Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Bakalářská práce

Systém pro paralelní spouštění obecného genetického algoritmu

Plzeň 2015 Jindřich Pouba

Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 7. května 2015

Jindřich Pouba

Poděkování

Rád bych poděkoval svému vedoucímu bakalářské práce, panu Ing. Tomáši Potužákovi, Ph.D., za vstřícnost, upřímnost a dobré rady během zpracování této práce.

Abstract

This thesis explores the options and functionality of genetic algorithms for general problem solving and options of their parallelization and execution in distributed environment. The goal of this thesis is to implement application, that will allow execution of general genetic algorithm in parallel and/or distributed computational environment and to test this application with respect to speedup that parallelism brings. The goals were met, implemented application shows noticeable speedup compared to sequential algorithm, which was confirmed by tests and measurements.

Keywords: genetic algorithm, parallelization, distributed computing

Abstrakt

Tato práce prozkoumává možnosti a funkce genetických algoritmů při řešení obecných problémů, možnosti jejich paralelizace a spouštění v distribuovaném prostředí. Cílem práce je napsat aplikaci, která bude umožňovat spouštění obecného genetického algoritmu v paralelním a/nebo distribuovaném výpočetním prostředí a tuto aplikaci otestovat zejména s ohledem na urychlení, které paralelizace přináší. Cíle byly splněny, napsaná aplikace přináší znatelné urychlení oproti sekvenčnímu algoritmu, což bylo potvrzeno testy a měřením.

Klíčová slova: genetický algoritmus, paralelizace, distribuované výpočty

Obsah

1 Úvod

Genetické algoritmy jsou metody řešení problémů, které využívají principy přírodního výběru a genetiky pro řešení složitých problémů z oblasti informatiky, financí, průmyslu i vědy[3, 13, 14]. Tyto metody jsou heuristické, což znamená, že jimi nalezená řešení nemusí být nejlepší, ale pro daný problém jsou *dostačující*. Termín *dostačující* je pro každý typ problému jiný, někdy algoritmus pro výpočet nejlepšího řešení neexistuje nebo není známý. Jindy by jeho výpočet trval tak dlouho, že by nalezené řešení po takové době ztratilo smysl [1, 2, 4].

Genetický algoritmus pracuje s určitou množinou řešení, kterou aplikací různých postupů postupně vylepšuje, dokud není řešení prohlášeno za dostačující. Genetický algoritmus je vhodný kandidát na paralelizaci, protože zde se velikost množiny řešení přímo odráží na časové náročnosti výpočtu a na kvalitě výsledného řešení. Větší množina řešení vyžaduje větší výpočetní výkon, kterého se dá lépe a levněji dosáhnout právě paralelizací na víceprocesorových počítačích (paralelní výpočetní prostředí) a/nebo rozdělením výpočtu mezi více počítačů (distribuované výpočetní prostředí) [4, 11].

Cílem této práce bylo vytvořit systém pro paralelní spuštění obecného genetického algoritmu v distribuovaném protředí a otestovat jeho funkčnost na několika specifických problémech. Nekteré tyto problémy budou vloženy přímo do aplikace, ale zároveň bude umožněno uživateli vkládat vlastní problémy (definovat a přidávat do aplikace vlastní fitness funkce) za běhu aplikace a bez nutnosti rekompilace programu.

O genetických algoritmech pojednává kapitola 2. Paralelizací obecně se zaobírá kapitola 3, paralelizací genetických algoritmů kapitola 4. O návrhu a implementaci samotné aplikace se píše v kapitolách 5 a 6. Testování funkcionality řeší kapitola 7 a měření výkonu je provedeno v kapitole 8.

2 Genetické algoritmy

Genetické algoritmy jsou heuristické postupy, které aplikují principy přírodního výběru, známé z biologie, na řešení složitých problémů z mnoha různých oborů lidské činnosti [3]. Genetický algoritmus si uchovává množinu řešení, zvanou *populace*, na kterou aplikuje operátory *křížení*, *mutace* a *výběr*. Každý z těchto postupů má za úkol simulovat evoluční techniku a postupně vylepšovat množinu řešení. Po aplikaci všech operátorů vstupní množinu řešení se postup opakuje a výsledná množina se stává vstupní do nového kola iterace. Populace v jedné iteraci se obvykle nazývá *generace*. Jedno řešení z množiny se také jinak označuje jako *jedinec* [1, 2, 4, 9].

Obvyklý postup popsaný pseudokódem:

```
population = initialize();
// first population is usually chosen randomly

for (each member of the population) computeFitness();

while (!condition()) {
   population = select(population);
   population = crossover(population);
   population = mutate(population);
   for (each member of population) computeFitness();
}
```

V následujících sekcích si postupně projdeme všechny části algoritmu. Výběr jedinců, v pseudokódu reprezentovaný funkcí *select()* je obsahem sekce 2.2. Křížení jedinců (funkce *crossover()*) řeší sekce 2.3, mutace (*mutate()*) najdeme v sekci 2.4 a ukončovací podmínka algoritmu *condition()* uzavírá výčet sekcí 2.5. Ještě předtím by ale bylo dobré se zmínit o počítačové reprezentaci jedince.

2.1 Reprezentace jedince

Jedinec představuje jednu možnost řešení daného problému. Pokud si řešený problém představíme jako funkci, pak jedinec je jejím vstupním parametrem.

Každý jedinec se může brát jako jeden vstupní parametr, ale ve většině případů je řešený problém složitá struktura, závisející na velkém množství vstupních parametrů. Jedinec je tedy ve skutečnosti tvořen sadou vlastností, které mu umožňují řešit daný problém. Popis každé této vlasnosti je uložen v jeho genetickém kódu a nazývá se *gen*. Soubor genů je obvykle nazýván *chromozom*. V biologii má jedinec obyvkle chromozomů více, ale pro účely genetických algoritmů a ve zbytku této práce budeme předpokládat, že *chromozom* je zápis všech genů jednoho jedince. Všechny geny všech jedinců v celé populaci se nazývají souhrným názvem *genofond* [12].

Nejčastěji používaný zápis genomu je lineání uspořádání genů, což znamená, že i-tý gen vždy reprezentuje stejnou vlastnost jedince. Pro jednotlivé geny se nejčastěji používá zápis pomocí řetězce binárních čísel nebo znaků dané délky. Chromozom si tak můžeme představit jako řadu znaků, popřípadě jedniček a nul. Reprezentace řetězcem je univerzální, proto na ní bude kladen největší důraz v průběhu práce [1, 2].

Jiné reprezentace chromozomů jsou také možné, například grafy, stromy, matice nebo kombinace prvků. Tyto zápisy kladou na křížení a mutace další podmínky (struktura stromu se nesmí porušit, graf musí zůstat souvislý, všechny prvky se musí v kombinaci objevit právě jednou a podobné), které závisí na charakteru řešené úlohy. Tyto reprezentace nejsou obecné a z hlediska této práce zajímavé [12].

2.2 Výběr

Výběr jedinců neboli *selekce* je pro další postup algoritmu důležitá část. Zde se určuje, která řešení poskytnou kvalitní geny do dalších generací a která řešení by pouze plýtvala výpočetním časem. Jak moc se daný jedinec hodí do další generace určuje hodnota tzv. *fitness funkce*, která popisuje schopnost jedince řešit daný problém. Jedince, kteří daný problém řeší dobře (lépe než ostatní) považujeme za nositele dobrých vlastností a princip dědičnosti říká, že potomci, kteří od nich tyto vlastnosti zdědí, budou mít také větší šanci na úspěch při řešení zadání [1, 2].

Fitness funkce tedy musí nějak odrážet schopnost jedince řešit daný problém. Jedinný další požadavek na tuto funkci je ten, že její hodnoty musí jít snadno porovnávat mezi sebou. Tak můžeme snadno zjistit, kteří jedinci jsou vhodnější k další reprodukci. Úplně nejčastější obor hodnot fitness funkce je

obor reálných čísel. Funkce udává pouze schopnost jedince řešit daný problém, ale neříká už ovšem to, jestli je dané řešení vhodné pro další reprodukci. Například někdy může být vhodné vybrat slabší jedince pouze kvůli tomu, že obohatí genofond. Jinak by se mohlo stát, že celá generace uvízne na lokálním maximu, ze kterého už se pomocí křížení nedostane (tento problém ovšem částečně řeší mutace, viz sekce 2.4) [1, 2].

Dále je dobré poznamenat, že při některých druzích selekce může být jedinec vybrán i vícekrát, v extrémních případech může být několikanásobně vybrán pouze jeden a tak zcela ovládnout generaci. Tento stav je nežádoucí, protože generace tvořená stejnými geny už se kromě mutace nijak nevyvíjí a neprohledává prostor řešení. Stručný popis nejčastěji používaných selekcí se nachází v následujících kapitolách.

2.2.1 Ruleta

Nejdříve se jedinci seřadí podle hodnoty fitness funkce a poté dojde k *normalizaci* této hodnoty. Normalizace spočívá v tom, že se hodnoty zmenší / zvětší tak, aby jejich poměr mezi sebou zůstal stejný, ale celkový součet byl rovný právě 1. Tyto normalizované hodnoty se použijí v výpočtu *kumulativních normalizovaných hodnot* pro každého jedince. Kumulativní normalizovaná hodnota je rovná sumě normalizované hodnoty a všech větších normalizovaných hodnot. První jedinec má tedy tuto hodnotu rovnou fitness funkci, hodnota druhého je součet fitness prvního a druhého jedince, atd. Poslední na seznamu musí mít tuto hodnotu rovnou jedné [2, 6, 10].

Dále se generují náhodná čísla od 0 do 1. Za každé takto vygenerované číslo se zvolí řešení, které má nejbližší vyšší kumulativní normalizovanou hodnotu. Generování náhodných čísel se opakuje dokud není vybrán dostatek jedinců [2, 6, 10].

Ruleta zajišťuje, že vhodnější jedinci mají větší šanci být vybráni, přesně podle hodnot fitness funkce. V případech, kdy jeden nebo malé množství jedinců má výrazně vyšší hodnoty fitness funkce než zbytek populace, může tento způsob být nevýhodný, protože je vysoká pravděpodobnost, že bude vybrán malý počet jedinců vícekrát a populace tak ztratí genovou diverzitu [2, 6, 10].

Tato metoda selekce také předpokládá, že hodnoty fitness funkce jsou čísla, která navíc musí být kladná.

2.2.2 Výběr podle pořadí

Postup je stejný jako u rulety, ale ještě před začátkem rulety se jako hodnota fitness funkce použije pořadí jedince od konce. Poslední jedinec bude mít hodnotu 1, předposlední 2, ..., druhý *n-1* a první *n*. Tento systém číslování nezohledňuje samotné hodnoty fitness funkce, ale zároveň eliminuje problém, že by jeden nebo malé množství jedinců ovládlo celou ruletu. Také se dá použít v případech, kdy hodnoty fitness nejsou kladná čísla [6, 10].

2.2.3 Turnaj

Vždy se vybere n jedinců z celé generace. Mezi nimi se uspořádá „turnaj“ a jeho vítěz (nejlepší z výběru) postoupí do další generace. Postup se opakuje a pořádají se další turnaje dokud není počet jedinců dostatečný [10].

2.2.4 Výběr prvních n

Nejjednodušší metoda výběru. Jedinci se seřadí podle fitness funkce a vybere se prvních n. Tato metoda nijak nezohledňuje samotné hodnoty fitness funkce, ale zajišťuje, že každý jedinec bude vybrán pouze jednou [6, 10].

2.3 Křížení

Křížení, anglicky *crossover*, je proces, při kterém ze dvou nebo více rodičů vznikne jeden nebo více potomků. Přitom musí platit principy dědičnosti, takže potomek dědí své geny přímo od jednoho z rodičů. Mohou však vznikat takové kombinace genů a tedy vlasnosti jedince, které ani jeden z rodičů nemá. Analogie pro člověkla platí například pro pohlaví, které (pro ilustraci) tvoří jeden gen a proto ho potomek může zdědit pouze od jednoho z rodičů. Oproti tomu krevní skupinu tvoří celý soubor genů a může se tak stát, že potomek má jinou krevní skupinu než oba rodiče [1, 2, 4].

Je hodno poznamenat, že rodiče při procesu křížení v genetických algoritmech mohou, ale nemusí zanikat. Proto pokud budeme mluvit o „výměně“ rodičovských genů, máme na mysli výměnu *kopií* genů rodičů.

2.3.1 Jednobodové křížení

Tato jednoduchá metoda křížení vyžaduje dva rodiče o stejné délce chromozomů. Náhodně se zvolí *bod řezu*, tj. místo v chromozomu, které rozdělí chromozom na dvě části. Potomci poté vznikají prostou výměnou částí chromozomu za bodem řezu. Například rodiče **AAAAA** a **BBBBB** pro bod řezu 2 vyprodukují potomky **AA**BBB a **BB**AAA [5, 6].

Je vidět, že pro hodnoty bodu řezu, které jsou příliš na okrajích chromozomu, má výsledný jedinec většinu genů právě od jednoho z rodičů.

Bod řezu se pro každé křížení náhodně generuje. Pokud by byl pevně zvolen, úloha by se v podstatě rozpadla na dva samostatné genetické algoritmy, každý pro jednu část chromozomu a fitness funkce by v podstatě udávala „kompatibilitu“ obou částí.

2.3.2 Dvoubodové křížení

Tato forma je podobná jednobodovému křížení. Rozdíl je pouze v tom, že body řezu jsou dva a část chromozomu určená v výměně je mezi nimi. Také si tento proces můžeme představit jako jednobodové křížení, ale provedené dvakrát. Stejný příklad s rodiči **AAAAA** a **BBBBB** by mohl generovat potomky **ABBAA** a **BAABB** [5, 6].

2.3.3 *n*-bodové křížení

Stejně jako jsme při přechodu z jednobodového křížení vytvořili dvoubodové přidáním dalšího bodu řezu, můžeme přidat n bodů řezu a vytvořit tak n-bodové křížení. Maximální hodnota n je rovna délce chromozomu minus 1 [5].

Je dobré poznamenat, že čím více se hodnota n blíží délce chromozomu, tím klesá počet genů, které se mezi rodiči mění opravdu náhodně. Pro maximální hodnotu n se problém opět rozdělí na dva genetické algoritmy, jeden pro sudé a druhý pro liché geny.

2.3.4 „Cut and splice“ crossover

Tento postup pracuje stejně jako jednobodové křížení, ovšem každý rodič má jiný bod řezu. Výslední potomci jsou vyvořeni jako obvykle spojením části jednoho rodiče před bodem řezu a části za bodem řezu druhého rodiče. Tento postup generuje potomky s rozdílnými délkami chromozomů. Příklad pro rodiče **AAAAA** a **BBBBB** může generovat potomky **AAABBBB** a **BAA**. Je tedy potřeba mít fitness funkci, která je připravená na rozdílné délky chromozomů jedinců [5].

Toto křížení navíc vkládá na místa, kde byly uloženy geny pro nějakou vlastnost, geny, které v rodiči mohly určovat vlastnosti jiné. Ukážeme si na příkladu. Geny rodičů očíslujeme $A_1A_2A_3A_4A_5A_6$ a $B_1B_2B_3B_4B_5B_6$. Nechť každý gen nebo kombinace určuje nějakou vlasnost jedince, tedy vlastnost α je určena prvním genem, β druhým, γ třetím a δ nechť je celý zbytek chromozomu, což eliminuje problém s rozdílnými délkami. Pořád se ale může stát, že délka jedince bude nedostatečná pro určení všech vlastností - tento problém budeme pro tento příklad zanedbávat. Zvolíme body řezu 2 a 4, tedy $A_1A_2||A_3A_4A_5A_6$ a $B_1B_2B_3B_4||B_5B_6$. Výslední potomci budou vypadat takto:

$$A_1A_2B_5B_6$$

$$B_1B_2B_3B_4A_3A_4A_5A_6$$

Zde vidíme, že vlastnost γ, kterou dříve určovaly geny s čísly 3 je nyní v prvním jedinci určena genem B_5, který se v rodiči podílel na vlasnosti δ. Tento postup je u obecného zadání problému v rozporu s principem dědičnosti, jelikož vlastnosti jedinců přestávají záviset na rodičích, ale spíše na zvolených bodech řezu i.e. na náhodě. Proto je vhodný pouze pro některé specifické problémy.

2.3.5 Rovnoměrné křížení

Tento styl křížení rozhoduje u každého genu zvlášť, od kterého rodiče se bude dědit. Vyžaduje dva nebo více rodičů se stejnými délkami a funkci, která vybere rodiče pro každý gen na základě náhody. Pro dva rodiče se pravděpodobnost výběru nastavuje typicky na 50% pro oba. Pro více rodičů se typicky nastaví obdobně jako $\frac{100}{n}\%$, kde n je počet rodičů. Další možný a využívaný způsob je rozdělit šanci podle hodnoty fitness funkce [5, 10].

Pro dostatečně velký počet genů se poměr genů od rodiče přibližně rovná jeho pravděpodobnosti výběru. Nemusí to ovšem být pravda pro každé jednotlivé křížení, proto se algoritmus dá ještě upravit tak, aby dané poměry dodržoval přesně [5].

2.3.6 Další možnosti křížení

Pro jiné než řetězcové zápisy chromozomů se mohou tyto možnosti křížení ukázat jako nevhodné. Pokud je například v chromozomu zakódována posloupnost, mohou jí některé možnosti křížení porušit tak, že se některé prvky budou v posloupnosti vyskytovat vícekrát a jiné vůbec. Nebo pokud je v chromozomu zakódován strom, mohly by některé možnosti křížení porušit jeho strukturu (vytvořit cykly) [6, 12].

Pro tyto případy je potřeba definovat specifické metody křížení pro každý problém. Tyto metody nejsou obecné a z hlediska této práce nezajímavé. Jsou tedy pouze zmíněny, ale dále nepopisovány.

2.4 Mutace

Mutace v případě genetického algoritmu je záměrná změna jednoho nebo více genů u jednoho jedince. Její primární cíl je zvýšit genetickou diverzitu v populaci. Prostým křížením se totiž nikdy nemohou v populaci objevit hodnoty genů, které v ní nebyly přítomny na začátku celého algoritmu [1, 2, 4].

Dalším přínosem mutace je, že nutí algoritmus prohledávat větší prostor řešení tím, že změnou některého genu donutí jedince „skočit“ v prostoru řešení na jiné místo. Tento postup zabraňuje algoritmu uvíznout na lokálním maximu, kdy by většina jedinců byla tolik podobná nebo dokonce stejná, což by efektivně zpomalilo nebo úplně zastavilo evoluci. Některé mutace mění hodnoty genů jen málo a tím pomáhají zlepšovat již nalezené hodnoty [1, 2].

Příliš mnoho mutace ovšem potlačuje evoluci a přináší velkou míru náhody do řešení problémů. Mutace totiž nezávisí na hodnotě fitness funkce ani na genech rodičů, ale pouze na náhodě [1, 2].

2.4.1 Převrácení bitu

U chromozomů, které tvoří pouze řada jedniček a nul je jedinná možnost, jak změnit hodnotu genu a to jeho převrácení ($0 \rightarrow 1$, $1 \rightarrow 0$). Počet bitů k převrácení může být pevně definován nebo může být definována šance na převrácení u každého bitu [6, 17].

2.4.2 Nahrazení hraniční hodnotou

Pokud je gen reprezentován jako číslo v nějakém roszahu, můžeme v případě mutace toto číslo nahradit hraniční hodnotou pro daný interval (minimum / maximum). Například u procent je to 0 nebo 100. Kterou hodnotu použít můžeme zvolit náhodně v poměru 50:50 nebo fixně zvolit na začátku algoritmu.

2.4.3 Postupné snižování hodnoty mutace

Pokud dojde k mutaci, je hodnota genu změněna o určitou hodnotu. Tato míra změny se postupně snižuje s počtem generací. Na začátku algoritmu se tedy hodnoty genů mění více, aby se prohledal co největší prostor řešení. Ke konci se hodnoty mění málo, aby se již pouze zlepšovala nalezená řešení [18].

Zde je těžké určit, jakou rychlostí se má míra mutace měnit. Většinou záleží na problému, velikosti prostoru řešení, na použitém křížení a počtu jedinců v populaci.

2.4.4 Rovnoměrná mutace

Mutovaný gen se nahradí náhodnou hodnotou z oboru hodnot pro daný gen. Rozložení hodnot této náhodné funkce je rovnoměrné, každá hodnota má stejnou šanci být vybrána [6, 17, 18].

2.4.5 Gaussovská mutace

K hodnotě genu se přičte nebo odečte náhodná hodnota, určená náhodně podle Gaussovské křivky. Gen má tedy větší šanci se změnit pouze o malou hodnotu než o velké číslo. Pokud dojde k přetečení / podtečení oboru hodnot, je hodnota nahrazena hraniční.

2.4.6 Další mutace

Stejně jako u křížení může pro specifické zápisy chromozomů mutace učinit dané řešení nepoužitelným. Proto je u specifických problémů nutné zvolit specifické mutace. Například pokud je v genech zakódována permutace množiny, mohla by změna některé hodnoty nějaké prvky vynechat a jiné použít vícekrát, popřípadě přidat prvky, které se v dané množině vůbec nevyskytují. U tohoto příkladu by se jako vhodná mutace dalo použít náhodné prohození dvou hodnot z chromozomu [6].

Další příklad může tvořit chromozom, kde na sobě geny nějakým způsobem závisí. Matice sousednosti neorientovaného grafu například musí být symetrická. Náhodná změna některého prvku v matici by jí učinila nesymetrickou a tím nevhodnou k dalšímu zpracování. Jedna z použitelných mutací by v tomto případě musela měnit dva na sobě závislé geny najednou [6].

2.5 Ukončovací podmínka

Genetický algoritmus jako takový nemá konec a je tedy na autorovi nebo uživateli, aby určil, kdy se má zastavit. V podstatě existují dva přístupy k problému, první závisí na hodnotě fitness funkce, druhý na čase nebo výpočetních kapacitách [8].

2.5.1 Hodnota fitness funkce

V případě hodnoty fitness funkce zastavujeme algoritmus v případě, že hodnota fitness funkce dosáhla určité hranice. Tento styl ukončení se používá v případech, kdy hledáme řešení, které má určité kvality, které je nutné

splnit, nebo výsledek spadá do určitého oboru přípustnosti. Na výstupu máme jistotu, že dané řešení je dostačující, ovšem na začátku nevíme nic o tom, jak dlouho bude trvat se k němu dopracovat. Navíc, pokud špatně zvolíme ukončovací podmínku nebo dané řešení prostě neexistuje, může se stát, že algoritmus neskončí nikdy. Typicky se jedná o problémy zadané stylem [8]:

- Najdi hodnoty vstupů pro které je hodnota fitness funkce alespoň 10
- Najdi řešení, které dokáže vyřešit alepoň 8 z 10 zadaných problémů[1]
- Vytrénuj klasifikátor tak, že bude mít úspěšnost alespoň 70%

2.5.2 Výpočetní kapacity

Pokud nám zaleží spíše na čase nebo výpočetních kapacitách, zvolíme konec algoritmu závislý na čase, počtu generací nebo počtu vyzkoušených jedinců. Často se jedná o problémy, kde nevíme nic o prostoru řešení, ani nemůžeme odhadovat hodnoty fitness funkce. Také se může jednat o problémy, kde každé lepší nalezené řešení zlepšuje užitek, který ovšem musíme porovnávat s výdaji na výpočetní kapacity. U tohoto zadání máme jistotu, že algoritmus někdy skončí, ovšem nevíme toho moc o nalezeném řešení. Možná je nejlepší možné, ale možná by algoritmus našel mnohem lepší řešení, kdyby běžel ještě nějaký čas. Požadavky můžeme zadat stylem [8]:

- Prováděj algoritmus tři hodiny a předlož nejlepší nalezené řešení
- Ukonči algoritmus po dvaceti generacích

2.5.3 Kombinace obou metod

Další možnost, která kombinuje oba postupy, je zadat ukončovací podmínku na základě rychlosti růstu fitness funkce. U většiny problémů, které genetické algoritmy řeší, totiž rychlost objevování lepších řešení postupně klesá. Čím více generací, tím větší je již prohledaný prostor řešení a tím menší je

[1] použití genetických algoritmů pro řešení problémů typu ano/ne nemusí být vhodné, více v sekci 2.6

šance, že při dalším prohledávání bude nalezeno lepší řešení. Pokud zastavovací podmínku nastavíme v závislosti na rychlosti růstu maximální hodnoty fitness funkce nalezených řešení, můžeme s určitou šancí říci, že další prohledávání by stálo více úsilí a výpočetního výkonu, než by nalezené zlepšení přineslo zisku. Pokud budeme předpokládat, že prostor řešení je konečný, můžeme s určitostí říci, že algoritmus *někdy skončí*. Nevíme sice čas, ale víme, že se tak stane. Zároveň nemůžeme nic říci o nalezeném řešení - může být nejlepší, ale také se může jednat o lokální maximum. Ukončovací podmínku pak definujeme stylem [8]:

- Pokud se hodnota fitness nezlepší po tři generace, prohlaš dané řešení za nejlepší nalezené
- Pokud se hodnota fitness zlepší mezi generacemi o méně než 5%, skonči

2.6 Omezení

Genetické algoritmy mají svá omezení a nehodí se na řešení všech typů problémů. Mezi nejčastější problémy patří výpočetní náročnost na mnohonásobný výpočet fitness funkce, který může být náročný sám o sobě [4]. Právě tento problém částěčně řeší nebo alespoň zlehčuje paralelizace a distribuce výpočtu fitness funkcí mezi více výpočetních stanic. Více v kapitole 4. Další možnost jak ulehčit výpočtu je místo složitých fitness funkcí použít jejich aproximace. Tento způsob sice může zhoršit kvalitu nalezených jednotlivých řešení, ale prozkoumá jich více a tak při vhodně zvolené aproximaci konverguje rychleji k lepším řešením [4, 11].

„Nejlepší řešení“ je vždy nejlepší pouze při porovnání s ostatními řešeními. Na konci algoritmu nemáme matematický ani jiný důkaz, že dané řešení je opravdu dobré. Tento fakt komplikuje volbu ukončovací podmínky [8].

Může se stát, že populace uvízne v lokálním maximu na úkor průzkumu celého prostoru řešení a nalezení globálního maxima. Tento problém z velké části závisí na povaze problému a tvaru prostoru řešení. Možností k ulehčení od tohoto problému je několik, některé přidávají *pokutu* k fitness funkci, pokud je na jednom místě příliš podobných jedinců, jiné metody při detekci přílišné podobnosti jedinců v generaci přidají novou náhodně generovanou množinu řešení. Také můžeme prostě zvýšit počet mutací nebo zvolit jinou

fitness funkci. Většina těchto řešení ovšem přidává výpočetní nároky [1, 2, 4, 6, 9].

Genetické algoritmy se také špatně používají na problémy typu ano / ne. Genetické postupy jsou založeny na postupném zlepšování stávajících řešení, ovšem u rozhodovacích problémů není způsob jak poznat, kterým směrem se vydat, která řešení jsou lepší než jiná. V tomto případě jsou algoritmy založené na náhodném tipování a testování řešení stejně účinné jako genetické postupy, ne-li účinější. Problém se částečně dá řešit, když můžeme rozhodovací problém opakovat pro různá zadání a jako fitness funkci použít úspěšnost alogritmu. Ovšem záleží na zadání a na tvaru prostoru řešení [1, 4, 6].

2.7 Použití

Genetické algoritmy mají nejčastější využití v optimalizačních úlohách, jako je plánování tras nebo organizace času. Mezi další příklady využití patří [3, 13, 14, 17, 18]:

- Počítačem řízený desing
- Trénink neuronových sítí
- Návrh složitých chemických sloučenin a jejich modelů
- Automatické řízení strojů
- Řízení a simulace silniční dopravy
- Umělá inteligence v počítačových hrách
- Předpovědi vývoje ekonomiky a akciových trhů

3 Paralelizace obecně

V minulosti byla paralelizace výpočtů doménou výzkumných výpočetních center. Zde se řešily problémy vyžadující obrovský výpočetní výkon, který nebylo možné jinak dosáhnout. V poslední době se ovšem tento obor informatiky stále více dostává do popředí, společně s dostupností vícejádrových procesorů [4, 11].

3.1 Základní pojmy

3.1.1 Sekvenční algoritmus

Sekvenční algoritmus je tvořen posloupností instrukcí, které se vykonávají v přesně daném pořadí, jak jsou ve zdrojovém kódu napsány. Rychlost takového algoritmu je pak daná prostým vynásobením počtu instrukcí a času provádění jedné instrukce.V případě problémů, kdy nemůžeme *optimalizovat* kód programu tak, aby jeho vykonání vyžadovalo méně instrukcí, nebo pokud je problém již navržen ve své optimální podobě, jedinnou možností jak urychlit běh, je snížení času potřebného k vykonání jedné instrukce a tedy zvýšení počtu instrukcí, které je procesor schopný vykonat za jednotku času, jinými slovy zvýšení frekvence procesoru [11, 15].

3.1.2 Paralelní algoritmus

Algoritmus paralelní je oproti tomu schopný vykonávat více instrukcí současně. Tento postup pochopitelně vyžaduje více jednotek, které jsou schopné instrukce vykonávat, tedy více jader procesoru, více procesorů nebo rovnou více počítačů. Tyto algoritmy s sebou přinášejí další problémy, se kterými se sekvenční algoritmy nesetkávají (více v sekci 3.2). Poskytují ovšem znatelné urychlení výpočtů. Zvlášť v poslední době, kdy jsou vícejádrové procesory mnohem běžnější a levnější než v minulosti. Většina výrobců procesorů se totiž vydala směrem přidávání počtu jader namísto dalšího zvyšování frekvence, jelikož to s sebou nese zvýšenou spotřebu elektrické energie, je dražší a procesory produkují více tepla, které se musí odvádět [11, 15].

3.1.3 Paralelizace

Paralelizace je proces, při kterém se sekvenční algoritmy mění na paralelní. Většinou se tak děje z důvodu zvýšení rychlosti a výkonu algoritmu. U paralelizace musíme především zajistit, aby výsledky paralelního výpočtu byly správné, a tedy stejné jako u jeho sekvenční verze [11, 19].

3.1.4 Vlákno

Vlákno je zjednodušeně řečeno část problému vykonávaná sekvenčně. Proces s jedním vláknem tedy tvoří sekvenční algoritmus, proces s více vlákny algoritmus paralelní. Vlákna ve většině implementací nejsou striktně oddělena jako procesy, ale sdílí společný paměťový prostor. Jedno vlákno je při svém běhu schopné plně zatížit jeden procesor [19].

3.1.5 Proces

Proces je samostatná instance programu běžící na nějakém počítači. Můžeme si jej představit jako množinu vláken a k nim přiřazený paměťový prostor. Narozdíl od vláken mohou procesy běžet na oddělených počítačích a ve většině případů ke komunikaci mezi sebou používají posílání zpráv (viz 3.3.2) [19].

3.1.6 Distribuovaný algoritmus

Jedná se o podmnožinu paralelního algoritmu, který je určený k běhu na různých procesorech, nějakým způsobem spojených mezi sebou. Na rozdíl od běhu na stejném procesoru, se zde mnohem více projevuje mezi-procesorová komunikační režie. Také je nutné počítat s výpadky nebo se zpožděním při posílání zpráv [4].

3.2 Problémy paralelních algoritmů

Mezi nejčastější problémy se kterými se musí autor paralelního algoritmu potýkat patří *souběh* (anglicky *race condition*), *vyhladovění* (*starvation*), *uvíznutí* (*deadlock*), *paralelní zpomalení* nebo nerovnoměrnost zatížení. [4, 11, 20]

3.2.1 Souběh

Termín souběh, anglicky *race condition*, popisuje situaci, kdy do sdílené paměti přistupuje více vláken najednou. Největší problém nastává pokud se více vláken pokouší zapsat do stejného místa v paměti. Jelikož u paralelního zpracování nikdy nemůžeme s jistotou vědět, které vlákno svou hodnotu zapíše jako první a které jako poslední, je výsledek této operace dopředu neznámý a neočekávatelný [20].

3.2.2 Vyhladovění

Vyhladovění neboli *starvation* je stav v algoritmu, kdy jsou jednomu vláknu neustále odpírány zdroje, které potřebuje k dokončení úkolu. Toto vlákno tedy svůj úkol nemůže dokončit nikdy [20].

3.2.3 Uvíznutí

Stav, kdy dvě nebo více vláken vzájemně čeká na výsledky ostatních nebo na zdroje jimi držené, se označuje jako uvíznutí, anglicky *deadlock*. Protože žádné z těchto vláken nepokračuje ve své činnosti a tedy nikdy výsledky ani zdroje nedodá, celý výpočet se zastaví a nikdy neskončí [20].

3.2.4 Paralelní zpomalení

Zpomalení plynoucí z paralelizace popisuje náklady na režii samotné paralelizace, tedy vytváření a desktrukci vláken a synchronizačních mechanismů a náklady na synchronizaci, včetně posílání zpráv mezi procesy. Někdy

se může stát, že právě tato režie je vyšší než samotný zisk plynoucí z paralelizace [4, 11, 20].

3.2.5 Rovnoměrnost zatížení

Tato vlastnost udává v jakém poměru je práce rozložena mezi jednotlivá vlákna. V ideálním případě rozložíme práci tak, aby odpovídala výpočetní síle daného vlákna. Tj. pro stejná vlákna rozložíme zátěž rovnoměrně. Toto je důležité zejména u problémů, kde další kroky výpočtu vyžadují výsledky ze všechn vláken a tak se při špatném rozložení zátěže musí dlouho čekat na nejvytíženější a tedy nejpomalejší vlákno [4, 11, 20].

3.3 Druhy paralelizace

S rozvojem paraleního zpracovávání dat se také rozvíjí teorie paralelizace, klasifikace problémů do kategorií a návrhy modelů paralelních algoritmů [4, 11].

3.3.1 Rozdělení problémů

Ne všechny problémy jsou vhodnými kandidáty na paralelizaci. Téměř u každého problému se objevují části, které se musí vykonat v daném pořadí, tedy sekvenčně. Většinou se jedná o části, které závisí na výsledcích výpočtů jiných částí algoritmu. Podle počtu těchto částí můžeme problémy rozdělit na *snadno paralizovatelné*, *hrubozrnné* a *jemnozrnné* [4, 7, 11].

Snadno paralelizovatelné problémy jsou takové, ve kterých není potřeba žádná nebo téměř žádná komunikace mezi jednotlivými vlákny. Jako příklad může sloužit zpracování obrazu, kde se obraz může rozdělit na jednotlivé části a ty se zpracují naprosto nezávisle [11].

Hrubozrnné a jemnozrnné problémy jsou takové, kde míra komunikace mezi vlákny je vysoká natolik, aby ovlivnila výkon nebo návrh algoritmu. Vzájemný poměr mezi komunikací a samotným výpočtem ve vláknech udává ke které kategorii problém přiřadíme. Problémy, kde vlákna většinu času

tráví výpočtem, označujeme za hrubozrnné a ty problémy, kde hlavní činnost vlákna tvoří synchronizace jako jemnozrnné [11].

Často se stává, že u jednoho problému můžeme zvolit míru *granularity*, což je termín označující míru rozkladu problému na menší části. V kontextu paralelizace se míra granularity přímo odráží na zařazení problému mezi jemnozrnné nebo hrubozrnné. Problémy rozložené na velký počet podproblémů mají granularitu vysokou a tedy se řadí mezi jemnozrnné problémy. Problémy rozložené na menší počet velkých částí mají granularitu nízkou a jsou tedy zařazeny k hrubozrnným [11].

U některých problémů můžeme míru granularity ovlivnit nastavením a tím ovlivnit poměr mezi synchronizací a výpočty. Pak závisí na autorovi algoritmu aby nalezl optimální míru granularity k paralelizaci, tedy optimální poměr mezi režií synchronizace a ztrátami plynoucími z nerovnoměrného rozložení zátěže [4, 11].

3.3.2 Rozdělení modelů podle druhu interakce mezi procesy

Interakce mezi jednotlivými vlákny udává mechanismy, kterými procesy nebo vlákna komunikují mezi sebou. Nejčastěji používané metody jsou *sdílená paměť* a *posílání zpráv*.

Sdílená paměť indikuje stav, kdy všechna vlákna mohou přistupovat ke všem datům v paměti. Zde je důležité používat správné konstrukce, které zabrání současnému zápisu do paměti více vlákny. Nejčastěji používaný způsob komunikace v případě vícejádrových procesorů [4, 11].

V případě posílání zpráv procesy nemají přístup do paměti jiných procesů a pro přenos dat musí používat zprávy, které si posílají mezi sebou. Často používaný model v případě distribuovaných systémů [4, 11].

Implicitní paralelizace je méně často používaný model, kdy se o paralelní zpracování postará přímo kompilátor nebo prostředí, kde program běží. Většinou je potřeba speciální prograváci jazyk se speciálními konstrukcemi, připravenými pro paralelní zpracování. Je hodno poznamenat, že kompilátory nebo prostředí na konci stejně používají posílání zpráv nebo sdílenou paměť [11].

3.3.3 Rozdělení modelů podle dekompozice problému

K provedení paralelizace musíme sekvenční problém rozdělit na části, které poté přiřadíme jednotlivým vláknům nebo procesům. Podle toho, jakým způsobem problém rozložíme, vybíráme vhodný model algoritmu [11]. Možné metody rozkladu jsou *datový paralelismus*, *funkční paralelismus* a jeho speciální případ zvaný *roura* [15].

Datový paralelismus spočívá v tom, že data se rozdělí mezi vlákna, která na nich poté vykonávají stejnou činnost. Častým příkladem je počítačová grafika, kde se na velké množství dat aplikuje stejná operace, například posunutí nebo otočení. Pokud víme, že operace trvá stejnou dobu na všech typech dat a víme, jak velká data musí být zpracována, jedná se o snadno paralizovatelou úlohu, kdy data zkrátka rovnoměrně rozdělíme mezi vlákna a necháme je počítat. Tento postup je také snadno škálovatelný, často při přidání dalších výpočetních jednotek nemusí autor dělat nic jiného, než rozdělit data v novém poměru. Ovšem datový paralelismus počítá s tím, že mezi daty je minimální nebo žádná závislost [15].

Funkční paralelismus, anglicky *task parallelism*, označuje takovou situaci, kdy vlákna vykonávají rozdílnou činnost na stejných nebo různých datech. Pokud máme dva naprosto rozdílné a na sobě nezávislé výpočty, můžeme je přiřadit dvěma rozdílným jádrům a tím snížit celkový čas potřebný k jejich vykonání. Například jedno vlákno aplikace může provádět výpočet, druhé vykreslovat uživatelské rozhraní a třetí odesílat výsledky přes Internet [15].

Případ funkční paralelizace je také roura, anglicky *pipeline*. Používá se v případech, kdy na datech musí být prováděny různé operace a kdy na sebe dané operace přesně navazují. Roura v podstatě znamená, že se výstup jedné operace použije jako vstup do operace další. Jako příklad je možné uvést zpracování videa, kdy se každý jednotlivý snímek musí například zbavit šumu, přidat kontrast a nakonec zobrazit na obrazovce. Vlákno A tedy může odstranit šum ze snímku *1* a předat ho vláknu B, které je zodpovědné za úpravu kontrastu. Během doby, kdy vlákno B pracuje, ovšem nemusí A na nic čekat a může zpracovávat snímek *2*. Stejně tak jako když vlákno B dokončí úpravu snímku *1*, předá ho vláknu C a okamžitě může začít pracovat na dalším snímku. V případě roury je nutno poznamenat, že k jejímu efektivnímu využití musí všechny činnosti trvat přibližně stejně dlouhou dobu [15].

3.4 Měření výkonu paralelních algoritmů

Výkon paralelního algoritmu je možné měřit stejnými způsoby jako výkon kteréhokoliv algoritmu. Dá se měřit délka běhu pro daná zadání, paměťová náročnost nebo maximální velikost zadání, pro které algoritmus skončí v přijatelném čase. Ovšem pro paralelní algoritmy je nejvíce důležité *zrychlení* oproti sekvenčnímu algoritmu, který řeší stejný problém [21].

3.4.1 Zrychlení

Zrychlení se dá snadno vypočítat jako poměr času běhu sekvenčního algoritmu k času běhu algoritmu paralelnímu pro stejné zadání. Zapíšeme vzorcem

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

kde S je zrychlení (z anglického *speedup*), T_s je čas běhu sekvenčního algoritmu a T_p je čas běhu jeho paralelní verze. Příklad: pokud sekvenční algoritmus běží čtyři hodiny a jeho paralelní verze pouze hodiny dvě, zrychlení se bude rovnat $S = \frac{4}{2} = 2$. Tedy paralelní algoritmus je dvakrát rychlejší než sekvenční [21].

3.4.2 Efektivita

Další velice důležitou veličnou je *efektivita* daného zrychlení. Efektivitu vypočteme jako poměr zrychlení k počtu použitých výpočetních jednotek. Zapsáno vzorcem:

$$E = \frac{S_n}{N}$$

kde E je efektivita, S_n je zrychlení při použití n výpočetních jednotek a N je počet výpočetních jednotek. Často se také vyjadřuje v procentech. Z této veličiny je vidět, že paralelní algoritmus, který pro stejný čas výpočtu potřebuje méně prostředků (procesorů a jader), je efektivnější, ačkoliv může dosahovat horšího zrychlení [21].

Paralelní algoritmus musí vykonat minimálně stejný, ale většinou větší počet instrukcí než jeho sekvenční verze. Instrukce navíc jsou způsobeny režií vláken a jejich synchronizací. Proto hodnota efektvity teoreticky nikdy

nemůže být větší než 100%. Pro efektivitu vyšší než 100% se používá termín *superlineární zrychlení* a ve většině případů ukazuje na chybu v návrhu paralelního algoritmu, nebo že došlo oproti sekvenční verzi k jeho optimalizaci – v tom případě ovšem porovnáváme dva rozdílné algoritmy. Může se stát, že takto vysoká efektivita je také důsledek změny organizace paměti, především lepšímu využívání pamet'ové cache [21].

Hodnota 100% je tedy ideální meta, které se paralelní algoritmy snaží dosáhnout. V podstatě říká, že při použití n výpočetních jednotek, se čas výpočtu sníží na $\frac{1}{n}$, tedy při použití dvou jader se čas sníží na polovinu, při použití čtyř na čtvrtinu atd. Snadno paralizovatelné problémy, které vyžadují jen minimální synchronizaci a režii okolo vytváření a desktrukce vláken, se této hranici velmi přibližují [21].

3.4.3 Škálovatelnost

Další veličina ke zvážení je *škálovatelnost*, která říká jak moc se zvyšuje zrychlení s přibývajícími výpočetními jednotkami. U většiny problémů škálovatelnost s přibývajícimi výpočetními jednotkami klesá. Zvyšuje se totiž řežie na vytváření a synchronizaci vláken. Praktické problémy mají také maximální počet výpočetních jednotek, které jsou schopné využít. Toto číslo je většinou rovné velikosti zadání problému. Například u velmi snadno paralizovatelného problému jako je násobení pole čísel konstantou. Pro zadání pole $A = [1\ 2\ 5\ 7\ 3\ 7]$ a $K = 3$ je výsledek $V = [3\ 6\ 15\ 21\ 9\ 21]$. Maximální počet vláken, které je tento problém schopný využít je 6, stejně jako velikost pole. Každé vlákno tak bude mít na starost jedno číslo. Přidáním dalších vláken se už výpočet nijak neurychlí [21].

K zobrazení škálovatelnosti a zrychlení se používají grafy, které mají na ose x počet výpočetních jednotek, na ose y zrychlení. V ideálním případě by *křivka zrychlení*, anglicky *speedup curve*, odpovídala ose prvního kvadrantu. Ve většině případů ovšem tato křivka roste pomaleji a tempo růstu se snižuje. To je způsobeno některými části algoritmu, které se paralelizovat nedají [21].

4 Paralelizace genetických algoritmů

Genetické algoritmy jsou dobrým kandidátem k paralelizaci. Větší výpočetní výkon se přímo úměrně rovná více vyzkoušeným řešením (počtu evaluací jedinců v populaci), což se rovná většímu průzkumu prostoru řešení. Pokud je algoritmus navržen dobře, to přímo vede k nalezení lepších řešení.

4.1 Obecně

Algoritmus jako takový ani nemá moc interních závislostí mezi daty. Každý jedinec je úplně nezávislý na ostatních, co se týče výpočtu fitness funkce a operátoru mutace. Operátor křížení vyžaduje jedinců více, ale zároveň většinou nevyžaduje celou populaci. Proto například při použití dvou rodičů je možné celou populaci rozdělit do dvojic a tyto dvojice zpracovávat paralelně. Výběr je často závislý na seřazení populace podle hodnoty fitness funkce a pro obecný algoritmus je tak závislý na celé populaci. Existují také výjimky, například při turnajové selekci můžeme „turnaje“ pořádat souběžně. Většina jednotlivých kroků algoritmu je tak dobrý kandidát na datovou paralelizaci [2, 11].

Jednotlivé části algoritmu jsou závislé na výsledcích přechozích kroků, ale jejich jednotlivé vykonávání nevyžaduje žádnou další synchronizaci. Například selekci můžeme uplatnit pouze na populaci, kde již máme u všech jedinců určenou hodnotu fitness funkce. Selekce samotná ale dále nevyžaduje další data ani informace o jiných částech algoritmu a běží zcela samostatně. Po skončení činnosti předá celou výslednou populaci ke křížení. Je tedy teoreticky možné použít funkční paralelizace a principu roury [2, 4, 11].

Ovšem narážíme zde na problém, že u obecných genetických algoritmů nemůžeme posoudit, která část algoritmu zabere největší procento času. U většiny praktických problémů se ukazuje, že nejvíce času algoritmus stráví počítáním fitness funkce [4]. Genetické algoritmy se obecně používají na řešení komplexních úloh, kde výpočet fitness funkce může trvat i hodiny. Oproti tomu operace křížení, mutace i selekce jsou tvořeny v podstatě triviálními operacemi, z nichž nejnáročnější je většinou řazení [4, 11].

Navíc náročnost výpočtu fitness funkce se s rostoucí složitostí problému

většinou zvyšuje daleko rychleji než nároky na genetické operace [1, 2, 4, 11]. Rychlost křížení a mutace závisí na délce chromozomu, selekce závisí pouze na hodnotách fitness funkcí a její výpočetní náročnost roste pouze pokud roste počet jedinců v populaci. Takže čím složitější problém, tím více se ukazuje rozdíl mezi časem stráveným výpočtem hodnoty fitness funkce a časem stráveným ve zbytku algoritmu. Z tohoto důvodu se použití roury i jiné funkční paralelizace zdá pro obecné genetické algoritmy jako nevhodné [2, 11].

4.2 Druhy paralelizace

4.2.1 Globální paralelizace

Globální paralelizace nijak nemění průběh algoritmu a ten se chová naprosto stejně jako jeho sekvenční verze [2, 4, 11]. Paralelizovány jsou pouze některé jeho části, mezi kterými dojde k synchronizaci. Aplikace tak počáteční populaci rozdělí na n částí, které přidělí n výpočetním jednotkám, které paralelně vypočtou fitness funkci a celou populaci opět „složí“ dohromady. Zde dojde k synchronizaci vláken, protože se musí počkat na složení celé populace, tedy počkat na nejpomalejší vlákno. Na celé populaci se provede selekce (sekvenční nebo paralelní) a znovu se vlákna synchronizují. Takto se pokračuje křížením a mutací [2, 11].

Při výběru, které operátory paralelizovat a které nechat sekvenční, je velmi důležité porovnat jak dlouho celá operace trvá. Jednak při porovnání s ostatními nebo při porovnání s režií na komunikaci, která bude při paralelním zpracování potřeba. Proto u aplikací, které používají sdílenou paměť nebo dokonce implicitní pralelizaci, může být výhodné provádět všechny části algoritmu paralelně. U jiných aplikací, zejména u distribuovaných systémů [4], kde jsou časové náklady na posílání zpráv vysoké, se může vyplatit paralelizovat pouze výpočet fitness funkce a nechat všechny ostatní části sekvenční. Záleží na typu a rozsahu problému, na velikosti populace a na počtu a velikosti genů v chromozomu [4].

4.2.2 Ostrovní model

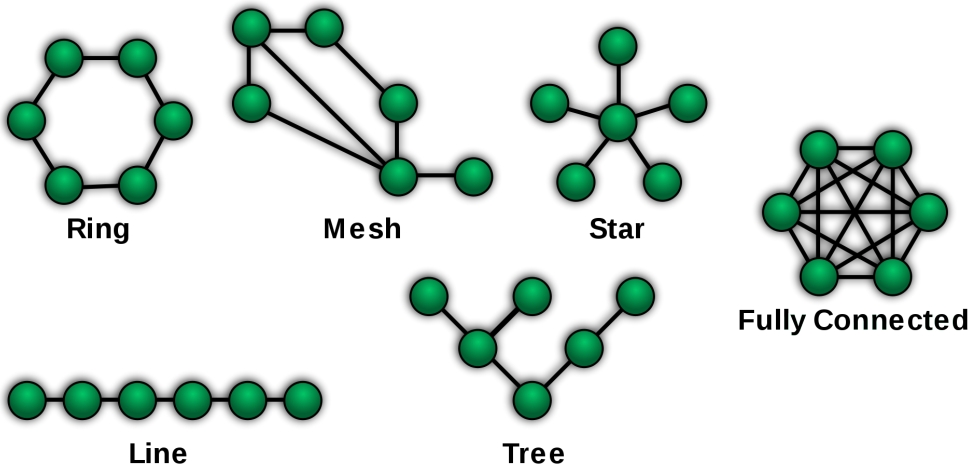
V případě sekvenčního algoritmu nebo globální paralelizace existuje pouze jedna populace se všemi jedinci a tak každý jedinec má šanci zplodit potomky se všemi ostatními. V přírodě tento model ovšem nefunguje, většinou se jedinci páří pouze ve více či méně uzavřených skupinách. Pokud tedy připustíme změnu genetického algoritmu tak, aby neexistovala jedna centrální populace, ale spíš menší počet nezávislých skupin, někdy nazývaných *kmeny*, můžeme celý algoritmus paralelizovat tímto způsobem. Každé skupině přiřadíme jednu výpočetní jednotku, která nad ní bude provádět všechny operace. Menší skupinky jedinců mají tendenci rychleji konvergovat k maximu, ale zároveň mohou poskytovat horší řešení [2, 4].

Tento způsob by ale v podstatě byl stejný jako vícenásobné spuštění stejného algoritmu. Navíc ani v přírodě nejsou jednotlivé kmeny naprosto izolované, ale dochází mezi nimi k *migraci*. Migrace označuje proces při kterém se jedinci z jednoho kmene přesouvají do jiných kmenů. V případě genetického algoritmu je důležité rozhodnout hlavně [2, 4]:

- jak často k migraci dochází
- kteří jedinci se přesunují
- zda migrující jedinci zmizí z domovské populace
- do kterých kmenů se přesunou
- které jedince v cílovém kmenu nahradí

Díky oddělení kmenů od sebe se pro tento druh paralelizace vžil název *ostrovní model*. Dle velikosti „ostrovů“, tedy množství jedinců v kmenech se dále dělí na hrubozrnný a jemnozrnný genetický algoritmus, viz rozdělení problémů podle paralelizace v sekci 3.3. Existují také extrémní případy, kdy na jednu výpočetní jednoku připadá jeden jedinec z celé populace. Většinou se jedná o speciálně navržený algoritmus, běžící na speciálním hardwaru, řešící speciální problém [2, 4].

U ostrovního modelu je rozněž důležitá *topologie*. Topologie znamená, jak jsou spolu jednotlivé prvky množiny provázány. V případě paralelních genetických algoritmů určuje spojení mezi jednotlivými kmeny, které udávají směry možné migrace. Příklady některých topologií můžeme vidět na obrázku 4.1 [2, 4].



Obrázek 4.1: Příklady topologí

Topologie mají význam při budování distribuovaných systémů, kde zpoždění při posílání zpráv není zanedbatelné, ale zároveň není zanedbatelná ani cena jednotlivých spojení mezi uzly. Proto je často důležité najít optimální topologii, která nebude mít ani zbytečně moc spojení, ale zároveň uzly budou moci komunikovat s ostatními uzly bez dlouhého čekání. Důležitý parametr tak může být délka nejdelší cesty v grafu topologie, která v případě genetického algoritmu určuje, kolik migrací musí proběhnout, aby se jedinec mohl dostat z jednoho kmene do všech ostatních. Například na obrázku 4.1 u topologie spojené jednou linkou („Line“) vidíme, že aby se jedinec dostal z jedné strany na druhou, musí proběhnout alespoň pět migrací. Naopak u úplného grafu („Fully connected“) stačí vždy migrace jen jedna. Ne vždy je dobré mít všechny cesty co nejkratší, může se pak stát, že jedinec s velmi vysokou fitness funkcí pronikne do všech kmenů příliš rychle a tak poškodí genovou diverzitu [4, 7].

Další parametr topolgie, který je třeba zvážit je odolnost sítě proti výpadkům. Například u topologie typu hvězda („Star“ na obrázku 4.1) se síť kompletně rozpadne při výpadku centrálního uzlu. Oproti tomu při výpadku uzlu z modelu „Ring“ budou ještě všechny uzly schopné komunikace se všemi ostatními.

5 Analýza aplikace

5.1 Požadavky

Napsaná aplikace na spouštění genetických algoritmů v paralelním a distibuovaném prostředí měla splňovat několik základních požadavků.

5.1.1 Programovací jazyk

Bude napsána v jazyce Java. Za prvé proto, aby se zajistila přenositelnost aplikace a její snadné nasazení na různých prostředích a operačních systémech. A za druhé proto, aby se případné rozšiřování obešlo bez překladu a linkování, ale pouze přidáním knihovního archivu s fitness funkcí. Právě Java umožňuje pomocí *reflexe*, nahlížet do již přeložených `*.jar` a `*.class` souborů a vykonávat jejich metody „z venku“.

5.1.2 Funkcionální požadavky

Aplikace bude umožňovat spouštění obecného genetického algoritmu. Tedy algoritmu, který neví nic o fitness funkci a jedince má reprezentované pouze jako řetězec, jinak řečeno pole bytů. Všechny metody křížení, mutace a selekce budou předpokládat řetězcovou reprezentaci genů a chovat se podle obecných popisů v sekci 2. Pouze implementace fitness funkce bude ponechána na uživateli. Na tuto implementaci jsou kladeny další podmínky a to, že musí být opět napsána v jazyce Java, přeložena a zabalena jako `.jar` archiv. Je vyžadována pouze jedna metoda, která jako parametr bere pole bytů a vrací číslo typu double, hlavička je tedy následující:

```
public static double fitness(byte[] array)
```

Aplikace bude umožňovat paralelní spouštění na jednom počítači stejně jako distribuované spouštení na více počítačích spojených do sítě. Z důvodu použitelnosti musí podporovat nejpoužívanější síťový protokol *IPv4*.

5.1.3 Požadavky na ovládání

Další požadavky mají usnadnit práci uživatelům aplikace. Jedná se zejména o požadavek na grafické uživatelské rozhraní, kterým půjde aplikaci snadno ovládat. Ovládání by mělo být možné centrálně z jednoho místa, aby v případě distribuovaného prostředí nebylo nutné všechny činnosti opakovat na všech počítačích zvlášť. Z důvodu časté izolace distribuovaných prostředí by také bylo vhodné ovládání aplikace umožnit z jiného místa než z místa, na kterém bude ve skutečnosti probíhat výpočet, tedy například, aby uživatel mohl zadat nový výpočet do prostředí ze svého počítače a nebyl nucen obsluhovat přímo server, na kterém výpočet poběží.

Faktor ke zvážení je také doba, kterou genetické algoritmy typicky potřebují k dokončení výpočtu. Tato toba je většinou delší a není moc vhodné, aby uživatel musel mít ovládací konzoli neustále připojenou k serveru, ale mohl jí bez omezení vypínat a zapínat a tedy kontrolovat výpočet podle svého uvážení.

5.2 Uživatel aplikace

Předpokládá se, že uživatel aplikace bude softwarový inženýr, vědecký pracovník nebo alespoň osoba znalá základů počítačových sítí a tedy, že uživateli aplikace nebude dělat problémy zadat správné adresy při spouštění programu a vyvořit jednoduchou distribuovanou síť. Dále aplikace předpokládá znalost genetických algoritmů do té míry, že uživatel bude schopen zadat základní nastavení mutací, křížení, selekce a ukončovací podmínky a tomuto nastavení porozumí.

5.3 Základní vize

Aplikace si klade za cíl využít co nejvíce výpočetního výkonu pro zrychlení výpočtu genetických algoritmů. Je určen pro co nejsnažší přechod od sekvenčního algoritmu k algoritmu paralelnímu. Nebude se tedy zabývat do hloubky speciálními změnami, které se na paralelních genetických algoritmech dají dělat a které více či méně ovliňují jejich funkčnost.

5.4 Paralelizace

Co se týče návrhu samotné paralelizace, aplikace není určená pro spouštění na speciálních distribuovaných sítích typu *extrémně jemné* paralizace. Zároveň nic nevíme o topologii dané sítě, takže algoritmus bude paralelizován jednoduchou datovou paralelizací za použití ostrovního modelu s občasnou migrací. Frekvence migrace se dá nastavit zvlášť, stejně jako doba celého výpočtu.

Je na volbě uživatele, jestli chce zvolit striktně stejnou funkčnost jako sekvenční model, tedy že jednotlivé uzly se synchronizují po každé iteraci algoritmu. Tím vytoří *globální model*, viz sekce 4.2.1, který uchovává pouze jednu populaci a deleguje na vlákna pouze jednotlivé činnosti v rámci jedné generace. Tento postup vyžaduje velké množství síťové komunikace a režie distribuce populací a sbírání výsledků. V závislosti na velikosti populace a hlavně délce trvání výpočtu fitness funkce je tento postup více či méně vhodný.

Nebo si uživatel může nastavit interval pro migraci delší než jedna generace a tedy nechat algoritmus rozvíjet *kmeny* samotné a migraci provádět pouze jednou za čas. Protože nic nevíme o topologii sítě, je velmi těžké nastavit, kam by jedinci vlastně měli migrovat. Algoritmus tedy pracuje podobně jako v případě *globálního modelu* a když nastane čas migrace, vytvoří jednu globální populaci, ze které selekcí vybere jedince k dalšímu zpracování. Tyto jedince vloží jako počáteční populaci do všech uzlů a výpočet pokračuje dokud nenastane další migrace.

V případě, že podmínka na migraci je stejná jako podmínka na konec algoritmu nebo je nastavená tak, že nastane až déle, samotná migrace neproběhne ani jednou. Uživatel si tedy může spustit více na sobě nazávislých genetických algoritmů a na konci už jen posbírat výsledky.

Nejblíže tento postup paralelizace popisuje model *farma* [11]. Spočívá v tom, že existuje centrální jednotka, která přiděluje vláknům práci. Vlákna samotná mezi sebou nekomunikují a tato centrální jednotka je zodpovědná za veškerou synchronizaci.

5.5 Síťová komunikace

Protože aplikace je primárně určená k běhu na více počítačích, je nutné navrhnout metody síťové komunikace a komunikační protokol. Z důvodu nasaditelnosti aplikace v běžných sítích musí být podporován nejpoužívanější síťový protokol IPv4. Nad protokolem IPv4 leží další protokoly transportní vrstvy, mezi kterými už máme možnost volby. Napsat si protokol *vlastní* je možnost ryze teoretická, z důvodu chybějící podpory v Javě i vě většině síťových zařízení. Další možnosti tedy jsou protokoly *Transmission Control Protocol* (TCP) a *User Datagram Protocol* (UDP).

Protokol TCP narozdíl od UDP poskytuje záruku, že zprávy dorazí ve správném pořadí a právě jednou. Oproti tomu UDP negarantuje pořadí, správnost ani počet obdržených zpráv. Ovšem má mnohem menší režii na komunikaci a je většině případů je tedy rychlejší. Pro správný chod aplikace je však nutné aby příkazy přicházely všechny a ve správném pořadí. Navíc se počítá s tím, že hlavní zátěží bude výpočet genetického algoritmu a ne synchronizace jednotlivých uzlů. Byl proto vybrán protokol TCP.

5.5.1 Komunikační protokol

V této části je popsán formát posílaných zpráv v síťové komunikaci. Určuje se jaký tvar má každá zpráva a jakou odpověď by měla druhá strana komunikace poslat zpět. Pro výběr prototolu a ním souvisejícího kódování zpráv musíme zvážit následující parametry: jaké zprávy budeme posílat, jejich délku, náročnost na zakódování a dekódování, náročnost implementace takových protokolů a stávající možnosti Javy.

Aplikace bude mezi jednotlivými uzly posílat více druhů zpráv. Jednak příkazy, kterými se aplikace ovládá. Těchto příkazů nebude moc, budou snadno zakódovatelné například jako číslo nebo krátký text. Odpovědi na tyto příkazy budou přesně definované, buď jako prosté sdělení typu „OK“ nebo jako jednoduché informace o průběhu výpočtů. Dalším typem zpráv, které musíme přenášet, jsou nastavení genetického algoritmu a výsledky. Používáme objekty v Javě a proto se nemusíme do detailu zabývat ještě kódováním a dekódováním těchto objektů, ale můžeme použít některé Javou podporované metody jako *serializace* nebo *RMI* (viz dále). Poslední věcí, kterou budeme posílat po síti jsou `.jar` soubory s fitness funkcemi. Přenášené soubory se musí přesně okopírovat na druhý konec komunikace a jedinnou

možností kódování by zde byla komprese. Zaměření této práce ovšem není na přenos souborů, proto byla tato možnost vynechána.

Serializace je proces, kterým Java z objektů vytvoří takovou reprezentaci, která se bude snadno posílat po síti. Jedinnou podmínkou na daný objekt je, aby on sám a všechny jeho atributy bylo možná zapsat jako binární řetězec. V Javě to znamená, že objekt implementuje rozhraní *Serializable* a všechny jeho atributy jsou buď jednoduché datové typy nebo objekty, které také implementují *Serializable*. Při dodržení této podmínky nám Java garantuje, že z takto vytvořené reprezentace dokáže procesem zvaným *deserializace* opět vyvořit původní objekt.

Remote Method Invocation, (RMI) je technologie Javy, umožňující vzdálené volání metod. V podstatě spočívá v tom, že aplikace *registruje* své metody na serveru a jiná klientská aplikace se poté k serveru může připojit a vyžádat si zavolání některé registrované metody. Ke svému vnitřnímu chodu RMI využívá právě serializaci. Nevýhodou RMI je složitější nastavení serveru a registrace metod.

Aplikace je určena pro co nejsnažší běh algoritmu, bez složitějšího nastavování sítí nebo parametrů pro *Java Virtual Machine*. Proto bylo rozhodnuto využít standardních síťových technologií, tedy protokolu TCP a *socketů* v Javě. *Socket* je ve zkratce abstrakce jednoho konce komunikace. RMI bylo kvůli svému složetějšímu nastavení zavrženo. Pro kódování zpráv byla, z důvodu usnadnění uživateli implementovat další nastavení bez nutnosti řešení jejich kódování a dekódování, použita serializace.

6 Implementace aplikace

V této části bude popsána implementace aplikace, která byla provedena podle analýzy v kapitole 5.

6.1 Struktura systému

Celý systém se skládá z několika částí, které každá běží jako samostatný proces na jednom počítači. Všechny procesy mohou běžet na jednom stroji, ale systém je určen k běhu v distribuovaném prostředí na více počítačích.

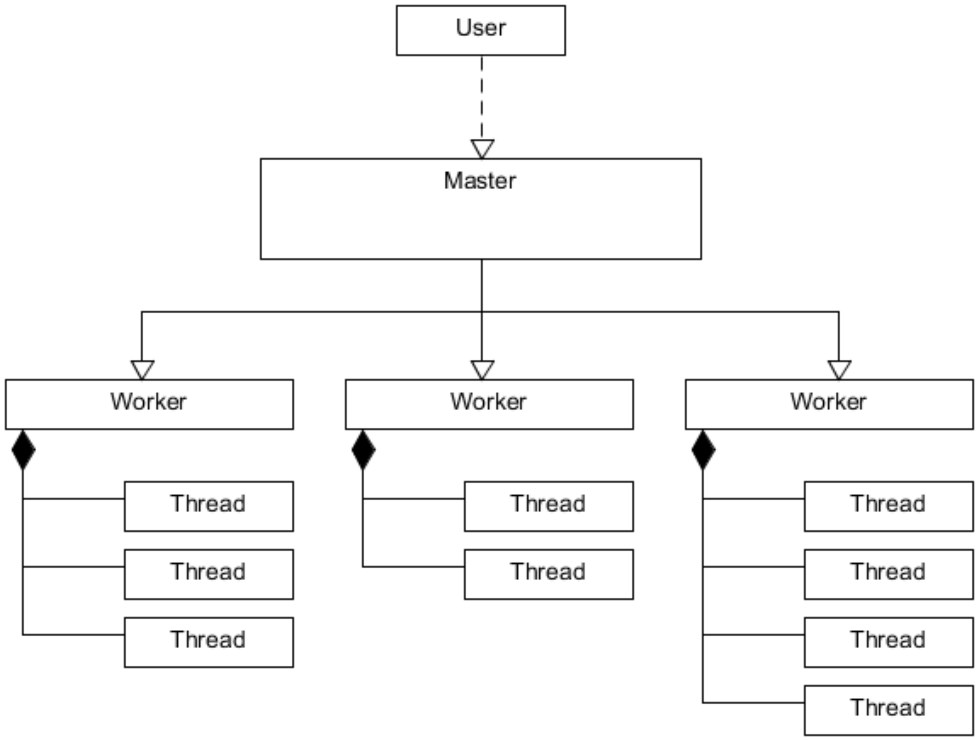
Základní rozdělení je na obrázku 6.1. Nejdůležitější části jsou *User*, *Master* a *Worker*. Směry šipek označují, která část spravuje kterou a tedy směrem po šipce proudí příkazy, směrem proti šipce výsledky. Část *Thread* označuje jednotlivá vlákna výpočtu a je proto pevnou součástí části *Worker*, což je naznačeno kompozicí.

Procesy *Worker* jsou myšleny pro spouštění na jednotlivých počítačích, kde ovládají více vláken (*Thread*), které každé odpovídá jednomu jádru procesoru (pro optimální výkon). Tato část je rozepsána v sekci 6.3.1.

Master je určen k synchronizaci jednotlivých *Workerů* a nevyžaduje nijak zvlášť velký výpočetní výkon. Je to ovšem ta část aplikace, která je nejvíce namáhána z hlediska síťové komunikace. Další informace v sekci 6.3.2.

User slouží k ovládání celé aplikace. Je naznačen přerušovanou čarou, jelikož není nutný k samotnému výpočtu a předpokládá se, že se uživatel bude připojovat přerušovaně, pouze pro nová zadání a kontrolu výsledků. Viz sekce 6.3.3.

Další části aplikace nejsou nutně rozdělené pro více počítačů nebo jader a nejsou v hrubém diagramu zakresleny. To ovšem nic neubírá na jejich důležitosti. Třídy a balíčky zaobírající se samotným genetickým algoritmem jsou obsahem sekce 6.2. Právě tyto třídy obsahují obsluhu vláken a synchronizačních mechanismů pro paralelní běh na jednom počítači. Sekce 6.3, nazvaná *Main*, obsluhuje spouštění aplikace a parsování parametrů. Velmi důležitá je sada balíků pro síťovou komunikaci, popsaná v sekci 6.8. Grafické uživatelské rozhraní je náplní spíše uživatelského manuálu v příloze A, ovšem základní

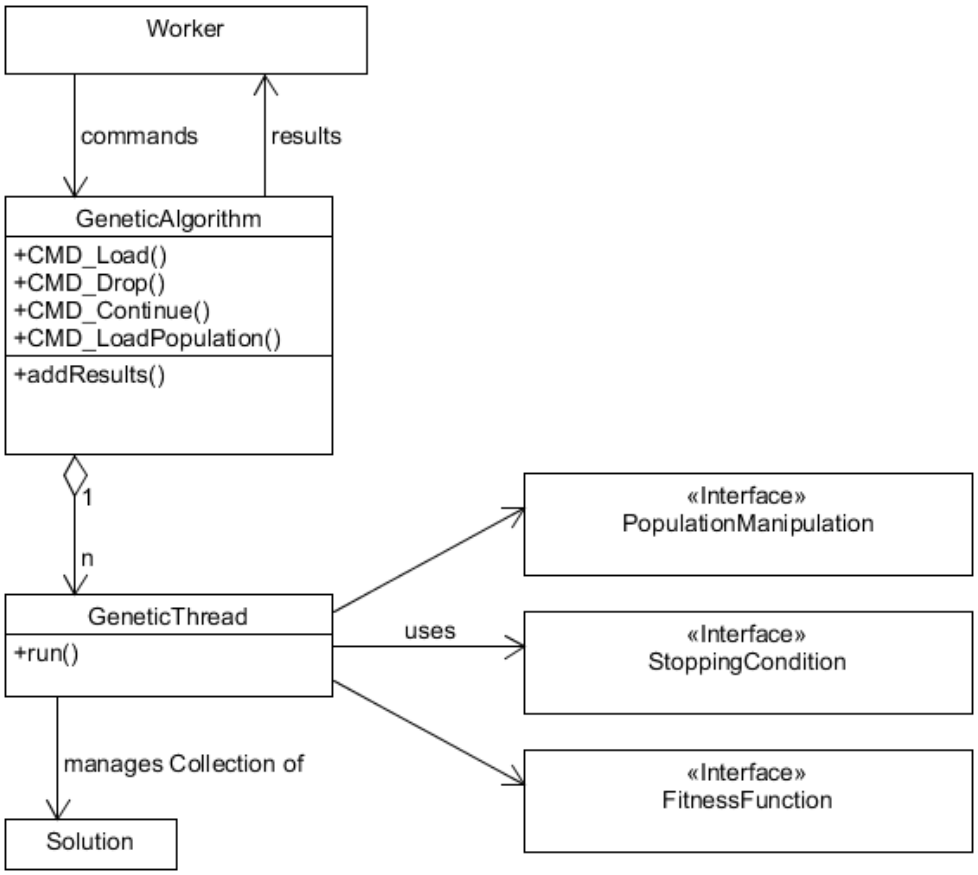


Obrázek 6.1: Základní diagram aplikace

principy jsou zmíněny i v sekci 6.10.

6.2 Genetický algoritmus

Sada balíčků `genetic` obsahuje samotný genetický algoritmus určený k paralelnímu spouštění. V zásadě je objekt třídy `GeneticAlgorithm` pouze sada jednotlivých objektů `GeneticThread`. Samotný algoritmus pouze rozdělí práci vláknům a čeká na dokončení. Poté udržuje množinu výsledků, které na vyzvání předá obsluhujícímu *Workerovi*. Také udržuje počáteční populaci, kterou přeposílá vláknům v případě, že došlo k migraci a algoritmus jí obdržel „shora“ od obsluhujícího *Mastera*. V této implmentaci si každé vlákno udržuje svojí populaci, nad kterou dělá všechny změny. UML Diagram balíčku je na obrázku 6.2.



Obrázek 6.2: UML Diagram balíčku `genetic`

6.2.1 Objekt `Settings`

`Settings` je objekt obsahující všechno potřebné nastavení k běhu paralelního genetického algoritmu. Obsahuje fitness funkci (nebo alespoň odkaz na `*.jar` soubor s funkcí), popis selekce, mutace, křížení i podmínek pro migraci a úplnou ukončovací podmínku.

6.2.2 Balík `ifaces`

`ifaces` je další balík, který obsahuje rozhraní popisující kontrakt jednotlivých částí algoritmu. Většina těchto částí je snadno pochopitelná, například třídy

implementující `Mutation` musí být schopné přijmout jako parametr populaci (kolekci řešení, `Solution`), provést mutaci a vrátit výslednou kolekci prvků stejného typu.

`Solution` je rozhraní definující jednoho jedince v algoritmu. Tento jedinec musí být schopný vytvořit svoji přesnou kopii pro účely křížení a musí mu jít přiřadit vypočtená fitness funkce. V době psaní této práce byla jedinnou implementující třídou `ByteArray`, která popisuje jedince s chromozomem zapsaným jako polem bytů.

UML diagram balíčku `genetic.ifaces` je v příloze F.

6.3 Větvení programu

Protože funkčnost aplikace se drasticky mění podle toho, jestli běží v režimu *Worker*, *Master* nebo *User*, byla jednou ze zvažovaných možností aplikaci rozdělit na tři samostatně stojící a funkční celky. Tato možnost byla nakonec zavržena, protože množství sdílených tříd by převážilo množství rozdílů. Například síťová komunikace si musí na obou koncích spojení odpovídat, nebo samotné nastavení genetického algoritmu musí být shodné jak v obslužné části *User*, tak ve vykonávající části *Worker*.

Aplikace je tedy pouze jedna, ale dá se přesně podle diagramu 6.1 spustit ve třech režimech:

- *User*, obsahující grafické uživatelské rozhraní a ovládací prvky
- *Master*, samostatně běžící serverová aplikace, která obsluhuje jednotlivé *Workery* a zároveň přijímá příkazy od části *User*
- *Worker*, výkonná část aplikace, ale zároveň samostatně stojící server, který přijmá příkazy od *Mastera*

Zvolený režim se definuje parametrem při spouštění, více v příloze A. Všechny hlavní režimy spouštění jsou součástí balíčku `main`.

6.3.1 Worker

Worker je jeden ze tří režimů, ve kterém se aplikace dá spustit (více o režimech spouštění v sekci 6.3). Zároveň je to sada tříd, které jeho funkčnost zastávají. *Worker* pouze obstarává síťovou komunikaci s *Masterem* a stará se o spouštění genetického algoritmu, v zásadě pouze zajišťuje komunikaci mezi třídami `WorkerMessageThread` a `GeneticAlgorithm`. Také si pomocí `FitnessManageru` udržuje přehled o `.jar` souborech s fitness funkcemi, které i schopen přijímat a aktualizovat, viz sekce 6.9.

6.3.2 Master

Master je další z režimů, ve kterém se aplikace dá spustit. Je to jádro veškeré synchronizace, udržuje si přehled o všech připojených *Workerech*, o stavu jejich činnosti a je schopen jim přidělovat práci. Zároveň si také udržuje svůj lokální adresář s `.jar` soubory, které je schopen přijímat od *Usera*, stejně tak jako odesílat na *Workery*. Právě *Master* je zodpovědný (při splnění podmínky migrace) za posbírání celé populace od všech výpočetních uzlů v síti, výběr nejlepších jedinců a jejich distribuce zpět na výpočetní uzly. UML diagram pro `master` je v příloze G.

Nepracuje ovšem sám od sebe, je řízen z části *User*, které posílá informace o činnosti všech *Workerů* stejně jako finální výsledky. Naopak od ní přijímá pokyny k práci, zejména nastavení genetického algoritmu, který se bude počítat.

6.3.3 User

User je uživatelské rozhraní určené k ovládání celého systému. Připojuje se přímo k části *Master*, které zadává příkazy a od které stahuje výsledky výpočtů. Je to jedinná část aplikace, která obsahuje grafické uživatelské rozhraní. Zde se graficky nastavuje nové zadání genetického aloritmu, které se posílá na *Master* k výpočtu, výpočet se spouští, zobrazuje se průběh a po skončení se stahují výsledky. Výsledky je možné uložit buď jako plainext nebo exportovat do jednoduchého XML. Zároveň je možné poslat na *Master* nový `.jar` soubor s fitness funkcí a příkazem si vynutit synchronizaci, tedy přeposlání nebo aktualizaci souboru na všech připojených *Workerech*.

Ve stejném balíčku jako `main.user` je nachází dvě rozhraní, která *User* implementuje. Jedná se o `Command` a `View`. Tyto rozhraní jsou přítomny pro případ dalšího rozšiřování aplikace, například o možnost konzolového automatizovaného spuštění, a autorovi případné implementace mají poskytnout nápovědu, jaké příkazy by *User* měl umět posílat na *Mastera* (část `Command`) a na jaké zprávy musí umět reagovat (část `View`).

6.4 Implementace jedince

Balíček `basic.solutions`, obsahuje implementaci jedinného zatím podporovaného zápisu jedince a to pomocí pole bytů. Implementující třída se jmenuje `ByteArray` a obsahuje pouze tři atributy:

```
private double fitness;
private final byte[] array;
private final int size;
```

Všechny metody jsou z důvodu univerzálního použití napsány co nejobecněji a není prováděna žádná kontrola zadávaných hodnot. Je hodno poznamenat, že metody `equals` a `hashCode` jsou překryty pouze kvůli možným dodatečně implementovaným metodám selekce, křížení nebo mutace. Všechny operátory v současné verzi aplikace (viz sekce 6.5) umožňují mít v populaci více stejných jedinců.

6.5 Genetické operátory

Sada balíčků `basic`, obsahuje implementace nejpoužívanějších křížení, mutací, selekcí a ukončovacích podmínek popsaných v kapitole 2. Všechny implementace jsou funkční pouze pro objekty třídy `ByteArray`, ale jsou napsány tak, aby nebylo těžké do nich případnou další funkčnost dodat.

6.5.1 Mutace

Mutace jsou obsahem balíčku `basic.manipulations.mutations`. Všechny dále popsané implementace se snaží dodržovat popisy v kapitole 2. Všechny implementované mutace mají dva atributy a to `solutionChance` a `geneChance`. Při zvažování, zda mutovat nebo ne se postupuje tak, že u každého řešení se generuje náhodné číslo, které je porovnáno se `solutionChance`. Když je menší, jedinec bude mutovat, když je větší, jedinec se dostane do další generace beze změny. U jedinců, kteří mutovat budou se celý proces opakuje u každého genu a pro hodnotu `geneChance`.

Postupné snižování hodnoty mutace (`Cooldown.java`) je implementováno tak, že třída si uchovává statickou hodnotu odchylky (`deviation`) a za každou mutovanou generaci tuto hodnotu sníží vynásobením číslem daným při vytváření objektu. Výchozí hodnota je nastavena na 127, což je polovina intervalu pro `byte`. Pokud je tedy `generationalDrop` nastaven na 0.5, s druhou mutací už odchylka bude 63.5, ve třetí iteraci bude opět poloviční ($63.5 \times 0.5 = 31.75$) a tak dále. Hodnota odchylky má vliv na změnu hodnoty genu, postupuje se stejně jako v případě *Gaussovské mutace* (viz dále).

Gaussovská mutace (`GaussChange.java`) ke každému mutovanému genu přičte hodnotu podle Gaussovského rozložení, v Javě implementováno jako metoda `nextGaussian()` třídy `Random`. Tato hodnota se vynásobí požadovanou odchylkou.

Hraniční hodnota (`EdgeValue.java`) má tři režimy. Buď se nahrazuje minimem, maximem nebo se mezi nimi rozhodne v poměru 50:50. Implementace je velice přímočará.

Rovnoměrné rozložení (`UniformChange.java`) zkrátka na místo mutovaného genu vloží náhodnou hodnotu generovanou funkcí `(byte)r.nextInt()`, kde `r` je objekt třídy `Random`.

6.5.2 Křížení

Implementace křížení jsou obsahem balíčku `basic.manipulations.crossovers` a stejně jako mutace se snaží dodržovat popisy v kapitole 2.

Jednobodové křížení (`OnePoint.java`) vyžaduje jedince stejné délky a pak už jenom náhodně generuje bod, kde se jedinci přeruší. Tato implementace

generuje pouze jednoho potomka z obou rodičů.

Dvoubodové křížení (`TwoPoints.java`) postupuje podobně jako jednobodové. Zvolí dva body, které porovnáním seřadí, a poté složí výsledného potomka tak, že vezme první a poslední část z prvního rodiče a a prostřední část z druhého. Může se stát, že obě generovaná čísla budou stejná a v tom případě je potomek stejný jako první rodič. Tato možnost je z důvodu urychlení křížení zanedbána a předpokládají se větší délky jedinců, takže bude nastávat zřídka.

n-bodové křížení (`NPoint.java`) vyžaduje jako parametr počet bodů řezu. Tyto body se náhodně generují, seřadí se do pole a pak se už jen jedinci prochází gen po genu a když se na bod zlomu narazí, změní se hodnota proměnné `useFirst` a začnou se kopírovat geny od druhého jedince.

„Cut and Splice“ křížení (`CutAndSplice.java`) si nejdříve vygeneruje oba body řezu, poté dopočítá správné délky jedinců a na správná místa v nových chromozomech kopíruje správné geny. Tato metoda generuje dva jedince.

Rovnoměrné křížení (`UniformChance.java`) vyžaduje jedince o stejné délce a pak už jen projde celý chromozom a u každého genu náhodně v poměru 50:50 určí rodiče.

6.5.3 Selekce

Metody výběru jsou obsahem balíčku `basic.manipulations.selections`. Implementované jsou čtyři selekce a to *ruleta*, *výběr podle pořadí*, *turnaj* a *výběr prvních n*. Implementace *výběru prvních n* (`Truncation.java`) a *turnaje* (`Tournament.java`) je přímočará a chová se tak, tak je uvedeno v popisu v kapitole 2. *Výběr podle pořadí* (`RankRoulette.java`) je v podstatě shodný s ruletou, pouze mu přechází určení nových hodnot fitness funkcí.

Ruleta (`Roulette.java`) probíhá tak, že se podle předpisu nejdříve spočítají normalizované hodnoty, které se uloží do `listNormalised`, což je `List` objektů `MyEntry`, které uchovávají jedince a k němu přiřazenou hodnotu. Dále se už jenom počítají *kumulativní normalizované hodnoty* pro jedince a jsou uloženy do mapy `mapNormalisedCumulative`. Mapa je sařezená podle klíčů, kterými jsou právě uložené kumulativní normalizované hodnoty. Pak už se jen generují náhodná čísla a z mapy se vybírají prvky, které jako první

odpovídají podmínce, že jejich hodnota je nejbližší vyšší. Proces se opakuje do té doby, než je počet vybraných jedinců dostatečný.

6.5.4 Ukončovací podmínky

Ukončovací podmínky a podmínky pro migrace jsou co se týče implementace stejné. Jsou umístěné v `basic.manipulations.conditions`. Implementovány jsou podmínky:

- `FitnessFunctionTreshhold.java`, která výpočet zastaví jakmile kterýkoliv jedinec z populace dosáhne limitní hodnoty fitness funkce
- `NumberOfGenerations.java`, která si staticky pamatuje kolikrát byla zavolána a po daném počtu generací výpočet ukončí
- `TimeLimit.java`, který pouze porovnává čas konstrukce podmínky (nebo posledního zavolání `reset()`) s aktuálním časem systému a ukončí výpočet v případě, že se hodnoty liší o více než je zadaný počet milisekund

6.6 Reflexe

Ke čtení již zabalených a kompilovaných `.jar` souborů s fitness funkcemi musíme použít funkci Javy, zvanou *reflexe*, anglicky *reflection*. Tato technologie umožňuje procházet již kompilované `.class` soubory a „z venku“ vytvářet jejich objekty a volat jejich metody. K úplnému určení metody, kterou chceme volat potřebujeme jméno `.jar` archivu, ve kterém je zabalená, jméno třídy (`.class` souboru), který jí obsahuje, a jméno samotné metody. Parametry i návratová hodnota metody jsou přesně dány specifikacemi aplikace.

Načítání a volání externí fitness funkce zařizuje třída `ExternalJar` z balíčku `basic.fitness`. Pro načítání slouží metoda `loadMethod()`, která je celá v příloze H. Nejdříve se otevře samotný `.jar` soubor jako objekt typu `JarFile` a vytvoří se objekt `ClassLoader`, pomocí kterého se jednotlivé třídy načítají. Pro každý soubor v archivu, reprezentovaný objektem `JarEntry` se nejdříve zjistí, zda se jedná o `.class` soubor. Jiné soubory nejsou produktem překladače Javy, nedají se načíst a nejsou zajímavé.

Hledáme pouze jeden `.class` soubor, který si uživatel zvolil. Pokud se tedy jméno shoduje, načteme ho pomocí dříve vytvořeného `Loaderu`. Dále už jen hledáme metodu se jménem, které opět určil uživatel. Po nalezení vhodné metody jí uložíme jako objekt `Method` k dalšímu užití. V případě, že kterákoliv část selže (nebyl nalezen `.jar`, třída nebo metoda) nenačte se nic a `method` zůstane nastavená na `null`.

Je dobré poznamenat, že vhodná metoda se nesmí načítat už při vytváření objektu `ExternalJar`, jelikož tento objekt je určen k posílání po síti a různé interpretaci na různých částech systému. Například proces *User* ani nemusí hledaný `.jar` archiv mít u sebe. Nebo na různých *Workerech* je soubor uložen v různých složkách. Je proto načítán až v případě nasazení na proces *Worker*.

Samotné vyvolávání funkce se vykoná prostým zavoláním

```
method.invoke(null, new Object[] { param });
```

kde `param` je pole bytů, získané z jedince. Pokud dojde k vyvolání výjimky při volání metody, je jako hodnota fitness funkce použita konstanta `Double.NaN`.

6.7 Testovací fitness funkce

Aplikace přichází s některými fitness funkcemi již implementovanými a určenými k testování funkčnosti. Tyto funkce většinou produkují nejlepší výsledky pro jedince reprezentované textem, takže je možné snadno zkontrolovat, jak si jedinci vedou. Jedná se o balíček `basic.fitness` a implementované testovací funkce jsou:

- `Alphabet` porovnává začátek jedince písmeno po písmenu s řetězcem „ABCDEFGHIJKLMNOPQRSTUVWXYZ“, za každé správné písmeno zvyšuje fitness funkci o jedna
- `HelloWorld` stejně jako Alphabet, ovšem porovnávaný řetězec je „Hello World !“
- `HighSum` prosté sečtení hodnot celého chromozomu, kde všechny hodnoty jsou reprezentovány jako `unsigned byte`

- `LetterHighSum` stejně jako `HighSum` sčítá hodnoty všech genů přímo, ovšem uvažuje pouze hodnoty, které se dají zapsat jako velké písmeno, tedy nejvyšší hodnoty fitness funkce dosáhne když je celý chromozom zapsán jako řetězec znaků „Z“

6.8 Síťová komunikace

Každá přenášená zpráva je objektem třídy `Message` a má dva neměnné atributy. Povinný řetezec `content`, který udává typ zprávy a volitelný `appendix`, což může být jakýkoliv serializovatelný objekt přikládaný ke zprávě. Celá implementace odpovídá návrhovému vzoru *přepravka* [16] a je umístěna v příloze I.

O samotné posílání a zpracovávání obdržených zpráv se stará abstraktní třída `ConnectionThread`. Jak název napovídá, jedná se o samostatné vlákno, které zapouzdřuje obsluhu soketu. Třída je abstraktní, jelikož vždycky musí být přiřazena k nějaké části aplikace, například *User* posílá a přijímá jiné příkazy než *Master*. Proto každá třída od ní dědící musí obsahovat metody `messageReceived()` a `connectionError()`.

`messageReceived()` znamená, že byla přijata nová zpráva, kterou je možné si vyzvednout z fronty zpráv zavoláním `getMessage()`.

`connectionError()` je zavolán pokaždé, když dojde k odpojení druhého konce komunikace, ať již standardním ukončením, timeoutem socketu, násilným přerušením pomocí RST paketu, nebo obdržením špatně formátované zprávy. Na tuto situaci reaguje každá část aplikace jiným způsobem. Například zatímco *User* se od *Mastera* může odpojit kdykoliv a bez dalších následků, při přerušení spojení mezi *Masterem* a *Workerem* musí *Master* reagovat, aby nezačal úkolovat odpojeného *Workera*.

Veškeré typy zpráv, rozdělené podle odesílatele a příjemce jsou ve třídách v balíčku `networking.messages.labels` a implementace používaných příloh ke zprávám je obsah balíčku `networking.messages.appendices`.

6.9 Posílání souborů

Aplikace je navržena pro snadné přidávání vlastních fitness funkcí v podobě `.jar` souborů s třídami a metodami, které je implementují. Archivy s fitness funkcemi je nutné mít na všech výpočetních uzlech, tedy tam, kde běží aplikace v režimu *Worker* nebo *Master*. Nahrávání archivu na každý jednotlivý uzel zvlášť je pro uživatele nepohodlné a v případě různých verzí by mohlo snadno vést k chybě. Proto je do aplikace přidána funkcionalita pro přidávání a aktualizaci souborů pouze jednou, tedy z části *User* směrem na *Master*. Poté už stačí jedním příkazem na synchronizaci říci *Masteru*, aby všechny archivy, které má, začal distribuovat na jednotlivé *Workery*.

O udržování seznamu archivů s fitness funkcemi a jeho aktualizaci se stará třída `FitnessManager` ze stejnojmenného balíčku. Ta si z důvodu jiného komunikačního protokolu a hlavně z důvodu nezávislosti na zbytku aplikace udržuje svůj vlastní server, na kterém přijímá soubory. Nezávislý na aplikaci je proto, že během výpočtu algoritmu, je možné nahrávat další soubory, stejně tak jako je možné celý server úplně vypnout (zatím pouze zásahem do kódu, tato funkcionalita není možná z ovládacího panelu). Nahrávání souborů během výpočtů je ovšem silně nedoporučováno z důvodu zpomalení výpočtu a vytížení sítě.

Komunikační protokol je zde velmi jednoduchý, pro každé navázané spojení se přečte prvních obdržených 1024 bytů, které znamenají jméno souboru v kódování UTF-8. Tento soubor se vytvoří a nakopíruje se do něj všechno co přijde komunikačním kanálem dál. Po stáhnutí a uložení celého souboru vypočítá *checksum* (viz dále) a aktualizuje se seznam ve `FitnessManageru`.

Není ovšem nutné posílat znovu soubory, které se již na druhém konci vyskytují. Proto požadavek na sychnronizaci vypadá tak, že si *Master* nejdříve vyžádá seznam všech archivů, které *Worker* má, společně s jejich kontrolním součtem (*checksum*).

6.9.1 Kontrolní součet

Kontrolní součet, jinak *checksum*, je výsledek takzvané *hashovací* funkce, která vezme na vstupu celý soubor a na výstupu generuje krátkou sekvenci znaků. Na samotnou funkci je kladen ten požadavek, aby i při sebemenší změně vstupního souboru byl výsledek odlišný. V případě souborů se stejným

jménem tak snadno poznáme, zda se soubor změnil či nikoliv. *Master* poté porovná svůj seznam `.jar` archvů se seznamem, který obdržel, a všechny soubory, které *Worker* nemá, nebo má jejich starou verzi, pošle k aktualizaci.

6.10 Uživatelské rozhraní

Vzhled a funkce dostupné přes grafické uživatelské rozhraní v režimu *User* jsou náplní uživatelského manuálu v příloze A, zde si pouze ve stručnosti popíšeme jednotlivé součásti.

Aplikace má čtyři hlavní okna, jejichž implementace je obsahem balíčku `user.gui.windows`.

- `MainWindow` je hlavní okno aplikace, kde jsou zobrazeny informace o připojeném *Masterovi* a jednotlivých k němu připojených *Workerech*, z tohoto okna je také přístupná většina ovládacích prvků
- `ConnectWindow` je malé okno sloužící pouze k zadání IP adresy a portu na který se *User* má připojit
- `ResultsWindow` je okno určené k prohlížení a ukládání výsledků
- `SetupWindow` je okno obsahující veškeré nastavení pro nové zadání genetického algoritmu

Hlavní okno se skládá z jednotlivých *panelů*. V balíčku `user.gui.panels` najdeme implementace panelů pro informace o *Masterovi* i o jednotlivých *Workerech*. Zároveň obsahuje třídu `PanelsConstants`, ve které jsou popsány rozměry všech panelů.

Okno nastavení obsahuje několik panelů pro nastavení jednotlivých možností algoritmu, tedy mutace, selekce, křížení, podmínky migrace a ukončovací podmínky. Vzhled a stavba těchto panelů je dána třídami v balíčku `user.gui.panels.settings`.

Každé jednotlivé nastavení může, ale nemusí mít svá další parametry. Například u *one-point crossover* není co dalšího nastavit, ale u *n-point* se jako parametr musí objevit počet bodů řezu. Tyto přídavná nastavení jsou definována v balíčcích `user.gui.panels.options.*`, kde je pro každou možnost vytvořen speciální balík se třídami.

7 Testování

7.1 Testovací prostředí

7.1.1 Počítač A

Hardware

- CPU: Intel Core 2 Duo E7300, taktovaný na frekvenci 3.41 GHz
- RAM: 4 GB DDR2
- Disk: Crucial 128 GB, Solid State disk

Software

- Operační systém: Windows 8.1
- Java, verze 1.8.0_25

7.1.2 Počítač B

Hardware

- CPU: Intel Core i5 2410M, taktovaný na frekvenci 2.3 GHz
- RAM: 4 GB DDR3
- Disk: 750 GB, klasický magnetický

Software

- Operační systém: Kali Linux
- Java, verze 1.8.0_20

7.2 Základní test funkcionality

Tento základní test je určen k ověření funkcionality všech částí genetického alogritmu, tedy všech možností křížení, mutace, selekce, migrační i ukončovací podmínky. Nejsou testovány všechny kombinace, ale každá možnost nastavení je otestována alespoň jednou.

7.2.1 Testovaná aplikace

Aplikace byla pro účely testování upravena. Zejména logování bylo opraveno tak, aby *Worker* logoval i zprávy typu *FINER*, tedy vypisoval v každém kroku alogritmu celou populaci. Upravená aplikace pro účely logování je součástí práce jako elektronická příloha. Hlavní třída aplikace byla upravena tak, aby spouštěla všechny tři hlavní části najednou a pro snaží čtení logů má část *Worker* pouze jedno vlákno. Celý kód metody `main` je následující:

```
public static void main(String[] args) throws IOException,
    InterruptedException {

   Worker w = new Worker(5000, "Worker", "<folder 1>", 1);
   w.start();
   Thread.sleep(500);

   Collection<Node> targets = new ArrayList<Node>();
   targets.add(new Node("127.0.0.1", 5000));
   Master m = new Master(targets, 5900, "Master", "<folder 2>");
   m.connectToWorkers();
   m.start();
   Thread.sleep(500);

   User u = new User();
   u.start();
   Thread.sleep(500);
   u.cmdConnect("127.0.0.1", 5900);
}
```

V této části netestujeme externí `.jar` soubory, proto složky `<folder 1>` a `<folder 2>` mohou zůstat prázdné.

7.2.2 Křížení

One-point

Barevně zvýrazněný log je v příloze C. Je z něj jasně vidět, že jedinci vznikají ze dvou rodičů pro zvolený bod řezu.

Two-point

Barevně zvýrazněný log je v příloze D. Stejně jako v případě one-point je jasně vidět, že jedinci vznikají z rodičů výměnou prostřední části chromozomů.

N-point

Celý log pro jednu generaci je v souboru *8.txt*. Aby se ukázal vliv počtu bodů řezu na křížení, byly zvoleny delší reprezentace jedince, nevhodné pro zápis do tohoto výčtu.

Cut and Splice

Celý log pro jednu generaci je v souboru *3.txt*. Je zobrazen kompletní log celého průběhu algoritmu, kde byla zvolena testovací fitness funkce *HighSum*, která přirozeně bude preferovat delší jedince. V logu si tak můžeme všimnout postupného nárůstu délky chromozomu.

Uniform chance

Celý log pro jednu generaci je v souboru *9.txt*. Stejně jako u N-point je zvolena větší délka chromozomu, aby se více ukázal vliv křížení.

7.2.3 Výběr

Výběr se testuje velmi těžko pouze za použití výčtu populace před a po selekci. Jedinnou výjimku tvoří truncation selection, u které můžeme snadno z logu *1.txt* poznat, že opravdu vybírá nejlepších n jedinců. U jiných druhů selekce je velký podíl náhody a je tak těžké určit, které výsledky jsou správné. Otestovaná metoda *rank-roulette* v logu *6.txt* opravdu vybírá více jedinců z předních pozic než jedinců z konce fronty (ve výsledné populaci je jejich poměr vyšší). Oproti tomu ruleta pro populaci, kde všichni jedinci mají fitness 0 (což je pro algoritmus špatná, ale přesto validní hodnota, ovšem takový jedinec by normálně neměl šanci být vybrán *vůbec*) vybírá neustále posledního jedince z populace a tím napsrosto zničí genovou diverzitu. Je to uvedeno jako příklad špatného použití rulety v logu *7.txt*.

Algoritmy selekce byly testovány a jejich funkčnost ověřena v průběhu jejich implementace pomocí white-box testů.

7.2.4 Mutace

Edge value

Hodnoty pro tento test byly nastaveny:

- šance pro vybrání jedince k mutaci: 50%
- šance pro mutaci jednotlivých genů: 80%
- hodnota mutace: náhodná v poměru 50:50.

Jak je vidět z logu *10.txt*, opravdu pouze přibližná polovina populace zmutovala, u mutovaných jedinců se poměr mutovaných genů k původním blíží 80% a hodnota mutace je rozvnoměrně rozložena mezi horní a dolní hranici.

Uniform chance

Hodnoty pro tento test byly nastaveny:

- šance pro vybrání jedince k mutaci: 20%
- šance pro mutaci jednotlivých genů: 50%

Hodnoty si můžeme ověřit z logu *11.txt*.

Gauss chance

Zde byly hodnoty nastaveny tak, aby všechny geny byly nuceny mutovat (100% pro obě šance) a odchylka rovná dvěma. Jak je z logu *4.txt* vidět, většina genů mění svou hodnotu jen velmi málo nebo vůbec, což přibližně odpovídá Gaussově rozdělení.

Cooldown

Log *5.txt* ukazuje postupný vývoj mutace pro prvních pět generací a pak pro generaci patnáctou. Mutace byla nastavena tak, aby opět mutovaly všechny geny ve všech jedincích a mezigenerační zpomalení bylo dáno na 50%. Můžeme si všimnout, že první generace mutuje v podstatě úplně náhodně, zatímco hodnoty páté generace se mění už mnohem méně. U patnátcé generace je již mutace potlačena úplně.

7.2.5 Migrační podmínka

Funkčnost byla ověřena i u migračních podmínek (*exchange condition*). Pomocí logů a výpisů na části *Master* můžeme ověřit, že když všechna vlákna splní tuto podmínku, dojde k migraci a sychronizaci s globální populací.

Podmínky migrace se také dají snadno ověřit sledováním stavu algoritmu přímo v uživatelském rozhraní. Informace o *Masteru* zahrnují také počet migací, které již proběhly. Proto se například pro nastavení podmínky časem dá snadno sledovat zda se čítač zvýší každých x sekund.

Nastavení migrační podmínk na typ *Fitness function Threshhold* je ve většině případů logická chyba. Jednak se čeká, až *všechna* vlákna splní tuto podmínku a za druhé typická fitness funkce se málokdy snižuje, takže od první migrace dál bude většinou docházet k dalším migracím kažou generaci.

7.2.6 Ukončovací podmínka

Všechny předchozí testy byly spouštěny po dobu deseti sekund, po které se algoritmus opravdu ukončil. Dalšími vyzkoušenými možnostmi byl počet generací nastavený na jednu popřípadě na pět. Všechny případy skončily úspěšně, algoritmus se ukončil v očekávanou dobu. Nutno poznamenat, že algoritmus vždy musí dokončit evaluaci celé populace. Například v případě, že výpočet fitness funkce trvá deset sekund pro jednoho jedince a v populaci máme šedesát jedinců na vlákno, poběží výpočet minimálně deset minut ($10s \times 60 = 600s = 10m$), než dojde k prvnímu vyhodnocení podmínky. Takže i kdyby byla nastavená na 5 sekund, alogritmus poběží deset minut.

Další věcí hodnou zmínky je pořadí v jakém se podmínky vyhodnocují. Nejdříve se čeká, až všechna vlákna splní svojí podmínku na migraci, a až poté se vyhodnotí celková ukončovací podmínka. Takže algoritmus běží vždy minimálně tak dlouho, než se naplní první migrační podmínka.

7.2.7 Další testy

Kromě výše zmíněných testů byly samozřejmě provedeny i testy celkového chování algoritmu na problémech charakterizovaných funkcemi ze sekce 6.7. Zde se testoval výkon algoritmu jako takového, tedy při jakých nastavení konverguje rychleji, kdy pomaleji a kdy vůbec. Tato práce si neklade za cíl stanovit nejlepší nastavení pro řešení těchto problémů, proto pouze zmíníme, že genetické algoritmy se zde dají s úspěchem použít a toto použití bylo testováním ověřeno.

7.2.8 Zhodnocení výsledků

Všechny testy dopadly úspěšně, aplikace nikdy nespadla ani neprodukovala špatné výsledky. Je zde ověřeno, že všechny metody fungují tak, jak je popsáno v kapitole 2. Je zde ovšem nutné poznamenat několik věcí, zmíněných dále.

V příkladech křížení se občas objeví takový potomek, který je naprosto shodný s jedním z rodičů. To je způsobeno tím, že rodiče jsou vybíráni náhodně z celé populace a může se stát, že je dvakrát vybrán ten samý jedinec.

Žádný způsob křížení, kromě *Cut and Splice*, ze dvou stejných rodičů nevyprodukuje potomka odlišného. Toto chování může být nežádoucí, ale u obecného genetického algoritmu se počítá s větší populací, než mají modelové příklady, a tedy, že se šance na výběr stejných rodičů sníží. Toto chování bylo také otestováno.

7.3 Posílání souborů

Aplikace také obsahuje funkci pro posílání `.jar` souborů s fitness funkcemi. Základní funkcionalita je taková, aby uživatel obsluhující část *User* na svém počítači vybral `.jar` soubor, který se nahraje na *Mastera*. Dalším krokem je požadavek na synchronizaci souborů, který přinutí *Master* k distribuci nových souborů na jednotlivé *Workery*. *User* vždycky při posílání souborů přepíše dosavadní soubory, distribuce směrem od *Mastera* na *Workery* probíhá pouze v případě, že soubor na *Workerovi* chybí nebo se liší jeho *checksum*, což značí jeho aktualizaci.

7.3.1 Testovaná aplikace

Aplikace nebyla pro tento test nijak upravena a každá část byla spuštěna standardním způsobem jako samostatný proces.

7.3.2 Síť

Oba počítače byly propojeny jedním routerem Tenda W311R+. Rychlost sítě na tomto prvku i na obou síťových kartách byla 100Mb/s.

7.3.3 Lokální posílání

První test běžel pouze na počítači A s využitím *loopback* internetového rozhraní. Všechny adresy byly nastavené na „127.0.0.1“. Byly spuštěny všechny části aplikace, *Worker* třikrát. Dále byl přes ovládací GUI odeslán soubor a požadavek na synchronizaci. Zda byl pokus úspěšný bylo ověřeno spuštěním algoritmu, který si fitness funkci bral z externího, poslaného `.jar` souboru.

Poté byl celý postup opakován pro jiný `.jar` soubor, ale se stejným jménem, což mělo simulovat aktualizaci archivu. Použité byly archivy `dot_delay.jar` a `dot_systime.jar`. Druhý jmenovaný byl pro účely testu přejmenován.

Test byl úspěšný, soubor byl zkopírován do složky *Mastera* a po požadavku na synchronizaci i do všech tří složek jednotlivých *Workerů*. Samotný algoritmus proběhnul bez problémů, vrácených řešení byl správný počet a hodnoty fitness funkce se rovnaly hodnotám zadaným v `.jar` souboru.

Druhý test byl stejně úspěšný jako ten první. Všechny soubory byly aktualizovány a vrácená řešení měla jinou fitness funkci, shodnou se zadáním v `dot_systime.jar`.

7.3.4 Síťové posílání

Síťový test využíval obou počítačů. Na počítači A běžel jeden *Worker* a *User*, počítač B měl spuštěné další dva *Workery* a *Master*. Jinak byl test stejný, jako první část *lokálního posílání*. Aktualizace se už dál nezkoušela.

Test byl úspěšný, soubory se zkopírovaly přesně podle očekávání i mezi různými operačními systémy.

8 Měření výkonu

V této kapitole srovnáme výkon implementované aplikace spuštěné paralelně se srovnatelným sekvenčním algoritmem. Budeme sledovat jak se výkon mění v závislosti na počtu vláken a z naměřených hodnot vypočteme zrychlení a efektivitu. Také provedeme měření v distribuovaném prostředí na dvou počítačích a porovnáme ho se sekvenčním i paralelním zpracováním.

Výkon genetického algoritmu se dá měřit pomocí maximální hodnoty fitness funkce, ovšem toto je značně neobjektivní hledisko z vysokým prvkem náhody. Proto budeme výkon měřit jako počet vypočtených fitness funkcí. To přímo odpovídá velikosti prohledaného prostoru řešení a poskytne to měřítko výkonu nezatížené náhodou.

8.1 Testovací prostředí

8.1.1 Počítače

Počítače na kterých budeme spouštět aplikace a měřit hodnoty jsou shodné jako v kapitole 7, jejich bližší specifikace nalezneme v sekci 7.1.

8.1.2 Sekvenční algoritmus pro porovnání

Pro objektivní srovnání potřebujeme algoritmus, který běží sekvenčně a vykonává přesně stejnou činnost jako testovaný systém. K tomuto účelu byla napsána třída `TestingMain`, jejíž celý kód je v příloze J. V zásadě tato třída pouze provede nezbytně nutné nastavení k chodu jednoho vlákna. Zejména se musí nastavit `GeneticAlgorithm` a `ExternalJar` tak, aby odkazovaly vlákno na správný `.jar` soubor s fitness funkcí. Je důležité si všimnout, že nad objektem `Worker` ani `GeneticAlgorithm` nejsou volány žádné metody, které by spustily další vlákna.

Nastavení samotého genetického algoritmu je probráno v sekci 8.1.3. Tak jak ho vidíme v metodě `main` je rovnocenné nastavení, které používáme pro samotnou aplikaci.

Je dobré poznamenat, že takto vytvořená testovací funkce je vhodná pouze pro účely testování. Na konci každého výpočtu totiž spadne na výjimku, jelikož celá aplikace není řádně incializována.

8.1.3 Nastavení algoritmu

Testovací algoritmus byl nastaven na tyto hodnoty:

- Mutace: *Hraniční hodnota*, obě šance na mutaci na úrovni 50% a je použito nahrazení kladnou hraniční hodnotou
- Selekce: *Výběr prvních n (Truncation selection)*
- Křížení: *Jednobodové*
- Počet jedinců v populaci: *2*
- Počet jedinců v populaci po selekci: *2*
- Počet bytů (délka) jedince: *30*
- Ukončovací podmínka: Po uplynutí 60 sekund
- Migrační podmínka: Po uplynutí 60 sekund – stejná jako ukončovací, takže k migraci nedojde nikdy
- Fitness funkce: *Externí* `.jar` *archiv*, viz sekce 8.1.5

8.1.4 Testovaný systém

Samotná aplikace není nijak upravena, běží všechny její tři součásti a při jednotlivých testech měníme pouze počet vláken v části *Worker*. Samozřejmě, že pro všechny běhy je nastavení stejné, specifikované v sekci 8.1.3.

Pro co nejobjektivnější měření je při spuštění programu přidán parametr JVM, který zabrání kompilačním a *in time* optimalizacím, které Java provádí. Tento přidaný parametr je:

```
-Djava.compiler=NONE
```

8.1.5 Testovaná fitness funkce

Testovaná fitness funkce má simulovat složité výpočty a měla by tak vytížit procesor za celou dobu jejího výpočtu. Pro objektivní měření by měla běžet pokaždé stejnou dobu. Zároveň chceme sledovat, kolikrát byla spuštěna, proto při každém spuštění vypíše do konzole tečku „.“. Celý kód funkce:

```
static volatile int a, b, c, d;

public static double compute(byte[] array) {
   System.out.print(".");

   for (a=0; a<40; a++)
      for (b=a; b < a*a*a; b++)
         for (c=0; c < 10; c++)
            d = a*b*c;

   return (double)(10.0);
}
```

Funkce vrací pokaždé hodnotu 10.0, prostřední část vyplněná vnořenými cykly má za cíl pouze zatížit procesor.

8.2 Popis testu

Nejdříve otestujeme výkon sekvenčního algoritmu. Několikrát ho spustíme s tím, že přesměrujeme výstup aplikace do souboru. Vždy po skončení výpočtu zapíšeme počet teček, které se objevily v souboru. Opakujeme pro druhý počítač.

Paralelní aplikaci testujeme stejně. Spustíme *Workera* se dvěma vlákny, *Master* a *User*. Nastavíme algoritmus podle sekce 8.1.5 a přesměrujeme výstup z *Workera* do souboru. Postupně opakujeme pro dvě až osm vláken a zapisujeme všechny hodnoty.

Poté spustíme algoritmus distribuovaně na obou počítačích zároveň a opět sledujeme jak se mění výkon. Zde je rozdíl v tom, že necháme Javu provést optimalizace, tedy nepřidáváme parametr `-Djava.compiler=NONE` ke spuštění.

Na každém počítači spustíme jeden *Worker*, na počítači A potom *Master* a *User*. Nastavení necháme stejné, ale použijeme optimální počet vláken pro každý počítač, tedy počet, který odpovídá počtu jader procesoru.

8.3 Výsledky

Naměřené hodnoty pro paralelní spuštění pro jednotlivý počet vláken jsou uvedeny v tabulkách E.1 a E.2 v příloze E. Pro distribuované spuštění byly naměřeny hodnoty uvedené v tabulce 8.3.

Tabulka 8.1: Hodnoty pro distribuovaný systém

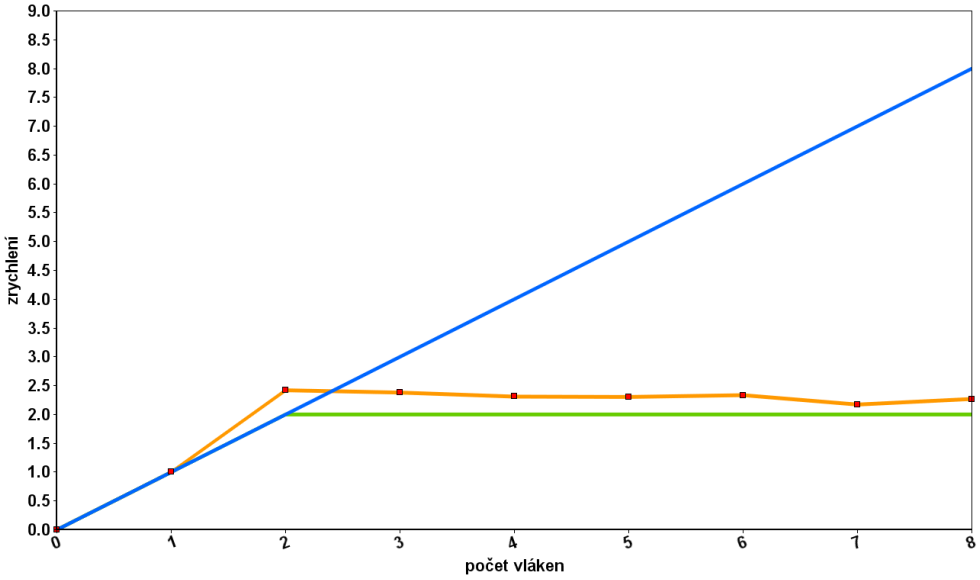
Počítač	Měření				
	1	2	3	4	5
A	614	610	568	656	550
B	739	893	893	897	797
Celkem	1353	1503	1461	1553	1347

8.4 Zhodnocení

Měření zde provedené obsahuje zhodnocení aplikace z hlediska dosažení vytyčených cílů. Cílem práce bylo napsat aplikaci, která zvýší výkon genetického algoritmu jeho paralelizací a možností spouštění v distribuované síti počítačů. Podle výsledků testů se zdá, že cíl byl splněn. Aplikace vykazuje nárůst počtu prověřených fitness funkcí za stejnou jednotku času u všech paralelních běhů v porovnání s během sekvenčním.

Dosažená zrychlení a jejich efektivitu vidíme na grafech 8.1 a 8.2. Modrá linka vždy ukazuje nejlepší možné zrychlení, dané ideální paralelizací. Zelená čára udává nejlepší možné zrychlení pro daný počítač, které je dané počtem jader jeho procesoru. Oranžová čára znázorňuje skutečně naměřené hodnoty pro zrychlení.

Z obou grafů je vidět, že bylo dosaženo *superlineární* zrychlení. U počítače A se oranžová čára skutečně naměřených hodnot velice blíží nejlepšímu možnému zrychlení pro daný počítač. Také si můžeme všimnout, že od dvou

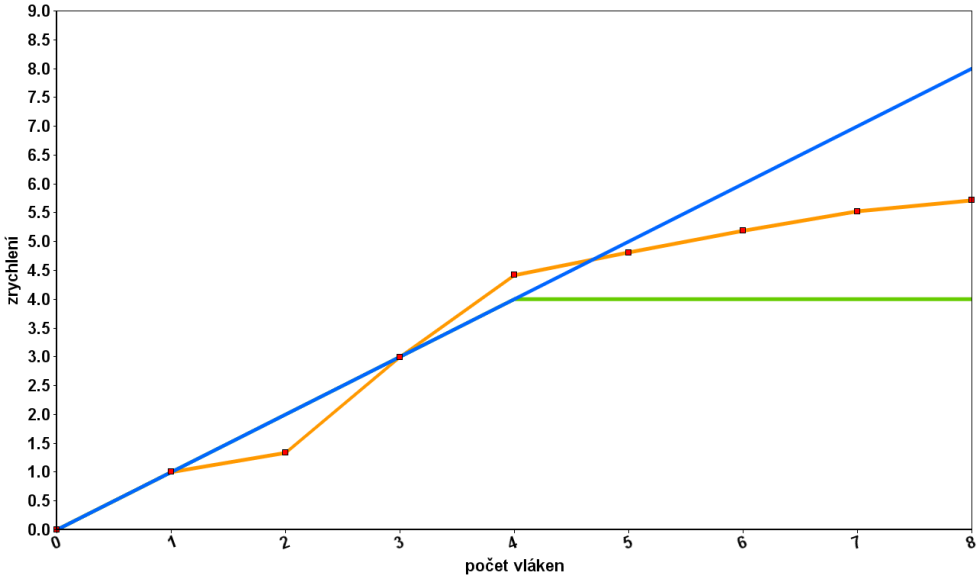


Obrázek 8.1: Graf zrychlení pro počítač A

procesorů dál se zrychlení už nezvyšuje. Naopak u počítače B, který by se čtyřmi jádry neměl mít hodnotu zrychlení nikdy vyšší než 4 si můžeme v tabulce přečíst i hodnotu 5.7 pro osm vláken. S rostoucím počtem vláken se zvyšuje i zrychlení, což by ukazovalo na nevyužité výpočetní kapacity. Ovšem i z grafu je vidět, že u čtyř vláken dojde ke zlomu a ačkoliv zrychlení pořád roste, tempo růstu se dost zpomalilo.

V obou případech se naměřené hodnoty pohybují nad teoretickou maximální hranicí pro dané počítače. Dosažené *superlineární* zrychlení se dá vysvětlit několika způsoby. Mohlo dojít k optimalizaci kódu při spuštění paralelní verze. Test byl navržen tak, aby spouštěné algoritmy byly co nejpodobnější, takže chtěnou optimalizaci můžeme vyloučit. Mohlo se ovšem stát, že kompilátor nebo virtuální stroj Javy přeložil třídy jinak při každém běhu a i přes parametr `-Djava.compiler=NONE` k optimalizacím došlo.

Další možné a časté vysvětlení pro *superlineární* zrychlení spočívá v lepším využití paměti, zejména procesorové cache. Toto vysvětlení je pravděpodobnější, protože zrychlení je větší na počítači B, který má modernější



Obrázek 8.2: Graf zrychlení pro počítač B

procesor a k dispozici více cache.

Také můžeme hledat důvod u operačního systému, zejména *task scheduler* může výkon vícevláknové aplikace drasticky ovlivnit. Oba počítače se také liší minor verzí Javy, ale zde pravděpodobně nebude důvod optimalizace a zrychlení. Poslední možností jak mohlo k takovým výsledkům dojít je chyba v implementaci aplikace nebo testovací třídě.

Pro distribuovaný systém jsou výsledky dobré. Oproti spouštění na každém počítači zvlášť došlo k urychlení, ale to není důsledek zapojení do sítě, ale pouze důsledek vynechání parametru pro zákaz optimalizací Javy. Je ale vidět, že oba počítače pracují a výkon je vyšší než na jednom počítači.

Aplikace zřejmě plní svůj učel a ke zrychlení oproti sekvenčnímu algoritmu skutečně dochází. Dosažené *superlineární* zrychlení není tak extrémně velké, aby se jednalo o očividnou chybu aplikace, ale přesto se projevuje a mohl by to být námět na další výzkum a více druhů testů a měření.

9 Závěr

V rámci této práce byly představeny genetické algoritmy a jejich využití v informatice. Také byly popsány nejpoužívanější metody pro křížení, mutace, výběr a ukončovací podmínky. Čtenář se dozvěděl o procesu obecné paralelizace algoritmů, jejích problémů a aplikace na genetické algoritmy.

Součástí práce je také aplikace, určená právě k paralelizaci a distribuovanému provozu genetických algoritmů. Cílem aplikace bylo zvýšit výkon algoritmu použitím většího počtu výpočetních jednotek. Tento cíl byl splněn jak je dokázáno testováním aplikace na některých specifických problémech a měřením výkonu.

Aplikace splňuje zadání, ovšem to nic nebrání jejímu dalšímu vývoji. V závisostli na možnostech dalšího užívání by se dala více rozvinout *migrace*, například přidat podporu pro další selekci, která by narozdíl od hlavní selekce určovala jedince k migraci. Dále by nemusela existovat jedna centrální populace, se kterou jedinci vždy začínají iteraci, ale ostrovy by si svou populaci mohly zanechat a pouze přijmout migrující jedince k sobě.

Užitečná by nejspíše mohla být i možnost přidělit každému ostrovu jinou fitness funkci, selekci, křížení nebo mutaci. Tento postup by se hodil spíše na testování samotného genetického algoritmu, než na hledání řešení problémů, ale mohl by být dobrým odrazným bodem při zvažování nastavení parametrů algoritmu.

Program jako takový také neumožňuje probíhající algoritmus „násilně“ zastavit (kromě ukončení všech *Workerků* přímo), což by mohla být dobrá schopnost v případech, kdy chceme sledovat vývoj fitness funkce ručně a algoritmus ukončit podle potřeby.

Literatura

[1] GOLDBERG, David E. *Genetic algorithms in Search, Optimization & Machine Learning*. Addison-Wesley Publishing Company, 2012
ISBN 0-201-15767-5

[2] HYNEK, Josef. *Genetické algoritmy a genetické programování*. Grada Publishing a.s., 2008
ISBN 8-024-72695-5

[3] TEDA, Jaroslav. *Genetické algoritmy a jejich aplikace v praxi*. [cit. 5.5.2015], dostupné na
`http://programujte.com/clanek/2005072601-geneticke-algoritmy-a-jejich-aplikace-v-praxi/`

[4] CANTÚ-PAZ, Erick. *A Summary of Research on Parallel Genetic Algorithms*. The Illinois Genetic Algorithms Laboratory (IlliGAL), 1995

[5] GWIAZDA, Tomasz D. *Genetic Algorithms Reference Vol.1 Crossover for single-objective numerical optimization problems*. Lomianki, 2006. ISBN 83-923958-3-2.

[6] OBITKO, Marek. *Introduction to Genetic Algorithms*. [cit. 5.5.2015], dostupné z
`http://www.obitko.com/tutorials/genetic-algorithms/index.php`

[7] WHITLEY, Darrell. *A genetic algorithm tutorial*. Statistics and computing 4.2, 1994, str. 65-85.

[8] SAFE, Martín. CARBALLIDO, Jessica. PONZONI, Ignacio. BRIGNOLE, Nélida. *On Stopping Criteria for Genetic Algorithms*. Advances in Artificial Intelligence, SBIA 2004

[9] *Učící se algoritmy, Genetické algoritmy*. [cit. 5.5.2015], dostupné z
`https://akela.mendelu.cz/ xpopelka/cs/ui/ucici/`

[10] POŠÍK, Petr. *Genetické algoritmy.* České vysoké učení technické Praha [cit. 5.5.2015], dostupné z
`http://labe.felk.cvut.cz/ posik/pga/theory/ga-theory.htm`

[11] DARLINGTON, J., GHANEM, M., TO, H. W., *Structured Parallel Programming.* Department of Computing Imperial College London, SW7

[12] GUILLAUMIER, Kristian. *Generic Chromosome Representation and Evaluation for Genetic Algorithms.* Department of Computer Science and AI, University of Malta

[13] SYED, Omar.51 *Applying genetic algorithms to recurrent neural networks for learning network parameters and architecture.* Master Thesis, Case Western Reserve University, 1995

[14] POTUŽÁK, Tomáš. *Distributed/parallel genetic algorithm for road traffic network division for istributed traffic simulation.* In: Intelligent Distributed Computing VII. Cham: Springer, 2013. Studies in Computational Intelligence, 2013, 511, s. 151-156. ISSN 1860-949X. ISBN 978-3-319-01570-5.

[15] BAL, Henri E., HAINES, Matthew *Approaches for Integrating Task and Data Parallelism.* 1998, IEEE Concurrency, vol 6, pp 74–84

[16] BOUDA, Radek. *Seriál návrhových vzorů – 1. díl.* [cit. 5.5.2015], dostupné z
`http://programujte.com/clanek/2012032900-serial-navrhovych-vzoru-1-dil/`

[17] BEASLEY, David, MARTIN, R. R., BULL, D. R. *An overview of genetic algorithms: Part 1. Fundamentals.* University computing 15 (1993)

[18] BESSAOU, M., SIARRY, P. *A genetic algorithm with real-value coding to optimize multimodal continuous functions*

[19] BENEŠ, Miroslav *Programování s vlákny* Katedra informatiky FEI VŠB-TU Ostrava [cit. 5.5.2015], dostupné z
`http://www.cs.vsb.cz/benes/vyuka/pte/texty/vlakna/index.html`

[20] SHARMA, Manoj. *Common problems of concurrency (Multi-Threading) in Java* March 2, 2014, [cit. 5.5.2015], dostupné z
`http://www.somanyword.com/2014/03/common-problems-of-concurrency-multi-threading-in-java/`

[21] KADAM, S.S. *Performance metrics for parallel systems* C-DAC, Pune [cit. 5.5.2015], dostupné z
`http://www.iacs.res.in/Performance_Metrics.pdf`

Seznam elektronických příloh

Aplikace Samotná aplikace je uložena ve složce *Application* jako spustitelný .jar archiv

Images Ve složce *Application\Images* jsouu umístěny obrázky nutné ke správnému chodu aplikace v režimu *User*

Testing jars V složce *TestingJars* jsou umístěny některé příklady archivů s fitness funkcemi

PDF Tato práce ve formátu PDF je uložena ve složce *Thesis*

JavaDoc Složka *JavaDoc* obsahuje vygenerovaný JavaDoc k aplikaci

JAR testovací aplikace Jar soubor *testing.jar* ve složce *TestingApp* obsahuje verzi aplikace kompilovanou tak, aby produkovala i *FINER* logy

Logy k testům Výstřižky z logů aplikace odkazované z testovacích protokolů jsou umístěné ve složce *Logs*

Zdrojové kódy Zdrojové kódy všech tříd aplikace jsou umístěné ve složce *Source*

A Uživatelský manuál

Tato příručka popisuje způsoby spuštění aplikce ve všech jejích režimech, návod na přidávání vlastního .jar archivu s fitness funkcí, spuštění genetického algoritmu a sběr výsledků.

A.1 Požadavky na prostředí

Všechny režimy aplikace vyžadují Javu ve verzi minimálně 1.8.

Požadavky na výkon se liší mezi jednotlivými částmi a závisí hlavně na problému, který chceme řešit. V zásadě ale platí, že *Workeři* by měli mít k dispozici co nejvíce výpočetního výkonu, část *Master* musí pro dobrou funkcionalitu mít zajištěno spojení ke všem *Workerům* přes protokol IPv4 a *User* musí jít propojit s *Masterem* také přes IPv4.

A.2 Spouštění

Všechny části aplikace se spouští z příkazové řádky zavoláním

```
java -jar <jarfile> <mode> <arguments>
```

kde <jarfile> je spustitelný archiv s celou aplikací, <mode> říká, ve kterém režimu aplikaci spustit a <arguments> jsou další parametry spuštění, pro každý režim jiné.

Možné režimy <mode> jsou „-w“ pro režim *Worker*, „-m“ pro *Master* a „-u“ pro režim *User*. Pokud je aplikace spuštěna úplně bez parametrů pouze zavoláním

```
java -jar <jarfile>
```

je automaticky spuštěn režim *User* bez dalších parametrů.

K aplikaci také patří složka *Images*, která obsahuje všechny obrázky k uživatelskému rozhraní. Je schválně umístěna mimo .jar soubor, jelikož tyto obrázky jsou nutné pouze k režimu *User* a není nutné je přikládat pro chod aplikace i v ostatních režimech.

A.2.1 Worker

Worker je hlavní výkonná část aplikace, která by měla být spuštěna jednou na každém počítači v distribuované síti. Pro nejvyšší výkon by také měl počet vláken u každého spuštěného *Workera* odpovídat počtu jader procesoru, na kterém běží. Spuštění aplikace k tomto režimu provedeme příkazem

```
java -jar <jarfile> -w <arguments>
```

Povolené argumenty jsou

```
-p, --port <port>          : port, kde apliace bude poslouchat
-n, --name <name>          : jméno této instance
-t, --threads <num>        : počet vláken k˜výpočtu
-jp, --jar-port <port>     : port pro˜file-server
-jf, --jar-folder <path>   : cesta ke složce s .jar soubory
```

Pokud máme aplikaci v archivu genetic.jar, složku s .jar archivy určíme jako *C:\Genetic\Files\Fitness*, jméno instance zvolíme jako *Worker 1*, počet vláken na 4 a port k aplikaci na 6001, spuštění provedeme příkazem

```
java -jar genetic.jar -w -p 6001 -n "Worker 1" -jf
"C:\Genetic\Files\Fitness\" -t 4
```

Po spuštění se všechny hodnoty vypíšou do konzole, aplikace poslouchá na daném portu a čeká na připojení *Mastera*.

A.2.2 Master

Master se stará o synchroniaci *Workerů*, migraci, sběr a prezentaci výsledků a umožňuje ovládání aplikace spojením s částí *User*. Spouští se příkazem

```
java -jar <jarfile> -m <arguments>
```

Přípustné agrumenty jsou

```
-p, --port <port>          : port, kde apliace bude poslouchat
-n, --name <name>          : jméno této instance
-jp, --jar-port <port>     : port pro˜file-server
-jf, --jar-folder <path>   : cesta ke složce s .jar soubory
-ws, --workers <list>      : seznam workerů pro˜připojení
```

Všechny argumenty kromě *-ws* jsou stejné jako v případě *Workera*, proto se zde zaměříme právě na formát zadávání seznamu *Workerů* pro připojení. Každý *Worker* se zadává ve formátu <ip adresa>:<port> a od sebe jsou odděleni středníky. Pokud se tedy chceme připojit ke třem *Workerům* na adresách 10.5.9.52 port 45, 192.168.5.78 port 5000 a 56.36.77.5 port 3501 zadáme argument *-ws* takto

```
-ws "10.5.9.52:45;192.168.5.78:5000;56.36.77.5:3501"
```

Celý příkaz pro spuštění by tak mohl vypadat například takto

```
java -jar genetic.jar -m -p 5000 -jf "C:\Genetic\Files\Master\"
-ws "10.5.9.52:45;192.168.5.78:5000;56.36.77.5:3501" -n
"Master"
```

Hned po spuštění se vypíšou zadané hodnoty a *Master* se ihned začne připojovat na *Workery*. V případš chybně zadaných argumentů se aplikace ukončí, jinak poslouchá na daném portu a čeká na připojení *Usera*. Pokud některý *Worker* není dostupný a připojení se nezdaří, vypíše se na konzoli příslušná hláška.

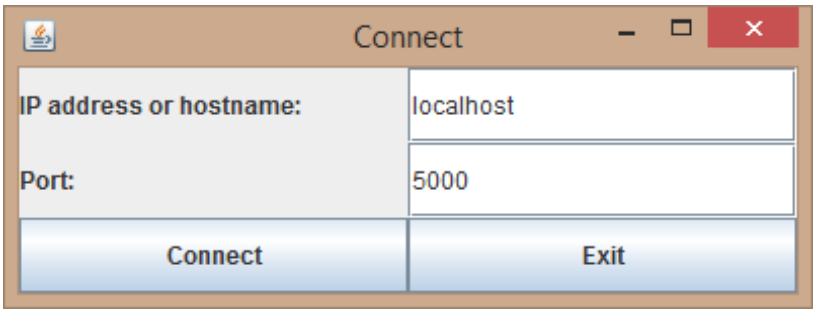
A.2.3 User

User je jedinná část aplikace, která má grafické uživatelské rozhraní. Režim *User* spustíme buď použitím parametru „-u“ nebo spuštěním aplikace bez parametrů. Jedinným možným argumentem za „-u“ je „-c <ip>:<port>“,

což má za následek okamžitý pokus o připojení na danou adresu a port a v případě úspěchu okamžité otevření hlavního okna. Příklady spuštění režimu *User*:

```
java -jar <jarfile>
java -jar <jarfile> -u
java -jar <jarfile> -u -c 192.168.5.65:5000
```

V případě otevření bez argumentu pro připojení vidí uživatel jako první okno pro zadání adresy a portu (viz obrázek A.1).



Obrázek A.1: Okno připojení

Po zadání adresy a portu se aplikace pokusí připojit a v případě úspěchu se otevře hlavní okno aplikace (obrázky A.2 a A.3). Zde se nachází většina ovládacích prvků aplikace a informace o průběhu algoritmu. Uprostřed okna vidíme stav a informace o *Masterovi* a jednotlivých *Workerech*. Stav *Mastera* může být

- *EMPTY*, což znamená, že není načteno žádné nastavení algoritmu
- *READY*, což znamená, nastavení je načteno a může se začít s výpočtem
- *COMPUTING*, kdy výpočet probíhá

Jednotliví *Workeři* mají ještě dva stavy navíc a to

- *NOT_CONNECTED*, když se s *Workerem* nedaří spojit
- *SENDING_RESULTS*, což znamená, že *Worker* posílá výsledky na *Master*

Obrázek A.2: Hlavní okno aplikace

Checkbox *update* u každého okna vypne nebo zapne aktualizaci informací o daném prvku. Vypnutá aktualizace může být výhodná pro menší zatížení sítě a navíc v případě velkého množství *Workerů* umožňuje scrollovat. Zde je výčet prvků na horní ovládací liště zleva a stručný popis jejich funkce:

Setup vyvolá nové okno pro nastavení nového algoritmu

Teardown zruší zavedené nastavení a umožní nahrát nové

Start zahájí algoritmus (v případě nového nastavení) nebo algoritmus zopakuje (v případě, že výpočet už skončil)

Obrázek A.3: Hlavní okno aplikace během výpočtu

Reconnect se znovu pokusí navázat spojení se všemi *Workery*. Použitelné pouze ve stavu *EMPTY*.

Load JAR vyvolá okno pro výběr souboru k odeslání na *Master*

Synchronize Po nahrání nebo aktualizaci všech .jar souborů se vyvoláním synchronizace všechny soubory pošlou z *Mastera* na všechny *Workery*

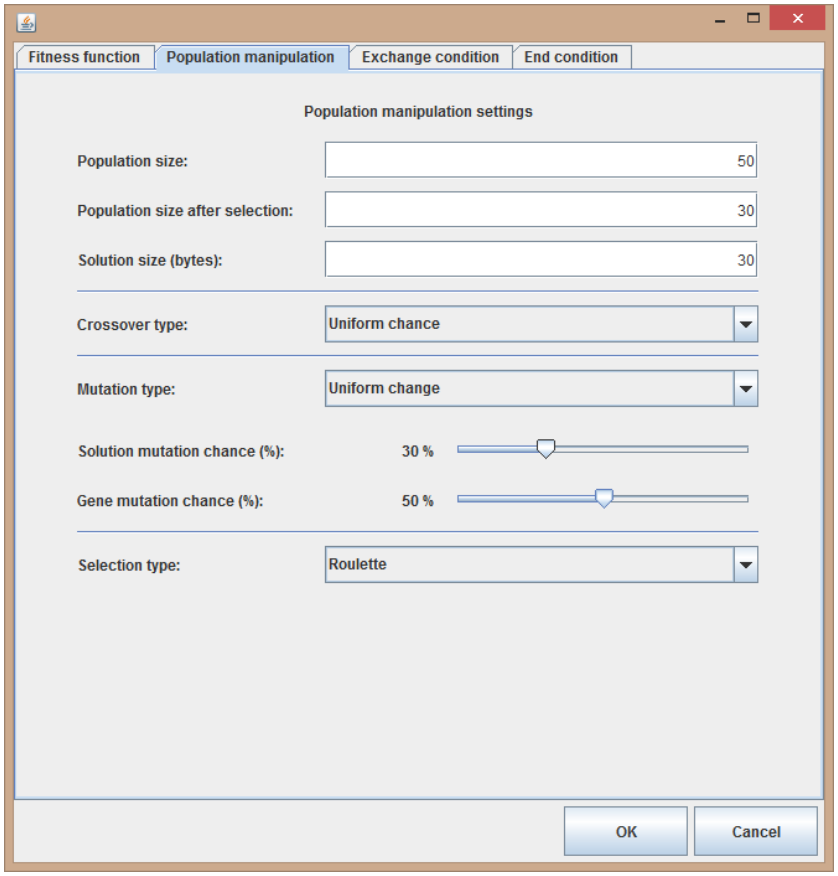
Send results dotaz na všechny zatím vypočtené výsledky algoritmů

Flush results vymaže z *Mastera* všechny výsledky

Exit ukončí aplikaci

A.2.4 Nastavení algoritmu

Nastavení genetického algoritmu má své vlastní okno, které je rozdělené do 4 panelů (viz obrázek A.4).



Obrázek A.4: Nastavení algoritmu

Tyto panely jsou *Fitness function*, kde se vybírá fitness funkce. Na výběr jsou již implementované funkce nebo možnost *External JAR*, kde si uživatel může zvolit svojí vlastní.

V části *Population manipulation* je možné navolit parametry populace, tedy její velikost, velikost po selekci, počet bytů jednoho řešení a možnosti křížení, mutace a selekce. Většina nastavení na tomto panelu má ještě další nastavení, kterým je možné daný genetický operátor více specifikovat.

Poslední dva panely obsahují nastavení podmínky migrace (nazvaná *exchange*) a ukončovací podmínky algoritmu. Možnosti na výběr jsou stejné

u obou a to:

- *Time Limit*, který udává počet sekund za který dojde k migraci / ukončení algoritmu.
- *Number of generations*, který udává počet generací (iterací algoritmu), za který dojde k migraci
- *Fitness function theshold*, kdy je podmínka splněna v okamžiku, kdy některý jedinec dosáhne daného limitu fitness funkce

A.2.5 Výsledky

Poté co algoritmus dokončí výpočet, jsou na *Master* nakopírovány všechny populace všech vláken ze všech připojených *Workerů*. Tyto výsledky na *Masterovi* zůstanou do té doby, než je někdo pomocí *Flush results* vymaže. Tedy přenáší se do dalších spuštění algoritmu, ale i do jiných nastavení algoritmů. Je to navrženo z důvodu testování, kdy si například můžeme pro jednu fitness funkci vyzkoušet více nastavení selekce nebo mutace a výsledky mít srovnány na jednom místě jako by se jednalo o výstup jednoho algoritmu.

Výsledky lze exportovat jako plaintext nebo jako XML soubor. Formát plaintextu je prostý zápis jednoho jedince na jednu řádku, kde jednotlivé geny (byty) jsou odděleny čárkami. Například

```
10,5,7,5,-5,45,-98,4,-102
45,45,6,-78,0,1,2,9,-50
78,5,-5,-5,-63,120,6,78
```

XML má jako kořenový element <results> a jednotlivé výsledky jsou elementy <result> s atributy *fitness* a *value*. Příklad:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<results>
  <result fitness="2.0" value="87,-128,9,119,-128,70,102"/>
  <result fitness="1.0" value="66,-128,-59,37,-84,70,-24"/>
</results>
```

B Implementace vlastních fitness funkcí

Zde ve stručnosti uvedeme formát vlastních fitness funkcí, které můžeme jako .jar soubory vkládat do běžící aplikace a postup jejich přidání.

B.1 Formát fitness funkce

Vkládaná fitness funkce musí být definvána jako metoda v Javě, které má jedinný parametr a to pole bytů a která jako návratovou hodnotu má typ double. Dále musí být *statická* a *veřejná*. Zapsáno kódem:

```
public static double fitness(byte[] array) {

   // do some computations on array

   return <value>;
}
```

Tato metoda musí být umístěna ve veřejné třídě. Tato třída musí být přeložena tak, aby šla spustit na Javě 1.8 a výsledný .class soubor musí být zabalen do .jar archivu. Pokud během výpočtu fitness funkce dojde k jakékoliv výjimce nebo chybě je jako hodnota fitness použita konstanta *Double.NaN*. Platí i v případě, že se funkci nebo celý archiv vůbec nepodaří najít, potom budou mít všichni jedinci z populace hodnotu fitness nastavenou na *NaN*, proto je lepší si před spuštěním funkčnost nejdříve vyzkoušet na kratším algoritmu.

B.2 Distribuce archivu

Archiv s funkcí musí být přítomen na všech *Workerech* a na *Masteru* a to ve složce, která se zadává jako parametr při spuštění. Archiv můžeme distribuovat ručně prostým kopírováním do příslušných složek na příslušných

počítačích nebo použít funkci pro synchronizaci, která se součástí aplikace.

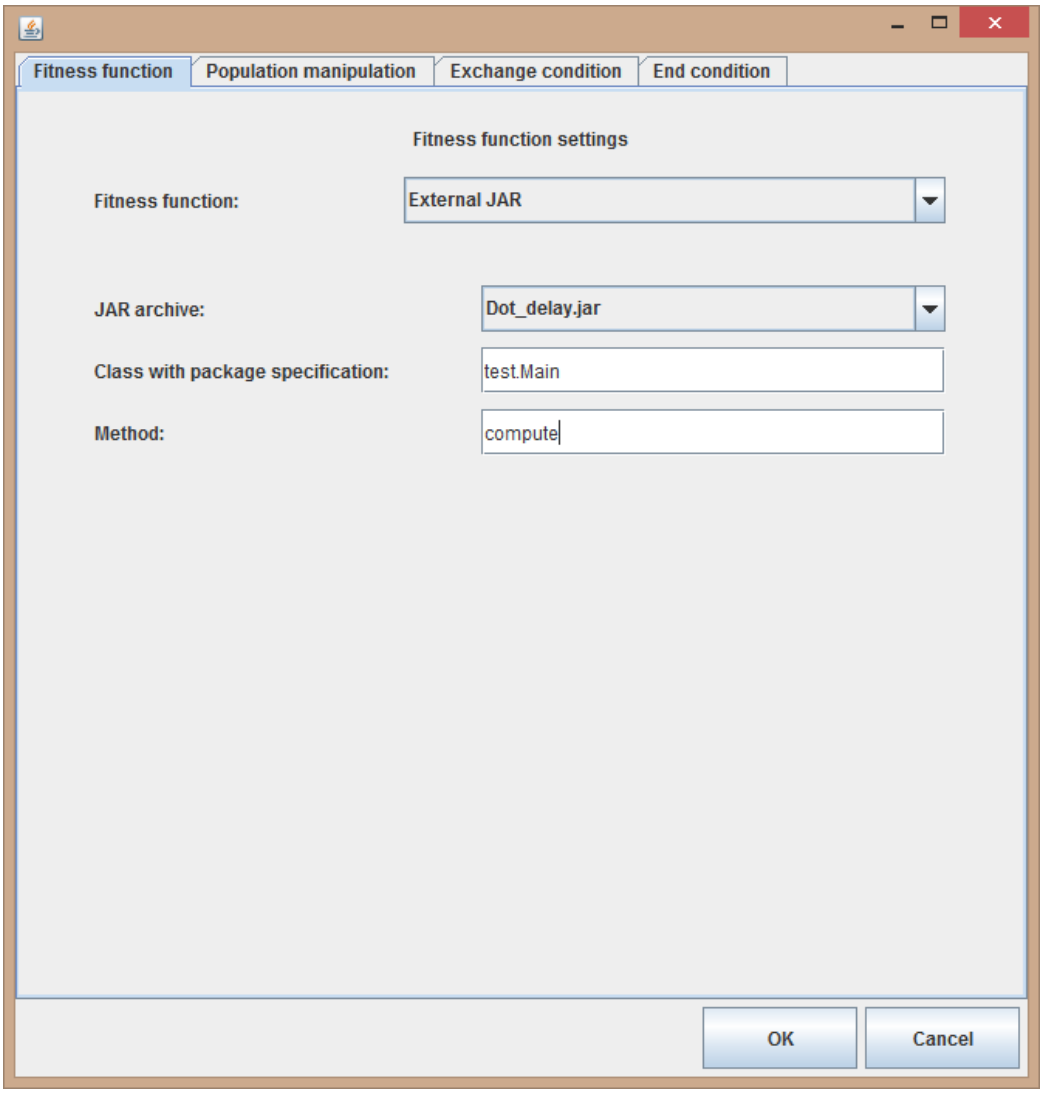
Funkce pro nahrávání a synchronizaci .jar souborů používá externí server a je určena k nahrávání malých .jar archivů, které moc nevytíží síť. Úspěšné nebo neúspěšné nahrání souboru a synchronizace nejsou užovateli nijak indikovány a je tedy na něm, aby si ověřil, že se soubor na všech místech opravdu vyskytuje.

Pro větší soubory než 1MB (například pro fitness funkce, které vyžadují další testovací data) je doporučeno kopírovat soubory do složek ručně. Také pokud fitness funkce vyžaduje ke svému chodu další soubory s jinou příponou než .jar, musíme tyto soubory nahrát na počítače v síti ručně. Doporučený postup je *Workery* a *Master* spustit až po nahrání všech souborů do správných složek.

B.3 Výběr funkce

Samotný výběr, který archiv a kterou funkci z něj použít se nastavuje v nastavení genetického algoritmu na panelu *Fitness function*, viz obrázek B.1.

Jako funkci vybereme *External JAR*. Seznam archivů, nahraných a synchronizovaných na *Masteru* poskytuje výběr ozančený jako *JAR archive*. Do dalšího pole napíšeme specifikaci třídy včetně balíčků za použití tečkové notace. Poslední textové pole je určené pro název samotné funkce. Příklad na obrázku B.1 je platné nastavení pro archiv *Dot_delay.jar* dodávaný společně s aplikací.



Obrázek B.1: Nastavení algoritmu

C One-point crossover log

Před křížením

```
13,84,95,-111,18,114,103,64,-40,-101,
87,-119,3,-13,95,112,59,10,-115,31,
68,103,-65,82,-94,-37,74,64,-48,-31,
-50,-60,17,122,122,-7,-69,67,-35,-10,
119,127,-60,-22,-90,-18,125,-33,25,-91,
49,-78,-125,-12,18,-97,125,71,122,-26,
24,109,41,97,-94,-2,-26,-123,87,-81,
-60,5,32,32,-9,-50,114,-115,-67,127,
95,17,-43,-46,-27,39,-17,5,-39,17,
-16,114,52,-26,16,-5,-114,-34,-125,119,
```

Po křížení

```
87,-119,3,-13,95,-5,-114,-34,-125,119,
-50,-60,17,97,-94,-2,-26,-123,87,-81,
13,-119,3,-13,95,112,59,10,-115,31,
-16,114,52,-26,18,-97,125,71,122,-26,
-16,114,52,-12,18,-97,125,71,122,-26,
95,17,-43,-46,-27,39,-17,71,122,-26,
49,-78,-125,-12,18,-2,-26,-123,87,-81,
13,84,95,-46,-27,39,-17,5,-39,17,
13,127,-60,-22,-90,-18,125,-33,25,-91,
-60,5,32,97,-94,-2,-26,-123,87,-81,
-60,5,32,32,-9,-50,114,-115,-67,127,
68,103,-65,82,-94,-37,74,64,-48,-31,
-16,114,52,82,-94,-37,74,64,-48,-31,
13,127,-60,-22,-90,-18,125,-33,25,-91,
119,127,-60,-22,-90,-18,125,-33,25,-91,
-16,114,52,97,-94,-2,-26,-123,87,-81,
-16,114,52,-26,-94,-2,-26,-123,87,-81,
13,84,95,-111,18,114,103,64,-40,127,
13,84,95,-111,-9,-50,114,-115,-67,127,
95,-60,17,122,122,-7,-69,67,-35,-10,
```

Celý log pro jednu generaci je v souboru *1.txt*.

D Two-point crossover log

Před křížením

```
66,2,101,-62,-25,59,109,-85,82,-102,
97,-123,95,119,-92,86,77,-110,-80,66,
78,43,21,67,-35,39,-53,76,-112,52,
2,23,112,-54,45,65,40,73,-59,54,
12,85,21,39,15,16,45,-1,-66,-66,
```

Po křížení

```
66,2,101,-62,-25,59,109,-85,82,-102,
2,23,112,39,15,16,40,73,-59,54,
66,2,21,-62,-25,59,109,-85,82,-102,
97,-123,21,39,15,16,45,-1,-66,66,
78,85,21,39,15,39,-53,76,-112,52,
97,-123,95,119,-92,86,77,-85,82,-102,
2,23,112,-54,45,65,40,73,-59,54,
2,23,112,-54,15,16,45,73,-59,54,
12,85,21,39,15,16,45,-1,-66,-66,
66,23,112,-62,-25,59,109,-85,82,-102,
```

Celý log pro jednu generaci je v souboru *2.txt*.

E Naměřené hodnoty pro zrychlení

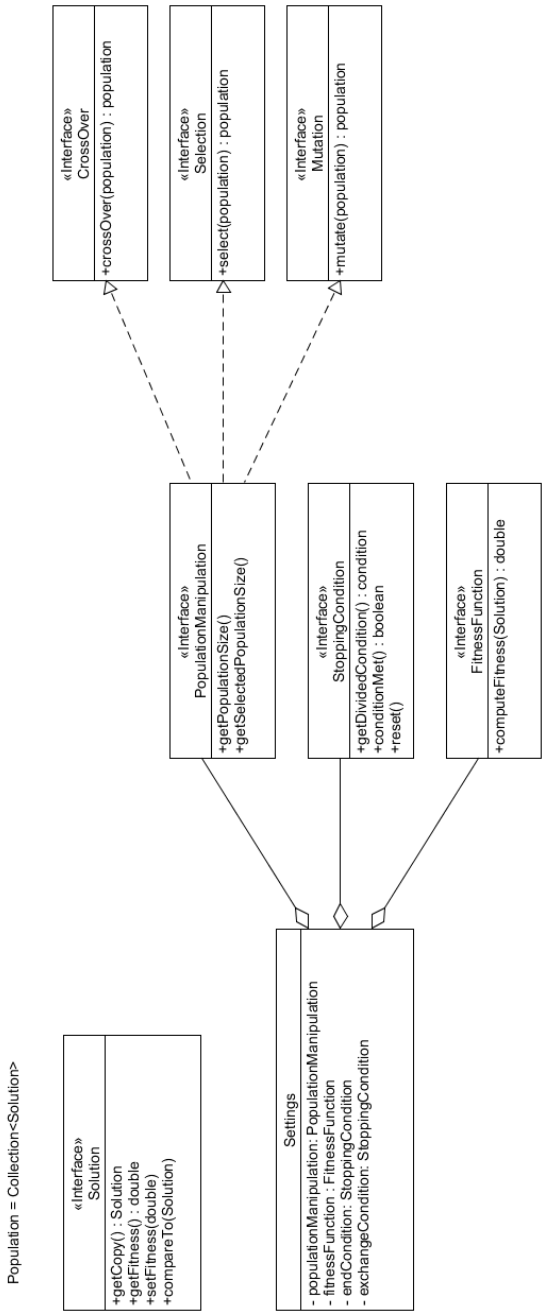
Tabulka E.1: Naměřené hodnoty pro počítač A

Počet vláken	Počet evaluací fitness funkce					Průměr	Zrychlení	Efektivita
	Měření							
	1	2	3	4	5			
1	127	127	127	127	127	127	-	-
2	277	285	313	349	313	307.4	2.420	121.02%
3	335	281	313	305	277	302.2	2.380	79.32%
4	275	301	325	269	297	293.4	2.310	57.76%
5	287	267	279	317	313	292.6	2.304	46.08%
6	311	291	313	279	289	296.6	2.335	38.92%
7	299	309	269	259	243	275.8	2.172	31.02%
8	299	271	287	283	301	288.2	2.269	28.37%

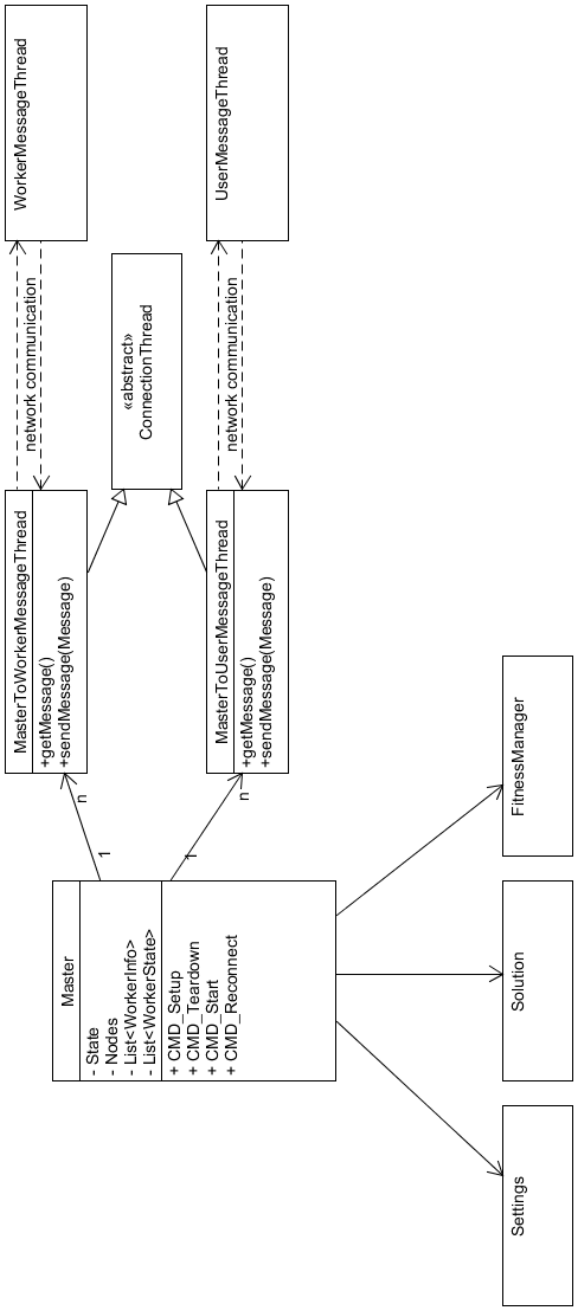
Tabulka E.2: Naměřené hodnoty pro počítač B

Počet vláken	Počet evaluací fitness funkce					Průměr	Zrychlení	Efektivita
	Měření							
	1	2	3	4	5			
1	147	143	143	147	141	144.2	-	-
2	233	209	185	173	169	193.8	1.344	67.20%
3	431	435	441	441	413	432.2	2.997	99.91%
4	683	641	621	629	609	636.6	4.415	110.37%
5	689	705	679	707	689	693.8	4.811	96.23%
6	747	745	765	747	735	747.8	5.186	86.43%
7	789	793	803	793	805	796.6	5.524	78.92%
8	827	807	839	845	801	823.8	5.713	71.41%

F UML diagram pro `genetic.ifaces`



G UML diagram pro `master`



H Kód metody pro použití reflexe

```
public void loadMethod() throws Exception {
  JarFile jarFile = new JarFile(path + filename);
  try {
    Enumeration<JarEntry> e = jarFile.entries();
    URL[] urls = { new URL("jar:file:" + path + filename + "!/") };
    URLClassLoader cl = URLClassLoader.newInstance(urls);

    boolean classFound = false;

    while (e.hasMoreElements()) {
      JarEntry je = (JarEntry) e.nextElement();
      if (je.isDirectory() || !je.getName().endsWith(".class")) {
        continue;
      }
      // -6 because of .class
      String className = je.getName().substring(0,
          je.getName().length() - 6).replace('/', '.');
      if (className.equals(classToLoad)) {
        classFound = true;
        Class<?> c = cl.loadClass(className);

        Method m = c.getDeclaredMethod(methodToUse, byte[].class);

        this.method = m;
        if (this.method == null) throw new Exception("Method not
            found");
      }
    }
    if (!classFound) throw new Exception("Class not found");
  }
  catch (Exception e) {
    e.printStackTrace();
    this.method = null;
  }
  finally {
    jarFile.close();
  }
}
```

I Kód třídy `Message.java`

```
package networking.messages;

import java.io.Serializable;

public class Message implements Serializable {

   private static final long serialVersionUID =
       3018121326309853506L;

   private final String content;
   private final Serializable appendix;

   public Message(String content, Serializable appendix) {
      super();
      this.content = content;
      this.appendix = appendix;
   }

   public Message(String content) {
      this(content, null);
   }

   public String getContent() {
      return content;
   }

   public Serializable getAppendix() {
      return appendix;
   }

}
```

J Testovací sekvenční algoritmus

```
package main;

/*
imported classes were ommited
*/

public class TestingMain {

   public static void main(String[] args) throws Exception {

      Mutation m = new EdgeValue(0.5, 0.5, EdgeValue.TYPE_MAX);
      Selection s = new Truncation(2);
      CrossOver co = new OnePoint(2);

      PopulationManipulation p = new ByteArrayManipulations(2, 2,
          30, m, co, s);

      StoppingCondition c2 = new TimeLimit(1000 * 60);

      ExternalJar fit = new ExternalJar("dot_delay.jar",
          "test.Main", "compute");
      fit.setPath("<folder>");
      fit.loadMethod();

      Worker w = new Worker(10, "AA", "<folder>");
      GeneticAlgorithm ga = new GeneticAlgorithm(1, w);

      GeneticThread t = new GeneticThread(p, c2, fit, ga);
      t.start();

   }

}
```