,993: INFO/MainProcess] celery@martina-300E4Z-300E5Z-300E7Z ready.
```

Slika 2.4: Pokretanje radnih procesa

Ako se želi pozvati funkcija `echo()` na radnom procesu, to nije moguće napraviti direktno. Potrebno je pozvati metodu `delay`:

```
>>> res=echo.delay('Python rocks!')
>>> print(type(res))
<class 'celery.result.AsyncResult'>
>>> print(res)
f9b56ec4-2841-462a-ac64-097884df0b19
>>> res.ready()
True
>>> res.result
'Python rocks!'
```

Slika 2.5: Korištenje metode delay

```
[2018-09-04 22:46:24,188: INFO/MainProcess] Received task: test.echo[f9b56ec4-2841-462a-ac64-097884df0b19]
[2018-09-04 22:46:24,326: INFO/ForkPoolWorker-1] Task test.echo[f9b56ec4-2841-462a-ac64-097884df0b19] succeeded
 in 0.1345282039999347s: 'Python rocks!'
```

Slika 2.6: Zapisi radnih procesa

Primjetno je da poziv `echo.delay('Python rocks!')` ne vraća string. Umjesto toga, postavlja zahtjev za izvršavanje funkcije `echo` u radni red koji radi na RabbitMQ ser-

veru te vraća instancu `AsyncResult`. Kao što je prikazano u `concurrent.futures` modulu, ovaj objekt je rezervirano mjesto za podatak koji će biti proizveden asinkronim pozivom. U ovom slučaju, asinkroni poziv je `echo` funkcija koja je stavljena u red i radni proces ju je pokupio.

Potrebno je provjeriti `AsyncResult` objekte da znamo jesu li spremni. Ako jesu, može se pristupiti rezultatu, što je u ovom slučaju string `'Python rocks!'`.

Na drugoj slici vidljivo je što se događa s radnim procesima. Oni su dobili zahtjeve za rad s `echo`.
Radni procesi se prekidaju s CRTL+C.

2.3 Python-RQ

U zadnja dva potpoglavlja bit će rečeno nešto o alternativama za Celery. Prva alternativa je **Python-RQ**. Temelji se na Redisu, na kojem su radni redovi, i sadrži rezultat kojeg vrate radni procesi. Namijenjen je onim aplikacijama gdje složena ovisnost zadataka ili usmjeravanje zadataka nije potrebno. Pogledajmo idući kod:

Listing 2.2: RedisQueue.py [5]

```
import redis

class RedisQueue(object):
    def __init__(self, name, namespace='queue', **redis_kwargs):
        """host='localhost', port=6379, db=0"""
        self.__db= redis.Redis(**redis_kwargs)
        self.key = '%s:%s' %(namespace, name)

    def qsize(self):
        return self.__db.llen(self.key)

    def empty(self):
        return self.qsize() == 0

    def put(self, item):
        self.__db.rpush(self.key, item)

    def get(self, block=True, timeout=None):
        if block:
            item = self.__db.blpop(self.key, timeout=timeout)
        else:
            item = self.__db.lpop(self.key)

        if item:
            item = item[1]
        return item
```

```
    def get_nowait(self):
        return self.get(False)
```

A neki od poziva izgledaju ovako:

```
martina@martina-300E4Z-300E5Z-300E7Z ~/Desktop $ python3
Python 3.5.2 (default, Nov 23 2017, 16:37:01)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from RedisQueue import RedisQueue
>>> q = RedisQueue('test')
>>> q.put('hello world')
>>> q.get()
b'hello world'
>>>
```

Kada se pogleda u Redis bazu s klijentom `redis-cli`, dobiju se idući rezultati:

```
martina@martina-300E4Z-300E5Z-300E7Z ~/Desktop $ redis-cli
127.0.0.1:6379> keys *
1) "queue:test"
127.0.0.1:6379> type queue:test
list
127.0.0.1:6379> llen queue:test
(integer) 1
127.0.0.1:6379> lrange queue:test 0 1
1) "hello world"
127.0.0.1:6379>
```

Zanimljivo je da će idući poziv `q.get()` blokirati ostale dok se ne stavi nova stavka u red. Primjetno je da, budući da se sve vrti na istom računalu, nije potrebno namještati IP adrese na poslužiteljskom računalu i utičnicu. No, kada bi se radilo s više računala (virtualnih strojeva), onda bi bilo potrebno postaviti da se spajaju na pravo poslužiteljsko računalo (u `def __init__` umjesto `**redis_kwargs` unese se poslužiteljsko računalo na kojem radi Redis).

Celery i Python-RQ su jako slični. Celery je raširenji u primjeni, no Python-RQ ga sustiže. Glavna prednost Python-RQ-a je što je jednostavniji za korištenje i implementaciju te nema potrebe za posrednikom (RabbitMQ).

2.4 Pyro

Pyro (Python Remote Objects) je biblioteka (paket) koja omogućava razvoj distribuiranih aplikacija čiji objekti mogu komunicirati preko mreže uz minimalni programerski napor.

Pruža niz moćnih značajki koje omogućuju brzu distribuciju distribuiranih aplikacija. Pyro je napisan u čistom Pythonu i zato se koristi na mnogim platformama i Python verzijama, uključujući Python 3.x.

Glavni nedostaci ovog paketa su što su neki objekti u kodu lokalni, dok su drugi objekti udaljeni. Razlog tome je postojanje velikog broja neispravnih načina pokretanja udaljenog koda koji su zanemareni jer je izvođenje tog dalekog koda skriveno iza proxy objekta. Pyro se ponekad može ispravno, ali i teško pokrenuti na ad hoc mreži, gdje se svi nazivi poslužiteljskih računala ne mogu saznati ili na mrežama na kojima je prijenos preko UDP-a onemogućen. No, svejedno ćemo na jednom jednostavnom primjeru pogledati kako radi. Instalacija je vrlo jednostavna: u terminalu se pokrene naredba

```
$ sudo pip install pyro4
```

Radi se o jednostavnom servisu koji vraća poruku onima koji ga pozovu (klijentima) odnosno, postoji komunikacija klijent-server.

Listing 2.3: greeting-server.py [6]

```
import Pyro4

@Pyro4.expose
class GreetingMaker(object):
    def get_fortune(self, name):
        return "Hello, {0}. Here is your fortune message:\n" \
               "Behold the warranty --" \
               " the bold print giveth and the fine " \
               "print taketh away.".format(name)

daemon = Pyro4.Daemon()
uri = daemon.register(GreetingMaker)

print("Ready. Object uri =", uri)
daemon.requestLoop()
```

Otvori se konzola (terminal) i pokrene se server (poslužitelj).

```
martina@martina-300E4Z-300E5Z-300E7Z ~/PycharmProjects/pyro $ python3 greeting-server.py
Ready. Object uri = PYRO:obj_b2fb1e0e5fc543e1adaab2b5bf303ebb@localhost:41359
```

Slika 2.7: Pokretanje servisa

Server je pokrenut. Pogledajmo kod s kojim se klijent spaja i poziva server:

Listing 2.4: greeting-client.py [6]

```
import Pyro4

uri = input("What is the Pyro uri of the greeting object? ").strip()
```

```
name = input("What is your name? ").strip()

greeting_maker = Pyro4.Proxy(uri)
print(greeting_maker.get_fortune(name))
```

Nakon toga pokreće se klijent u novom terminalu/konzoli:

```
martina@martina-300E4Z-300E5Z-300E7Z ~/PycharmProjects/pyro $ python3 greeting-client.py
What is the Pyro uri of the greeting object? PYRO:obj_b2fb1e0e5fc543e1adaab2b5bf303ebb@localhost:41359
What is your name? Martina
Hello, Martina. Here is your fortune message:
Behold the warranty -- the bold print giveth and the fine print taketh away.
```

Slika 2.8: Pokretanje klijenta i spajanje na server

Dakle, nakon pokretanja klijenta, na upit `What is the Pyro uri of the greeting object?` potrebno je kopirati URI iz konzole gdje je pokrenut server. Ali, Pyro nudi mogućnost i da se to odvije automatski. Potrebno je imenovati objekte koristeći logička imena i ime servera da bi se pronašao odgovarajući URI. Nakon napravljenih manjih izmjena u skriptama `greeting-server.py` i `greeting-client.py`:

Listing 2.5: greeting-server.py [6]

```
# saved as greeting-server.py
import Pyro4

@Pyro4.expose
class GreetingMaker(object):
    def get_fortune(self, name):
        return "Hello, {0}. Here is your fortune message:\n" \
               "Tomorrow's lucky number is 12345678.".format(name)

daemon = Pyro4.Daemon()
ns = Pyro4.locateNS()
uri = daemon.register(GreetingMaker)
ns.register("example.greeting", uri)
print("Ready.")
daemon.requestLoop()
```

Listing 2.6: greeting-client.py [6]

```
import Pyro4

name = input("What is your name? ").strip()

greeting_maker = Pyro4.Proxy("PYRONAME:example.greeting")
print(greeting_maker.get_fortune(name))
```

Kod za klijenta je nešto jednostavniji jer se koristi server za pronalazak objekta. Program treba saznati ime Pyro servera koji je pokrenut. Potrebno je pokrenuti jedan server tako da se u novi terminal upiše `pyro4-ns`:

```
martina@martina-300E4Z-300E5Z-300E7Z ~/PycharmProjects/pyro $ pyro4-ns
Not starting broadcast server for localhost.
NS running on localhost:9090 (127.0.0.1)
Warning: HMAC key not set. Anyone can connect to this server!
URI = PYRO:Pyro.NameServer@localhost:9090
```

Slika 2.9: Pokretanje Pyro servera

Klijent i server se pokreću jednako kao i prije. Više nema potrebe kopiranja i lijepljenja URI klijentu, već će on sam biti automatski otkriven.

```
martina@martina-300E4Z-300E5Z-300E7Z ~/PycharmProjects/pyro $ python3 greeting-server2.py
Ready.
```

Slika 2.10: Pokretanje servera

```
martina@martina-300E4Z-300E5Z-300E7Z ~/PycharmProjects/pyro $ python3 greeting-client2.py
What is your name? Martina
Hello, Martina. Here is your fortune message:
Tomorrow's lucky number is 12345678.
```

Slika 2.11: Pokretanje klijenta

Ako se želi provjeriti ime servera, upiše se naredba `pyro4-nsc list`:

```
martina@martina-300E4Z-300E5Z-300E7Z ~/PycharmProjects/pyro $ pyro4-nsc list
--------START LIST
Pyro.NameServer --> PYRO:Pyro.NameServer@localhost:9090
    metadata: ['class:Pyro4.naming.NameServer']
example.greeting --> PYRO:obj_621b3098db60438f811ab525c5d6df42@localhost:42137
--------END LIST
```

Slika 2.12: Lista pokrenutih servera

Poglavlje 3

Studijski primjeri

U ovom poglavlju pokazat će se dvije Celery aplikacije i kako one rade. Prva će raditi (računati) pretvorbe između raznih valuta (kao jedna vrsta tečajne liste), a druga distribuirano sortiranje algoritmom Merge Sort. U oba primjera koristit će se Celery.

3.1 Razmjena valuta

Budući da se radi o jednoj vrsti tečajne liste, koristi se Pythonov modul `CurrencyConverter`[8] za pretvorbe iz jedne valute u drugu. Za ovu aplikaciju potrebna su tri računala ili virtualna stroja. Budući da je sve pokrenuto lokalno, potrebno je samo otvoriti tri terminala. U prvom terminalu (računalu) pokrene se server RabbitMQ, kako je pokazano u drugom poglavlju. U drugom terminalu pokrenu se radni procesi, a u trećem glavni kod. Program je podijeljen u dvije skripte: `currency.py` i `main_currency.py`:

Listing 3.1: currency.py

```
from threading import Thread
from queue import Queue
from currency_converter import CurrencyConverter

import celery
import urllib.request
app = celery.Celery('currency',
            broker='amqp://localhost',
            backend='amqp://localhost')

@app.task
def get_rate(pair):
    c = CurrencyConverter()
    value=c.convert(1, pair[:3], pair[3:])
    return(pair, value)
```

Listing 3.2: main_currency.py

```
from currency import get_rate
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('pairs', type=str, nargs='+')
args = parser.parse_args()

results = [get_rate.delay(pair) for pair in args.pairs]
for result in results:
    try:
        pair, rate = result.get(timeout=1)
    except:
        print('Ops! Got an exception.')
    else:
        print(pair, rate)
```

Praktično je istražiti nekoliko mogućih ponašanja, poput uspješnog poziva, poziva koji ne radi zbog nedostatka radnih procesa pa poziv ne uspijeva i podiže se iznimka. Prvi slučaj je gdje sve funkcionira.

Kao što je rečeno, potrebno je pokrenuti RabbitMQ server te Redis, ukoliko ga imamo. No, i u ovom primjeru je i dalje dovoljan samo RabbitMQ. Na drugom računalu u terminalu se pokrene skripta `currency.py` u radnom direktoriju `book`:

```
$ celery -A currency worker --loglevel=info
```

I na kraju, na trećem računalu u terminalu u istom direktoriju pokrene se `main_currency.py`:

```
$ python3.5 main_currency.py EURUSD CHFUSD GBPUSD GBPCHF USDEUR
EURHRK
```

Ako se pogleda terminal gdje su pokrenuti radni procesi, vidljivi su zapisi slični ovima:

```
[tasks]
  . currency.get_rate

[2018-09-04 22:54:35,861: INFO/MainProcess] Connected to amqp://guest:**@127.0.0.1:5672//
[2018-09-04 22:54:35,895: INFO/MainProcess] mingle: searching for neighbors
[2018-09-04 22:54:36,949: INFO/MainProcess] mingle: all alone
[2018-09-04 22:54:36,983: INFO/MainProcess] celery@martina-300E4Z-300E5Z-300E7Z ready.
[2018-09-04 22:55:19,584: INFO/MainProcess] Received task: currency.get_rate[18a2e7fb-331d-4aa6-99bd-1abf864cefb2]
[2018-09-04 22:55:19,585: INFO/MainProcess] Received task: currency.get_rate[243babb8-fc31-470c-98a2-5b1b2d968cff]
[2018-09-04 22:55:19,586: INFO/MainProcess] Received task: currency.get_rate[ed75e116-21c5-4ba9-8032-2c587a1bfda1]
[2018-09-04 22:55:19,588: INFO/MainProcess] Received task: currency.get_rate[26c85420-3426-4d30-89ba-7455675571f8]
[2018-09-04 22:55:19,590: INFO/MainProcess] Received task: currency.get_rate[653ab737-87be-4521-8d40-08a1e55f24c9]
[2018-09-04 22:55:19,591: INFO/MainProcess] Received task: currency.get_rate[b46163be-dd01-4edf-abf9-d03e8bf7e253]
[2018-09-04 22:55:20,492: INFO/ForkPoolWorker-1] Task currency.get_rate[18a2e7fb-331d-4aa6-99bd-1abf864cefb2] succeeded
 in 0.9052004720001605s: ('EURUSD', 1.1602)
[2018-09-04 22:55:20,650: INFO/ForkPoolWorker-3] Task currency.get_rate[ed75e116-21c5-4ba9-8032-2c587a1bfda1] succeeded
 in 1.0619310839999798s: ('GBPUSD', 1.2965591229619033)
[2018-09-04 22:55:20,695: INFO/ForkPoolWorker-2] Task currency.get_rate[243babb8-fc31-470c-98a2-5b1b2d968cff] succeeded
 in 1.106821451000087s: ('CHFUSD', 1.0062445793581958)
[2018-09-04 22:55:20,695: INFO/ForkPoolWorker-4] Task currency.get_rate[26c85420-3426-4d30-89ba-7455675571f8] succeeded
 in 1.1032749839998814s: ('GBPCHF', 1.2885129018919794)
[2018-09-04 22:55:21,107: INFO/ForkPoolWorker-1] Task currency.get_rate[653ab737-87be-4521-8d40-08a1e55f24c9] succeeded
 in 0.6117822339999748s: ('USDEUR', 0.8619203585588693)
[2018-09-04 22:55:21,152: INFO/ForkPoolWorker-3] Task currency.get_rate[b46163be-dd01-4edf-abf9-d03e8bf7e253] succeeded
 in 0.4949285499999405s: ('EURHRK', 7.4176)
```

Slika 3.1: Radni procesi

```
(book) martina@martina-300E4Z-300E5Z-300E7Z ~/venvs/book $ python3.5 main_currency.py
 EURUSD CHFUSD GBPUSD GBPCHF USDEUR EURHRK
EURUSD 1.1602
CHFUSD 1.0062445793581958
GBPUSD 1.2965591229619033
GBPCHF 1.2885129018919794
USDEUR 0.8619203585588693
EURHRK 7.4176
```

Slika 3.2: Pokretanje glavnog programa

U terminalu gdje su pokrenuti radni procesi, svaki zadatak (proces) dobije jedinstveni ID. Dakle, svaki proces će obaviti svoju pretvorbu te ju poslati RabbitMQ-u, koji će čuvati te rezultate dok ih glavni program ne zatraži.

No što bi se dogodilo da nema radnih procesa? Točnije, da ih ne pokrenemo? Tada bi se na računalu (u terminalu) gdje se pokreće

`$ python3.5 main_currency.py EURUSD CHFUSD GBPUSD GBPCHF USDEUR EURHRK`

javila iznimka i to onoliko puta koliko postoji računanja (6):

```
(book) martina@martina-300E4Z-300E5Z-300E7Z ~/venvs/book $ python3.5 main_currency.py
 EURUSD CHFUSD GBPUSD GBPCHF USDEUR EURHRK
Ops! Got an exception.
Ops! Got an exception.
Ops! Got an exception.
Ops! Got an exception.
Ops! Got an exception.
Ops! Got an exception.
```

Slika 3.3: Iznimke

Potrebno je biti pažljiv i uhvatiti sve iznimke koje bi mogli podignuti zadatci. Treba imati na umu da kod koji radi na daljinu možda neće uspjeti iz niza razloga i da iznimka nije nužno povezana sa samim kodom. Stoga je bitno biti u mogućnosti reagirati u takvim situacijama.

3.2 Distribuirano sortiranje

Ponekad postoje složeniji zadatci nego što su do sada bili prikazani. U takvim slučajevima, rezultat jednog ili više zadataka treba biti proslijeđen drugom zadatku. Da bismo to pokazali, pogledajmo sljedeću aplikaciju. Radi se o distribuiranom sortiranju pomoću algoritma Merge Sort. Merge Sort je algoritam tipa ”podijeli pa vladaj”. Dijeli ulazni niz u dvije polovice, poziva se na dvije polovice rekurzivno, a zatim spaja dvije sortirane polovice. Ujedno će se pokazati i performanse kada se sortira niz od 1000, 10000 i 100000 brojeva.

Za aplikaciju su potrebne dvije skripte: `mergesort.py` i `main.py`

Listing 3.3: mergesort.py

```
import celery


app = celery.Celery('mergesort',
                    broker='amqp://localhost',
                    backend='amqp://localhost')


@app.task
def sort(xs):
    lenxs = len(xs)
    if(lenxs <= 1):
        return(xs)

    half_lenxs = lenxs // 2
    left = xs[:half_lenxs]
    right = xs[half_lenxs:]
    return(merge(sort(left), sort(right)))
```

```
def merge(left, right):
    nleft = len(left)
    nright = len(right)

    merged = []
    i = 0
    j = 0
    while i < nleft and j < nright:
        if(left[i] < right[j]):
            merged.append(left[i])
            i += 1
        else:
            merged.append(right[j])
            j += 1
    return merged + left[i:] + right[j:]
```

Listing 3.4: main.py

```
import random
import time
from celery import group
from mergesort import sort, merge
import json

sequence = list(range(1000))

random.shuffle(sequence)
with open("shuffle.txt","w") as wfile:
    json.dump(sequence,wfile)
t0 = time.time()

n = 4
l = len(sequence) // n
subseqs = [sequence[i * l:(i + 1) * l] for i in range(n - 1)]
subseqs.append(sequence[(n - 1) * l:])
for i in range(n+1):
    for j in range(i):
        with open("chunck%s.txt" %j,"w") as wfile:
                        json.dump(subseqs[j],wfile)

partials = group(sort.s(seq) for seq in subseqs)().get()
with open("sorted.txt","w") as wfile:
        json.dump(partials,wfile)


result = partials[0]
```

```
for partial in partials[1:]:
        result = merge(result, partial)
        with open("result.txt","w") as wfile:
                json.dump(result,wfile)

dt = time.time() - t0
print('Distribuiranom merge sortu treba %.02fs' % (dt))

t0 = time.time()
truth = sort(sequence)
dt = time.time() - t0
print('Lokalnom merge sortu treba %.02fs' % (dt))

assert result == truth
assert result == sorted(sequence)
```

Prvo se generira lista od 1000 brojeva slučajnog redoslijeda metodom `random.shuffle`. Tada se ta lista podijeli na 4 manje podliste otprilike jednakih duljina.

Celery nudi brojne mogućnosti za izvršavanje zadataka, a `group` je jedna od njih. Ona dozvoljava mogućnost izvršavanja trenutnog zadatka tako da ga se izgradi u virtualni zadatak. Obraćanjem pozornosti na metodu `get()`, vidljivo je da je ona nužna u slučaju kada rezultat iz pozadine nije dostupan sve dok svi zadatci (sortiranja) ne budu gotovi i dok se ne vrati rezultat u listu. `group` poziva metodu koja uzima listu popisanih zadataka (dobije se u primjeru pozivom metode `s()` s argumentima zadatka). Popisi zadataka su mehanizmi koje Celery koristi za prosljeđivanje zadataka kao argumente drugim zadacima bez izvršavanja na mjestu.
Ostatak koda samo spaja sortirane podnizove lokalno, dva odjednom. Nakon što distribuirano sortiranje završi, tada se ponovno sortira početni zadani niz, ali lokalno, koristeći isti algoritam te se usporede rezultati distribuiranog Merge Sorta s ugrađenim algoritmom za sort. Potrebno je napomenuti da se aplikaciju pokreće na tri računala (terminala) kao u prethodnom primjeru: na jednom se pokrene RabbitMQ, na drugom računalu se u radnom direktoriju pokrenu radni procesi
`celery -A mergesort worker --loglevel=info`
te se na trećem pokrene glavni kod, također u istom radnom direktoriju: `python3.5 main.py`
Pogledajmo rezultate za niz od 1000, 10000,100000 brojeva:

```
(book) martina@martina-300E4Z-300E5Z-300E7Z ~/venvs/book $ python3.5 main.py
Distribuiranom merge sortu treba 0.67s
Lokalnom merge sortu treba 0.06s
```

Slika 3.4: Rezultati sortiranja niza od 1000 brojeva

```
(book) martina@martina-300E4Z-300E5Z-300E7Z ~/venvs/book $ python3.5 main.py
Distribuiranom merge sortu treba 1.06s
Lokalnom merge sortu treba 0.47s
```

Slika 3.5: Rezultati sortiranja niza od 10000 brojeva

```
(book) martina@martina-300E4Z-300E5Z-300E7Z ~/venvs/book $ python3.5 main.py
Distribuiranom merge sortu treba 3.62s
Lokalnom merge sortu treba 5.69s
```

Slika 3.6: Rezultati sortiranja niza od 100000 brojeva

Ako pogledamo terminal u kojem su pokrenuti radni procesi, primjetno je da su zadatci primljeni, odrađeni te da su rezultati poslani natrag.

```
[tasks]
  . mergesort.sort

[2018-09-04 23:46:14,371: INFO/MainProcess] Connected to amqp://guest:**@127.0.0.1:5672//
[2018-09-04 23:46:14,386: INFO/MainProcess] mingle: searching for neighbors
[2018-09-04 23:46:15,422: INFO/MainProcess] mingle: all alone
[2018-09-04 23:46:15,463: INFO/MainProcess] celery@martina-300E4Z-300E5Z-300E7Z ready.
[2018-09-04 23:46:18,990: INFO/MainProcess] Received task: mergesort.sort[b6364f7d-1cb7-4264-a18f-76d3839b071e]
[2018-09-04 23:46:18,993: INFO/MainProcess] Received task: mergesort.sort[cc74adf3-722a-43fd-aef2-b30604975045]
[2018-09-04 23:46:19,004: INFO/MainProcess] Received task: mergesort.sort[85690fc3-4edb-4f54-8923-63514c527c8d]
[2018-09-04 23:46:19,005: INFO/MainProcess] Received task: mergesort.sort[0e7cbef1-86f9-44ab-86e0-af769b8dba30]
[2018-09-04 23:46:19,065: INFO/ForkPoolWorker-1] Task mergesort.sort[85690fc3-4edb-4f54-8923-63514c527c8d] succeeded
in 0.05959033600004204s: [4, 8, 9, 19, 20, 23, 26, 27, 30, 40, 46, 57, 61, 66, 67, 69, 72, 79, 80, 81, 83, 92, 93, 94
, 95, 96, 103, 105, 107, 108, 115, 116, 127, 129, 134, 145, 146, 150, 153, 155, 158, 162, 166, 168, 169, 179, 180, 18
1, 184, 188, 189, 190, 199, 201, 203, 211, 215, 216, 218, 221, 222, 227, 231, 234, 243, 244, 246, 250, 252, 253, 265,
 266, 276, 289, 291, 294, 297, 301, 303, 304, 306, 320, 321, 322, 323, 328, 329, 331, 344, 353, 358, 361, 368, 380, 3
84, 388, 395, 420, 421, 425, 427, 439, 442, 443, 451, 456, 464, 480, 481, 485, 488, 492, 494, 497, 502, 509, 516, 521
, 528, 529, 530, 531, 532, 536, 537, 545, 549, 553, 557, 558, 576, 580, 584, 586, 587, 591, 593, 596, 598, 599, 600,
603, 608, 611, 613, 615, 634, 635, 642, 647, 649, 650, 654, 655, 657, 659, 662, 665, 670, 672, 675, 682, 686, 688, 68
9, 691, 692, 698, 699, 702, 705, 711, 714, 720, 721, 722, 724, 738, 741, 742, 745, 747, 748, 751, 752, 753, 756, 757,
 762, 768, 775, 783, 789, 794, 798, 800, 804, 810, 819, 821, 826, 833, 834, 835, 836, 840, 842, 848, 852, 855, 85...,
 ...]
```

Slika 3.7: Radni procesi- Merge Sort

Kao što se može primijetiti, distribuirano sortiranje radi brže na većem nizu brojeva, dok na nizu od 1000 i 10000 radi sporije od lokalnog sortiranja.
Jednostavna implementacija koja koristi višestruke procese (čak i `multiprocessing` ili `concurrent.futures`) pokazuje da se može očekivati nekoliko puta povećanje performansi s ovim jednostavnim algoritmom.

Glavni problem je što je u Celeryu sinkronizacija jednostavnih funkcija skupa i treba se koristiti samo onda kada je nužna. Razlog tome je što Celery provjerava status dijela rezultata iz grupe da budu spremni, tako da kasniji zadaci mogu biti zakazani, što može dovesti do većih troškova.

Zaključak

Distribuirani sustavi danas su široko primjenjivi. Koriste se u raznim područjima: od telekomunikacija, mrežnih aplikacija kao što su distribuirane baze podataka ili World Wide Web, do aplikacija preko kojih se može kontrolirati sustav u realnom vremenu, npr. sustav za kontrolu leta. Zbog tako široke primjene, distribuirane aplikacije su implementirane u raznim programskim jezicima, pa tako i u Pythonu.

Tijekom pisanja ovog rada uočeno je da se dosta aplikacija razvija pomoću **Django** - Pythonov Web aplikacijski okvir te **Flower** - aplikacijski okvir za praćenje radnih procesa u Celeryu, uz pakete navedene u radu. Paketi Celery i Python-RQ trenutno su najrašireniji Pythonovi paketi za razvoj distribuiranih aplikacija, stoga se navode neke sličnosti i razlike među njima:

- **Dokumentacija.** Python-RQ ima jednostavnu i lako razumljivu dokumentaciju. Dokumentacija za Celery je kompliciranija zbog puno opcija koje nudi na početku dok se uspostavlja okruženje pa ju treba više proučavati.
- **Praćenje rada.** Oba paketa nude odlične aplikacijske okvire u kojima se može pratiti što se događa: Flower za Celery i RQ-Dashboard za Python-RQ.
- **Posrednik.** Ovdje veliku prednost ima Celery. U Celeryu se mogu koristiti i RabbitMQ i Redis (ili samo RabbitMQ kao što je prikazano u radu), dok Python-RQ koristi samo Redis. Ovo je bitno jer više različitih posrednika garantira bolju sigurnost. Redis ne garantira stopostotni prijenos poruke ili zadatka prema radnim procesima.
- **Operacijski sustavi.** Python-RQ radi samo na sustavima koji imaju podršku za operaciju `fork` (Unix).
- **Podrška u programskim jezicima.** Python-RQ radi samo u Pythonu, a Celery ima mogućnost da se zadaci šalju radnim procesima koji mogu biti i u drugim programskim jezicima.

Iz ovih činjenica može se zaključiti da se Celery više koristi za razvoj distribuiranih aplikacija, no Python-RQ ga polako sustiže. Oba paketa bi se trebala nastaviti razvijati

te koristiti za razvoj i u budućnosti, budući da se već sad koriste za razvoj na *super-računalima* poput **High Performance Computing (HPC) klastera**.

Bibliografija

[1] Francesco Pierfederici. *Distributed Computing with Python*. Packt Publishing, Birmingham, UK, 2012.

[2] Robert Manger. *Skripta*. PMF, Matematički odsjek, Zagreb.

[3] *RabbitMQ*,
`https://www.rabbitmq.com`
Pristupljeno: kolovoz, 2018.

[4] *Redis*,
`http://redis.io`
Pristupljeno: kolovoz, 2018.

[5] *Simple Python Queue with Redis*,
`http://peter-hoffmann.com/2012/python-simple-queue-redis-queue.html` Pristupljeno: kolovoz, 2018.

[6] *Intro and example*,
`https://pythonhosted.org/Pyro4/intro.html`
Pristupljeno: kolovoz, 2018.

[7] *How To Install RabbitMQ In Ubuntu*
`https://www.linuxhelp.com/how-to-install-rabbitmq-in-ubuntu/`
Pristupljeno: kolovoz, 2018.

[8] *CurrencyConverter 0.13.6*
`https://pypi.org/project/CurrencyConverter/`
Pristupljeno: kolovoz, 2018.

[9] *Jython Documentation*
`https://wiki.python.org/jython`
Pristupljeno: kolovoz, 2018.

Sažetak

U ovom radu proučavane su distribuirane aplikacije u programskom jeziku Python. Distribuirani, pa i paralelni sustavi, danas su široko primjenjivi. Prikazan je Amdahlov zakon koji govori da uložen napor u paralelizaciju postojećeg algoritma nadmašuje dobitke u performansama. Također su proučavani pojmovi višedretvenosti, višestrukih procesa i višeprocesorskih redova te neke njihove značajke i primjene. Pokazano je i kako se uspostavlja Celery pomoću kojeg je moguće razvijati distribuirane aplikacije i navedene su dvije njegove alternative. Na kraju rada, proučavane su dvije distribuirane aplikacije. Pokazalo se kako se vrši komunikacija između pojedinih dijelova aplikacija u određenim situacijama te što je potrebno promijeniti da rade na više računala. Praćeno je izvršavanje druge aplikacije (distribuirano sortiranje) te je zaključeno da, na velikom skupu podataka, bolje vrijeme izvršavanja ima distribuirana aplikacija sortiranja u odnosu na algoritam sortiranja koji se izvršavao na lokalnom računalu.

Summary

In this work, distributed applications are studied in the Python programming language. Distributed and parallel systems are today widely applied. Amdahl's law is introduced, which states that the effort to parallelize the existing algorithm outweighs performance gains. Also, multiple threads, multiple processes and multiprocess queues are studied and some of their features and applications. It was also demonstrated how Celery is configured, by which it is possible to develop distributed applications and two of its alternatives are listed. At the end of the work, two distributed applications were studied. Communication between specific parts of the application in certain situations has been shown and what needs to be changed to work on multiple computers. The other application was executed (distributed sorting) and it was concluded that, on a large data set, better execution time is achieved by the distributed sorting application compared to the sorting algorithm executed on a local computer.

Životopis

Rođena sam 19. veljače 1993. u Zagrebu, gdje sam pohađala Gimnaziju Lucijana Vranjanina (opći smjer). Godine 2011. upisujem Preddiplomski sveučilišni studij Matematika na matematičkom odsjeku Prirodoslovno-matematičkog fakulteta u Zagrebu, a 2015. upisujem diplomski studij Računarstvo i matematika. Tijekom ljeta 2016. godine sudjelovala sam na Ericsson Summer Campu, a kroz akademsku godinu 2017./2018. radila sam u Zagrebačkoj banci na IT odjelu za aplikativni razvoj upravljanja dokumentacijom. U svibnju 2018. sudjelujem na Combisovom try{code}catch hackatonu u sklopu Combisove konferencije.