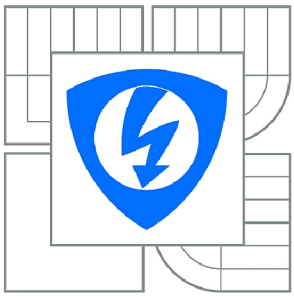
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

ÚSTAV TELEKOMUNIKACÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

DEPARTMENT OF TELECOMMUNICATIONS

ŠKÁLOVATELNOST MODELU GENETICKÉHO PROGRAMOVÁNÍ

SCALABILITY OF GENETIC PROGRAMMING MODEL

DIPLOMOVÁ PRÁCE

MASTER'S THESIS

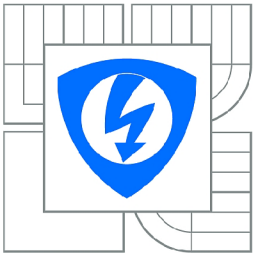
AUTOR PRÁCE — Bc. LUKÁŠ KOZEMPEL

AUTHOR

VEDOUCÍ PRÁCE — Ing. RADIM BURGET

SUPERVISOR

BRNO 2010



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky a komunikačních technologií

Ústav telekomunikací

Diplomová práce

magisterský navazující studijní obor
Telekomunikační a informační technika

Student:	Bc. Lukáš Kozempel	***ID:***	74840
Ročník:	2	***Akademický rok:***	2009/2010

NÁZEV TÉMATU:

Škálovatelnost modelu genetického programování

POKYNY PRO VYPRACOVÁNÍ:

Seznamte se s pojmy problematikou genetického programování (GP) a popište je. Seznamte se s ostrovním modelem pro GP a vrstvení struktury populace dle stáří jedinců (Age-layer population structure). Navrhněte a implementujte tzv. ostrovní model a zabezpečte komunikaci mezi jednotlivými stanicemi. Implementujte aplikaci pro spuštění v prostr(edí výpočetního gridu. Popište její spuštění a jeho parametry.

DOPORUČENÁ LITERATURA:

[1] KOROL A. B., PRIEGEL S. I.,Recombination Variability and Evolution: Algorithms of estimation and population-genetic models, Springer; 1 edition (August 1994)
[2] Koza J. R., Genetic Programming: On the Programming of Computers by Means of Natural Selection, The MIT Press

Termín zadání:	29.1.2010	***Termín odevzdání:***	26.5.2010

Vedoucí práce: Ing. Radim Burget

prof. Ing. Kamil Vrba, CSc.
Předseda oborové rady

UPOZORNĚNÍ:

ABSTRAKT

Tématem práce je praktická realizace jednoho ze způsobů paralelního zpracování genetického programování, tzv. ostrovního modelu. První část je teoretická. Popisuje pojmy genetického programování, Age-layered population structure a ostrovní model. Ve druhé části je popsána realizace ostrovního modelu v jazyce Java.

KLÍČOVÁ SLOVA

genetické programování, ostrovní model, java

ABSTRACT

Theme of this thesis is practical realization of so-called Island model which is one of way of parallel genetic programming. First part is theoretical. This part is describing terms of genetic programming, age-layered population structure and island model. In second part of thesis is described realization of island model in Java language.

KEYWORDS

genetic programming, island model, java

KOZEMPEL, L. *Škálovatelnost modelu genetického programování*. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2010. 60 s. Vedoucí práce Ing. Radim Burget.

PROHLÁŠENÍ

Prohlašuji, že svou diplomovou práci na téma „Škálovatelnost modelu genetického programování“ jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení § 152 trestního zákona č. 140/1961 Sb.

V Brně dne ....

(podpis autora)

Děkuji vedoucímu diplomové práce Ing. Radimu Burgetovi za velmi užitečnou metodickou pomoc a cenné rady při zpracování práce. Dále děkuji MetaCentru za umožnění přístupu k jejich počítačům.

OBSAH

SEZNAM OBRÁZKŮ

ÚVOD

Genetické programování je systematická metoda řešení problému, kdy algoritmus na začátku „ví“, co je potřeba udělat a hledá řešení, jak to udělat aplikováním operátorů, podobným genetickým operátorům vyskytujícím se v přírodě, na populaci programů. Výstupem genetického programování je počítačový program.

Algoritmu genetického programování můžeme předložit množinu naměřených hodnot proměnných a algoritmus bude hledat funkci, která by co nejlépe aproximovala vztah mezi proměnnými. Tímto způsobem je možné znovuobjevit fyzikální zákon nebo dokonce vytvořit zcela nový vynález.

V genetickém programování se náhodně vygenerovaná populace počítačových programů za pomoci genetických operátorů (zejména křížení a mutace) reprodukuje a vzniká tak nová populace obsahující jedince reprezentované programy s lepšími vlastnostmi. Tento cyklus se opakuje, dokud algoritmus nedojde k přijatelnému řešení. Při řešení složitějších problémů narůstá čas potřebný k nalezení řešení problému. Pro zkrácení času výpočtů složitých problémů a snížení pravděpodobnosti, že řešení uvízne v lokálním optimu, je možné využít paralelního zpracování výpočtů.

Tato práce se zabývá implementací jednoho ze způsobů paralelního zpracování algoritmů genetického programování, tzv. ostrovního modelu. Při tomto způsobu paralelizace je počáteční populace rozdělena na tzv. subpopulace. Je vytvořeno tolik subpopulací, kolik je v síti stanic, které se podílejí na výpočtech. Každá stanice zpracovává jednu subpopulaci, která se vyvíjí odděleně od ostatních. Kvůli efektivnosti využití prostředků stanice jsou tyto subpopulace dále rozděleny na tolik částí, kolik má procesor jader a jsou zpracovávány jednotlivými jádry paralelně. Pro urychlení výpočtů dochází mezi subpopulacemi jednotlivých jader k výměně jedinců.

Mezi subpopulacemi dochází ke dvěma druhům výměny jedinců (migrace). Prvním druhem je migrace mezi subpopulacemi, které se nacházejí na stejné stanici. V tom případě není potřeba vytvářet zabezpečené síťové spojení. Dojde pouze k překopírování migrujícího jedince do vstupní fronty cílové subpopulace a při další generaci je začleněn do populace. Druhým typem je migrace mezi populacemi nacházejícími se na různých stanicích v síti. V druhém případě je vytvořeno zabezpečené síťové spojení mezi stanicemi a migrující jedinec je odeslán cílové stanici.

Aby nebylo nutné do konfiguračního souboru každé stanice vkládat IP adresy a počty procesorů všech ostatních stanic účastnících se výpočtů, funguje mezi stanicemi automatické rozesílání informací o stanicích. Na jedné stanici je aplikace spuštěna v řídícím módu. Ostatní stanice po spuštění posílají této řídící stanici svoji IP adresu a číslo udávající počet jader procesorů. Řídící stanice došlé informace přeposílá všem ostatním stanicím a ty si údaje ukládají do tabulek, které následně využívají při migraci jedinců.

První část práce se věnuje teoretickému úvodu, kde jsou objasněny pojmy genetického programování jako např. metody výběru jedinců, reprodukční operátory, age-layered population structure, ostrovní model atd. Potom následuje část zabývající se vlastní implementací paralelního zpracování, zabezpečené komunikace, migrace mezi stanicemi v síti i jádry procesorů a automatického rozesílání informací o ostatních stanicích účastnících se výpočtů. Na konci této druhé části je popsáno spuštění ostrovního modelu ve výpočetním gridu.

1 POJMY GENETICKÉHO PROGRAMOVÁNÍ

1.1 Genetický algoritmus

Genetický algoritmus je založen na zjednodušeném Darwinovském principu evoluce, kde se živočichové a rostliny vyvíjeli po mnoho generací podle principu přirozeného výběru a přežití těch nejschopnějších.

Pracuje s populací jedinců, kde každý představuje možné řešení problému. Ve většině případů je jedinec reprezentován jedním chromozomem skládajícím se z řetězce bitů. Každému individuu je přiřazeno ohodnocení, které kvalitativně vyjadřuje míru vlastností, které vedou k řešení daného problému. Imitací přirozeného výběru a pomocí operátorů křížení a mutace dochází k reprodukci a k vytvoření nové populace.[5][17]

Pokud je tento proces mnohokrát zopakován, vlastnosti jedinců představující lepší řešení daného problému se šíří populací a kombinují se s vlastnostmi dalších jedinců s vysokým ohodnocením. Po určitém počtu generací se objeví jeden nebo i několik jedinců, kteří představují přijatelné řešení daného problému.[9]

Podrobnější informace o genetických algoritmech lze nalézt v [27] a v [20].

1.2 Genetické programování

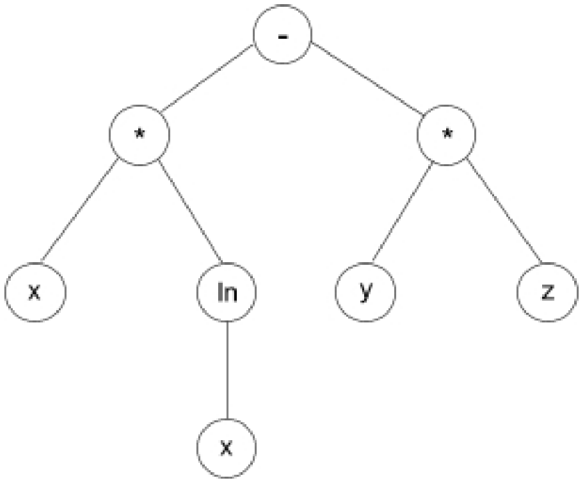
Genetické programování je možno považovat za rozšíření genetických algoritmů. Chromozomy již nejsou řetězce pevné délky, ale hierarchicky strukturované počítačové programy, které po spuštění mohou představovat možné řešení daného problému. Tyto struktury jsou zpravidla založeny na ADT stromy (tree-gp) či orientovaných acyklických grafech (lineární GP). Podle [23] jsou v genetickém programování nejpoužívanějšími strukturami ADT stromy a proto budou použity i při popisu v dalším textu. Informace o lineárním genetickém programování je možné najít v [23].[9][13]

V pokročilejší formě genetického programování mohou být programy složeny z více podprogramů. V tomto případě je genetický algoritmus reprezentován sadou stromů, které spojuje hlavní uzel.[23]

Syntaktický strom popisuje strukturu programu a skládá se z terminálů (proměnných a konstant) a neterminálů (funkcí). Pro daný problém je definována množina funkcí F (například aritmetické operace, matematické funkce, logické funkce...) a množina terminálů T, kterými je možné daný problém řešit a vytvořit syntaktický strom. Příklad jednoduchého syntaktického stromu je na Obr. 1.1. Počet hran vycházejících z uzlu, který odpovídá funkci f je dán počtem argumentů této funkce.

Podle [12] musí množina terminálů a neterminálů splňovat požadavky uzavřenosti a postačitelnosti.

Požadavek uzavřenosti znamená, že funkce z množiny funkcí F je schopná přijmout jakýkoliv terminál z množiny T a výsledek jakékoliv funkce z množiny F. Tímto se zabraňuje vzniku chyb za běhu programu. Obsahuje-li například množina funkcí F pouze logické operátory AND, OR, NOT a množina terminálů T proměnné typu boolean, potom je požadavek uzavřenosti splněn. Pokud ale množina funkcí F obsahuje aritmetickou operaci dělení a v množině terminálů budou reálné proměnné, potom tento požadavek splněn není, protože je možné vygenerovat případ, kdy je požadováno dělení nulou. V tomto případě je pro splnění požadavku uzavřenosti nutné definovat funkci, která korektně řeší výjimečné situace a vyhne se tak případným chybám za běhu programu. Jednou z cest řešení tohoto problému jsou gramatiky[22]. Další informace o genetickém programování lze najít v [26] a [21].[12]



Obr. 1.1: Syntaktický strom výrazu x.ln(x)-y.z [9]

Požadavek postačitelnosti vyžaduje, aby bylo pomocí množiny funkcí F a množiny terminálů T možné vyjádřit řešení daného problému.

1.3 Počáteční populace

Programy v počáteční populaci jsou vytvářeny náhodným výběrem funkcí a terminálů z definovaných množin F a T. Aby nebyly syntaktické stromy v počáteční populaci příliš složité, omezuje se jejich maximální hloubka (délku cesty z kořene stromu k nejvzdálenějšímu terminálu). Pro generování syntaktických stromů počáteční populace existují dvě základní metody:

- Úplná metoda - všechny stromy mají předepsanou maximální hloubku. Pokud je generován uzel stromu v hloubce, která je menší než maximální hloubka, náhodný výběr je možný pouze z množiny funkcí F. V maximální hloubce je potom možný výběr pouze z množiny terminálů T.[23][9]

- Růstová metoda - vznikají mnohem rozmanitěji tvarované syntaktické stromy. S výjimkou poslední vrstvy (s maximální hloubkou) se v každé vrstvě náhodně rozhoduje, jestli bude použita funkce nebo terminál. Pokud je použit terminál, je větev hotova bez ohledu na to, jestli bylo dosaženo maximální hloubky. Správným nastavením pravděpodobnosti výběru funkce a výběru terminálu je možné ovlivňovat výsledek generování. Pokud budou zvýhodněny terminály, výsledné stromy budou jednodušší a menší. Pokud budou naopak zvýhodněny funkce, generované stromy budou pravidelnější a větší.[23][9]

Pro širokou škálu problémů je podle [12] nejlepší metoda půl napůl, protože většinou předem nevíme, jakou velikost a tvar má výsledné řešení mít. Tato metoda generuje rozmanité stromy různých tvarů a velikostí. Pokud je maximální hloubka počáteční populace 6, potom je vygenerováno 20 % stromů s maximální hloubkou 2, 20 % s maximální hloubkou 3 atd. až do hloubky 6. Pro každou hodnotu hloubky je vytvořeno 50 % stromů úplnou metodou a 50 % růstovou metodou.

1.4 Selekce

Po vytvoření počáteční populace, je každý syntaktický strom ohodnocen příslušnou fitness funkcí podle vhodnosti jeho vlastností a podle tohoto ohodnocení je proveden výběr dvojic jedinců za účelem reprodukce. Podle [20] jsou nejpoužívanějšími mechanismy selekce:

- Ruletový mechanismus selekce s pravděpodobností výběru individua přímo úměrnou jeho ohodnocení (fitness-proportionate selection) - Velikost výseče na ruletovém kole přiřazené jedinci závisí na jeho ohodnocení. Čím je jeho ohodnocení vyšší, tím větší výseč na ruletovém kole je mu přidělena. Jedinci s vyšším ohodnocením mají tedy větší šanci, že budou vybráni. Ruletové kolo vytvoříme tak, že sečteme ohodnocení všech členů populace a v poměru ohodnocení jedince a sečteného ohodnocení určíme jednotlivé kruhové výseče.

- Stochastické univerzální vzorkování (Stochastic universal sampling) - Na rozdíl od předchozího mechanismu, který se musí k výběru k rodičů použít k-krát, tento vybírá všech k rodičů najednou. V ruletovém kole je místo jedné kuličky k

kuliček, které se současně pohybují po ruletovém kole. Tento mechanismus pomáhá k udržení rozmanitosti jedinců určených k další reprodukci, avšak stejně jako předchozí mechanismus i tento neřeší problém předčasné konvergence.

- Výběr podle pořadí (Rank selection) - Oproti předchozím dvěma metodám, tato metoda předchází předčasné konvergenci do lokálního optima, protože výseče na ruletovém kole nejsou přidělovány úměrně hodnotě fitness funkce. Jedinci jsou seřazeni podle velikosti ohodnocení a poté (podle jejich pořadí) je jim přidělena výseč na ruletovém kole. Pravděpodobnost, že bude i-tý jedinec vybrán, je dána vztahem [9]:

$$p_i = \frac{i}{\sum_{j=1}^{N} j} = \frac{2i}{N(N+1)}, \tag{1.1}$$

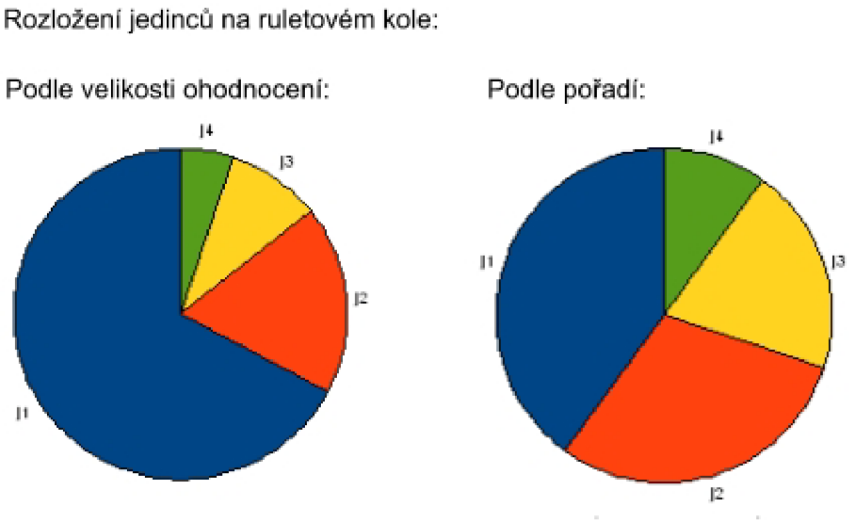
  kde N je velikost populace a $i \in \{1, ..., N\}$. Např. pokud bychom měli čtyři jedince, potom by přiřazené výseče byly podle pořadí od nejlépe ohodnoceného jedince: 40%, 30%, 20% a 10%. Výhoda tohoto mechanismu je ta, že pomáhá udržovat selektivní tlak i ke konci výpočtu, kdy už převládají silní jedinci a rozdíl v jejich ohodnocení je již nepatrný. Problém může nastat v situaci, kdy v populaci nejsou žádní výrazní jedinci a jejich rozptyl ohodnocení je nepatrný. V tom případě budou rozdíly mezi jedinci zveličeny a rozdíl v ohodnocení nejlepšího a nejhoršího individua bude N-násobný (N je velikost populace). Porovnání rozložení jedinců na ruletovém kole u mechanismu selekce s pravděpodobností výběru individua přímo úměrnou jeho ohodnocení a výběru podle pořadí je na Obr. 1.2.[9]

- Turnajová selekce - U této metody je z populace náhodně vybráno k jedinců, z nichž pro další reprodukci je vybrán jedinec s nejvyšší hodnotou ohodnocení. Všech k jedinců je vráceno zpět do populace a proces se opakuje, dokud není vybrán potřebný počet rodičů k reprodukci. Podrobně je tato metoda popsána v [8].

- Elitizmus - Může být použit u většiny metod výběru. Umožňuje genetickému algoritmu ponechat si nejlepší jedince z aktuální populace a bez jakéhokoliv výběru je převést do další generace. Bez použití elitizmu by tito jedinci vůbec nemuseli být vybráni pro postup do další generace nebo by mohli být zničeni mutací. Podle [20] mnoho badatelů potvrdilo, že elitizmus značně zvyšuje výkon genetických algoritmů.

- Setrvalý stav - U této metody je v každé generaci nahrazena jen malá část populace (nejhůře ohodnocení jedinci) potomky a zbytek populace zůstává nezměněn.

jedinec	pravděpodobnost podle ohodnocení	pravděpodobnost podle pořadí
j1	67,2 %	40 %
j2	18,5 %	30 %
j3	9,1 %	20 %
j4	5,2 %	10 %



Obr. 1.2: porovnání fitness-proportionate selection a rank selection[9]

Některé další metody jsou popsány v [8].

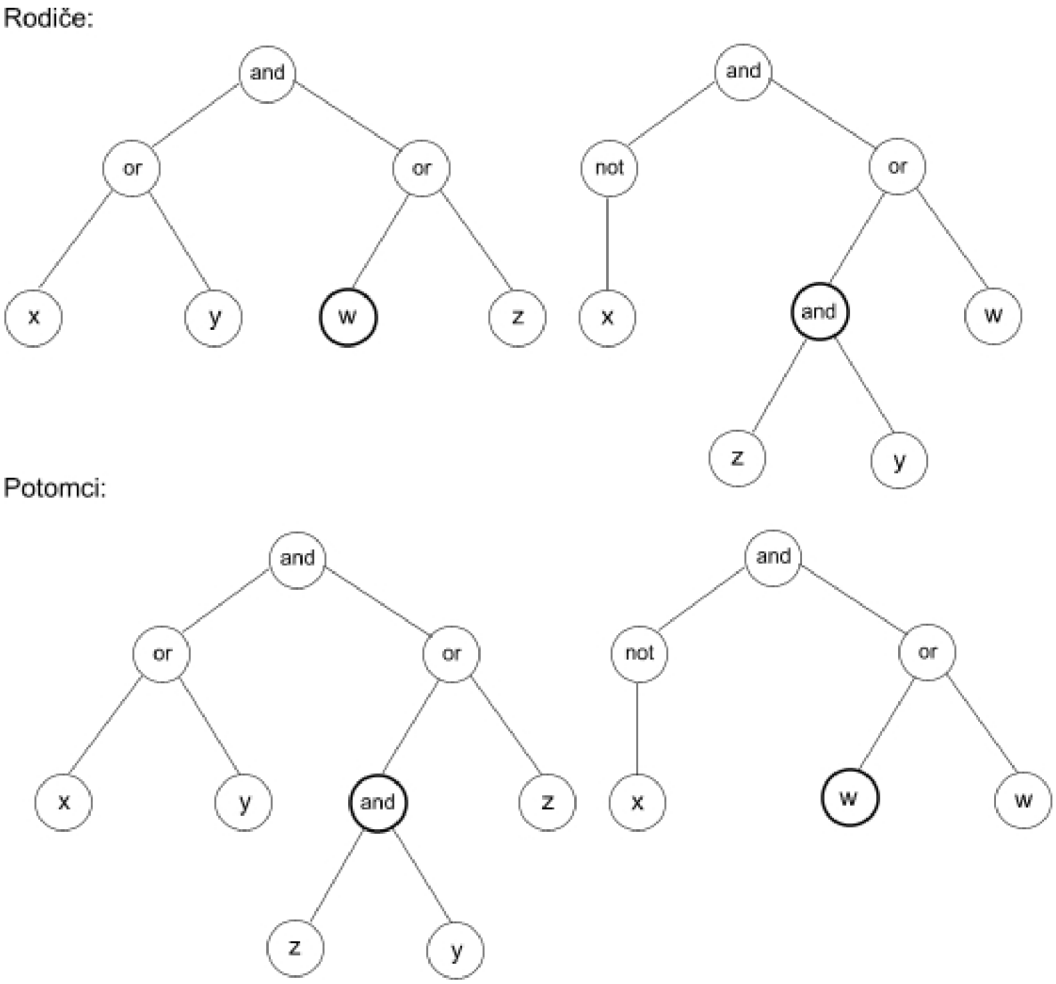
1.5 Genetické operátory

1.5.1 Operátor křížení

Po provedení výběru jedinců pro další reprodukci následuje vytvoření potomků. K tomu jsou využívány tři základní operátory: křížení, mutace a reprodukce. V některých literaturách ([14]) lze najít i jiné alternativní operátory.

Křížení je prováděno tak, že se v každém rodičovském syntaktickém stromě náhodně zvolí bod křížení a poté jsou prohozeny podstromy nacházející se pod těmito body křížení. Příklad takového křížení je na Obr. 1.3. Zvýrazněné uzly jsou body křížení.

Aplikováním operátoru křížení může nastat situace, že velká část stromu jednoho rodiče je nahrazena jedním listem stromu druhého rodiče. Tím vzniká strom velké hloubky. Postupně by vznikaly velké a komplikované struktury. Proto je dobré omezit maximální hloubku vznikajících stromů potomků. V [12] je doporučená maximální



Obr. 1.3: Příklad operace křížení

hloubka trojnásobek maximální hloubky v počáteční populaci. Pokud je křížením vytvořen strom, který má větší hloubku než je maximální hloubka, je odmítnut a nahrazen jedním z rodičů.

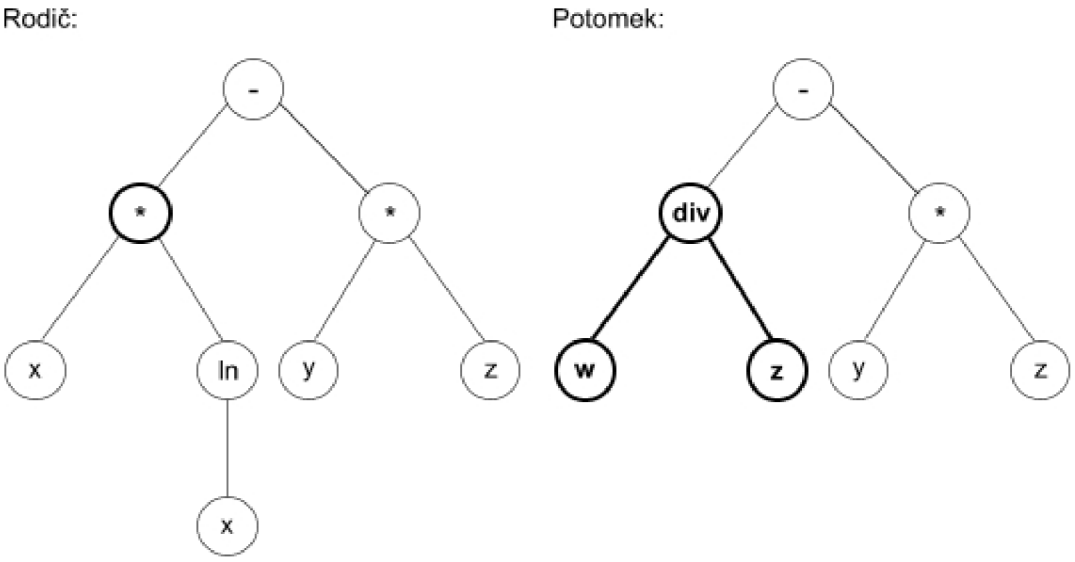
1.5.2 Operátor reprodukce

Na základě selekčního mechanismu je z populace vybrán jedinec a tento jedinec je beze změny překopírován do nové populace.

1.5.3 Operátor mutace

Operátor mutace náhodně a s malou pravděpodobností (0,001 až 0,05) přináší náhodné změny do struktur dané populace. Tento operátor vnáší do populace rozmanitost a brání tak předčasné konvergenci. Při aplikaci tohoto operátoru je náhodně vybrán uzel a celý podstrom, který z tohoto uzlu vychází je nahrazen náhodně

vygenerovaným stromem. Příklad operace mutace je na Obr. 1.4. Na rodičovském stromě je tučně vyznačen vybraný uzel pro mutaci, na stromě potomka potom nový podstrom.



Obr. 1.4: Příklad operace mutace

Maximální hloubka se obvykle stanovuje stejná jako maximální hloubka počáteční populace. Podle [9] existují i další typy operátoru mutace:

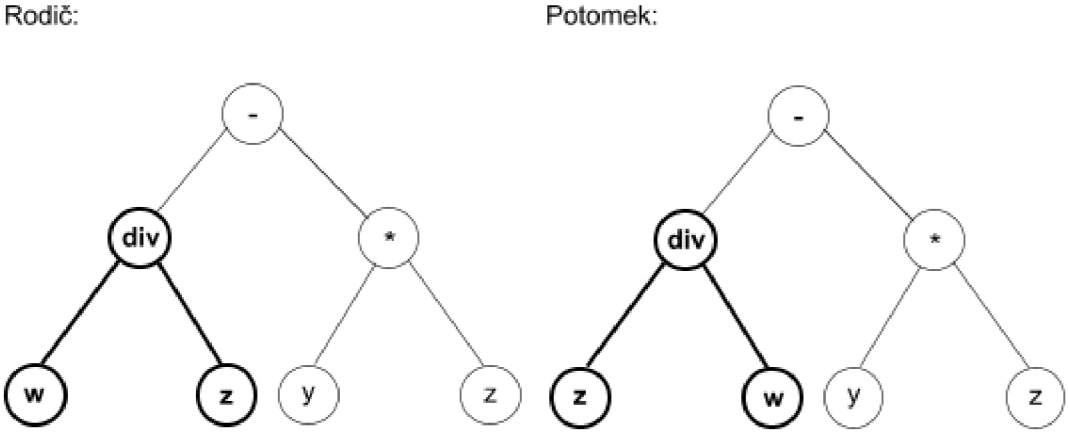
- Uzlová mutace - nahrazuje neterminální uzel neterminálem se stejným počtem argumentů, nebo terminální uzel jiným terminálem.
- Vyzvedávající mutace - nahradí celý syntaktický strom některým z jeho podstromů.
- Smršťující mutace - nahrazuje náhodně zvolený podstrom jediným terminálem.

Někteří autoři se snaží experimentálně dokázat, že operátor mutace může být užitečnější. Např. v experimentu v [16] je porovnáván výpočet, kdy je jako hlavní operátor použit operátor mutace (bez použití operátoru křížení) s výpočtem, kdy je jako hlavní operátor použit operátor křížení (bez použití operátoru mutace). Z výsledků vyplývá, že úspěšnější je výpočet s použitím operátoru křížení, i když v některých případech je rozdíl velmi malý.

1.5.4 Doplňkové operátory

Doplňkové operátory se v genetickém programování používají jen někdy. V některých literaturách je považován za doplňkový operátor také operátor mutace. Podle [9] jsou nejčastěji používanými doplňkovými operátory:

- Operátor permutace - tento operátor je asexuální (pracuje pouze s jednou rodičovskou strukturou). Náhodně je zvolen vnitřní uzel odpovídající funkci s n argumenty a tyto argumenty jsou pak náhodně prohozeny. Pokud je tato náhodně vybraná funkce komutativní, potom je hodnota vrácená tímto operátorem stejná jako hodnota vstupující do tohoto operátoru. Příklad operace permutace je na Obr. 1.5.

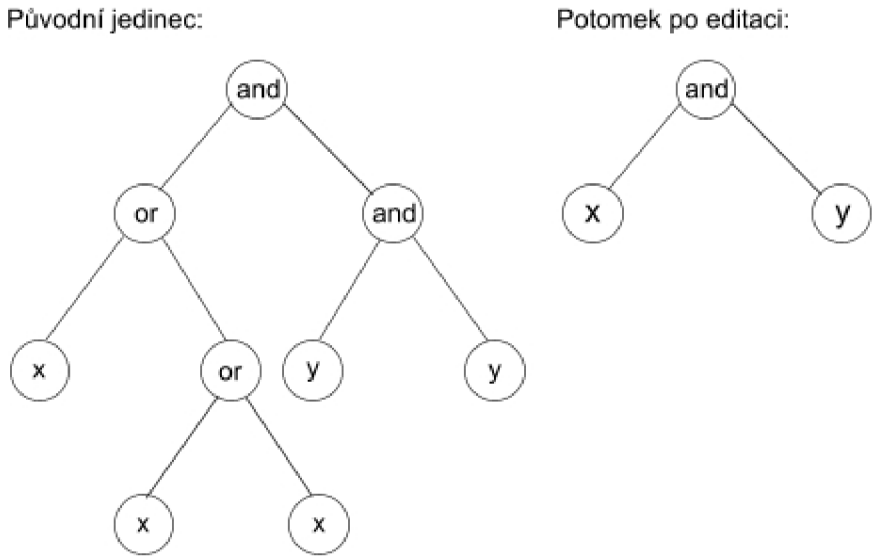


Obr. 1.5: Příklad operace permutace

- Operátor editace - i tento operátor je asexuální. Automaticky upravuje a zjednodušuje syntaktické stromy v průběhu jejich evoluce a tím pozitivně ovlivňuje celkový vývoj populace. Například pokud bude syntaktický strom obsahovat funkci and(Z,Z), je možné ji nahradit pouze jedním terminálem Z. Nevýhoda tohoto operátoru je, že pokud se tento operátor používá během evoluce příliš často, příliš redukuje rozmanitost populace a prodlužuje celkový čas běhu algoritmu. Proto se operátor editace spíše využívá pouze ke kosmetickým úpravám výstupu programu, aby byl výstup lépe čitelný pro uživatele. Více o tomto operátoru je možné najít v [29]. Příklad operace editace je na Obr. 1.6.

- Operátor zapouzdření - tento operátor je také asexuální. Umožňuje automatickou identifikaci potenciálně užitečného podstromu a jeho pojmenování tak, aby mohl být odkazován a používán opakovaně.[12] Tento operátor náhodně vybere podstrom a nahradí ho novou funkcí bez parametrů, která smazaný podstrom obsahuje. Tímto můžeme zabránit zničení podstromu křížením nebo mutací. Výhodou je zjednodušení programu a možnost opakovaného použití kódu. Více informací o tomto operátoru je možné nalézt v [15].

- Decimace - Tato operace má dva parametry: procento jedinců, kteří přežijí a podmínka, při jejíž splnění je operace vykonána. Pokud například první parametr říká, že má přežít 10 % jedinců, druhý parametr říká, že po třetí



Obr. 1.6: Příklad operace editace

generaci, potom po vytvoření populace a proběhnutí tří generací zbývá pro postup do další generace už jen 10 % původní populace. S výhodou se dá tento parametr využít u velkých počátečních populací, protože velké populace mají velké nároky na výpočetní čas. Takto se pomocí operace decimace odstraní jedinci obsahující nesmyslná řešení a zkrátí se celkový čas běhu programu.

1.6 Age-Layered population structure (ALPS)

Tento způsob zpracování algoritmu genetického programování je podobný Ostrovnímu modelu (viz. dále). U metody ALPS jsou jedinci rozděleni podle věku do vrstev zpracovávaných paralelně. Toto dělení brání předčasné konvergenci, protože si konkurují pouze jedinci s přibližně stejným věkem a mladí jedinci jsou chráněni před staršími, kteří měli více času vyvinout se do lokálního optima.

Pro každou vrstvu je stanoven maximální věk, který v ní mohou jedinci mít (vyjímkou je pouze poslední vrstva, kde mohou být jedinci s jakýmkoli věkem). Pokud jedinec dosáhne věku, který je pro aktuální vrstvu maximem, je porovnána jeho hodnota fitness funkce s nejhorším jedincem v následující vyšší vrstvě a pokud je jeho hodnota fitness funkce větší, nahradí ho. V opačném případě je z populace odstraněn. Maximální věk v jednotlivých vrstvách může být odstupňován lineárně, podle fibonacciho posloupnosti, polynomiálně nebo exponenciálně. V souvislosti s maximálním věkem se ještě zavádí parametr „věkový rozdíl“ (age-gap), který určuje maximální věk první vrstvy. Maximální věk dalších vrstev je jeho násobkem a jeho velikost závisí na použitém systému. Například pro polynomiální systém stupňování s věkovým

rozdílem 20 bude maximální věk první vrstvy 20, druhé vrstvy 40, třetí vrstvy 80, čtvrté vrstvy 180 atd.

Pravidla metody ALPS jsou následující. Na začátku je náhodně vygenerována populace jedinců v první vrstvě, kterým je přiřazen věk 0. Pokud jedinec v dané generaci vyprodukoval nějaké potomky nebo byl zkopírován do další generace například díky elitizmu, je jeho věk inkrementován o 1. Pokud se jedinec neúčastnil reprodukce ani nebyl zkopírován do další generace, zůstává jeho věk nezměněn. Potomkům, kteří vzniknou křížením nebo mutací je přiřazen věk 1 plus věk jejich nejstaršího rodiče. Jedinci se vyvíjejí společně s jedinci nacházejícími se ve stejné vrstvě i ve vrstvě předchozí.

Na začátku však nejsou aktivní všechny vrstvy, ale pouze první vrstva, ve které je náhodně vygenerována nová populace jedinců. Pro polynomiální systém odstupňování maximálního věku vrstev s věkovým rozdílem 20 je druhá vrstva aktivována až v 20. generaci. Třetí vrstva se začne používat až po 80. generaci, čtvrtá vrstva po 180. generaci atd. V pravidelných intervalech (v tomto případě každou 20. generaci) je v první vrstvě náhodně vygenerována nová populace s věkem nula, jejíž jedinci zcela nahradí jedince v této vrstvě.[7]

1.7 Ostrovní model

1.7.1 Úvod

Ostrovní model je způsob paralelního zpracování algoritmu genetického programování, kdy je počáteční populace rozdělena na subpopulace (kmeny) a každý kmen je zpracováván jednou stanicí nebo procesorem. Kmeny jsou během selekce, reprodukce a ohodnocování jedinců od sebe zcela izolovány. Ovlivňují se pouze tím, že si v určitých intervalech nebo nepravidelně posílají mezi sebou vybrané jedince (migrace). Protože se kmeny vyvíjejí odděleně, je možné v každém kmenu použít jiné selekční mechanismy a reprodukční operátory a tak snížit riziko, že algoritmus uvízne v lokálním optimu.

Můžeme například rozdělit kmeny do dvou skupin. Jedna skupina bude upřednostňovat jedince s vyšší hodnotou fitness funkce a tím bude udržovat velký selekční tlak, který povede rychleji k řešení (většinou však do lokálního optima). Druhá skupina bude používat selekční mechanismy, které více využívají náhody a operace mutace v nich bude probíhat častěji. Tím se zvýší rozmanitost jejich subpopulací ale za cenu toho, že tyto kmeny budou konvergovat k řešení mnohem pomaleji než předchozí skupina. Jedinci, migrující z kmenů této druhé skupiny, potom mohou dostat kmeny z první populace z lokálního optima a nasměrovat ke globálnímu optimu.[25]

1.7.2 Interval migrace

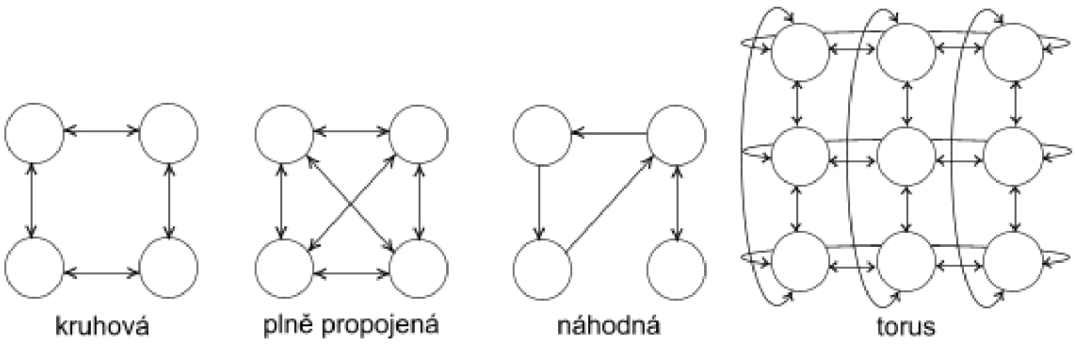
Migrace mezi kmeny může probíhat buď synchronně každou n-tou generaci (určuje se tzv. migrační interval) nebo asynchronně v nepravidelných intervalech. Při použití těchto modelů však může dojít k tomu, že budou dobří jedinci migrovat do subpopulace příliš brzy a ovlivní tak aktuální (možná správný) směr prohledávání. Proto existují ještě další modely, ve kterých dochází k migraci až po splnění určitého kritéria. Například v [2] je použit model, kdy se každý kmen vyvíjí izolovaně od ostatních a vždy když začne konvergovat k řešení, požádá sousední kmeny o zaslání vybraných jedinců. Z výsledků experimentů v [11] a v [28] vyplývá, že se zvyšováním frekvence migrace roste selekční tlak a algoritmus konverguje k řešení mnohem rychleji.

1.7.3 Migrační strategie

Jedinci k migraci mohou být vybírání na základě hodnoty fitness funkce, prostřednictvím turnaje nebo náhodně. V cílovém kmenu může migrující jedinec nahradit buď jedince s nejhorší hodnotou fitness funkce nebo náhodně vybraného jedince. Výsledky experimentů popsaných v [4] ukázaly, že pokud jsou migrující jedinec a jedinec, který má být nahrazen vybráni na základě fitness funkce (nejlepší nebo nejhorší v populaci), zvyšuje se selekční tlak v kmenu. Tento způsob výběru jedinců k migraci je podle [28] nejčastější.

1.7.4 Migrační topologie

Migrační topologie určuje, které kmeny si mezi sebou budou posílat jedince. Některé migrační topologie jsou znázorněny na obrázku 1.7. Další je možné nalézt v [25].



Obr. 1.7: Některé migrační topologie

1.7.5 Ostrovní model s reinicializací

Tato modifikace zavádí do ostrovního modelu pravidelnou reinicializaci některých subpopulací. Reinicializace zvýší rozmanitost jedinců v subpopulaci a v případě dostatečně velkého selekčního tlaku v ostatních kmenech pomůže efektivně nalézt globální optimum. V [25] jsou navrhovány dvě metody výběru kmenů, které budou znovu inicializovány.

V první metodě jsou v pravidelných intervalech vybíráni nejlepší jedinci z každého kmenu a porovnáváni. Pokud jsou někteří jedinci podobní, je subpopulace, ze které je jedinec s nižší hodnotou fitness, kompletně reinicializována.

U druhé metody je v pravidelných intervalech z každého kmene vybráno n jedinců s nejvyšší hodnotou fitness funkce. Pokud je některý kmen blízko optima a je jasné, že se již nebude dále výrazně měnit, migrují nejlepší jedinci do jiné subpopulace a tento kmen je reinicializován.[25]

Více informací o paralelních algoritmech a migraci lze najít v [3], [30] a v [6].

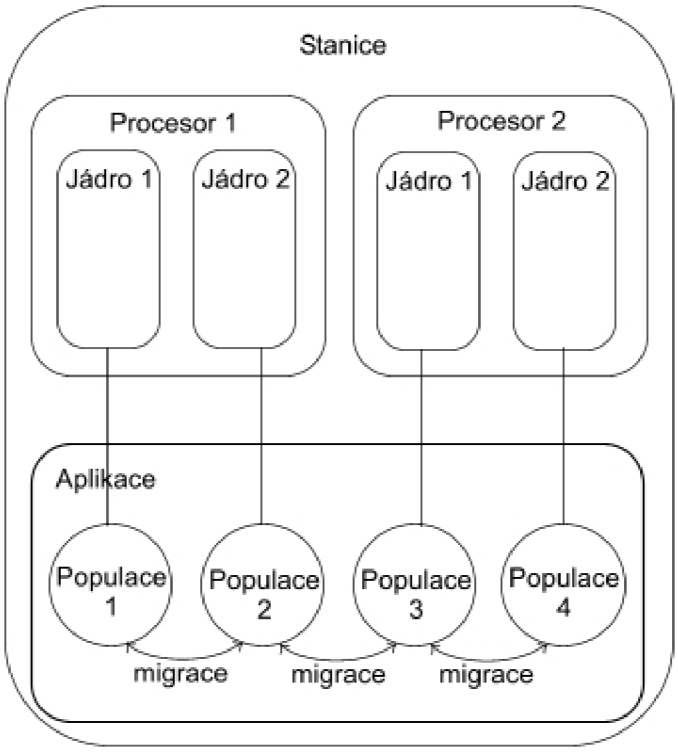
2 IMPLEMENTACE OSTROVNÍHO MODELU

2.1 Úvod

Každá stanice, na které bude ostrovní model spuštěn, obsahuje serverovou i klientskou část. Při spuštění výpočtů algoritmu genetického programování je spuštěna i serverová část, která naslouchá na nastaveném portu a zajišťuje začlenění jedinců přicházejících z ostatních stanic v síti do příslušné populace. V okamžiku, kdy má dojít k migraci jedince na jinou stanici v síti, je vytvořena klientská část, která zajistí odeslání jedince.

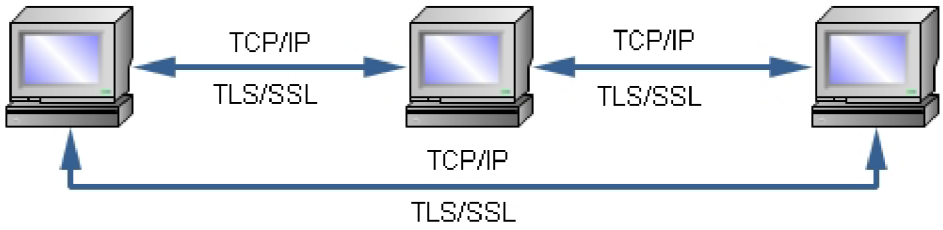
Protože na začátku výpočtů bude na každé stanici vytvořeno tolik populací, kolik má jader procesoru, bude během výpočtů docházet ke dvěma druhům migrace.

Při migraci mezi dvěma jádry procesoru téže stanice není potřeba vytvářet zabezpečené TCP/IP spojení. V tomto případě dojde pouze k překopírování migrujícího jedince ze zdrojové populace do příslušné vstupní fronty cílové populace. Při příští generaci jsou jedinci, čekající ve vstupní frontě, začleněni do cílové populace a podle nastavení nahrazují buď nejslabší nebo náhodně vybrané jedince. Tento typ migrace je znázorněn na Obr. 2.1



Obr. 2.1: Migrace mezi populacemi na téže stanici

Druhým typem migrace je migrace mezi populacemi nacházejícími se na různých stanicích v síti. U tohoto typu je vytvořeno protokolem TLS/SSL zabezpečené TCP/IP spojení a přes toto spojení jsou migrující jedinci přenášeni (Obr. 2.2). Serverová část spuštěná na cílové stanici potom, stejně jako u předchozího typu migrace, zařadí příchozí jedince do příslušné vstupní fronty a v následující generaci jsou jedinci zařazeni do cílové populace.



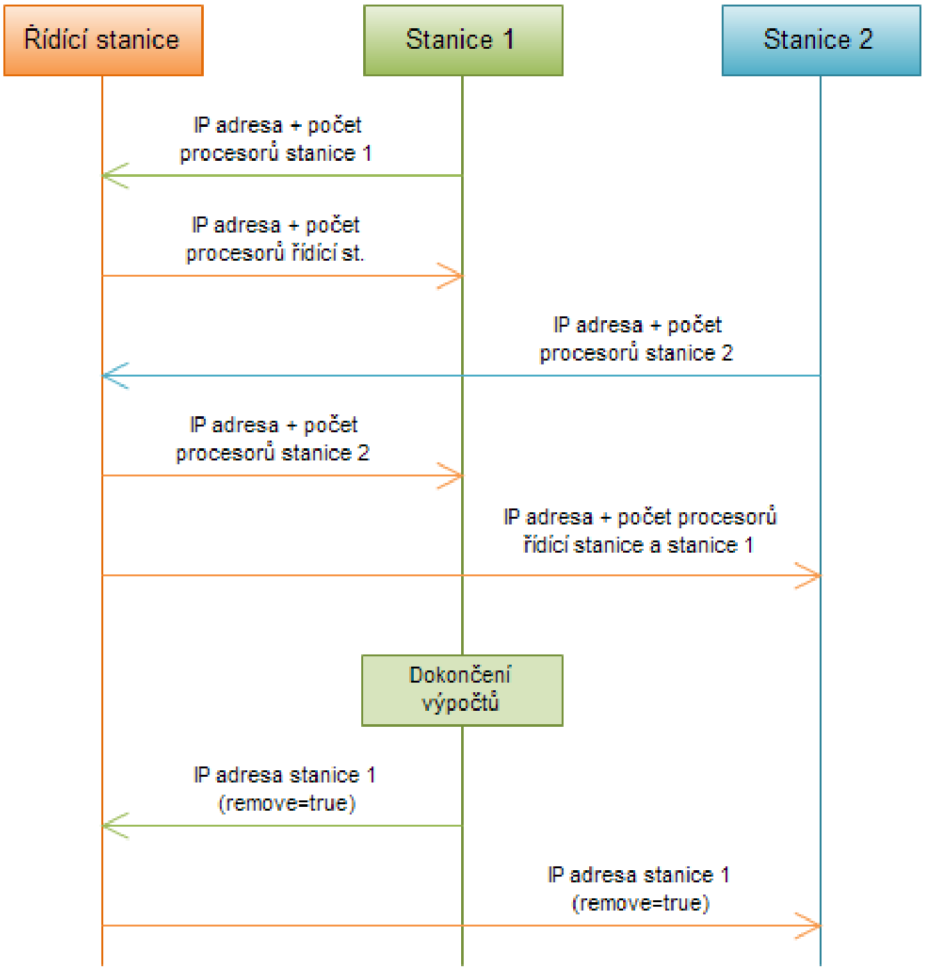
Obr. 2.2: Migrace mezi populacemi umístěnými na různých stanicích

Aby nebylo nutné do konfiguračního souboru na každé stanici zvlášť vkládat IP adresy a počty jader procesorů ostatních stanic v síti, je na jedné stanici spuštěna řídící část starající se o jejich automatické rozesílání. Pro správnou funkci musí být řídící stanice spuštěna jako první. Další stanice potom při spuštění výpočtů odešlou své údaje řídící stanici, která je rozešle všem ostatním stanicím (včetně lokálního předání vláknům, provádějících výpočet na řídící stanici) a aktuální stanici pošle údaje o všech stanicích, které se až dosud u ní takto zaregistrovaly. Po skončení výpočtů odešle stanice zprávu s proměnnou remove nastavenou na true, což znamená, že si jí má řídící stanice vymazat ze seznamu stanic provádějících výpočet a tuto informaci rozeslat všem ostatním stanicím, jenž si jí také odstraní ze seznamu stanic. Příklad této komunikace je znázorněn na obrázku 2.3.

UML diagram tříd celého ostrovního modelu je uveden v příloze A.3. Kvůli velikosti nejsou v diagramu uvedeny parametry jednotlivých metod.

2.2 Implementace jádra ostrovního modelu

Jádrem ostrovního modelu je třída Base, která se skládá ze dvou částí. První část je statická, aby k jejím proměnným a metodám mohly přistupovat i ostatní třídy. Tato část zajišťuje uchovávání údajů o ostatních stanicích účastnících se výpočtů a práci s nimi. Druhá část implementuje spuštění příslušného počtu vláken, které zpracovávají jednotlivé populace.



Obr. 2.3: Příklad komunikace mezi řídící stanicí a odstaními stanicemi

2.2.1 Statická část jádra

Každému vláknu je po jeho spuštění přiřazen automaticky jeho identifikátor. Protože by bylo složité získávat identifikátory vláken, které zpracovávají populace na ostatních stanicích, je každému vláknu ještě přiřazeno pořadové číslo (číslo procesoru), jenž je poté používáno při migraci k určení cílové populace. Převod čísla procesoru na identifikátor vlákna zajišťuje vektor Cores.

Další vektory AllStationsIp a AllStationsCores tvoří společně tabulku IP adres všech stanic provádějících výpočty (včetně sebe s adresou „localhost“) a čísel jejich procesorů. Tabulka 2.1 znázorňuje příklad, jak tato tabulka může vypadat. Z této tabulky jsou potom při migraci vybírány adresy, kam mají být migrující jedinci odesláni. Pro přidání nebo odstranění stanice z tabulky jsou zde metody AddStation a RemoveStation, které jsou volány serverovou částí po přijetí příslušných zpráv.

pořadí	IP adresa	číslo procesoru
1	localhost	1
2	localhost	2
3	192.168.1.5	1
4	192.168.1.4	1
5	192.168.1.4	2

Tab. 2.1: Příklad tabulky stanic a jejich procesorů

Konstruktor třídy Base vytváří nejprve instanci serverové části, která naslouchá na nastaveném portu, poté vloží do tabulky IP adres adresu localhost s čísly procesorů. Nakonec zjistí z konfiguračního souboru, jestli je tato stanice řídící a má se starat o správu a rozesílání informací o ostatních stanicích provádějících výpočet. Pokud tato stanice není stanicí řídící, odešle informace o své IP adrese a počtu jader procesorů řídící stanici, která se postará o rozeslání těchto informací ostatním stanicím (viz. kap. 2.5). Nejdůležitější část kódu konstruktoru je ve výpisu 1.

```
Base.cfg=new Config("config_default.ini");
server=new Server();
server.start(cfg.getServerPort());

for (int i=1;i<=Base.processorsCount;i++){
  AllStationsIp.add("localhost");
  AllStationsCores.add(i);
}

try{
  if (!cfg.getMasterStationAddress().equals(
  InetAddress.getLocalHost().getHostAddress().toString())){
    this.SendAddress();
  }
}catch (UnknownHostException e) {
  e.printStackTrace();
}
```

Výpis kódu 1: Kód konstruktoru třídy Base

2.2.2 Část paralelního zpracování

Paralelní zpracování zajišťuje metoda StartEvolutions, jejímž parametrem je instance třídy provádějící vlastní výpočet genetického programování. Tato metoda

spustí tolik paralelních vláken výpočtů, kolik má stanice jader procesorů. Každé vlákno zpracovává jednu populaci, která se vyvíjí odděleně od ostatních s občasnou migrací jedinců mezi populacemi na téže stanici i do ostatních stanic v síti. Využitím všech jader procesoru se sníží celkový čas výpočtu algoritmu genetického programování.

První řádek následujícího výpisu kódu (2) zjistí počet procesorů. Následující řádek vytvoří pole, do kterého se budou ukládat jednotlivá vlákna, aby k nim bylo možno později přistupovat a ovládat je.

```
int procCount=Runtime.getRuntime().availableProcessors();
CoreThread threads[]=new CoreThread[procCount];
```

Výpis kódu 2: Uložení identifikátorů jednotlivých vláken

Potom následuje smyčka, která spustí tolik paralelních vláken, kolik je jader procesoru. CoreThread je třída dědící od Thread, jejíž metoda run obsahuje kód, jenž má vlákno vykonat. V tomto případě metoda run uloží identifikátor vlákna do vektoru Cores ve statické části a zahájí vlastní výpočet. Další smyčka volá u všech vláken metodu join a tím říká, že má hlavní vlákno v tomto místě kódu počkat na dokončení výpočtů všech ostatních vláken (výpis kódu 3).

```
for (int j=0; j<procCount; j++){
  threads[j]=new CoreThread(evolutionStart);
  threads[j].start();
}

for (int j=0; j<procCount; j++){
  threads[j].join();
}
```

Výpis kódu 3: Spuštění jednotlivých vláken

Po skončení vlastního výpočtu metoda run odstraní své údaje ze seznamu IP adres ve statické části a pokud se již na stanici neprovádí žádné výpočty, je odeslána zpráva o ukončení výpočtu řídící stanici. V případě, že má v konfiguračním souboru položka exit_after_finding_one_solution hodnotu true je tato zpráva odeslána ihned po nalezení prvního řešení. Po odeslání zprávy je ukončeno TCP/IP spojení se serverem a přerušeny výpočty ostatních vláken na aktuální stanici.

2.3 Zabezpečená síťová komunikace

2.3.1 Výběr frameworku pro síťovou komunikaci

Na internetu jsou k dispozici ke stažení dva frameworky podporující zabezpečení přenášených dat pomocí SSL, které by se daly použít pro výměnu jedinců mezi stanicemi.

Prvním z nich byl Java Work, který je ke stažení z www.javawork.org. Na stránkách je k dispozici dokumentace a několik příkladů použití, ve kterých se mezi klientem a serverem přenášel objekt typu HashMap nebo prostý text. Pomocí standardního filtru typu codec se mi však nedařilo přenášet některé objekty a nikde jsem nenašel informace o tom, jestli je možné vytvořit vlastní codec filtr, který by objekt před přenosem převáděl na zprávu vhodnou k přenosu.

Druhým frameworkem, který jsem zkoušel, byl Mina framework od Apache Software Foundation stažený z mina.apache.org. Framework obsahuje také dokumentaci a vzorové příklady. Oproti Java Work je na stránkách mina.apache.org i několik tutoriálů, ze kterých je jasné, jak framework používat a jak si vytvořit vlastní codec filtr. Z tohoto důvodu jsem si pro implementaci ostrovního modelu vybral Mina framework.

2.3.2 Implementace serverové části

Před spuštěním serverové části, která přijímá data od klientů je nutné nastavit filtry, kterými budou procházet přijaté zprávy. Proto je definován nový objekt typu SocketAcceptorConfig a přidán do řetězce IO filtrů filtr pro šifrování a dešifrování SSL komunikace a filtr pro kódování a dekódování objektů, aby se daly objekty přenášet mezi stanicemi. Při vytváření objektu sslFilter je jako parametr předána instance třídy, která implementuje SSL zabezpečení. Kód této třídy je popsán v kapitole 2.3.4.

```
SocketAcceptorConfig cfg = new SocketAcceptorConfig();
try{
  SSLFilter sslFilter=new SSLFilter(BogusSSL.getInstance(true));
  cfg.getFilterChain().addLast("sslFilter", sslFilter);
}catch (Exception e1) {}

cfg.getFilterChain().addLast("logger",new LoggingFilter());
cfg.getFilterChain().addLast("codec",new ProtocolCodecFilter(new
ObjectSerializationCodecFactory()));
```

Výpis kódu 4: Nastavení filtrů serverové části

Pro přenos objektů je možné použít codec typu ObjectSerializationCodecFactory nebo si vytvořit vlastní filtr, který daný objekt převede na typ ByteBuffer a na druhé straně převede zpět. Mina framework také umožňuje definovat filtr typu LoggingFilter využívající knihovnu slf4j, který do okna loguje všechnu komunikaci. Nastavení filtrů je ukázáno ve výpise kódu 4.

Potom již stačí serverovou část spustit vytvořením objektu typu SocketAcceptor a zavoláním jeho metody bind, jejímiž parametry jsou: port, na kterém bude server naslouchat, objekt, který dědí od IoHandler a reaguje na události při komunikaci (např. na přijetí zprávy nebo na vytvoření spojení) a objekt typu SocketAcceptorConfig, ve kterém jsou nastaveny IO filtry (výpis 5).

```
IoAcceptor acceptor = new SocketAcceptor();
acceptor.bind(new InetSocketAddress(port),
  new ServerHandler(), cfg);
```

Výpis kódu 5: Spuštění serverové části

Třída ServerHandler, reagující na události při komunikaci, obsahuje metody messageReceived a sessionCreated. Při vytváření její instance je testováno, zda je aktuální stanice stanicí řídící. Pokud ano, je vytvořena instance třídy SendingAddresses, která zajišťuje rozesílání informací o stanicích účastnících se výpočtů.

Metoda messageReceived testuje, zda je přijatý objekt typu TransportOperator, který slouží k přenosu migrujících jedinců, nebo typu TransportAddress, v němž se přenášejí IP adresy a počty procesorů ostatních stanic účastnících se výpočtů. V případě objektu typu TransportOperator je jedinec zařazen do vstupní fronty příslušné populace. Pokud je přijatý objekt typu TransportAddress a stanice není stanicí řídící, je v závislosti na nastavení proměnné remove přijatá IP adresa buď vložena nebo odstraněna ze seznamu okolních stanic ve třídě Base (viz. kap. 2.5.3). Pokud je objekt typu TransportAddress přijat řídící stanicí, je přijatý objekt předán metodě processAddress třídy SendingAddresses, která ho rozešle ostatním stanicím.

```
public void sessionCreated(IoSession session){
  System.out.println("Spojení navázáno...");
  if (session.getTransportType() == TransportType.SOCKET)
    ((SocketSessionConfig)session.getConfig())
      .setReceiveBufferSize(2048);
}
```

Výpis kódu 6: Metoda sessionCreated

Metoda sessionCreated vypíše informace o tom, že bylo navázáno spojení a pokud je spojení typu TCP/IP nastavuje velikost bufferu na 2048 B. Obsah těchto dvou metod je ve výpisech kódu 6 a 7.

```
public void messageReceived(IoSession session, Object msg)
throws Exception{
  if (msg instanceof TransportOperator){
    MigrationAdapter.addToQueue(((TransportOperator)msg));
  }else if (msg instanceof TransportAddress){
    if (this.masterStation==false){
      if (((TransportAddress)msg).getRemove()==false)
        Base.AddStation(((TransportAddress)msg).getIpAddress(),
          ((TransportAddress)msg).getThreadCount());
      else
        Base.RemoveStation(((TransportAddress)msg)
          .getIpAddress().get(0));
    }else{
      if (((TransportAddress)msg).getRemove()==true)
        session.close();
      this.ipManagement.processAddress((TransportAddress)msg);
    }
  }
}
```

Výpis kódu 7: Metoda messageReceived

2.3.3 Implementace klientské části

Klientskou část, která zajišťuje TCP/IP spojení se serverovou částí jiné stanice, implementuje třída Client. Navazování spojení s každou stanicí je prováděno pouze jednou. Po navázání spojení je identifikátor relace uložen do proměnné OpenConnections typu HashMap, která mapuje IP adresy na identifikátory relací. Po dokončení výpočtů jsou potom stanice z proměnné OpenConnections odstraněny.

Při vytváření instance třídy Client její konstruktor testuje, zda již nebylo spojení s požadovanou stanicí navázáno. Pokud ano, je do privátní proměnné session uložen identifikátor relace. Pokud ne, je zavolána metoda createConnection, která spojení se stanicí naváže a uloží identifikátor relace jak do proměnné session, tak i do proměnné OpenConnections.

Parametrem metody createConnection je IP adresa stanice, s niž má být spojení navázáno (proměnná string_address). Podobně jako v serverové části, je i v klientské části nutné nastavit SSL filtr. Rozdíl je pouze v tom, že je zde potřeba nastavit proměnnou UseClientMode na true, aby byl při potvrzování použit mód klienta. Potom je již možné zavolat metodu connect objektu SocketConnector a předat mu jako parametr adresu a port serveru, objekt reagující na události při komunikaci

a objekt typu SocketConnectorConfig, ve kterém jsou nastaveny IO filtry. Obsah metody createConnection je znázorněn ve výpise 8.

```
SocketConnector connector=new SocketConnector();
SocketConnectorConfig config = new SocketConnectorConfig();

SSLContext sslContext = BogusSSL.getInstance(false);
SSLFilter sslFilter = new SSLFilter(sslContext);
sslFilter.setUseClientMode(true);
config.getFilterChain().addLast("sslFilter", sslFilter);

InetSocketAddress address=new InetSocketAddress(string_address,
  this.cfg.getServerPort());
ConnectFuture future = connector.connect(address,
  new ClientHandler(), config);
future.join();
session = future.getSession();
OpenConnections.put(string_address, this.session);
```

Výpis kódu 8: Metoda createConnection

Posílat objekty potom můžeme tak, že zavoláme session.write(objekt_k_poslani). Tímto příkazem je tento objekt poslán přes codec filtr a SSL filtr na definovaný port (nastavený v konfiguračním souboru serverové části) na adrese, jenž je určena proměnnou string_address.

Ve třídě ClientHandler jsou definovány metody sessionCreated a messageSent. Metoda sessionCreated přidává do řetězce filtrů codec filtr a logovací filtr. Obsah této metody je ve výpisu kódu 9.

```
private static IoFilter LOGGING_FILTER = new LoggingFilter();
private static IoFilter CODEC_FILTER = new ProtocolCodecFilter(
new ObjectSerializationCodecFactory());

public void sessionCreated(IoSession session) throws Exception {
  session.getFilterChain().addLast("logger", LOGGING_FILTER);
  session.getFilterChain().addLast("codec", CODEC_FILTER);
}
```

Výpis kódu 9: Metoda sessionCreated třídy ClientHandler

Druhou metodou je metoda messageSent, která kontroluje, zda byla odeslána zpráva řídící stanici informující jí, že tato stanice již ukončila výpočty. Pokud byla tato zpráva odeslána, je zavolána metoda session.getCloseFuture().join(), která počká až

bude spojení ukončeno ze strany serveru a potom ho ukončí také metodou session.close().

2.3.4 Implementace zabezpečení

Pro zajištění integrity dat, autentizaci serveru a důvěrnosti dat je použit protokol TLS (Transport Layer Security). Před vlastním přenosem zabezpečených dat musí proběhnout autentizace serveru (tzv. podání ruky). TLS pro autentizaci serveru využívá algoritmus RSA a jeho veřejné a soukromé klíče.

Proces autentizace serverové části probíhá následovně. Klient spustí proces podání ruky zasláním inicializační zprávy. Na tuto zprávu server odpoví zasláním svého certifikátu, který obsahuje jeho veřejný klíč, klientovi. Klient ověří identitu serveru a z přijatého certifikátu získá veřejný klíč serveru. Vygeneruje klíč aktuální relace, kterým se budou symetricky šifrovat přenášená data. Tento klíč zašifruje získaným veřejným klíčem a odešle serveru. Server pomocí svého soukromého klíče dešifruje klíč relace. Klíčem relace je následně šifrována všechna komunikace mezi serverem a klientem. Během přenosu dat může server nebo klient zažádat o znovuustavení spojení. V tomto případě se zopakuje proces podání rukou.[24]

Třída BogusSSL, která je použita při vytváření SSL kontextu v serverové i klientské části, obsahuje tyto tři metody:[1]

- getInstance(boolean server)
- createBogusServerSSLContext()
- createBogusClientSSLContext().

První z nich - metoda getInstance s parametrem, který říká, jestli jde o server nebo klienta, testuje, zda již existuje SSL kontext pro daného klienta nebo server. Pokud ještě SSL kontext neexistuje, zavolá se metoda createBogusServerSSLContext() pro server nebo createBogusClientSSLContext() pro klienta, která kontext vytvoří.

Metoda createBogusServerSSLContext() nejdříve vytvoří nové uložiště pro kryptografické klíče a certifikáty. Každý klíč v uložišti je chráněn vlastním heslem a druhým heslem (může být stejné) je chráněna integrita celého uložiště. Na výběr je několik implementací tohoto uložiště. Výchozí je implementace s názvem JKS. Druhou je implementace JCEKS využívající k zabezpečení silnější algoritmus Triple DES. V manuálu pro verzi Java SE 6 však není doporučována, protože jí základní verze Java SE 6 nepodporuje. Třetím typem je pkcs12, který využívá algoritmus RSA a je primárně určený pro uchovávání a transport soukromých klíčů a certifikátů. Jeho standart však ve verzi Java SE 6 ještě není ustálen a v [10] je uvedeno,

že by měla být použita jiná implementace než tato. Proto jsem v mém programu použil výchozí implementaci JKS (výpis 10).

```
KeyStore ks = KeyStore.getInstance("jks");
```

Výpis kódu 10: Vytvoření uložiště klíčů

Dále metoda načte certifikát pomocí metody getResourceAsStream a metodou load ho vloží do uložiště. Druhým parametrem metody load je pole znaků reprezentující heslo k otevření certifikátu. V kódu 11 je ukázáno použití obou metod.

```
char[] KEYPASS = cfg.getKeyPass().toCharArray();
char[] STOREPASS = cfg.getStorePass().toCharArray();
InputStream in = null;
in = BogusSSL.class.getResourceAsStream("bogus.cert");
ks.load(in, STOREPASS);
```

Výpis kódu 11: Načtení certifikátu

Potom je vytvořen nový „správce klíčů“ (key manager), je inicializován a je mu předáno vytvořené uložiště a heslo, které chrání privátní klíč. Nakonec je vytvořen SSL kontext, který je inicializován těmito třemi parametry: vytvořeným správcem klíčů, objektem typu TrustManager, který spravuje klíče a certifikáty a algoritmem generátoru pseudo náhodných čísel (pokud necháme null, bude použit výchozí algoritmus SHA1PRNG, který pomocí SHA-1 počítá hash z náhodné počáteční hodnoty, kterou pro každou operaci 64-bitový čítač inkrementuje o jedničku).

```
KeyManagerFactory kmf = KeyManagerFactory.getInstance("SunX509");
kmf.init(ks, KEYPASS);
SSLContext sslContext = SSLContext.getInstance("TLS");
sslContext.init(kmf.getKeyManagers(),
  BogusTrustManager.X509_MANAGERS, null);
```

Výpis kódu 12: Vytvoření správce klíčů

Metoda createBogusClientSSLContext() je značně jednodušší než ta, která vytváří SSL kontext pro server. U klienta je možné hned vytvořit nový SSL kontext a ten inicializovat předáním objektu TrustManager (viz. výpis 13).

```
SSLContext context = SSLContext.getInstance("TLS");
context.init(null, BogusTrustManager.X509_MANAGERS, null);
```

Výpis kódu 13: Vytvoření SSL kontextu klienta

2.3.5 Vygenerování certifikátu

Pro vygenerování certifikátu je možné použít program keytool, který je součástí sady nástrojů Java Development Kit a Java Runtime Environment a slouží ke správě veřejných a soukromých klíčů a certifikátů pro autentizaci. Keytool se spouští z příkazové řádky a pro vygenerování veřejného a soukromého klíče může mít tyto parametry (všechny parametry jsou uvozovány znaménkem mínus):

- genkeypair - vygeneruje veřejný a soukromý klíč, vloží veřejný klíč do certifikátu standartu X.509 a tento certifikát spolu se soukromým klíčem vloží jako novou položku do uložiště klíčů
- alias - slouží k identifikaci v uložišti klíčů
- keyalg - algoritmus, který bude použit k vygenerování klíčů např. DES, DSA, RSA...
- keysize - velikost vygenerovaného klíče v bitech
- validity - délka platnosti certifikátu (počet dnů)
- keystore - název vygenerovaného certifikátu
- storetype - typ uložiště např. JKS, pkcs12...
- storepass - heslo k ochraně integrity uložiště, minimální délka je 6 znaků
- keypass - heslo k ochraně privátního klíče, minimální délka je také 6 znaků
- dname - za tímto příkazem následují v uvozovkách tyto parametry: CN= obecné označení, OU=útvar, O=organizace, L=místo, S=stát, C=kód země.

Vytvoření certifikátu:

```
keytool -genkeypair -alias myKey -keysize 512 -validity 365
-keyalg RSA -dname "CN=genetic, OU=UTKO CA, O=FEKT, L=Brno,
S=Czech republic, C=CZ" -keypass password -storepass password
-keystore bogus.cert
```

2.4 Implementace migrace mezi populacemi

Základní třídou implementující migraci je abstraktní třída MigrationAdapter. Od ní jsou potom odvozeny třídy MigrationAccordingToFitness a RandomMigration. První z odvozených tříd implementuje migraci jedinců, kteří jsou vybírání podle jejich hodnoty fitness funkce. Druhá odvozená třída vybírá jedince k migraci náhodně.

2.4.1 Abstraktní třída MigrationAdapter

Třída MigrationAdapter obsahuje vektor GenerationsCounts a HashMap WaitingOperators. Jejími hlavními metodami jsou: migrationIn, migrationOut, migrationOverTCP, localMigration, migrationReplaced, incrementGenerationsCount, addToQueue a returnAndCleanQueue.

Ve vektoru GenerationsCounts jsou uloženy a každou generaci aktualizovány počty generací, které proběhly v jednotlivých populacích. Aktualizaci počtu generací v tomto vektoru provádí metoda incrementGenerationsCount, jež je volána každou generaci metodou migrationOut. Údaje v tomto vektoru jsou potřeba, pokud je v konfiguračním souboru nastaven migrační práh (jedinci začnou migrovat až po nastaveném počtu generací) nebo migrační interval.

HashMap s názvem WaitingOperators mapuje identifikátory vláken na vektory, které slouží jako vstupní fronty jednotlivých populací. Pro přidávání migrujících jedinců do fronty je zde metoda addToQueue, která je volána serverovou částí při přijetí migrujícího jedince. Jedinec je do fronty vkládán v objektu třídy TransportOperator, aby bylo možné v případě prohození jedinců určit IP adresu stanice, která jedince odeslala. Pro vrácení fronty určité populace a její následné smazání slouží metoda returnAndCleanQueue, jež je volána metodou migrationIn. Tyto dvě metody jsou uvedeny ve výpisu 14 a 15.

```
public static void addToQueue(TransportOperator o){
  if (Base.getCores().get(o.getTargetThreadNumber()-1) != null){
    int coreNum=o.getTargetThreadNumber()-1;
    Long threadId=Base.getCores().get(coreNum);
    synchronized(WaitingOperators){
      if (!WaitingOperators.containsKey(threadId)){
        Vector<TransportOperator> individuals=
          new Vector<TransportOperator>();
        individuals.add(o);
        WaitingOperators.put(threadId, individuals);
      }else
        WaitingOperators.get(threadId).add(o);
    }
  }
}
```

Výpis kódu 14: Metoda addToQueue

Metoda migrationOut se stará o výběr stanic, do kterých v dané generaci proběhne migrace. Má jediný parametr a tím je populace, ze které budou jedinci migrovat. Pokud již byl překročen migrační práh a od poslední migrace uběhl migrační

interval nastavený v konfiguračním souboru (viz. kap. 2.6), testuje se pro každou stanici a její populaci, zda v aktuální generaci dojde k migraci (porovnává se náhodně vygenerované číslo s číslem udávajícím pravděpodobnost migrace). Pokud má dojít k migraci, je zavolána metoda migrate, které je předána populace a číslo populace, do níž mají jedinci migrovat. Tato metoda zajišťuje výběr jedinců k migraci a je implementována v odvozených třídách. Ve výpisu 16 se nachází obsah metody migrationOut.

```
public Vector<TransportOperator> returnAndCleanQueue(Long queue){
  Vector<TransportOperator> tmp;
  synchronized(WaitingOperators){
    tmp=WaitingOperators.get(queue);
    WaitingOperators.remove(queue);
  }
  return tmp;
}
```

Výpis kódu 15: Metoda returnAndCleanQueue

```
public void migrationOut(Population p){
  int gN=Base.getCores().indexOf(Thread.currentThread().getId());
  int populationNumber=incrementGenerationsCount(genNumber);
  if (populationNumber>cfg.getMigrationThreshold()
  && populationNumber%cfg.getMigrationInterval()==0){
    Random rnd = new Random();
    for (int i=0;i<Base.getAllStationsIp().size();i++){
      if (!(gN+1==Base.getAllStationsCores().get(i)
      && Base.getAllStationsIp().get(i).equals("localhost"))){
        if (cfg.getMigrationRate() > rnd.nextDouble())
          migrate(p,i);
      }
    }
  }
}
```

Výpis kódu 16: Metoda migrationOut

Podle toho, kde se cílová populace nachází, je volána buď metoda migrationOverTCP, která naváže spojení s cílovou stanicí pomocí třídy Client a odešle migrujícího jedince nebo metoda localMigration, která pomocí metody addToQueue provede překopírování migrujícího jedince do vstupní fronty cílové populace.

Metoda migrationIn (výpis 17) pomocí metody returnAndCleanQueue zkopíruje jedince ze vstupní fronty příslušné populace, následně frontu vymaže a poté za-

volá metodu migrationInCopy, kterou implementují odvozené třídy a která zajišťuje začlenění migrujícího jedince do cílové populace.

```
public Population migrationIn(Population p){
  Long currentThread=Thread.currentThread().getId();
  if (WaitingOperators.containsKey(currentThread)){
    Vector<TransportOperator> tmp;
    tmp=returnAndCleanQueue(currentThread);
    p=migrationInCopy(tmp,p);
  }
  return p;
}
```

Výpis kódu 17: Metoda migrationIn

Poslední metodou je metoda migrationReplaced. Pokud je v objektu typu TransportOperator, který zajišťuje přenos migrujících jedinců, nastavena proměnná exchange na true (viz. kap. 2.4.4), je jedinec, který má být nahrazen příchozím migrujícím jedincem, odeslán do populace, která migrujícího jedince poslala.

2.4.2 Třída MigrationAccordingToFitness

```
protected void migrate(Population p, int StationNumber){
  int j;
  for (j=0;(j<this.cfg.getMigrationSize() && j<p.size());j++){
    try{
      if (Base.getAllStationsIp().get(StationNumber)
      .equals("localhost"))
        localMigration(new TransportOperator(p.getIndividual(j),
        Base.getAllStationsCores().get(StationNumber),
        exchange()));
      else
        migrationOverTCP(new TransportOperator(
        p.getIndividual(j),Base.getAllStationsCores().
        get(StationNumber),exchange()),
        Base.getAllStationsIp().get(StationNumber));
    }catch(ArrayIndexOutOfBoundsException e){}
  }
}
```

Výpis kódu 18: Metoda migrate třídy MigrationAccordingToFitness

Metoda migrate třídy MigrationAccordingToFitness, jejíž obsah je ve výpisu 18, vezme ze vstupní populace n jedinců (velikost n je určena parametrem migration_size

v konfiguračním souboru) a podle umístění cílové populace zavolá buď metodu migrationOverTCP nebo localMigration, které zajistí zkopírování migrujícího jedince do cílové populace. Při volání metod migrationOverTCP a localMigration je zachytávána výjimka ArrayIndexOutOfBoundsException, protože během provádění migrace mohla být stanice odstraněna ze seznamu okolních stanic z důvodu dokončení výpočtů.

Metoda migrationInCopy odstraní jedince s nejhorší hodnotou fitness funkce a nahradí ho migrujícím jedincem. Pokud je u migrujícícho jedince nastavena proměnná exchange na true, je zavolána metoda migrationReplaced, která zajistí odeslání jedince, který má být nahrazen. Obsah metody migrationInCopy je ve výpisu 19.

```
protected Population migrationInCopy(
Vector<TransportOperator> tmp, Population p){
  Population tmpPopulation=new Population();
  for (TransportOperator o : tmp) {
    if (p.size()>0){
      this.migrationReplaced(o, p.getIndividual(p.size()-1));
      p.remove(p.size()-1);
      tmpPopulation.add(o.getOperator());
    }
  }
  p.add(tmpPopulation);
  return p;
}
```

Výpis kódu 19: Metoda migrationInCopy třídy MigrationAccordingToFitness

2.4.3 Třída RandomMigration

Metoda migrate (výpis kódu 20) třídy RandomMigration je o něco složitější než u třídy MigrationAccordingToFitness. Migrující jedinci se ze vstupní populace vybírají náhodně. Do vektoru alreadySent jsou ukládáni jedinci, kteří již byli v aktuální generaci odesláni, aby nemohlo dojít k tomu, že bude jeden jedinec poslán do jedné populace vícekrát.

```
protected void migrate(Population p, int StationNumber){
  Random rnd=new Random();
  Vector<Operator> alreadySent=new Vector<Operator>();
  int j;
  for (j=0;(j<cfg.getMigrationSize() && j<p.size());j++){
    int pos=rnd.nextInt(p.size()-1);
    if (!alreadySent.contains(p.getIndividual(pos))){
      try{
        if (Base.getAllStationsIp().get(StationNumber)
        .equals("localhost"))
          localMigration(new TransportOperator(
            p.getIndividual(pos),Base.getAllStationsCores()
            .get(StationNumber),exchange()));
        else
          migrationOverTCP(new TransportOperator(
            p.getIndividual(pos),Base.getAllStationsCores()
            .get(StationNumber),exchange()),
            Base.getAllStationsIp().get(StationNumber));
          alreadySent.add(p.getIndividual(pos));
      }catch(ArrayIndexOutOfBoundsException e){}
    }else j--;
  }
}
```

Výpis kódu 20: Metoda migrate třídy RandomMigration

Metoda migrationInCopy funguje podobně jako u třídy MigrationAccordingToFitness jen s tím rozdílem, že migrující jedinec nahrazuje náhodně vybraného jedince. Kód této metody je ve výpisu 21.

2.4.4 Přenos migrujících jedinců

Jedinci jsou přenášeni v objektu třídy TransportOperator, jehož parametry jsou: migrující jedinec, číslo procesoru na cílové stanici a logická hodnota, zda má jedinec, který bude v cílové populaci nahrazen, migrovat do aktuální populace (viz. kap. 2.6). Třída obsahuje následující privátní proměnné:

- sendingOperator - migrující jedinec
- targetThreadNumber - číslo procesoru zpracovávající cílovou populaci

- exchange - logická hodnota zda má jedinec, který bude v cílové populaci nahrazen migrovat do aktuální populace; následující dvě proměnné jsou nastaveny konstruktorem pouze v případě, že hodnota této proměnné je true
- sourceThreadNumber - číslo procesoru zpracovávající aktuální populaci
- sourceIPAddress - IP adresa stanice, která odeslala jedince

```
protected Population migrationInCopy(
Vector<TransportOperator> tmp, Population p){
  Population tmpPopulation=new Population();
  Random rnd=new Random();
  for (TransportOperator o : tmp) {
    if (p.size()>0){
      int pos=rnd.nextInt(p.size()-1);
      migrationReplaced(o, p.getIndividual(pos));
      p.remove(pos);
      tmpPopulation.add(o.getOperator());
    }
  }
  p.add(tmpPopulation);
  return p;
}
```

Výpis kódu 21: Metoda migrationInCopy třídy RandomMigration

2.5 Automatické rozesílání informací o stanicích

2.5.1 Úvod

Bez automatického rozesílání informací o IP adresách a počtu jader procesorů ostatních stanic provádějících výpočet by bylo nutné do konfiguračního souboru na každé stanici ručně vypsat IP adresy a počty procesorů všech ostatních stanic. Při velkém počtu stanic by to bylo velmi složité. Navíc by se potom nedal tento ostrovní model spustit ve výpočetním gridu, protože tam není předem určeno, jakou IP adresu budou mít jednotlivé stanice, na kterých se budou provádět výpočty.

V konfiguračním souboru se nachází položka main_station, která obsahuje IP adresu stanice, na níž bude spuštěna část zajišťující rozesílání informací o stanicích v síti. Výpočet na této stanici musí být spuštěn jako první. Každá další stanice po spuštění výpočtu vloží svoji IP adresu a číslo udávající počet jader procesoru do objektu třídy TransportAddress (viz. kap. 2.5.3) a odešle řídící stanici, která

si uchovává tabulku s IP adresami a počty procesorů všech stanic provádějících výpočty. Řídící stanice přepošle příchozí informace všem ostatním stanicím z tabulky a nové stanici pošle informace o všech stanicích v tabulce. Nakonec si do tabulky přidá i tuto novou stanici.

Dokončí-li stanice výpočet, odešle řídící stanici zprávu s její IP adresou a s proměnnou remove nastavenou na true. Pokud má položka exit_after_finding_one_solution v konfiguračním souboru hodnotu true stačí, aby výpočet dokončilo pouze jedno vlákno. V opačném případě je zpráva s nastavenou proměnnou remove odeslána až po dokončení výpočtů všech vláken. Po přijetí této zprávy řídící stanicí je zpráva opět přeposlána ostatním stanicím a odstraněna z tabulky stanic na řídící stanici i na všech ostatních stanicích.

V případě, že je položka exit_after_finding_one_solution nastavena na true a podřízená stanice obdrží zprávu, v niž je nastavena proměnná remove na true, automaticky ukončí spojení s řídící stanicí a přeruší všechny své výpočty.

2.5.2 Implementace automatického rozesílání informací

Rozesílání informací o ostatních stanicích zajišťuje třída SendingAddresses, která obsahuje metody: processAddress, sendBroadcast, sendAddress a removeAddress. První z nich, metoda processAddress (její kód je ve výpisu 22), je volána serverovou částí po přijetí zprávy typu TransportAddress. Přijetí této zprávy řídící stanicí může znamenat buď to, že se do výpočtů zapojila nová stanice nebo že některá stanice ukončila výpočty. Proto se na začátku metody processAddress testuje, zda je nastavena proměnná remove na true.

```
public void processAddress(TransportAddress taddress){
  if (taddress.getRemove()==false){
    sendAddress(this.ipAddress,this.threadCount,
      taddress.getIpAddress().get(0),false);
    sendBroadcast(taddress);
    this.ipAddress.add(taddress.getIpAddress().get(0));
    this.threadCount.add(taddress.getThreadCount().get(0));
  }else{
    removeAddress(taddress);
  }
}
```

Výpis kódu 22: Metoda processAddress

Pokud je proměnná remove nastavena na false, znamená to, že se jedná o novou stanici. V tomto případě je zavolána metoda sendAddress, která pomocí třídy Client odešle informace o všech stanicích v síti této nové stanici. Poté je zavolána

metoda sendBroadcast (výpis 23), jenž projde tabulku stanic a každé z nich přepošle informace o nové stanici. Protože probíhají výpočty také na řídící stanici, musí se testovat, zda zdrojová a cílová stanice nejsou totožné. Pokud ano, dojde pouze k překopírování pomocí metody AddStation třídy Base. Ostatním stanicím se informace o nové stanici přeposílají, stejně jako nové stanici, pomocí metody sendAddress. Nakonec si řídící stanicí tuto novou stanici uloží do své tabulky IP adres.

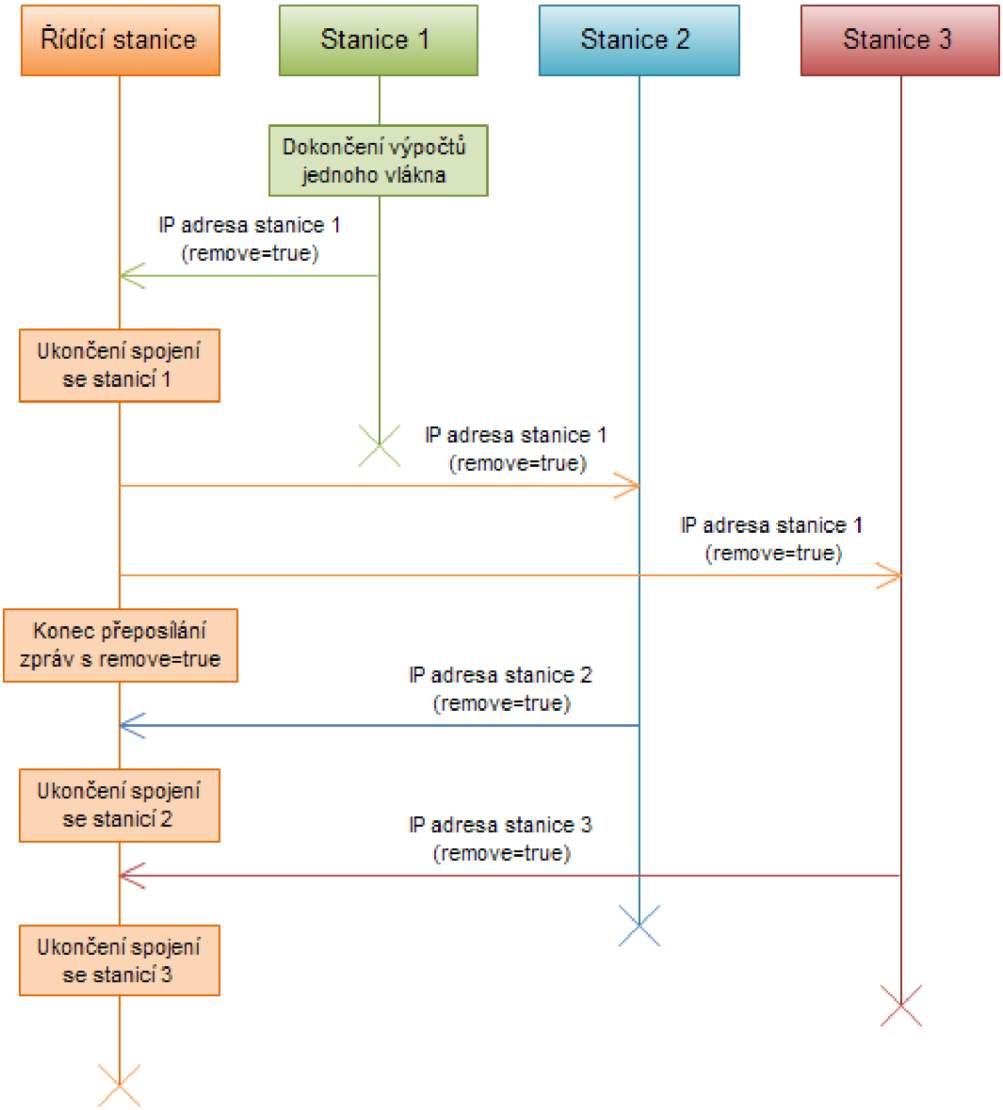
```
private void sendBroadcast(TransportAddress taddress){
  for (int i=0; i<this.ipAddress.size(); i++){
    if (!this.thisIP.equals(this.ipAddress.get(i))){
      this.sendAddress(taddress.getIpAddress(),
        taddress.getThreadCount(),this.ipAddress.get(i),
        taddress.getRemove());
    }else{
      if (taddress.getRemove()==false)
        Base.AddStation(taddress.getIpAddress(),
          taddress.getThreadCount());
      else
        Base.RemoveStation(taddress.getIpAddress().get(0));
    }
  }
}
```

Výpis kódu 23: Metoda sendBroadcast

V opačném případě (když je proměnná remove nastavena na true) je zavolána metoda removeAddress, která zdrojovou adresu odstraní z tabulky IP adres. Poté je opět zavolána již známá metoda sendBroadcast. Ta funguje podobně jako v předchozím případě, pouze s tím rozdílem, že pokud je zdrojová a cílová adresa stejná, nevolá se metoda AddStation, ale metoda RemoveStation třídy Base.

Pokud má proměnná exitAfterFindingSolution, která obsahuje hodnotu položky exit_after_finding_one_solution v konfiguračním souboru, hodnotu true, je zavolána metoda sendBroadcast pouze při prvním přijetí zprávy s proměnnou remove nastavenou na true. Zprávy s proměnnou remove=true, které potom následují, pouze informují řídící stanici, že stanice, která zprávu odeslala ukončuje výpočty a řídící stanice s ní může ukončit TCP/IP spojení a vymazat ji ze svého seznamu stanic provádějících výpočty. Průběh komunikace řídící stanice s ostatními stanicemi po dokončení výpočtů jednoho vlákna je znázorněn na obrázku 2.4.

Na konci metody removeAddress je kontrolováno, jestli ještě nějaká stanice provádí výpočty. V případě, že už je na všech stanicích výpočet ukončen, je ukončen běh aplikace i na řídící stanici. Kód metody removeAddress je ve výpisu 24.



Obr. 2.4: Komunikace stanic po dokončení výpočtů jednoho vlákna s nastavením exit_after_finding_one_solution=true

2.5.3 Přenos řídících informací

Pro přenos IP adres a počtu jader procesorů mezi řídící stanicí a ostatními stanicemi se používá objekt třídy TransportAddress. Třída obsahuje následující privátní metody:

- ipAddress - vektor IP adres
- threadCount - vektor počtu jader procesorů
- remove - logická hodnota určující, zda mají být adresy ve vektoru ipAddress přidány do tabulky okolních stanic nebo z ní odebrány

Pokud je objekt odeslán obyčejnou stanicí při zahájení nebo ukončení výpočtů, obsahuje vektor ipAddress vždy pouze jednu IP adresu a vektor threadCount pouze jedno číslo udávající počet jader procesorů. Více položek ve vektorech posílá pouze řídící stanice nové stanici po obdržení jejích údajů.

```
private void removeAddress(TransportAddress taddress){
  synchronized(this.ipAddress){
    synchronized(this.threadCount){
      int pos=ipAddress.indexOf(taddress.getIpAddress().get(0));
      if (pos!=-1){
        this.ipAddress.remove(pos);
        this.threadCount.remove(pos);
      }
    }
  }
  if (exitIslandModel==false){
    sendBroadcast(taddress);
    if (exitAfterFindingSolution==true)
      exitIslandModel=true;
  }
  if (this.ipAddress.size()==0 && this.threadCount.size()==0)
    System.exit(0);
}
```

Výpis kódu 24: Metoda removeAddress

2.6 Konfigurace ostrovního modelu

Parametry ostrovního modelu je možné nastavit v INI souboru config_default.ini, který obsahuje dvojice parametr=hodnota. Parametry jsou rozděleny do dvou sekcí. V první sekci jsou tyto parametry týkající se vlastní migrace:

- migration_rate - hodnota pravděpodobnosti, která určuje, zda v aktuální generaci dojde pro konkrétní cílový kmen k migraci jedinců. Pokud je tento parametr nastaven na 1, bude migrace záviset pouze na parametru migration_interval popsaném níže.
- migration_size - pokud na základě pravděpodobnosti určené předchozím parametrem dojde k migraci, tak tento parametr určuje počet jedinců, kteří budou z aktuálního kmene vybráni a budou migrovat do cílového kmene

- migration_exchange_rate - hodnota pravděpodobnosti určující, zda při migraci dojde k prohození migrujícího jedince s jedincem, který má být nahrazen (př: migrující jedinec a nahradí v cílové populaci náhodně vybraného jedince b a ten migruje do zdrojové populace jedince a)

- random_selection_out - pokud je tento parametr roven true, migrující jedinci se vybírají náhodně; v případě hodnoty false se migrující jedinci vybírají podle hodnoty fitness funkce (jsou vybráni ti s nejvyšší hodnotou fitness funkce)

- random_selection_replace - pokud je parametr roven true, jedinci v cílovém kmenu, kteří mají být nahrazeni, jsou vybíráni náhodně; v opačném případě jsou vybíráni ti s nejhorším ohodnocením

- migration_threshold - počet generací před zahájením migrace

- migration_interval - specifikuje, jak často se bude pokoušet kmen o migraci (jestli proběhne migrace potom také závisí na parametru migration_rate). Pokud je tento parametr například nastaven na 10, bude se algoritmus pokoušet o migraci každou desátou generaci. V případě, že ho nastavíme na 1, bude se algoritmus pokoušet o migraci každou generaci a migrace tak bude záviset pouze na parametru migration_rate.

Ve druhé sekci je pouze parametr exit_after_finding_one_solution, který určuje, zda má být celý výpočet ostrovním modelem ukončen po nalezení prvního řešení (pokud je nastaven na hodnotu true). Pokud je nastaven na hodnotu false, je ostrovní model ukončen po dokončení výpočtu všech vláken na všech stanicích účastnících se výpočtů.

Ve třetí sekci jsou parametry, které se týkají síťové komunikace stanic. Těmito parametry jsou:

- server_port - port na kterém naslouchají serverové části a na který se připojují klientské části aplikace

- storepass_ssl - heslo chránící integritu uložiště klíčů, ve kterém je uložen certifikát s privátním klíčem pro zabezpečenou TLS/SSL komunikaci

- keypass_ssl - heslo chránící soukromý klíč pro TLS/SSL komunikaci

- main_station - IP adresa stanice, na které bude spuštěna instance třídy SendingAddresses, která zajišťuje rozesílání informací o stanicích účastnících se výpočtů; ostatní stanice se k této řídící stanici připojují a posílají své údaje

Načítání informací z konfiguračního souboru zajišťuje třída Config. Jejímu konstruktoru je předána relativní cesta k INI souboru, ve kterém je konfigurace uložena. Na strojích ve výpočetním gridu však relativní cesty nefungují, protože aktuálním adresářem je vždy domovský adresář konkrétního stroje. Proto je nutné převést relativní cestu na absolutní. V následujícím výpisu (25) je kód konstruktoru třídy Config.

```
public Config(String filename){
  if (new File(filename).exists()==false){
    Class<Config> configClass = Config.class;
    URL url = configClass.getResource("Config.class");
    try{
      String url_str=URLDecoder.decode(url.toString(),"UTF-8");
      int start=6;
      if (System.getProperty("os.name").toString()
        .equals("Linux")) start=5;
      this.fileName = url_str.substring(start, url_str.length()-
        44)+filename;
    }catch (UnsupportedEncodingException e){
      e.printStackTrace();
    }
  }else
    this.fileName=filename;
  update();
}
```

Výpis kódu 25: Konstruktor třídy Config

Na začátku se konstruktor pokusí načíst soubor pomocí relativní cesty. Pokud se mu to nepodaří, zjistí si absolutní cestu ke třídě Config, z níž si vezme jen tu část k hlavnímu adresáři ostrovního modelu. Připojením relativní cesty k této části získá absolutní adresu ke konfiguračnímu souboru, kterou si uloží do proměnné fileName. Nakonec je zavolána metoda update(), která provede načtení parametrů z konfiguračního souboru do proměnných.

Metoda update() pomocí metody getProperty načítá hodnoty položek, které jsou převedeny na správný typ a ukládány do příslušných privátních proměnných. Pomocí příslušných „getter“ metod jsou potom hodnoty proměnných zpřístupněny ostatním metodám. V následující kódu (26) je uvedena část obsahu metody update.

```
public void update() {
  try{
    Properties p = new Properties();
    p.load(new FileInputStream("config_default.ini"));
    migrRate = Double.parseDouble(p.getProperty(
      "migration_rate"));
    migrSize = Integer.parseInt(p.getProperty("migration_size"));
    migrExchangeRate = Double.parseDouble(
      p.getProperty("migration_exchange_rate"));
    randomSelectionOut = Boolean.parseBoolean(
      p.getProperty("random_selection_out"));
    migrThr = Integer.parseInt(p.getProperty(
      "migration_threshold"));
  }catch (FileNotFoundException e){
    e.printStackTrace();
  }catch (IOException e){
    e.printStackTrace();
  }
}
```

Výpis kódu 26: Část metody update

2.7 Spuštění ostrovního modelu

2.7.1 Vlastní spuštění ostrovního modelu

Aby bylo možné ostrovní model spustit, je nutné vytvořit dvě třídy a to třídu implementující rozhraní IEvolutionStart a hlavní třídu, která provede spuštění ostrovního modelu a výpočtů.

```
public void start(){
  IntegerEvaluator ie = new IntegerEvaluator();
  DefaultEvolutionSpecifier e =
    new DefaultEvolutionSpecifier(ie);

  GPCore c = new GPCore("config_example1.ini", e,
    getOperatorDatabase());
  c.evolve();
}
```

Výpis kódu 27: Příklad metody Start třídy implementující IEvolutionStart

Rozhraní IEvolutionStart obsahuje pouze metodu Start, která nemá žádné parametry. Ve třídě, která toto rozhraní implementuje, musí metoda start obsahovat vlastní spuštění výpočtu genetického programování. Ve výpisu kódu 27 je ukázka příkladu metody start.

V hlavní třídě je potom vytvořena instance třídy Base ostrovního modelu a zavolána její metoda StartEvolutions, které je jako parametr předána instance třídy implementující rozhraní IEvolutionStart. Při spouštění ve výpočetním gridu je nutné aplikaci a poté konstruktoru třídy Base předat údaj o počtu požadovaných jader procesorů (viz. kap. 2.7.2). Pokud je konstruktoru Base předána nula, zjistí si počet procesorů sám metodou Runtime.getRuntime().availableProcessors(). Příklad, jak by mohla vypadat hlavní třída ostrovního modelu je ve výpisu 28.

```
public class Main {
  public static void main(String[] args){
    int procCount;
    if (args.length==1){
      try{
        procCount=Integer.parseInt(args[0]);
      }catch(Exception e){ procCount=0; }
    }else
      procCount=0;
    Base base=new Base(procCount);

    //třída implementující rozhraní IEvolutionStart:
    ExecEvolution ev = new ExecEvolution();
    base.StartEvolutions(ev);
  }
}
```

Výpis kódu 28: Spuštění ostrovního modelu

2.7.2 Spuštění ostrovního modelu ve výpočetním gridu

Pro testování ostrovního modelu jsem používal výpočetní grid MetaCentra ([18]). Na strojích v gridu běží Unixové operační systémy. Protože je zde mnoho uživatelů, nespouští se výpočetní úlohy přímo, ale zadávají se do plánovacího systému PBSPro, který uživateli stroje přidělí až když jsou k dispozici. Úloha se do plánovacího systému zadává spolu s požadavky na počet strojů a jejich procesorů, vlastnostmi strojů, velikostí potřebné paměti a také, do jaké fronty čekající úlohu zařadit. K dispozici je fronta short (délka běhu úlohy do 2 hodin), normal (délka běhu úlohy do

24 hodin) a long (délka běhu úlohy do 30 dnů). Pro spouštění aplikace v gridu jsem používal příkaz:

```
qsub -q short -l mem=1500mb -l nodes=3:ppn=2:x86:linux:nfs4
execute_tasks.sh .
```

V uvedeném příkladu jsou požadovány tři stroje s pamětí 1 500 MB, každý s dvěma procesory. Stroje jsou typu x86, na kterých je operační systém linux a souborový systém nfs4. Délka běhu úlohy je do dvou hodin. Více informací o možnostech spuštění aplikace ve výpočetním gridu MetaCentra lze nalézt v [19]. Soubor execute_tasks.sh je skript pro Bash, který zajistí spuštění aplikace na požadovaném počtu strojů. Obsah tohoto skriptu je ve výpisu 29.

```
#!/bin/bash
cp /storage/home/muj_login/config_default_backup.ini
/storage/home/muj_login/config_default.ini

numc=`wc -l $PBS_NODEFILE`
nn=`sort -u $PBS_NODEFILE | wc -l`
np=$(($numc/$nn));

for((i=1; i<=nn; i++))
do
  pbsdsh -n $i /storage/home/muj_login/task.sh $i $np &
  if [$i -eq 1] ; then
    sleep 10;
  fi
done
wait
```

Výpis kódu 29: Obsah skriptu execute_tasks.sh

První řádek vytvoří soubor config_default.ini, ve kterém je uložena konfigurace ostrovního modelu popsaná v kapitole 2.6 pouze bez parametru main_station. Protože dopředu neznáme IP adresu stanice, na níž bude spuštěna řídící stanice, bude tento řádek doplněn skriptem později. Následující řádek spočítá, kolik procesorů celkem máme přiděleno a výsledek uloží do proměnné numc. Seznam přidělených strojů se nachází v souboru, jehož cesta je uložena v proměnné $PBS_NODEFILE. Pokud žádáme o více jader procesoru, je stejná adresa v souboru obsažena vícekrát (např. pokud žádáme o pět stanic s třemi procesory, bude v souboru dohromady patnáct adres stanic). Na dalším řádku je spočítán počet přidělených stanic a výsledek je uložen do proměnné nn. Vydělením numc/nn dostaneme počet procesorů na jednu

stanici. Poté už následuje cyklus přes přiřazené stroje, ve kterém je postupně spouštěn Bashový skript task.sh na všech stanicích, jenž spustí vlastní aplikaci. Skriptu task.sh je jako parametr předáváno pořadové číslo stanice a počet přidělených procesorů. Ve výpisu kódu 30 se nachází obsah tohoto skriptu.

```
#!/bin/bash
if [ $1 -eq 1 ] ; then
  IP=`ifconfig  | grep 'inet addr:'| grep -v '127.0.0.1' |
    cut -d: -f2 | awk '{ print $1}'`;
  echo "main_station=$IP"
  >> /storage/home/muj_login/config_default.ini
  sleep 2;
fi
. /packages/run/modules-2.0/init/bash
module add ics.muni.cz jdk-1.6.0
java -classpath /storage/home/muj_login/bin:/storage/home/
muj_login/lib/mina-core-1.1.7.jar:/storage/home/muj_login/lib/
cloning-1.4.jar:/storage/home/muj_login/lib/junit.jar:/
storage/home/muj_login/lib/log4j.jar:/storage/home/muj_login/
lib/mina-core-1.1.7-sources.jar:/storage/home/muj_login/
lib/mina-filter-codec-netty-1.1.7.jar:/storage/home/muj_login/
lib/mina-filter-codec-netty-1.1.7-sources.jar:/storage/home/
muj_login/lib/mina-filter-ssl-1.1.7.jar:/storage/home/muj_login/
lib/mina-filter-ssl-1.1.7-sources.jar:/storage/home/muj_login/
lib/mina-integration-jmx-1.1.7.jar:/storage/home/muj_login/lib/
mina-integration-jmx-1.1.7-sources.jar:/storage/home/muj_login/
lib/mina-integration-spring-1.1.7.jar:/storage/home/muj_login/
lib/mina-integration-spring-1.1.7-sources.jar:/storage/home/
muj_login/lib/objenesis-1.2.jar:/storage/home/muj_login/lib/
slf4j-api-1.5.8.jar:/storage/home/muj_login/lib/
slf4j-jdk14-1.5.8.jar
bin/cz.vutbr.feec.utko.xkozem00.sandbox.Main $2
```

Výpis kódu 30: Obsah skriptu task.sh

Pokud je pořadové číslo této stanice jedna, je IP adresa stanice uložena do konfiguračního souboru config_default.ini do položky main_station. Poté je inicializován systém modulů a konkrétní verze jazyka Java. Nakonec je spuštěna vlastní aplikace a je jí předán počet přidělených procesorů (v parametru classpath je nutné vyjmenovat všechny potřebné knihovny).

3 ZÁVĚR

Cílem této práce bylo navrhnout ostrovní model genetického programování s efektivním využitím prostředků stanice a navrhnout a zabezpečit komunikaci mezi stanicemi při výměně jedinců. K implementaci zabezpečené komunikaci mezi stanicemi byl využit Mina framework s protokolem TLS/SSL. Efektivní využití prostředků stanic je zajištěno rozdělením populace na menší subpopulace, které jsou přiděleny jednotlivým procesorovým jádrům stanice. Každé jádro tak zpracovává jednu subpopulaci.

Byla implementována zabezpečená komunikace mezi více stanicemi a migrace jedinců mezi jejich populacemi. Aplikace každé stanice obsahuje serverovou a klientskou část. Klientská část odesílá migrující jedince serverové části cílové stanice, která zajišťuje začlenění těchto jedinců do cílové subpopulace. Dále bylo implementováno přidělení subpopulací jednotlivým jádrům procesorů a migrace jedinců mezi populacemi na téže stanici. K migraci mezi populacemi jsou využívány vstupní fronty, do kterých jsou příchozí jedinci ukládáni a při příští generaci jsou přesunuti z fronty do příslušné cílové populace.

Pro usnadnění konfigurace ostrovního modelu na jednotlivých stanicích bylo implementováno automatické rozesílání informací o ostatních stanicích účastnících se výpočtů. Na jedné stanici je aplikace spuštěna v řídícím módu a zajišťuje přeposílání informací (IP adresa a počet jader procesorů), které dostává od jednotlivých stanic po spuštění jejich výpočtů, ostatním stanicím.

Pro zabezpečení komunikace je využíván protokol TLS/SSL, pomocí kterého je před vlastní migrací jedinců mezi stanicemi pomocí certifikátu autentizována serverová část (která přijímá odeslané jedince) a pomocí veřejného a soukromého klíče serveru je provedena výměna klíčů pro symetrické šifrování posílaných jedinců.

LITERATURA

[1] *Apache Mina Documentation* [online]. c2004-2007 [cit. 2009-12-09]. Dostupný z WWW: <http://mina.apache.org/report/trunk/xref/org/apache/mina/example/echoserver/ssl/BogusSSLContextFactory.html>.

[2] BRANKE, Jürgen, KAMPER, Andreas, SCHMECK, Hartmut. Distribution of evolutionary algorithms in heterogeneous networks. In *Proceedings of Genetic and Evolutionary Computation Conference - GECCO 2004.* Seattle : Springer-Verlag, 2004. s. 923-934. Dostupný z WWW: <http://www.cs.york.ac.uk/rts/docs/GECCO_2004/Conference%20proceedings/papers/3102/31020923.pdf>.

[3] CANTÚ-PAZ, Erick. A survey of parallel genetic algorithms. In *Calculateurs Paralleles, Reseaux et Systems Repartis : Volume 10, Number 2.* Paris : [s.n.], 1998. s. 141-171. Dostupný z WWW: <http://neo.lcc.uma.es/cEA-web/documents/cant98.pdf>.

[4] CANTÚ-PAZ, Erick. Migration policies, selection pressure, and parallel evolutionary algorithms. In *Journal of Heuristics : Volume 7, Number 4.* [s.l.] : [s.n.], 2001. s. 311-334. Dostupný z WWW: <http://www.evolutionaria.com/publications/jheur-pressure.pdf>.

[5] FORREST, Stephanie. Genetic Algorithms: Principles of Natural Selection Applied to Computation. In *SCIENCE : Volume 261, Issue 5123.* [s.l.] : [s.n.], 1993. s. 872-878. Dostupný z WWW: <`http://www.cs.unm.edu/~melaniem/courses/Reading_files/Forrest1993.pdf`>.

[6] GUSTAFSON, Steven, BURKE, Edmund. The speciating island model: an alternative parallel evolutionary algorithm. In *Journal of Parallel and Distributed Computing : Volume 66, Issue 8.* [s.l.] : [s.n.], 2006. s. 1025-1036.

[7] HORNBY, Gregory. ALPS: The Age-Layered Population Structure for Reducing the Problem of Premature Convergence. In *Proceedings of Genetic and Evolutionary Computation Conference – GECCO-2006.* [s.l.] : [s.n.], 2006. s. 815-822. Dostupný z WWW: <`http://www.cs.york.ac.uk/rts/docs/GECCO_2006/docs/p815.pdf`>.

[8] HUAYANG, Xie. *An Analysis of Selection in Genetic Programming.* London : [s.n.], 2009. 227 s. Victoria University of Wellington. Dizertační práce. Dostupný z WWW: <http://researcharchive.vuw.ac.nz/handle/10063/837>.

[9] HYNEK, Josef. *Genetické algoritmy a genetické programování.* 1. vyd. Praha : Grada, 2008. 200 s. ISBN 978-80-247-2695-3.

[10] *JDK 6 Documentation* [online]. c2006 [cit. 2009-10-27]. Dostupný z WWW: <http://java.sun.com/javase/6/docs/>.

[11] KICINGER, Rafal, ARCISZEWSKI, Tomasz, DE JONG, Kenneth. *Distributed Evolutionary Design : Island-Model Based Optimization of Steel Skeleton Structures in Tall Buildings.* [s.l.] : [s.n.], 2004. 12 s. George Mason University. Dostupný z WWW: <http://www.kicinger.com/publications/pdf/KicingerICCCBE-X2004.pdf>.

[12] KOZA, Jon R. *Genetic programming : On the Programming of Computers by Means of Natural Selection.* Cambridge, Massachusetts : MIT Press, 1998. 813 s. ISBN 0-262-11170-5.

[13] KOZA, John, POLI, Riccardo. Genetic programming. In *Search Methodologies.* [s.l.] : Springer US, 2005. s. 127-164. Dostupný z WWW: <`http://www.springerlink.com/content/l64g618r24l82v26`>.

[14] LANGDON, William, QURESHI, Adil. *Genetic Programming : Computers using "Natural Selection" to generate programs.* [s.l.] : [s.n.], 1995. 45 s. University College London. Dostupný z WWW: <www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/surveyRN76.pdf>.

[15] LÓPEZ, Edgar, POLI, Riccardo, COELLO COELLO, Carlos. *Reusing Code in Genetic Programming.* [s.l.] : [s.n.], 2004. 10 s. Dostupný z WWW: <http://cswww.essex.ac.uk/staff/poli/papers/eurogpedgar2004.pdf>.

[16] LUKE, Sean, SPECTOR, Lee. A Revised Comparison of Crossover and Mutation in Genetic Programming. In *Proceedings of the Third Annual Genetic Programming Conference (GP98).* San Fransisco : Morgan Kaufmann, 1998. s. 208-213. Dostupný z WWW: <`http://cs.gmu.edu/~sean/papers/revisedgp98.pdf`>.

[17] MAN, K. F., TANG, K. S., KWONG, S. Genetic Algorithms: Concepts and Applications. In *IEEE TRANSACTIONS ON INDUSTRIAL ELECTRONICS : Volume 43, Issue 5.* [s.l.] : [s.n.], 1996. s. 519-534.

[18] *MetaCentrum* [online]. 2009-10-01 [cit. 2010-03-03]. Dostupné z WWW: <http://meta.cesnet.cz>.

[19] *MetaCentrum* [online]. 2009-11-25 [cit. 2010-03-03]. Spouštění aplikací. Dostupné z WWW: <http://meta.cesnet.cz/cs/docs/aplikace/run/index.html>.

[20] MITCHELL, Melanie. *An Introduction to Genetic Algorithms.* Cambridge, Massachusetts : MIT Press, 1996. 158 s. ISBN 0-262-13316-4.

[21] NEDJAH, Nadia, ABRAHAM, Ajith, DE MACEDO, Luiza. *Genetic Systems Programming* [s.l.] : [s.n.], 2006. 233 s. Studies in Computational Intelligence; sv. 13. ISBN 978-3-540-29849-6.

[22] O'NEILL, Michael, RYAN, Conor. Grammatical Evolution. In *IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION Volume 5, Issue 4.* [s.l.] : [s.n.], 2001. s. 349-358.

[23] POLI, Riccardo, LANGDON, William, MCPHEE, Nicholas. *A Field Guide to Genetic Programming.* [s.l.] : [s.n.], 2008. 250 s. Dostupný z WWW: <http://www.lulu.com/product/download/a-field-guide-to-genetic-programming/2502914>. ISBN 978-1-4092-0073-4.

[24] *Progress software* [online]. c1993-2009 [cit. 2009-12-08]. Dostupný z WWW: <http://www.iona.com/support/docs/e2a/asp/5.0/mainframe/ssl/html/Intro4.html>.

[25] SEKAJ, Ivan. Robust parallel genetic algorithms with re-initialisation. In *Parallel Problem Solving from Nature - PPSN VIII.* Berlín : Springer, 2004. s. 411-419. Dostupný z WWW: <http://www.springerlink.com/content/c8kytl9wjl26h2h8/fulltext.pdf>.

[26] SETTE, Stefan, BOULLART, Luc. Genetic programming: principles and applications. In *ENGINEERING APPLICATIONS OF ARTIFICIAL INTELLIGENCE : Volume 14, Issue 6.* [s.l.] : [s.n.], 2001. s. 727-736. Dostupný z WWW: <http://linkinghub.elsevier.com/retrieve/pii/S0952197602000131>.

[27] SCHMITT, Lothar. Theory of genetic algorithms. In *THEORETICAL COMPUTER SCIENCE : Volume 259, Issue 1-2.* [s.l.] : [s.n.], 2001. s. 1-61.

[28] SKOLICKI, Zbigniew, DE JONG, Kenneth. The Influence of Migration Sizes and Intervals on Island Models. In *Proceedings of Genetic and Evolutionary Computation Conference – GECCO-2005.* [s.l.] : ACM Press, 2005. s. 1295-1302. Dostupný z WWW: <http://cs.gmu.edu/~eclab/papers/skolicki05influence.pdf>.

[29] SMITH, Peter. *Controlling Code Growth in Genetic Programming.* London : [s.n.], [200-?]. 6 s. City University. Dostupný z WWW: <http://www.soi.city.ac.uk/~peters/pub/Leicester.ps>.

[30] TANG, J., LIM, M., ONG, Y., Er, M. Study of migration topology in island model parallel hybrid-GA for large scale quadratic assignment problems. In *Control, Automation, Robotics and Vision Conference, 2004. ICARCV 2004 8th.* [s.l.] : [s.n.], 2004. s. 2286-2291. Dostupný z WWW: <`http://wieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1469788`>.

SEZNAM PŘÍLOH

A PŘÍLOHY

A.1 Obsah přiloženého CD

- Diplomová práce ve formátu pdf
- Program ostrovní model včetně zdrojových kódů
  - adresář bin: zkompilované třídy programu pro Java SE 1.6
  - adresář Framework: framework od Ing. Radima Burgeta implementující genetické programování
  - adresář lib: knihovny Mina frameworku
  - adresář priklad_pouziti: příklad použití ostrovního modelu (jednoduché srovnání čísel)
  - adresář src: zdrojové kódy ostrovního modelu
  - soubor config_default.ini: kompletní konfigurační soubor pro ostrovní model
  - soubor config_default_backup.ini: konfigurační soubor pro ostrovní model spouštěný ve výpočetním gridu

A.2 Spuštění aplikace umístěné na CD

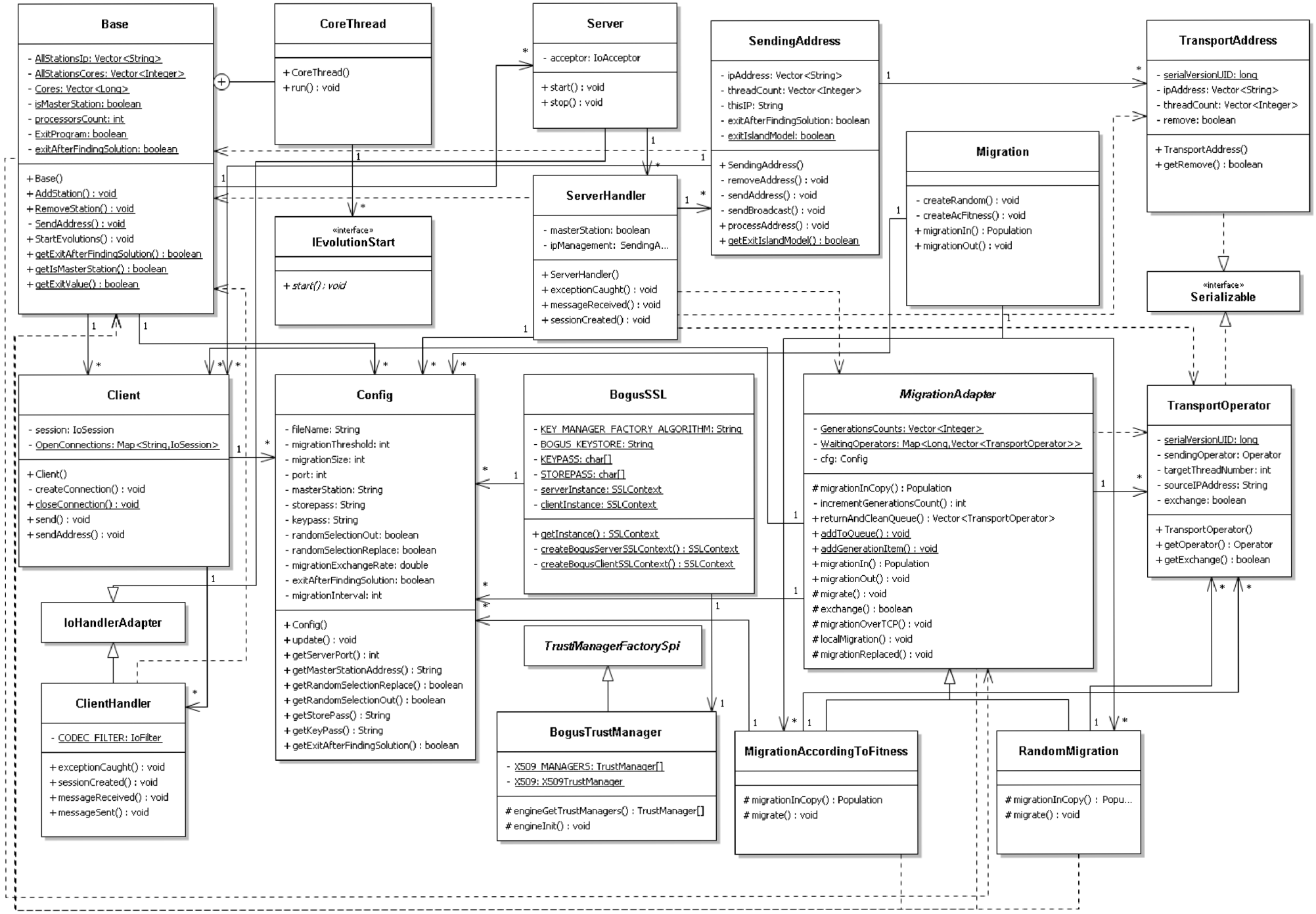
Pro spuštění aplikace je potřeba mít nainstalovaný Java Runtime Environment 1.6. Spuštění ostrovního modelu ve výpočetním gridu již bylo popsáno v kapitole 2.7. V případě spuštění aplikace na PC stačí ostrovní model nastavit v konfiguračním souboru config_default.ini a potom spustit z příkazové řádky Windows následujícím příkazem (znak * zastupuje absolutní cestu k adresáři s ostrovním modelem, celý příkaz je v jednom řádku a za středníky nejsou mezery). V případě spouštění v Linuxu stačí použít pro oddělení jednotlivých cest místo středníků dvojtečky.

```
java -classpath */IslandModel;*/IslandModel/bin;
*/IslandModel/lib/mina-core-1.1.7.jar;
*/IslandModel/bin/config_default.ini;
*/IslandModel/lib/cloning-1.4.jar;
*/IslandModel/lib/junit.jar;
*/IslandModel/lib/log4j.jar;
*/IslandModel/lib/mina-core-1.1.7-sources.jar;
*/IslandModel/lib/mina-filter-codec-netty-1.1.7.jar;
```

```
*/IslandModel/lib/mina-filter-codec-netty-1.1.7-sources.jar;
*/IslandModel/lib/mina-filter-ssl-1.1.7.jar;
*/IslandModel/lib/mina-filter-ssl-1.1.7-sources.jar;
*/IslandModel/lib/mina-integration-jmx-1.1.7.jar;
*/IslandModel/lib/mina-integration-jmx-1.1.7-sources.jar;
*/IslandModel/lib/mina-integration-spring-1.1.7.jar;
*/IslandModel/lib/mina-integration-spring-1.1.7-sources.jar;
*/IslandModel/lib/objenesis-1.2.jar;
*/IslandModel/lib/slf4j-api-1.5.8.jar;
*/IslandModel/lib/slf4j-jdk14-1.5.8.jar
 cz.vutbr.feec.utko.xkozem00.priklad_pouziti.Main
```

A.3 Diagram tříd ostrovního modelu

Na následující stránce je znázorněn diagram tříd ostrovního modelu. U metod nejsou uvedeny jejich parametry. Uvedením parametrů jednotlivých metod by velikost celého diagramu značně narostla.


Base
- AllStationsIp: Vector<String>
- AllStationsCores: Vector<Integer>
- Cores: Vector<Long>
- isMasterStation: boolean
- processorsCount: int
- ExitProgram: boolean
- exitAfterFindingSolution: boolean
+ Base()
+ AddStation() : void
+ RemoveStation() : void
- SendAddress() : void
+ StartEvolutions() : void
+ getExitAfterFindingSolution() : boolean
+ getIsMasterStation() : boolean
+ getExitValue() : boolean
CoreThread
+ CoreThread()
+ run() : void
Server
- acceptor: IoAcceptor
+ start() : void
+ stop() : void
SendingAddress
- ipAddress: Vector<String>
- threadCount: Vector<Integer>
- thisIP: String
- exitAfterFindingSolution: boolean
- exitIslandModel: boolean
+ SendingAddress()
- removeAddress() : void
- sendAddress() : void
- sendBroadcast() : void
+ processAddress() : void
+ getExitIslandModel() : boolean
TransportAddress
- serialVersionUID: long
- ipAddress: Vector<String>
- threadCount: Vector<Integer>
- remove: boolean
+ TransportAddress()
+ getRemove() : boolean
«interface»
IEvolutionStart
+ start() : void
ServerHandler
- masterStation: boolean
- ipManagement: SendingA...
+ ServerHandler()
+ exceptionCaught() : void
+ messageReceived() : void
+ sessionCreated() : void
Migration
- createRandom() : void
- createAcFitness() : void
+ migrationIn() : Population
+ migrationOut() : void
«interface»
Serializable
Client
- session: IoSession
- OpenConnections: Map<String,IoSession>
+ Client()
- createConnection() : void
+ closeConnection() : void
+ send() : void
+ sendAddress() : void
Config
- fileName: String
- migrationThreshold: int
- migrationSize: int
- port: int
- masterStation: String
- storepass: String
- keypass: String
- randomSelectionOut: boolean
- randomSelectionReplace: boolean
- migrationExchangeRate: double
- exitAfterFindingSolution: boolean
- migrationInterval: int
+ Config()
+ update() : void
+ getServerPort() : int
+ getMasterStationAddress() : String
+ getRandomSelectionReplace() : boolean
+ getRandomSelectionOut() : boolean
+ getStorePass() : String
+ getKeyPass() : String
+ getExitAfterFindingSolution() : boolean
BogusSSL
- KEY_MANAGER_FACTORY_ALGORITHM: String
- BOGUS_KEYSTORE: String
- KEYPASS: char[]
- STOREPASS: char[]
- serverInstance: SSLContext
- clientInstance: SSLContext
+ getInstance() : SSLContext
- createBogusServerSSLContext() : SSLContext
- createBogusClientSSLContext() : SSLContext
MigrationAdapter
- GenerationsCounts: Vector<Integer>
- WaitingOperators: Map<Long,Vector<TransportOperator>>
- cfg: Config
migrationInCopy() : Population
- incrementGenerationsCount() : int
+ returnAndCleanQueue() : Vector<TransportOperator>
+ addToQueue() : void
+ addGenerationItem() : void
+ migrationIn() : Population
+ migrationOut() : void
# migrate() : void
# exchange() : boolean
# migrationOverTCP() : void
# localMigration() : void
# migrationReplaced() : void
TransportOperator
- serialVersionUID: long
- sendingOperator: Operator
- targetThreadNumber: int
- sourceIPAddress: String
- exchange: boolean
+ TransportOperator()
+ getOperator() : Operator
+ getExchange() : boolean
IoHandlerAdapter
ClientHandler
- CODEC_FILTER: IoFilter
+ exceptionCaught() : void
+ sessionCreated() : void
+ messageReceived() : void
+ messageSent() : void
TrustManagerFactorySpi
BogusTrustManager
- X509_MANAGERS: TrustManager[]
- X509: X509TrustManager
engineGetTrustManagers() : TrustManager[]
engineInit() : void
MigrationAccordingToFitness
migrationInCopy() : Population
migrate() : void
RandomMigration
migrationInCopy() : Popu...
migrate() : void