11. domaća zadaća – paralelizacija (1)

Uvod

Počevši od ove domaće zadaće fokus zadataka bit će na različitim oblicima paralelizacije algoritama evolucijskog računanja.

Priprema

Prvi dio ove zadaće odnosi se na pripremu generičke infrastrukture koju ćete koristiti u ostalim dijelovima zadaće.

Algoritmi evolucijskog računanja u nizu različitih operacija trebaju pristup generatoru slučajnih brojeva. Implementacije takvih generatora najčešće nisu višedretveno sigurne pa se pri pisanju višedretvenih programa postavlja pitanje kako na svako mjesto gdje je to potrebno dostaviti referencu na generator koji dretva koja ga na tom mjestu treba može sigurno koristiti. Jedno od mogućih rješenja ovog problema je pobrinuti se da svaka dretva ima svoj vlastiti primjerak generatora.

S obzirom na neke specifičnosti postojeće implementacije generatora slučajnih brojeva koji nam stoji na raspolaganju u Javi, željeli bismo također biti u mogućnosti konkretnu implementaciju generatora promijeniti a bez da ostatak koda bude svjestan toga.

U nastavku ćemo stoga pokušati pronaći rješenje na sljedeće probleme:

- kôd koji treba generator slučajnih brojeva ne smije razlikovati situaciju u kojoj se izvodi u jednodretvenom ili pak višedretvenom okruženju (drugim riječima, ne želimo pisati operator mutacije koji razmatra u kakvom se okruženju izvodi);
- kôd koji treba generator slučajnih brojeva ne smije znati s kakvom implementacijom generatora radi (to treba biti konfigurabilno na jednom mjestu "negdje vani");
- ne želimo slati reference na generatore slučajnih brojeva posvuda po kodu i pamtiti ih kroz konstruktore i privatne varijable operatora mutacije, križanja i slično.

Kako bismo zadovoljili postavljene zahtjeve, najprije ćemo osigurati mogućnost promjene implementacije generatora slučajnih brojeva: funkcionalnost takvog generatora definirat ćemo kroz sučelje `IRNG` koje je dano u nastavku.

```
package hr.fer.zemris.optjava.rng;

/**
 * Sučelje koje predstavlja generator slučajnih brojeva.
 *
 * @author marcupic
 */
public interface IRNG {
	/**
	 * Vraća decimalni broj iz intervala [0,1) prema uniformnoj distribuciji.
	 *
	 * @return slučajno generirani decimalni broj
	 */
	public double nextDouble();
	/**
	 * Vraća decimalni broj iz intervala [min,max) prema uniformnoj distribuciji.
	 *
	 * @param min donja granica intervala (uključiva)
	 * @param max gornja granica intervala (isključiva)
	 *
	 * @return slučajno generirani decimalni broj
	 */
	public double nextDouble(double min, double max);
	/**
	 * Vraća decimalni broj iz intervala [0,1) prema uniformnoj distribuciji.
```

```
         *
         * @return slučajno generirani decimalni broj
         */
        public float nextFloat();
        /**
         * Vraća decimalni broj iz intervala [min,max) prema uniformnoj distribuciji.
         *
         * @param min donja granica intervala (uključiva)
         * @param max gornja granica intervala (isključiva)
         *
         * @return slučajno generirani decimalni broj
         */
        public float nextFloat(float min, float max);
        /**
         * Vraća cijeli broj iz intervala svih mogućih cijelih brojeva prema uniformnoj distribuciji.
         *
         * @return slučajno generirani cijeli broj
         */
        public int nextInt();
        /**
         * Vraća cijeli broj iz intervala [min,max) prema uniformnoj distribuciji.
         *
         * @param min donja granica intervala (uključiva)
         * @param max gornja granica intervala (isključiva)
         *
         * @return slučajno generirani cijeli broj
         */
        public int nextInt(int min, int max);
        /**
         * Vraća slučajno generiranu boolean vrijednost. Vrijednosti se izvlače
         * iz uniformne distribucije.
         *
         * @return slučajno generirani boolean
         */
        public boolean nextBoolean();

        /**
         * Vraća decimalni broj iz normalne distribucije s parametrima (0,1).
         *
         * @return slučajno generirani decimalni broj
         */
        public double nextGaussian();
}
```

Napišite jednu implementaciju ovog sučelja (razred `hr.fer.zemris.optjava.rng.rngimpl.RNGRandomImpl`) koja interno koristi vlastiti primjerak razreda `java.util.Random` koji stvori u konstruktoru).

Objekte (odnosno različite implementacije) koje znaju dohvaćati generatore slučajnih brojeva opisat ćemo sljedećim sučeljem.

```
package hr.fer.zemris.optjava.rng;

/**
 * Sučelje koje predstavlja objekte koji sadrže generator slučajnih
 * brojeva i koji ga stavljaju drugima na raspolaganje uporabom
 * metode {@link #getRNG()}. Objekti koji implementiraju ovo sučelje
 * ne smiju na svaki poziv metode {@link #getRNG()} stvarati i vraćati
 * novi generator već moraju imati ili svoj vlastiti generator koji vraćaju,
 * ili pristup do kolekcije postojećih generatora iz koje dohvaćaju i vraćaju
 * jedan takav generator (u skladu s pravilima konkretne implementacije ovog
 * sučelja) ili isti stvaraju na zahtjev i potom čuvaju u cache-u za istog
 * pozivatelja.
 *
 * @author marcupic
 */
public interface IRNGProvider {

        /**
         * Metoda za dohvat generatora slučajnih brojeva koji pripada
         * ovom objektu.
         *
         * @return generator slučajnih brojeva
         */
        public IRNG getRNG();

}
```

Potom ćemo napraviti dvije implementacije takvog sučelja: razred `ThreadLocalRNGProvider` sadržavat će `ThreadLocal` kolekciju objekata tipa `IRNG` dok će razred `ThreadBoundRNGProvider` pretpostaviti da trenutna dretva sama implementira sučelje `IRNGProvider` pa će trenutnu dretvu ukalupiti u to sučelje i vratiti pohranjeni generator slučajnih brojeva.

http://docs.oracle.com/javase/7/docs/api/java/lang/ThreadLocal.html

U metodi `getRNG()` razreda `ThreadLocalRNGProvider` u slučaju da trenutna dretva nema u `ThreadLocal` objektu pridružen generator stvorit ćete primjerak vaše implementacije sučelja `IRNG`, referencu pohraniti u `ThreadLocal` objekt i istu vratiti pozivatelju. Uočite, `ThreadLocal` je parametrizirani razred pa ćete privatnu varijablu tog tipa deklarirati ovako:

```
private ThreadLocal<IRNG> threadLocal = new ThreadLocal<>();
```

Konačno, kako bismo osigurali da reference na generatore ne trebamo slati okolo po kodu, osigurat ćemo mehanizam koji će nam dopustiti da generator dohvatimo upravo tamo gdje nam treba. Evo ideje: koristeći oblikovni obrazac *jedinstveni objekt* (engl. Singleton) definirat ćemo razred RNG s jednom javnom statičkom metodom getRNG() koju ćemo moći pozivati od bilo gdje iz programa. Taj će razred prilikom statičke inicijalizacije pročitati sadržaj datoteke `rng-config.properties` i u njoj potražiti ključ `rng-provider` koji će imati pridruženo ime razreda čiji jedan primjerak treba stvoriti, privatno ga zapamtiti i potom koristiti za razrješavanje svakog poziva statičke metode `getRNG()`. Ogledni sadržaj datoteke `rng-config.properties` prikazan je u nastavku.

```
rng-provider = hr.fer.zemris.optjava.rng.provimpl.ThreadLocalRNGProvider

# kasnije probati efikasniju implementaciju:
# rng-provider = hr.fer.zemris.optjava.rng.provimpl.ThreadBoundRNGProvider
```

Gruba struktura razreda RNG prikazana je u nastavku.

```
package hr.fer.zemris.optjava.rng;

import java.util.Properties;

public class RNG {

	private static IRNGProvider rngProvider;

	static {
		// Stvorite primjerak razreda Properties;
		// Nad Classloaderom razreda RNG tražite InputStream prema resursu rng-config.properties
		// recite stvorenom objektu razreda Properties da se učita podatcima iz tog streama.
		// Dohvatite ime razreda pridruženo ključu "rng-provider"; zatražite Classloader razreda
		// RNG da učita razred takvog imena i nad dobivenim razredom pozovite metodu newInstance()
		// kako biste dobili jedan primjerak tog razreda; castajte ga u IRNGProvider i zapamtite.
	}

	public static IRNG getRNG() {
		return rngProvider.getRNG();
	}

}
```

Za parsiranje datoteke `rng-config.properties` poslužite se razredom Properties koji nudi svu potrebnu funkcionalnost.

http://docs.oracle.com/javase/7/docs/api/java/util/Properties.html

Važno: izvornik datoteke `rng-config.properties` treba se nalaziti u *src* direktoriju. Prilikom svakog snimanja i promjene te datoteke, Eclipse će njezinu kopiju automatski prebaciti u *bin* direktorij. Po pokretanju programa, Classloader koji ćete tražiti učitavanje te datoteke istu će tražiti isključivo u aktivnom classpath-u (a to je direktorij *bin* u kojem se nalaze i sve .class datoteke odnosno

prevedeni izvorni kodovi, direktorij *src* se ne pretražuje). Stoga vodite računa da tu datoteku ne uređujete vanjskim editorima jer oni neće kopiju prebaciti u *bin* direktorij (ili po povratku u Eclipse napravite *Refresh* projekta).

Ako ste sve napravili kako je opisano, sljedeći testni program će raditi besprijekorno:

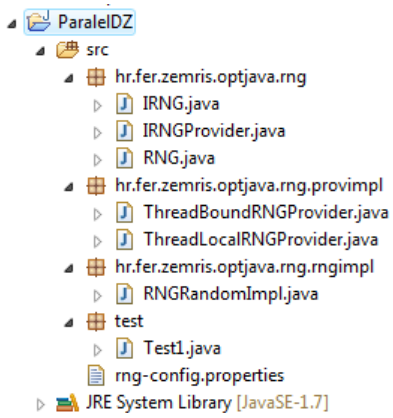
```
package test;

import hr.fer.zemris.optjava.rng.IRNG;
import hr.fer.zemris.optjava.rng.RNG;

public class Test1 {

	public static void main(String[] args) {
		IRNG rng = RNG.getRNG();

		for(int i = 0; i < 20; i++) {
			System.out.println(rng.nextInt(-5, 5));
		}
	}

}
```

Uočite kako koristimo napisanu infrastrukturu: gdje god nam treba generator slučajnih brojeva, naprosto ga dohvatimo pozivom:

```
IRNG rng = RNG.getRNG();
```

i potom koristimo. Vodite računa pri tome da poziv ne ugurate baš u *for*-petlju jer se ipak radi o pozivu metode: dohvatite generator po ulasku u metodu i potom ga koristite. Stvorena infrastruktura pobrinut će se da se za svaku dretvu stvori isključivo po jedan generator i da se taj dalje koristi.

Struktura projekta do ovog tretnutka trebala bi izgledati ovako:



Prebacite li se u datoteci `rng-config.properties` na `ThreadBoundRNGProvider`, testni program će puknuti uz iznimku:

```
Exception in thread "main" java.lang.ClassCastException: java.lang.Thread cannot be cast to
hr.fer.zemris.optjava.rng.IRNGProvider
	at hr.fer.zemris.optjava.rng.provimpl.ThreadBoundRNGProvider.getRNG(ThreadBoundRNGProvider.java:10)
	at hr.fer.zemris.optjava.rng.RNG.getRNG(RNG.java:20)
	at test.Test1.main(Test1.java:9)
```

To je normalno jer smo se prebacili na implementaciju koja pretpostavlja da je dretva koja treba generator (u ovom slučaju to je Javina dretva *main* koja izvodi metodu *main*) ukalupljiva u `IRNGProvider`, što naravno ne stoji. Želimo li koristiti ovaj provider, moramo sami napisati naš novi razred `EVOThread` koji nasljeđuje razred `Thread`, implementira sučelje `IRNGProvider` i ima privati primjerak generatora slučajnih brojeva koji vraća pri pozivu metode `getRNG()`. Razred je prikazan u nastavku.

```
package hr.fer.zemris.optjava.rng;

import hr.fer.zemris.optjava.rng.rngimpl.RNGRandomImpl;

public class EVOThread extends Thread implements IRNGProvider {

	private IRNG rng = new RNGRandomImpl();

	public EVOThread() {
	}

	public EVOThread(Runnable target) {
		super(target);
	}

	public EVOThread(String name) {
		super(name);
	}

	public EVOThread(ThreadGroup group, Runnable target) {
		super(group, target);
	}

	public EVOThread(ThreadGroup group, String name) {
		super(group, name);
	}

	public EVOThread(Runnable target, String name) {
		super(target, name);
	}

	public EVOThread(ThreadGroup group, Runnable target, String name) {
		super(group, target, name);
	}

	public EVOThread(ThreadGroup group, Runnable target, String name,
			long stackSize) {
		super(group, target, name, stackSize);
	}

	@Override
	public IRNG getRNG() {
		return rng;
	}

}
```

Kod koji koristi ovako napisanu dretvu prikazan je u nastavku.

```
package test;

import hr.fer.zemris.optjava.rng.EVOThread;
import hr.fer.zemris.optjava.rng.IRNG;
import hr.fer.zemris.optjava.rng.RNG;

public class Test2 {

	public static void main(String[] args) {

		Runnable job = new Runnable() {

			@Override
			public void run() {
				IRNG rng = RNG.getRNG();

				for(int i = 0; i < 20; i++) {
					System.out.println(rng.nextInt(-5, 5));
				}
			}
		};

		EVOThread thread = new EVOThread(job);
		thread.start();
	}

}
```

Zadatak

U nastavku je dan razred `GrayScaleImage` koji nudi nekoliko gotovih funkcija (komentari su izbačeni zbog uštede na prostoru ali bi kod sam po sebi trebao biti dovoljno jasan).

```
package hr.fer.zemris.art;

import java.awt.image.BufferedImage;
import java.awt.image.WritableRaster;
import java.io.File;
import java.io.IOException;

import javax.imageio.ImageIO;

public class GrayScaleImage {

	private int width;
	private int height;
	private byte[] data;

	public GrayScaleImage(int width, int height) {
		this.width = width;
		this.height = height;
		this.data = new byte[height*width];
	}

	public int getHeight() {
		return height;
	}

	public int getWidth() {
		return width;
	}

	public void clear(byte color) {
		int index = 0;
		for(int h = 0; h < height; h++) {
			for(int w = 0; w < width; w++) {
				data[index] = color;
				index++;
			}
		}
	}

	public byte[] getData() {
		return data;
	}

	public void rectangle(int x, int y, int w, int h, byte color) {
		int xs = x;
		int xe = x + w -1;
		int ys = y;
		int ye = y + h -1;

		// Ako sigurno ne crtam:
		if(width<=xs || height<=ys || xe<0 || ye<0) return;

		if(xs<0) xs = 0;
		if(ys<0) ys = 0;
		if(xe>=width) xe = width-1;
		if(ye>=height) ye = height-1;

		for(int yl = ys; yl <= ye; yl++) {
			int index = yl*width+xs;
			for(int xl = xs; xl <= xe; xl++) {
				data[index] = color;
				index++;
			}
		}
	}

	public void save(File file) throws IOException {
		BufferedImage bim = new BufferedImage(width, height, BufferedImage.TYPE_BYTE_GRAY);
		int[] buf = new int[1];
		WritableRaster r = bim.getRaster();
		int index=0;
		for(int h = 0; h < height; h++) {
```

```
                        for(int w = 0; w < width; w++) {
                                buf[0] = (int)data[index] & 0xFF;
                                r.setPixel(w, h, buf);
                                index++;
                        }
                }
                try {
                        ImageIO.write(bim, "png", file);
                } catch(IOException ex) {
                        throw ex;
                } catch(Exception ex) {
                        throw new IOException(ex);
                }
        }

        public static GrayScaleImage load(File file) throws IOException {
                BufferedImage bim = ImageIO.read(file);
                if(bim.getType()!=BufferedImage.TYPE_BYTE_GRAY) {
                        throw new IOException("Slika nije grayscale.");
                }
                GrayScaleImage im = new GrayScaleImage(bim.getWidth(), bim.getHeight());
                try {
                        int[] buf = new int[1];
                        WritableRaster r = bim.getRaster();
                        int index=0;
                        for(int h = 0; h < im.height; h++) {
                                for(int w = 0; w < im.width; w++) {
                                        r.getPixel(w, h, buf);
                                        im.data[index] = (byte)buf[0];
                                        index++;
                                }
                        }
                } catch(Exception ex) {
                        throw new IOException("Slika nije grayscale.");
                }
                return im;
        }

}
```

Razred omogućava učitavanje i pohranu *grayscale* slika s/na disk (to su slike koje za svaki slikovni element mogu zapamtiti jednu od 256 nijansi sive boje). Razred također omogućava stvaranje "prazne" slike, popunjavanje slike jednom bojom (metoda `clear`) te crtanje pravokutnika zadane početne točke te definirane širine i visine (pozitivni brojevi!).

Zadatak koji ćete rješavati je sljedeći: dobit ćete PNG sliku koja je pohranjena u 256 nijansi sive boje. Genetskim algoritmom ćete evoluirati naputak kako dobiti najbolju aproksimaciju takve slike ako sliku generirate tako da najprije čitavu sliku popunite pozadinskom bojom i potom na nju docrtate fiksan broj pravokutnika. Svaki pravokutnik pri tome je određen s 5 parametara:

- x-koordinata gornjeg lijevog ugla,
- y-koordinata gornjeg lijevog ugla,
- širina pravokutnika (> 0),
- visina pravokutnika (> 0),
- boja kojom treba popuniti površinu pravokutnika.

Ako s Np označimo broj pravokutnika, ukupna veličina kromosoma trebala bi biti 1+5*Np (pozadinska boja plus Np puta 5 parametara po pravokutniku).

Za ovaj problem prirodno je definirana funkcija kazne kao suma apsolutnih odstupanja nacrtanog slikovnog elemenata i slikovnog elementa u originalnoj slici. Pri tome pažnju treba posvetiti načinu na koji se računaju te razlike jer se interno za pohranu intenziteta pridruženog svakom slikovnom elementu koristi tip podatka `byte` koji je 8-bitni ali predstavlja brojeve s predznakom. Posljedica toga je:

```
byte b1 = (byte)128;
byte b2 = (byte)127;
int razlika = b1-b2;   // ==> bit će -255 a ne 1 kako bi očekivali
```

Napišemo li pak izračun razlike ovako:

```
int razlika = ((int)b1 & 0xFF) - ((int)b2 & 0xFF);
```

rezultat će biti 1.

Detalje kako ćete razraditi način predstavljanja rješenja, kako ćete napisati operatore križanja i mutacije i slično ostaje na Vama da razradite. Kako ne bi bilo nejasnoća oko načina izračuna kazne, u nastavku dajem jednu moguću implementaciju. Neka je s modelirano `GASolution` jedno generičko rješenje a s `IGAEvaluator` generički evaluator. U tom slučaju i uz pretpostavku da je kromosom polje integera, evaluator je prikazan u nastavku.

```
package hr.fer.zemris.generic.ga;

public abstract class GASolution<T> implements Comparable<GASolution<T>> {
	protected T data;
	public double fitness;

	public GASolution() {
	}

	public T getData() {
		return data;
	}

	public abstract GASolution<T> duplicate();

	@Override
	public int compareTo(GASolution<T> o) {
		return -Double.compare(this.fitness, o.fitness);
	}
}
```

```
package hr.fer.zemris.generic.ga;

public interface IGAEvaluator<T> {
	public void evaluate(GASolution<T> p);
}
```

```
class Evaluator implements IGAEvaluator<int[]> {

	private GrayScaleImage template;
	private GrayScaleImage im;

	public Evaluator(GrayScaleImage template) {
		super();
		this.template = template;
	}

	public GrayScaleImage draw(GASolution<int[]> p, GrayScaleImage im) {
		if(im==null) {
			im = new GrayScaleImage(template.getWidth(), template.getHeight());
		}
		int[] pdata = p.getData();
		byte bgcol = (byte)pdata[0];
		im.clear(bgcol);
		int n = (pdata.length-1)/5;
		int index = 1;
		for(int i = 0; i < n; i++) {
			im.rectangle(
				pdata[index], pdata[index+1], pdata[index+2], pdata[index+3],
				(byte)pdata[index+4]
			);
			index+=5;
		}
		return im;
	}
```

```
	@Override
	public void evaluate(GASolution<int[]> p) {
		// Ovo nije višedretveno sigurno!
		if(im == null) {
			im = new GrayScaleImage(template.getWidth(), template.getHeight());
		}
		draw(p, im);

		byte[] data = im.getData();
		byte[] tdata = template.getData();
		int w = im.getWidth();
		int h = im.getHeight();

		double error = 0;
		int index2=0;
		for(int y = 0; y < h; y++) {
			for(int x = 0; x < w; x++) {
				error += Math.abs(((int)data[index2]&0xFF)-((int)tdata[index2]&0xFF));
				index2++;
			}
		}

		p.fitness = -error;

	}

}
```

Napomena: dani evaluator nije višedretveno siguran jer čuva referencu na jednu sliku u memoriji po kojoj crta pa potom uspoređuje s originalom. Jedna mogućnost kako ovo učiniti višedretveno sigurnim jest da metoda `evaluate()` svaki puta stvori novu praznu sliku. Međutim, kako je slika veliki potrošač memorije, neprestano alociranje novih slika ekstremno brzo će potrošiti svu raspoloživu memoriju pa će stoga Javin skupljač smeća neprestano raditi i čistiti memoriju što će srušiti performanse programa. U određenom smislu, slike su skup resurs baš kao što su i generatori slučajnih brojeva. Pokušajte ovaj problem riješiti na sličan način – želite da svaka dretva ima svoj vlastiti evaluator koji će imati svoju vlastitu kopiju slike po kojoj će raditi; alternativno, može postojati jedan evaluator ali će on onda morati na neki način dohvatiti primjerak slike koji pripada trenutnoj dretvi koja ga izvodi. I jedno i drugo rješenje su OK.

Podzadatak 1

Napišite generacijski GA kod kojeg je paraleliziran izračun dobrote. U okviru rješavanja ovog zadatka ne smijete koristiti `ExecutorService` usluge već sami morate stvarati dretve i njima upravljati, kako smo objasnili na predavanju. Koristite dva reda: jedan u koji će glavna dretva ubacivati jedinke koje treba evaluirati te drugi iz kojeg će glavna dretva preuzimati evaluirane jedinke i smještati ih dalje u populaciju. Red treba biti modeliran sučeljem `Queue`:

http://docs.oracle.com/javase/7/docs/api/java/util/Queue.html

a kao implementaciju odaberite `ConcurrentLinkedQueue`:

http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentLinkedQueue.html

Kao radnike stvorite onoliko dretvi koliko Vaše računalo ima procesora (ovo neka program sam dozna prilikom pokretanja).

Program smjestite u razred `hr.fer.zemris.optjava.dz11.Pokretac1` koji preko komandne linije dobiva sedam argumenata:

- putanju do originalne PNG slike
- broj kvadrata kojima se aproksimira slika
- veličina populacije s kojom se radi
- broj generacija koje je maksimalno dozvoljeno odraditi

- minimalni fitness (definiran kao -kazna) uz koji algoritam može stati i prije
- stazu do txt datoteke u koju će biti ispisani optimalni parametri (jedan broj po retku)
- stazu do slike koju će program generirati i koja prikazuje pronađenu aproksimaciju

Glavna dretva ovdje se brine o svemu (selekcija, križanje, mutacija) osim o evaluaciji: neevaluirane jedinke gura u prvi red i kada ih sve tamo nagura, isti broj ih vadi iz drugog reda čime dovršava jednu generaciju.

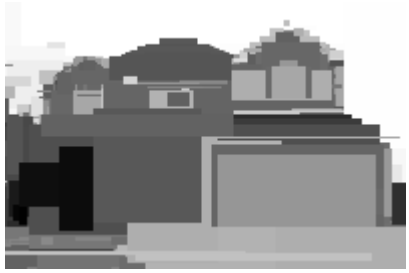
Kada je program gotov, treba uredno pogasiti radnike (ključna riječ/google: *thread queue poisoning*).

Podzadatak 2

Razmotrite sada paralelizaciju koja dozvoljava da se više stvari odvija u paraleli: neka zadaća radnika bude jedan kompletan postupak izrade djeteta: selekcija roditelja, križanje, mutacija i evaluacija. Glavna dretva i dalje brine o generaciji: razlika je što sada u prvi red gura zadatke (tipa: stvori mi 2 djeteta; sam zadatak može imati referencu na populaciju u trenutnoj generaciji iz koje se bira) a iz drugog reda vadi polje od upravo toliko stvorenih i evaliranih djeteta. Dopuštanjem da se radniku preda broj djece koju treba stvoriti odmah omogućavamo i smanjenje sinkronizacijskih troškova: radimo li s populacijom od 100 jedinki, u prvi red možemo nagurati ili 100 zadataka da se stvori po jedno dijete, ili 50 zadataka da se stvore po dva djeteta, ili 25 zadataka da se stvore po 4 djeteta – manji broj poslova, manji troškovi sinkronizacije, ali potencijalno (ako pretjeramo) može dovesti do situacije da su sve dretve gotove i ne rade ništa osim jedne koja se sama trudi napraviti puno djece – u idealnoj situaciji, nitko ne bi smio biti besposlen.

Program koji to radi na ovaj način smjestite u razred `hr.fer.zemris.optjava.dz11.Pokretac2` koji preko komandne linije prima isti broj argumenata kao i prethodni program. Eksperimentalno utvrdite koji je broj stvaranja djece po jednom zadatku optimalan.

Primjer originalne slike i rekonstruirane slike s 200 pravokutnika i relativno velikom pogreškom dan je u nastavku.



Slika je uploadana na Ferka u dodatak domaćim zadaćama (`kuca-200-133.png`).

Napomene:

Za učitavanje i snimanje generiranih slika iz/u PNG razred `GrayScaleImage` sadrži sve potrebne metode.

Rok za predaju Eclipse projekta je četvrtak, 9. siječnja 2014. u 14:00.