VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

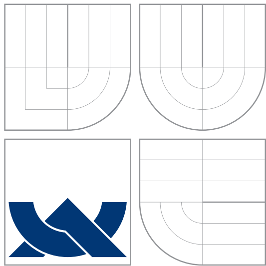
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

EVOLUČNÍ NÁVRH KOMBINAČNÍCH OBVODŮ NA POČÍTAČOVÉM CLUSTERU

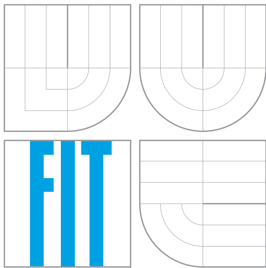
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE　　Bc. RICHARD PÁNEK
AUTHOR

BRNO 2015



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

EVOLUČNÍ NÁVRH KOMBINAČNÍCH OBVODŮ NA POČÍTAČOVÉM CLUSTERU

EVOLUTIONARY DESIGN OF COMBINATIONAL CIRCUITS ON COMPUTER CLUSTER

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. RICHARD PÁNEK
AUTHOR

VEDOUCÍ PRÁCE Ing. RADEK HRBÁČEK
SUPERVISOR

BRNO 2015

Abstrakt

Tato diplomová práce se zabývá evolučními algoritmy a jejich použitím při návrhu kombinačních obvodů. Pro tento typ úloh je nejvhodnější genetické programování, především pak CGP. Dále se zabývá výpočtem na počítačových clusterech a použitím evolučních algoritmů na nich. Pro tento výpočet se nejvíce hodí ostrovní modely s CGP. Pro jejich zlepšení je navržen nový způsob rekombinace v CGP. Tento návrh je implementován a testován na počítačovém clusteru.

Abstract

This master's thesis deals with evolutionary algorithms and how them to use to design of combinational circuits. Genetic programming especially CGP is the most applicable to use for this type of task. Furthermore, it deals with computation on computer cluster and the use of evolutionary algorithms on them. For this computation is the most suited island models with CGP. Then a new way of recombination in CGP is designed to improve them. This design is implemented and tested on the computer cluster.

Klíčová slova

Evoluční algoritmy, genetické programování, kartézké genetické programování, ostrovní modely, počítačové clustery, paralelní programování.

Keywords

Evolutionary Algorithm, Genetic Programing, Cartesian Genetic Programming, Island Models, Computer Cluster, Parallel Programming.

Citace

Richard Pánek: Evoluční návrh kombinačních obvodů
na počítačovém clusteru, diplomová práce, Brno, FIT VUT v Brně, 2015

Evoluční návrh kombinačních obvodů na počítačovém clusteru

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Radka Hrbáčka a uvedl jsem všechny použité zdroje informací.

....
Richard Pánek
26. května 2015

Poděkování

Rád bych poděkoval paní Ing. Michaele Šikulové za námět a literaturu a také panu Ing. Radku Hrbáčkovi za vedení.

© Richard Pánek, 2015.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod **5**

2 Evoluční algoritmus **6**
2.1 Od přírody k informatice 6
2.1.1 Rekombinace (křížení) 7
2.1.2 Mutace 9
2.1.3 Výběr nové populace (selekce) 10
2.2 Genetické programování 10
2.2.1 Terminály 10
2.2.2 Funkce 10
2.2.3 Křížení 11
2.2.4 Mutace 11
2.2.5 Shrnutí genetického programování 11
2.3 Kartézské genetické programování (CGP) 12
2.3.1 Vývoj v CGP 14
2.3.2 Mutace v CGP 14
2.3.3 Fitness v CGP při návrhu obvodů 14

3 Počítačové clustery, paralelní programování **15**
3.1 Paralelní programování 16
3.1.1 OpenMP 16
3.1.2 MPI 16

4 Evoluční návrh obvodů na clusteru **18**
4.1 Ostrovní modely 18
4.1.1 Ostrovní modely s CGP 19
4.2 Návrh vylepšení 19
4.3 Implementace původní varianty 20
4.4 Implementace vylepšení 21

5 Testování **23**
5.1 Čtyřbitová sčítačka 23
5.2 Pětibitová sčítačka 25
5.3 Šestibitová sčítačka 25
5.4 Sedmibitová sčítačka 30
5.5 Osmibitová sčítačka 36
5.6 Devítibitová sčítačka 42
5.7 Dvoubitová násobička 42

5.8 Tříbitová násobička 44
5.9 Čtyřbitová násobička 48

6 Závěr **51**

Seznam obrázků

2.1 Schéma evolučního algoritmu. 8
2.2 Rodičovské chromozomy. 9
2.3 Ukázka dvoubodového křížení. 9
2.4 Ukázka uniformního křížení. 9
2.5 Ukázka mutace třetího genu. 10
2.6 Ukázka křížení genetického programování. 11
2.7 Ukázka mutace genetického programování. 12
2.8 Schéma CGP. 13
2.9 Chromozom CGP. 13

4.1 Rekombinace lineárního CGP. 20
4.2 Procentuální úspěšnost nového křížení 22

5.1 Grafy statistik při návrhu čtyřbitové sčítačky 24
5.2 Grafy statistik při návrhu pětibitové sčítačky na jednom uzlu 26
5.3 Grafy statistik při návrhu pětibitové sčítačky na dvou uzlech 27
5.4 Grafy statistik při návrhu šestibitové sčítačky na jednom uzlu 28
5.5 Grafy statistik při návrhu šestibitové sčítačky na dvou uzlech 29
5.6 Grafy statistik při návrhu šestibitové sčítačky na třech uzlech 31
5.7 Grafy statistik při návrhu šestibitové sčítačky na čtyřech uzlech 32
5.8 Grafy statistik při návrhu sedmibitové sčítačky na jednom uzlu 33
5.9 Grafy statistik při návrhu sedmibitové sčítačky na dvou uzlech 34
5.10 Grafy statistik při návrhu sedmibitové sčítačky na třech uzlech 35
5.11 Grafy statistik při návrhu sedmibitové sčítačky na čtyřech uzlech 37
5.12 Grafy statistik při návrhu sedmibitové sčítačky na pěti uzlech 38
5.13 Grafy statistik při návrhu osmibitové sčítačky na dvou uzlech 39
5.14 Grafy statistik při návrhu osmibitové sčítačky na třech uzlech 40
5.15 Grafy statistik při návrhu osmibitové sčítačky na čtyřech uzlech 41
5.16 Grafy statistik při návrhu osmibitové sčítačky na pěti uzlech 43
5.17 Grafy statistik při návrhu osmibitové sčítačky na šesti uzlech 44
5.18 Grafy statistik při návrhu devítibitové sčítačky 45
5.19 Grafy statistik při návrhu dvoubitové násobičky 46
5.20 Grafy statistik při návrhu tříbitové násobičky na 16 ostrovech 47
5.21 Grafy statistik při návrhu tříbitové násobičky na 24 ostrovech 49
5.22 Grafy statistik při návrhu čtyřbitové násobičky 50

Seznam tabulek

5.1 Výsledky čtyřbitové sčítačky . . . 24
5.2 Výsledky pětibitové sčítačky na jednom uzlu . . . 26
5.3 Výsledky pětibitové sčítačky na dvou uzlech . . . 27
5.4 Výsledky šestibitové sčítačky na jednom uzlu . . . 28
5.5 Výsledky šestibitové sčítačky na dvou uzlech . . . 29
5.6 Výsledky šestibitové sčítačky na třech uzlech . . . 31
5.7 Výsledky šestibitové sčítačky na čtyřech uzlech . . . 32
5.8 Výsledky sedmibitové sčítačky na jednom uzlu . . . 33
5.9 Výsledky sedmibitové sčítačky na dvou uzlech . . . 34
5.10 Výsledky sedmibitové sčítačky na třech uzlech . . . 35
5.11 Výsledky sedmibitové sčítačky na čtyřech uzlech . . . 37
5.12 Výsledky sedmibitové sčítačky na pěti uzlech . . . 38
5.13 Výsledky osmibitové sčítačky na dvou uzlech . . . 39
5.14 Výsledky osmibitové sčítačky na třech uzlech . . . 40
5.15 Výsledky osmibitové sčítačky na čtyřech uzlech . . . 41
5.16 Výsledky osmibitové sčítačky na pěti uzlech . . . 43
5.17 Výsledky osmibitové sčítačky na šesti uzlech . . . 44
5.18 Výsledky devítibitové sčítačky . . . 45
5.19 Výsledky dvoubitové násobičky . . . 46
5.20 Výsledky tříbitové násobičky na 16 ostrovech . . . 47
5.21 Výsledky tříbitové násobičky na 24 ostrovech . . . 49
5.22 Výsledky čtyřbitové násobičky . . . 50

Kapitola 1

Úvod

Návrh rozsáhlých kombinačních obvodů je náročný, protože je zapotřebí velké množství znalostí a zkušeností, aby byl navrhnut kvalitní obvod. Takový přístup je časově náročný a drahý. Proto se přišlo s myšlenkou, že bychom se mohli inspirovat přírodními procesy. V přírodě se vše postupně vyvíjí, slabší jedinci zanikají a silnější přežívají. Tento vývoj je uskutečněn v generacích, kdy se předávají informace z rodičů na potomky. V přírodě tento vývoj trvá tisíce i milióny let. Ovšem na počítačích jsme schopni tento vývoj zkrátit na čas mnohokrát kratší než jeden lidský život.

Na počítačích je tento přístup označen jako evoluční algoritmy. Právě ty napodobují proces vývoje známý z přírody a používají ho pro výpočet. Evoluční přístup je možné aplikovat na množství problémů, které se v informatice objevují. Tato práce je zaměřena pouze na malou oblast, jak už bylo řečeno, na návrh kombinačních obvodů.

Evoluční algoritmus pracuje v iteracích a snaží se nalézt optimální řešení problému v daném prohledávaném prostoru. U složitých problémů je pravděpodobné, že tento prostor bude značně rozsáhlý. Proto by nám na klasickém stolním počítači hledání řešení zabralo více času, než si můžeme dovolit. Je potřeba hledat způsob, jak tento čas zkrátit na přiměřené množství. Jedním z přístupů je využití počítačových clusterů. Počítačové clustery jsou složeny z počítačů propojených komunikační sítí. Ovšem na takovém clusteru nemůžeme použít běžný program pro počítač, protože nemáme jen jeden procesor, ale stovky. Proto se klade důraz na rozložení výpočtu mezi tyto procesory tak, abychom je efektivně využili a omezili režii spojenou například s komunikací mezi procesory na minimum, a o to více využili výpočetní výkon. Je zřejmé, že si nevystačíme s jednoduchým evolučním algoritmem, ale musíme jej přizpůsobit pro výpočet na clusteru.

Kapitola 2

Evoluční algoritmus

Evoluční algoritmy [5] jsou založeny na principu vývoje jedinců v přírodě. Každý jedinec si nese kód sebe sama v každé své buňce, která obsahuje molekulu DNA, u některých virů molekulu RNA se stejnou funkcí. Dále se budeme zabývat pouze molekulou DNA. Tu si můžeme představit jako dva řetězce symbolů. Tyto řetězce jsou ovšem komplementární, tudíž nám bude stačit pracovat pouze s jedním. Každý řetězec se skládá pouze z omezeného množství symbolů, u DNA to jsou čtyři symboly, které odpovídají čtyřem nukleotidům. Při vytváření nového jedince jsou na začátku dva řetězce od dvou rodičů. Ovšem nový jedinec potřebuje opět jen jeden řetězec. V přírodě proto existuje mechanismus, který z těchto dvou řetězců vytvoří jeden. Tímto mechanismem je křížení, kdy se řetězce přiloží k sobě a do výsledného se vždy vezme symbol vždy jen od jednoho rodiče. Takto vznikne jeden nový řetězec, který je kombinací svých rodičů. Ovšem protože příroda není dokonalá, během tohoto procesu může dojít i ke vzniku chyby. Touto chybou je myšlen případ, kdy místo aby na dané pozici byl symbol od jednoho z rodičů, dostane se sem zcela náhodný symbol. Dále budeme tento vznik chyby nazývat mutace.

2.1 Od přírody k informatice

Úvod kapitoly stručně popisuje fungování v přírodě, nyní se zaměříme na možnost implemetace evoluce v počítači.

Nejdříve je potřeba daný problém zakódovat do chromozomu, který se skládá z jednotlivých genů. Chromozom je tedy ekvivalentem DNA molekuly v přírodě. Toto kódování slouží pro převod z genotypu na fenotyp. Fenotypem je myšlen výsledek, v přírodě to je právě jedinec a pro nás je to jedno z řešení problému. Genotyp je zakódování, v přírodě je to již zmíněná molekula DNA, pro nás chromozom. Zvolit správné zakódování není triviální problém, protože na něm záleží kvalita kandidátních řešení. Není vždy zřejmé, jaké povede nejlépe k cíli, vždy záleží na schopnostech a zkušenostech vývojáře.

Když máme problém zakódovaný v chromozomu, musíme umět určit kvalitu daného jedince. K ohodnocení využíváme fitness funkci. Ta má na vstupu chromozom a jejím výsledkem je hodnota, obvykle kladné číslo. Čím je číslo větší, tím je jedinec kvalitnější. Je možné pracovat i s možností, kdy menší číslo odpovídá lepšímu jedinci. Díky fitness funkci tedy můžeme jedince seřadit od nejkvalitnějšího. Právě takto seřazení jedinci budou potřeba při výběru jedinců. Výběr se využívá v rámci vývoje populace jednak při určování jedinců, kteří budou vytvářet potomky, a pak také pří vytváření nové populace.

Evoluční algoritmus vždy pracuje s populací. Populace je množina jedinců, každý repre-

zentovaný chromozomem. Na začátku je populace náhodně vygenerována. Druhou možností je počáteční populaci zvolit, tedy nadefinovat ručně. Tento způsob se využívá zejména u optimalizačních úloh, kdy již máme nějaké řešení a hledáme lepší. Velikost populace je jedním z parametrů algoritmu. Ovlivňuje jak pravděpodobnost nalezení řešení, tak i prostorovou a časovou složitost. Malá populace může zůstat jen v lokálním optimu, zatímco příliš velká nenalezne řešení v rozumném čase. Je třeba optimální velikost zjistit experimentálně nebo sofistifikovaně odhadnout.

Výpočet probíhá v generacích, přičemž každá generace se skládá z následujících částí. Vstupem i výstupem je populace složená s chromozomů. Nejdříve je populace ohodnocena spočítáním fitness funkce pro každý její chromozom. Když je nalezeno řešení problému výpočet končí. Druhou možností ukončení výpočtu je dosažení předem definovaného počtu generací, tzn. na začátku zvolíme, jak dlouho má výpočet trvat. Jinak pokračujeme výběrem jedinců (selekcí) pro rekombinaci (křížení). Dalším parametrem je počet jedinců, kteří takto vzniknou. Opět platí, že malý počet může uváznout v lokálním extrému a velký bude časově náročný. Následuje mutace, kdy jsou u potomků náhodně změněny náhodně vybrané geny. Jelich počet je dalším parametrem a je udávám v procentech. Noví jedinci jsou ohodnoceni fitness funkcí. Nakonec je vybrána nová populace, která bude na vstupu další generace. Schéma evolučního algoritmu je na obrázku 2.1.

Nyní si blíže vysvětlíme jednotlivé kroky. Počáteční populace, ohodnocení populace a ukončovací podmínka byla vysvětlena v předchozích odstavcích.

2.1.1 Rekombinace (křížení)

Křížení si rozdělíme na dvě části. První je výběr rodičů, protože v populaci bývá většinou více než dva jedinci, je potřeba z nich právě dva vybrat. Druhou částí bude samotné provedení křížení. Existují dva základní typy křížení.

Základní typy pro výběr rodičů:

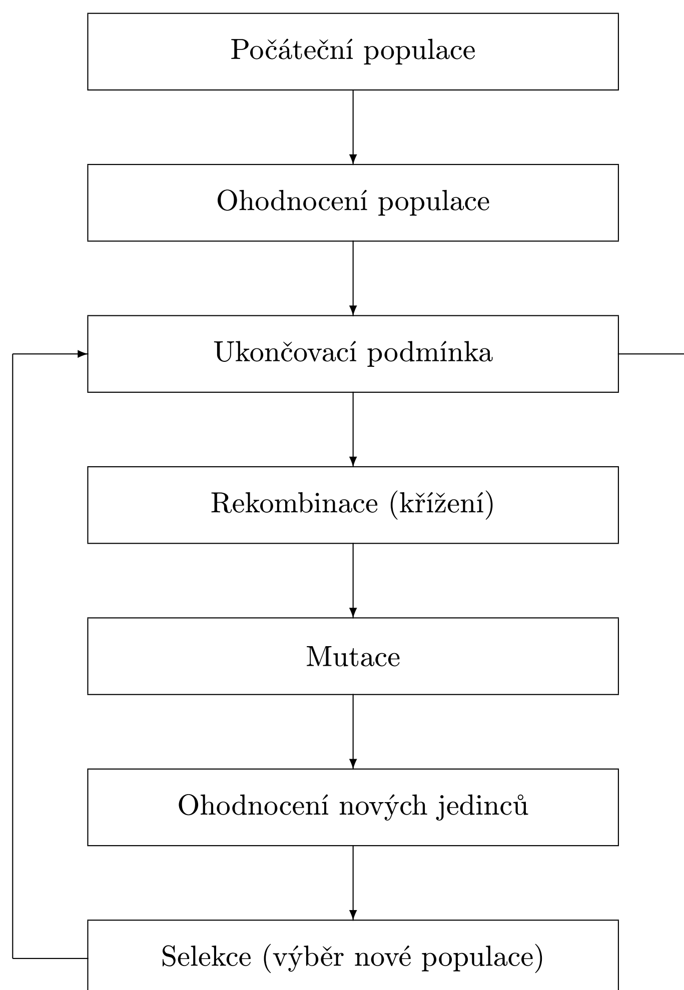
- **Ruleta** – jedná se o přístup, kdy každý jedinec má určitou pravděpodobnost, že se stane rodičem. Tato pravděpodobnost je odvozena od jeho ohodnocení. Neboli je přímo úměrná jeho fitness hodnotě tzn. čím vyšší fitness, tím vyšší pravděpodobnost výběru. Také ji můžeme vyjádřit následujícím vzorcem:

$$p_i = \frac{fitness(i)}{\sum fitness}$$

  Můžeme si představit tuto metodu jako kruh rozdělený na kruhové výseče, jejichž velikost odpovídá pravděpodobnosti výběru. Toto kolo roztočíme jako ruletu a na kterém místě se zastaví, toho jedince vezmeme. Je zřejmé, že největší šanci má jedinec s nejlepší fitness.

- **Uspořádání** – jak již název napovídá nedříve uspořádáme podle hodnoty fitness. Pravděpodobnosti výběru pak lineárně nebo exponenciálně rozdělíme tak, aby nejvíce měl nejkvalitnější jedninec a nejméně i nula měl nejhorší. Oproti ruletě rovnoměrněji rozloží pravděpodobnost výběru jedinců.

- **Turnaj** – náhodně vybereme vždy dva jedince z populace do turnaje. Turnaj vyhraje ten kvalitnější z nich. Tento postup má podobné výsledky jako předchozí a jeho přínosem je vyhnutí se setřiďování populace podle fitness. Proto je tento přístup často využíván.

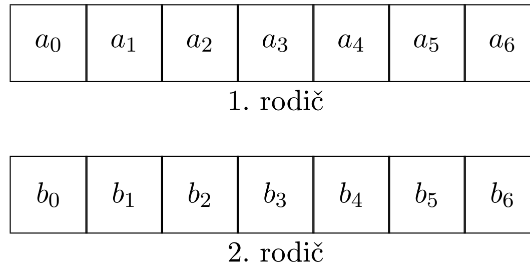


Obrázek 2.1: Schéma evolučního algoritmu.

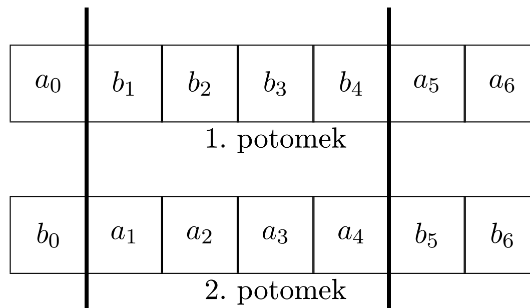
Protože evoluční algoritmy nespecifikují jednotlivé genetické operace, ukáži možnost křížení na příkladu genetických algoritmů, kde jsou dvě následující možnosti:

- **Jednobodové a vícebodové křížení** – náhodně vybereme bod nebo více bodů mezi geny na chromozomu. Poté do nového potomka bereme geny vždy z jednoho rodiče, dokud nedojdeme k prvnímu bodu křížení. Poté pokračujeme a vybíráme geny ze druhého rodiče až k dalšímu bodu křížení. Takto pokračujeme dokud nedorazíme na konec chromozomu. Tímto způsobem můžeme vytvořit dva nové potomky, pokud u prvního začnene s výběrem genů z prvního rodiče a u druhého naopak z druhého rodiče. Na obrázku 2.3 je znázorněn výsledek dvoubodového křížení, kde body křížení jsou mezi nultým a prvním genem a druhý je mezi čtvrtým a pátým genem. Rodičovské geny jsou znázorněny na obrázku 2.2.

- **Uniformní křížení** – je přístup, kdy pro každý gen nového jedince vybíráme náhodně, ze kterého rodiče jej vybereme. Takto může vzniknout i více různých jedinců ze dvou rodičů. Tento přístup sice rozbíjí ucelené celky, ale může být výhodný pro

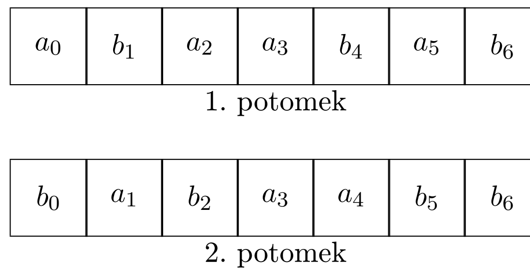


Obrázek 2.2: Rodičovské chromozomy.



Obrázek 2.3: Ukázka dvoubodového křížení.

vícerozměrné problémy s množstvím lokálních extrémů tím, že zabraňuje předčasné konvergenci. Ukazka uniformního křížení je na obrázku 2.4, rodičovské chromozomy jsou na obrázku 2.2.



Obrázek 2.4: Ukázka uniformního křížení.

2.1.2 Mutace

Mutace je náhodná změna náhodně vybraných genů. Vnáší do populace něco nového, co nelze získat křížením jedinců z populace. K mutaci zpravidla dochází v setinách až malých jednotkách procenta. Nejtypičtější je negace vybraného bitu, ale některé metody vyžadují speciální operátory mutace. Ukázka zmutování třetího genu 1. rodiče z obrázku 2.2 je na obrázku 2.5.

a_0	a_1	a_2	$\neg a_3$	a_4	a_5	a_6

Obrázek 2.5: Ukázka mutace třetího genu.

2.1.3 Výběr nové populace (selekce)

Když jsou vytvořeni noví jedinci, je potřeba vybrat ty, kteří vstoupí do další generace. K tomu nám slouží dva základní mechanismy:

- Prvním je úplná obnova populace, tzv. „vymírání rodičů“, kdy do nové populace jsou zahrnuti pouze nově vzniklí potomci.
- Druhou možností je částečná obnova, kdy jsou do nové populace zahrnuti všichni rodiče a pouze jeden nejméně kvalitní je nahrazen nejkvalitnějším potomkem.

Protože obě tyto možnosti jsou extrémní, často se využívá jejich kombinace. Ta spočívá v tom, že se nahraruje pouze určité procento nejméně kvalitních jedinců původní populace nejkvalitnějšími nově vytvořenými potomky. Toto procento se odvykle pohybuje v rozmezí 20 % až 50 %.

2.2 Genetické programování

Genetické programování je podtřídou evolučních algoritmů. Jeho hlavní zvláštností je, že fenotypen je spustitelný program. O jeho zrod a následné rozšíření se zasadil John Koza.

Ovšem generování spustitelných struktur má řadu specifik. Původní varianta pracovala nad stromovou strukturou programovacího jazyka LISP. Obnáši to například nekonstantní velikost chromozomu, aby bylo možné vytvářet různě rozsáhlé programy. Konstantní délka chromozomu by proto znamenala značné omezení. Také bylo potřeba zavést speciální operátory rekombinace (křížení) a mutace. Výpočet fitness funkce se provádí tím způsobem, že je spustitelná struktura provedena pro předem definovanou množimu vstupů. Následně jsou výsledky porovnány s referenčními výstupy pro daný typ úlohy. Fitness funkce je potom nepřímo úměrná sumě odchylek výstupů od referenčních výstupů.

Aby bylo možné chromozom dekódovat na spustitelný program, je potřeba, aby obsahoval terminály a funkce, jejichž množinu je potřeba specifikovat vždy při inicializaci. Je zřejmé, že výsledný program se bude skládat pouze z těchto komponent.

2.2.1 Terminály

Terminály slouží pro vstup hodnot do programu. Mohou jimi být jak proměnné tak i konstanty. Nebo i funkce bez parametrů, které vrací vygenerovanou hodnotu. Proměnnými do programu dostáváme vstupní hodnoty z trénovací množiny. Konstanty jsou z předem definovaného rozsahu, například podmnožina reálných čísel. Označení terminál je dáno reprezentací stromovou strukturou, protože se vyskytují v listech na pozici terminálních symbolů.

2.2.2 Funkce

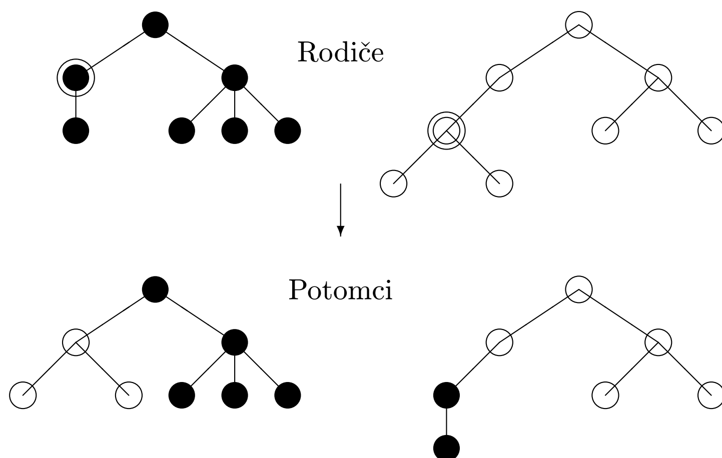
Funkcemi je myšleno cokoliv, co větví stromovou strukturu. Neboli klasické funkce s parametry a libovolné programové struktury z programovacích jazyků. Například podmínky,

cykly, skokové příkazy a další. Mezi funkce patří například logické funkce a aritmetické funkce. Zde si musíme dát pozor, abychom používali jen tzv. chráněné (protected) funkce. Jde například o dělení, které není definováno pro dělení nulou. Musí být proto upraveno tak, aby při dělení nulou vracelo nějakou specifickou konstantu, například buď nulu nebo co největší číslo.

Je potřeba volit funkce s rozvahou, aby odpovídaly hledanému řešení problému. Podobně jako u ostatních parametrů evolučních algoritmů, pokud bude velké množství různých funkcí, prohledávaný prostor značně naroste a s ním také čas. Naopak, pokud bude funkcí málo, algoritmus nemusí najít vhodné řešení.

2.2.3 Křížení

Křížení je specifické tím, že vybírá vždy dva uzly, kořeny podstromu. U každého předem vybraného jedince jeden uzel. Výběr jedinců probíhá standardně, jako je popsáno výše u evolučních algoritmů. Následně se zvolené dva podstromy u jedinců prohodí v místech uzlů a tím vzniknou noví dva potomci. Existují také rozšíření, kdy klademe na vybírání uzlů stromu omezující podmínky, jako například, že uzly s terminálními symboly se vybírají s menší pravděpodobností. Příklad takového křížení je na obrázku 2.6.



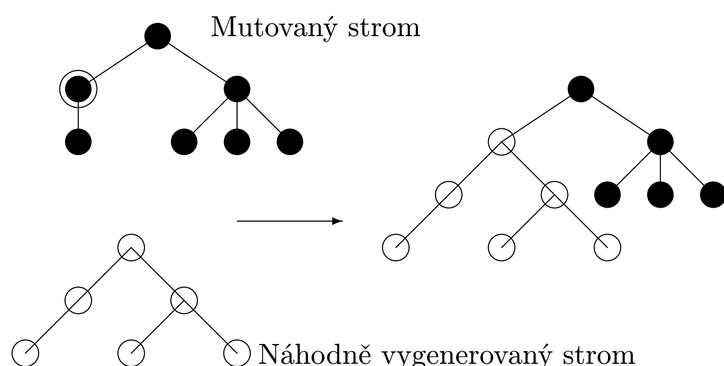
Obrázek 2.6: Ukázka křížení genetického programování.

2.2.4 Mutace

Mutace probíhá obdobně jako u klasického evolučního algoritmu. Jediným rozdílem je, že vybíráme uzel stromu, který chceme zmutovat, a místo negace vygenerujeme náhodně nový podstrom, stejně jako při generování počáteční populace. Ten napojíme do stromu v místě vybraného uzlu. Mutace obvykle probíhá s 5% pravděpodobností. Ukázka mutace je na obrázku 2.7.

2.2.5 Shrnutí genetického programování

Genetické programování je zajímavá metoda, ale zatím nepřináší úspěchy na složitých problémech v rozumném čase. Je to dáno následujícími faktory.



Obrázek 2.7: Ukázka mutace genetického programování.

Při generování kódu dochází k vytvoření intronů. Introny jsou části kódu programu, který je sice sémanticky správný, ale neprovádí užitečnou funkčnost. Příkladem je výraz $x = x + 0$. Takových a podobných výrazů se může vytvořit velké množství a způsobují prodlužování času při vyhodnocování programů. Jedinou pozitivní vlastností je, že chrání užitečný kód před genetickými operátory.

Škálovatelnost také není optimální, protože rozsáhlé problémy vedou na dlouhé chromozomy. Ty pak způsobují rozšíření prohledávaného prostoru a s tím spojený nárůst času ve všech etapách evoluce.

Asi poslední problém, který může nastat, je přetrénování (overfitting), které spočívá v dokonálém naučení na trénovacích datech, ale neschopnosti generalizovat na obecná data.

2.3 Kartézské genetické programování (CGP)

Kartézké genetické programování [4], jak už název napovídá, je modifikace genetického programování, kdy je program místo do stromu uspořádán do acyklického grafu. Tento graf je tvořen na podmnožině prostoru kartézské mřížky uzlů. Každý uzel reprezentuje určitou funkci. Pomocí genetického algoritmu se vyvíjí funkce v uzlech a propojení uzlů do grafu.

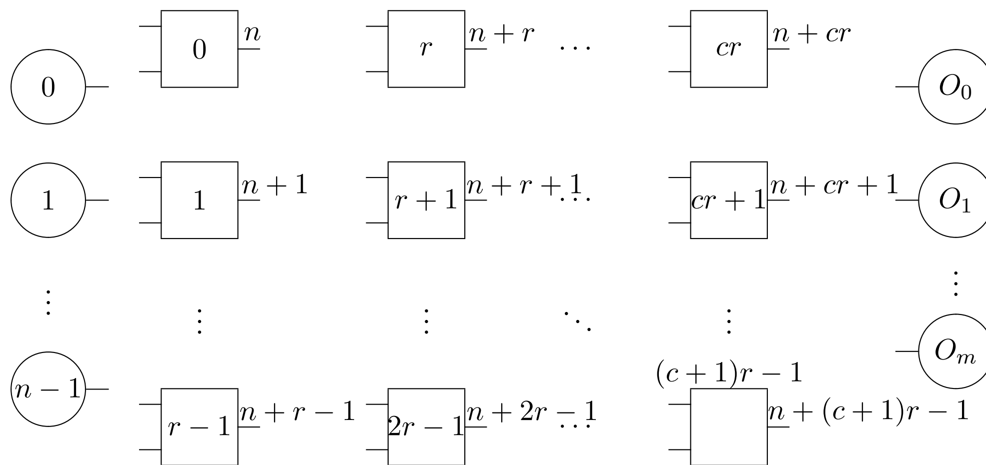
CGP má jednu specifickou, ovšem významnou zvláštnost a to, že nemá definován operátor rekombinace (křížení), tudíž využívá pouze mutaci.

Dále se budeme zabývat využitím CGP při návrhu kombinačních obvodů [6], proto si neukážeme základní verzi, ale mírnou úpravu. Například pro návrh obvodů se počítá s funkcemi o dvou vstupech, naproti tomu obecně je tento počet libovolný.

Schéma CGP je na obrázku 2.8, kružnice vlevo označují vstupy, kružnice vpravo výstupy. Čtverce znázorňují jednotlivé funkce. Každý kartézský program má následující parametry:

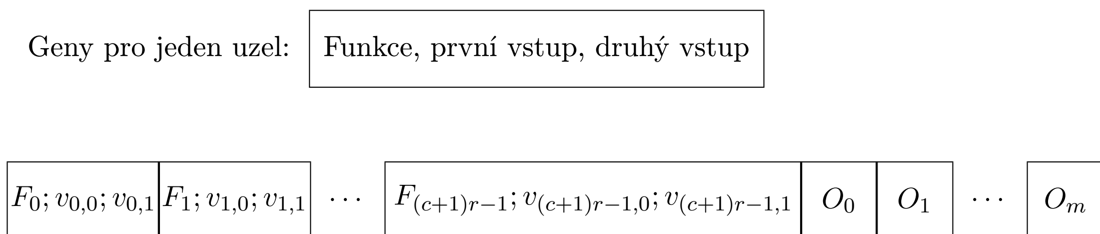
- n počet vstupů (parametrů) kartézského programu. Pro každou přípustnou kombinaci vstupů musíme mít nadefinován očekávaný výsledek.
- m počet výstupů kartézského programu.
- r počet řádků mřížky s funkcemi.
- c počet sloupců mřížky s funkcemi. Tento počet a počet řádků nám ovlivňuje velikost prohledávaného prostoru.

- i počet vstupů funkcí. Obecně je libovolnou konstantou, ale při návrhu obvodů je roven dvěma. Je to dáno tím, že základní hradla jsou dvouvstupá (AND, OR, XOR, NAND, ...). Výjimku tvoří hradlo negace NOT, tudíž můžeme brát v úvahu například jen první vstup a s druhým nepočítat.
- Γ množina funkcí je také parametr, který zásadně ovlivňuje prohledávaný prostor. Je třeba volit přiměřenou množinu funkcí, aby pokryly řešenou problematiku. Při návrhu obvodů se využívají funkce, které lze složit z tranzistorů a vypálit na plochu čipu.
- L-back parametr nám udává, z kolika předchozích sloupců můžeme připojovat výstupy na vstupy funkcí v daném sloupci. Vždy můžeme připojit vstupy funkcí na vstupy programu. Pro L-back roven jedné můžeme vyhodnocovat program proudově, protože v každém kroku budeme mít na funkcích v každém sloupci oddělená data.



Obrázek 2.8: Schéma CGP.

Chromozon kartézského programu je na obrázku 2.9. Jedná se o posloupnost kladných celých čísel. Obsahuje výběr funkce z definované množiny povolených funkcí pro každý uzel a výběr všech vstupů uzlů. Tím je myšleno číslo výstupu, který se přiřadí na vstup. Samozřejmě výběr musí splňovat podmínku L-back parametru. A také definuje připojení výstupů programu na výstupy uzlů. Pro každý uzel jsou v chromozomu vyhrazeny tři geny a pro každý výstup programu jeden gen. Délka chromozomu je konstantní, lze vyjádřit výrazem: $cr(i+1)+m$, kde označení odpovídá předchozímu popisu.



Obrázek 2.9: Chromozom CGP.

2.3.1 Vývoj v CGP

Jelikož na rozdíl od klasického genetického programu CGP nepodporuje křížení, je zde několik rozdílů při vývoji. Prvním je, že do každé generace evoluce vstupuje vždy jen jeden jedinec. Znamená to, že na začátku se náhodně vygeneruje jen jeden jedinec nebo více a poté se z nich vybere jen ten nejkvalitnější. Samozřejmě je možnost vyrobit jedince na míru úloze, pokud například už máme řešení a chceme najít lepší. V každé generaci se z rodičovského chromozomu generuje pomocí mutace určitý počet potomků, tento počet je vstupním parametrem. Pokud je potomek kvalitnější než jeho rodič, nahradí jej pro další výpočet.

Konstantní velikost chromozomu má výhodu při využití paměti počítače, protože ji můžeme předem zadefinovat a už se nezmění. Také vede ke vzniku intronů, na nichž dochází k tzv. neutrálním mutacím. To jsou mutace, které nevedou ke změně fenotypu. Ovšem jsou důležité, protože v další generaci může dojít k mutaci, která takto změněnou část kartézské mřížky připojí. Není vyloučeno, že právě zde bude nalezeno kvalitnější řešení. Proto i změněný jedinec se stejnou fitness hodnotou je považován za kvalitnějšího než jeho rodič.

2.3.2 Mutace v CGP

Mutace je jediný genetický operátor v CGP. Jsou na ni kladeny některá omezení. Můžeme mutovat libovolný gen chromozomu, tedy jak propojení, tak i funkci jednotlivých uzlů. Při mutování funkce je náhodně vybráno číslo, které odpovídá funkci. Jednotlivé funkce jsou číslovány od nuly. Při mutování propojení kartézské mřížky musíme vygenerovat číslo, které není v rozporu s L-back podmínkou. Pravděpodobnost mutace je obvykle 5 %.

Existuje i varianta, kdy mutaci na chromozomu provádíme tak dlouho, dokud se nám neprojeví na změně fenotypu jedince. Jinými slovy, pokud mutace jednoho genu byla neutrální, nezměnila fenotyp jedince, vybereme další gen a ten také zmutujeme. Takto pokračujeme, dokud nezískáme jedince s jinou funkcionalitou. Tento způsob je výhodný, protože nám ubude jeden parametr, pravděpodobnost mutace, a také se nemusíme starat o neutrální mutace, protože každý nový jedinec bude mít jinou fitness hodnotu.

2.3.3 Fitness v CGP při návrhu obvodů

Při návrhu odvodů klademe důraz především na korektnost výstupů. Ovšem hned v zápětí chceme odvod s minimální plochou na čipu, s minimální spotřebou a minimálním zpožděním. Je zřejmé, že tyto podmínky jsou protichůdné. Proto v rámci fitness funkce je potřeba, aby zohledňovala náš záměr. Typicky chceme nejdříve najít funkční obvod, pokud jsme jej nezadali jako výchozího rodiče, a poté jej optimalizovat. Uspokojení všech podmínek vede na multikriteriální problém.

Kapitola 3

Počítačové clustery, paralelní programování

Počítačové clustery [9] jsou složené z velkého množství obvykle stejných počítačů propojených vysokorychlostní komunikační sítí. Byly sestaveny z důvodů počítání rozsáhlých problémů, na které už nestačila kapacita jednoho stroje. Je zřejmé, že každý stroj potřebuje také program, podle kterého bude fungovat. Ovšem programy na běžné počítače nelze efektivně použít na clustrech. Pro jejich využití je potřeba provést paralelizaci. Ovšem to může znamenat od doplnění kódu až po kompletní nový návrh programu, aby byla co nejlépe využita výpočetní síla počítačového clusteru. S návrhem a implemetací těchto programů je spojen pojem paralelní programování.

V celku se jedná o přístup MIMD (Multiple Instruction, Multiple Data), což znamená, že ve stejnou chvíli běží množství programů nad množstvím dat. U klasického počítače hovoříme o SISD (Single Instruction, Single Data), neboli v jeden okamžik počítá právě jeden program nad danými daty. Nebo jsme schopni provádění například zvektorizovat a pak mluvíme o SIMD (Single Instruction, Multiple Data). Postupně provádíme stejný program na různých datech. Počítačový cluster propojením více SIMD vytvoří, jak už bylo řečeno, MIMD. Na clusteru typicky běží současně větší množství úloh nad různými daty.

Na počítačovém clusteru se výpočty provádí pomocí úloh. Každá úlohá má program, který bude vykonávat, data, se kterými bude pracovat, a také požadavky na zdroje a potřebný čas. Taková úloha pak čeká ve frontě úloh na uvolnění potřebných zdrojů. Když jsou zdroje volné a je na řadě, je nahrána na příslušné počítače a poté se začne vykonávat. Je zde zavedena politika, která určuje pořadí ve frontě úloh. Je možné mít i více front úloh a také záleží na požadavcích na zdroje a čas. Pokud by úloha neskončila v předem deklarovaném čase, je ukončena a zdroje jsou uvolněny pro další úlohu. Zdroji je myšleno, na kolika počítačích, procesorech a vláknech bude výpočet proveden. Je zřejmé, že je musí mít programátor přesně spočítané předem.

Právě výpočet rozdělený na úlohy je pro clustery specifický. Na rozdíl od běžného počítače, kde máme neustále přístup k programu, výstupům atd., u úloh je to jiné. Po vytvoření úlohy se všemi vstupy už nemůžeme do výpočtu zasahovat. Vše řídíme z čelního uzlu, což je přípojné místo clusteru, ke kterému se dostaneme. Na něm připravujeme úlohy se všemi parametry a také do něj dostaneme výsledky po skončení úlohy.

3.1 Paralelní programování

Paralelní programování dělíme na dva následující přístupy podle architektury systému. Jedním je systém se sdílenou pamětí (Shared Memory), druhým je model se zasíláním zpráv (Message Passing). Každému modelu odpovídá určitá specifikace, přitom nejvýznamnější jsou následující dvě. První je OpenMP, která slouží pro práci s vlákny jádra procesoru. Druhou je MPI, která má několik implementací od různých tvůrců. MPI specifikuje komunikaci mezi procesory pomocí zasílání zpráv. Obě tyto specifikace jsou důležité pro paralelní programování, které je nezbytné pro výpočet na počítačovém clusteru.

3.1.1 OpenMP

OpenMP [8] (Multi-Processing) je založeno na paralelním provádění výpočtu pomoci více vláken procesoru současně. Pracuje na systémech se sdílenou pamětí. Podporuje programovací jazyky C, C++ a Fortran. Programátorovi k použití slouží direktivy pro překladač `#pragma` ..., které dále specifikují potřebnou funkcionalitu. Typickým úkolem je zparalelizování cyklů, kdy každé vlákno provádí jinou iteraci cyklu. Je potřeba zajistit, aby jednotlivé iterace byly na sobě nezávislé. Dále specifikuje možnosti synchronizace vláken pomocí například bariér. Také umožňuje obsluhu kritické sekce, kdy danou část programu může provádět právě jen jedno vlákno současně, tzn. neumožnit souběžné provádění. Nebo je možné zajistit provádění úseků kódu sekvenčně i v paralelní části programu.

Výhodou je možnost použití již dříve napsaných programů. Stačí nám na vhodná místa doplnit příslušné direktivy a program tak paralelizovat. Ovšem nemusíme dosáhnou takového urychlení, jako kdybychom vytvářeli program od začátku a již při návrhu počítali s možností paralelizace.

3.1.2 MPI

MPI (Message-Passing Interface) je založené na posílání zpráv mezi procesory. Znamená to, že se zabývá systémy bez sdílené paměti, kde právě posílání zpráv je jedinou možností komunikace.

Základní funkce jsou odeslání (send) a příjem (receive) zpráv. U obou existuje několik variant jako například blokující a neblokující nebo synchronizující varianta a jiné. Každá z možností se hodí pro jiný přístup a je možné i kombinavat různé odesílací a příjímací varianty. Pokud chceme zabránit uváznutí (deadlock) je dobré využít neblokující variantu.

Dále MPI specifikuje i řadu kolektivních komunikačních funkcí. Těmi jsou rozeslání zprávy všem (broadcast), rozdělení dat od jednoho všem (scatter), naopak shromáždění dat od všech (gather), redukce a jiné, u kterých praxe ukázala, že jsou potřebné.

MPI také specifikuje komunikátory. Komunikátor seskupuje jednotlivé procesory a umožňuje jim mezi sebou posílat zprávy. Procesor je vždy v globálním komunikátoru, ale může být i v několika dalších. Vytvoření dalších je vhodné pro kolektivní komunikace, kdy ji chceme provést jen na podmnožině procesorů. Dále mohou komunikátory simulovat fyzické propojení procesorů do konkrétní topologie.

MPI dále specifikuje datové typy, které využívá při zasílání zpráv. Jsou obdobné jako datové typy jazyka C.

Oproti OpenMP, zde není možné stávající program doplnit tak, aby fungoval na více procesorech. Je potřeba počítat s paralelizací pomocí MPI už při návrhu programu. Navíc všechny procesory pracují se stejným programem. Každý procesor zjistí svoje pořadové číslo

v komunikátoru zavoláním příslušné funkce. Poté se může identifikovat pomocí tohoto čísla a zjistit tak, zda má například odesílat nebo příjímat data.

Kapitola 4

Evoluční návrh obvodů na clusteru

Evoluční algoritmy jsou pro složitější problémy prostorově i časově náročné, protože musí prohledat rozsáhlý stavový prostor, což zabere značný čas. Proto se nabízí využití výpočetní síly počítačového clusteru. Avšak klasický genetický program, který běží v jednotlivých na sobě závislých iteracích, není možné vzít tak, jak je, a použít na clusteru. Protože pro návrh obvodu se nejvíce hodí CGP, budu nadále počítat s ním.

Existuje přístup mezi evolučními algoritmy, který se nazývá Ostrovní modely [7] (Island Models). Tento přístup je založen na souběžném běhu evolučních algoritmů na jednotlivých „ostrovech“, které jsou vzájemně izolované. To odpovídá i architektuře počítačového clusteru, pokud připustíme, že ostrov odpovídá jednotlivému počítači v clusteru.

4.1 Ostrovní modely

Základem je předem definovaný počet ostrovů, což je další parametr evolučního algoritmu. Na každém ostrově se nezávisle na sobě vyvíjejí jedinci odpovídající jednotlivým řešením. Ale protože může nastat situace, kdy by některý z ostrovů mohl uváznout v lokálním extrému, je potřeba jedince na tomto ostrově zrekombinovat s nějakým jiným výhodným jedincem. Takového jedince najdeme na jiném ostrově. Existuje množství přístupů, jak zajistit výměnu jedinců mezi ostrovy. Ideální případ by byl, kdyby vždy, když je jeden z ostrovů uvázlý v lokálním extrému, tak by dostal nejlepší jedince z ostatních ostrovů, s nimi se rekombinoval a pokračovalo se dál. Ovšem tento přístup je složitý z pohledu rozeznání, že ostrov uvázl, dále výběru nejkvalitnějšího jedince atd. Navíc může být vhodné kombinovat jedince z různých ostrovů, i když zrovna neuvázli v lokálním extrému. Proto je možné specifikovat konkrétní čas, kdy se mají jedinci rozeslat mezi ostrovy k rekombinaci.

Čas, kdy jedince budeme posílat na jiné ostrovy, můžeme specifikovat například počtem generací. Znamená to, že zavedeme další parametr, podle kterého se určí, po kolika generacích se vždy ostrovy rozhodnou přeposlat jedince na okolní ostrovy. Ale zde se může zbytečně plýtvat časem, protože na jednotlivých ostrovech může stejný počet generací trvat různě dlouho. Tomu by šlo předejít, pokud bychom čas přeposílání jedinců specifikovali pomocí reálného času a nezáleželo by, kolik se stihlo generací. Tím bychom zajistili, že se všechem přidělený čas využije na výpočet a ne na čekání na ostatní.

Dále je zde několik způsobů, jak vyměňovat jedince. Jestli se mají jedinci z jednotlivých ostrovů rozeslat na všechny ostatní ostrovy nebo jen na sousední ostrovy. Sousední ostrovy jsou myšleny z hledista použité topologie k uspořádání počítačů v clusteru. Podle implementované topologie mají uzly buď všechny stejný počet sousedů nebo různý podle jejich

umístění.

Každý ostrov po přijetí nových jedinců utvoří nové potomky jejich rekombinací se svými jedinci. Tím se do populace na každém ostrově dostávají nové informace, které pomohou k rychlejší konvergenci. Následuje další období samostatného vývoje v rámci ostrova, než nastane čas pro další spolupráci mezi ostrovy při výměně jedinců.

Výpočet končí nalezením řešení na kterémkoliv ostrově nebo po vyčerpání maximálního množství vymezaného času. Čas může být specifikovám například počtem generací nebo reálným časem.

4.1.1 Ostrovní modely s CGP

Ostrovní modely s CGP [2] fungují standardně tak, že na každém ostrově běží samostatné CGP. Avšak je zde jedna změna a to, že CGP nepodporuje rekombinaci. Proto v místě, kdy si ostrovy předávají nejlepší jedince, nelze provést rekombinaci přijatého a stávajícího nejkvalitnějšího jedince. Proto se pouze porovná jejich kvalita a pro další výpočet se bere ten kvalitnější z nich. Následuje opět izolovaný výpočet na každém ostrově do dalšího bodu rozeslání nejlepších jedinců.

4.2 Návrh vylepšení

U ostrovních modelů s CGP se nevyužije celý potenciál, který metoda poskytuje. Je to dáno tím, že není možné zrekombinovat dva jedince. Proto se chci dále v rámci práce zabývat implementací a otestováním následujícího modelu rekombinace, který bude aplikovám právě při distribuci nejlepších jedinců mezi ostrovy.

Rekombinace bude založena na výběru dvou propojovacích hran uzlů CGP grafu, každá u jednoho jedince. Následně se vzájemně vymění počáteční body těchto hran a tím vzniknou dva noví jedinci. Dále bude potřeba uspořádat jejich chromozomy tak, aby obsahovaly všechny nové aktivní uzly. Mohla by totiž nastat situace, kdy bude mít nový jedinec více aktivních uzlů, než jsme schopni do konstantní délky chromozomu zakódovat. Pak už by se nejednalo o korektního jedince a nemohli bychom s ním dále počítat.

Budu počítat s tzv. lineárním CGP programem. Znamená to, že kartézská mřížka má pouze jeden řádek a počet sloupců záleží na potřebách daného problému. Tento model je běžně využíván pro výpočet.

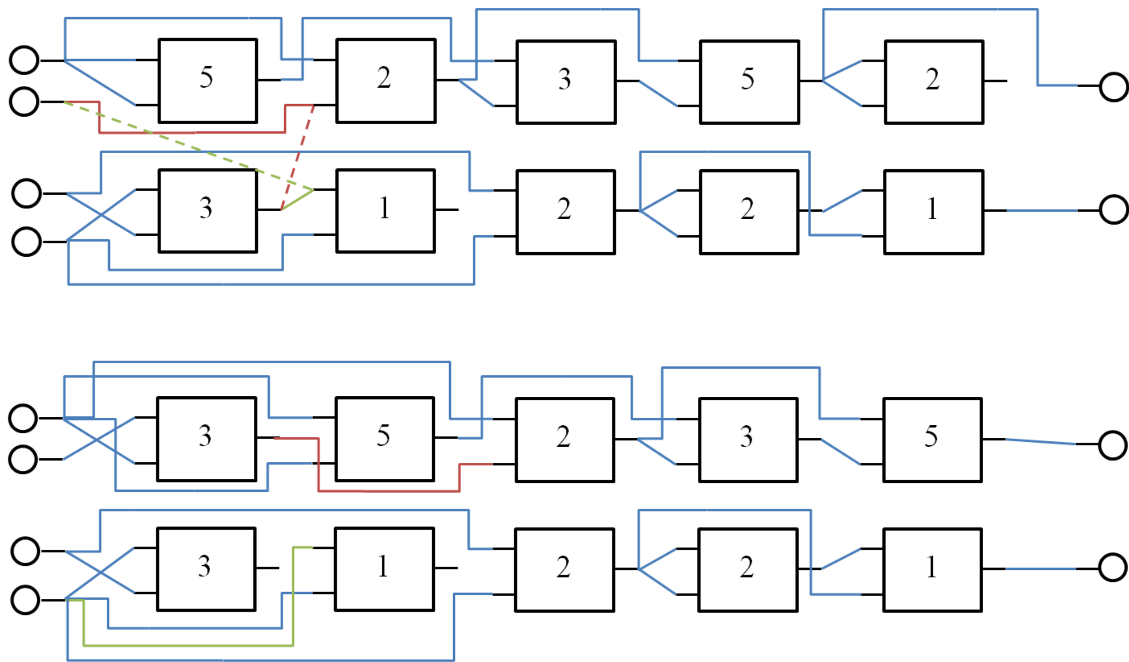
Na obrázku 4.1 je příklad, jak bude mnou navržená rekombinace vypadat na pětisloupcovém lineárním CGP. Ve vrchní části jsou zobrazeny dva rodičovští jedinci, jejichž chomozomy jsou následující:

```
(5,0,0)(2,0,1)(3,2,3)(5,3,4)(2,5,5)(5)

(3,1,0)(1,2,1)(2,0,1)(2,4,4)(1,5,4)(6)
```

Barevně jsou označeny vybrané hrany k rekombinaci a přerušovanou barevnou čarou nové hrany po rekombinaci. Dále jsou na obrázku noví potomci, jejichž chomozomy jsou:

```
(3,1,0)(5,0,0)(2,0,2)(3,3,4)(5,4,5)(6)

(3,1,0)(1,1,1)(2,0,1)(2,4,4)(1,5,4)(6)
```

Je vidět, že druhý potomek se změnil jen mírně, tato změna odpovídá i mutaci. Ale první potomek je změněn podstatně, protože je zde přidán nový uzel na začátek a zbytek uzlů je



Obrázek 4.1: Rekombinace lineárního CGP.

posunut. Nový uzel na začátku je zkopírovám ze druhého rodiče. Dále je na chromozomech vidět, že před místem rekombinace jsou čísla vstupů nezměněná, kdežto za místem křížení se posunula o jedna. Je to dáno právě vložením nového uzlu. Ovšem pokud bychom křížili větší jedince, tento problém se rozroste. Přidaných uzlů bude více, bude je potřeba začlenit na různá místa tak, aby nebyly narušeny podmínky, kdy hrany musí směřovat vždy z předchozích uzlů a nikoliv z budoucích. Také bude potřeba zajistit splnění L-back podmínky. Jinak může nastat situace, kdy uzly, které byly aktivní před rekombinací, po ní už aktivní nebudou a následně je bude možné například nahradit novými uzly, které jsou od druhého rodiče a je potřeba je začlenit do chromozomu.

4.3 Implementace původní varianty

Vyšel jsem z implementace klasických ostrovních modelů [2], která zajišťuje návrh kombinačních odvodů pomocí ostrovních modelů s CGP. Ostrovy jsou zde vytvořeny pomocí MPI procesorů, což zajistí jejich nezávislé paralelní provádění, pokud máme dostatečný hardware. Komunikace mezi ostrovy je prováděna pomocí zasílání zpráv a kolektivní komunikace. Tím je také zajištěna synchromizace, protože se komunikuje po určeném počtu generací, jejiž vykonání nemusí skončit současně. Na každém ostrově běží klasický CGP program. Jeho provádění lze také urychlit pomocí vláken OpenMP při vyhodnocování nové populace. Jedince nové populace můžeme počítat nezávisle na ostatních, tudíž každého v jednom vlákně. Tím je zajištěno paralelní vyhodnocení fitness funkce, jejiž časová náročnost je právě tím důvodem k urychlení.

4.4 Implementace vylepšení

Původní implementaci jsem dále rozšířil o možnost křížení v době, kdy si ostrovy původně pouze rozdistribuovaly globálního nejlepšího jedince a nahradily jím svého nejlepšího jedince. Po prostudování fungování programu jsem mu doplnil možnost zadat parametr pro variantu křížení a také jeho zpracování. Dále jsem přidal další větev, která jej bude obsluhovat. Pro křížení jsem vytvořil modul s funkcí `chrom_crossover`, která jej zajistí. Jako parametry přijímá ukazatel na parametry CGP, ukazatele na dva rodičovské chromozomy, ve druhém chromozomu zároveň vrací nového potomka, a dále ukazatele na pole potřebné pro vyhodnocení fitness funkce. Těmi jsou požadované výstupy, maska těchto výstupů a pole pro ukládání hodnot výstupů z jednotlivých uzlů mřížky CGP. Původní program jsem dále upravil tak, aby všechny tyto parametry tato nová funkce, která se volá místo zkopírování jednoho chromozomu za druhý, dostala. Funkce tedy získá globálního nejlepšího jedince a nejlepšího jedince svého ostrova.

Funkce pro svou činnost potřebuje pět chromozomů. Těmi jsou dva rodiče, které funkce dostala jako parametr. Třetí chromozom obsahuje nejlepšího jedince, kterým je ze začátku nejlepší globální jedinec. Ten je následně nahrazován novými lepšími jedinci vzešlými z křížení, pokud jsou takoví nalezeni. Poslední dva chromozomy jsou potřeba pro výpočet. Důvodem je uchování rodičovských chromozomů v původním stavu po celou dobu hledání nových jedinců křížením. Na začátku každé iterace jsou do nich zkopírováni rodičovské chromozomy a následně jsou v procesu křížení podle potřeby modifikováni.

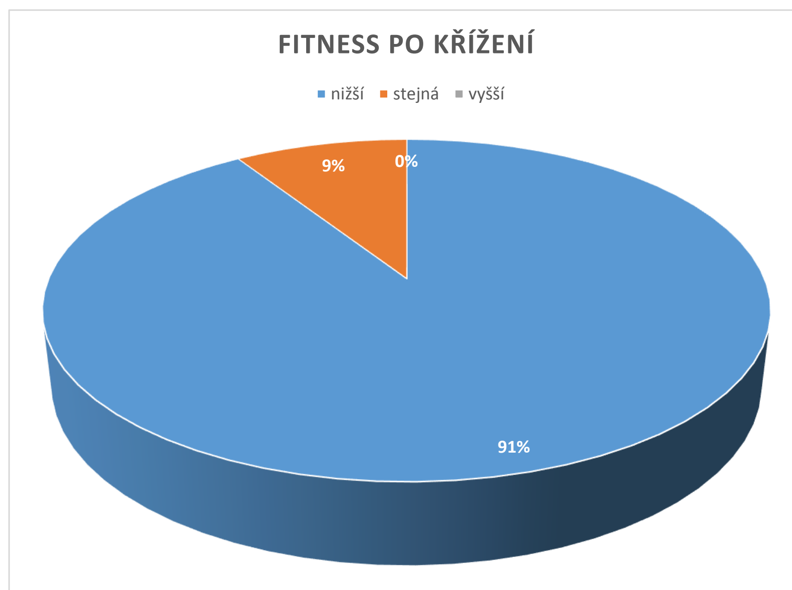
Předzpracování před křížením funguje následovně. Jsou náhodně vybrána místa ke křížení. Těmi jsou propoje mřížky CGP, neboli náhodný generátor čísel vygeneruje dvě čísla z rozsahu čísel těchto propojů. Následně je jeden z rodičů zvolen jako příjemce, druhý jako dárce. V dalším kole se prohodí, ovšem nechají si stejné, předem vybrané propoje ke křížení. U obou se zjistí aktivní uzly v případe, že příjemce se chová, jakoby v místě křížení neměl na vstupu už žádné předchozí uzly mřížky. Což je realizováno tak, že je toto místo připojeno na první vstup. Dárce se naopak chová tak, že má aktivní uzly pouze ty, které mají výstup v bodě křížení. To je zajištěno připojením všech výstupů chromozomu k výstupu uzlu, který se má křížit. Takto se zjistí aktivní uzly a také jejich počet. Pokud počet aktivních uzlů přesáhne počet uzlů mřížky CGP, dále se nepokračuje a vyzkouší se prohození dárce a příjemce nebo se hledají jiná místa ke křížení.

Následuje samotné křížení. Při něm začínáme na prvním aktivním uzlu dárce a prvním neaktivním uzlu příjemce. Ale pokud je místo křížení u příjemce ještě před prvním neaktivním uzlem, je potřeba provést posun aktivních uzlů. Posun je realizován tak, že je nalezen první neaktivní uzel po místě křížení. Dále se posunou jednotlivé aktivní uzly od místa křížení až po toto volné, neaktivní místo o jedno místo tak, že poslední z řady zabere volné místo atd. Kromě samotného posunu je potřeba upravit všechny ukazatele propojů tak, aby ukazovali na správné uzly po posunu a nezměnila se funkcionalita. Takto stačí upravit jen následující uzly po místě posunu včetně posunutých. Nyní, když máme volné místo před místem křížení u příjemce, vložíme na něj první aktivní uzel dárce. Následně upravíme zbylé aktivní uzly dárce, aby pokud ukazovaly na přidaný uzel, ukazovaly nyní na jeho novou pozici u příjemce. Takto se pokračuje, dokud má dárce nějaký aktivní uzel. Vždy, když není volný neaktivní uzel před místem křížení u příjemce, jsou aktivní uzly odsunuty, jak bylo popsáno výše. Nakonec je upravem vstup v místě křížení příjemce tak, aby ukazoval na poslední vložený uzel od dárce, což bylo jeho místo křížení.

Teď, když máme nového potomka, je potřeba spočítat jeho fitness hodnotu. K tomu je využita funkce z původní implementace, které jako paramerty potřebuje strukturu s chro-

mozomem jedince, očekávané výstupy s jejich maskou a také pole pro uložení výstupů z jednotlivých uzlů mřížky CGP. Ta je dále porovnána s hodnotou zatím nejlepšího jedince a pokud je stejná nebo vyšší, nový jedinec jej nahrádí a stává se novým maximem.

V průběhu zkoušení funkčnosti implementace vyvstala otázka, kolik takových křížení provést. Proto jsem spočítal a do následujícího grafu 4.2 zobrazil, v kolika procentech je nalezen jedinec s vyšší, stejnou a nižší fitness hodnotou než nejlepší jedinec. Počítal jsem



Obrázek 4.2: Procentuální úspěšnost nového křížení

s křížením vždy po 1000 generacích evoluce, každé křížení proběhlo 1000 krát a bylo provedeno na 10 procesorech pro více než 500 000 generací. Z výsledků je zřejmé, že lepšího jedince je prakticky nemožné nalézt, ale je možné získat stejně dobrého jiného jedince. Zvolil jsem proto maximální počet pokusů na křížení roven 100 a pokud je nalezen stejně dobrý nebo lepší nový jedinec, je křížení ukončeno. Nenalezení lepšího jedince nám nemusí vadit, protože cílem křížení je dostat do ostrovů diverzitu, která by mohla vést k rychlejší konvergenci při řešení pomocí evolučního algoritmu.

Touto implementací zanedbávám L-Back podmínku, protože uspořádat chromozon tak, aby ji splňoval, by bylo velice náročné, pokud by se to vůbec podařilo uskutečnit.

Kapitola 5

Testování

Pro testování implementace jsem využíval počítačový cluster Anselm [3], který je součástí národního superpočítačového centra IT4Inovations. Anselm má 209 uzlů, každý s 16 procesovy (2 krát 8 jader) a minimálně 64 GB paměti RAM, což odpovídá 4 GB na jádro. Využíval jsem část clusteru se 180 uzly s 2x Intel Sandy Bridge E5-2665, 2,4 GHz, protože zbytek uzlů je s GPU nebo MIC akcelerací, kterou bych nevyužil. Pro testování jsem vždy využíval jen přiměřený počet uzlů, který odpovídal náročnosti testované úlohy.

V rámci testování jsem porovnával svoje řešení se standardním řešením ostrovních modelů s CGP při návrhu kombinačních obvodů různě velkých sčítaček a násobiček. Zaměřil jsem se na:

- čas, za který se daný obvod navrhne. Což je nejdůležitější parametr, který budu sledovat. Zajímá mě především, která varianta bude rychlejší.
- úspěšnost, v kolika bězích se v daném počtu generací podaří nalézt řešení.
- funkčnost, procentuální shodu nalezeného řešení s referenčním. Tato shoda je určena z Hammingovy vzdálenosti mezi výstupy, které dává nalezené řešení a očekávanými výstupy.
- počet generací, které jsou potřeba pro nalezení řešení. Avšak je to spíše doplňkový údaj k potřebnému času, protoře předpokládám, že u křížení se bude stejný počet generací vyhodnocovat déle než v původní variantě.
- plochu, kterou by na čipu zabralo dané řešení. Což je ovšem jen informativní prvek, protože pracuji se záměrem najít co nejrychleji řešení v co nejméně generacích a už se nezabývám další optimalizací.

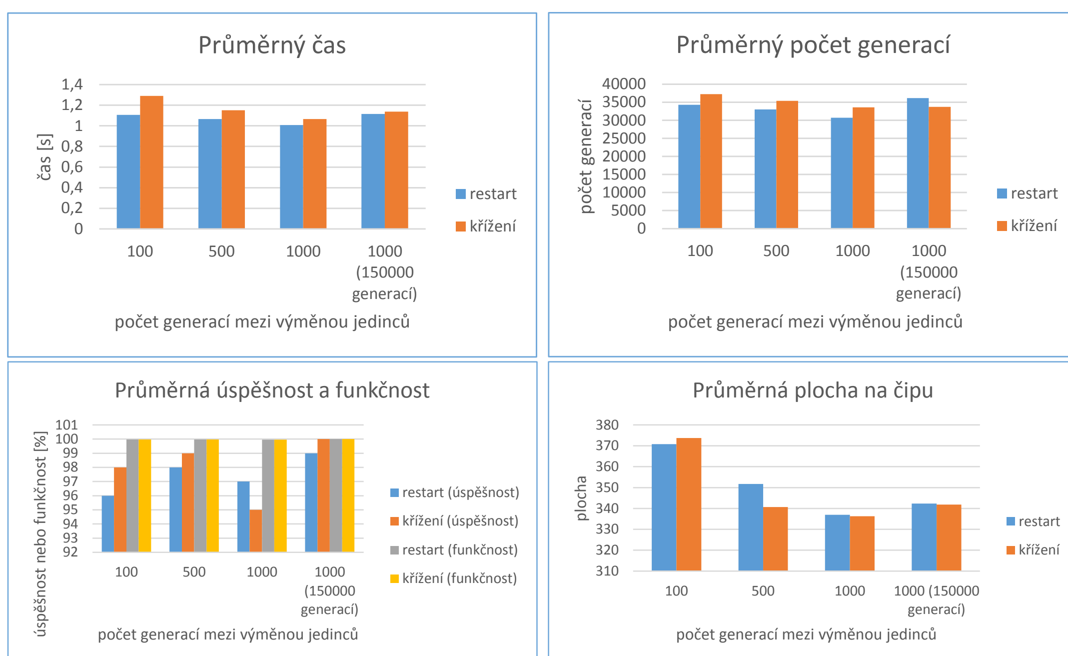
Každé nastavení budu testovat na 100 nezávislých bězích jak pro původní variantu, tak pro novou s křížením.

5.1 Čtyřbitová sčítačka

Čtyřbitovou sčítačku jsem testoval na jednom uzlu tzn. 16 procesorech, což odpovídá 16 ostrovům. Testoval jsem výměnu nejlepších potomků po 100, 500 a 1000 generacích. Maximální počet generací jsem stanovil na 100 000 a u výměny jedinců po 1000 generacích také na 150 000. Výsledky jsou shrnuty v tabulce 5.1 a v grafech na obrázku 5.1. Z nich vyplývá, že křížení se nejvíce vyplatilo pouze v posledním případě z hlediska času i počtu

výměna	typ		čas [s]	generace	hamming	funkčnost	úspěšnost	plocha
100	restart	min	0,11437	3700	0	97,81		148
		max	5,12592	150000	28	100		640
		avg	1,1055	34337	0,4	99,97	96	370,78
100	křížení	min	0,17245	2900	0	98,44		172
		max	4,97065	150000	20	100		600
		avg	1,29088	37278	0,2	99,98	98	373,72
500	restart	min	0,18063	6000	0	98,44		180
		max	5,24474	150000	20	100		714
		avg	1,06566	33045	0,2	99,98	98	351,7
500	křížení	min	0,20984	6000	0	98,44		156
		max	4,53179	150000	20	100		650
		avg	1,15136	35380	0,2	99,98	99	340,7
1000	restart	min	0,14885	5000	0	97,5		176
		max	4,42573	100000	32	100		560
		avg	1,00793	30740	0,5	99,96	97	336,98
1000	křížení	min	0,16842	5000	0	98,44		176
		max	3,28547	100000	20	100		596
		avg	1,06672	33630	0,5	99,96	95	336,24
1000	restart	min	0,2061	5000	0	99,77		168
150 000	generací	max	4,47855	150000	3	100		558
		avg	1,11588	36170	0	100	99	342,3
1000	křížení	min	0,15954	5000	0	100		194
150 000	generací	max	3,88753	114000	0	100		528
		avg	1,13661	33730	0	100	100	341,88

Tabulka 5.1: Výsledky čtyřbitové sčítačky



Obrázek 5.1: Grafy statistik při návrhu čtyřbitové sčítačky

generací. Ale kromě výměny po 100 generacích křížení našlo řešení v průměru s menší plochou na čipu. A také kromně výměny po 1000 generacích při maximálně 100 000 generacích našlo křížení řešení problému vícekrát než původní varianta. Avšak průměrná funkčnost je prakticky stejná u obou variant, těsně pod nebo stoprocentní.

5.2 Pětibitová sčítačka

Pětibitovou sčítačku jsem zkoušel navrhovat na jednom a dvou uzlech clusteru, což odpovídá 16 a 32 ostrovům modelu. Pro obě varianty jsem zkoušel výměnu nejlepších jedinců po 100, 500 a 1000 generacích. Maximální počet generací jsem stanovil na 200 000.

Z výsledků návrhu na jednom uzlu z tabulky 5.2 a grafů na obrázku 5.2 plyne, že křížení je lepší než původní varianta. Je to dáno průměrným nižším časem běhu, průměrným menším počtem potřebných generací a průměrně větší úspěšností. Pro průměrnou plochu na čipu už to tak jednoznačné není, protože pro výměnu po 100 a 1000 generacích bylo lepší křížení, ale pro výměnu po 500 generacích byla naopak lepší původní varianta. Avšak tento test je jen informativní, protože jsme se nezabývali optimalizací plochy na čipu. Obě metody našly prakticky průměrně stejně funkční výsledky.

Z výsledků na dvou uzlech clusteru z tabulky 5.3 a grafů na obrázku 5.3 vyplývá, že křížení není jednoznačně lepší, než na jednom uzlu clusteru. Průměrný čas a počet generací potřebných na vývoj je lepší pro výměnu po 100 a 500 generacích, u výměně po 1000 generacích je lehce lepší původní varianta. Křížení ovšem dosáhlo větší průměrné úspěšnosti, nalezlo více správných řešení. Ale původní variantě se podařilo nalézt řešení v průměru s menší plochou na čipu, což už, jak jsem psal výše, je pouze orientační ukazatel. Průměrná funkčnost je opět prakticky stejná u obou variant.

5.3 Šestibitová sčítačka

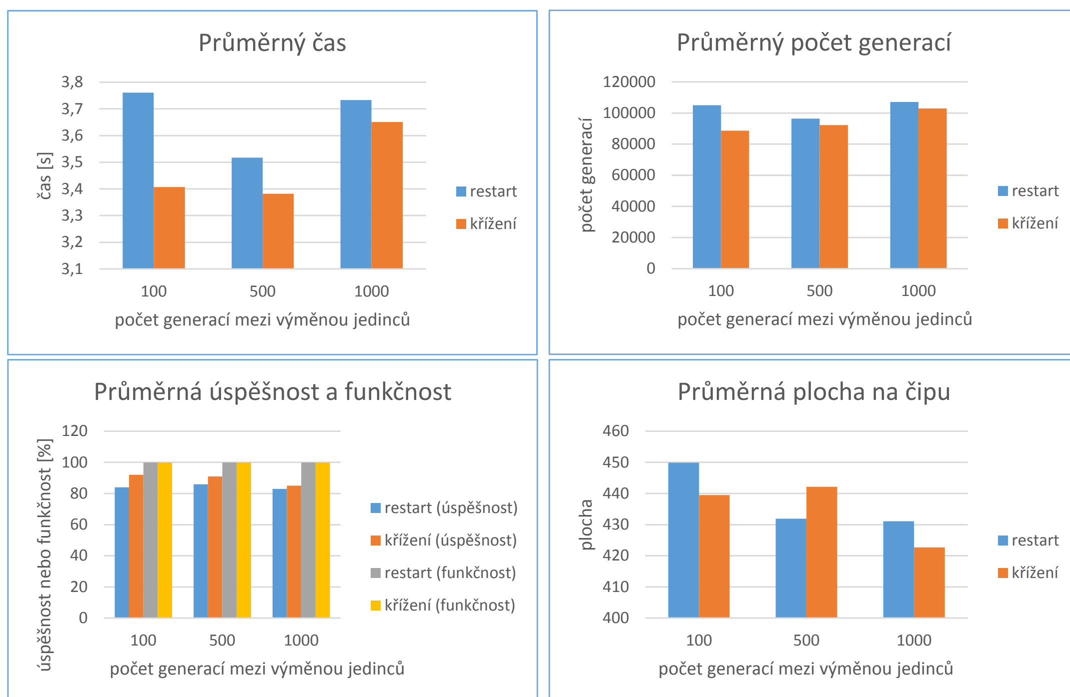
Šestibitovou sčítačku jsem zkoušel navrhovat postupně na jednom až čtyřech uzlech clusteru. To odpovídá 16, 32, 48 a 64 ostrovům modelu. Opět každý test pro výměnu jedinců mezi ostrovy po 100, 500 a 1000 generacích.

Pro jeden uzel jsem zvolil maximální počet generací na 350 000. Výsledky z testů na něm jsou shrnuty v tabulce 5.4 a grafech na obrázku 5.4. Je z nich zřejmé, že pro výměnu po 100 a 500 generacích je rychlejší původní algoritmus, kterému také na vývoj stačí menší počet generací. Ale našel řešení s větší plochou na čipu při těchto bězích. Oproti tomu při výměně po 1000 generacích je rychlejší křížení a stačí mu také méně generací vývoje, ale opět nalezne řešení s větší plochou na čipu. Záleží tedy na tom, co požadujeme, protože každá varianta je lepší na něco jiného a pro jiné parametry. Což je očekávané, protože pro každý problém se vždy lépe hodí určitý evoluční algoritmus a pro jiný zase jiný nebo aspoň s jinými parametry. Z hlediska úspěšnosti se podařilo vícekrát nalézt řešení původnímu algoritmu. Průměrná funkčnost je prakticky blíží stu procent.

Pro testování na dvou uzlech jsem zvolil maximální počet generací na 300 000. Výsledky jsou pak shrnuty v tabulce 5.5 a grafech 5.5. Je z nich zřejmé, že křížení si vedlo lépe, i když nikterak výrazně. Z hlediska průměrného času je největší rozdíl vidět u výměny nejlepšího jedince po 1000 generacích. Ovšem z hlediska průměrného počtu generací potřebných na nalezení řešení jsou velké rozdíly u výměny potomků při 100 a 1000 generacích. U výměny po 500 generacích je oproti tomu rozdíl minimální. Úspešnost vyšla také lépe pro křížení, ikdyž při výměně po 1000 generacích byla naprosto stejná. Jediné, v čem byla původní vari-

výměna	typ		čas [s]	generace	hamming	funkčnost	úspěšnost	plocha
100	restart	min	0,71506	20500	0	98,37		268
		max	7,75427	200000	100	100		660
		avg	3,76088	104932	4	99,93	84	449,86
100	křížení	min	0,6607	14500	0	97,59		238
		max	9,17515	200000	148	100		814
		avg	3,40666	88591	3,1	99,95	92	439,5
500	restart	min	0,64855	19000	0	98,4		236
		max	8,76921	200000	98	100		712
		avg	3,51766	96430	5,1	99,92	86	431,92
500	křížení	min	0,86671	21500	0	98,44		266
		max	8,47855	200000	96	100		626
		avg	3,38227	92165	2,8	99,96	91	442,14
1000	restart	min	0,60479	19000	0	97,53		240
		max	7,66029	200000	152	100		640
		avg	3,73366	107140	7,7	99,87	83	431,1
1000	křížení	min	0,67255	20000	0	97,14		236
		max	8,12184	200000	176	100		608
		avg	3,65103	102940	5,9	99,9	85	422,7

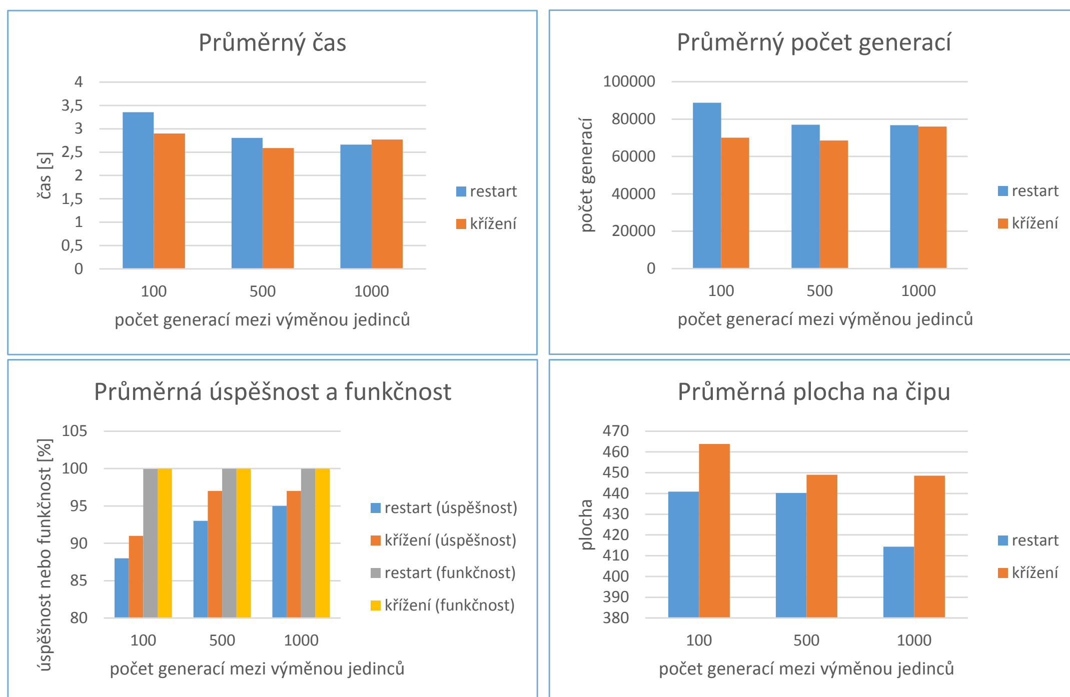
Tabulka 5.2: Výsledky pětibitové sčítačky na jednom uzlu



Obrázek 5.2: Grafy statistik při návrhu pětibitové sčítačky na jednom uzlu

výměna	typ		čas [s]	generace	hamming	funkčnost	úspěšnost	plocha
100	restart	min	0,28975	8700	0	98,83		256
		max	8,76033	200000	72	100		714
		avg	3,3568	88778	2,7	99,96	88	440,9
100	křížení	min	0,62332	13500	0	99,22		290
		max	9,26417	200000	48	100		722
		avg	2,89801	70036	1,3	99,98	91	463,9
500	restart	min	0,48914	14000	0	98,99		264
		max	8,04495	200000	62	100		688
		avg	2,80666	76995	1,6	99,97	93	440,26
500	křížení	min	0,56089	14000	0	97,79		194
		max	8,26823	200000	136	100		714
		avg	2,58936	68515	1,8	99,97	97	449,06
1000	restart	min	0,5054	15000	0	99,54		234
		max	7,11974	200000	28	100		668
		avg	2,66025	76750	0,6	99,99	95	414,32
1000	křížení	min	0,78404	23000	0	99,22		262
		max	8,28134	200000	48	100		730
		avg	2,76844	75960	0,5	99,99	97	448,56

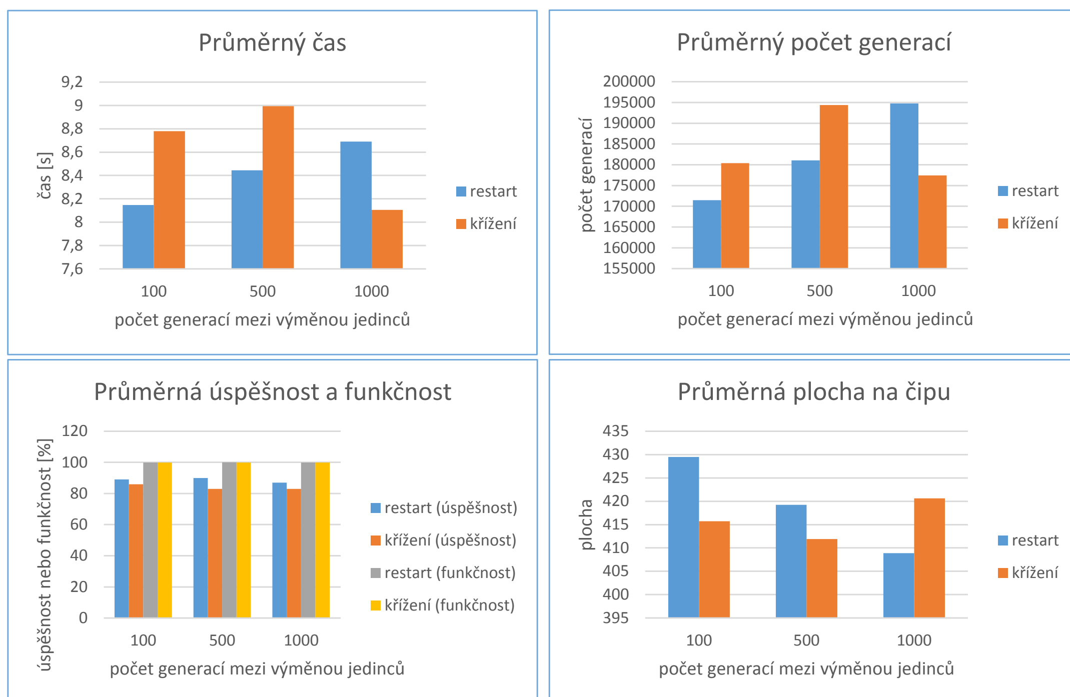
Tabulka 5.3: Výsledky pětibitové sčítačky na dvou uzlech



Obrázek 5.3: Grafy statistik při návrhu pětibitové sčítačky na dvou uzlech

výměna	typ		čas [s]	generace	hamming	funkčnost	úspěšnost	plocha
100	restart	min	1,34519	28800	0	98,61		276
		max	20,04502	350000	399	100		708
		avg	8,14787	171449	13,8	99,95	89	429,48
100	křížení	min	1,64765	28700	0	99,02		216
		max	22,98782	350000	280	100		678
		avg	8,77886	180384	12,9	99,95	86	415,74
500	restart	min	2,13059	45000	0	99,36		266
		max	18,6799	350000	184	100		614
		avg	8,44534	181045	5,1	99,98	90	419,26
500	křížení	min	1,07945	28000	0	99,44		254
		max	18,90985	350000	160	100		654
		avg	8,99397	194360	9,2	99,97	83	411,9
1000	restart	min	1,79303	41000	0	98,91		254
		max	18,70877	350000	312	100		734
		avg	8,69134	194750	18,7	99,93	87	408,9
1000	křížení	min	1,59921	37000	0	98,77		242
		max	18,68241	350000	352	100		642
		avg	8,10468	177410	12,2	99,96	83	420,6

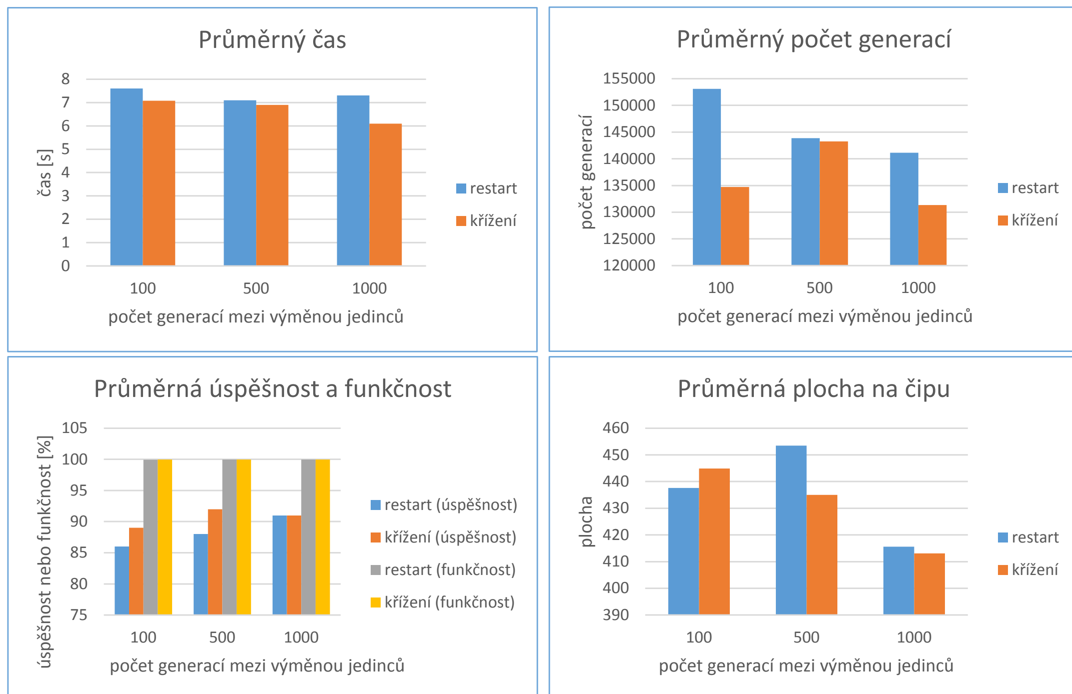
Tabulka 5.4: Výsledky šestibitové sčítačky na jednom uzlu



Obrázek 5.4: Grafy statistik při návrhu šestibitové sčítačky na jednom uzlu

výměna	typ		čas [s]	generace	hamming	funkčnost	úspěšnost	plocha
100	restart	min	1,06166	21100	0	98,6		274
		max	18,128	300000	400	100		624
		avg	7,59999	153081	16,1	99,94	86	437,6
100	křížení	min	0,674	10200	0	99,34		210
		max	16,50462	300000	188	100		706
		avg	7,07398	134713	8,9	99,97	89	444,86
500	restart	min	0,94319	19000	0	98,55		294
		max	20,15179	300000	416	100		734
		avg	7,10102	143860	10,1	99,96	88	453,46
500	křížení	min	1,40403	31000	0	99,25		226
		max	17,14002	300000	216	100		644
		avg	6,90334	143255	7,2	99,97	92	434,98
1000	restart	min	1,26759	31000	0	99,61		246
		max	18,06656	300000	112	100		690
		avg	7,30994	141130	5,2	99,98	91	415,58
1000	křížení	min	1,00069	25000	0	99,55		242
		max	16,08313	300000	128	100		578
		avg	6,09919	131330	4	99,99	91	413,08

Tabulka 5.5: Výsledky šestibitové sčítačky na dvou uzlech



Obrázek 5.5: Grafy statistik při návrhu šestibitové sčítačky na dvou uzlech

anta lepší byla plocha nalezeného řešení při výměně po 100 generacích, ovšem optimalizace plochy není předmětem zkoumání.

Testování na třech uzlech proběhlo pro maximální počet generací stanoven na 250 000. Výsledky těchto testů jsou shrnuty v tabulce 5.6 a grafech na obrázku 5.6. Testy pro výměnu jedinců po 1000 generacích vyšly v průměru lépe pro křížení. Vyhodnocování mu trvalo v průměru kratší čas, potřebovalo v průměru méně generací na vývoj a našlo řešení s menší potřebnou plochou na čipu. Úspěšnost měly obě varianty stejnou. Naproti tomu pro původní variantu byla úspěšnější výměna po 100 a 500 generacích, protože průměrně spotřebovala méně času, méně generací, byla úspěšnější, našla vícekrát správné řešení a také měla řešení v průměru s menšími nároky pro plochu na čipu. Funkčnost se jako obvykle v průměru pohybuje těsně pod hranicí sta procent.

Pro testování šestibitové sčítačky na čtyřech uzlech jsem zvolil maximální počet generací na 200 000. Výsledky testů jsou shrnuty v tabulce 5.7 a grafech na obrázku 5.7. Tyto testy dopadly značně lépe pro původní variantu než pro křížení. Pro křížení dopadl lépe jen průměrný čas pro výměnu potomků po 500 generacích a úspěšnost pro výměnu po 1000 generacích. Ve všech ostatních testech dopadla, jak už jsem psal, lépe původní varianta. Největší rozdíl je patrný na průměrném počtu generací při výměně potomků po 100 generacích. Průměrná funkčnost se opět blíží stu prucent bez rozdílu metody.

5.4 Sedmibitová sčítačka

Pro návrh sedmibitové sčítačky jsem zvolil testy na jednom až pěti uzlech clusteru. Což ostrovnímu modelu přiděluje 16, 32, 48, 64 a 80 ostrovů. Maximální počet generací pro nalezení řešení byl stanoven na 1 000 000. Pro každý počet ostrovů byly otestovány tři možnosti výměny nejlepších jedinců a to po 100, 500 a 1000 generacích vývoje.

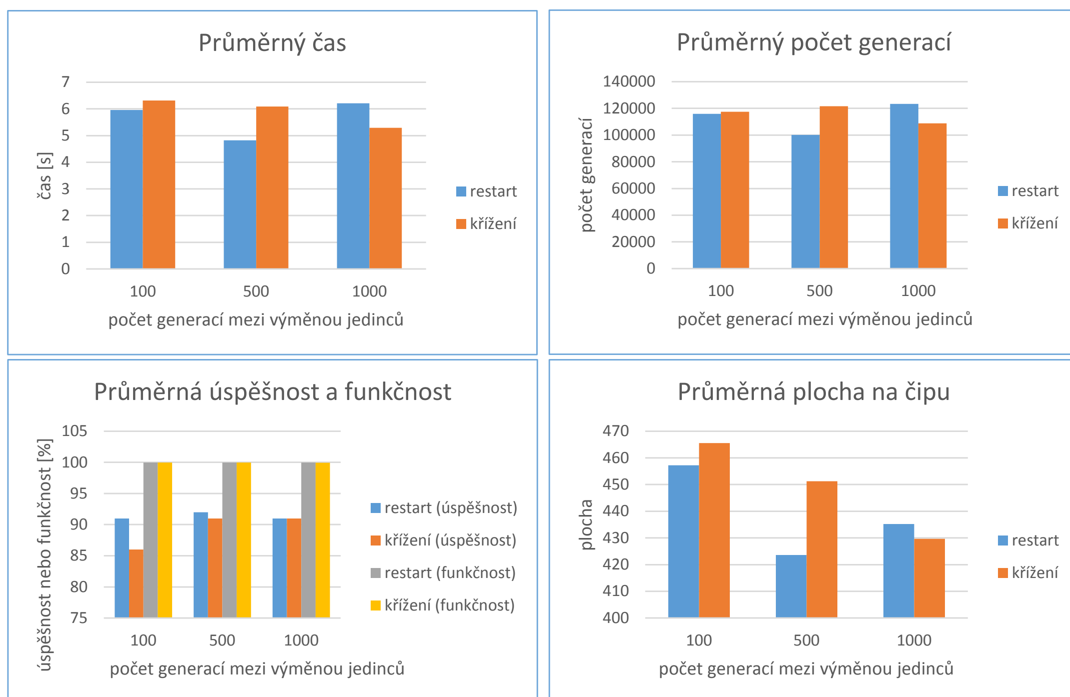
Shrnutí výsledků z testování na jednom uzlu, šestnáci ostrovech, jsou zobrazeny v tabulce 5.8 a grafech na obrázku 5.8. Je z nich patrná převaha křížení při výměně nejlepších jedinců po 1000 generacích a také převaha původní metody pro výměnu po 100 a 500 generacích. Ovšem rozdíly nejsou nikterak výrazné pro průměrný čas a úspěšnost nalezení řešení v jednotlivých bězích. Jedinný výraznější rozdíl je pro průměrný počet generací při výměně po 1000 generacích. Značné rozdíly jsou i pro plochu na čipu, ale její minimalizace nebyla testy podporována. Průměrná funkčnost se jako obvykle blíží stu procent.

V tabulce 5.9 a grafech na obrázku 5.9 je zobrazeno shrnutí výsledků z testů na dvou uzlech, 32 osrovech modelu. Výměna nejlepších jedinců po 1000 generacích dopadla lépe pro původní metodu ve všech zkoumaných kritériích. Při výměně po 100 a 500 generacích je lepší křížení z hlediska průměrného času běhu a maximálního počtu potřebných generací. Původní metoda našla při každém nastavení vícekrát správné řešení. Avšak ve většině zkoumaných hodnot je rozdíl mezi metodami jen minimální. Větší rozdíl je patrný u průměrného počtu generací u výměny jedinců po 100 generacích, ale už s ním tolik nekoresponduje úspora času. Je to dáno tím, že u křížení se provede více vyhodnocování než u původní varianty. Průmerná funkčnost nalezených řešení je sto procentní, nebo se jí blíží u obou variant.

Shrnutí výsledku testů na třech uzlech je zobrazeno v tabulce 5.10 a grafech na obrázku 5.10. Je z nich patrné, že křížení se vyplatilo při výměně jedinců po 1000 generacích. Oproti tomu při výměně jedinců po 100 a 500 generacích byla úspěšnější původní varianta. I když při výměně po 500 generacích křížení potřebovalo méně generací, tak ale spotřebovalo více času, což je podstatnější parametr. I když většina rozdílu mezi variantami je prakticky zanedbatelná, výjimku tvoří průměrný počet generací potřebných pro nalezení řešení při

výměna	typ		čas [s]	generace	hamming	funkčnost	úspěšnost	plocha
100	restart	min	0,92178	20100	0	99,22		264
		max	15,51015	250000	224	100		808
		avg	5,96014	115857	6,7	99,98	91	457,24
100	křížení	min	0,95705	16200	0	99,05		286
		max	16,22396	250000	272	100		740
		avg	6,31629	117491	8,3	99,97	86	465,6
500	restart	min	0,66987	15000	0	99,22		230
		max	13,89599	250000	224	100		636
		avg	4,82417	100220	4,9	99,98	92	423,58
500	křížení	min	1,2818	23000	0	99,11		292
		max	14,3998	250000	256	100		656
		avg	6,08808	121610	7,4	99,97	91	451,28
1000	restart	min	1,06225	23000	0	99,33		298
		max	14,30133	250000	192	100		606
		avg	6,21188	123380	6,6	99,98	91	435,26
1000	křížení	min	1,04442	26000	0	95,9		252
		max	15,63759	250000	1176	100		720
		avg	5,2892	108770	15,9	99,94	91	429,64

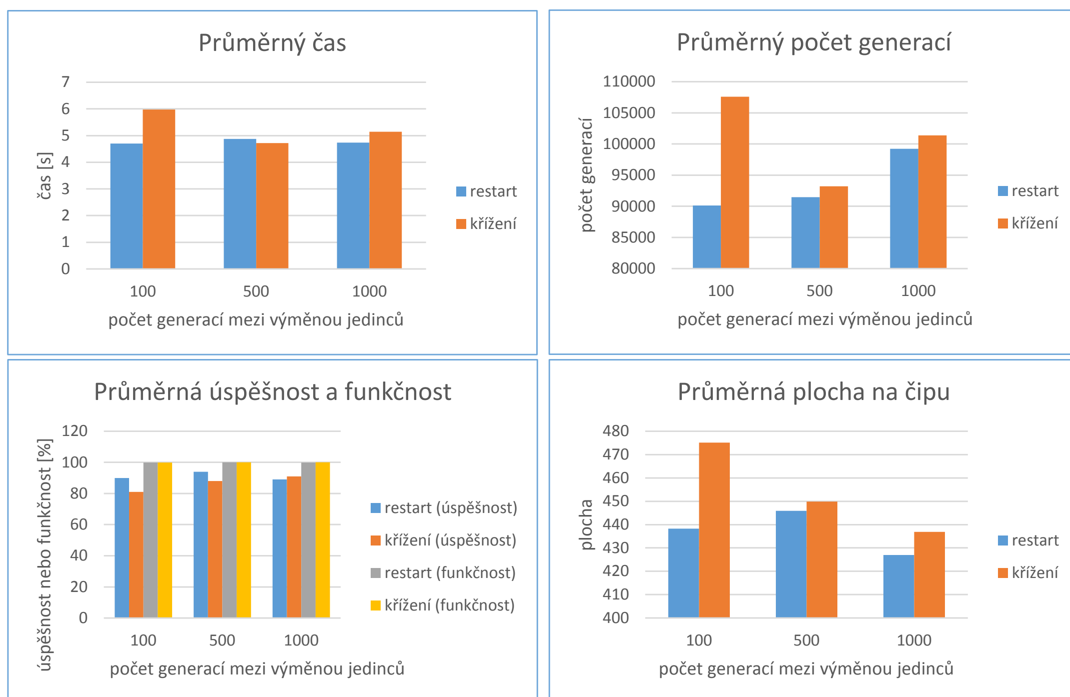
Tabulka 5.6: Výsledky šestibitové sčítačky na třech uzlech



Obrázek 5.6: Grafy statistik při návrhu šestibitové sčítačky na třech uzlech

výměna	typ		čas [s]	generace	hamming	funkčnost	úspěšnost	plocha
100	restart	min	0,70025	12900	0	99,22		256
		max	12,38526	200000	224	100		654
		avg	4,69822	90144	9,1	99,97	90	438,24
100	křížení	min	0,98935	16700	0	98,52		272
		max	12,87803	200000	424	100		856
		avg	5,97861	107571	16,6	99,94	81	475,08
500	restart	min	0,89576	19500	0	99,51		272
		max	14,22845	200000	140	100		630
		avg	4,87406	91465	3,9	99,99	94	445,9
500	křížení	min	1,00641	18000	0	99,33		274
		max	10,94466	200000	192	100		676
		avg	4,7178	93215	7,2	99,98	88	449,82
1000	restart	min	0,91804	19000	0	99,22		188
		max	12,09919	200000	224	100		618
		avg	4,73519	99230	8,4	99,97	89	427
1000	křížení	min	1,31615	22000	0	99,11		266
		max	11,50441	200000	256	100		660
		avg	5,14704	101380	5,9	99,98	91	436,9

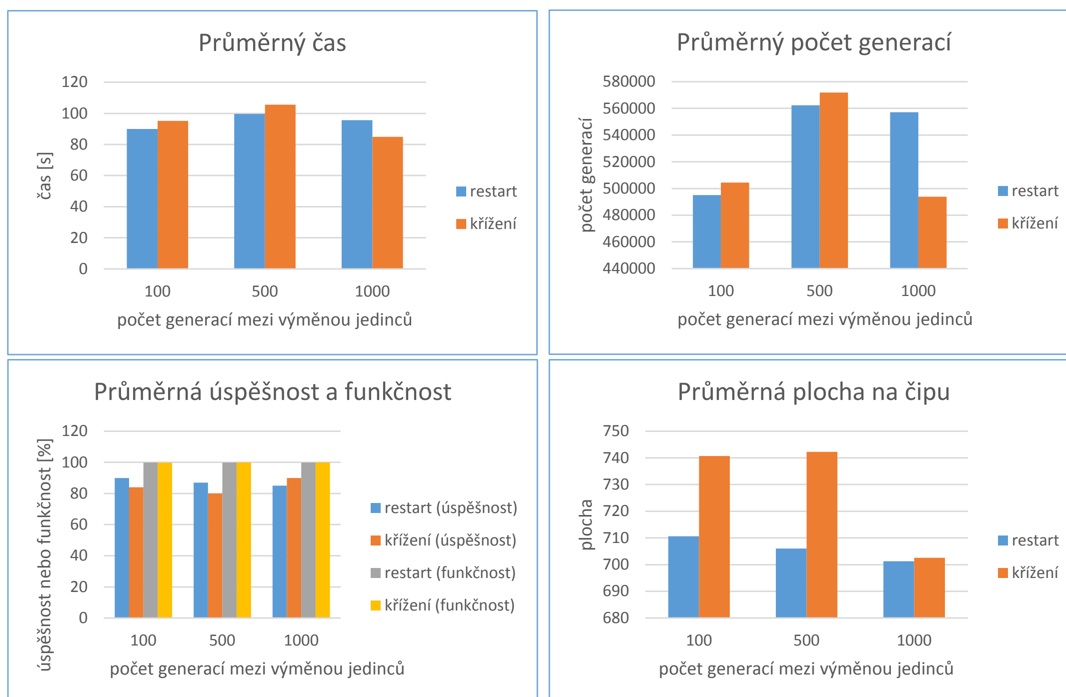
Tabulka 5.7: Výsledky šestibitové sčítačky na čtyřech uzlech



Obrázek 5.7: Grafy statistik při návrhu šestibitové sčítačky na čtyřech uzlech

výměna	typ		čas [s]	generace	hamming	funkčnost	úspěšnost	plocha
100	restart	min	16,12153	89900	0	98,96		426
		max	313,10868	1000000	1360	100		1144
		avg	89,91161	495038	56,1	99,96	90	710,64
100	křížení	min	12,30199	72100	0	91,26		504
		max	332,63366	1000000	11456	100		1270
		avg	95,14241	504427	395,4	99,7	84	740,68
500	restart	min	16,12594	100500	0	98,13		438
		max	245,33671	1000000	2456	100		944
		avg	99,61498	562340	62,7	99,95	87	706,02
500	křížení	min	13,96999	98000	0	93,48		458
		max	287,48764	1000000	8552	100		1068
		avg	105,61346	571820	113,4	99,91	80	742,28
1000	restart	min	17,54933	114000	0	99,13		472
		max	261,22761	1000000	1136	100		1060
		avg	95,60021	557130	62,9	99,95	85	701,22
1000	křížení	min	14,65683	71000	0	96,36		360
		max	296,79926	1000000	4768	100		976
		avg	84,95801	493830	76,1	99,94	90	702,58

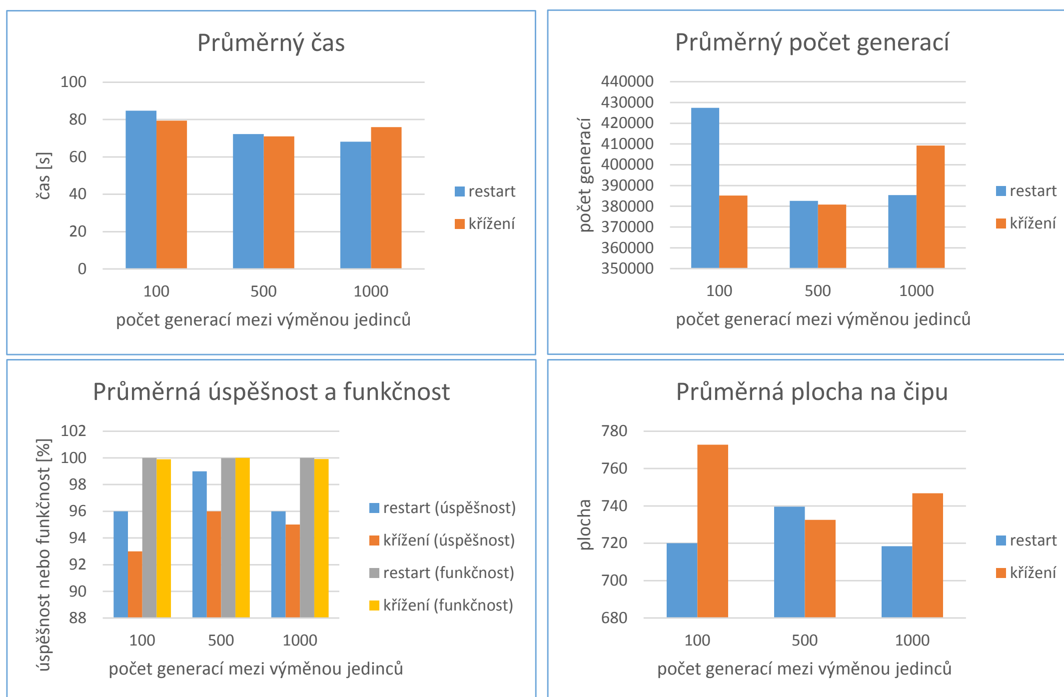
Tabulka 5.8: Výsledky sedmibitové sčítačky na jednom uzlu



Obrázek 5.8: Grafy statistik při návrhu sedmibitové sčítačky na jednom uzlu

výměna	typ		čas [s]	generace	hamming	funkčnost	úspěšnost	plocha
100	restart	min	7,15238	55500	0	99,84		444
		max	380,28034	1000000	208	100		1210
		avg	84,78057	427428	4,4	100	96	720,08
100	křížení	min	5,12939	26800	0	92,93		408
		max	350,68344	1000000	9264	100		1216
		avg	79,40744	385215	129,9	99,9	93	772,74
500	restart	min	11,23793	56500	0	98,41		548
		max	257,68069	1000000	2080	100		1168
		avg	72,28834	382620	20,8	99,98	99	739,62
500	křížení	min	7,44625	44500	0	99,9		426
		max	378,34161	1000000	128	100		1082
		avg	70,99591	380855	2,3	100	96	732,54
1000	restart	min	5,40739	34000	0	99,68		432
		max	207,45331	1000000	416	100		1034
		avg	68,21142	385390	6,2	100	96	718,38
1000	křížení	min	15,63278	82000	0	91,96		494
		max	318,68416	1000000	10536	100		1056
		avg	75,96382	409300	120,1	99,91	95	746,82

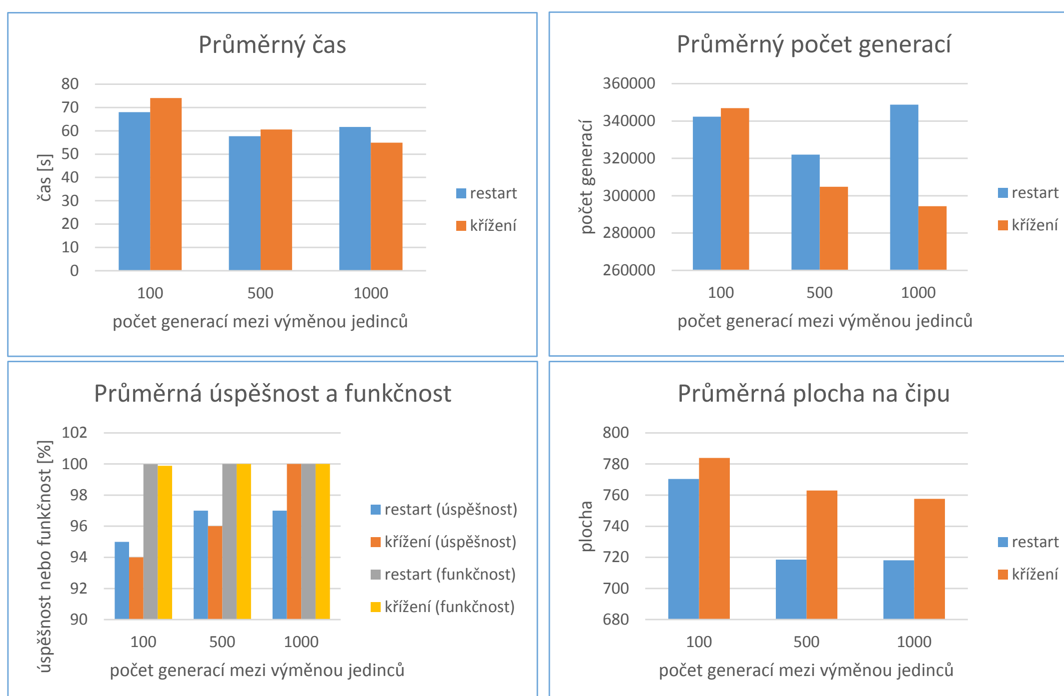
Tabulka 5.9: Výsledky sedmibitové sčítačky na dvou uzlech



Obrázek 5.9: Grafy statistik při návrhu sedmibitové sčítačky na dvou uzlech

výměna	typ		čas [s]	generace	hamming	funkčnost	úspěšnost	plocha
100	restart	min	8,37487	66100	0	99,58		492
		max	303,45246	1000000	548	100		1134
		avg	67,9658	342321	10,4	99,99	95	770,4
100	křížení	min	8,18681	43000	0	94,32		472
		max	387,68922	1000000	7448	100		1514
		avg	74,10081	346887	158,7	99,88	94	783,86
500	restart	min	11,7406	66500	0	99,77		464
		max	282,08405	1000000	304	100		1024
		avg	57,72186	322025	4,8	100	97	718,58
500	křížení	min	9,36999	57000	0	99,93		486
		max	303,40329	1000000	96	100		1156
		avg	60,61688	304725	2	100	96	763
1000	restart	min	8,18941	62000	0	99,73		490
		max	236,22595	1000000	356	100		1032
		avg	61,62691	348790	6,2	100	97	718,12
1000	křížení	min	13,67363	82000	0	100		498
		max	209,51997	836000	0	100		1122
		avg	54,94072	294340	0	100	100	757,58

Tabulka 5.10: Výsledky sedmibitové sčítačky na třech uzlech



Obrázek 5.10: Grafy statistik při návrhu sedmibitové sčítačky na třech uzlech

výměně potomků po 1000 generacích. Jen pro zajímavost, původní variantě se v průměru podařilo nalézt řešení s menší plochou na čipu. V průměru se nám vždy podařilo nálézt až stoprocentně funkční řešení.

Výsledky testování na čtyřech uzlech jsou shrnuty v tabulce 5.11 a grafech na obrázku 5.11. Z nich je patrné, že křížení je rychlejší pro výměnu nejlepších jedinců po 500 a 1000 generacích. Naopak pro výměnu po 100 generacích byla rychlejší původní varianta. Stejné výsledky jsou i pro maximální počet generací potřebných pro nalezení řešení. Úspěšnější byla původní varianta, našla více správných řešení pro výměnu jedinců po 100 a 1000 generacích, křížení pak pro výměnu po 500 generacích. Průměrná funkčnost je opět stoprocentní, nebo se jí blíží. Řešení nalezá původní metodou by v průměru zabrala méně plochy na čipu, ale test nebyly zaměřeny na hledání minimální plochy, ale na nalezení jakéhokoliv řešení.

V tabulce 5.12 a grafech na obrázku 5.12 jsou shrnuty výsledky testování na 80 ostrovech modelu. V této konfiguraci je lepší původní varianta, která dosáhla nižšího průměrného času při výměně potomků po 100 a 1000 generacích a při výměně po 500 generacích jsou průměrné hodnoty spotřebovaného času prakticky totožné. Stejné výsledky jsou i u průměrného počtu generací při výměně jedinců po 100 generacích. Avšak při ostatních výměnách je zvětšující se rozdíl v potřebném počtu generací ve prospěch křížení, jenže nebyl následován lepším časem. Původní metodě se také podařilo vícekrát nalézt řešení problému než při křížení a také nalezlo řešení s menší plochou na čipu.

5.5 Osmibitová sčítačka

Osmibitová sčítačka je už značně složitý problém, proto jsem z testováním začal na dvou uzlech a pokračoval až na pět uzlů clusteru. Pro model to znamená počítání na 32, 48, 64 a 80 ostrovech. Maxinální počet generací pro nalezení řešení byl stanoven na 900 000. I jako v ostatních případech byla testována výměna nejlepších jedinců po 100, 500 a 1000 generacích.

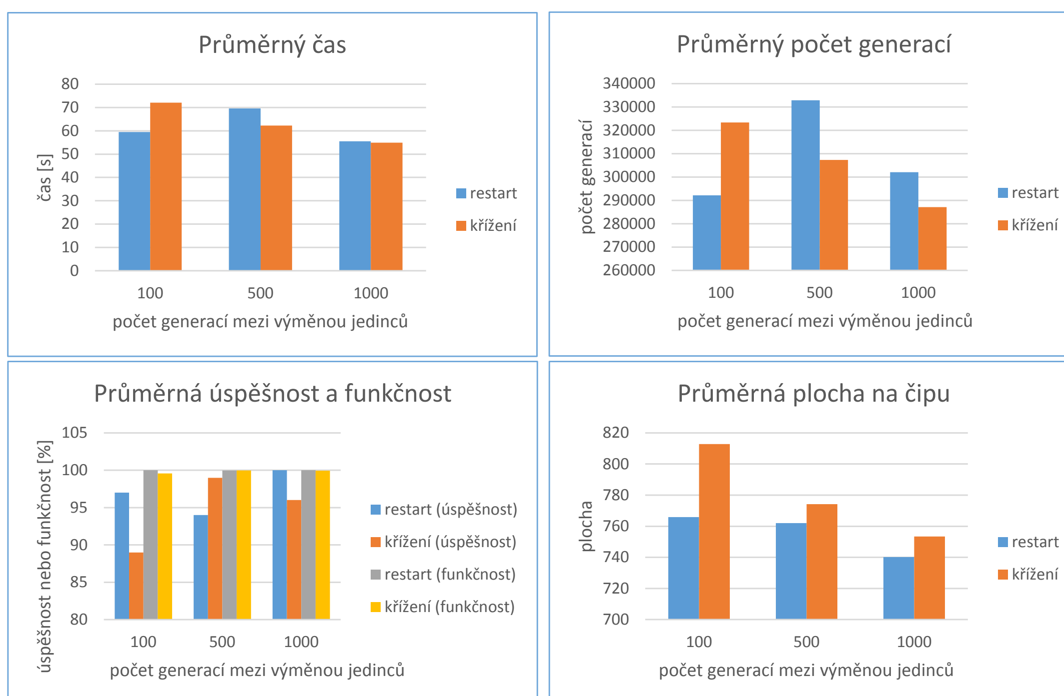
Testování na dvou uzlech je zobrazeno v tabulce 5.13 a grafech na obrázku 5.13. Z výsledků plyne značná rychlostní převaha křížení oproti původní variantě při výměně jedinců po 500 a 1000 generacích. Při výměně po 100 generacích je rychlejší původní varianta, avšak už ne tak výrazně. Ve všech případech stačilo křížení v průměru menší počet generací na nalezení řešení. Také úspěšnost nalezení správného řešení byla u křížení vyšší. Funkčnost je pro obě varianty prakticky totožná, blížící se stu procent. Dokonce se křižení podařilo v průměru nalézt i řešení zabírající menší plochu na čipu, ale je to informace jen pro zajímavost.

Výsledky testů na třech uzlech jsou shrnuty v tabulce 5.14 a grafech na obrázku 5.14. Je z nich patrné, že křížení je lepší pro výměnu jedinců po 500 a 1000 generacích. Je to patrné ze všech zkoumaných veličin. Naproti tomu je původní varianta lepší pro výměnu jedinců po 100 generacích, také ve všech zkoumaných kriteriích. Jen tyto rozdíly mezi jednotlivými variantami nejsou výrazné. Průměrná funkčnost se opět pohybuje těsně pod hranicí sta procent.

Shrnutí výsledků testování na čtyřech uzlech je zobrazeno v tabulce 5.15 a grafech na obrázku 5.15. Plyne z nich, že křížení dopadlo nejlépe pro výměnu potomků po 500 generacích a také bylo, i když ne tak výrazně, lepší i pro výměnu po 1000 generacích. Oproti tomu původní varianta byla v průměru rychlejší při výměně po 100 generacích. Pro výměnu potomků po 100 a 500 generacích koresponduje s předchozými výsledky s velkými rozdíly mezi variantami také průměrný počet potřebných generací pro nalezení řešení. Pro výměnu po 1000 generacích stačilo původní variantě méně generací k nalezení řešení, ale rozdíl není

výměna	typ		čas [s]	generace	hamming	funkčnost	úspěšnost	plocha
100	restart	min	6,55458	41700	0	99,95		432
		max	218,14055	1000000	64	100		984
		avg	59,44117	292193	1,1	100	97	765,8
100	křížení	min	10,79048	45400	0	84,96		482
		max	342,11298	1000000	19712	100		1256
		avg	72,11222	323346	559,5	99,57	89	812,86
500	restart	min	10,91541	56000	0	99,24		502
		max	304,22404	1000000	996	100		1158
		avg	69,56508	332835	16,4	99,99	94	762,02
500	křížení	min	9,7286	51000	0	97,49		508
		max	289,70363	1000000	3296	100		1422
		avg	62,24483	307345	33	99,97	99	774,22
1000	restart	min	12,08107	71000	0	100		498
		max	177,13969	989000	0	100		1050
		avg	55,49312	302090	0	100	100	740,14
1000	křížení	min	7,205	56000	0	95,59		456
		max	346,42986	1000000	5784	100		1208
		avg	54,89404	287110	83,9	99,94	96	753,4

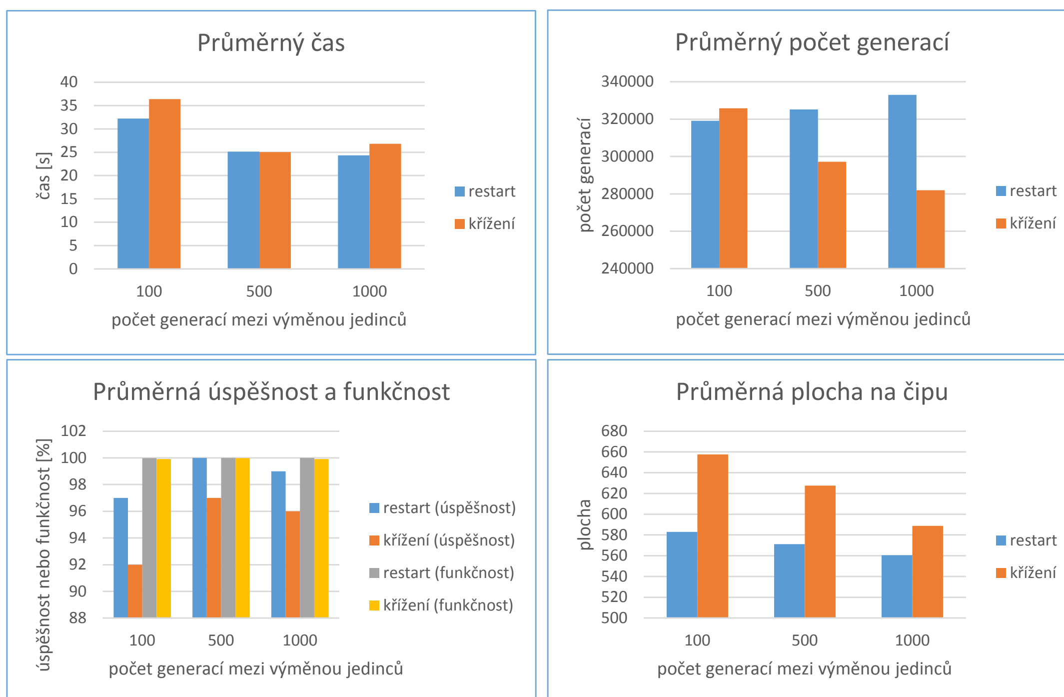
Tabulka 5.11: Výsledky sedmibitové sčítačky na čtyřech uzlech



Obrázek 5.11: Grafy statistik při návrhu sedmibitové sčítačky na čtyřech uzlech

výměna	typ		čas [s]	generace	hamming	funkčnost	úspěšnost	plocha
100	restart	min	7,59114	90100	0	99,54		436
		max	109,82441	1000000	608	100		706
		avg	32,22965	319040	10,6	99,99	97	583
100	křížení	min	1,86285	23600	0	97,72		406
		max	252,2606	1000000	2984	100		2364
		avg	36,38101	325804	114,9	99,91	92	657,74
500	restart	min	4,41243	62500	0	100		354
		max	93,97202	986500	0	100		842
		avg	25,14679	325180	0	100	100	571,16
500	křížení	min	4,21131	62500	0	97,54		448
		max	95,84348	1000000	3228	100		986
		avg	25,01601	297140	32,7	99,98	97	627,5
1000	restart	min	4,92864	81000	0	99,87		400
		max	80,40111	1000000	176	100		824
		avg	24,3365	333010	1,8	100	99	560,46
1000	křížení	min	4,23407	60000	0	95,86		396
		max	109,18784	1000000	5432	100		918
		avg	26,79645	281960	102,2	99,92	96	588,84

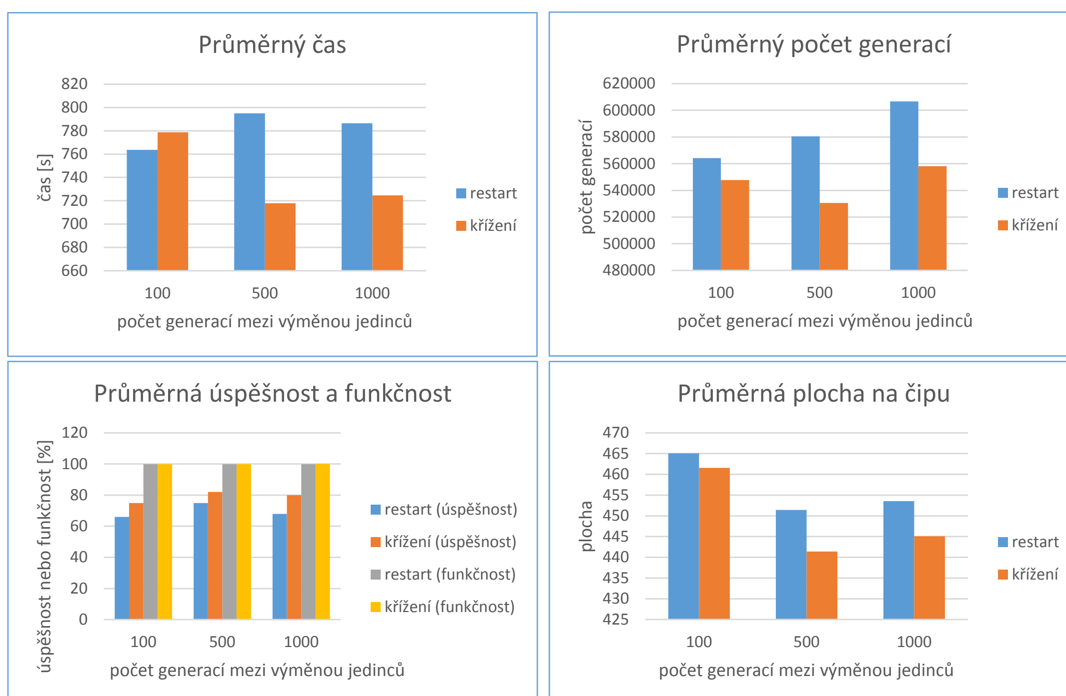
Tabulka 5.12: Výsledky sedmibitové sčítačky na pěti uzlech



Obrázek 5.12: Grafy statistik při návrhu sedmibitové sčítačky na pěti uzlech

výměna	typ		čas [s]	generace	hamming	funkčnost	úspěšnost	plocha
100	restart	min	105,03491	93700	0	99,4		314
		max	1741,7758	900000	3522	100		726
		avg	763,764045	564130	270,58	99,9541	66	465,08
100	křížení	min	127,31161	88300	0	95,22		286
		max	1686,13623	900000	28176	100		678
		avg	778,8298942	547760	662,32	99,8877	75	461,56
500	restart	min	61,20474	48000	0	99,39		310
		max	1848,37819	900000	3584	100		738
		avg	794,9992389	580530	200,08	99,9659	75	451,42
500	křížení	min	195,55615	115000	0	99,23		290
		max	1520,69151	900000	4544	100		694
		avg	717,9154529	530555	178,72	99,9697	82	441,4
1000	restart	min	139,79115	107000	0	99,4		314
		max	1571,49944	900000	3552	100		636
		avg	786,4823233	606580	327,6	99,9444	68	453,54
1000	křížení	min	128,13307	98000	0	99,74		306
		max	1892,77572	900000	1536	100		592
		avg	724,5561944	558160	109,12	99,9814	80	445,08

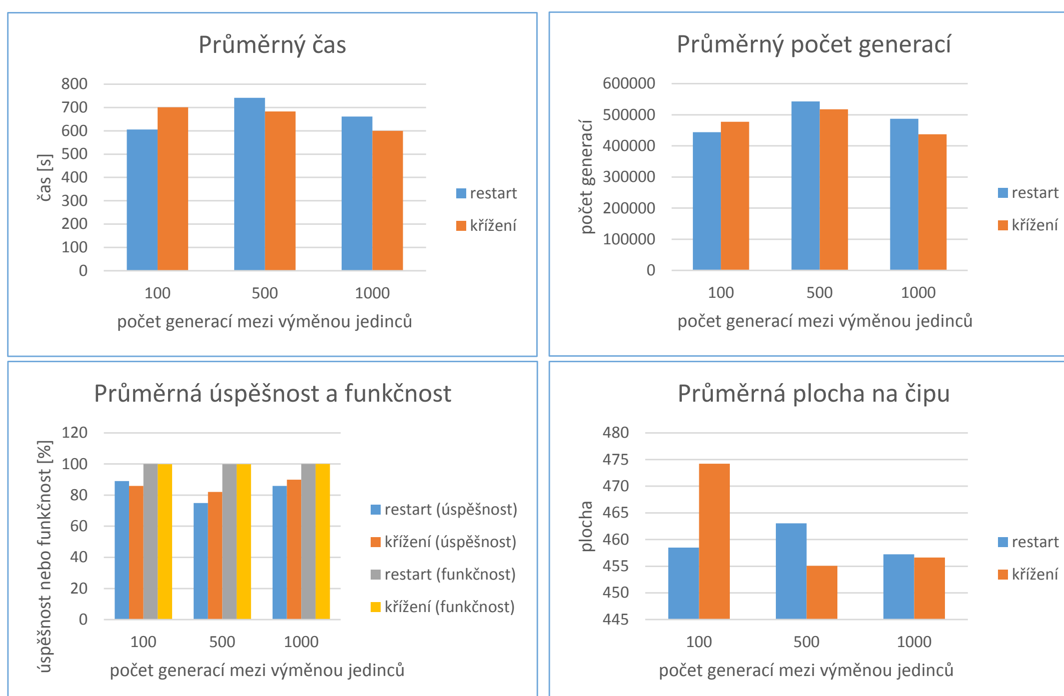
Tabulka 5.13: Výsledky osmibitové sčítačky na dvou uzlech



Obrázek 5.13: Grafy statistik při návrhu osmibitové sčítačky na dvou uzlech

výměna	typ		čas [s]	generace	hamming	funkčnost	úspěšnost	plocha
100	restart	min	103,91024	62200	0	99,74		304
		max	2194,02726	900000	1536	100		704
		avg	605,4811278	443867	61,76	99,9897	89	458,48
100	křížení	min	118,24898	70900	0	96,2		326
		max	1844,27774	900000	22432	100		682
		avg	701,2006646	477107	392,32	99,9334	86	474,22
500	restart	min	116,92527	111000	0	99,67		312
		max	1890,8779	900000	1920	100		614
		avg	742,0422938	543055	146	99,975	75	463,06
500	křížení	min	130,85294	108000	0	98,3		312
		max	1573,27264	900000	10016	100		618
		avg	682,6316876	517640	262,24	99,9553	82	455,06
1000	restart	min	119,37374	93000	0	99,75		334
		max	1586,48138	900000	1472	100		674
		avg	660,8286634	486930	56,08	99,9907	86	457,24
1000	křížení	min	112,87311	112000	0	99,76		306
		max	1731,57207	900000	1408	100		628
		avg	599,7983029	437130	60,32	99,9898	90	456,64

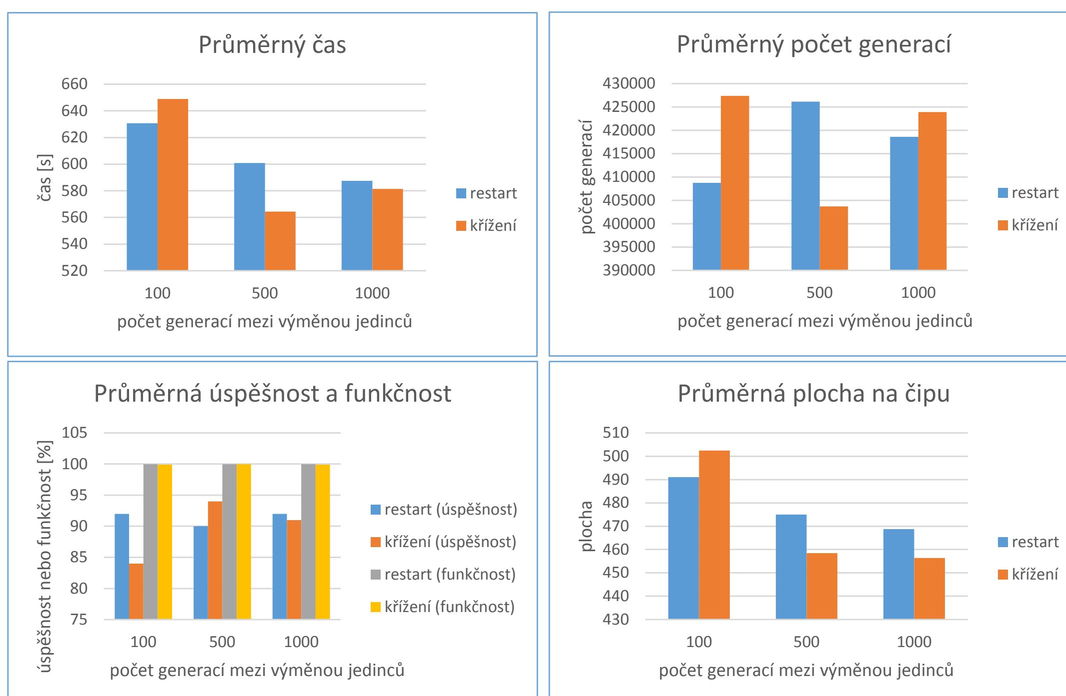
Tabulka 5.14: Výsledky osmibitové sčítačky na třech uzlech



Obrázek 5.14: Grafy statistik při návrhu osmibitové sčítačky na třech uzlech

výměna	typ		čas [s]	generace	hamming	funkčnost	úspěšnost	plocha
100	restart	min	86,77896	55700	0	99,53		350
		max	1747,65239	900000	2784	100		750
		avg	630,631752	408743	61,44	99,9897	92	491,06
100	křížení	min	69,92063	58400	0	94,67		316
		max	1986,25408	900000	31456	100		880
		avg	648,9131916	427366	658,36	99,8883	84	502,4
500	restart	min	105,42258	76000	0	99,25		300
		max	1740,31992	900000	4416	100		700
		avg	600,9100218	426125	97,6	99,9832	90	474,98
500	křížení	min	99,86574	76000	0	99,54		308
		max	1569,70492	900000	2688	100		616
		avg	564,423609	403680	52,48	99,9909	94	458,44
1000	restart	min	92,89175	49000	0	99,65		324
		max	1549,46847	900000	2064	100		636
		avg	587,4677452	418600	62,24	99,9894	92	468,72
1000	křížení	min	105,7988	102000	0	92,91		328
		max	1682,04818	900000	41792	100		770
		avg	581,3716663	423900	538,72	99,9086	91	456,36

Tabulka 5.15: Výsledky osmibitové sčítačky na čtyřech uzlech



Obrázek 5.15: Grafy statistik při návrhu osmibitové sčítačky na čtyřech uzlech

tak výrazný jako u předešlých výměn potomků. Křížení se podařilo nalézt více správných řešení pro výměnu po 500 generacích, ale pro ostatní výměny byla úspěšnější původní varianta. Opět byla řešení nalezaná oběma variantami v průměru téměř stoprocentně funkční. Průměrná velikost plochy nalezeného řešení na čipu koresponduje s rychlostí, tedy při výměně potomků po 500 a 1000 generacích nalezlo menší plochu křížení a po 100 generacích původní varianta.

V tabulce 5.16 a grafech na obrázku 5.16 jsou shrnuty výsledky z návrhu osmibitové sčítačky na pěti uzlech, 80 ostrovech modelu. Je z nich zřejmé, že pro křížení dopadly lépe testy pro výměnu potomků po 100 a 1000 generacích z hlediska průměrného času a pro výměnu po 500 generacích byla rychlejší původní metoda, ovšem tento rozdíl je prakticky zanedbatelný. Ve všech testech stačilo křížení v průměru méně generací pro nalezení řešení a také nalezlo více správných řešení než původní varianta. Průměrná funkčnost všech nalezených řešení u obou variant se blíží stu procent. Pro zajímavost menší plochu na čipu by zabrala řešení nalezená původní variantou při výměně jedinců po 100 generacích a křížení po 500 a 1000 generacích. Ovšem tyto rozdíly jsou jen minimální.

Testováno bylo i na šesti uzlech clusteru, ale z hlediska časové a prostorové náročnosti na výpočetní zdroje byly otestovány jen varianty s výměnou jedinců po 500 generacích. Jejich výsledky jsou zobrazeny v tabulce 5.17 a grafech na obrázku 5.17. V tomto případě mělo jasnou převahu křížení, které bylo v průměru rychlejší a stačilo mu méně generací na nalezení řešení. Také našlo vícekrát správné řešení problému a dokonce i jím nalezaná řešení by v průměru zabrala menší plochu na čipu.

5.6 Devítibitová sčítačka

Nalezení obvodu pro devítibitovou ščítačku je velice časově a prostorově náročný problém. K jeho vyřešení je potřeba značné množství výpočetních zdrojů. Z těchto důvodů byly provedeny nezávislé testy jen v deseti bězích, což může výsledky zkreslovat a tudíž mohou být značně zavadějící. Uvádím je proto spíše informativně, že i takto složité problémy jsou realizovatelné.

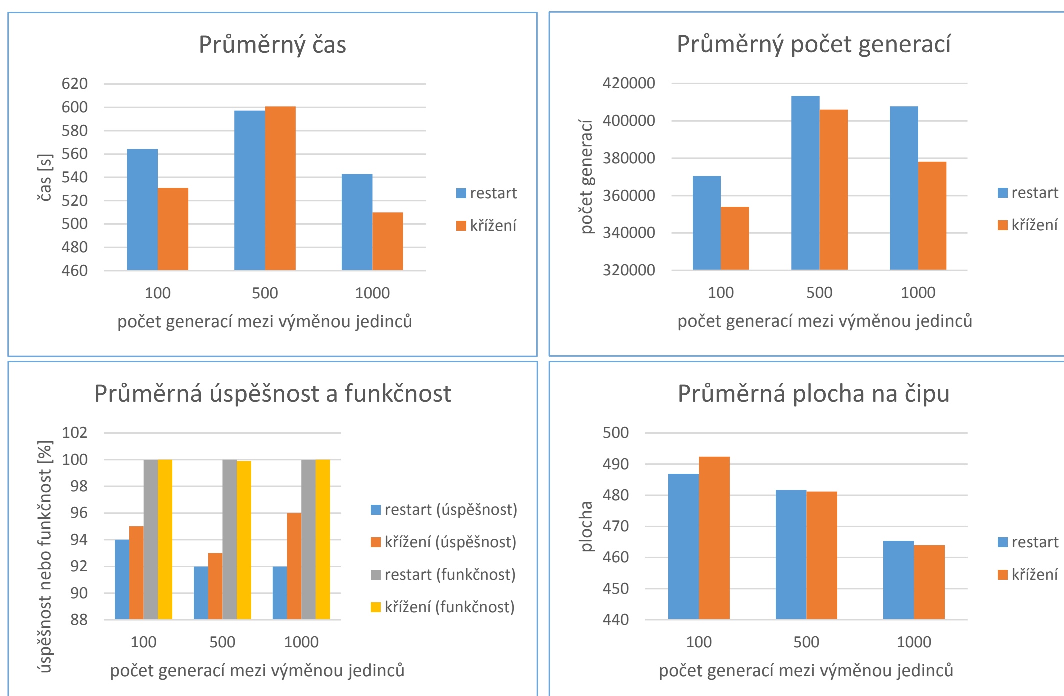
Testování pro maximální počet generací potřebných pro nalezení stanovený na 500 000 a výměnu nejlepších jedinců po 1000 generacích je shrnuto v tabulce 5.18 a grafech na obrázku 5.18. Na 160 ostrovech bylo z přehledem rychlejší křížení. Stačilo mu v průměru méně generací, našlo vícekrát správné řešení a také jím nalezené řešení by zabralo v průměru menší plochu na čipu. Na 240 ostrovech je pro obě varianty prakticky stejný průměrný potřebný čas. Počet generací má nižší křížení, ale bez vlivu na spotřebovaný čas. Našlo také o jedno řešení více než původní varianta. Původní variantě se podařilo nalézt řešení s menší potřebnou plochou na čipu. Optimalizace plochy na čipu ale nebyla předmětem testu. Průměrná funkčnost nalezených řešení od obou variant se opět blíží stu procent.

5.7 Dvoubitová násobička

Dvoubitová násobička je triviální problém, proto mi stačilo stanovit maximální počet generací na 10 000 a počítat na 16 ostrovech modelu. Opět jsem zkoušel varianty výměn nejlepších jedinců po 100, 500 a 1000 generacích. Výsledky testů jsou shrnuty v tabulce 5.19 a grafech na obrázku 5.19. Z hlediska rychlosti bylo lepší křížení při výměně po 1000 generacích, původní varianta při výměně po 100 generacích a při výměně po 500 generacích byly obě metody prakticky stejně rychlé. Křížení stačil ve všech testech v průměru

výměna	typ		čas [s]	generace	hamming	funkčnost	úspěšnost	plocha
100	restart	min	80,91905	58400	0	99,69		352
		max	1907,48852	900000	1824	100		680
		avg	564,2487441	370431	44	99,9925	94	486,94
100	křížení	min	77,75695	59700	0	99,8		334
		max	1658,06697	900000	1152	100		734
		avg	530,991147	353978	22,4	99,9961	95	492,4
500	restart	min	109,24653	81500	0	99,85		338
		max	1546,39095	900000	896	100		660
		avg	597,1287818	413325	23,84	99,996	92	481,72
500	křížení	min	59,38205	43000	0	90,09		308
		max	2006,58184	900000	58432	100		762
		avg	600,8219804	406060	602,56	99,8977	93	481,16
1000	restart	min	83,32913	68000	0	99,65		300
		max	1489,25593	900000	2048	100		652
		avg	542,8416534	407740	55,36	99,9906	92	465,36
1000	křížení	min	80,50087	71000	0	99,85		320
		max	1350,49144	900000	896	100		674
		avg	509,9511306	378210	19,36	99,9967	96	463,92

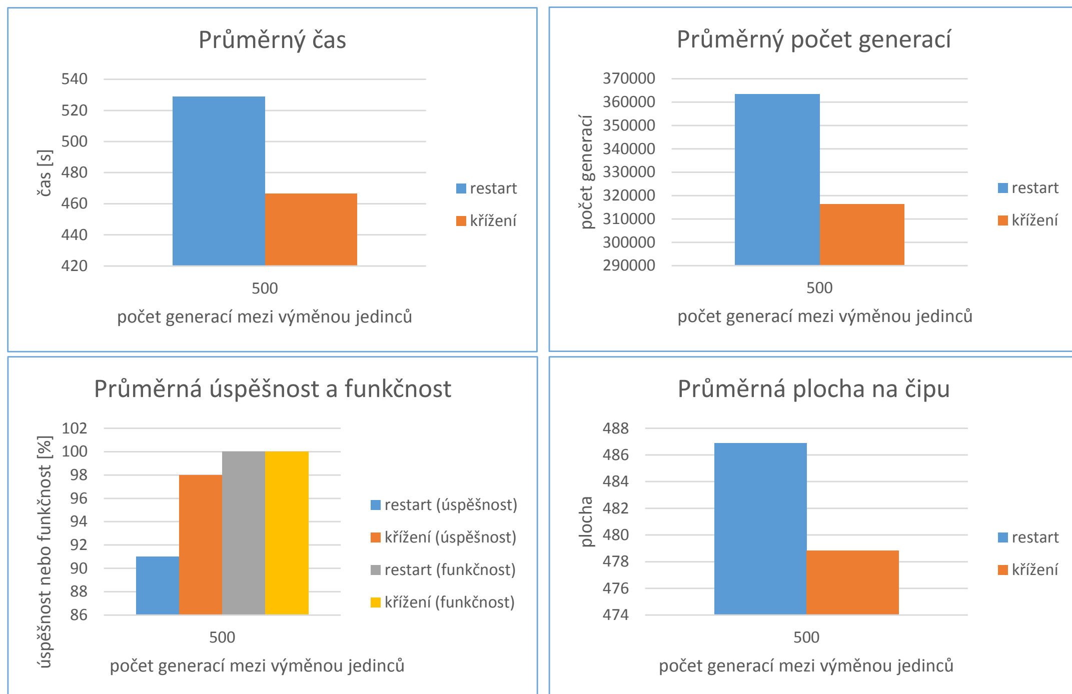
Tabulka 5.16: Výsledky osmibitové sčítačky na pěti uzlech



Obrázek 5.16: Grafy statistik při návrhu osmibitové sčítačky na pěti uzlech

výměna	typ		čas [s]	generace	hamming	funkčnost	úspěšnost	plocha
500	restart	min	63,00692	42500	0	99,84		338
		max	1732,70561	900000	960	100		692
		avg	528,6738199	363465	28,88	99,9952	91	486,88
500	křížení	min	84,42628	58000	0	99,76		304
		max	1477,34916	900000	1408	100		676
		avg	466,5006318	316445	16	99,9973	98	478,82

Tabulka 5.17: Výsledky osmibitové sčítačky na šesti uzlech



Obrázek 5.17: Grafy statistik při návrhu osmibitové sčítačky na šesti uzlech

menší počet generací k nalezení řešení. Obě metody vždy našly řešení a tudíž je funkčnost ve všech případech stoprocentní. Pro zajímavost výrazně menší průměrnou plochu na čipu našlo křížení při výměně po 1000 generacích. Při ostatních výměnách našla řešení s průměrně menší plochou na čipu původní varianta, ale už s nepatrnějším rozdílem od křížení.

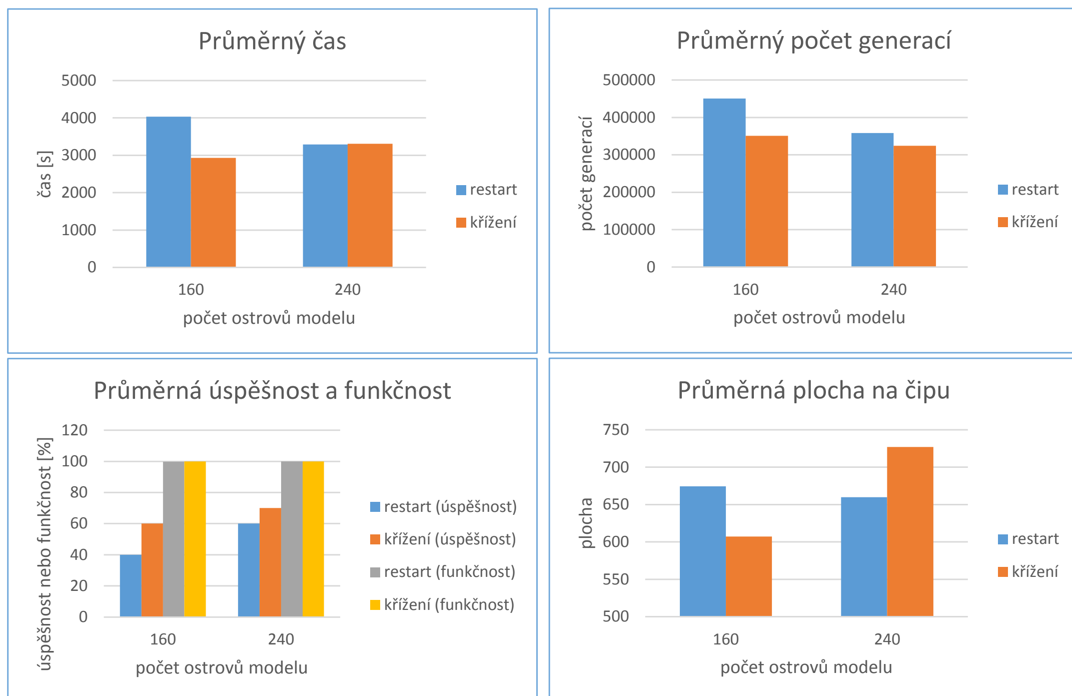
5.8 Tříbitová násobička

Návrh tříbitové násobičky byl testován na 16 a 24 ostrovech modelu. Maximální počet generací byl stanoven na 100 000. Pro obě varianty byla vyzkoušeny výměna jedinců po 100, 500 a 1000 generacích

V tabulce 5.20 a grafech na obrázku 5.20 jsou zobrazeny výsledky z testování na 16 ostrovech modelu. Vyplývá z nich, že křížení bylo výhodnější při výměně po 1000 generacích a naopak původní varianta v ostatních případech. Tyto výsledky jsou patrné ze všech zkoumaných kritérií. Ovšem mimo průměrného počtu potřebných generací jsou rozdíly mezi variantami poměrně malé. Jen původní variantě se podařilo vždy nalézt řešení s menší

ostrovů	typ		čas [s]	generace	hamming	funkčnost	úspěšnost	plocha
160	restart	min	2518,98836	304000	0	98,72		556
		max	5595,48052	500000	33600	100		854
		avg	4032,83426	450700	5347,2	99,8	40	674,4
160	křížení	min	1354,3376	154000	0	99,38		436
		max	4687,03894	500000	16384	100		768
		avg	2928,15555	350900	2278,4	99,91	60	607,2
240	restart	min	835,85667	128000	0	99,85		570
		max	5001,7421	500000	3840	100		778
		avg	3289,16447	358300	857,6	99,97	60	659,8
240	křížení	min	950,82043	108000	0	99,75		612
		max	6588,83619	500000	6464	100		814
		avg	3307,36363	324200	1203,2	99,95	70	727

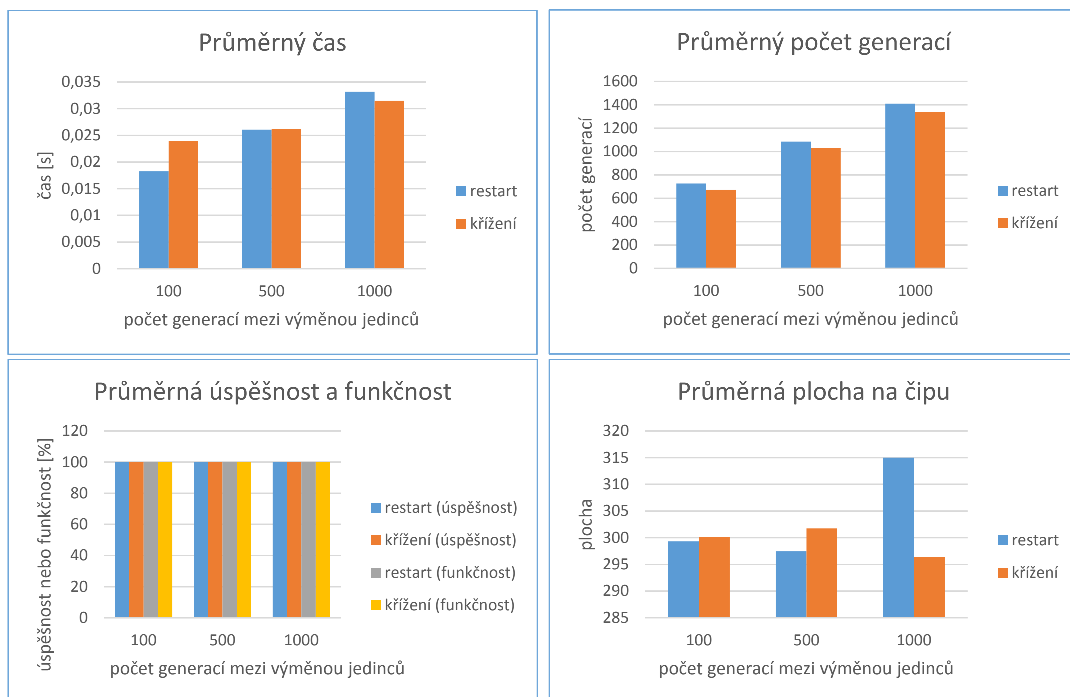
Tabulka 5.18: Výsledky devítibitové sčítačky



Obrázek 5.18: Grafy statistik při návrhu devítibitové sčítačky

výměna	typ		čas [s]	generace	hamming	funkčnost	úspěšnost	plocha
100	restart	min	0,00663	300	0	100		156
		max	0,1304	5300	0	100		472
		avg	0,01824	727	0	100	100	299,3
100	křížení	min	0,0055	200	0	100		142
		max	0,06003	1900	0	100		534
		avg	0,02394	672	0	100	100	300,14
500	restart	min	0,01025	500	0	100		154
		max	0,06211	2500	0	100		548
		avg	0,02606	1085	0	100	100	297,46
500	křížení	min	0,01047	500	0	100		152
		max	0,05211	2000	0	100		476
		avg	0,02615	1030	0	100	100	301,76
1000	restart	min	0,01983	1000	0	100		126
		max	0,05133	2000	0	100		634
		avg	0,0332	1410	0	100	100	315
1000	křížení	min	0,01959	1000	0	100		142
		max	0,05733	2000	0	100		580
		avg	0,0315	1340	0	100	100	296,36

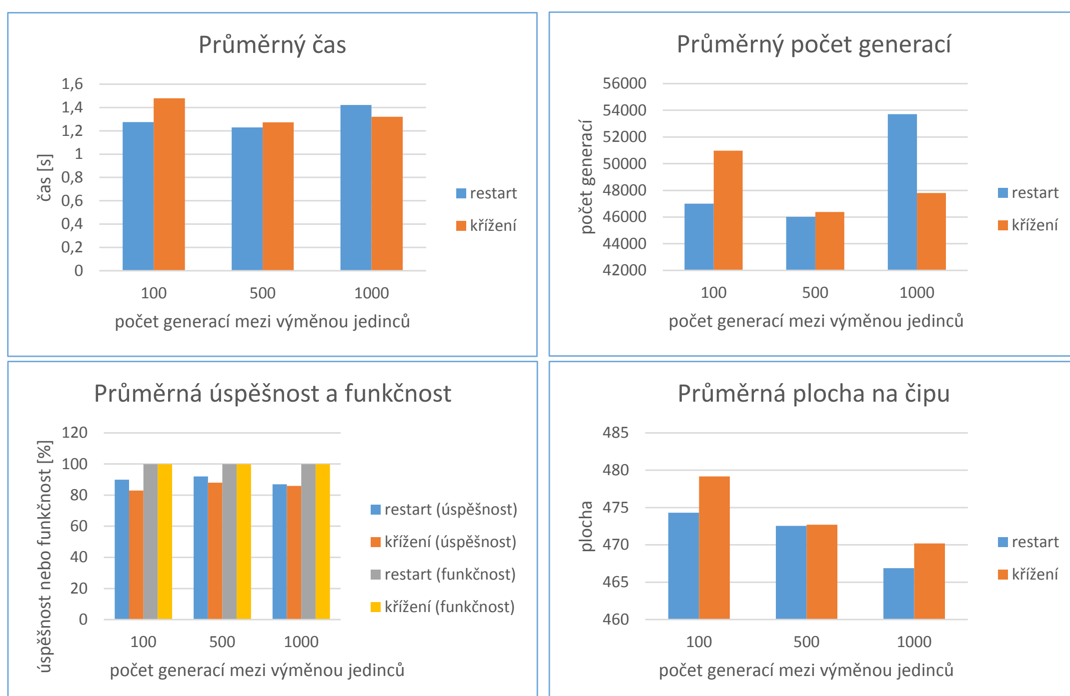
Tabulka 5.19: Výsledky dvoubitové násobičky



Obrázek 5.19: Grafy statistik při návrhu dvoubitové násobičky

výměna	typ		čas [s]	generace	hamming	funkčnost	úspěšnost	plocha
100	restart	min	0,13798	5400	0	98,7		306
		max	2,92498	100000	5	100		708
		avg	1,27575	47001	0,2	99,94	90	474,32
100	křížení	min	0,26634	7900	0	98,96		278
		max	3,13657	100000	4	100		730
		avg	1,47995	50973	0,2	99,93	83	479,18
500	restart	min	0,27054	10500	0	98,7		302
		max	2,85868	100000	5	100		682
		avg	1,23008	46015	0,1	99,96	92	472,54
500	křížení	min	0,25937	9000	0	99,22		282
		max	2,95761	100000	3	100		684
		avg	1,27285	46380	0,2	99,94	88	472,7
1000	restart	min	0,32086	13000	0	99,48		340
		max	2,91398	100000	2	100		704
		avg	1,42166	53710	0,2	99,95	87	466,88
1000	křížení	min	0,29984	12000	0	98,96		324
		max	3,04994	100000	4	100		722
		avg	1,321	47800	0,2	99,94	86	470,2

Tabulka 5.20: Výsledky tříbitové násobičky na 16 ostrovech



Obrázek 5.20: Grafy statistik při návrhu tříbitové násobičky na 16 ostrovech

plochou na čipu, jejíž optimalizace, ale nebyla předmětem testování.

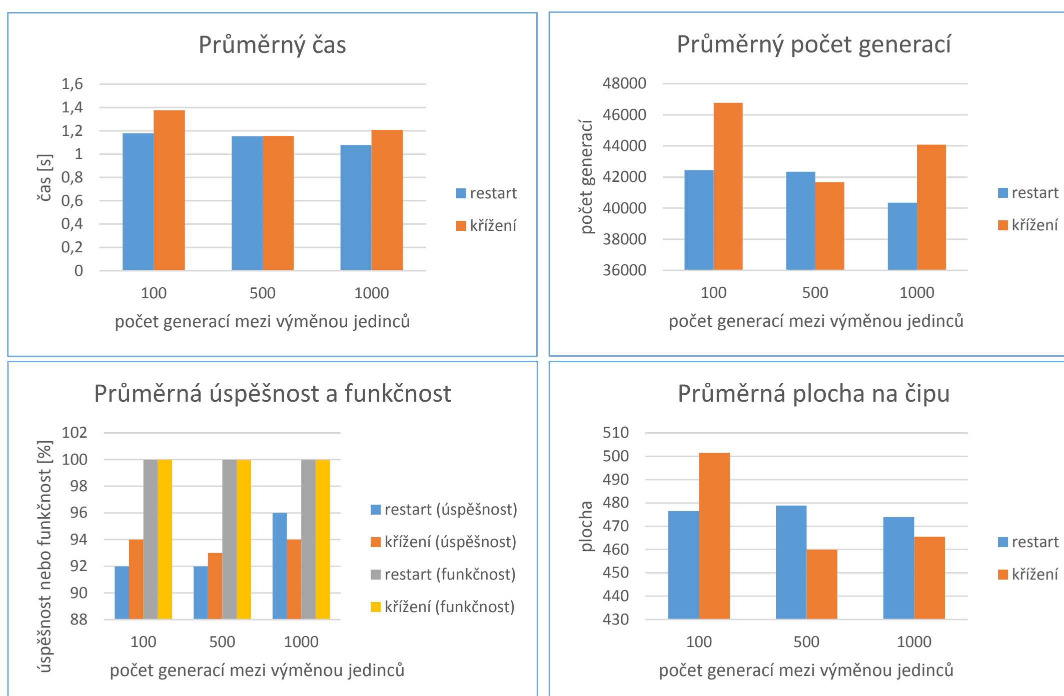
Testování návrhu tříbitové násobičky na 24 ostrovech modelu je shrnuto v tabulce 5.21 a grafech na obrázku 5.21. Z hlediska průměrného času byla lepší původní varianta až na výměnu jedinců po 500 generacích, kdy byly obě varianty stejně rychlé a křížení přitom v průměru stačil menší počet generací. Při výměně po 100 a 1000 generacích naopak stačil původní variantě v průměru menší počet generací. Více správných řešení našlo křížení při výměně po 100 a 500 generacích a při výměně po 1000 byla úspěšněnjší původní varianta. Řešení s průměrně menší plochou na čipu našlo křížení při výměně po 500 a 1000 generacích a původní varianta při výměně po 100 generacích.

5.9 Čtyřbitová násobička

Návrh čtyřbitové násobičky je oproti tříbitové velice náročný. Postupně jsem zkoušel potřebný počet generací a ostrovů pro návrh až jsem se dostal na 2,5 milionu pro maximální počet generací a 120 ostrovů modelu. A ani tak nejsou výsledky moc dobré, protože se mi podařilo nalézt jen málo stoprocentně funkčních řešení. Kompletní výsledky z testování jsou v tabulce 5.22 a grafech na obrázku 5.22. Průměrny čas má lepší původní varianta, ale je to dáno hlavně tím, že většina testů skončila dosažením maximálního počtu generací, aniž by nalezla řešení. Průměrný čas na vyhodnocení jedné generace má křížení větší, právě protože provádí samotné křížení, které nějaký čas zabere. Ovšem i jak se ukázalo zde, stačí mu v průměru menší počet generací. Je to dáno také tím, že křížení našlo vícekrát řešení problému tzn. bylo úspěšnější. Avšak ve všech bězích, i když nebylo nalezeno řešení, byly obě metody velice blízko jeho nalezení. Tomu odpovídá průměrná funkčnost, která se i tak blíží stu procent. Původní variantě se podařilo v průměru nalézt řešení s menší plochou na čipu, ale při tak málo nalezených řešeních to není vypovídající ukazatel.

výměna	typ		čas [s]	generace	hamming	funkčnost	úspěšnost	plocha
100	restart	min	80,91905	58400	0	99,69		352
		max	1907,48852	900000	1824	100		680
		avg	564,2487441	370431	44	99,9925	94	486,94
100	křížení	min	77,75695	59700	0	99,8		334
		max	1658,06697	900000	1152	100		734
		avg	530,991147	353978	22,4	99,9961	95	492,4
500	restart	min	109,24653	81500	0	99,85		338
		max	1546,39095	900000	896	100		660
		avg	597,1287818	413325	23,84	99,996	92	481,72
500	křížení	min	59,38205	43000	0	90,09		308
		max	2006,58184	900000	58432	100		762
		avg	600,8219804	406060	602,56	99,8977	93	481,16
1000	restart	min	83,32913	68000	0	99,65		300
		max	1489,25593	900000	2048	100		652
		avg	542,8416534	407740	55,36	99,9906	92	465,36
1000	křížení	min	80,50087	71000	0	99,85		320
		max	1350,49144	900000	896	100		674
		avg	509,9511306	378210	19,36	99,9967	96	463,92

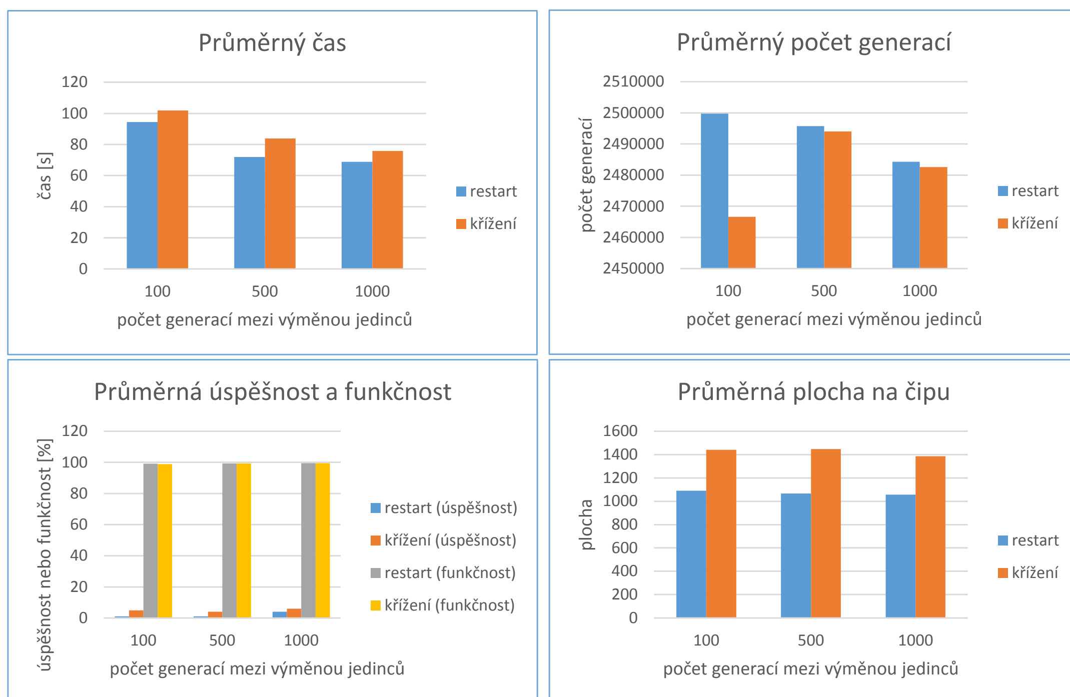
Tabulka 5.21: Výsledky tříbitové násobičky na 24 ostrovech



Obrázek 5.21: Grafy statistik při návrhu tříbitové násobičky na 24 ostrovech

výměna	typ		čas [s]	generace	hamming	funkčnost	úspěšnost	plocha
100	restart	min	88,24182	2473700	0	97,12		928
		max	99,52444	2500000	59	100		1274
		avg	94,35869	2499737	16,3	99,21	1	1090,06
100	křížení	min	59,50938	1474500	0	90,92		622
		max	127,78678	2500000	186	100		2370
		avg	101,79531	2466632	22,6	98,9	5	1441,14
500	restart	min	61,10334	2075000	0	97,27		898
		max	77,42382	2500000	56	100		1238
		avg	71,90475	2495750	13,6	99,34	1	1066,74
500	křížení	min	66,81412	2161500	0	92,29		670
		max	113,284	2500000	158	100		2394
		avg	83,91914	2494015	13,9	99,32	4	1446,44
1000	restart	min	54,7005	1958000	0	97,36		796
		max	74,66531	2500000	54	100		1288
		avg	68,84927	2484280	12	99,42	4	1056,4
1000	křížení	min	46,68451	1631000	0	95,51		716
		max	94,56955	2500000	92	100		2724
		avg	75,84555	2482560	9,8	99,52	6	1385,22

Tabulka 5.22: Výsledky čtyřbitové násobičky



Obrázek 5.22: Grafy statistik při návrhu čtyřbitové násobičky

Kapitola 6

Závěr

V rámci diplomové práce jsem se zabýval problematikou evolučních algoritmů. Dále jejich aplikací při návrhu kombinačních obvodů, na což se nejvíce hodí CGP. Také možností využití počítačových clusterů pro tento typ úloh, přičemž nejvhodnější je použití ostrovních modelů. Protože u ostrovních modelů je vhodná rekombinace, ale CGP ji nepodporuje, byla proto v rámci mojí diplomové práce navržena a naimplementována. Následně byla otestována a porovnána s původní variantou.

Z testování návrhu sčítaček na počítačové clusteru vyplynulo, že křížení nepřináší zásadní zlepšení pro ostrovní modely s CGP, ale na druhou stranu ani výrazné zhoršení. Prakticky při návrhu každé sčítačky existují nastavení paramertů CGP takové, že křížení je lepší než původní varianta, ale i takové, kdy je naopak lepší původní varianta. Proto se domnívám, že křížení má svůj smysl a také značný potenciál. Je pouze potřeba vyvážit čas, který zabere samotné křížení a čas, který je nechán pro izolovaný vývoj na jednotlivých ostrovech. Také by se dal zlepšit okamžik, kdy dochází ke komunikaci mezi ostrovy. Nebyl by dán počtem provedených generací mezi komunikací, ale právě uplynulým časem od poslední komunikace. Zabránilo by se tím zbytečnému čekání na pomalé ostrovy, kterým trvá vyhodnocování jedné generace v průměru i s křížením déle. Toto čekání by bylo vykryto smysluplným výpočtem dalšího vývoje.

Nyní jsem při křížení počítal pouze s nejlepším jedincem na ostrově a globálním nejlepším jedincem systému. Dále by se dalo zabývat možností, že se bude křížit nejen mezi těmito dvěma jedinci, ale mezi nejlepšími jedinci ze všech ostrovů. Znamenalo by to, že by v daném okamžiku výměny potomků byly rozdistribuováni nejlepší potomci ze všech ostrovů na všechny ostrovy. Na každém ostrově by probíhalo křížení mezi všemi takovými jedinci a následně by z nich vzešel nový potomek pro další izolovaný běh ostrova.

Největší výhodou křížení oproti původní variantě je zavedení jisté diverzity mezi ostrovy. Každý ostrov počítá se stejně dobrým jedincem, ti mají stejnou fitness hodnotu, ale už nemají stejný chromozom. Tomu odpovídá stejný fenotyp, ale rozdílný genotyp. Je tudíž větší šance na rychlejší konvergenci při hledání řešení problému. Také tu je potenciální výhoda pro multikriteriální návrh [1], kdy hledáme zároveň více možných řešení, které pak tvoří Paretovu frontu. Například můžeme hledat řešení, které bude zabírat nejmenší plochou na čipu a současně bude počítat s nejmenším zpožděním, což jsou potenciálně protichůdné požadavky.

Literatura

[1] Hrbáček, R.: Parallel Multi-Objective Evolutionary Design of Approximate Circuits. *GECCO'15*, July 11-15 2015, aCM 978-1-4503-3472-3/15/07.

[2] Hrbáček, R.; Sekanina, L.: Towards Highly Optimized Cartesian Genetic Programming: From Sequential via SIMD and Thread to Massive Parallel Implementation. *GECCO '14 Proceedings of the 2014 conference on Genetic and evolutionary computation*, 2014: s. 1015–1022, iSBN 978-1-4503-2662-9.

[3] IT4Innovations: Anselm Cluster Documentatio. `https://docs.it4i.cz/anselm-cluster-documentation`, [cit. 13. 5. 2015].

[4] Miller, J. F.: *Cartesian Genetic Programming*. Springer Berlin Heidelberg, 2011, iSBN 978-3-642-17310-3.

[5] Schwarz, J.; Sekanina, L.: *Aplikované evoluční algoritmy EVO*. FIT VUT v Brně, 2006.

[6] Sekanina, L.; Walker, J. A.; Kaufmann, P.; aj.: *Evolution of Electronic Circuits*. Springer Berlin Heidelberg, 2011, s. 125–179, iSBN 978-3-642-17310-3.

[7] Tomassini, M.: *Spatially Structured Evolutionary Algorithms*. Springer Berlin Heidelberg New York, 2005, iSBN 978-3-540-24193-5.

[8] Wikipedia: OpenMP. `http://en.wikipedia.org/wiki/OpenMP`, 12. 12. 2014 [cit. 4. 1. 2015].

[9] Wikipedia: Počítačový cluster. `http://cs.wikipedia.org/wiki/Počítačový_cluster`, 2. 2. 2015 [cit. 8. 3. 2015].