VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV TELEKOMUNIKACÍ
DEPARTMENT OF TELECOMMUNICATIONS

GENETICKÉ ALGORITMY – IMPLEMENTACE PARALELNÍHO ZPRACOVÁNÍ
GENETIC ALGORITHMS - IMPLEMENTATION OF MULTIPROCESSING

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. Martin Tuleja**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. Václav Oujezský, Ph.D.**
SUPERVISOR

BRNO 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní obor **Telekomunikační a informační technika**
Ústav telekomunikací

Student: Bc. Martin Tuleja ***ID:*** 183603
Ročník: 2 ***Akademický rok:*** 2017/18

NÁZEV TÉMATU:

Genetické algoritmy – implementace paralelního zpracování

POKYNY PRO VYPRACOVÁNÍ:

Cílem diplomové práce je návrh vlastní knihovny genetického algoritmu v jazyce Python pro paralelní zpracování. Částečně je možné využít knihovny třetích stran. V teoretické části nastudujte a popište možnosti implementace. V praktické části navrhnuté řešení implementujte v jazyce Python a otestujte na jednom z možných příkladů, jako je problém batohu, osmi dam či maximalizace dané funkce.

DOPORUČENÁ LITERATURA:

[1] SHEPPARD, Clinton. Genetic Algorithms with Python. Austin, Texas, USA, 2016-2017. ISBN: 978-15-4032-0-0-9

[2] Pilgrim, M. Ponořme se do Python(u) 3. CZ.NIC, z.s.p.o., 2010, ISBN: 978-80-904248-2-1

Termín zadání: 5.2.2018 ***Termín odevzdání:*** 21.5.2018

Vedoucí práce: Ing. Václav Oujezský, Ph.D.
Konzultant:

prof. Ing. Jiří Mišurec, CSc.
předseda oborové rady

ABSTRAKT

Genetické algoritmy sú moderné algoritmy určené na riešenie optimalizačných problémov. Vznikli inšpiráciou z evolučných procesov v prírode. Ich paralelizáciou sa dosiahne nielen vyšších rýchlostí, ale aj nových a lepších riešení. Paralelné genetické algoritmy sú taktiež bližšie k skutočným pomerom v prírode ako ich sekvenčné náprotivky. Táto práca popisuje najpoužívanejšie spôsoby paralelizácie genetických algoritmov. Následne ponúka návrh a implementáciu v jazyku Python. Nakoniec je implementácia overená vo viacerých testovacích scenároch.

KĽÚČOVÉ SLOVÁ

AMQP, Celery, genetický algoritmus, hierarchický model, hrubozrnný model, jemnozrnný model, master-slave model, paralelizácia, Python, RabbitMQ, SCOOP

ABSTRACT

Genetic algorithms are modern algorithms intended to solve optimization problems. Inspiration originates in evolutionary principles in nature. Parallelization of genetic algorithms provides not only faster processing but also new and better solutions. Parallel genetic algorithms are also closer to real nature than their sequential counterparts. This paper describes the most used models of parallelization of genetic algorithms. Moreover, it provides the design and implementation in programming language Python. Finally, the implementation is verified in several test cases.

KEYWORDS

AMQP, Celery, coarse-grained model, fine-grained model, genetic algorithm, hierarchical model, master-slave model, parallelization, Python, RabbitMQ, SCOOP

TULEJA, Martin. *Genetické algoritmy – implementace paralelního zpracování*. Brno, 2018, 77 s. Diplomová práca. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací. Vedúci práce: Ing. Václav Oujezský, Ph.D.

VYHLÁSENIE

Vyhlasujem, že som svoju diplomovú prácu na tému „Genetické algoritmy – implementace paralelního zpracování“ vypracoval(a) samostatne pod vedením vedúceho diplomovej práce, využitím odbornej literatúry a ďalších informačných zdrojov, ktoré sú všetky citované v práci a uvedené v zozname literatúry na konci práce.

Ako autor(ka) uvedenej diplomovej práce ďalej vyhlasujem, že v súvislosti s vytvorením tejto diplomovej práce som neporušil(a) autorské práva tretích osôb, najmä som nezasiahol(-la) nedovoleným spôsobom do cudzích autorských práv osobnostných a/alebo majetkových a som si plne vedomý(-á) následkov porušenia ustanovenia § 11 a nasledujúcich autorského zákona Českej republiky č. 121/2000 Sb., o práve autorskom, o právach súvisiacich s právom autorským a o zmene niektorých zákonov (autorský zákon), v znení neskorších predpisov, vrátane možných trestnoprávnych dôsledkov vyplývajúcich z ustanovenia časti druhej, hlavy VI. diel 4 Trestného zákoníka Českej republiky č. 40/2009 Sb.

Brno ....

podpis autora(-ky)

POĎAKOVANIE

Rád by som poďakoval vedúcemu diplomovej práce pánovi Ing. Václavovi Oujezskému, Ph.D. za odborné vedenie, konzultácie, trpezlivosť a podnetné návrhy k práci.

Brno ....

podpis autora(-ky)

OBSAH

ZOZNAM OBRÁZKOV

ÚVOD

Táto práca sa zaoberá návrhom a implementáciou paralelného spracovania genetických algoritmov. Genetické algoritmy sú triedou moderných algoritmov vzniknutých inšpiráciou z prírody, označovaných ako evolučné algoritmy. Spôsob fungovania týchto algoritmov ich predurčuje na paralelné spracovanie. Práve paralelizácia je obľúbenou metódou zrýchľovania nielen genetických algoritmov. Genetické algoritmy však prinášajú viacero spôsobov paralelizácie, ktoré prinášajú ďalšie výhody, ale aj nevýhody pre riešenie optimalizačných problémov.

Cieľom tejto práce je popísať najpoužívanejšie spôsoby paralelizácie genetických algoritmov a následne využiť výstupy teoretickej časti pre návrh implementácie. Ako implementačný jazyk pre genetický algoritmus bol vybratý Python, preto aj návrh je realizovaný so zreteľom na tento jazyk. Návrh tak obsahuje taktiež prehľad rôznych spôsobov paralelizácie v tomto jazyku. Tie sú detailne popísané a porovnané. Následne sú zakomponované do celkového návrhu.

V teoretickej časti tejto práce je popísaný všeobecný sekvenčný genetický algoritmus. Je popísané jeho fungovanie, základne operátory a trieda problémov, ktoré algoritmus rieši. Jadrom tejto práce je prehľad a detailný popis rôznych spôsobov paralelizácie genetických algoritmov. Následne sú popísané ich výhody, nevýhody a taktiež aj ich porovnanie medzi sebou.

Praktická časť sa zaoberá návrhom a implementáciou paralelizácie genetických algoritmov. Úvod tejto časti predstavuje teóriu všeobecných princípov paralelizácie. Výstupy z tejto časti sú použité pri samotnom návrhu. Návrh obsahuje prehľad rôznych Python modulov poskytujúcich paralelné spracovanie. Tie sú následne ohodnotené a výber je zúžený na štyri moduly, ktoré sú ďalej popísane a porovnané.

Praktická časť ďalej obsahuje popis realizácie návrhu. Implementácia je detailne popísaná z hľadiska paralelizácie, komunikácie a z hľadiska genetického spracovania, kde je implementácia popísaná pre každý model paralelného genetického algoritmu zvlášť. Súčasťou praktickej časti je nakoniec aj testovanie výsledného modulu. Je popísaný testovací scenár a prostredie, v ktorom testovanie prebiehalo. Nakoniec sú uvedené získané dáta, ktoré sú následne vyhodnotené.

1 TEÓRIA GENETICKÝCH ALGORITMOV

Táto časť detailne popisuje všeobecný sekvenčný genetický algoritmus. Uvádza dôležité termíny, ktoré budú použíté v ďalších častiach. Jadrom je však popis rôznych spôsobov paralelizácie genetických algoritmov.

1.1 Genetické algoritmy

Genetické algorimy (ďalej len GA) patria medzi evolučné heuristické (v niektorých prameňoch aj metaheuristické) stochastické optimalizačné algoritmy [1] [2] [3]. Pred detailným popisom budú popísané triedy algoritmov, ktorých sú GA súčasťou.

Podľa autora [4] patria GA medzi algoritmy strojového učenia, pričom využívajú učenie založené na indukcii a empírii. GA si generujú vlastné riešenia pomocou svojich operátorov, kde vyhodnotia doterajšie riešenia (empíria) a ak je ich pravdivostná hodnota dostatočná, použijú sa ako základ pre generovanie nových (indukcia). Z tohto hľadiska sú GA „príbuzné“ algoritmom umelej inteligencie (AI).

Trieda evolučných algoritmov je súborom algoritmov, ktoré sú založené na princípoch fungovania evolúcie. Algoritmy využívajú teóriu publikovanú roku 1859 britským prírodovedcom Charlesom Darwinom v diele *O pôvode druhov prírodným výberom, čiže uchovaním prospešných plemien v boji o život* [5]. Aj keď prvotné náznaky využitia princípu evolúcie v informatike boli už v 50. rokoch minulého storočia, kvôli nedostatočnej metodike a v tej dobe ešte slabému hardvérovému výkonu bol výskum tejto oblasti v priebehu ďalších rokov minimálny [1]. Prvým signálom bola publikácia od amerického profesora psychológie, elektrotechniky a informatiky Johna Hollanda *Adaptation in Natural and Artificial Systems, (Adaptácia v prirodzených a umelých systémoch)* v roku 1975 [6].

Evolučné algoritmy sú skôr všeobecnou definíciou, okrem spomínaných genetických algoritmov zahŕňajú taktiež evolučné stratégie a evolučné a genetické programovanie. Evolučné stratégie sú jednými z prvých úspešných stochastických algoritmov (60. roky minulého storočia). Nevyužívali princípy evolúcie tak striktne ako napríklad genetické algoritmy (prvé algoritmy používali len operátory mutácie). Namiesto binárnej reprezentácie taktiež používajú vektory reálnych čísel. Genetické a evolučné programovanie je rozšírením genetických a evolučných algoritmov, kde sa namiesto bežnej binárnej reprezentácie dát používajú hierarchicky štruktúrované programy (funkcie). Výsledkom evolúcie je tak program, ktorý rieši nejaký problém (napríklad pohyb mravca, ktorý prijíma príkazy, po bludisku).

Ďalšou triedou, do ktorej patria genetické algoritmy, je množina algoritmov riešiacich optimalizačné problémy. Optimalizačný problém je vo všeobecnosti problém nájdenia najlepšieho riešenia zo všetkých možných. Optimalizačné problémy

sú diferencované podľa množiny z ktorej vychádzajú ich premenné. Ak je množina konečná (alebo aspoň spočitateľná) a diskrétna, problémy spadajúce do tejto kategórie sú kombinatorické (problém batohu, problém obchodného cestujúceho). Naopak, ak je množina spojitá a nekonečná, patria do nej minimalizačné, resp. maximalizačné problémy.

Medzi optimalizačné problémy patria aj NP problémy, ktoré sa (údajne) nedajú riešiť v polynomiálnom čase pomocou deterministických algoritmov. Minimalizačné, resp. maximalizačné problémy sa riešia rôznymi gradientnými (aj negradientnými) optimalizačnými metódami. Niektoré problémy z tejto oblasti sa však takto riešia veľmi obtiažne, preto sa často pristupuje k evolučným algoritmom [2]. Takisto aj v oblasti kombinatorických problémov sa vyskytujú problémy, ktoré patria do množiny NP problémov a preto ich nie je možné riešiť deterministickými algoritmami. Medzi možné náhrady tak patria aj GA [7]. GA sa označujú za stochastické, pretože obsahujú viaceré kvazináhodné operátory, napr. selekcia, mutácia [1].

Poslednou triedou algoritmov, do ktorej GA patria sú heuristické (resp. metaheuristické) algoritmy. Niektoré pramene radia GA medzi heuristické algoritmy [1] a niektoré zase medzi metaheuristické [7] [3]. Algoritmy riešiace optimalizačné problémy sa delia na dve skupiny - exaktné a heuristické [7]. Exaktné algoritmy dokážu (teoreticky) poskytovať optimálne riešenia problémov, kým heuristické a metaheuristické poskytujú iba suboptimálne. Zjednodušene to znamená, že negarantujú poskytovanie toho najlepšieho (a správneho) riešenia, čo ale neznamená, že ho poskytnúť nemôžu. Pri rôznych problémoch sa akceptujú kvôli dlhému výpočetnému času aj riešenia, ktoré síce nie sú správne ale blížia sa k tomu.

Metaheuristika nahradila v minulosti používaný pojem „moderná heuristika“ a definuje algoritmy, ktorých vyššia vrstva („meta“ znamená „za“ alebo na „vyššej vrstve“) riadí nižšiu (heuristickú) vrstvu, ktorá je často závislá od konkrétneho problému. Ich hlavným prínosom je zrýchlenie algoritmu a zlepšenie prehľadávania priestoru riešení.

1.1.1 Optimalizačné problémy

Pre pochopenie činnosti GA je potrebné najprv pochopiť problémy, na ktoré sa aplikujú. V predošlej kapitole 1.1 boli tieto problémy zjednodušene definované ako problémy, ktorých cieľom je nachádzať maximá, resp. minimá. Túto úlohu však spĺňa veľké množstvo algoritmov. Aby vynikli výhody GA je potrebné optimalizáciu popísať detailnejšie.

Úlohu minimalizácie (princíp maximalizácie je ten istý) je možné podľa [8] zapísať

v tvare:

$$f(x) \rightarrow min, x \in \mathbb{X} \tag{1.1}$$

Pričom f je účelová funkcia, $\mathbb{X}$ je množina prípustných riešení. Rozlišujeme globálne a lokálne minimum (obrázok 1.1): Bod $x^* \in \mathbb{X}$ sa nazýva

1. bodom globálneho minima f na $\mathbb{X}$ alebo tiež globálne riešenie úlohy (1.1) ak

$$f(x^*) \leq f(x), \forall x \in \mathbb{X} \tag{1.2}$$

2. bodom lokálneho minima f na $\mathbb{X}$ alebo tiež lokálne riešenie úlohy (1.1) ak $\exists \epsilon > 0$ také, že

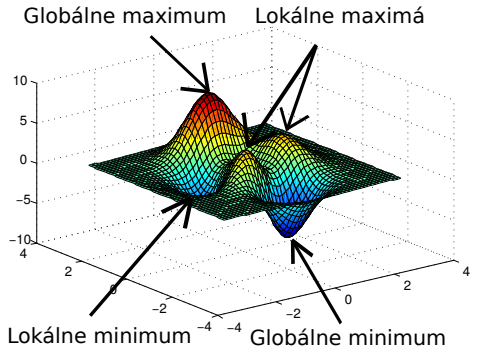
$$f(x^*) \leq f(x), \forall x \in \mathbb{X} \bigcap \mathcal{O}_\epsilon(x^*), \tag{1.3}$$

kde okolie $\mathcal{O}_\epsilon(x^*) = \{x \in \mathbb{R}^n \mid ||x - x^*|| < \epsilon\}$ je otvorená guľa s polomerom ϵ a so stredom v bode x^*.

Pre globálne riešenie úlohy (1.1) podľa [8] definujeme:

$$x^* = arg \min_{x \in \mathbb{X}} f(x) \tag{1.4}$$

Pre riešenie optimalizačných problémov sa používajú GA vtedy, ak neexistuje známe riešenie problému alebo ak sú známe riešenia nedostatočné (častý je problém „uviaznutia v lokálnom minime“, resp. „maxime“ alebo sú časovo náročné). Ich výhodou je, že pri generovaní nových riešení využívajú náhodnosť. Takto dokážu vytvoriť riešenia, ktoré prekonajú lokálne minimá. Problém „uviaznutia v lokálnom minime“ spočíva v tom, že pri dosiahnutí lokálneho minima, dané riešenie označíme za konečné, pretože v okolí $\mathcal{O}_\epsilon(x^*)$ už žiadne lepšie neexistuje (možno vidieť na obrázku 1.1). Evolučné algoritmy generujú veľký počet riešení, ktoré v spojení s náhodnosťou pokrývajú väčší priestor množiny prípustných riešení $\mathbb{X}$.



Obr. 1.1: Problém nájdenia globálneho maxima.

1.1.2 Činnosť genetických algoritmov

Genetické algoritmy sú najviac používanými evolučnými algoritmami v rámci stochastických optimalizačných algoritmov [2]. Z princípov evolúcie využívajú náhodný výber jedincov za účelom reprodukcie, vytváranie nových jedincov pomocou kríženia (a mutácie) a vytvorenie novej populácie na základe ohodnotenia jedincov. Na úvod je potrebné definovať základné pojmy v spojitosti s genetickými algoritmami, ktoré sú používané v odbornej literatúre [1] [2]. Genetický algoritmus pracuje nad populáciou

$$P = \{\alpha_1, \alpha_2, ..., \alpha_p\} \tag{1.5}$$

kde α_n je binárny vektor a p je kardinalita populácie P. Populácia je množina binárnych vektorov, ktoré reprezentujú jedincov populácie. V genetike sa označuje jedinec ako *fenotyp*. Jeho reprezentácia je označovaná ako *genotyp*. Každá bunka jedinca obsahuje určitý počet chromozómov (väčšinou viac ako jeden). Z praktických dôvodov sa však u GA predpokladá, že jedinec má iba jeden chromozóm. Takto možno označiť chromozóm ako ekvivalent genotypu. Chromozóm sa teda definuje ako binárny vektor α fixnej dĺžky k

$$\alpha = (\alpha_1, \alpha_2, ..., \alpha_k) \in \{0, 1\}^k \tag{1.6}$$

Kľúčovým pojmom v genetike je spôsobilosť (*fitness*), teda miera vhodnosti alebo reprodukčnej schopnosti jedinca. Je to teda reálne číslo, podľa ktorého možno predpokladať pravdepodobnosť toho, že sa jedinec bude rozmnožovať. Spôsobilosť definujeme ako účelovú funkciu nad množinou binárnych vektorov dĺžky k

$$f : 0, 1^k \to R \tag{1.7}$$

ktorá ohodnotí každý binárny vektor reálnym číslom. Úlohou GA je teda nájsť globálne minimum (alebo maximum) ako bolo spomenuté v 1.4. GA vylepšuje obyčajný stochastický algoritmus pridaním princípov evolúcie, ktoré narozdiel od slepého náhodného generovania bodov riadia smer vývoja populácie. To zabezpečujú operátory kríženia a selekcie, ktoré sú závislé od aktuálnej hodnoty spôsobilosti. Samotné kríženie by však mohlo viesť k jednotvárnosti jedincov a teda k predčasnej konvergencii algoritmu (a k nedostatočným riešeniam). Rozmanitosť v populácii tak zabezpečuje operátor mutácie.

Popis algoritmu

Táto časť uvádza implementáciu GA v pseudopascalovskej verzii (prevzatý a upravený z [2]) a ďalej ho detailne popisuje :

```
1  procedure Evolution(input t_max; output P_fin);
2  begin t:=0;
3      P:={náhodne generovaná populácia chromozómov};
4      while t < t_max and fitness(P) < f_ok do
5      begin t:=t+1; Q:=∅;
6          while |Q| < |P| do
7          begin ohodnoť každého jedinca a kvazináhodne vyber
8          dva rodičovské chromozómy a_1, a_2 ∈ P podľa fitness;
9              if random < P_repro then
10                 (a'_1,a'_2) = O_repro (a_1,a_2);
11             else (a'_1,a'_2) = (a_1,a_2);
12             Q:=Q ∪ {a'_1,a'_2};
13         end; P:=Q;
14     end; P_fin:=P;
15 end;
```

Algoritmus najprv náhodne inicializuje populáciu P, teda vygeneruje binárne vektory $a_1, a_2, ..., a_p$. Populácia sa následne obmieňa pomocou cyklu na riadku 4 počas t_{max} generácií alebo dokým ešte neexistuje dostatočné riešenie f_{ok}. Nová populácia vzniká z tej predošlej, a to reprodukciou jedincov. Najprv prebieha kvazináhodný výber párov (jedince a_1 a a_2), pričom sa ale často zohľadňuje spôsobilostná hodnota. Tu sa uplatňuje princíp prirodzeného výberu, teda schopnejší jedinci majú väčšiu šancu nájdenia partnera. Následne je na každý pár aplikovaný reprodukčný operátor O_{repro} s pravdepodobnosťou P_{repro}, čo taktiež uplatňuje princíp prirodzeného výberu, teda schopnejší jedinci majú väčšiu šancu reprodukcie. Operátor reprodukcie zahŕňa mutáciu a kríženie, ktoré budú detailnejšie popísané ďalej. Ak sa operátor neaplikuje, obaja jedinci z páru sa okopírujú do nových chromozómov, ako nové jedince. Novovzniknutý pár sa uloží do novej populácie a tento priebeh algoritmu od riadku 6 do riadku 13 sa opakuje až dokým nie je kardinalita novej populácie rovná kardinalite tej starej. Na konci každej generácie nová populácia nahradí tú aktuálnu. Po prebehnutí t_{max} generácií alebo dosiahnutí dostatočného riešenia sa proces ukončí a výsledok algoritmu predstavuje populácia P_{fin}.

Genetické operátory

Genetické operátory sú základnými prvkami GA a sú inšpirované z prírody. Patria medzi nich operátory mutácie, kríženia a selekcie. Operátor selekcie je jednoducho kvazináhodný výber jedincov, či už za účelom reprodukcie alebo migrácie (viď kapitola o paralelných GA 1.2). Je to kvazináhodný proces kvôli zohľadňovaniu spôsobilostnej hodnoty pri náhodnom výbere. Imituje tak prirodzený výber, kedy zdatnejší jedinci majú väčšiu šancu vzdorovať predátorom, chorobám ale taktiež aj konkurentom v boji o suroviny a tak žiť dostatočne dlho a splodiť potomstvo. Operátor mutácie imituje v genetike nachádzajúcu sa mutáciu, teda zmenu genetického materiálu DNA alebo RNA. Operátor kríženia imituje biologickú rekombináciu, teda vzájomnú výmenu častí chromozómov. Ďalej budú detailnejšie popísané operátory mutácie a kríženia, operátor selekcie je naozaj jednoduchý a v rámci tejto práce nepotrebuje detailnejší popis.

Operátor mutácie O_{mut} stochasticky modifikuje chromozóm α s pravdepodobnosťou P_{mut} na nový α':

$$\alpha' = O_{mut}(\alpha), \tag{1.8}$$

kde chromozómy α a α' sú definované ako binárne reťazce ako bolo definované v 1.6 a operátor modifikuje reťazec bit po bite, kde i-ty bit je modifikovaný podľa:

$$\alpha'_i = \begin{cases} 1 - \alpha_i & (\text{pre } random < P_{mut}) \\ \alpha_i & (\text{ostatné prípady}) \end{cases} \tag{1.9}$$

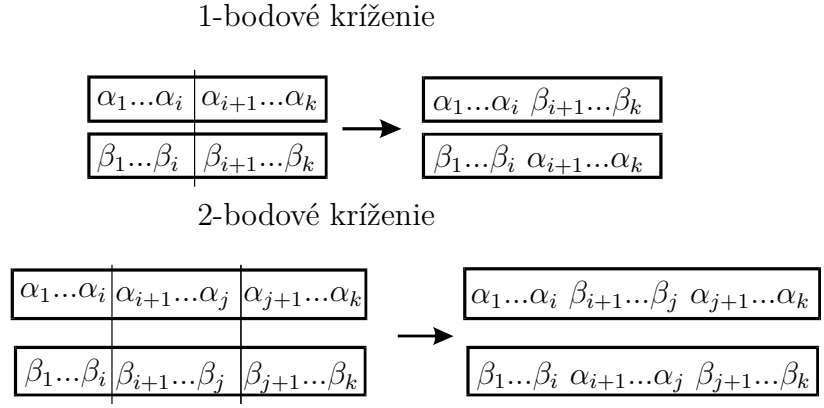
Každý bit je tak s pravdepodobnosťou P_{mut} invertovaný. V prípade, že $P_{mut} = 0$ operátor mutácie by binárny vektor vôbec nemenil. Mutácie vnášajú do populácie pozitívne, neutrálne alebo aj negatívne zmeny. Ich doménou je vnášanie rozmanitosti do populácie a zamedzenie predčasnej konvergencii populácie k nedostatočným riešeniam.

Operátor kríženia O_{cross} priradí dva nové $\alpha', \beta' \in \{0,1\}^k$ chromozómy pre dva kvazináhodne vybrané chromozómy $\alpha, \beta \in \{0,1\}^k$ z populácie:

$$(\alpha', \beta') = O_{cross}(\alpha, \beta) \tag{1.10}$$

Kríženie vymieňa jednu alebo viacero častí z chromozómu, pričom počet častí na výmenu je určený implementáciou alebo ponechaný na náhodu. Dĺžka a umiestnenie jednotlivých častí môže taktiež byť náhodná alebo kvazináhodná. Často sa využíva generovanie tzv. bodu (môže ich byť viacero) kríženia, ako možno vidieť na obrázku 1.2. Ten chromozóm rozdelí a jednotlivé časti následne vstupujú do kvazináhodného procesu, ktorý rozhodne, od ktorého rodiča sa bude dediť. Kríženie zabezpečuje výmenu výhodných vlastností jedincov a ich vzájomnú kombináciu. Práve tento operátor je jedinečný ako pre GA tak aj pre evolučné procesy v prírode. Pohlavná

reprodukcia je v prírode doménou vyspelejších druhov, čo implikuje jej víťazstvo v prirodzenom výbere a vďaka tomu máme empirický dôkaz, že táto metóda naozaj funguje a to nielen v prírode [2].



Obr. 1.2: Operácia kríženia nad dvoma chromozómami. Zvislá čiara znázorňuje bod kríženia [2].

Zhrnutie

Táto kapitola popisuje všeobecný GA, je však potrebné mať na zreteli, že existuje veľké množstvo modifikácii. Algoritmy sa môžu líšiť v generovaní populácie, implementácii spôsobilostnej funkcie, počte generácií, pravdepodobnosti mutácie, kríženia a selekcie. Nemali by však porušovať základné vlastnosti GA: stochastickosť a heuristickosť. Náhodnosť v selekcii, mutácii a krížení by mala byť zachovaná. Pravdepodobnosť mutácií by mala byť v intervale od 1 do 5 percent, pri krížení by to malo byť od 75 do 95 percent [1]. Mutácia by tak mala len zabraňovať jednotvárnosti populácie, pričom práve kríženie by malo určovať jej smerovanie. Existuje veľa rôznych modifikácii operátoru selekcie, avšak podľa autora [1] nemajú až taký vplyv na algoritmus ako spôsobilostná funkcia, ktorej by sa malo venovať najviac času. GA sú vnútorne paralelné, pracujú súčasne s celou populáciou. V prírode sa populácia môže deliť na rôzne kmene alebo národy, ktoré sú viacmenej izolované. Kmene sa následne môžu deliť na viacmenej izolované susedstvá, ktoré sú v konečom dôsledku tvorené viacmenej izolovanými jedincami. V prírode funguje evolúcia paralelne, preto prirodzene vznikli imitáciou z prírody paralelné genetické algoritmy.

1.2 Paralelné genetické algoritmy

Paralelizácia algoritmov je vhodným nástrojom pre zefektívenie a zrýchlenie algoritmu. GA sú veľmi vhodné pre veľkú množinu problémov, avšak niektoré z nich

vyžadujú väčšie množstvo času a takto sa GA stávaju pre nich nepoužiteľnými. Časovo najviac náročná operácia vrámci GA je spôsobilostná funkcia. Táto funkcia sa vykonáva pre každého jedinca a je nezávisla od ostatných. To ju predurčuje na paralelné spracovanie. Genetické operátory: mutácie a kríženie môžu takisto fungovať izolovane, keďže operujú nad jedným, resp. dvoma jedincami. Tieto operátory sú však omnoho jednoduchšie ako spôsobilostná funkcia, a preto je možné, že skonzumujú viac času komunikáciou ako by získali svojím paralelizovaným spracovaním [9]. Komunikácia je takisto problémom aj u ďalšieho genetického operátora - selekcie, ktorá často potrebuje informácie o celej populácii [9]. Preto sa bude ďalej uvažovať len paralelizácia spôsobilostnej funkcie.

Inšpiráciou z prírody sa vytvorili metódy, ktoré pracujú (paralelne) nad viacerými populáciami, resp. jednou populáciou, ktorá je rozdelená na viacero subpopulácií, tak ako vo svete je mnoho relatívne izolovaných populácií (kmeňov). Metódy paralelizácie GA sa tak líšia počtom populácií alebo typom architektúry, pre ktorú sú určené. Klasifikácia podľa [9], použitá vo väčšine odbornej literatúry [11], [12], [13] definuje 4 základné typy:

1. globálne jedno-populačné master-slave GA
2. jedno-populačné jemnozrnné GA
3. viac-populačné hrubozrnné GA
4. hierarchické GA

Jednotlivé modely, okrem toho, že sami definujú nejakú topológiu alebo architektúru, sú implementované na rôznych (hardvérových) architektúrach. V ďalších kapitolách sa bude vyskytovať pojem „uzol“ topológie alebo architektúry. Uzol tak môže byť jeden bod logickej architektúry, teda bod modelu, alebo bod fyzickej architektúry, teda výpočetná jednotka (napríklad procesor alebo počítač).

1.2.1 Globálny jedno-populačný master-slave model

Tento typ paralelizovaného GA pracuje len s jednou populáciou presne ako obyčajný GA, avšak s rozdieným spracovaním spôsobilostnej funkcie. Prívlastkom „globálny“ sa označuje kvôli operátorom selekcie a kríženia, ktoré sú aplikované nad celou populáciou.

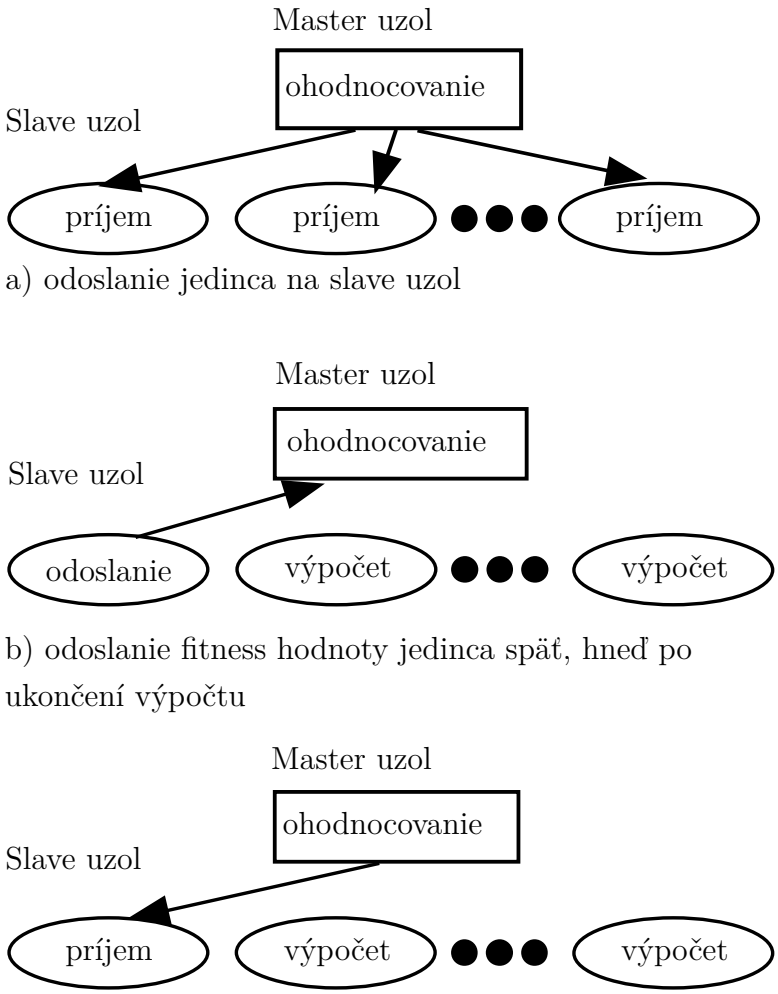
Spôsobilostná funkcia sa vykonáva paralelizovane na architektúre master-slave. *Master*, centrálny uzol architektúry (vo všeobecnosti býva v systéme len jeden) zohráva vedúcu úlohu. Pracuje nad celou populáciou tak ako obyčajný GA, avšak s rozdielným spracovaním spôsobilostnej funkcie, ktorej výpočet distribuuje medzi *slave* uzly. *Slave* uzly sú podriadené *master* uzlu a vykonávajú úlohy, ktoré im *master* uzol prikáže. V tomto prípade vypočítavajú spôsobilostnú funkciu pre jedného jedinca.

Sekvenčný spôsob vypočítavania fitnes funkcie pre $n, n \in \mathbb{N}$ jedincov v populácii, kde čas jedného výpočtu spôsobilostnej funkcie je $t_f, t_f \in \mathbb{R}_{>0}$, zaberie výsledný čas $t_v = t_f * n$. Pri paralelnom spracovaní (pri čase potrebnom na výmenú dát medzi *master* a *slave* uzlami $t_k, t_k \in \mathbb{R}_{>0}$) je výsledný čas $t_v = t_f + 2 * t_k$. Dôležitým faktom je, že vo vzťahu chýba parameter n, z čoho vyplýva, že paralelným spracovaním sa zbavujeme závislosti medzi veľkosťou populácie a časom potrebným pre výpočet. To však platí iba v prípade, že pre každého jedinca prislúcha jeden *slave* uzol. To sa pri veľkých populáciách v realite dosiahnúť nedá, preto sa v praxi kombinuje sekvenčné a paralelné spracovanie, kedy uzol, ktorý už dokončil výpočty a odoslal ich *master* uzlu, dostane vzápätí nové dáta ďalšieho neohodnoteného jedinca [10].

Ďalšie spracovanie v prípade, že dáta posledného, ešte neohodnoteného jedinca v populácii už boli odoslané, rozdeľuje globálne jedno-populačné master-slave GA na (rozdiel vo fungovaní možno vidieť na obrázku 1.3):

- synchrónne – *master* uzol začne odosielať dáta z novej populácie až po spracovaní tej predošlej na všektých *slave* uzloch
- asynchrónne – v prípade, že sú dáta všetkých jedincov z populácie už odoslané na *slave* uzly, *master* uzol začne odosielať dáta z novej populácie voľným uzlom

Synchrónne master-slave GA fungujú ako obyčajné GA, avšak rýchlejšie. Asynchrónne fungujú odlišne, a preto vyžadujú iný prístup pri modifikácii obyčajného GA na asynchrónny. Z toho dôvodu sú viac rozšírené synchrónne a ďalej sa bude pri globálnych jedno-populačných master-slave GA uvažovať len ich synchrónna varianta.



a) odoslanie jedinca na slave uzol

b) odoslanie fitness hodnoty jedinca späť, hneď po ukončení výpočtu

c) odoslanie ešte neohodnoteného jedinca (aktuálnej populácie, pre synchrónny GA alebo aj novej populácie, pre asynchrónny GA) na slave uzol

Obr. 1.3: Globálny jedno-populačný master-slave GA [10].

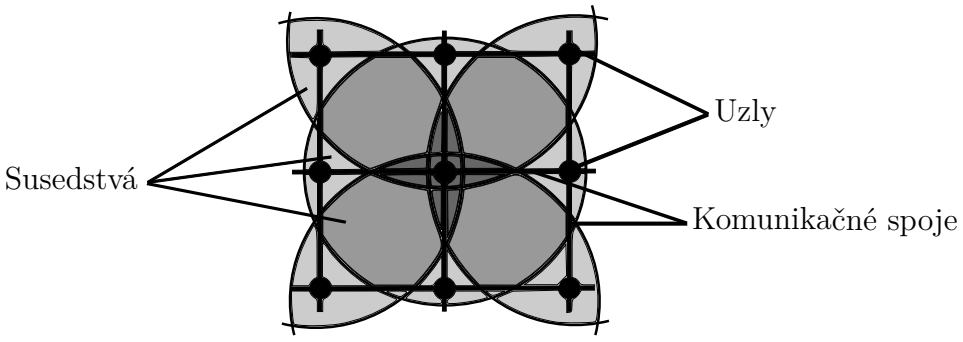
Z hľadiska implementácie môžu globálne jedno-populačné master-slave GA využívať zdieľanú pamäť, kam sa celá populácia uloží a *slave* uzly môžu načítavať a priamo zapisovať dáta jedincov, ktorí im boli priradení *master* uzlom. Druhou možnosťou je využiť distribuovanú pamäť, pričom populácia bude uložená na *master* uzle, ktorý bude rozposielať dáta *slave* uzlom a následne spracovávať už prijaté. Počet *slave* uzlov v architektúre master-slave býva rôzny a závisí na konkrétnom hardvéri a na konkrétnej aplikácii algoritmu. Väčší počet uzlov zosilňuje paralelizáciu a tým efektivitu a rýchlosť algoritmu. Na druhú stranu sa však pridávaním uzlov zvyšujú aj nároky na komunikáciu medzi *slave* a *master* uzlami. Preto býva často problém nájsť ich správny počet. Počet uzlov môže taktiež byť konštantný počas celého výpočtu GA alebo môže sa po čase meniť. Vo viacuživateľskom prostredí by správne ro-

zloženie záťaže (load-balancing) medzi procesormi zvýšilo efektivitu nielen GA ale aj celého systému. Tento typ GA získava na efektivite ak máme k dispozícii väčší počet procesorov a k tomu populáciu, s veľkosťou odpovedajúcou ich počtu [10]. V prípade, že počet procesorov je menší, nasadením systému na rozloženie záťaže môže algoritmus znovu získať na efektivite. V prípade menšieho počtu procesorov algoritmus stráca na efektivite kvôli existencii *master* uzlu a k tomu prislúchajúcej komunikácii medzi ním a *slave* uzlami. Azda najväčšou výhodou tohto GA je, že nemodifikuje fungovanie obyčajného GA a preto je veľmi jednoduché pre užívateľa modifikovať svoj obyčajný GA a aplikovať na neho master-slave architektúru.

1.2.2 Jedno-populačný jemnozrnný model

GA pracuje s jednou globálnou populáciou, ktorá je priestorovo rozptýlená do uzlov, pričom ich umiestnenie určuje topológia, ktorá je rozdelená do viacerých subpopulácií, resp. susedstiev. Rozptýlenie populácie je „jemnozrnné (fine-grained)“, teda populácia je rozptýlená do veľkého počtu uzlov. Jeden uzol tak obsahuje iba malý počet jedincov, väčšinou iba jedného alebo dvoch. Všetky uzly sú identické (jediné v čom sa líšia je jedinec/jedinci, ktorého/ktorých reprezentujú) a sú prepojené s uzlami v ich susedstve. Spôsob prepojenia uzlov medzi sebou je definovaný topológiou jemnozrnného GA. Počet uzlov je oveľa väčší ako pri iných typoch pararelných GA, preto sa tento GA označuje aj prívlastkom „masívne paralelný“.

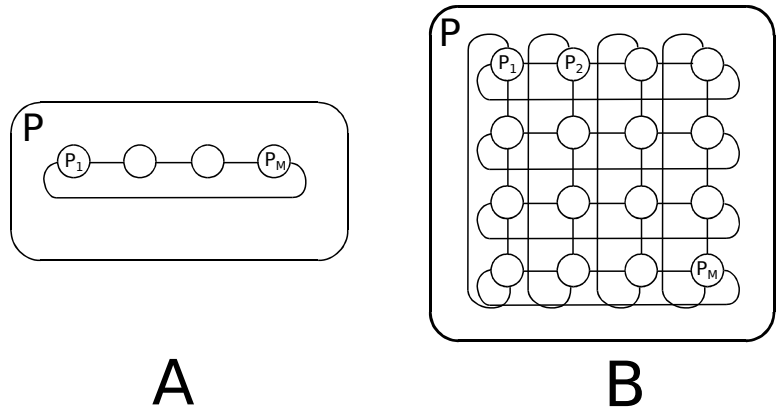
Susedstvá sú definované ako najbližšie okolie uzlu, kde definujeme jeden uzol ako centrálny. Každý uzol je centrálnym uzlom práve jedného susedstva ale zároveň môže byť súčasťou viacerých, z čoho vyplýva, že sa susedstvá možu prekrývať (ako je vidieť na obrázku 1.4). GA modifikuje svojou činnosťou iba centrálny prvok. Práve prekrývanie susedstiev zaručuje (a ovplyvňuje) šírenie jedincov s najlepšou spôsobilosťou. Spravidla platí, že rozľahlejšie susedstvá zaručujú väčšie prekrývanie a tým aj rýchlejšie šírenie jedincov s lepšou spôsobilosťou. Na druhú stranu sa však kvôli relatívnej izolovanosti susedstiev nejlepší jedinci nešíria tak rýchlo ako pri iných typoch GA, čo zvyšuje rozmanitosť populácie. Susedstvá pokrývajú celú topológiu uzlov a môžu mať rôzne tvary. Rôzne topológie uzlov a ich susedstiev implikujú aj rôzne chovanie GA.



Obr. 1.4: Prekrývanie susedstiev [11].

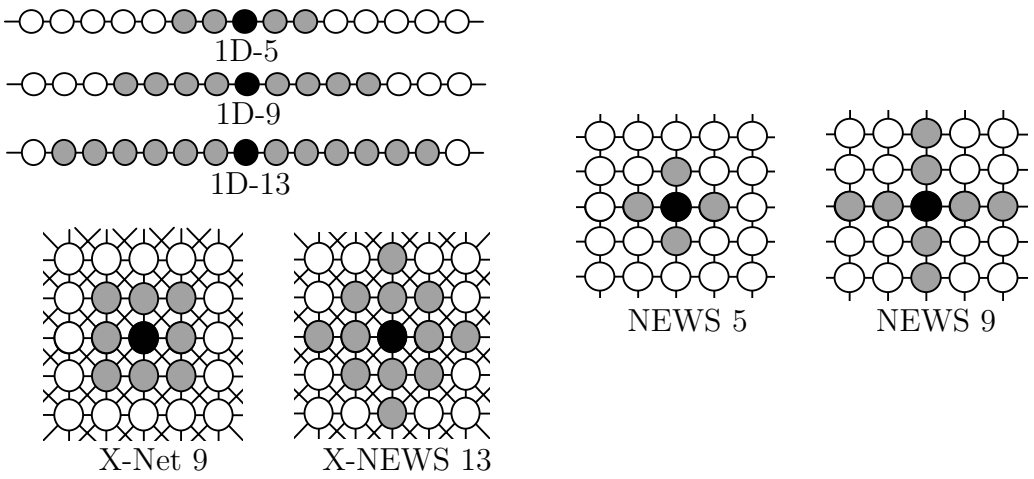
Susedstvá ohraničujú priestor, v ktorom môže operovať selekcia. Selekcia je takto oproti iným typom GA lokálna a paralelná (po susedstvách). Každý jedinec sa zúčastňuje procesu selekcie iba v rámci susedstiev, ktorých je súčasťou (ak je jedinec súčasťou 5 susedstiev, bude prechádzať 5 procesmi selekcie). Ako bolo spomenuté, v rámci jedného susedstva je modifikovaný (krížením a mutáciou) len jeho centrálny prvok.

Často používané topológie sú jedno a dvoj-rozmerné mriežky, ktoré sú často používané aj pre rozmiestnenie výpočetných prvkov paralelných počítačov, čo je aj dôvodom prečo ich GA používajú [9]. Vylučuje sa ale použitie topológie zbernice [11]. Pri navrhovaní topológie sú častým problémom jej okraje. Kedže chceme aby boli všetky uzly prepojené s rovnakým počtom uzlov, musíme zabezpečiť, aby sa okrajové uzly (napríklad v prípade štvorcovej topológie sú to uzly, ktoré sú umiestnené na obvode štvorca) prepojili. Najčastejšie sa to rieší zakrivením priestoru, ako možno vidieť na obrázku 1.5. Z priamky sa tak stane kružnica, z dvojrozmernej mriežky sa stane toroid (teleso v priestore, získané rotáciou uzavretej rovinnej krivky okolo osi ležiacej v rovine krivky a zároveň nepretínajúc krivku).



Obr. 1.5: Diagram A ako príklad toroidovskej topológie v 1D a diagram B tiež ako toroidovská topológia ale v 2D [2].

Typ topológie potom ovplyvňuje aj rozvrhovanie susedstiev (rôzne tvary susedstiev je možno vidieť na obrázku 1.6).



Obr. 1.6: Rôzne susedstvá jednopopulačného jemnozrnného GA [11].

Paralelne sa vykonávajú všetky operátory GA, avšak spôsobom odlišným od toho, ktorý poznáme pri obyčajných GA. Ako bolo spomenuté, selekcia sa aplikuje len lokálne. Ostatné typy GA využívaju naproti tomu globálnu, centralizovanú a sekvenčnú selekciu, ktorá vyžaduje zhromaždenie veľkého počtu dát, čo môže viesť ku komunikačným problémom („bottleneck“). Mutácia je nezávislá od ostatných jedincov, preto môže byť vykonávaná na každom uzle zvlášť a to bez akejkoľvek komunikácie medzi uzlami. Kríženie, ako operátor nad dvoma jedincami, už bude vyžadovať komunikáciu, ktorej miera bude závisieť na hustote populácie a selekčného algoritmu.

Odlišné správanie genetických operátorov však ovplyvňuje chovanie algoritmu, a preto tento typ GA nemusí fungovať na konkrétnom probléme rovnako ako obyčajný GA. Túto nevýhodu prevažuje výhoda, ktorá spočíva v efektívnej, flexibilnej, škálovateľnej a „masívne paralelnej“ implementácii na hardvéri [11]. Uzly systému sú jednoduché, uniformné, ich komunikácia je lokálna a v pravidelných intervaloch.

Pri návrhu systému sa musí brať do úvahy aj počet jedinov na jeden uzol. Vyšší počet jedincov by mal za následok zrýchlenie algoritmu, na druhú stranu by však zvýšil aj zložitosť celého systému. Pri pridávaní jedincov do uzlov musíme brať do úvahy zvyšujúce nároky na pamäť, výkon procesoru a aj na komunikačnú infraštruktúru.

1.2.3 Viac-populačný hrubozrnný model

Tento model, tiež označovaný ako „distribuovaný“, „viac-kmeňový“ alebo „ostrovný“, pracuje nad viacerými populáciami alebo „kmeňmi (anglicky deme)“, pričom proces evolúcie prebieha na každej z nich asynchrónne a relatívne izolovane. Ak reprezentujeme každý kmeň ako jeden uzol, možno prehlásiť, že na každom uzle prebieha samostatný GA. Ich izolovanosť je však relatívna z dôvodu existencie migrácie. Ak migrácia povolená nebude, budú populácie úplne izolované. V opačnom prípade bude miera migrácie určovať aj mieru izolácie. Relatívna izolovanosť však prináša možnosť kombinácie rôznych parametrov GA, odlišných metód selekcie alebo aj reprodukčných operátorov. Fungovanie algoritmu je znázornené na obrázku 1.7.

Tak ako pri ostatných modeloch, aj tu hrá topológia, čiže usporiadanie uzlov, veľkú rolu. Väčšinou sa používajú topológie, ktoré sú statické, teda definujú sa na začiatku a po dobu vykonávania GA sa nemenia. Používajú sa najmä hyperkocky, toroidové mriežky a kruhy. Často sa však topológia upraví tak, aby kopírovala topológiu výkonných jednotiek paralelného počítača, keďže po hranách topológie medzi sebou uzly komunikujú [9]. Druhou možnosťou sú dynamické topológie, pri ktorých jedinci migrujú do kmeňov podľa nejakých pravidiel. Pravidlá sú nastavené tak, aby sa migráciou dosiahlo zlepšenie daného kmeňa. Pravidlá sledujú parametre ako napríklad rozmanitosť kmeňa alebo genotypová odlišnosť dvoch kmeňov. Špeciálnym typom dynamickej topológie je náhodná topológia, kde výber cieľového kmeňa pre migráciu je náhodný.

Podobnosť topológií hrubozrnných a jemnozrnných modelov implikuje podobnosť medzi týmito modelmi. Deliacu čiaru je však ťažko určiť. Jeden z možných spôsobov je porovnať počet uzlov a počet jedincov v jednom z nich [13]. Ak je počet uzlov väčší ako počet jedincov v jednom z nich, tak sa jedná o jemnozrnný model, v opačnom prípade je to teda hrubozrnný.

Migrácia jedincov medzi kmeňmi je určená viacerými parametrami: topológiou,

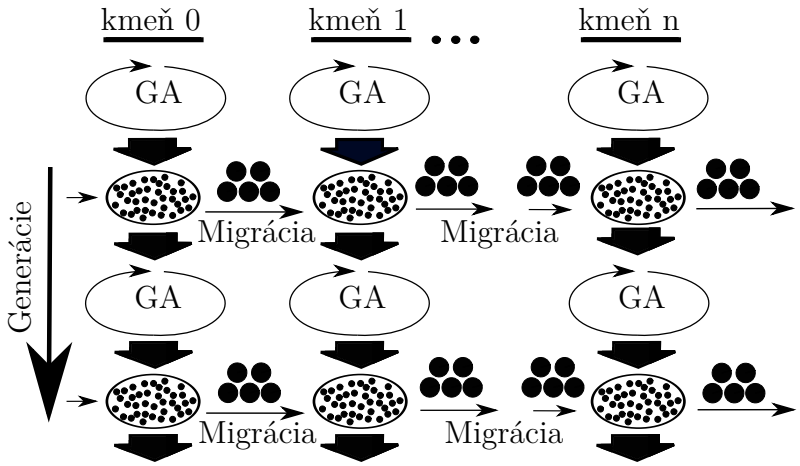
migračnou mierou a migračným intervalom. Topológia definuje prepojenia medzi kmeňmi, miera migrácie určuje počet jedincov, ktoré budú migrovať a interval určuje ako často migrácia nastane. Jedinci s väčšou spôsobilosťou sa šíria rýchlejšie v menších kmeňoch ako vo väčších. Ich šírenie je teda rýchlejšie ako v prípade obyčajného alebo master-slave jednopopulačného GA.

V prípade bez použitia migrácie sú riešenia menej optimálne ako pri jednopopulačných GA. Pri nízkych úrovniach migrácie sa riešenia k nim približujú, použitie migrácie tak (zdanlivo) výrazne nezlepšuje riešenia. So zvyšujúcou sa mierou migrácie je optimálnosť riešení porovnateľná s tými, ktoré možno pozorovať pri jednopopulačných. V niektorých prípadoch sú riešenia dokonca lepšie [9]. Nevýhodou rozdelenia jednej populácie na kmene je riziko predčasnej kovergencie, ktorá produkuje suboptimálne riešenia (riešením môže byť povolenie migrácie až po konvergencii ako je uvedené v [9]).

Po skončení migrácie sa navýši počet jedincov v niektorých kmeňoch. Problém v prípade statických kmeňov je zrejmý, v prípade dynamických je problémom možné zväčšovanie populácie do nekonečna (alebo do skončenia algoritmu). Existujú rôzne metódy, napríklad migrácia najlepších jedincov a vylúčenie najhorších. Každá metóda má však odlišný vplyv na chovanie algoritmu.

Práve pridané parametre tohto modelu komplikujú jeho aplikáciu a dementujú jeho zdanlivo jednoduchú aplikáciu ako implementáciu viacerých sériových GA na viacerých uzloch. Chovanie GA je (podobne ako u jemnozrnných GA) zmenené a nedá sa jednoducho aplikovať modifikáciou obyčajných GA. Každý kmeň môže byť odlišne nastavený a tak môže konvergovať rôznymi spôsobomi. To je však v prvom rade výhodou tohto modelu a to z dôvodu zvýšenia rozmanitosti, ktorá dovoľuje algoritmu preskúmať rôzne časti prehľadaváneho priestoru. Rozmanitosť taktiež zabraňuje predčasnej konvergencii, ktorá by mohla priniesť suboptimálne riešenia.

Výhody, ktoré z toho plynú sú tak výrazné, že niektorí autori odporúčajú implementovať tento model GA aj na štandardnom sekvenčnom procesore [1]. Pri paralelnej implementácii sa odporúča použiť architektúru s distribuovanou pamäťou. Hlavným dôvodom pre tento výber je problém so synchronizáciou všetkých uzlov, keďže na každom prebieha samostatný a relatívne izolovaný GA [12].



Obr. 1.7: Viac-populačný hrubozrnný model [14].

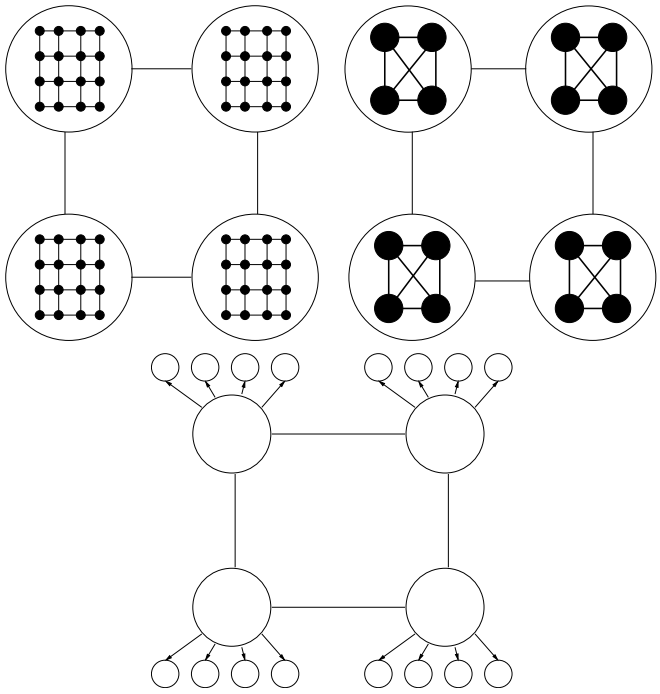
Relatívna izolovanosť každého kmeňa umožňuje vznik **heterogénneho** GA, kde sa nastavenie každej populácie môže líšiť. Každý kmeň tak môže používať rôzne implementácie operátorov selekcie, kríženia, mutácie alebo aj spôsobilostnej funkcie. Model, ktorý obsahuje heterogénnu aj spôsobilostnú funkciu, sa označuje ako **injekčný** (z anglického injection) model. Tento model je vhodný pre problémy, kde je nutné hodnotiť väčší počet kritérií, pričom sa každý kmeň môže zamerať práve na jedno [15]. Čiastočné riešenia (jedinci s najlepšou spôsobilostnou hodnotou v rámci kmeňa) následne migrujú do iného kmeňa, kde sa adaptujú na iné kritérium. Zameranie kmeňa na jedno kritérium je dosiahnuté nastavením rôzných parametrov GA, ale hlavne špecifickou implementáciou spôsobilostnej funkcie. Každé kritérium sa musí logicky ohodnocovať rozličným spôsobom. Ak sa komplexný problém rozdelí na viacero kritérií, dosiahne sa to, že jedinec bude po migrácii cez všetky kmene komplexne adaptovaný na daný problém.

1.2.4 Hierarchický model

Vývoj paralelizácie GA priniesol kombináciou dvoch alebo viacerých typov paralelných GA nové možnosti do tejto oblasti v podobe hierarchického modelu. Kombinácia viacerých typov GA prináša prirodzene kombináciu benefitov každého použitého modelu a sľubuje lepší výkon ako keby sme použili tieto modely samostatne [9]. Na druhú stranu však dosahuje vyššie stupne komplexity, aj keď existujú modifikácie tohto modelu, ktoré si udržujú podobnú komplexitu ako štandardné typy. Prívlastok „hierarchický“ naznačuje, že vo vzťahu medzi použitými modelmi bude nejaká forma hierarchie. Najčastejšie sa používa dvojvrstvový model, kde je na vyššej vrstve niektorý z multipopulačných modelov. Vznik tohto modelu bol umožnený práve spôsobom paralelizácie hrubozrnných modelov, kde na každom uzle pracuje samostatný

GA. Model GA na úrovni uzlov sa môže líšiť od toho, ktorý je použitý na globálnej úrovni. Rôzne kombinácie možno vidieť na obrázku 1.8. Vo všetkých kombináciách sa na vyššej vrstve nachádza hrubozrnný model. Vľavo je to kombinácia s jemnozrnným modelom na nižšej vrstve. Na obrázku vpravo je kombinácia hrubozrnného modelu na oboch vrstvách, kde na tej nižšej je úroveň migrácie rýchlejšia a topológia je hustejšia ako na vyššej. Spodný obrázok reprezentuje kombináciu s master-slave modelom na nižšej vrstve. Kombinovať rovnaké modely GA v hierarchii pod sebou má zmysel ak v jednom z týchto modelov zmeníme niektoré parametre. Jedným z príkladov môže byť už spomenuté použitie nižšej vrstvy, ktorá je zároveň hustejšia (čo sa počtu uzlov týka), s vysokou frekvenciou výskytu migrácie a vyššej vrstvy, ktorá je nastavená opačne [9].

Hierarchické modely tak ukazujú svoj potenciál v efektivite pri viacprocesorovej architektúre. Tieto modely dokážu nájsť riešenia rovnakej kvality (a aj lepšej) ako master-slave, jemnozrnné alebo hrubozrnné modely, ale za menší čas.



Obr. 1.8: Rôzne typy hierarchického modelu [9].

1.2.5 Porovnanie paralelných modelov

Populárna klasifikácia [9] delí triedu paralelných GA na 4 typy. Je to však hrubé delenie a niektoré typy ležia na pomedzí alebo sa navzájom kombinujú. Podľa štúdií [9] a [16], ktoré prinášajú prehľad odborných článkov v oblasti paralelných GA,

možno konštatovať, že väčšina implementácií sa od seba líši a teda aj porovnávania rôznych modelov bývajú odlišné. Čo však možno povedať s istotou je, že paralelné GA prinášajú okrem zrýchlenia výpočtov, ktoré prináša paralelizácia sama o sebe, aj lepšie výsledky ako obyčajné sekvenčné jednopopulačné GA. Dokážu lepšie prehľadávať priestor riešení a nájsť lepšie riešenia a to v kratšom čase. Je dôležité však podotknúť, že nie všetky modely dosahujú týchto výhod. Master-slave model prináša len zrýchlenie algoritmu, prehľadávanie priestoru je však nezmenené. Na druhú stranu však poskytuje jednoduchú implementáciu. Ak je cieľom použitia paralelizácie v GA zrýchlenie už existujúceho GA, je master-slave model najlepšou voľbou. Pri použití iných metód by bolo potrebné venovať veľké úsilie nastavenia paralelného modelu na konkrétny problém tak, aby poskytoval aspoň také dobré výsledky ako pôvodný algoritmus.

Ak je cieľom použitia paralelizácie v GA aj zlepšenie prehľadávania priestoru, odporúča sa použitie jemnozrnného, hrubozrnného alebo hierarchického modelu. Aj keď sa tieto 3 modely používajú na jeden spoločný účel, majú medzi sebou veľa rozdielov.

Ako bolo spomenuté v kapitole 1.2.3, určiť deliacu čiaru medzi jemnozrnnými a hrubozrnnými modelmi nie je kvôli ich podobnosti vôbec jednoduché. Porovnať ich prehľadávanie priestoru a rýchlosť je taktiež obtiažnou úlohou. Zhrnutie rôznych štúdií v [9] uvádza rôzne porovnania týchto modelov a pritom konštatuje, že porovnanie nemôže byť absolútne, ale musí byť spravené so zreteľom na konkrétny optimalizačný problém, keďže niektoré štúdie uvádzajú, že jemnozrnný model je lepší, kdežto niektoré uvádzajú opak. Uvedená teoretická štúdia však konštatuje, že pri dostatočnom počte procesorov je jemnozrnný model rýchlejší (nezávisle od veľkosti populácie), pričom sa však nezohľadňovali komunikačné a pamäťové nároky. Hrubozrnný model je zrejme pomalší ako jemnozrnný ale prináša veľké množstvo možností implementácie pri injekčnom modeli.

Hierarckický model je označovaný za model, ktorý dokáže redukovať výpočetný čas oveľa viac ako ostatné modely a dokáže prinášať minimálne také dobré riešenia ako ostatné.

Každý model ponúka rôzne možnosti implementácie. Líšiť sa môže topológia, veľkosť populácií (a taktiež susedstiev pri jemnozrnnom modeli), implementácia genetických operátorov a migrácia.

Väčšinou sa využíva synchrónna migrácia, ktorá nastáva v rovnakých preddeterminovaných intervaloch. Asynchrónnu migráciu spúšťa nejaká udalosť. Príkladom môže byť implementácia, kde algoritmus zastaví migráciu, ak je už blízko konvergencie. Dalším možným príkladom môže byť opak, kedy migrácia začína, až keď je populácia kompletne konvergovaná (tak sa obnoví rozmanitosť populácie). Z toho nastáva dôležitá otázka, kedy je správne migrovať, koľko jedincov by malo byť mi-

grovaných a čo sa stane s jedincami, ktorý budú v novej populácii zrazu prebytočný (ak sa bude používať konštantný počet jedincov v populácii). Vo všeobenosti však príliš vysoká alebo nízka frekvencia migrácie môže mať negatívny dopad na kvalitu riešení. Zo spomínaných prehľadov rôznych modelov možno ale konštatovať, že modely s migráciou prinášajú vo všeobecnosti lepšie výsledky ako tie bez nej.

Štúdie, ktoré porovnávali rôzne topológie používali rôzne optimalizačné problémy a prišli s rôznymi výsledkami. To implikuje, že rôzne topológie sú optimálne pre rôzne optimalizačné problémy. Pri hrubozrnnom modeli niektoré štúdie dokonca uviedli, že tvar topológie nie je až taký dôležitý, kým je topológia husto prepojená a nie je príliš veľká, čo by spôsobilo nedostatočné „miešanie“ medzi jedincami. Vo všeobecnosti však menšie topológie konvergujú rýchlejšie avšak s rizikom nižšej kvality riešení. Nájdenie správnej veľkosti je teda pri implementácii dôležitým krokom.

Paralelizovanie GA je veľmi populárna metóda ako vylepšiť existujúci obyčajný GA algoritmus alebo vyriešiť problémy, ktoré obyčajné GA vyriešiť nedokážu. Paralelizácia evolučných procesov je viac príbuzná reálnym procesom evolúcie, ktoré možno pozorovať v prírode. Maximálne využitie paralelizácie je podmienené jej implementáciou na paralelizovanom hardvéri. Aj keď implementácia jemnozrnného, hrubozrnného alebo hierarchického modelu na sekvenčných výpočetných jednotkách neprináša veľké zrýchlenie, odporúča sa ju použiť na úkor obyčajného GA.

2 NÁVRH IMPLEMENTÁCIE V JAZYKU PYTHON

Táto časť je zameraná na návrh implementácie paralelného GA. Jedná sa o návrh, ktorý by zahŕňal implementáciu všetkých štyroch typov paralelného GA. Ako je zrejmé z teoretickej časti, návrh bude zahrňovať dve časti: návrh paralelného vykonávania procesov a návrh komunikácie medzi nimi. Ako implementačný jazyk bol vybraný jazyk Python vo verzii Python 3.6.3 s implementáciou typu CPython. Každý teoretický návrh implementácie bol overený prototypom v prostredí, ktoré pozostávalo zo štyroch pracovných staníc. Jedna stanica – operačný systém Fedora vo verzii 25 a kernel vo verzii *4.11.6-201* – virtualizoval 3 ďalšie stanice s operačným systémom Ubuntu 17.10 a kernelom vo verzii *4.13.0-16*.

2.1 Teoretický úvod

Paralelné spracovanie môže mať podľa [17] štyri rôzne podoby:

1. model zdieľanej pamäte
2. viac-vláknový model
3. model distribuovanej pamäte, resp. model posielania správ
4. model paralelných dát

Model zdieľanej pamäte poskytuje asynchrónny prístup k zdieľanej pamäti. Výhodou tohto modelu je šetrenie času a komunikačných zdrojov, keďže sa tu nevyžaduje žiadna komunikácia. Nevýhodou je obtiažné riadenie prístupu k pamäti, aby sa vyhlo konfliktom pri čítaní a zapisovaní.

Program môže pri svojej sekvenčnej časti využívať aj vlákna, ktoré sú vykonávané paralelne so sekvenčnou časťou. Viac-vláknový model taktiež často využíva zdieľanú pamäť. Preto dedí výhody a nevýhody modelu zdieľanej pamäti. Implementácia jazyka Python zvaná CPython (interpreter jazyka je implementovaný v jazyku C) využíva GIL - global interpreter lock, ktorý zabraňuje vláknam naraz vykonávať „bytecode“ (interpreter prekladá Python kód do „medzi-jazyka“ zvaného „bytecode“). Okrem rôznych výhod, ktoré poskytuje, však zamedzuje využitiu viac-vláknového modelu na viacerých procesoroch. Pre paralelné spracovanie na viacerých procesoroch prináša jazyk Python „procesy“[18]. Proces je vo všeobecnosti spustená inštancia programu. Vlákno je sekvenčný tok riadenia v rámci procesu. Ich hlavný rozdiel podstatný v tejto práci je, že vlákna medzi sebou zdieľajú pamäťový priestor, pričom procesy sú izolované.

Model distribuovanej pamäte predpokladá, že každý procesor má vlastnú pamäť. Takto môžu byť vykonávané viaceré úlohy ako keby boli vykonávané na viacerých

pracovných staniciach. Paralelizáciu vykonávania úloh a výmenu dát zabezpečuje programátor. Ako možno vyčítať z názvu tohto modelu, výmena dát spočíva v posielaní správ, ktoré tie dáta obsahujú. Správy možno posielať po sieti a tak zabezpečiť paralelné spracovanie na viacerých pracovných staniciach.

Model paralelných dát definuje dátovú štruktúru, nad ktorou pracuje viacero procesov. Rozdiel oproti modelu zdieľanej pamäti je ten, že každý proces operuje nad inou časťou dát. Delenie častí medzi procesy zabezpečuje programátor.

Ďalším delením modelov paralelizácie podľa [17] je delenie na základe dekompozície úloh:

1. doménová dekompozícia – dáta sú dekomponované medzi viacero procesov, ktoré vykonávajú tú istú úlohu
2. funkčná dekompozícia – problém je dekomponovaný medzi viacero procesov, pričom každý vykonáva niečo iné

2.2 Návrh paraleného výpočtu

V tejto časti bude popísaný návrh paralelizácie výpočtov vykonávaných v rámci štyroch modelov paralelného GA. Vo všeobecnosti ide o paralelizovanie všetkých uzlov, ktoré sa v topológii každého modelu nachádzajú. Keďže každý jedinec-uzol potrebuje rovnaké výpočty, jedná sa teda o doménovú dekompozíciu. Master-slave model potrebuje paralelizovať len výpočet spôsobilostnej funkcie. Pri ostatných modeloch je to paralelizovanie buď každého jedinca, kmeňa alebo celého GA. Každý model teda rozdelí úlohy podľa svojej špecifikácie. Jedna úloha tak môže byť výpočet spôsobilostnej funkcie nad jedným jedincom, výpočet celého GA nad kmeňom atď. Výsledky každej úlohy musia byť odoslané na jedno miesto, kde sa následne spracujú. Master-slave model pridelí vypočítanú hodnotu každému jedincovi. Ostatné modely zozbierajú výsledky od jednotlivých jedincov alebo kmeňov a hľadajú medzi nimi najlepšie riešenie.

Tieto úlohy boli v tejto práci simulované pozastavením činnosti na niekoľko sekúnd. Jemnozrnný, hrubozrnný a hierarchický model vyžadujú komunikáciu medzi jednotlivými uzlami, čo predstavujú v topológii hrany. Preto každá úloha obsahovala časť, kde si vymenila dáta s ostatnými a tie následne vypísala. Častým dôvodom komunikácie medzi uzlami je migrácia.

2.2.1 Python moduly

Moduly pre paralelizáciu v rámci jazyka Python boli čerpané z [17] a [19]. Z veľkého počtu možností boli vybraté štyri moduly, ktoré sú následne detailnejšie popísané.

Prvou a najjednoduchšou možnosťou bolo použitie vláken. Veľkou výhodou je tiež, že vlákna sú podporované v rámci štandardných knižníc jazyka Python. Možnosti paralelizácie sú však veľmi obmedzené, a preto, ako už bolo spomenuté, je výhodnejšie použiť pre paralelizáciu Python procesy, ktoré sú tiež súčasťou štandardných knižníc.

Najväčšie možnosti paralelizácie poskytujú moduly, ktoré podporujú aj paralelizáciu na viacerých pracovných staniciach. Na základe týchto kritérií boli vybraté moduly: Multiprocessing, Celery, PyCSP, SCOOP, Pyro4, Dispy, Rpyc a Disco. Posledné štyri moduly nie sú súčasťou užšieho výberu kvôli rôzným problémom, ktoré sa pri ich testovaní objavili. Išlo buď o neúspešnú inštaláciu a konfiguráciu modulov alebo o neúspešnú implementáciu testovacieho scenára. Konkrétne dôvody neúspechu neboli objasnené, ale možným dôvodom môže byť ich nekompatibilita s operačnými systémami pracovních staníc alebo s verziou jazyka Python. Moduly, ktoré boli spustiteľné sú detailnejšie popísane ďalej.

Modul Multiprocessing

Modul multiprocessing poskytuje podľa oficiálnej dokumentácie na [20] paralelné spracovanie pomocou procesov, ktoré môžu byť spustené na lokálnej ale aj vzdialenej pracovnej stanici. Poskytuje API podobné modulu poskytujúcemu vlákna. Jadrom tohto modulu je trieda *Process*, ktorá samozrejme reprezentuje proces.

Na výmenu objektov medzi procesmi sa využívajú triedy *Queue* a *Pipe*. Trieda *Queue* je v podstate fronta FIFO a *Pipe*, v preklade „rúra“, poskytuje 2 objekty – teda oba konce „rúry“, pomocou ktorých môžu dva procesy komunikovať, keďže táto trieda poskytuje obojsmernú komunikáciu.

Zdieľanie dát medzi procesmi zabezpečujú triedy *Value* a *Array*, kde prvá zabezpečuje zdieľanie jednej hodnoty a druhá celého poľa hodnôt. Druhým spôsobom zdieľania dát medzi procesmi je využitie triedy *Manager*, ktorá poskytuje flexibilnejší spôsob zdieľania ako predošlý. Je to vďaka podporovaniu ľubovoľných typov zdieľaných objektov a taktiež schopnosti pracovať cez sieť na viacerých pracovných staniciach. Paralelné vykonávanie úloh má na starosti trieda *Pool*, ktorá vytvorí zadaný počet procesov pre úlohu, ktorú je potreba vykonať.

Testovací scenár obsahoval dve pracovné stanice, na ktorých boli spustené dva Python programy. Na jednej pracovnej stanici bol spustený server, ktorý rozdeľoval prácu na jednotlivé úlohy, pričom klient na druhej pracovnej stanici tieto úlohy vykonával. Kód bol inšpirovaný návodom [21], pričom bol rozšírený o vykonávanie úloha na dvoch pracovných staniciach. Bola zachytená aj komunikácia medzi stanicami, pričom bolo zistené použitie TCP protokolu.

Modul Celery

Modul Celery poskytuje asynchrónnu frontu, do ktorej sa vkladajú úlohy pomocou posielania správ. Systém zabezpečujúci posielanie správ má v rukách užívateľ. Ten si môže vybrať z viacerých možností systémov zvaných „message broker“. Celery podporuje *RabbitMQ* (odporúčaný), *Redis* a *Amazon SQS*. Podľa informácií uvedených na oficiálnej stránke [22], sa Celery zameriava na operácie v reálnom čase, podporuje však aj plánovanie úloh.

Jednotky, ktoré vykonávajú zadané úlohy sa volajú „workers“. Podporujú vykonávanie asynchrónne aj synchrónne. Úlohy možno zadávať asynchrónne a distribuovane, teda z rôznych procesov na jednej pracovnej stanici ale aj z rôznych pracovných staníc v sieti, ak majú sieťové spojenie so systémom „message broker“. Úlohy sú potom vykonávané spomínanou výpočetnou jednotkou („worker“). Môže ňou byť proces na jednej pracovnej stanici alebo viacero procesov na rôznych pracovných staniciach v sieti, ktoré však, ako v predošlom prípade, musia mať sieťové spojenie so systémom „message broker“.

Ukladanie výsledkov zabezpečuje taktiež užívateľ. Pomocou modulu *SQLAlchemy* môže použiť databázy *Sqlite*, *Mysql*, *Postgresql* alebo *Oracle*. Ďalej môže použiť systém *Redis*, ktorý sa používa ako „message broker“ a aj ako databáza, „framework“ *Django*, „message broker“ *RabbitMQ*, databázu *Cassandra* alebo aj obyčajný súborový systém operačného systému.

Celery modul poskytuje triedu *Celery*, ktorá poskytuje spustiteľný objekt. Základnými parametrami, ktorými sa môže tento objekt konfigurovať je *backend*, *broker* a úloha, ktorá sa má vykonať. *Backend* definuje použitý systém na ukladanie a vracanie výsledkov a parameter *broker* definuje použitie systému „message broker“.

Spustením programu sa vytvorí úloha a umiestní sa do poradovníka vykonávaných úloh, ktorý je zabezpečovaný samotným modulom. Následne sa spustí jedna alebo viacero vykonávajúcich jednotiek „worker“ na jednom alebo viacerých pracovných staniciach. Každá jednotka následne sekvenčne vykonáva zadané úlohy. Spustenie viacerých jednotiek vyúsťuje do paralelného spracovania. Výsledky úloh sú nasledne uložené a odoslané naspäť pomocou systému definovaného pomocou parametru *backend*.

Testovací scenár bol inšpirovaný podľa návodu pre tento modul v [17] a [23]. Pridané bolo spracovanie úloh jednotkami „worker“ na viacerých pracovných staniciach v sieti. Podľa odporúčania v dokumenácii na oficiálnej stránke modulu [22] bol použitý „message broker“ *RabbitMQ* a „backend“ *Postgresql*. Keďže systém *RabbitMQ* bol už aj tak súčasťou testovacieho scenára, bol využitý aj na otestovanie komunikácie medzi jednotlivými úlohami pomocou Python modulu *pika*. Každá úloha najprv odoslala správu a následne čakala na prijatie správy od inej úlohy. Po zadaní

úloh sa spustili jednotky „worker“ na dvoch pracovných staniciach.

Modul PyCSP

Tento Python modul je založený na komunikácii sekvenčných procesov, ktorá je synchrónna a je zabezpečená rôznymi typmi kanálov [17]. Tieto typy sú: One2One (jednosmerný medzi dvoma procesmi), One2Any (viacsmerný medzi jedným a viacerými procesmi), a Any2One (viacsmerný medzi viacerými procesmi a jedným procesom).

Modul poskytuje objekt *Channel*, ktorý reprezentuje jeden kanál. Ten poskytuje objekty zápisu, čítania a ukončenia činnosti. Úlohe vykonávajúcej zápis sa priradzuje objekt zápisu a tak isto je to aj v ostatných prípadoch. Nakoniec, model poskytuje funkcie pre vykonávanie zadaných procesov (v terminológii tohto modulu sa označuje úloha ako proces): *Parallel*, *Spawn* a *Sequence*. Prvá funcia zabezpečuje paralelné synchrónne vykonanie procesov, druhá ich paralelné asynchrónne vykonanie a posledná ich sekvenčné a synchrónne vykonanie.

Testovací scenár bol inšpirovaný z [17] a rozšírený o vykonávanie cez sieť. Podľa oficiálnej dokumentácie [24] tento modul podporuje vykonávanie na viacerých pracovných staniciach. Vykonávanie zápisu a čítania cez kanál v rámci jedného programu bolo úspešné. Tak isto aj spustenie v rámci viacerých programov na jednej pracovnej stanici. Spustenie na viacerých pracovných staniciach však bolo neúspešné. Objasniť problém sa však nepodarilo.

Modul SCOOP

Scalable Concurrent Operations in Python (SCOOP), v preklade škálovateľné súbežné operácie v Python je modul, ktorý sa primárne používa na vedecké výpočty [17]. Poskytuje paralelizáciu úloh na heterogénnych uzloch (na jednej alebo viacerých pracovných staniciach [25]).

Podobne ako pri module Celery, aj tu sa definuje vykonávajúca jednotka ako „worker“ a systém zabezpečujúci komunikáciu medzi nimi ako „message broker“. V tomto prípade však všetky aspekty komunikácie zabezpečuje samotný modul. Modul poskytuje triedu *futures*, ktorá poskytuje tieto funkcie: *map*, *map_as_completed* a *mapReduce*. Všetky tri funkcie fungujú ako paralelná verzia funkcie *map*, ktorá je súčasťou štandardných knižníc jazyka Python. Prvá vracia generátor, ktorý iteruje cez výsledky, ktoré vznikli modifikáciou vstupov funkciou, ktorá bola zadaná. Druhá je skoro totožná, avšak vracia výsledky akonáhle sú dostupné. Posledná funkcia poskytuje paralelizáciu nejakého operátora (napríklad sčítania alebo odčítania) nad dátami, ktoré vráti funkcia *map*, ktorá je volaná už v rámci *mapReduce*.

Podľa oficiálnej dokumentácie v [25] sa SCOOP používa pre: evolučné algoritmy, *Monte Carlo* simulácie, dolovanie dát a problémy prechodu grafom.

Testovací scenár bol vymyslený bez predlohy, keďže implementovať paralelné vykonávanie v tomto module je naozaj jednoduché. Na zdrojovej vykonávacej jednotke bola zadaná funkcia *map* s 10 iteráciami. Druhá vykonávacia jednotka bola umiestnená na inú pracovnú stanicu. Tam bolo potrebné umiestniť funkciu, ktorá sa má vykonať a nainštalovať potrebné moduly aby SCOOP fungoval. Úlohy mali byť teda spustené na dvoch pracovných staniciach, pričom si mali vymeniť správy pomocou systému *RabbitMQ*. Test sa spustil jediným príkazom, ktorý definoval počet vykonávajúcich jednotiek a pracovné stanice, na ktorých mali byť úlohy vykonávané. Modul sa už potom postaral o vykonanie úloh nielen na zdrojovej stanici ale aj na vzdialenej a to bez zásahu užívateľa. Pri odchytení komunikácie medzi stanicami bolo zistené, že modul využíva pre komunikáciu protokol SSH. Odporúča sa nastaviť SSH pomocou systému verejných kľúčov, takto odpadne nutnosť pri každom spustení programu vkladať heslo.

2.2.2 Porovnanie modulov

V tejto časti boli popísané moduly využité pre paralelný výpočet. Výhoda modulu Multiprocessing je v tom, že je súčasťou štandarných knižníc jazyka Python, a preto nie je nutné inštalovať moduly tretích strán. Implementácia paralelného spracovania na viacerých pracovných staniciach však bola obtiažnejšia ako pri ostatných moduloch. Modul Celery poskytuje veľa možností a je využívaný renomovanými aplikáciami ako je napríklad Instagram, Mozilla add-ons a AdRoll [22]. Taktiež práca s vykonávajúcimi jednotkami „worker“ je jednoduchá a intuítivná. Nevýhodou je pomerne zložité vracanie výsledkov cez rôzne systémy tretích strán a nutnosť spúšťania vykonávajúcich jednotiek na každej pracovnej stanici zvlášť. Modul PyCSP poskytuje jednoduchú implementáciu úloh a komunikáciu medzi nimi. Problémom je však už spomínaná neúspešná implementácia scenára na viacerých pracovných staniciach. Posledný modul SCOOP prináša najjednoduchšiu implementáciu úloh zo spomínaných modulov. Najväčšou výhodou je centrálne spúšťanie vykonávania úloh na viacerých pracovných staniciach bez potreby zásahu užívateľa na všetkých staniciach.

2.3 Návrh komunikácie

Návrh paralelného výpočtu zahŕňal okrem iného aj posielanie úloh na pracovné stanice a taktiež posielanie výsledkov výpočtov späť na zdrojovú stanicu. Paralelné modely GA však vyžadujú komunikáciu aj počas výpočtu. Ide napríklad o posielanie jedincov pre migráciu pri hrubozrnnom a hierarchickom modeli a výmenu dát pre selekciu, mutáciu a kríženie v rámci susedstiev pri jemnozrnnom modeli. Každý model

obsahuje viacero uzlov, ktoré sú navzájom prepojené istým spôsobom, ktorý je definovaný topológiou modelu. Každý uzol môžno reprezentovať ako jednú úlohu, preto je potrebné navrhnúť systém, ktorý bude implementovať komunikačnú topológiu daného modelu.

Často využívaným modulom pre paralelné spracovanie úloh a vytváranie komunikačných topológií je *mpi4py* [17]. Tento modul bol v rámci tejto práce úspešne otestovaný, avšak distribuované vykonávanie úloh na viacerých pracovných staniciach zvláda obtiažne. Azda jediný návod na distribuované spracovanie [26] bol veľmi zložitý, predpokladal napríklad vytvorenie zdieľaných adresárov cez NFS. Práve kvôli zložitosti nebol tento modul vybratý pre návrh komunikácie.

Ďalšou možnosťou pri návrhu komunikácie je využitie niektorej z implementácií protokolu AMQP. Z viacerých možností si autor [17] vybral už spomínaný *RabbitMQ*. Podľa oficiálnej web stránky [27] ide o najpoužívanejší systém „message broker" s viac ako 35 000 aplikáciami po celom svete. Keďže bol využívaný už pri návrhu paralelného výpočtu, bude použitý aj pri návrhu komunikačnej topológie.

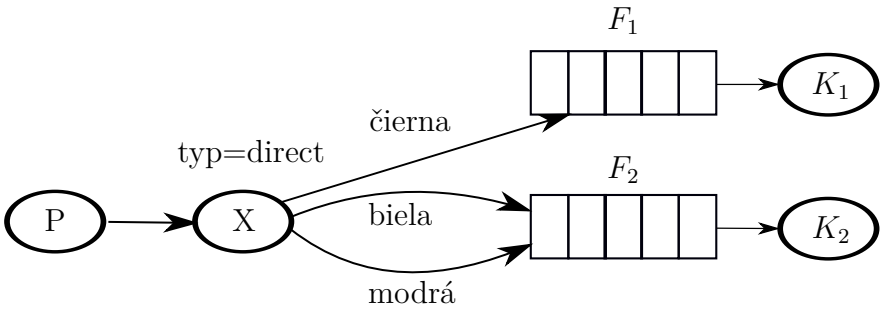
RabbitMQ

RabbitMQ je napísaný v jazyku Erlang a je založený na platforme OTP (Open Telecom Platform) [17]. Inštalácia, podľa oficiálnej dokumentácie na web stránke [27], je jednoduchá, predpokladá však nainštanovanú podporu pre jazyk Erlang. Podporuje operačné systémy Linux (Debian, Ubuntu, RHEL, CentOS, Fedora a Solaris), Windows a MacOS. V tejto práci bol inštalovaný na systémy Ubuntu a Fedora.

Inštalácia je nutná iba na jednom uzle, keďže systém je centralizovaný. Na ostatných uzloch stačí nainštalovať Python modul *pika*. Ten poskytuje objekt *connection*, ktorý reprezentuje spojenie s centrálnym uzlom. Vyžaduje parametre: IP adresu a prihlasovacie údaje. Následne poskytuje objekt *Channel*, ktorý zase poskytuje vytváranie front (pomocou *queue_declare*) a výmen (pomocou *exchange_declare*). Ďalej poskytuje funkcie *basic_publish* pre posielanie správ a *basic_consume* pre ich prijatie. V oficiálnej terminológii je odosielateľ správ označovaný ako producent (*producer*), prijímateľ zase ako konzumér (*consumer*). Odoslané správy sa pred odoslaním radia do fronty (*queue*). Producent však neodosiela správy priamo do fronty ale do výmen (*exchange*). Výmeny prijímajú správy od producentov a následne ich radia do front. Radenie má viacero typov, napríklad priame radenie (*direct*), radenie do všetkých front bez rozdielu (*fanout*) alebo radenie na základe tém (*topic*), ktoré sa označuje rovnako (*topic*). Producent tak na začiatku komunikácie definuje typ výmeny a spôsob radenia do front. Konzument následne definuje, z ktorých front chce správy prijímať.

Jednoduchým príkladom môže byť posielanie správ medzi jedným producentom

a dvoma konzumentami priamym radením do front. Prvý konzument chce prijímať len čierne správy. Druhý konzument len biele a modré. Výmena tak bude radiť správy na základe farby do dvoch front, každá pre jedného konzumenta. Tento scenár je možno vidieť na obrázku 2.1, kde producent je označený písmenom P, konzumenti písmenom K, fronty písmenom F a výmena písmenom X.



Obr. 2.1: Zjednodušený model RabbitMQ. Prebratý a upravený z oficiálnej dokumentácie na web stránke [27].

Návrh komunikácie pomocou RabbitMQ

Radenie správ do front môže mať využitie pri implementácii komunikačnej topológie paralelného modelu GA. Pri jemnozrnnom modeli by každý uzol prijímal správy z jednotlivých susedstiev, ktorých je súčasťou. Správy by sa teda radili do front na základe susedstiev a každý uzol by prijímal správy len z niektorých front. Takisto by uzol posielal správy len do tých front, ktoré reprezentujú susesedstvá, ktorých je súčasťou. Pri hrubozrnnom a hierarchickom modeli by každý kmeň komunikoval pomocou výmen len so susednými kmeňmi.

Testovací scenár tohto systému bol realizovaný na troch pracovných staniciach. Na jednej z nich bol spustený RabbitMQ a jeden z konzumentov. Na ďalších dvoch boli spustení producent a ďalší konzument. Boli otestované všetky spomenuté typy výmen. Taktiež bol otestovaný scenár, ktorý simuloval správanie RPC. Dáta posielané v správach boli objekty serializované podľa štandardu JSON. Na to bol použitý Python modul *json*. Všetky scenáre boli prebraté a upravené z oficiálnej dokumentácie na web stránke [27].

2.4 Zhodnotenie

V tejto časti bol popísaný návrh implementácie paralelného GA. Z teoretického hľadiska teda ide o implementáciu modelu distribuovanej pamäte, resp. modelu posielania správ, pomocou doménovej dekompozície. Pre implementáciu paralelného vykonávania úloh tu boli popísané viaceré Python moduly. Za najlepší bol

označený modul SCOOP. Spôsob použitia modulu SCOOP ho predurčuje na implementáciu master-slave modelu. Implementácia jemnozrnného a hrubozrnného modelu vyžaduje o trochu zložitejšiu implementáciu. Implementácia hierarchického modelu sá zdá byť veľmi obtiažnou až nemožnou z dôvodu potreby paralelizácie na dvoch úrovniach naraz. Pre implementáciu komunikácie bol označený za najlepšiu možnosť systém RabbitMQ.

Z existujúcich riešení možno spomenúť modul GAFT (Genetic algorithm framework in Python), ktorý poskytuje okrem štandardného výpočtu GA aj možnosť paralelizácie pomocou MPI modulov (MPICH alebo OpenMPI) [28]. Ďalším zaujímavým modulom je DEAP (Distributed Evolutionary Algorithms in Python) [29]. Tento modul poskytuje paralelizáciu pomocou už spomínaného modulu SCOOP. Poskytuje veľké množstvo funkcií avšak implementácia jedného z paralelných modelov je už na užívateľovi.

3 IMPLEMENTÁCIA PYTHON MODULU

Overením správnosti návrhu modulu je úspešná realizácia implementácie. V tejto časti je popísaná implementácia z troch hľadísk. Prvým hľadiskom bola paralelizácia a zakomponovanie modulu SCOOP do výsledného Python modulu. Druhým hľadiskom bola komunikácia v rámci topológie GA a tretím hľadiskom bola implementácia samotného modelu paralelného GA. Keďže sa jednotlivé hľadiská medzi sebou prelínajú, možno nájsť pri opise jednotlivých hľadísk poznámky aj k iným hľadiskám.

3.1 Popis modulu

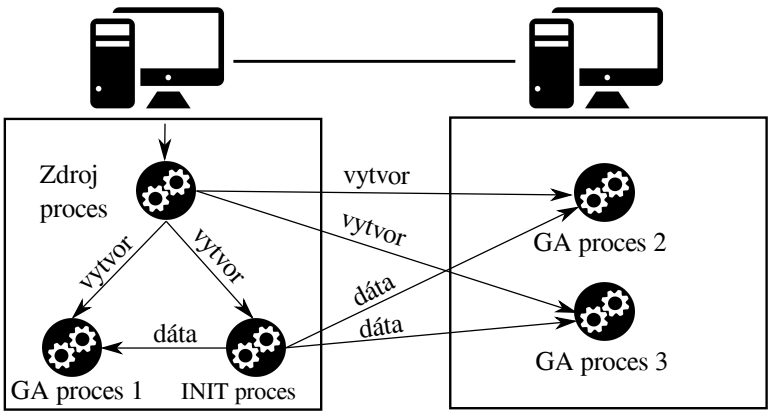
Implementácia navrhovaného Python modulu pre paralelizáciu GA je zhrnutá do modulu s názvom *parallel_ga_processing*. Modul bol vytvorený podľa návodu [31] pomocou Python nástroja *pip*. Modul *parallel_ga_processing* je možné vyhľadať medzi modulmi na oficiálnej stránke nástroja alebo priamo na odkaze [32]. Implementácia modulu využíva externé Python moduly SCOOP, Pika, Numpy a Jsonpickle. Ako bolo spomenuté, modul SCOOP sa používa pre paralelizáciu výpočtov a modul Pika pre komunikáciu s RabbitMQ serverom. Modul Numpy sa využíva pre vedecké výpočty v jazyku Python. V implementácii boli využité funkcie na prácu s maticami. Posledný externý modul Jsonpickle poskytuje funkcie na serializáciu a deserializáciu komplexných Python objektov do a z formátu JSON. Pri posielaní dát medzi procesmi sa dáta formovali do objektov, ktoré boli následne konvertované do textovej podoby vo formáte JSON. Prijatá správa bola na druhej strane zase konvertovaná do objektovej podoby. Nástroj *pip* sa postará pri inštalácii modulu *parallel_ga_processing* o inštaláciu všetkých závislostí, teda v tomto prípade o inštaláciu týchto 4 externých modulov.

Modul *parallel_ga_processing* obsahuje dva podmoduly *algorithmRunners* a *geneticAlgorithms*. Podmodul *algorithmRunners* poskytuje funkcie pre spúšťanie GA a podmodul *geneticAlgorithms* poskytuje základnú funkcionalitu GA. Užívateľ tak môže využiť Python triedy poskytované podmodulom *geneticAlgorithms* a implementovať si vlastný GA. Popisovaná štruktúra je štruktúrou Python modulu vytvoreného pomocou nástroja *pip*. Samotný kód obsahuje ďalšie adresáre, ktoré niesú poskytované užívateľom cez nástroj *pip*. Ide napríklad o adresáre s testmi alebo s príkladmi použitia. Tie sú potom prístupné spolu s celým kódom na portáli Git [33] alebo v prílohe B.

3.2 Implementácia paralelizácie

Paralelizácia GA bola implementovaná s využitím funkcií poskytovaných modulom SCOOP. Konkrétne sa jednalo o Python triedu *ScoopApp* a funkciu *map* zo SCOOP podmodulu *futures*. Trieda *ScoopApp* poskytuje vytváranie paralelných procesov na jednej alebo viacerých pracovných staniciach. Počet procesov a pracovných staníc, na ktorých sa majú procesy spustit, si definuje užívateľ. Každý proces spúšťa Python program na danej ceste v adresárovej štruktúre pracovnej stanice. Cesta k Python súboru, ktorý obsahuje daný program je taktiež definovaná užívateľom.

Z toho jasne vyplýva, že realizácia paralelizácie sa skladá z minimálne dvoch procesov. Princíp vytvárania procesov medzi dvoma pracovnými stanicami je ilustrovaný na obrázku 3.1. Prvý proces (Zdroj proces) spúšťa najprv INIT proces, ktorý vykonáva program umiestnený na danej ceste v adresárovom strome lokálnej stanice. INIT proces vytvára a rozďeľuje dáta medzi ďalšie procesy. Zdroj proces zatiaľ spúšťa aj zvyšné procesy, či už na lokálnej alebo vzdialenej stanici na danej ceste v adresárovom strome. INIT proces si najprv pridelí vlastnú porciu dát a ostatné porcie odošle na výpočet na zvyšné procesy. Po odoslaní prechádza INIT proces k výpočtu GA tak ako aj zvyšné procesy. Po skončení výpočtu zhromažďuje dáta z ostatným procesov. Po zhromaždení dát zo všetkých procesov INIT proces končí. Zdroj proces od začiatku čaká na ukončenie INIT procesu. V momente jeho ukončenia začne ukončovať aj zvyšné procesy na všetkých staniciach. Nakoniec modul SCOOP ukončí svoju aktivitu a Zdroj proces taktiež skončí.



Obr. 3.1: Princíp vytvárania viacerých procesov pomocou modulu SCOOP.

Implementácia paralelizácie sa nachádza v podmodule *algorithmRunners* v súbore *launcher*. V ňom sa nachádza len jedna funkcia s názvom *launch*. Funkcia obsahuje

tieto argumenty:

- *hosts_list* – zoznam pracovných staníc, na ktorých majú byť procesy spustené
- *num_of_workers* – počet procesov
- *path* – cesta v adresárovom strome k Python programu, ktorý má byť vykonaný
- *executable* – názov súboru, v ktorom sa Python program nachádza

Spustením tejto funkcie sa modul SCOOP pripojí pomocou SSH na každú pracovnú stanicu a spustí tam potrebný počet procesov, ktoré budú vykonávať zadaný Python program. Rozdelenie procesov medzi stanice sa nedá zo strany užívateľa ovplyvniť. Všetko, čo sa týka procesov zabezpečuje samotný SCOOP.

3.3 Implementácia komunikácie medzi procesmi

Komunikáciu medzi procesmi zabezpečuje server RabbitMQ. Pripojenie k serveru sa realizuje pomocou Python modulu Pika. Implementácia bola realizovaná pomocou príkladov implementácie na oficiálnej stránke dokumentácie pre modul Pika [34] a [35]. Oba príklady implementácie boli upravené tak, aby mohli fungovať spolu. Implementácia sa nachádza v podmodule *geneticAlgorithms* v súbore *messenger.py*. V súbore je definovaná Python trieda *Messenger*, ktorá dedí od triedy *Thread*. To značí, že inštancia tejto triedy sa má spúšťať ako vlákno.

Vytvorením vlákna sa v programovaní vo všeobecnosti zabezpečuje asynchronita. Jej použitie je veľmi rozšírené hlavne pri programovaní aplikácií, ktoré musia komunikovať cez sieť. V tomto prípade teda výpočet GA a komunikácia medzi procesmi bežia paralelne a nezávisle. Inštancia modulu tak môže v jednom vlákne vykonávať funkcie GA a v druhom vlákne prijímať a odosielať dáta ďalším procesom. Vlákno *Messenger* je interne implementované ako 2 na sebe nezávislé vlákna. Jedno vlákno sa používa na príjem dát, druhé vlákno na ich odosielanie.

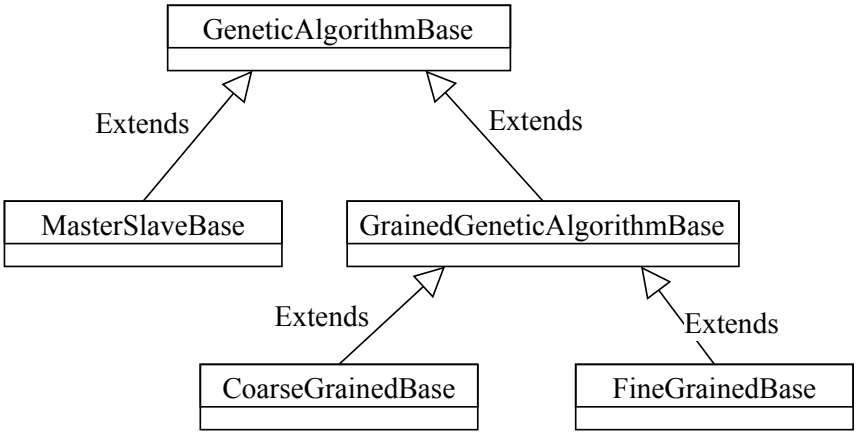
Dáta určené na odoslanie sa serializujú pomocou už spomínaného modulu Jsonpickle a jeho funkcie *encode*. Prijaté dáta sa následne deserializujú pomocou toho istého modulu a jeho funkcie *decode*. Správy sú prenášané vo formáte JSON a sú kódované do UTF-8 – bezstratového kódovania.

Inštancia Python triedy *Messenger* je ďalej označovaná ako „Komunikátor“.

3.4 Implementácia modelov paralelného GA

Implementácia modelov paralelného GA sa po zhodnotení obtiažnosti implementácie hierarchického modelu obmedzila na master-slave, jemnozrnný a hrubozrnný model. Implementácia modelov sa delí na dve časti. Prvou je implementácia samotného algoritmu a druhou je implementácia spúšťania algoritmu.

Implementácia algoritmu sa nachádza v podmodule *geneticAlgorithms* a jej štruktúra je ilustrovaná UML diagramom na obrázku 3.2. Hlavnými Python triedami sú triedy *MasterSlaveBase*, *CoarseGrainedBase* a *FineGrainedBase*. Pomocné triedy sú *GeneticAlgorithmBase* a *GrainedGeneticAlgorithmBase*. Slovo „base“ v názve tried označuje, že sa jedná o základnú triedu a tá môže byť ďalej modifikovaná užívateľom. Pomocná trieda *GeneticAlgorithmBase* poskytuje základne funkcie GA: inicializácia náhodnej populácie, generovanie náhodného chromozómu, kríženie, mutácia a pseudonáhodný výber jedincov na základe ohodnotenia jedincov. Pomocná trieda *GrainedGeneticAlgorithmBase* zase poskytuje spoločné funkcie pre jemnozrnný a hrubozrnný modul ako napríklad vytvorenie a zavretie spojenia s RabbitMQ serverom pomocou objektu *Messenger*.

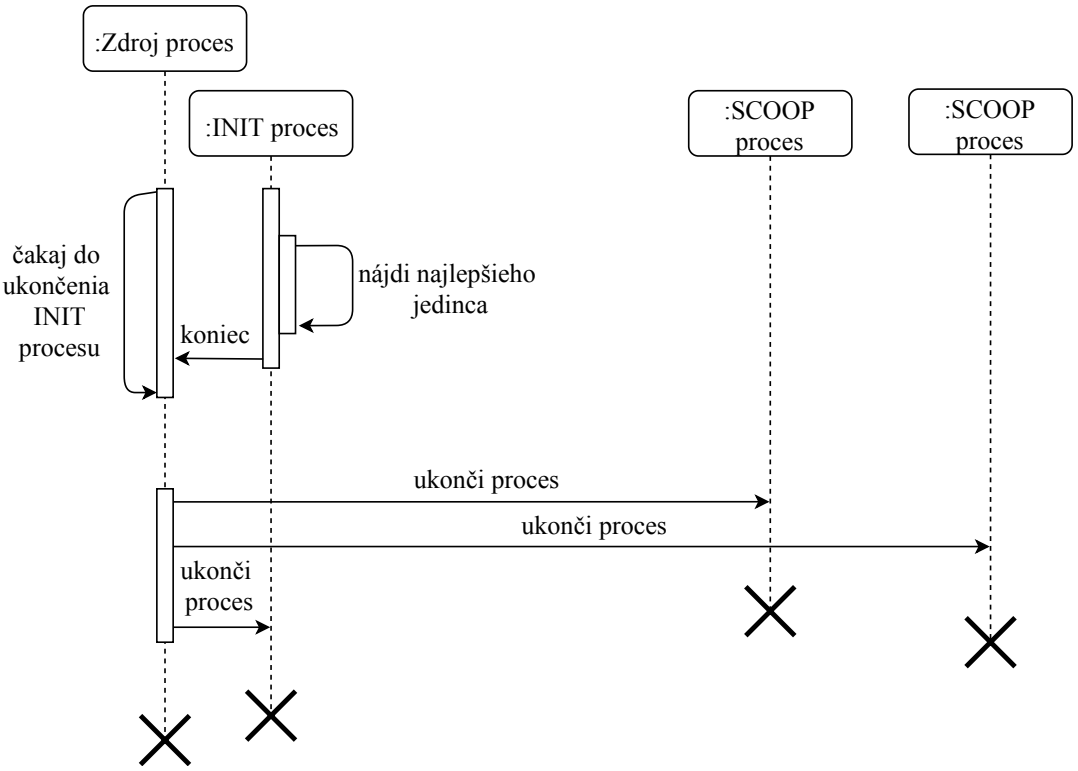


Obr. 3.2: UML diagram štruktúry tried GA.

3.4.1 Implementácia master-slave modelu

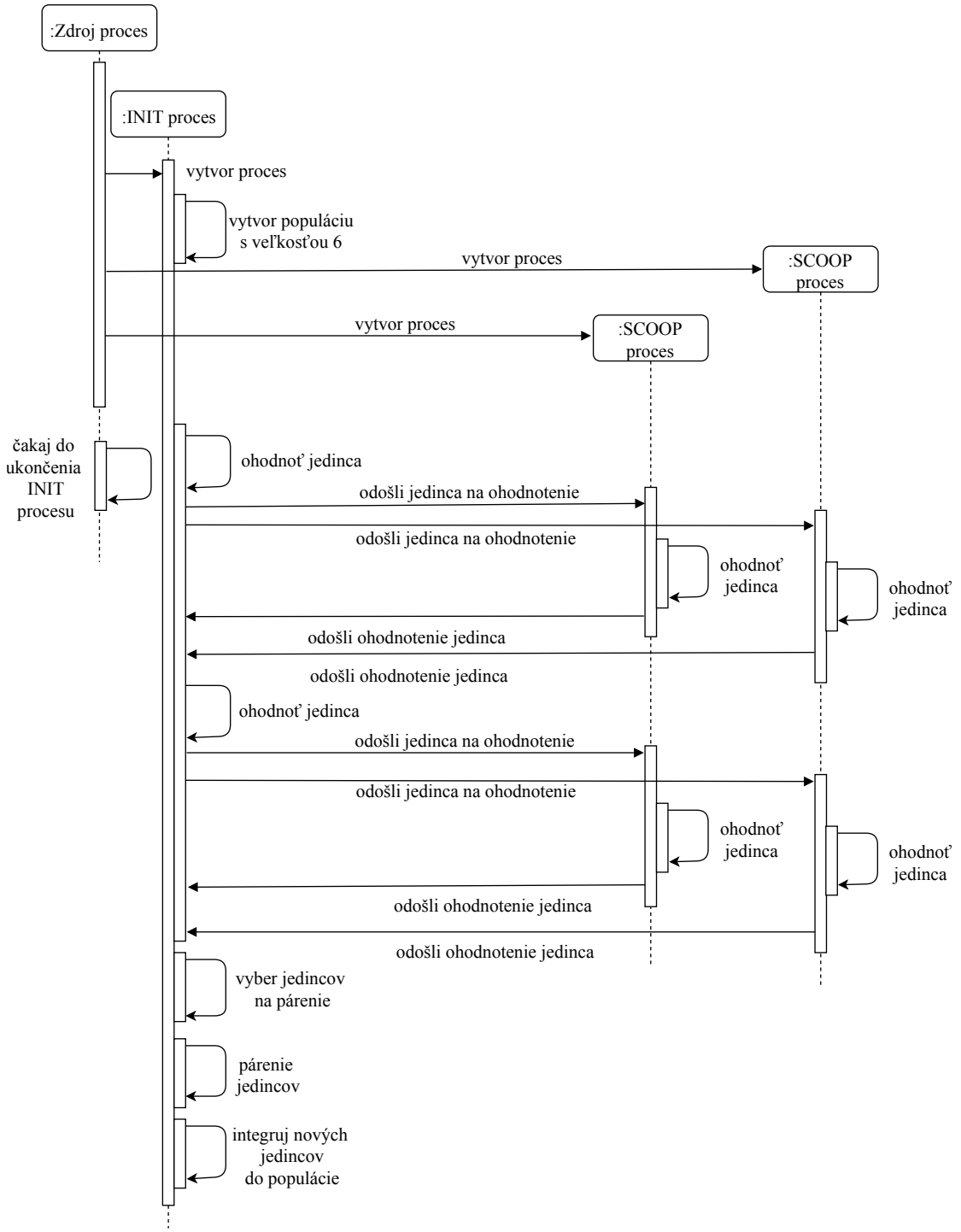
Implementácia samotného algoritmu je reprezentovaná Python triedou *MasterSlaveBase*. Tá dedí funkcionalitu z triedy *GeneticAlgorithmBase* a následne kopíruje priebeh klasického GA v tomto poradí:

1. ohodnotenie jedincov
2. pseudonáhodný výber jedincov na párenie
3. kríženie
4. mutácia
5. integrovanie nových jedincov do populácie
6. ak to ešte nebola posledná generácia, opakovanie priebehu od bodu 1
7. ukončenie priebehu modulu podľa diagramu 3.3



Obr. 3.3: UML diagram ukončenia master-slave modulu pre 3 uzly.

Prvý bod priebehu, teda ohodnotenie jedincov, prebieha paralelne pomocou SCOOP procesov. Využíva sa už spomínaná funkcia *futures.map*, ktorá odošle predložené dáta ostatným procesom na spracovanie. Dáta všetkých jedincov populácie su tak efektívne rozložené medzi všetky procesy na výpočet spôsobilostnej funkcie. Celkový priebeh štartu modulu (prvá iterácia) je ilustrovaný UML diagramom na obrázku 3.4. Využívanie funkcie *futures.map* a odosielanie dát má podľa diagramu na starosti INIT proces.



Obr. 3.4: UML diagram jednej iterácie master-slave modelu s 3 SCOOP procesmi.

Po skončení poslednej iterácie sa vyberie najlepší jedinec a prehlási sa za výsledné riešenie. V prípade nájdenia najlepšieho riešenia v skoršej iterácii sa populácia naplní správnym riešením a k páreniu jedincov už v ďalších generáciách nedôjde.

Spúšťanie algoritmu je implementované v podmodule *algorithmRunners* v súbore

masterSlaveGaRunner.py. Ten obsahuje funkciu *run_master_slave_ga*. Funkcia obsahuje tieto argumenty:

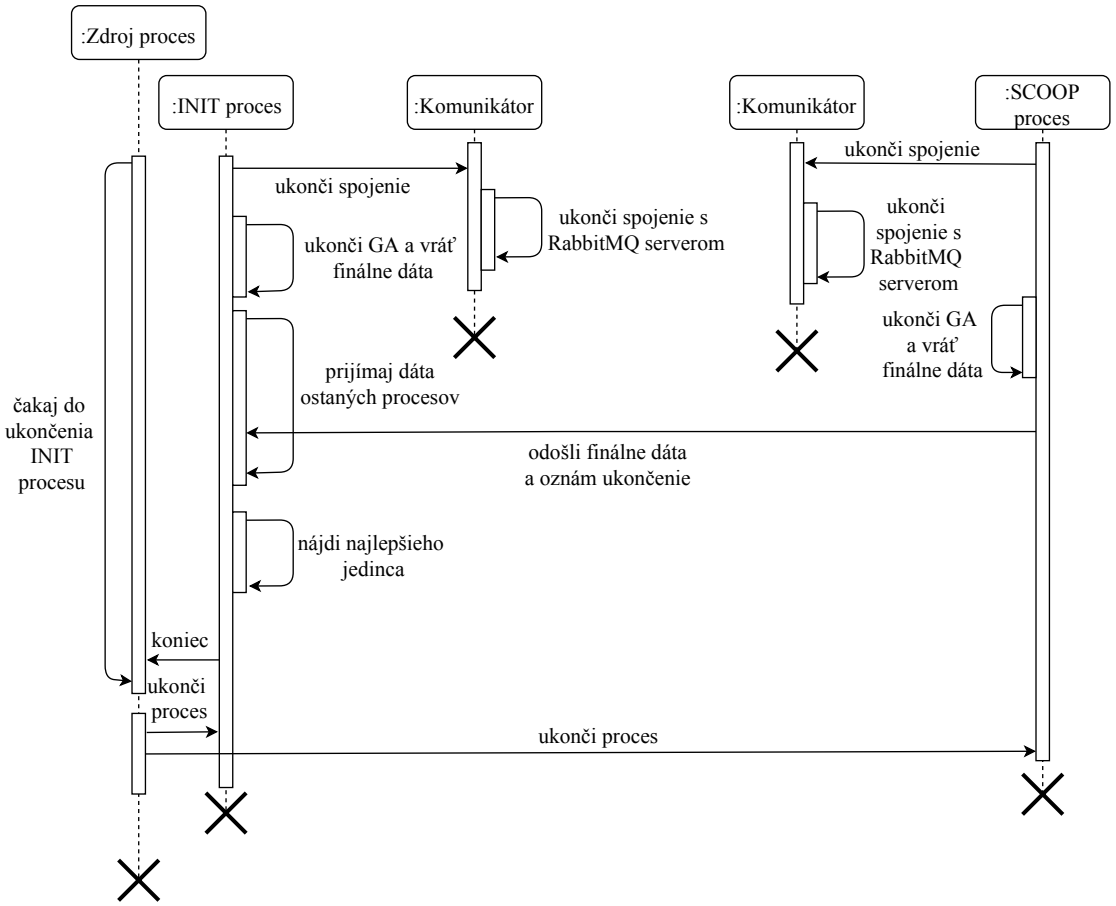
- *population_size* – veľkosť populácie
- *chromosome_size* – počet bitov chromozómu
- *number_of_generations* – počet generácií
- *fitness* – spôsobilostná funkcia

Užívateľ vytvorí na zvolenom mieste v adresárovom strome Python súbor, kde zavolá túto funkciu s vyplnenými argumentmi a implementovanou spôsobilostnou funkciou. Cestu a meno súboru potom zadá ako argumenty do spúšťajúcej funkcie *launch*. V prípade spúšťania modulu na viacerých staniciach budú vzdialené stanice (všetky okrem tej zdrojovej) obsahovať súbor len so spôsobilostnou funkciou. Funkcia *run_master_slave_ga* je teda stále len na zdrojovej stanici.

3.4.2 Implementácia jemnozrnného modelu

Hlavnou Python triedou pre tento model je Python trieda *FineGrainedBase*. Pomocou dedenia rozširuje funkcionalitu od triedy *GrainedGeneticAlgorithmBase*. Priebeh algoritmu už nekopíruje priebeh klasického GA. Výpočet GA je rozdelený paralelne medzi uzly topológie. Priebeh jemnozrnného modelu na jednom uzle je možné zhrnúť do týchto bodov:

1. ohodnotenie jedinca (každý uzol pracuje len s jedným jedincom)
2. odoslanie aktuálneho jedinca aj s jeho spôsobilostnou hodnotou uzlom (procesom) v susedstve
3. príjem dát od uzlov (procesov) v susedstve
4. pseudonáhodný alebo determinovaný výber jedného z prijatých jedincov
5. párenie (kríženie a mutácia) aktuálneho jedinca s vybratým jedincom
6. ak to ešte nebola posledná generácia, opakovanie priebehu od bodu 1
7. ukončenie priebehu modulu podľa diagramu 3.5



Obr. 3.5: UML diagram ukončenia modulu pri jemnozrnnom a hrubozrnnom modeli.

Ak niektorý uzol nájde najlepšie riešenie v skoršej generácii, výpočet spôsobilostnej funkcie a párenie jedincov sa už na danom uzle nebude vykonávať. Keďže ostatné uzly topológie ešte neukončili aktivitu a očakávajú od daného uzla dáta, uzol musí byť aktívny aj naďalej, aj keď iba vo fáze odosielania a prijímania dát.

Paralelné spracovanie je implementované v podmodule *algorithmRunners* v súbore *fineGrainedGaRunner.py*. Funkcia *run_fine_grained_ga* obsiahnutá v tomto súbore sa používa ako INIT funkcia. Jej argumentmi sú:

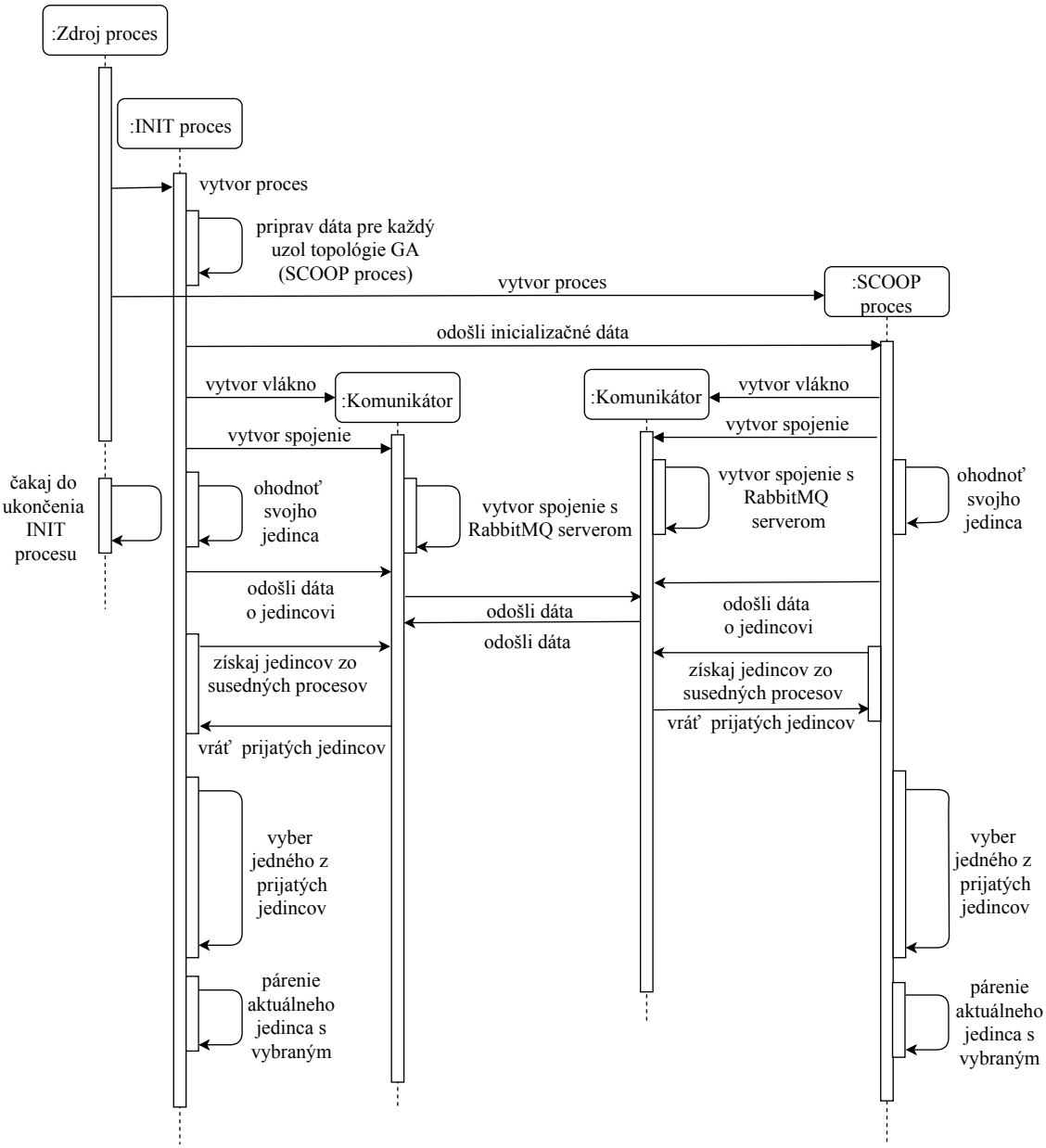
- *population_size* – veľkosť populácie, v tomto prípade sú to 2 čísla (rozmery) x:y
- *chromosome_size* – počet bitov chromozómu
- *number_of_generations* – počet generácií
- *neighbourhood_size* – polomer susedstva
- *server_ip_addr* – IP adresa RabbitMQ serveru
- *server_user* – názov užívateľského účtu na RabbitMQ serveri
- *server_password* – heslo na RabbitMQ serveri
- *fitness* – spôsobilostná funkcia

- *mate_best_neighbouring_individual* – párenie najlepšieho susediaceho jedinca (hodnoty pravda/nepravda). Určuje, či sa jedinec vyberie z prijatých pseudonáhodne alebo determinovane (vyberie sa najlepší jedinec).

Veľkosť populácie sa líši od master-slave modelu. Kvôli vytváraniu susedstiev a komunikácii medzi uzlami v rámci susedstiev je nutné poznať presné rozmery populácie. Preto je argument reprezentovaný v dvoch rozmeroch. Tvar topológie je dôležitý pri definovaní veľkosti a tvaru susedstiev.

Funkcia *run_fine_grained_ga* najprv vytvorí populáciu s pseudonáhodne vygenerovanými jedincami. Následne pomocou SCOOP funkcie *futures.map* rozdistribuuje jedincov medzi procesy. Každý proces reprezentuje jeden uzol a pracuje nad jedným jedincom populácie. Po skončení všetkých generácií sa v tejto funkcii dáta zozbierajú a vyberie sa najlepšie riešenie. Priebeh prvej iterácie spolu s distribúciou jedincov je ilustrovaný UML diagramom na obrázku 3.6.

Súbor *fineGrainedGaRunner.py* v podmodule *algorithmRunners* obsahuje taktiež funkciu *run_fine_grained_ga_remote*, ktorá sa používa pri spušťaní modulu na viacerých staniciach. Obsahuje rovnaké argumenty ako funkcia *run_fine_grained_ga*. Neobsahuje však už vytváranie populácie a distribúciu jedincov, keďže to zabezpečuje INIT proces. Volanie tejto funkcie by teda malo byť obsiahnuté v danom súbore na všetkých pracovných staniciach okrem tej zdrojovej.



Obr. 3.6: UML diagram jednej iterácie jemnozrnného modelu pre 2 uzly.

3.4.3 Implementácia hrubozrnného modelu

Implementácia algoritmu sa nachádza v tomto prípade v Python triede *CoarseGrainedBase*. Takisto ako pri jemnozrnnom modeli dedí od triedy *GrainedGeneticAlgorithmBase* a upravuje priebeh klasického GA na spracovanie na každom uzle takto:

1. ohodnotenie populácie (každý uzol pracuje s celou populáciou)
2. pseudonáhodný výber jedincov na párenie
3. párenie jedincov (kríženie a mutácia)

4. odoslanie n najlepších jedincov do uzlov (procesov) v susedstve
5. príjem dát od uzlov (procesov) v susedstve
6. pseudonáhodný výber jedincov z prijatých
7. integrácia vybraných jedincov do populácie
8. ak to ešte nebola posledná generácia, opakovanie priebehu od bodu 1
9. výber najlepšieho jedinca na každom uzle (procese)
10. ukončenie priebehu modulu podľa diagramu 3.5

Správanie modulu v prípade nájdenia najlepšieho riešenia v skoršej generácii je zhodné so správaním jemnozrnného modulu.

Spúšťanie a paralelizácia SCOOP procesov je implementovaná podobne ako pri jemnozrnnom modeli. INIT proces vykonáva funkciu *run_coarse_grained_ga*, ostatné procesy vykonávajú už popísaný priebeh bez vytvárania populácie. Pri využití viacerých pracovných staníc sa na vzdialených pracovných staniciach využíva funkcia *run_coarse_grained_ga_remote*. Argumenty sú tak ako v predošlom prípade rovnaké pre obe funkcie a sú to tieto:

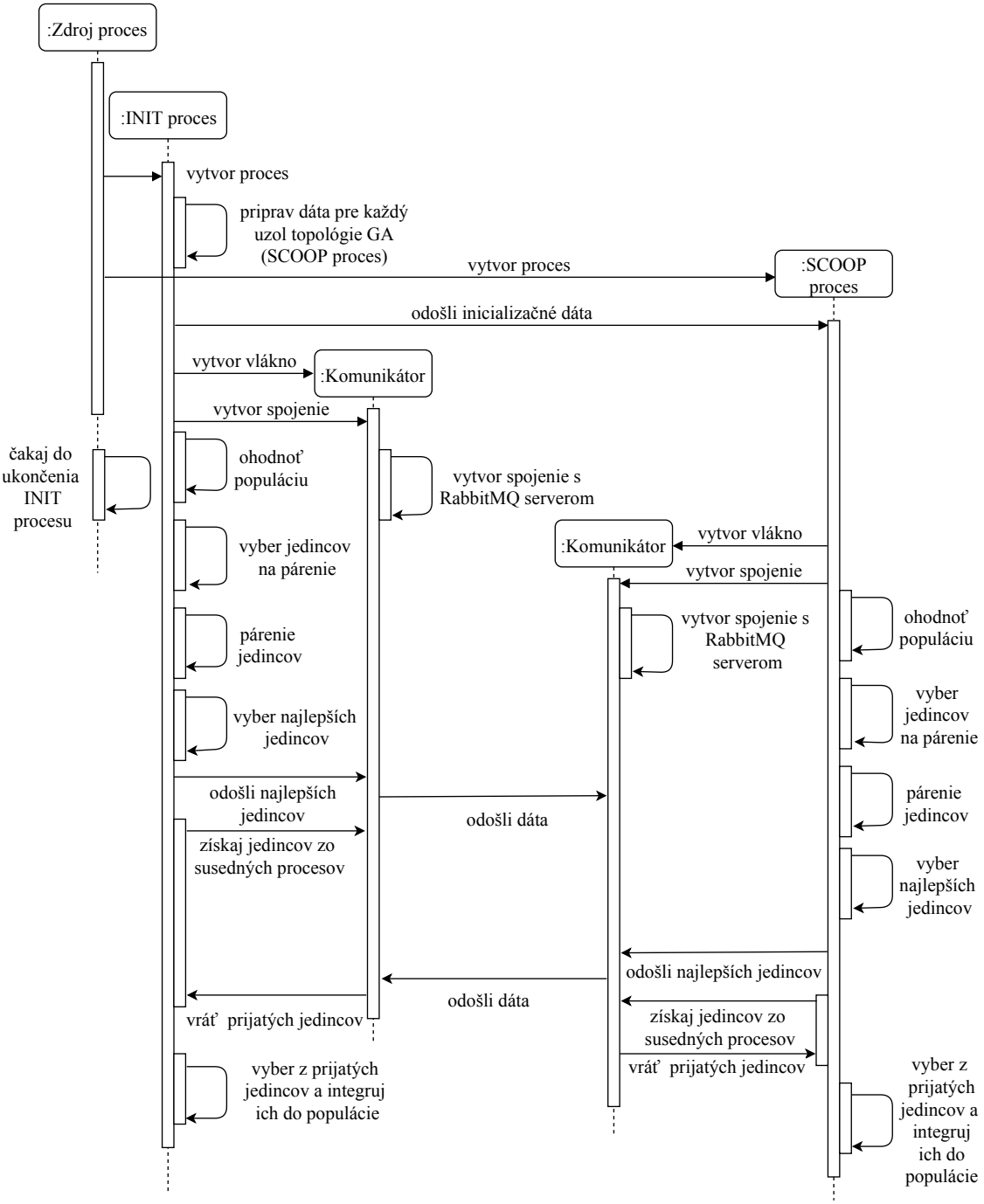
- *population_size* – veľkosť populácie, v tomto prípade sú to 2 čísla (rozmery) x:y
- *deme_size* – veľkosť populácie v jednom kmeni (uzle)
- *chromosome_size* – počet bitov chromozómu
- *number_of_generations* – počet generácií
- *neighbourhood_size* – polomer susedstva
- *server_ip_addr* – IP adresa RabbitMQ serveru
- *server_user* – názov užívateľského účtu na RabbitMQ serveri
- *server_password* – heslo na RabbitMQ serveri
- *fitness* – spôsobilostná funkcia
- *num_of_migrants* – počet jedincov, ktorí budú migrovať

Odlišnosti od jemnozrnného modelu možno odvodiť od nových parametrov, ktoré sa pri jemnozrnnom modeli nevyskytujú. Hrubozrnný model má dve premenné na veľkosť populácie. Jeden určuje veľkosť populácie na vyššej úrovni, teda na úrovni topológie uzlov. Veľkosť je podobne, ako pri jemnozrnnom modeli, reprezentovaná v dvoch rozmeroch, ktoré neskôr definujú tvar susedstiev. Druhou premennou je veľkosť populácie na úrovni jedného uzla, keďže pri hrubozrnnom modeli obsahuje každý uzol vlastnú populáciu.

Druhým rozdielom je odosielanie jedincov medzi uzlami. Pri jemnozrnnom modeli každý uzol posielal svojho jedinca do susedných uzlov. Keďže pri hrubozrnnom modeli má každý uzol viacero jedincov v rámci svojej populácie, vyberie si n najlepších jedincov, ktoré následne odošle. Ich počet si definuje užívateľ argumentom *num_of_migrants*.

Posledným rozdielom je inicializácia dát pre SCOOP procesy. Narozdiel od jem-

nozrnného modelu, kde sa každému procesu vygeneruje jeden jedinec, pri hrubozrnnom modeli sa vygeneruje pre každý proces celá populácia. Vygenerovanú populáciu proces využíva pri prvej iterácii, v rámci ktorej ju následne modifikuje párením jedincov. Priebeh prvej iterácie prevedený do UML diagramu možno vidieť na obrázku 3.7.



Obr. 3.7: UML diagram jednej iterácie hrubozrnného modelu pre 2 uzly.

4 TESTOVANIE VÝSLEDNÉHO MODULU

V tejto časti sa nachádza overenie funkčnosti implementácie a prínosu paralelizácie pre GA. Paralelizácia vo všeobecnosti prináša zvýšenie rýchlosti algoritmu a rozloženie záťaže. Zvyšovaním miery paralelizácie sa však zvyšujú nároky na jej riadenie a taktiež sa zvyšuje objem potrebnej komunikácie. Preto nemusí mať paralelizácia na algoritmus vždy len pozitívny vplyv. Okrem funkčnosti modulu sa teda vyhodnocuje taktiež vplyv paralelizácie na algoritmus.

4.1 Testovací scenár

Verifikácia prínosu paralelizácie GA spočíva v porovnaní paralelných modelov GA voči sériovému. Kedže sú GA stochastické, nemožno poskytnúť exaktné porovnanie. V tomto prípade sa používa štatistika opakovaných pokusov. V tejto práci bol počet opakovaní pokusu nastavený na 10. Z desiatich výsledkov sa následne vyberie jedna hodnota pomocou štatistickej funkcie *medián*.

Implementácia poskytuje paralelizáciu na úrovni procesov, teda paralelizáciu na úrovni jednej pracovnej stanice, a paralelizáciu na úrovni viacerých pracovných staníc. Porovnanie modelov GA bolo realizované iba na úrovni jednej pracovnej stanice z dôvodu konzistencie. Na každej pracovnej stanici beží operačný systém a množtvo dalších podporných procesov, ktoré využívajú a tým aj zaťažujú procesor a pamäť. Vyťaženie sa však môže v čase líšiť. Zapojením ďalších pracovných staníc do testovacieho scenára sa rozdiel len zväčší. Preto bolo testovanie na úrovni viacerých pracovných staníc obmedzené len na testovanie funkčnosti modulu a porovnávanie modelov bolo ponechané na úrovni jednej pracovnej stanice.

Paralelizácia GA spočíva v paralelizácii funkcií GA, ktoré sú časovo náročné. Najviac matematických operácií sa deje v rámci spôsobilostnej funkcie. Pre toto testovanie bola zvolená spôsobilostná funkcia 4.1.

$$f(x) = \sum_{i=1}^{d} x_i^2 \tag{4.1}$$

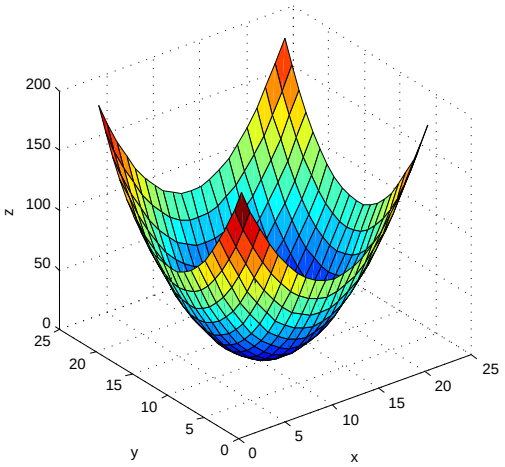
Zvolená bola jednoduchá funkcia, pri ktorej netrvá dlho nastavovanie chovania GA. Zámer tejto práce bol v paralelizácii, riešenie zložitejších funkcií by ale vyžadovalo analýzu danej funkcie a následne prispôsobenie GA. Zvolenú funkciu možno vidieť v 3-rozmernom prevedení na obrázku 4.1. Ako na ňom možno vidieť, správne riešenie minimalizácie sa nachádza v nulovom bode. Pri testovaní však nebola použitá 3-rozmerná varianta. Premenná x_i mohla nadobúdať len bitové hodnoty a premenná d určovala počet bitov. Táto funkcia (a zároveň aj jej implementácia) bola vybraná z množiny testovacích funkcií používaných programom evolučných výpočtov DEAP

[30]. Premenná d bola nastavená na hodnotu 20, teda algoritmus minimalizoval funkciu s hodnotami o dĺžke 20 bitov. Správny výsledok minimalizácie by teda mal byť bitový vektor:

$$[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]$$

V prílohe B sa nachádzajú záznamy aktivity algoritmu (známe pod anglickým názvom *logs*), kde je možné overiť výsledky poskytované algoritmom.

Veľkosť populácie bola postupne zvyšovaná pre každý pokus od veľkosti 64. Pri sériovom a master-slave GA môže byť veľkosť populácie ľubovoľná, v prípade jemnozrnného a hrubozrnného GA to však neplatí. Keďže pri týchto modeloch uzly populácie medzi sebou musia komunikovať, rozmery a usporiadanie populácie ovplyvňuje chovanie algoritmu. Pre jednoduchosť bola zvolená topológia so štvorcovým tvarom a s tvarom susedstva *X-Net 9* (už spomínaným pri 1.6). Využilo sa taktiež zakryvenie priestoru na tvar toroidu (ako bolo spomínané pri 1.5). Najmenšia topológia bola teda veľkosti 8×8, tak aby sa aj pri najmenších veľkostiach susedstiev dosiahlo relatívnej izolovanosti a aby prekrývanie medzi susedstvami nebolo príliš veľké. Následne pokusy pokračovali topológiami: $9 \times 9, 10 \times 10, 11 \times 11, 12 \times 12, 13 \times 13, 14 \times 14, 15 \times 15$. Keďže sa táto práca venuje paralelizácii, upravovanie chovania GA pre konkrétny riešený problém by bolo príliš zložité a časovo náročné. Z tohto dôvodu boli použité zjednodušené nastavenia GA. Veľkosť susedstva bola nastavná na minimálnu hodnotu – 1. Takto malo každé susedstvo veľkosť 9. Špecifický parameter hrubozrnného modelu *num_of_migrants* bol nastavený taktiež na minimálnu hodnotu – 1. Špecifický parameter jemnozrnného modelu *mate_best_neighbouring_individual* bol povolená, takže algoritmus nezavádzal náhodnosť do výberu susediacich potomkov.



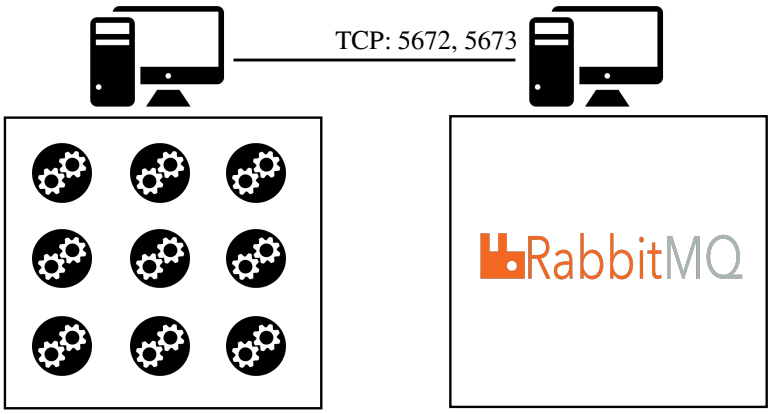
Obr. 4.1: Funkcia použitá na testovanie modelov GA.

Keďže súčasné procesory dokážu riešiť aj náročné matematické výpočty v extrémne malom časovom intervale, pre potreby testovania bolo do implementácie spôsobilostnej funkcie umiestnené oneskorenie 1 tisíciny sekundy. Týmto spôsobom bolo zretelnejšie vidieť pôsobenie paralelizácie.

4.2 Testovacie prostredie

Testovanie modelov paralelných GA bolo realizované na linuxovom serveri s operačným systémom Ubuntu vo verzii 17.10. Server obsahoval procesor *Intel(R) Xeon(R) CPU L5408 2.13GHz* so 4 jadrami. Veľkosť *cache* pamäte bola na tomto procesore 6144 KiB. Server obsahoval taktiež 32GiB operačnej pamäte (konkrétne to bolo 32943484 KiB).

Testovacie prostredie pozostávalo z RabbitMQ serveru a Python modulu paralelného GA. Ten pomocou modulu SCOOP vytvára procesy na zvolenej pracovnej stanici podľa obrázka 4.2.



Obr. 4.2: Testovacie prostredie pre jednú pracovnú stanicu.

Príprava RabbitMQ serveru

Keďže jemnozrnný a hrubozrnný model vyžadujú použitie RabbitMQ serveru, bola inštancia tohto serveru zahrnutá do testovacieho prostredia. Kvôli zníženiu zaťaženia na testovacom serveri bola inštancia umiestnená na druhý server, ktorý mal sieťové spojenie s testovacím serverom. RabbitMQ bol nainštalovaný na server vo verzii 3.6.10. Návod na inštaláciu možno nájsť v prílohe A.

RabbitMQ inštancia používa TCP porty 5672 a 5673. Tie využíva pre komunikáciu s klientmi – v tomto prípade procesmi paralelného GA.

Príprava Python modulu paralelného GA

Modul implementovaný v tejto práci je súčasťou modulov poskytovaných Python nástrojom *pip*. Keďže bol modul implementovaný v jazyku Python vo verzii 3.6, vyžaduje sa inštalácia podpory tohto jazyka v správnej verzii. Modul SCOOP následne požaduje taktiež inštaláciu SSH. Príprava modulu paralelného GA pre testovacie prostredie pozostáva príkazov v návode A. Modul SCOOP spúšťa na serveri daný počet procesov, pričom sa každý proces (iba v prípade jemnozrnného a hrubozrnného modelu) pripojí cez komunikačnú sieť na server so spusteným RabbitMQ. V prípade spúšťania procesov na viacerých pracovných staniciach boli nevyhnutné ďalšie nastavenia prostredia.

4.2.1 Paralelizácia na úrovni viacerých pracovných staníc

Pri testovaní na dvoch pracovných staniciach bolo odpozorované obmedzenie zo strany operačného systému alebo hardvéru na výkon, ktorý paralelný GA potreboval. Zvyšovaním počtu iterácií, počtu bežiacich procesov alebo veľkosti chromozómu nad

určitú mieru boli procesy neaktívne, aj keď z pohľadu operačného systému aktívne boli. Miera výkonu sa medzi jednotlivými pokusmi menila, možno však povedať, že táto miera činila 36 procesov (čo je minimálny počet procesov pre jemnozrnný a hrubozrnný model), 22 iterácií a veľkosť chromozómu 4 bity. Zvýšením jednej z týchto veličín boli tieto procesy neaktívne a paralelný GA zostal bežať bez odozvy.

Je však dôležité podotknúť, že procesy sa zastavili len na vzdialenej pracovnej stanici, procesy na stanici, ktorá modul spúšťala, fungovali bez problémov. Aj tento fakt prispel k rozhodnutiu realizovať merania pri scenári s jednou pracovnou stanicou. Testovanie s viacerými pracovnými stanicami bolo nakoniec realizované bez meraní.

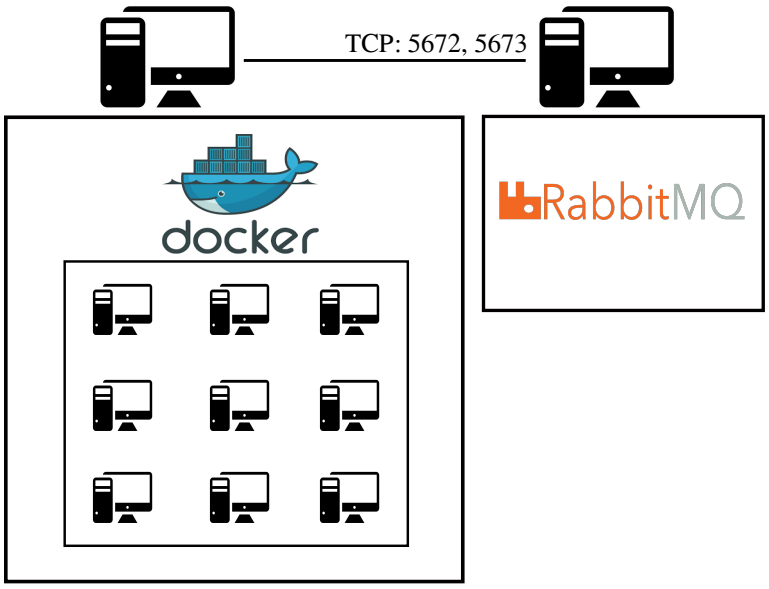
Príprava Python modulu paralelného GA

Testovacie prostredie tak muselo byť rozšírené na veľký počet pracovných staníc. Keďže sa nepodarilo získať väčší počet pracovných staníc, boli stanice virtualizované pomocou platformy Docker. Docker poskytuje virtualizovanie na úrovni operačného systému vytváraním tzv. kontajnerov. Každý kontajner môže v sebe ukrývať program alebo aj celý operačný systém. Pri testovaní bol použitý Docker vo verzii *18.03.1-ce (community edition) - build 9ee9f40*. Veľkosť chromozómu bola nastavená na 20 bitov, aby bolo overené prekonanie spomínaného obmedzenia.

Testovacie prostredie tak pozostávalo z dvoch serverov. Na jednom bežala platforma Docker a na tom druhom RabbitMQ server. Docker ďalej virtualizoval pracovné stanice s operačným systémom Ubuntu vo verzii 17.10. Vytvorenie potrebných kontajnerov bolo realizované pomocou návodov na oficiálnej stránke platformy Docker [36].

Na obrázku 4.3 možno vidieť scenár s dvoma fyzickými servermi a virtualizovanými pracovnými stanicami, na ktorých môže bežať modul paralelného GA. Keďže modul SCOOP paralelizuje výpočty pomocou vytvárania procesov pomocou SSH bolo nastavenie SSH na kontajneroch realizované pomocou návodu [37]. Následne bola na každej stanici (v tomto prípade na každom kontajneri) doinštalovaná podpora SSH serveru, jazyka Python a modulu paralelného GA. Pri nastavení SSH sa použil prihlasovací systém „verejného kľúča“. Súbor s nastaveniami platformy Docker sa nachádza v prílohe B.

Nakoniec bola potreba vytvoriť spustiteľné Python súbory na každej stanici a to na správnom mieste v adresárovom strome. Cesta k súboru by mala byť rovnaká pre všetky stanice, vrátane stanice, kde sa celý modul spúšťa.



Obr. 4.3: Testovacie prostredie pre viaceré pracovné stanice.

Príprava RabbitMQ serveru

Nakoniec je potrebné taktiež uviesť, že pre úplné splnenie princípov jemnozrnného a hrubozrnného modelu je potreba smerovať komunikáciu medzi pracovnými stanicami priamo a nie cez jeden server. Preto je potreba nastaviť RabbitMQ na každej pracovnej stanici zvlášť v móde *cluster*. Spustenie RabbitMQ v móde *cluster* možno realizovať pomocou návodu na oficiálnej stránke RabbitMQ [38]. Keďže však boli pracovné stanice virtualizované na jednom stroji, zaťaženie, ktoré by RabbitMQ vytvoril v každom kontajneri by znemožnilo testovanie, keďže sa pri testovaní zistilo, že scenár s platformou Docker vyťažuje procesor serveru takmer na 100%.

Dôvodom nastavenia serveru RabbitMQ na každú pracovnú stanicu je samozrejme zmenšenie oneskorenia spôsobeného komunikačnou sieťou. Keďže však boli všetky pracovné stanice virtualizované v rámci jedného stroja, komunikačné oneskorenie tam bolo preto minimálne.

4.3 Výsledky testovania

Pri testovaní modulu paralelného GA sa merali tieto veličiny:

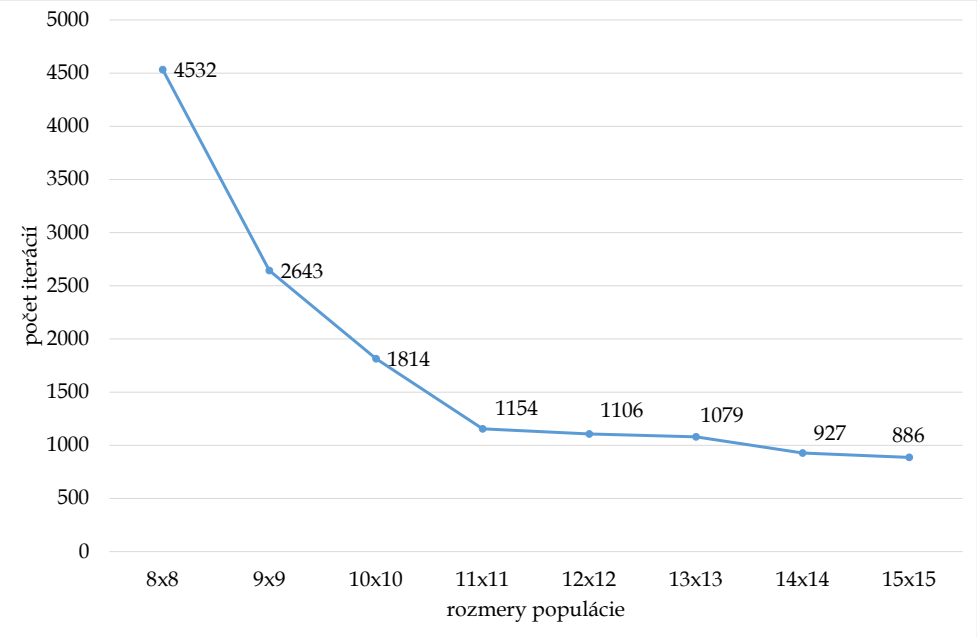
- počet iterácii potrebných pre dosiahnutie správneho výsledku
- čas potrebný pre dosiahnutie správneho výsledku
- vyťaženie procesora modulom paralelného GA
- veľkosť využitej operačnej pamäte modulom paralelného GA

Merania boli realizované najprv pre sériový GA a následne aj pre paralelné modely: master-slave, jemnozrný a hrubozrnný model. Prínos paralelizácie GA je zjavný z porovnania medzi sériovým a paralelným modelom. Prínosné sú však aj porovnávania medzi jednotlivými paralelnými modelmi.

4.3.1 Porovnanie počtu iterácií jednotlivých modelov GA

Počet iterácií GA závisí od jeho chovania a od špecifických parametrov daného typu GA. Keďže chovanie master-slave modelu je rovnaké ako chovanie sériového modelu, parametre GA boli pre oba modely zhodné. Z toho dôvodu teda štatistika počtu iterácií nebola realizovaná pre každý zvlášť, ale iba na sériovom modeli.

Výsledky meraní ilustrované na grafe 4.4 ukazujú predpokladané chovanie a to klesanie počtu iterácií so zväčšovaním populácie. Je to dôsledkom už spomínaného faktu, že väčšia populácia znamená väčšie pokrytie priestoru riešení. Pri väčšom pokrytí priestoru je tak väčšia pravdepodobnosť nájdenia riešenia. Miera klesania počtu iterácií sa približuje exponenciálnemu klesaniu.



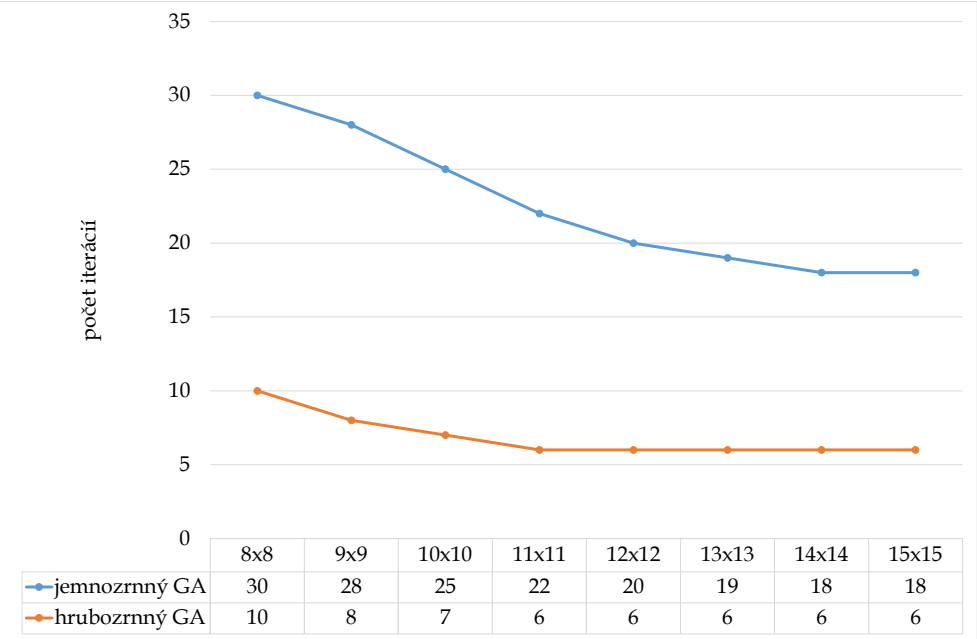
Obr. 4.4: Počet iterácii sériového a master-slave GA.

Jemnozrnný a hrubozrnný model už modifikujú chovanie GA, a preto sa predpokladá iný priebeh ako pri sériovom, resp. master-slave modeli. Je vhodné poznamenať, že rozmery populácie pri hrubozrnnom modeli definujú rozmery topológie. Každý uzol topológie je pritom samostatný GA so svojou vlastnou populáciou. Pri testovaní bola veľkosť populácie každého uzlu zhodná s rozmermi topológie. Napríklad pri rozmere 8×8 má jemnozrnný model populáciu s veľkosťou 64, hrubozrnný model

má však 64 populácií s veľkosťou 64. Zväčšovaním rozmerov populácie sa teda pri hrubozrnnom modeli zväčšovala aj populácia na uzloch. Týmto spôsobom veľkosť topológie vplývala nielen na chovanie uzlov medzi sebou ale aj na chovania uzlov samotných.

Výsledky merania zobrazené na obrázku 4.5 ukazujú, že tieto dva modely sú z pohľadu počtu iterácií oveľa efektívnejšie, keďže počet iterácií potrebných na nájdenie správneho riešenia bol zhruba 100-násobne menší ako pri sériovom. Hrubozrnný model vyžadoval ešte 3-násobne menej iterácii ako jemnozrnný. To sa môže pripísať už spomínanemu faktu, že hrubozrnný model obsahuje v každom uzle samostatnú populáciu, a teda pokrýva oveľa väčší priestor riešení.

Taktiež je zaujímavý priebeh grafov oboch modelov, keďže ich priebehy vyzerajú zhruba rovnako. To sa dá pripísať faktu, že tieto dva modely sú chovaním naozaj podobné (oba napríklad využívajú migráciu, ktorú sériový alebo master-slave model nemajú). Ďalším zistením bolo, že so zväčšovaním rozmerov populácie sa počet iterácií lineárne neznižuje, dokonca zostane až statický. Jedným z dohadov, ktorý by to vysvetľoval je, že oba modely potrebujú istý počet iterácií, kým sa dôkladne zmigrujú gény z jedného uzla na druhý. Zväčšovaním topológie sa zrejme migrácia stáva obtiažnejšou.



	8x8	9x9	10x10	11x11	12x12	13x13	14x14	15x15
jemnozrnný GA	30	28	25	22	20	19	18	18
hrubozrnný GA	10	8	7	6	6	6	6	6

Obr. 4.5: Porovnanie jemnozrnného a hrubozrnného modelu v počte iterácií.

4.3.2 Porovnanie výpočetného času jednotlivých modelov GA

Prínos paralelizácie pre GA sa najlepšie zhodnotí porovnaním času, ktorý potrebovali jednotlivé modely na nájdenie správneho riešenia. Síce porovnanie počtu iterácií hodnotí efektívnosť samotného algoritmu, paralelné spracovanie však významne ovplyvňuje rýchlosť algoritmu. Rýchlosť master-slave modelu závisí od toho, ako rýchlo dokáže model SCOOP spúšťať procesy a vymieňať si s nimi dáta. Pri jemnozrnnom a hrubozrnnom modeli je ďalším faktorom okrem modelu SCOOP aj server RabbitMQ, pomocou ktorého sa realizuje migrácia jedincov.

Pred porovnaním jednotlivých modelov je vhodné uviesť namerané oneskorenia spôsobené modulom SCOOP. Časy boli v tomto prípade a aj v ostatných prípadoch vyčítané zo záznamov (*logs*) modulu. Tabuľka 4.1 ukazuje oneskorenie pri štarte a pri konci modulu.

Rozmery populácie	8×8	9×9	10×10	11×11	12×12	13×13	14×14	15×15
Štart [s]	4	6	7	8,5	10	11,5	13,5	15,5
Koniec [s]	1,5	1,7	2,2	2,5	3,2	3,5	4,1	4,7

Tab. 4.1: Oneskorenie paralelných modelov GA spôsobené paralelizáciou pomocou SCOOP

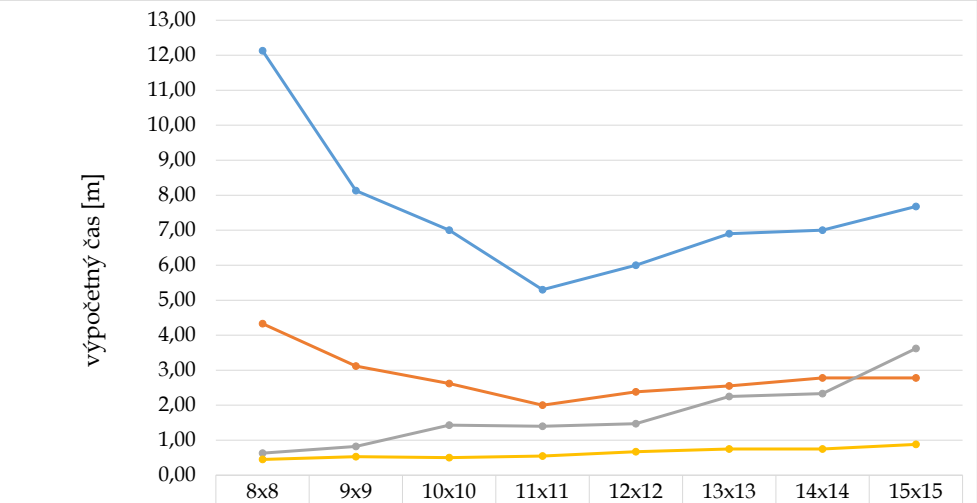
Taktiež je vhodné pripomenúť fakt, že v prípade jemnozrnného a hrubozrnného modelu sa pridávalo oneskorenie 1 sekundy pri vytváraní spojení s RabbitMQ serverom a to kvôli ubezpečeniu, že sa komunikácia medzi uzlami zaháji korektne.

Výsledky meraní času potrebného na nájdenie správneho riešenia a ich ilustrácie ukazuje graf 4.6. Pre lepšiu vizualizáciu porovnania bol výpočetný čas prepočítaný na minúty. Tak ako pri priebehu počtu iterácií, aj tu si možno všimnúť podobnosť priebehov sériového a master-slave modelu a taktiež aj podobnosť medzi jemnozrnným a hrubozrnným modelom. Priebehy sériového a master-slave sa dokonca podobajú na priebehy pri počte iterácií, avšak pri rozmere 11×11 sa trend znižujúceho sa času mení. Výpočetný čas sa ďalej so zväčšujúcimi rozmermi populácie začína zväčšovať. V prípade jemnozrnného a hrubozrnného modelu sa čas zväčšuje so vzrastajúcimi rozmermi populácie už od začiatku priebehu. Tento jav je možno vysvetliť tým, že algoritmus musí spracovať čoraz väčšiu populáciu a nárast tohto času prevyšuje čas ušetrený menším počtom iterácií. To platí pre všetky modely GA vrátane sériového. Pri paralelných modeloch sa ešte k tomu pripočítavá čas spotrebovaný modulom SCOOP 4.1. Keďže boli susedstvá nastavené pri testovaní konštantne na veľkosť 9, čas spotrebovaný komunikáciou by mal byť rovnaký pre

všetky rozmery populácie. Jediné oneskorenie v komunikácii mohlo byť spôsobené tým, že so zväčšovaním topológie sa zvyšovalo aj zaťaženie RabbitMQ serveru a ten mohol mať problém spracovávať všetky požiadavky včas.

Podľa výsledkov ilustrovaných v grafoch 4.4, 4.5 a 4.6 možno konštatovať, že rozmer populácie 11×11 bol pre nájdenie správneho riešenia pre 20 bitové číslo v priemere najrýchlejší. Prínos paralelizácie je z grafu 4.6 zrejmý, rozdiely v priebehu meraní boli spriemerované a výsledky boli takéto:

- master-slave model bol 2,65-násobne rýchlejší ako sériový model
- hrubozrnný model bol 6,27-násobne rýchlejší ako sériový a 2,33-násobne rýchlejší ako master-slave model
- jemnozrnný model bol 12,75-násobne rýchlejší ako sériový, 4,75-násobne rýchlejší ako master-slave a 2,6-násobne rýchlejší ako hrubozrnný model



	8x8	9x9	10x10	11x11	12x12	13x13	14x14	15x15
sériový GA	12,13	8,13	7,00	5,30	6,00	6,90	7,00	7,68
master-slave GA	4,33	3,12	2,62	2,00	2,38	2,55	2,78	2,78
hrubozrnný GA	0,63	0,82	1,43	1,40	1,47	2,25	2,33	3,62
jemnozrnný GA	0,45	0,53	0,50	0,55	0,67	0,75	0,75	0,88

Obr. 4.6: Porovnanie výpočetného času modelov GA.

Ohodnotenie prínosu paralelizácie sa podľa publikácie [17] všeobecne realizuje taktiež pomocou parametrov zrýchlenia (*SpeedUp*), efektívnosti (*Efficiency*) a škálovania (*Scaling*). Parameter zrýchlenia sa vypočíta podľa vzorca 4.2.

$$S = \frac{T_s}{T_p} \tag{4.2}$$

kde T_s je výpočetný čas pri sériovom algoritme a T_p je výpočetný čas pri paralelnom algoritme. Parameter efektívnosti sa následne vypočíta z hodnoty zrýchlenia podľa vzorca 4.3.

$$E = \frac{S}{p} \tag{4.3}$$

kde p je počet výpočetných jednotiek. Parameter škálovania nemá vzorec, ale všíma si straty výkonu algortimu pri vzájomnom zvyšovaní počtu výpočetných jednotiek a obtiažnosti výpočtov. Parameter p v tejto práci odpovedá počtu procesov.

Parametre modelu master-slave boli najlepšie pri rozmere populácie 8×8:

$$S = 2,801; E = 0,044$$

Škálovateľnosť bola podľa priebehu grafu pozitívna do rozmeru 11×11, pri ďalšom zväčšovaní rozmerov populácie výkonnosť klesala.

Parametre hrubozrnného modelu boli taktiež najlepšie pri rozmere 8×8:

$$S = 19,254; E = 0,301$$

Škálovateľnosť bola od tohto rozmeru už len negatívna.

Parametre jemnozrnného modelu dosahovali najlepšie hodnoty taktiež pri rozmere 8×8:

$$S = 26,956; E = 0,421$$

Škálovateľnosť bola podobná ako pri hrubozrnnom modeli.

4.3.3 Porovnanie hardvérového vyťaženia jednotlivých modelov GA

Implementácia paralelizácie zo sebou vo všeobecnosti prináša väčšie nároky na hardvér. Pri meraní parametrov samotného algoritmu sa v tejto práci merali taktiež parametry systému. Merania boli realizované pomocou linuxového nástroja *top*. Na úvod je vhodné poznamenať, že tieto výsledky môžu byť zkreslené z dôvodu prítomnosti bežiaceho operačného systému a jemu príslušiacich programov, ktoré takisto zaťažovali systém. Možné zkreslenie výsledkov však bola snaha priviesť na minimum tým, že oba servery mali nanovo inštalovaný operačný systém a pevný disk prešiel formátovaním. Taktiež je nutné poznamenať, že keďže sa jednalo o servery, tak v operačnom systéme nebežalo žiadne grafické prostredie a tým sa taktiež znížilo zaťaženie systému.

Nakoniec je ešte vhodné poznamenať, že od nameraného vyťaženia pamäte aj CPU bolo následne odpočítané vyťaženie v normálnom stave, teda bez spusteného GA. Takto získane hodnoty odrážali len vyťaženie modulu paralelného GA.

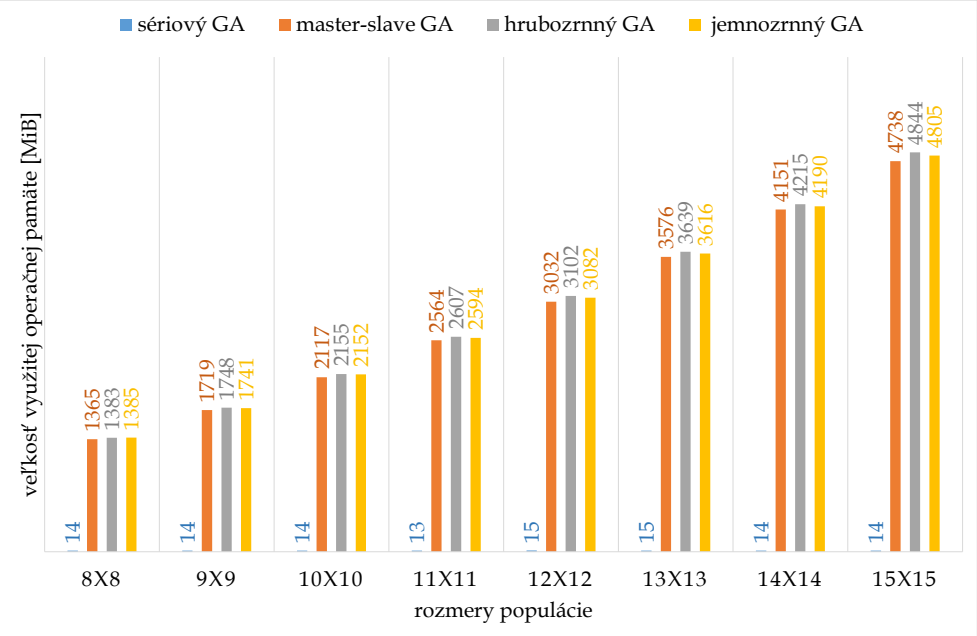
Porovnanie vyťaženia operačnej pamäte

Výsledky meraní na grafe 4.7 ukazujú, že sériový model GA mal podľa očakávaní najnižšie nároky na operačnú pamäť. V jeho prípade nedochádzalo k vytváraniu procesov pomocou SCOOP a ani k vytváraniu spojení s RabbitMQ serverom. So zväčšujúcimi sa rozmermi populácie sa využitie pamäte zdanlivo nemenilo. Pri rozmeroch

11×11, 14×14 a 15×15 dokonca využitá pamäť klesla. To sa však nedá pripisovať menšej spotrebe pamäte modulom GA, ide skôr o náhodný pokles využitia pamäte operačným systémom v danom období testovania.

Pri paralelných modeloch GA sú už výsledky meraní využitia pamäte jednoznačnejšie. So zväčšujúcimi sa rozmermi populácie sa zvyšovalo aj využitie pamäte. Pre všetky 3 paralelné modely možno skonštatovať, že nárast využitia pamäte bol zhruba lineárny. Je to prekvapivý fakt s ohľadom nato, že sa populácia zvyšovala kvadraticky.

Najväčšie využitie pamäte mal hrubozrnný model. Druhým v poradí bol jemnozrnný model a nakoniec najnižšie využitie pamäte mal master-slave model. Rozdiely medzi paralelnými modelmi bol však minimálny. Jednoznačné je však niekoľko tisíc násobne väčšie využitie pamäte paralelnými modelmi oproti sériovému modelu.



Obr. 4.7: Porovnanie modelov GA vo vyťažení operačnej pamäte.

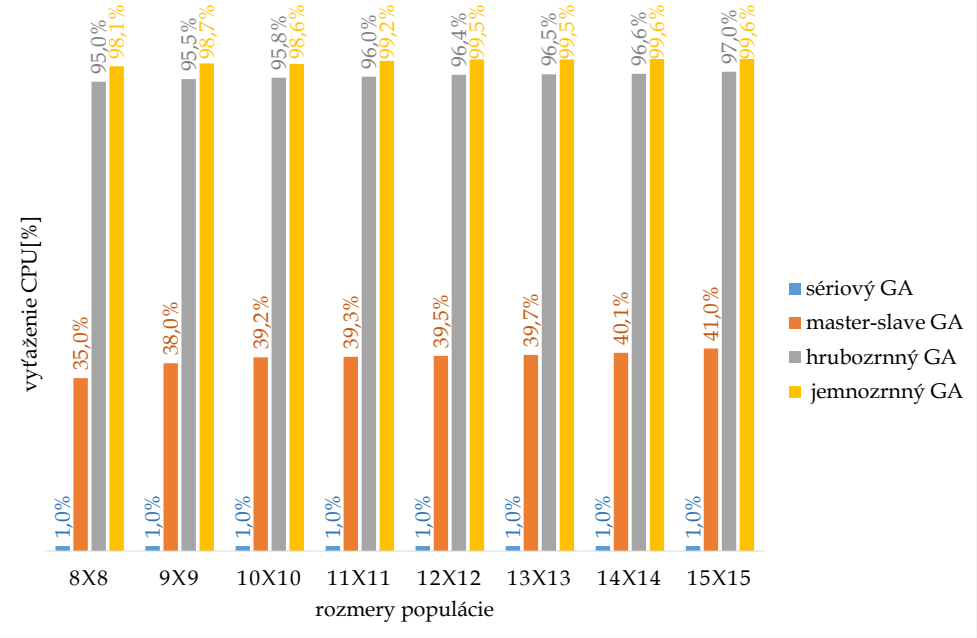
Porovnanie vyťaženia CPU

Ako bolo spomenuté pri porovnávaní využitia pamäte, tak aj pri meraní vyťaženia CPU je nutné poznamenať, že namerané výsledky môžu byť skreslené kvôli bežiacemu operačnému systému a jeho programom.

Výsledky na grafe 4.8 ukazujú v prípade sériového modelu podobný scenár ako pri meraní pamäte. Sériový model sa počas celého priebehu držal na úrovni 1% vyťaženia CPU. Master-slave model bol oproti ostatným paralelným modelom v prípade CPU efektívnejší ako to bolo v prípade využitia pamäte. Hrubozrnný a jemnozrnný model

využívali CPU takmer na 100%, pričom jemnozrnný model využíval CPU asi o 3% viac ako hrubozrnný.

Priebeh využitia CPU sa takmer nedá označiť ani za lineárny, aj keď istý nárast hodnôt je v priebehu prítomný. Je však možné konštatovať, že zväčšovanie rozmerov populácie nemá taký výrazný vplyv na CPU ako mal na operačnú pamäť.



Obr. 4.8: Porovnanie modelov GA vo vyťažení CPU.

4.3.4 Overenie paralelizácie modulu na viacerých pracovných staniciach

Ako už bolo spomenuté, všetky merania sa realizovali na jednej pracovnej stanici. Táto časť sa preto venuje overeniu funkčnosti modulu pre paralelné GA na viacerých pracovných staniciach.

Podľa testovacieho scenára zobrazeného na obrázku 4.3, pozostávalo prostredie z pracovných staníc vytvorených platformou Docker. Rozmer populácie bol zvolený na najmenší možný – 8×8. Overiť funkčnosť tohto scenára možno kontolou záznamov (*logs*) z modulu paralelného GA. Modul SCOOP poskytuje pre vytváranie záznamov vlastný podmodul zvaný *logger*. Ten bol využívaný v každej časti modulu GA. Takto sa vytvárali záznamy nielen pri činnosti modulu SCOOP, ale aj pri činnosti GA alebo pri odosielaní a prijímaní dát. Všetky záznamy sú súčasťou prílohy B. Keďže sa však jedná o súbory s obrovským počtom záznamov, v ktorých sa preto ťažko orientuje, pre overenie funkčnosti je tu výsek najdôležitejších záznamov týkajúcich sa iba paralelizácie, teda modulu SCOOP.

Prvá časť je výsek záznamov vygenerovaných pri štarte modulu:

```
2018-05-03 22:40:09,843] launcher INFO SCOOP 0.7 1.1 on linux using
    Python 3.6.3 [GCC 7.2.0], API: 1013
[2018-05-03 22:40:09,843] launcher INFO Deploying 36 worker(s) over
    36 host(s).
[2018-05-03 22:40:09,843] launcher DEBUG The python executable to
   execute the program with is: /usr/bin/python3.6.
[2018-05-03 22:40:09,843] launcher INFO Worker distribution:
[2018-05-03 22:40:09,843] launcher INFO    localhost: 0 + origin
[2018-05-03 22:40:09,844] launcher INFO    root@172.17.0.2: 1
[2018-05-03 22:40:09,844] launcher INFO    root@172.17.0.3: 1
[2018-05-03 22:40:09,844] launcher INFO    root@172.17.0.4: 1
[2018-05-03 22:40:09,844] launcher INFO    root@172.17.0.5: 1
[2018-05-03 22:40:09,844] launcher INFO    root@172.17.0.6: 1
[2018-05-03 22:40:09,844] launcher INFO    root@172.17.0.7: 1
[2018-05-03 22:40:09,844] launcher INFO    root@172.17.0.8: 1
[2018-05-03 22:40:09,844] launcher INFO    root@172.17.0.9: 1
[2018-05-03 22:40:09,844] launcher INFO    root@172.17.0.10: 1
[2018-05-03 22:40:09,844] launcher INFO    root@172.17.0.11: 1
[2018-05-03 22:40:09,844] launcher INFO    root@172.17.0.12: 1
[2018-05-03 22:40:09,844] launcher INFO    root@172.17.0.13: 1
[2018-05-03 22:40:09,845] launcher INFO    root@172.17.0.14: 1
[2018-05-03 22:40:09,845] launcher INFO    root@172.17.0.15: 1
[2018-05-03 22:40:09,845] launcher INFO    root@172.17.0.16: 1
[2018-05-03 22:40:09,845] launcher INFO    root@172.17.0.17: 1
[2018-05-03 22:40:09,845] launcher INFO    root@172.17.0.18: 1
[2018-05-03 22:40:09,845] launcher INFO    root@172.17.0.19: 1
[2018-05-03 22:40:09,845] launcher INFO    root@172.17.0.20: 1
[2018-05-03 22:40:09,845] launcher INFO    root@172.17.0.21: 1
[2018-05-03 22:40:09,845] launcher INFO    root@172.17.0.22: 1
[2018-05-03 22:40:09,845] launcher INFO    root@172.17.0.23: 1
[2018-05-03 22:40:09,845] launcher INFO    root@172.17.0.24: 1
[2018-05-03 22:40:09,845] launcher INFO    root@172.17.0.25: 1
[2018-05-03 22:40:09,846] launcher INFO    root@172.17.0.26: 1
[2018-05-03 22:40:09,846] launcher INFO    root@172.17.0.27: 1
[2018-05-03 22:40:09,846] launcher INFO    root@172.17.0.28: 1
[2018-05-03 22:40:09,846] launcher INFO    root@172.17.0.29: 1
[2018-05-03 22:40:09,846] launcher INFO    root@172.17.0.30: 1
[2018-05-03 22:40:09,846] launcher INFO    root@172.17.0.31: 1
[2018-05-03 22:40:09,846] launcher INFO    root@172.17.0.32: 1
[2018-05-03 22:40:09,846] launcher INFO    root@172.17.0.33: 1
[2018-05-03 22:40:09,846] launcher INFO    root@172.17.0.34: 1
[2018-05-03 22:40:09,846] launcher INFO    root@172.17.0.35: 1
[2018-05-03 22:40:09,846] launcher INFO    root@172.17.0.36: 1
```

V záznamoch si možno všimnúť, že bol využitý modul SCOOP vo verzii 0.7 na operačnom systéme Linux s jazykom Python vo verzii 3.6.3. Druhý riadok ukazuje,

že bolo spustených 36 procesov (*workers*) na 36 pracovných staniciach (*hosts*). Na zvyšných riadkoch možno vidieť záznamy spúšťania procesov. Každý záznam obsahuje dvojicu – pracovná stanica:počet procesov na stanici. V záznamoch možno vidieť, že na zdrojovej stanici (*localhost*) nebol spustený žiaden nový proces, zostal len proces INIT. Ďalej možno vidieť, že na každej stanici bol spustený práve jeden proces.

Druhá časť záznamov je výsek z ukončenia modulu (neboli vybrané záznamy pre všetkých 36 procesov ako v predošlom prípade):

```
[2018-05-03 22:43:49,831] workerLaunch (127.0.0.1:45217) DEBUG
    Closing workers on root@172.17.0.4 (1 workers).
[2018-05-03 22:43:49,983] workerLaunch (127.0.0.1:45217) DEBUG
    Closing workers on root@172.17.0.3 (1 workers).
[2018-05-03 22:43:50,135] workerLaunch (127.0.0.1:45217) DEBUG
    Closing workers on root@172.17.0.2 (1 workers).
[2018-05-03 22:43:50,290] workerLaunch (127.0.0.1:45217) DEBUG
    Closing workers on localhost (1 workers).
[2018-05-03 22:43:50,291] brokerLaunch (127.0.0.1:45217) DEBUG
    Closing local broker.
[2018-05-03 22:43:50,291] launcher   (127.0.0.1:45217) INFO     Finished
    cleaning spawned subprocesses.
exit code 0
```

Konečná fáza modulu zobrazená v záznamoch ukazuje ukončenie 3 procesov na vzdialených staniciach a následne aj INIT procesu na zdrojovej stanici. Po tom ako boli všetky procesy korektne ukončené môže modul SCOOP ukončiť svoj modul *broker*, ktorý sa stará o komunikáciu medzi ním a jednotlivými procesmi. Až potom sa môže ukončiť aj zdrojový proces. Posledný záznam obsahuje oznámenie o ukončení procesu (*exit*) s výsledným kódom (*code*) 0. Kód 0 znamená, že všetko prebehlo bez problémov.

Nájdením správneho riešenia pri veľkosti chromozómu 20 bitov boli overené všetky 3 modely paralelného GA na viacerých spracovných staniciach. Master-slave model bol overený 7000 iteráciami, jemnozrnný model 70 iteráciami a hrubozrnný model 10 iteráciami. Rozdielny počet iterácií ako pri scenári na jednej pracovnej stanici si netreba vysvetľovať ako zhoršenie efektívnosti GA. Chovanie GA je rovnaké na ľubovoľnom počte pracovných staníc. Pri scenári na jednej pracovnej stanici však boli výsledky zpriemerované, pri scenári na viacerých staniciach išlo iba o overenie, že modul je schopný iterovať do vysokých čísel a dokáže zvládnuť veľké zaťaženie.

4.4 Zhodnotenie testovania

V tejto časti bol popísaný testovací scenár spolu s prostredím, v ktorom sa mali testy realizovať a nakoniec boli popísané aj výsledky meraní. Scenár zohľadňoval stochastickosť GA viackrát opakovanými meraniami. Scenár taktiež zadefinoval najdôležitejšie parametre, ktoré sa mali pri testovaní merať. Bol to počet iterácií, výpočetný čas a vyťaženie CPU a operačnej pamäte.

Spomínané modely GA boli úspešne otestované pri scenári s jednou pracovnou stanicou a taktiež pri scenári s viacerými pracovnými stanicami. Testovanie na viacerých pracovných staniciach odhalilo pri zvyšovaní záťaže nad určitú hranicu obmedzenie zo strany operačného systému.

Počet iterácií sa zväčšovaním rozmerov populácie zmenšoval pri všetkých troch modeloch GA. Výpočetný čas mal podobný priebeh aj keď v niektorých prípadoch sa naopak zvyšoval. Celkovo však modely paralelného GA prinášali úsporu času oproti sériovému modelu. Najrýchlejším modelom bol jemnozrnný model.

Použitie parametrov zrýchlenia, efektívnosti a škálovania pre porovnanie medzi sériovým a paralelnými modelmi prinieslo ďalšie výsledky. V priemere bol pre paralelné modely rozmer populácie 11×11 najrýchlejší, rozmer 8×8 najefektívnejší. Teoretické zrýchlenie pre rozmer 8×8 bolo 64-násobné. Master-slave dosiahol takmer 3-násobné zrýchlenie, hrubozrnný model sa priblížil k 20 násobku zrýchlenia a jemnozrnný dosiahol najlepší výsledok zo všetkých s takmer 27-násobným zrýchlením oproti sériovému modelu. Teoretická hodnota pre parameter efektívnosť bola rovná jednému násobku zrýchlenia pre výpočetnú jednotku. Najbližšie k tejto hodnote sa dostal jemnozrnný model so 4 desatinami násobku zrýchlenia pre výpočetnú jednotku. Hrubozrnný model zaostal za jemnozrnným modelom o jednu desatinu. Master-slave model dosiahol len 4 stotiny násobku zrýchlenia pre výpočetnú jednotku. Keďže sa teoretické hodnoty zrýchlenia a efektívnosti nedajú v praxi dosiahnuť, tieto výsledky sú hodnotené ako pozitívne. Posledný parameter – škálovateľnosť, bol hodnotený pozitívne iba v prípade master-slave modelu.

Porovnanie vyťaženia operačnej pamäte prinieslo predpokladané výsledky. So zväčšujúcimi sa rozmermi populácie rástlo aj vyťaženie pamäte. Prekvapivý je však fakt, že master-slave mal podobné výsledky ako ostatné modely paralelného GA, aj keď nemusel spracovávať dáta prijaté z ostatných procesov, keďže master-slave model neobsahuje žiadnu komunikáciu medzi procesmi. Sériový model bol podľa predpokladov pamäťovo najmenej náročný.

Vyťaženie CPU nenarastalo so zväčšujúcimi rozmermi populácie tak dramaticky ako vyťaženie pamäte. Ďalším rozdielom oproti vyťaženiu pamäte bolo výrazne nižšie vyťaženie CPU pri master-slave modeli oproti ostatným modelom paralelného GA. Najnižšie vyťaženie mal však podľa predpokladu sériový model.

5 ZÁVER

V tejto práci bol popísaný genetický algortimus, jeho fungovanie a použitie. Ďalej boli popísané rôzne modely paralelného vykonávania, ktoré boli následne porovnané. Výsledkom bolo zhodnotenie, že každý model je efektívnejší na iné optimalizačné problémy a je tažko porovnať ich výkonnosť. Na druhej strane sú však všetky paralelné modely výkonnejšie ako obyčajný sériový model.

Ďalej bol navrhnutý spôsob implementácie paralelných modelov. Paralelné spracovanie bolo najprv teoreticky popísané a následne boli teoretické výstupy zakomponované do samotného návrhu. Návrh bol rozdelený do dvoch častí: do návrhu paralelného výpočtu a návrhu komunikácie. V obidvoch častiach boli popísané rôzne moduly jazyka Python, ktoré možno použiť. Pri výbere modulov došlo k viacerým neúspešným integráciám modulov na pracovné stanice, ktoré boli súčasťou testovacieho scenára. Tie, ktoré boli úspešne integrované, boli detailne popísané z hľadiska ich výhod a nevýhod a boli vybraté najlepšie možnosti. Nakoniec bol navrhnutý spôsob ako zakomponovať Python moduly do implementácie modelov paralelného genetického algoritmu.

V praktickej časti boli implementované modely: jemnozrnný, hrubozrnný a master-slave. Hierarchický model bol z dôvodu obtiažnej implementácie z hľadiska paralelizácie vynechaný. Implementácia bola detailne popísaná z hľadiska paralelizácie, komunikácie a z hľadiska chovania genetického algoritmu. Algoritmy boli následne ilustrované pomocou UML diagramov.

Nakoniec bolo vykonané testovanie spomínaných modelov. Bola overená funkčnosť paralelizácie vrámci jednej pracovnej stanici a taktiež aj medzi viacerými pracovnými stanicami. Pri paralelizácii medzi viacerými pracovnými stanicami bolo v niektorých prípadoch odpozorované obmedzenie zo strany operačného systému. Pri testovaní na jednej pracovnej stanici boli namerané hodnoty výpočetného času, počtu iterácií a hardvérového vyťaženia. Výsledky jednoznačne potvrdili prínos paralelizácie pre genetické algoritmy, keďže všetky tri modely paralelného genetického algoritmu dosiahli výrazné zrýchlenie a zefektívnenie oproti sériovému modelu.

LITERATÚRA

[1] HYNEK, Josef. *Genetické algoritmy a genetické programování*. Praha: Grada, 2008, 182 s. : il. (prevažne fareb.). ISBN 978-80-247-2695-3.

[2] KVASNIČKA Vladimír, Jiří POSPÍCHAL a Peter TIŇO. *Evolučné algoritmy*. Bratislava: Slovenská technická univerzita v Bratislave vo Vydavateľstve STU, 2000, 215 s : ilustrácie. ISBN 80-227-1377-5.

[3] BLUM, Christian a Andrea ROLI. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys (CSUR)* [online]. ACM, 2003, **35**(3), 268-308 [cit. 2017-12-01]. DOI: 10.1145/937503.937505. ISSN 0360-0300. Dostupné z: `https://dl-acm-org.ezproxy.lib.vutbr.cz/citation.cfm?doid=937503.937505`

[4] Michalski Ryszard, Carbonell Jaime a Mitchell Tom. *Machine Learning: An Artificial Intelligence Approach*. Heidelberg, Nemecko: vydavateľ Springer-Verlag Berlin Heidelberg, 2014, 825 s. ISBN 978-0-08-051055-2

[5] Darwin, Charles. *On the Origin of Species by Means of Natural Selection, Or, The Preservation of Favoured Races in the Struggle for Life*. University of Oxford, 1859. 502 s.

[6] Holland, John Henry. *Adaptation in Natural and Artificial Systems*. 1. vyd. University of Michigan Press, 1975. 211 s.

[7] DE GIOVANNI, Luigi. *Methods and Models for Combinatorial Optimization: Heuristis for Combinatorial Optimization* [online]. Padova, Taliansko: The University of Padova, 2016 [cit. 2017-11-11]. Dostupné z: `http://www.math.unipd.it/~luigi/courses/metmodoc1617/m02.meta.en.partial01.pdf`

[8] BUŠA, Ján. *Mini minimalizácia* [online]. Košice: vydavateľstvo FEI TU, 2011 [cit. 2017-11-1]. ISBN 80-8073-XXX-X. Dostupné z: `http://web.tuke.sk/fei-km/sites/default/files/prilohy/10/OM-Minimalizacia-Busa.pdf`

[9] CANTÚ-PAZ, Erick. *A Survey of Parallel Genetic Algorithms* [online]. Urbana-Champaign, štát Illinois, USA: The University of Illinois, 1998 [cit. 2017-11-18]. Dostupné z: `http://neo.lcc.uma.es/cEA-web/documents/cant98.pdf`

[10] HIROYASU, T., M. MIKI a Y. TANIMURA. The differences of parallel efficiency between the two models of parallel genetic algorithms on PC cluster systems. In: *Proceedings Fourth International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region* [online]. IEEE Publishing,

2000, **2**, s. 945-948 [cit. 2017-11-18]. DOI: 10.1109/HPC.2000.843576. ISBN 0-7695-0590-2. Dostupné z: http://ieeexplore.ieee.org/document/843576

[11] EKLUND, Sven E. A massively parallel architecture for distributed genetic algorithms. *Parallel Computing* [online]. Elsevier B.V, 2004, **30**(5), 647-676 [cit. 2017-11-18]. DOI: 10.1016/j.parco.2003.12.009. ISSN 0167-8191. Dostupné z: http://www.sciencedirect.com.ezproxy.lib.vutbr.cz/science/article/pii/S0167819104000365

[12] FOGEL, David B., Thomas BACK a Zbigniew. MICHALEWICZ. *Evolutionary Computation 2: Advanced Algorithms and Operators*. Philadelphia: Institute of Physics Publishing, 2000. ISBN 9780750306652.

[13] POŠÍK, Petr. *Paralelní genetické algoritmy* [online]. Praha, 2001 [cit. 2017-11-25]. Dostupné z: http://ida.felk.cvut.cz/cgi-bin/docarc/public.pl/document/38/Posik2001Diplomka.pdf. Magisterská práca. České vysoké učení technické.

[14] Galván-González, Sergio. Finding the Optimal Shape of the Flow Passage in a Francis Runner Using Numerical Tools. In: *Conference: II Latin American Hidro Power and Sistems Meeting* [online]. La Plata, Argentína, 2015, **2**, s.13 [cit. 2017-11-26]. Dostupné z: https://www.researchgate.net/publication/279756862_Finding_the_Optimal_Shape_of_the_Flow_Passage_in_a_Francis_Runner_Using_Numerical_Tools

[15] LONES, Michael. Sean Luke: essentials of metaheuristics. *Genetic Programming and Evolvable Machines* [online]. Boston: Springer US, 1109, **12**(3), 333-334 [cit. 2017-11-26]. DOI: 10.1007/s10710-011-9139-0. ISSN 1389-2576. Dostupné z: https://link-springer-com.ezproxy.lib.vutbr.cz/article/10.1007/s10710-011-9139-0

[16] CHIPPERFIELD, A. J. a P. J. FLEMING. *Parallel Genetic Algorithms: A Survey* [online]. Sheffield, Veľká Británia: The University of Sheffield: Department of Automatic Control and Systems Engineering, 1994 [cit. 2017-11-26]. Dostupné z: http://eprints.whiterose.ac.uk/79633/1/acse%20research%20report%20518.pdf

[17] ZACCONE, Giancarlo. *Python Parallel Programming Cookbook*. Packt Publishing, 2015. ISBN 1785289586.

[18] RPyC - use cases: Parallel Execution. In: *RPyC - Transparent, Symmetric Distributed Computing* [online]. 2017 [cit. 2017-12-09]. Dostupné z: https://rpyc.readthedocs.io/en/latest/docs/usecases.html

[19] Parallel Processing and Multiprocessing in Python. In: *Python* [online]. 2017 [cit. 2017-12-10]. Dostupné z: https://wiki.python.org/moin/ParallelProcessing

[20] Multiprocessing - Process-based parallelism. In: *Python* [online]. 2017 [cit. 2017-12-10]. Dostupné z: https://docs.python.org/3.6/library/multiprocessing.html

[21] Distributed computing in Python with multiprocessing. In: *Eli Bendersky's website* [online]. 2017 [cit. 2017-12-10]. Dostupné z: https://eli.thegreenplace.net/2012/01/24/distributed-computing-in-python-with-multiprocessing

[22] Celery: Distributed Task Queue. *Celery project* [online]. 2017 [cit. 2017-12-10]. Dostupné z: http://www.celeryproject.org

[23] How to build docker cluster with celery and RabbitMQ in 10 minutes. In: *Medium* [online]. [cit. 2017-12-11]. Dostupné z: https://medium.com/@tonywangcn/how-to-build-docker-cluster-with-celery-and-rabbitmq-in-10-minutes-13fc74d21730

[24] PyCSP Wiki. In: *Github* [online]. [cit. 2017-12-11]. Dostupné z: https://github.com/runefriborg/pycsp/wiki

[25] SCOOP. In: *Github* [online]. [cit. 2017-12-11]. Dostupné z: https://github.com/soravux/scoop

[26] Running an MPI Cluster within a LAN. In: *A Comprehensive MPI Tutorial Resource* [online]. 2017 [cit. 2017-12-11]. Dostupné z: http://mpitutorial.com/tutorials/running-an-mpi-cluster-within-a-lan

[27] *RabbitMQ - Messaging that just works* [online]. 2017 [cit. 2017-12-11]. Dostupné z: https://www.rabbitmq.com

[28] GAFT - A Genetic Algorithm Framework in pyThon. In: *Github* [online]. [cit. 2017-12-11]. Dostupné z: https://github.com/PytLab/gaft

[29] DEAP - Distributed Evolutionary Algorithms in Python. In: *Github* [online]. [cit. 2017-12-11]. Dostupné z: https://github.com/DEAP/deap

[30] Benchmarks for DEAP. In: *Github* [online]. [cit. 2018-04-15]. Dostupné z: https://github.com/DEAP/deap/tree/master/deap/benchmarks

[31] How To Package Your Python Code. [online]. [cit. 2018-03-09]. Dostupné z: https://python-packaging.readthedocs.io/en/latest/index.html

[32] This package provides tools for processing hard problems with parallel genetic algorithm. [online]. [cit. 2018-05-13]. Dostupné z: https://pypi.org/project/parallel_ga_processing/

[33] My Master thessis. In: *Github* [online]. [cit. 2018-05-13]. Dostupné z: https://github.com/lucker7777/parallel_ga_processing

[34] Asynchronous consumer example. [online]. [cit. 2018-04-10]. Dostupné z: https://pika.readthedocs.io/en/stable/examples/asynchronous_consumer_example.html

[35] Asynchronous publisher example. [online]. [cit. 2018-04-10]. Dostupné z: https://pika.readthedocs.io/en/stable/examples/asynchronous_publisher_example.html

[36] Docker Documentation. [online]. [cit. 2018-04-18]. Dostupné z: https://docs.docker.com/

[37] Ubuntu-ssh. In: *Github* [online]. [cit. 2018-04-18]. Dostupné z: https://github.com/JenNava/ubuntu-ssh

[38] Clustering Guide. [online]. [cit. 2018-03-09]. Dostupné z: https://www.rabbitmq.com/clustering.html

[39] Installing on Debian and Ubuntu. [online]. [cit. 2018-03-08]. Dostupné z: https://www.rabbitmq.com/install-debian.html

ZOZNAM VELIČÍN A SKRATIEK

AI	artificial intelligence
AMQP	advanced message queuing protocol
API	application programming interface
CPU	central processing unit
DNA	deoxyribonucleic acid
FIFO	first in – first out
GA	genetický algoritmus
GiB	gibibajt
GHz	gigahertz
INIT	initialization
IP	internet protocol
JSON	javascript object notation
KiB	kibibajt
m	minúta
MiB	mebibajt
MPI	message passing interface
NFS	network file system
NP	nondeterministic polynomial
RHEL	red hat enterprise linux
RNA	ribonucleic acid
RPC	remote procedure call
s	sekunda
SSH	secure shell
TCP	transmission control protocol
UML	unified modeling language
UTF	unicode transformation format

ZOZNAM PRÍLOH

A NÁVOD NA SPUSTENIE

Príprava RabbitMQ serveru

RabbitMQ server sa inštaluje pomocou návodu na oficiálnej stránke RabbitMQ [39]. Je nutné podotknúť, že inštalácia RabbitMQ vyžaduje taktiež nainštalovanie podpory pre jazyk Erlang. Funkčnosť je možné overiť príkazom *rabbitmqctl status*. Prihlasovanie na RabbitMQ server sa realizuje pomocou už existujúceho účtu s menom *guest* a prihlasovacím heslom *guest*. Pri prihlasovaní zo vzdialenej pracovnej stanice je však potreba vytvoriť nový účet. Ten sa vytvorí týmito príkazmi:

```
rabbitmqctl add_user {meno} {heslo}
rabbitmqctl set_user_tags {meno} administrator
rabbitmqctl set_permissions -p / {meno} ".*" ".*" ".*"
```

RabbitMQ poskytuje taktiež webové rozhranie pre správu účtov, kanálov, spojení a pre vyčítanie štatistík. Pripojiť sa na toto rozhranie možno pomocou webového prehliadača na adresu *http://meno-serveru:15672/*. V prípade, že nemožno použiť webový prehliadač alebo je potreba dát v JSON formáte, možno využiť taktiež API, ktoré RabbitMQ poskytuje na adrese *http://meno-serveru:15672/api/*. Následne možno nájsť na oficiálnych stránkach RabbitMQ aj dokumentáciu k API v príslušnej verzii. Webové rozhranie k RabbitMQ sa inštaluje nasledujúcim príkazom (druhým príkazom možno skontrolovať, či sa rozhranie korektne nastavilo a taktiež možno zistiť verziu, v ktorej sa nainštalovalo):

```
rabbitmq-plugins enable rabbitmq_management
curl -i -u meno:heslo "http://server:15672/api/overview"
```

Nakoniec je vhodné pripomenúť, že aj keď je v prostredí nutný RabbitMQ server len na jednej zo staníc, pre efektívnejšiu komunikáciu medzi pracovnými stanicami je možnosť využiť RabbitMQ server na každej stanici v móde *cluster*. Nastavovanie tohto módu je popísané v návode na oficálnej stránke RabbitMQ [38].

Príprava Python modulu paralelného GA

Príprava modulu paralelného GA na každej pracovnej stanici (pre operačný systém Ubuntu vo verzii 17.10) pozostáva z týchto príkazov:

```
apt-get install openssh-server
apt-get install python3.6
apt-get install python3-pip
python3.6 -m pip install parallel_ga_processing
```

Ako už bolo spomenuté, pre použitie SSH je výhodné využiť prihlasovanie pomocou verejných kľúčov. Najprv sa vygenerujú kľúče na každej stanici pomocou linuxového nástroja *ssh-keygen* a následne sa skopírujú verejné kľúče pomocou linuxového nástroja *ssh-copy-id*. Tie treba skopírovať z každej stanice, na ktorú je potreba sa prihlásiť, na stanicu, na ktorej sa spúšťa SCOOP.

Nakoniec je potreba vytvoriť Python súbory. Na zvolenej zdrojovej stanici sa vytvorí na ľubovoľnej ceste v adresárovom strome zdrojový súbor s týmto kódom (hodnoty môžu byť ľubovoľné):

```
1 from parallel_ga_processing import algorithmRunners
2
3 if __name__ == '__main__':
4     algorithmRunners.launch(
5         hosts_list=["localhost", "user@192.168.56.2"],
6         num_of_workers=36, executable="run.py",
7         path="/tmp/")
```

V prípade viacerých staníc je nutné zvoliť si cestu rovnakú pre všetky stanice. Nemožno preto zvoliť cestu */home/user/*, keďže stanice môžu mať rôznych užívateľov. V tejto práci sa využívala cesta */tmp/*. Treba však pripomenúť, že operačný systém všetky dáta v tomto adresári pri vypnutí maže.

Následne je potreba vytvoriť súbory na zvolenej ceste a to na každej stanici. Súbor v prípade použitia jemnozrnného a hrubozrnného modelu vyzerá podobne, preto boli zvolené príklady pre hrubozrnný a master-slave model.

Súbor *run.py* na zdrojovej stanici pre hrubozrnný model:

```
1  from parallel_ga_processing import algorithmRunners
2
3  def fitness(chromosome):
4      return sum(gene * gene for gene in chromosome)
5
6  if __name__ == '__main__':
7      algorithmRunners.run_coarse_grained_ga(
8          population_size=(6, 6), deme_size=10,
9          chromosome_size=4, number_of_generations=10,
10         neighbourhood_size=1, server_user="genetic1",
11         server_password="genetic1",
12         server_ip_addr="192.168.56.1",
13         num_of_migrants=1, fitness=fitness)
```

Súbor *run.py* na ostatných staniciach pre hrubozrnný model (v prípade použitia modulu len na jednej pracovnej stanici sa tento kód nepoužíva):

```
from parallel_ga_processing import algorithmRunners

def fitness(chromosome):
    return sum(gene * gene for gene in chromosome)

algorithmRunners.run_coarse_grained_ga_remote(
    population_size=(6, 6), deme_size=10,
    chromosome_size=4, number_of_generations=10,
    neighbourhood_size=1, server_user="genetic1",
    server_password="genetic1", num_of_migrants=1,
    server_ip_addr="192.168.56.1",fitness=fitness)
```

Parametre museli byť nastavené v súboroch na všetkých staniciach zhodne. To platí aj pre parameter *server_ip_addr*, ktorý musel byť nastavený na rovnakú IP adresu RabbitMQ serveru. Nemožno teda na zdrojovej stanici nastaviť IP adresu 127.0.0.1, aj keď logicky je to správne. Toho obmedzenie je zapríčinené modulom SCOOP.

Súbor *run.py* na zdrojovej stanici pre master-slave model:

```
from parallel_ga_processing import algorithmRunners

def fitness(chromosome):
    return sum(gene * gene for gene in chromosome)

if __name__ == '__main__':
    algorithmRunners.run_master_slave_ga(
        population_size=36,
        chromosome_size=4, number_of_generations=10,
        fitness=fitness)
```

Súbor *run.py* na ostatných staniciach pre master-slave model (v prípade použitia modulu len na jednej pracovnej stanice sa tento kód nepoužíva):

```
def fitness(chromosome):
    return sum(gene * gene for gene in chromosome)
```

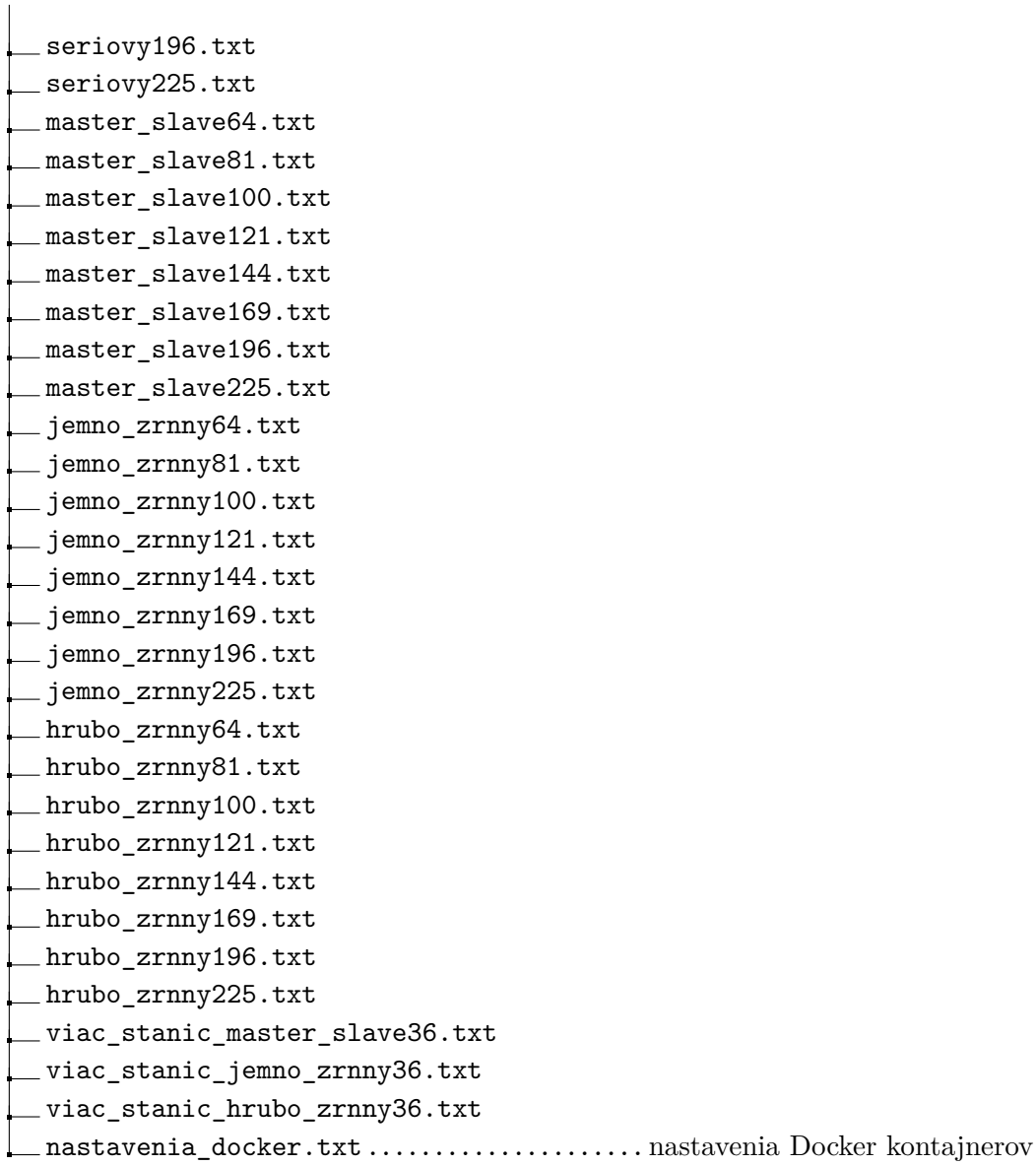
Modul sa nakoniec spustí týmto príkazom:

```
python3.6 {cesta-k-zdrojovému-súboru}
```

B OBSAH PRILOŽENÉHO CD

V tejto elektronickej prílohe sa nachádza elektronická verzia tejto práce, výsledný kód implementácie a záznamy, ktoré boli vygenerované modulom pri testovaní. Názvy súborov so záznamami sú tvorené typom paralelného GA a číslom, ktoré označuje veľkosť populácie pri danom meraní. Taktiež sa tu nachádzajú záznamy vygenerované pri testovaní modulu s paralelizáciou na viacerých pracovných staniciach. Nakoniec je prítomný aj textový súbor s nastaveniami Docker kontajnerov, ktoré boli taktiež použité pri paralelizácii na viacerých pracovných staniciach.

```
/ ..... koreňový adresár priloženého CD
├── diplomova_praca.pdf ..... text práce v elektronickej verzii
├── parallel_ga_processing ..... kód modulu GA
│   ├── examples ..... ukážkové kódy pre spúšťanie modulu
│   │   ├── __init__.py
│   │   ├── runCoarseGrainedExample.py
│   │   ├── runFineGrainedExample.py
│   │   └── runMasterSlaveExample.py
│   ├── parallel_ga_processing
│   │   ├── algorithmRunners
│   │   │   ├── __init__.py
│   │   │   ├── coarseGrainedGaRunner.py
│   │   │   ├── fineGrainedGaRunner.py
│   │   │   ├── launcher.py
│   │   │   └── masterSlaveGaRunner.py
│   │   ├── geneticAlgorithms
│   │   │   ├── __init__.py
│   │   │   ├── coarseGrainedBase.py
│   │   │   ├── decorator.py
│   │   │   ├── fineGrainedBase.py
│   │   │   ├── geneticBase.py
│   │   │   ├── geneticGrainedBase.py
│   │   │   ├── masterSlaveBase.py
│   │   │   └── messenger.py
│   │   └── __init__.py
│   ├── tests ..... testy
│   │   └── test_parallel.py
│   ├── __init__.py
│   └── setup.py ..... nastavenie pre nástroj Pip
└── logs ..... záznamy modulu pri meraní
    ├── seriovy64.txt
    ├── seriovy81.txt
    ├── seriovy100.txt
    ├── seriovy121.txt
    ├── seriovy144.txt
    └── seriovy169.txt
```


seriovy196.txt
seriovy225.txt
master_slave64.txt
master_slave81.txt
master_slave100.txt
master_slave121.txt
master_slave144.txt
master_slave169.txt
master_slave196.txt
master_slave225.txt
jemno_zrnny64.txt
jemno_zrnny81.txt
jemno_zrnny100.txt
jemno_zrnny121.txt
jemno_zrnny144.txt
jemno_zrnny169.txt
jemno_zrnny196.txt
jemno_zrnny225.txt
hrubo_zrnny64.txt
hrubo_zrnny81.txt
hrubo_zrnny100.txt
hrubo_zrnny121.txt
hrubo_zrnny144.txt
hrubo_zrnny169.txt
hrubo_zrnny196.txt
hrubo_zrnny225.txt
viac_stanic_master_slave36.txt
viac_stanic_jemno_zrnny36.txt
viac_stanic_hrubo_zrnny36.txt
nastavenia_docker.txt
nastavenia Docker kontajnerov