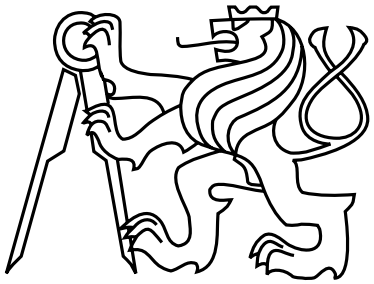
Na tomto místě bude oficiální zadání vaší práce

- Toto zadání je podepsané děkanem a vedoucím katedry,
- musíte si ho vyzvednout na studijním oddělení Katedry počítačů na Karlově náměstí,
- v jedné odevzdané práci bude originál tohoto zadání (originál zůstává po obhajobě na katedře),
- ve druhé bude na stejném místě neověřená kopie tohoto dokumentu (tato se vám vrátí po obhajobě).

České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů



Diplomová práce

Paralelizace logické syntézy

Bc. Ondřej Šírek

Vedoucí práce: Ing. Petr Fišer, Ph.D.

Studijní program: Elektrotechnika a informatika, strukturovaný, Navazující magisterský

Obor: Výpočetní technika

29. prosince 2011

Poděkování

Mé poděkování patří vedoucímu projektu Ing. Petrovi Fišerovi, Ph.D., za jeho pomoc při tvorbě mé diplomové práce a Gabriele Tschiedelové za pomoc při korekci gramatiky.

Prohlášení

Prohlašuji, že jsem práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.
Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 2. 1. 2012 .....

Abstract

The purpose of this thesis is to create a parallel application for the synthesis tool ABC. The work is divided into three parts.

In the first part of this work the possibility of implementation work is analyzed and discussed. There is discussed how to use the ABC tool for parallelization of logic synthesis, and how to incorporate it into a parallel environment.

Another part is the description of the implementation of solutions. This section contains a description of program structure and its key parts.

The last part are measurement results, processing and subsequent evaluation. This section describes how to launch the program on different platforms. Then there is described how the testing was processed and how were measured results and their final evaluation.

In the result evaluation was confirmed that it is possible to achieve better results by parallelization of logic sysnthesis than by sequential execution. Using parallelization can find an a sequence of commands that is suitable for sequential execution.

Abstrakt

Účelem této práce je vytvořit paralelní aplikaci pro syntezní nástroj ABC. Práce je rozdělena do tří částí.

V první části práce jsou analyzovány a diskutovány možnosti implementace práce. Je zde rozebráno, jakým způsobem využít nástroj ABC pro paralelizaci logické syntézy, a jak ho zakomponovat do paralelního prostředí.

Další částí je popis implementace samotného řešení. Tato část obsahuje popis struktury programu a jeho nejdůležitějších částí.

V poslední části jsem provedl měření výsledků, jejich zpracování a následné vyhodnocení. V této části je popsáno zprovoznění programu pod různými operačními systémy. Dále je zde popsán průběh testování, jakým způsobem byly zpracovány naměřené výsledky, a jejich konečné zhodnocení.

Ve vyhodnocení výsledků bylo potvrzeno, že paralizací logické syntézy je možno dosáhnout lepších výsledků než sekvenčním vykonáváním. Pomocí paralelizace můžeme naleznout sekvenci příkazů, která je vhodná pro sekvenční vykonávání.

Obsah

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

Cílem této práce je vytvořit paralelní spouštění různých syntézních kroků nad různými daty, a to pomocí nástroje logické syntézy *ABC*. Nejprve je třeba se seznámit se samotným systémem *ABC*.

1.1 Nástroj na logickou syntézu ABC

ABC je rostoucí softwarový systém pro syntézu a ověřování binárních sekvenčních logických obvodů, objevujících se v synchronních hardwarových projektech. Vývoj projektu probíhá na kalifornské univerzitě v Berkeley.

Nástroj obsahuje základní datové struktury pro reprezentaci a manipulaci kombinačních a sekvenčních, na technologii nezávislých, sítí.

Pro vstup ze souboru a výstup do souboru používá tento nástroj několika formátů. Binární BLIF, binární PLA, formát BENCH, Verilog a další. Dále využívá procedury pro detekci strukturně rozdílných vyjádření stejných Booleovských funkcí (FRAIG).

Více o tomto systému je možno se dozvědět na webu projektu [6]

1.1.1 AIG (And-Invertor Graf)

And-Invertor graf (AIG) je orientovaný, acyklický graf, který představuje strukturální realizaci logické funkčnosti obvodu nebo sítě.

AIG se skládá ze dvou-vstupových uzlů, reprezentujících logický operátor AND, koncových uzlů, označených názvy proměnných, a hran, které volitelně obsahují značky označující logickou negaci.

Převod ze sítí logických hradel na AIG je velmi rychlý a škálovatelný. Stačí každé hradlo vyjádřit pomocí hradel AND a invertorů.

1.1.2 Kombinační syntéza

Jelikož se tato práce zabývá především paralelizací logické syntézy, je třeba poukázat na to, jak syntéza probíhá v tomto nástroji.

Rychlá a efektivní syntéza je dosažena DAG-aware přepisováním [5] AIG (And-Invertor Graf). Přepisování se provádí pomocí knihovny předpočítaných řezů AIG nebo collapsováním a refactoringem logiky kuželů s 10-20 vstupy.

Pro před-počítávání řezů AIG se pak používá příkaz *rewrite*, zkráceně označován jako *rw*. Pro collapsování [5] a refactoring [5] se používá příkaz *refactor*, zkráceně označován jako *rf*.

Experimentálně je prokázáno, že iterativní volání těchto dvou transformací a jejich prokládání vyvažováním AIG příkazem *balance*, zkráceně označovaným jako *b*, výrazně snižuje velikost AIG, a má tendenci snižovat počet úrovní AIG.

Kombinací těchto základních příkazů pro syntézu je možné vytvořit skripty, které se využívají pro logickou syntézu. Prozatím nejznámějšími jsou skripty *resyn*.

Stručný popis jednotlivých příkazů:

- **balance** - Předpokládá, že vstup je AIG a vytváří ekvivalentní AIG. To má minimální zpoždění (měřeno pomocí logických úrovní dvou-vstupého hradla AND). Invertory se nezapočítávají do počtu logických úrovní. Příkaz balance má dvě fáze. První fází je pokrývání AIG, při kterém dochází k hledání AND hradel s více vstupy. Druhou fází je vyvažování stromu, kde dochází k vyvažování uzlů v grafu AIG, zbylých po nalezení všech AND hradel s více vstupy ve fázi pokrývání AIG.
- **refactor** - Hledá reconvergence-driven řez [4]. Po nalezení tohoto řezu jej převede do faktorizované formy. Pokud nahrazením pomocí této faktorizované formy dojde ke zlepšení AIG grafu, tato faktorizovaná forma použije.
- **rewrite** - Provádí hledání specifických řezů. Pomocí srovnávání s řezy v před-počítané tabulce může provést náhradu v AIG grafu. Tímto lze dosáhnout zlepšení tohoto grafu.

U příkazů rewrite a refactor je možné použít přepínač *-z* (zero-cost replacements). V případě použití tohoto přepínače se při nahrazování částí AIG grafu vezme v potaz i varianta, při které tímto nahrazením nedojde k žádnému zlepšení ani zhoršení počtu AND uzlů. Smyslem využití tohoto přepínače je to, že díky němu může vzniknout jiná struktura AIG grafu, která může v dalším použití vést ke zlepšení.

1.2 Zpracování zadání

Pro vytvoření paralelizace tohoto systému je nutno prostudovat jeho možné využíti a zjistit, jakým způsobem by bylo možno ho využít v paralelním prostředí. Základním krokem by mělo být určení, jakým způsobem se bude nástroj ABC v paralelním prostředí spouštět.

V dalším kroku bude třeba určit, na jaké úrovni a jakým způsobem si budou systémy mezi sebou vyměňovat informace, a jak spolu budou komunikovat. Z důvodu, že se jedná o neustále se vyvíjející systém, je třeba zajistit, aby propojením pokud možno vznikl co nejmenší zásah do systému. Pokud by se to podařilo, bylo by možno v budoucnu využívat nejnovější verze *ABC*.

Dále bychom měli prostudovat možnosti využití paralelního prostředí a stanovit si, jaké paralelní prostředí zvolit pro samotnou implementaci. Po výběru prostředí se udělá návrh jeho struktury a komunikace.

Hlavním cílem by pak ve výsledku měla být aplikace, která při využití paralelizace tohoto systému bude v nejlepším případě dosahovat lepších výsledků než sekvenční provádění *ABC*. Další motivací je nalezení určité sekvence příkazů, která by u většiny sítí vedla ke zlepšení. Tu by pak bylo možno využít jako nový skript nástroje *ABC*, jenž by se mohl využívat při logické syntéze.

Stávající skripty nástroje *ABC* jsou složeny z různých elementárních příkazů. Ty jsou v této sekvenci nějakým způsobem poskládané. Jestli je jejich posloupnost správná či nikoliv není jasné.

Spouštění těchto skriptů se provádí iterativně. Motivací je naleznout správnou sekvenci správných příkazů. Tím docílíme dobrého iterativního procesu.

Abychom dosáhli tohoto cíle, budeme spouštět náhodně elementární příkazy a pokusíme se při zpracování výsledků odhalit správnou posloupnost. Aby se zvýšila šance, že tuto sekvenci odhalíme co nejdříve, zvolíme pro práci paralelní prostředí.

K dosažení stanovených cílů je třeba takto vytvořenou aplikací provést testování nad velkým množstvím dat. Jedině to nám zaručí, že dosažené výsledky budou směrodatné. V neposlední řadě je pak nutno naměřené výsledky nějakým způsobem zpracovat a vyhodnotit.

Kapitola 2

Analýza a návrh řešení

Před vlastní implementací je důležité si stanovit, jakým způsobem budeme provádět paralelizaci logické syntézy. Je zde několik alternativních způsobů provedení. V této části jsou tedy jednotlivé varianty stručně popsány, spolu s jejich klady a zápory. Na základě hodnocení je pak vybráno nejlepší řešení, které je popsána v implementační části.

U výběru způsobu implementace je nutno počítat jak s výběrem paralelního prostředí, tak se samotným způsobem paralelizace.

2.1 Použití nástroje ABC v implementaci

Nejdůležitějším bodem u nástroje *ABC* (z hlediska implementace) je určení způsobu spouštění. V podstatě jsou zde dvě varianty provedení.

- Spuštění pomocí spustitelného souboru
- Spuštění pomocí statické knihovny

2.1.1 Spuštění pomocí spustitelného souboru

První variantou je spouštění spustitelného souboru. Hlavní problém této varianty spočívá v tom, že spustitelný soubor je spouštěn programově. Jednotlivé příkazy lze pak zadávat pouze pomocí skriptu, který obdrží program při spuštění.

Realizace by mohla vypadat tak, že se před spuštěním vygeneruje každému procesu skript. Ten bude obsahovat příkazy k vykonání, spolu s obvody, nad kterými se budou provádět. Jednotlivé procesy pak budou spouštět spustitelný soubor *ABC* s takovýmto skriptem. Ve výsledku dojde ke ztrátě veškeré efektivity paralelního vykonávání, neboť není předem známé, jak dlouho bude který skript zpracováván. Mohlo by však i stát, že se jeden proces bude stále pracovat, zatímco ostatní budou nevytíženy. Z výše uvedených důvodů není tento způsob vhodný k implementaci.

Obměnou tohoto způsobu je generování skriptu samotnými procesy. Každý proces bude generovat skript pouze na základě právě obdrženého příkazu a obvodu. Poté se spustí spustitelný soubor s takto vygenerovaným skriptem. Tímto se zabrání nevytíženosti procesů a

bude možno rozdělovat práci naopak podle jejich vytíženosti. Dojde k tomu, že se pro každý příkaz spustí instance *ABC*. To je oproti variantě jednoho spuštění výpočetně náročnější. Z tohoto důvodu byl i tento návrh implementace zavržen a hledalo se lepší řešení.

2.1.2 Spuštění pomocí statické knihovny

Nástroj *ABC* obsahuje také možnost kompilace statické knihovny. Tento způsob velice usnadňuje práci s *ABC*, neboť je možnost ho jednou spustit a pak libovolně vykonávat příkazy podobně, jak je tomu při jejich ručním zadávání. Při spuštění *ABC* jako statické knihovny jsou ke komunikaci s tímto nástroje využívány následující funkce:

- **Abc_Start()** – Tato funkce slouží k samotnému spuštění *ABC*. Funkce využívá Abc_FrameGetGlobalFrame(), jež alokuje paměť pro rámec *pAbc*, který je využíván pro komunikaci s *ABC* a zajistí jeho inicializaci. Tato funkce se nachází v souboru *mainLib.c*.
- **Abc_Stop()** – Po zavolání této funkce, nacházející se v *mainLib.c*, se provede ukončení rámce *pAbc* a poté jeho dealokace. Tímto se zajistí ukončení celého *ABC*.
- **Abc_FrameGetGlobalFrame()** - Tato funkce, kterou využívá jak **Abc_Start()**, tak **Abc_Stop()** slouží k předání rámce *pAbc*. Toho využíváme k získání reference na *pAbc* i v našem programu, abychom ji následně mohli předat dalším funkcím. Funkce se nachází v souboru *mainFrame.c*.
- **Cmd_CommandExecute(void * pAbc, char * sCommand)** – Tato funkce se nachází v souboru *cmdApi.c* a slouží k samotnému vykonávání příkazů *ABC*. Uloží příkaz do historie *ABC*. Poté zjistí, nejedná-li se o skript. Pokud ano, rozdělí ho na jednotlivé příkazy, a ty postupně vykonává.

2.1.3 Přesměrování standardního výstupu do souboru

U obou variant spouštění nástroje *ABC* je zásadní problém se zpracováním výstupu. Ať už se *ABC* spouští jako spustitelný soubor, nebo jako statická knihovna, veškeré výstupy z něho jdou na standardní výstup.

Při řešení této komplikace musíme brát ohled na to, že nástroj *ABC* se neustále vyvíjí. Je třeba upravit přesměrování standardního výstupu za pomoci co nejmenšího zásahu do tohoto nástroje, abychom v budoucnu mohli provést tyto změny na aktuální verzi.

Pro naše potřeby nám stačí přesměrovat pouze výstup příkazu *print_stats*, který poskytuje údaje o obvodu. Ostatní výstupy není třeba zachovávat, neboť se s nimi dále nepracuje.

Projitím části kódu nástroje *ABC* se určily změny nutné k předělání aktuální verze *ABC*. To, že je lze aplikovat i na novějších verzích, bylo vyzkoušeno při samotném testování.

Poprvé by tyto změny aplikovány na starší verzi stažené z [6]. Před samotným měřením pak byla do aplikace zapracována nejnovější verze *ABC*, která byla stažena z úložiště tohoto projektu [7]. Tím bylo ověřeno, že je možno tímto způsobem upravit jakoukoliv novou verzi nástroje *ABC* a využít ji jako statickou knihovnu v této práci.

Změny v nástroji *ABC*, zajišťující přesměrování výstupu příkazu *print_ stats* do souboru, jsou popsány v kapitole 3.4.
Pro získaní požadovaných dat si pak stačí přímo ve vytvořené aplikaci otevřít příslušný soubor pro nástroj *ABC*, vykonat příkaz a přečíst výsledek ze souboru.

Při testování došlo k následujícímu problému. Pokud má systém *ABC* otevřený výstupní soubor po celou dobu jeho vykonávání, není možné z něho v aplikaci načítat výsledky zpracování. Z tohoto důvodu aplikace výstupní soubor před jeho otevřením zavře systému *ABC*.

Z funkčního hlediska by to neměl být problém. *ABC* v tuto chvíli nezpracovává žádný příkaz, a tudíž nevyžaduje přístup k výstupnímu souboru. Po přečtení dat z tohoto souboru se pak opět zpřístupní systému *ABC*. Proto se vytváří výstupní soubor pomocí následující funkce.

```
fopen ("filename","a+")
```

Ta funkce zajistí, že se při otevírání souboru zachovají předchozí data a nová data se zapisují na konec souboru. Pokud by k tomuto nedošlo, data by se opakovaným otevíráním souboru smazala.

Připisováním dat na konec souboru, dochází však k narůstání velikosti tohoto souboru. K dosažení aktuálních dat musíme navíc projít celý soubor a dostat se k poslední položce. Tyto nedostatky by bylo možno vyřešit jedním z následujících způsobů:

- Vytvoření funkce, která zajistí načítání posledního řádku souboru bez nutnosti procházet celý soubor.
- Smazaní obsahu souboru. To by se provedlo tak, že se vytvoří nový soubor stejného názvu pomocí následující funkce.

  ```
  fopen ("filename","w")
  ```

  Takto vytvořený soubor smaže obsah předchozího souboru se stejným názvem a zachází s ním jako s nově vytvořeným.

Jelikož předchozí data v souboru již byla zpracována, nejsou relevantní. Proto zvolíme variantu smazání souboru, která zároveň zbytečně nezvyšuje paměťovou náročnost. Samotná implementace výměny dat mezi *ABC* a aplikací je pak popsána v kapitole 3.5.

2.1.4 Parsovaní výsledků

Přesměrování do souboru, které bylo popsáno v předchozí kapitole, však vrací pouze statistiky o obvodu, které vypisuje *ABC* na standardní výstup. Je třeba tedy vytvořit nějaký parser, který projde tuto statistiku a získá z ní požadované údaje.

Tato aplikace se zaobírá zlepšováním obvodů. Proto nás zajímají údaje o počtu hradel AND a počtu úrovní. Tento parser byl založen na předchozí znalosti struktury výpisů nástroje *ABC*. Na základě analýzy těchto výpisů vyplynulo, že údaje jsou vypisovány v následujícím obecném formátu.

```
Networkname : parametr = hodnota parametr = hodnota parametr = hodnota
```

Po analýze bylo zjištěno, že jako rozdělovací znak je nejlépe zvolit „=“. Pokud se načte řetězec do znaku „=“, získáme při zpětném procházení této sekvence parametr a hodnotu. Parametr se obdrží při čtení ukončeném přečtením druhé mezery. Hodnota se pak rovná zbytku řetězce.

Jednotlivé parametry si musíme pamatovat a zpracovávat zpětně, neboť po přečtení prvního znaku „=“ získáme pouze parametr, ale nikoliv hodnotu. Ze stejného důvodu nesmíme na konci zpracování opomenout poslední hodnotu.

Při zpracování se pak parametr porovná s požadovaným údajem a při shodě se zapíše do výstupní struktury. Díky této struktuře procházení můžeme pro zpracování zvolit jakékoliv další údaje. Samotná implementace parseru je pak popsána v kapitole 3.6.

2.2 Způsoby paralelizace nástroje ABC

Cílem této práce je vytvořit paralelní spouštění různých syntézních kroků nad různými daty pomocí nástroje logické syntézy *ABC*. Paralelizaci tohoto nástroje můžeme chápat několika odlišnými způsoby, proto v následujících odstavcích uvádím potencionální uvažované způsoby implementace.

- Rozdělení vstupního obvodu a následné paralelní vykonávání viz. 2.2.1.
- Vykonávání příkazů nad předem danými obvody viz. 2.2.2.
- Vykonávání příkazů nad různými obvody viz. 2.2.3.

2.2.1 Rozdělení vstupního obvodu a následné paralelní vykonávání

Jedním z možných způsobů paralelizace je rovnoměrné rozdělení daného vstupního obvodu na části a následné paralelní vykonávání příkazů logické syntézy nad těmito částmi.

Struktura programu by v tomto případě byla následující: Řídící proces nejprve rozdělí obvod na dílčí části. Ty poté distribuuje mezi ostatní procesy, které nad částmi obvodu vykonávají dané syntézní kroky. Po vykonání syntézy předávají procesy své výsledky řídícímu procesu, který obvod opět skládá.

Tento cyklus se provádí nad každým syntézním krokem, který je obsažen ve spouštěcím skriptu. Tato varianta by měla přínos v rychlosti vykonávání. Dále je zde možnost, že by výsledky syntézy byly úspěšnější než při vykonávání syntézy nad celými obvody.

Předchozí tvrzení nemůžeme u syntézního nástroje *ABC* ani potvrdit, ani vyvrátit, dokud neprovedeme samotné testování.

Tato varianta je vhodná pro realizaci a mohla by poskytnout zajímavé výsledky. Nebyla však zvolena, neboť již byla realizována sekvenčním způsobem. Jediným přínosem paralelizace tohoto způsoby by byla rychlost provádění. Varianta tedy nebyla zvolena pro realizaci.

2.2.2 Vykonávání příkazů nad předem danými obvody

Jinou z možností paralelizace logické syntézy je provedení vybraných příkazů či skriptů nad předem danou skupinou obvodů.

V tomto případě program obdrží složku obsahující různé obvody a vykonávací příkaz (skript). Paralelně se vykonají zadané příkazy nad obvody po předem určený počet iterací.

Z hlediska zadaného tématu, při kterém se mají vykonávat různé syntézní kroky nad různými daty, je tato varianta naprosto bezpředmětná. Tuto variantu však zde uvádím proto, že nakonec byla zakomponována do implementace.

Důvodem byla možnost použít této verzi pro různé testovací případy. V své práci jsem tuto variantu použil například pro měření sekvenčního spouštění skriptů *resyn*.

Použití se ukázalo velice výhodné z časového hlediska. Variantu je možno použít pro jakékoliv příkazy (skripty) a obvody. Zakomponování do výsledné aplikace jsem provedl přes přepínač *-n* aplikace **MPI_abc**. Implementace této varianty je pak popsána v kapitole 3.3

2.2.3 Vykonávání příkazů nad náhodnými obvody

Poslední možností paralelizace je vykonávání vybraných příkazů *ABC* nad náhodně zvolenými obvody. Princip fungování této varianty je následující.

Program pracuje vždy s jedním vstupním obvodem. Použitím různých syntézních kroků se snaží tento obvod co možná nejvíce vylepšit. Jednotlivé syntézní kroky se vykonávají nad náhodně vybranými obvody.

Obvody se náhodně vybírají ze složky obvodů, určené k uchovávání obvodů. Všechny obvody obsažené v této složce jsou výsledky původního vstupního obvodu po provedení několika různých syntézních kroků. Po zpracování každého syntézního kroku se výsledný obvod uloží do složky obvodů, odkud se pak náhodně vybírají obvody pro zpracování.

Samotná paralelizace tohoto principu pak vypadá následovně. Program obdrží při spuštění libovolný vstupní obvod a vybrané syntézní příkazy. Každý proces, který zrovna nevykonává práci, zažádá řídící proces o obvod a příkaz, který nad ním má vykonat. Řídící proces mu zašle obvod, který náhodně vybere ze složky obvodů, a příkaz.

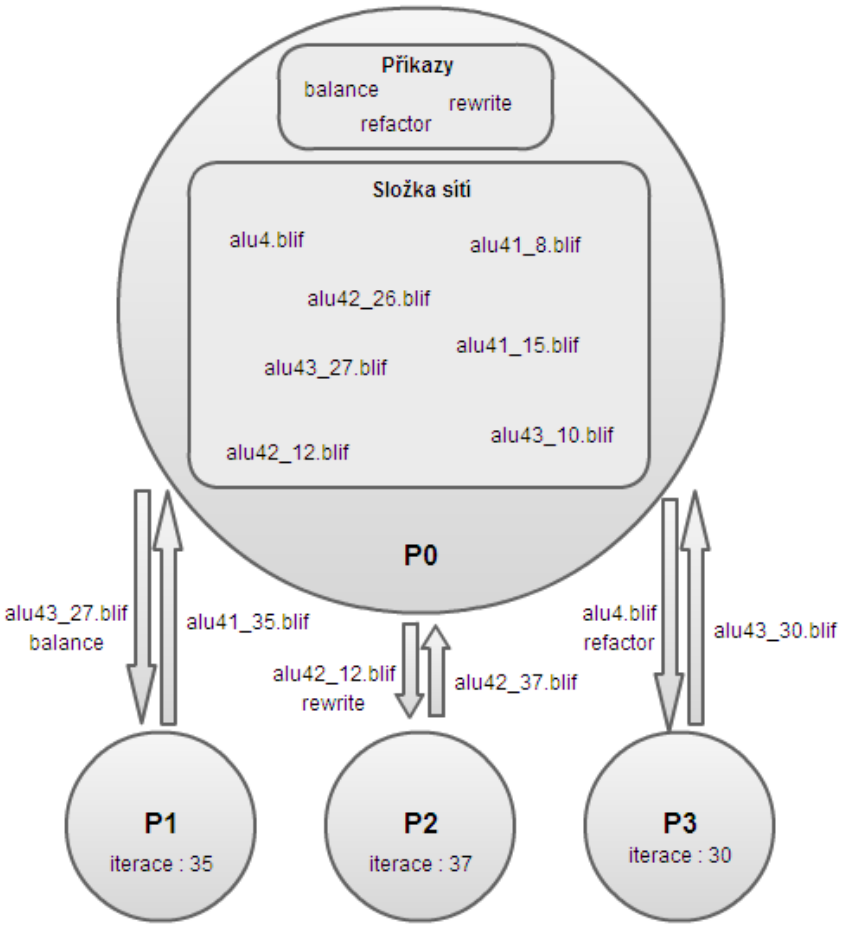
Při první žádosti o obvod složka obsahuje pouze vstupní obvod. Důvodem je, že nebyl proveden žádný syntézní krok a nebyl tedy žádný obvod uložen do složky obvodů. Procesy po první žádosti o obvod dostanou vstupní obvod programu. Nad ním vykonají určený syntézní příkaz a uloží výsledný obvod do složky obvodů. Poté se již ve složce obvodů nachází více obvodů. Je tedy možno při žádosti o obvod, vybrat náhodně jeden a z nich a zaslat.

Princip vykonávání a náhodného výběru obvodů je znázorněn na obrázku 2.1.

Tento způsob vykonávání se opakuje po zadaný počet iterací. Řídící proces si navíc ukládá informace o vykonávání, kvůli pozdější rekonstrukci provádění.

Varianta je výhodná jak z hlediska rychlosti, neboť je dobře škálovatelná, tak z hlediska využití potenciálu nástroje *ABC*.

Spouštění základních syntézních kroků nad náhodnými obvody může vést k posloupnosti příkazů, která je výhodná pro vykonávání obvodů. Tato varianta byla shledána jako nejvýhodnější způsob implementace a po dohodě s vedoucím práce byla zvolena pro realizaci.



Obrázek 2.1: Struktura vykonávání příkazů nad náhodnými obvody

2.3 Volba prostředí pro paralelní výpočty

Pro implementaci musíme zvolit vhodné prostředí pro paralelní výpočty. V tomto ohledu je velké množství možností.

V důsledku toho bylo hledáno co nejlepší řešení. Při studování různých porovnání těchto prostředí jsem objevil článek [8], který hodnotí jednotlivé varianty. Po prostudování závěrů tohoto materiálu byli do užšího výběru vybráni dva kandidáti, a to *MPI* (Message Passing Interface) a *PVM* (Paralel Virtual Machine). Pro správný výběr prostředí je třeba se podívat na jejich výhody a nevýhody.

2.3.1 Výhody a nevýhody MPI

Hlavní výhody MPI:

- Jakýkoliv paralelní algoritmus může být vyjádřen v paradigmatu MPI.

- Lze ho vykonávat jak na systému s distribuovanou pamětí, tak na systému se sdílenou pamětí.
- Umožňuje explicitní kontrolu komunikace, což vede k vysoké účinnosti souběhu komunikace a výpočtů.
- Problémy s umístěním dat jsou pouze zřídka.
- Paralelní prostředí MPI je přenositelné.
- Aktuální implementace jsou efektivní a optimální.

Nevýhody MPI:

- Obtížný vývoj aplikací. Přepisování stávající sériové aplikace pomocí MPI často vyžadují rozsáhlou restrukturalizaci sériového kódu.
- Nevýhodné pro malé problémy, kde převládají náklady na komunikaci.
- Dynamické vyvažování zátěže, je obtížně proveditelné.
- Existují varianty realizace celé knihovny MPI od různých výrobců. Některé nemusí obsahovat všechny funkce, zatímco jiné nabízejí různé rozšíření.

2.3.2 Výhody a nevýhody PVM

Hlavní výhody PVM:

- Pravděpodobně nejvíce přenositelné prostředí ze všech dostupných.
- Pružné a škálovatelné paralelní prostředí.
- Odolné vůči chybám dynamického přidávání a mazání procesů.
- Každý počítač UNIX, který je připojen k internetu se může stát součástí paralelního virtuálního stroje.

Nevýhody PVM:

- V závislosti na architektuře a realizaci, může PVM být pomalejší než jiná paralelní prostředí.
- PVM není standard (na rozdíl od MPI).
- To je nedostatek u některých funkcí zasílání zpráv.

Jelikož jsou tato dvě prostředí výkonnostně velmi podobná, bylo nutno určit si klady a zápory u jednotlivých variant. Na jejich základě jsem pak zvolil paralelní prostředí *MPI*. Mezi výhody této volby patří výkonnost tohoto prostředí, zkušenosti s prostředím z předchozího studia, jednoduché zprovoznění na testovacích strojích, a v neposlední řadě možnost využití výpočetního clusteru *STAR* pro testování.

2.3.3 MPI (Message Passing Interface)

Message Passing Interface (*MPI*) [9] je knihovna implementující protokol pro podporu paralelního řešení výpočetních problémů v počítačových clusterech. Jedná se o rozhraní na zasílání zpráv mezi jednotlivými uzly.

Je možno zasílat jak point-to-point zprávy, tak globální zprávy. Většina implementací používá jako transportní protokol TCP. *MPI* je nezávislé na programovacím jazyku. Jedná se především o síťový protokol. Při návrhu rozhraní byl vždy kladen důraz na výkon, škálovatelnost a přenositelnost.

MPI programy pracují vždy s procesy i přesto, že se často mluví o procesorech. Pro nejlepší výkon je třeba přidělit každému procesu jeden procesor. Tím se odbourá nutnost přepínání kontextu, která vede ke zpoždění.

K mapování jednotlivých procesů na procesory nedochází při překladu, nýbrž při samotném běhu programu. Mapování provádí agent, který spustil *MPI* program. Většinou se tento agent jmenuje *mpirun* či *mpiexec*.

MPI obsahuje velké množství funkcí mezi které patří následující:

- operace odeslání / přijetí point-to-point zprávy - blokující nebo neblokující
- výběr mezi kartézskou a grafovou topologií procesů
- výměna dat mezi dvojicemi procesorů
- kombinování mezivýsledků výpočtů
- synchronizace uzlů
- získávání informací týkajících se sítě, např. počet procesů, identita procesoru, na kterém běží daný proces, apod.

Pro komunikaci mezi procesy se používají komunikátory. To jsou skupiny procesů při běhu *MPI* aplikace, které se dají dynamicky vytvářet. Všechny procesy dané aplikace jsou obsaženy ve skupině *MPI_COMM_WORLD*. Jednotlivé procesy jsou identifikovány podle ranku, což je jeho pořadové číslo od nuly, uvnitř skupiny.

V *MPI* pak existují dva způsoby komunikace:

- Point-to-point komunikace
- Kolektivní komunikace

2.3.3.1 Point-to-point komunikace

Je určena pro realizaci komunikace mezi dvěma procesy. Klasickým příkladem je funkce **MPI_Send()**, která pošle zprávu od jednoho procesu jinému procesu. Tyto funkce se nejčastěji využívají pro komunikaci typu master-slave.

Blokující funkce:

```
int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source,
int tag, MP_Comm comm, MPI_Status *status)

int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm)
```

Neblokující funkce:

```
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source,
int tag, MPI_Comm comm, MPI_Request *request)

int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm, MPI_Request *request)
```

Jednotlivé parametry funkcí pak jsou:

- buf - ukazatel na data, která chceme odeslat, respektive na místo v paměti, kam chceme uložit přijatou zprávu
- count - počet přijatých hodnot
- datatype - datový typ, např. *MPI_ CHAR*, *MPI_ INT*, *MPI_ LONG*, apod.
- source - proces, od kterého chceme číst
- tag - identifikace zprávy, slouží spíše pro vizualizaci nebo ladění
- comm - handle komunikátoru
- status - informace o stavu operace

2.3.3.2 Kolektivní komunikace

Hromadná komunikace se používá při zasílání zpráv mezi všemi procesy. Nejběžnějším typem je **MPI_Bcast()**, který se využívá například jako synchronizační bariéra. Dalšími typy kolektivní komunikace pak jsou:

- **MPI_Bcast** – pošle zprávu od procesu s rankem 0 všem ostatním procesům ve skupině
- **MPI_Scatter** – pošle data od jedné úlohy všem ostatním ve skupině
- **MPI_Gather** – shromáždí hodnoty ze skupiny procesů
- **MPI_Reduce** – zredukuje data všech procesů na jednu hodnotu. Operaci redukce si lze vybrat v parametru op

```
int MPI_Bcast(void* buffer,int count,MPI_Datatype datatype,int root,
MPI_Comm comm);

int MPI_Scatter(void* sendbuf,int sendcount,MPI_Datatype sendtype,
void* recvbuf,int recvcount,MPI_Datatype recvcount,int root,MPI_Comm comm);

int MPI_Gather(void* sendbuf, int sendcount,MPI_Datatype sendype,
void* recvbuf,int recvcount,MPI_Datatype recvtype,int root,MPI_Comm comm);

int MPI_Reduce(void *sendbuf,void *recvbuf,int count,
MPI_Datatype datatype,MPI_Op op,int root,MPI_Comm comm)
```

Kapitola 3

Implementace paralelizace logické syntézy

Tato část práce obsahuje popis samotné implementace paralelizace logické syntézy za pomoci nástroje *ABC* a paralelního prostředí *MPI*.

Základem samotné implementace je cyklus *MPI*, ve kterém probíhá komunikace mezi jednotlivými procesy. Dále je nutno uvážit, jakým způsobem bude implementováno spouštění *ABC* a parsování výsledků.

3.1 Program MPI_abc

Výstupem implementace se stal program **MPI_abc**. Funkčnost tohoto programu je popsána v následujících kapitolách.

Program se skládá ze tří variant, které se spouští pomocí následujících přepínačů:

- **-r** - randomizovaná verze, u které se vybírají obvody pro vykonávání náhodně
- **-e** - randomizovaná verze s vestavěným elitismem, to spočívá v tom, že obvody vybírá náhodně jak u předchozí varianty, avšak při vybírání vybírá náhodně, jestli předá náhodný obvod, nebo prozatím nejlepší dosažený (pravděpodobnost vybrání prozatím nejlepšího obvodu se odvíjí od počtu syntézních příkazů předaných při spuštění aplikace, pravděpodobnost je 1 : Počet syntézních příkazů)
- **-n** - nerandomizovaná verze spouští předem dané příkazy nad obvody po určený počet iterací

3.2 Paralelní spouštění ABC

Před samotnou implementací kódu je třeba si ujasnit strukturu vykonávání. Program by měl pracovat následujícím způsobem:

Program na vstupu dostane název obvodu k vykonávání, příkazy, které bude provádět, a počet iterací. V jednotlivých cyklech se pak každému procesu zašle náhodně jeden obvod ze složky obvodů a příkaz, který nad ním má vykonat.

Složka obvodů při spuštění aplikace obsahuje pouze vstupní obvod. V průběhu vykonávání do ní jednotlivé procesy ukládají obvody po provedení příkazu. Tato složka obsahuje vždy pouze obvody, které vznikly ze vstupního obvodu, vykonáním různých syntézních kroků.

Proces tento příkaz vykoná a vytvořený obvod uloží do výše zmíněné složky obvodů. Jako zprávu o ukončení vykonávání pak zašle informace o obvodu a další potřebné údaje.

Z předchozího náznaku vykonávání pak plyne, že musíme vyčlenit jeden proces jako řídící (master) a zbylé jako podřízené (slave). Kdyby tomu tak nebylo, je zde možnost, že proces rozdělující práci ostatním bude při žádosti vykonávat příkaz, čímž se zpomalí celé provádění.

Před samotným cyklem paralelizace je nutno provést inicializaci všech procesů. Na začátku inicializace se u každého procesu spustí samotné *MPI* pomocí funkce **MPI_Init()**. Přes funkce **MPI_Comm_rank()** a **MPI_Comm_size()** je pak možno získat rank příslušného procesu a počet běžících procesů.

U řídícího procesu s rankem 0 se při inicializaci vytvoří datová struktura na uchovávání údajů o zpracovaných obvodech *array_net* a načtou se příkazy určené k vykonávání *Command_Array*. U ostatních procesů stačí při inicializaci spustit *ABC* pomocí funkcí **Abc_Start()** a **Abc_FrameGetGlobalFrame()**.

Samotné vykonávání programu se pak provádí v cyklu, ve kterém jsou posílány a odchytávány zprávy, na jejichž základě se vykonávají příkazy *ABC*. Samotná struktura cyklu pak vypadá takto:

```
while (true)
{
  counter++;
  if ((counter % CHECK_MSG_AMOUNT)==0)
  {
    if(read_enable)
    {
      MPI_Iprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD,
      &flag, &status);
    }
    if (flag)
    {
      switch (status.MPI_TAG)
      {
        case MSG_WORK_REQUEST :
        {
          state = ACTIVE;
          flag = 0;

          MPI_Recv(&from, 1, MPI_INT, MPI_ANY_SOURCE, MSG_WORK_REQUEST,
          MPI_COMM_WORLD, &status);
```

```
  if (iteration == MAX_ITERATION)
  {
    sendNoWork();
  }
  else
  {
    generateRandomNetAndSend();
  }

  break;
}
case MSG_WORK_SENT :
{
  state = ACTIVE;
  flag = 0;

  MPI_Recv(buffer, LENGTH, MPI_PACKED, MPI_ANY_SOURCE,
  MSG_WORK_SENT, MPI_COMM_WORLD, &status);

  doNetworkAndSendResult();

  break;
}
case MSG_RESULT :
{
  flag = 0;

  MPI_Recv(buffer, LENGTH, MPI_PACKED, MPI_ANY_SOURCE, MSG_RESULT,
  MPI_COMM_WORLD, &status);

  receiveResultAndWrite();

  break;
}
case MSG_WORK_NOWORK :
{
  flag = 0;

  MPI_Recv(&from, 1, MPI_INT, MPI_ANY_SOURCE, MSG_WORK_NOWORK,
  MPI_COMM_WORLD, &status);

  break;
}
```

```
        case MSG_TOKEN :
        {
          flag = 0;
          int getTOKEN;

          MPI_Recv(&getTOKEN, 1, MPI_INT, MPI_ANY_SOURCE, MSG_TOKEN,
          MPI_COMM_WORLD, &status);
          sendToken(getTOKEN);

          break;
        }
        case MSG_FINISH :
        {
          flag = 0;
          read_enable = 1;

          writeResultsAndFinalize();

          MPI_Finalize();
          exit (0);

          break;
        }
      }
    }
  }
  if(state == ACTIVE)
  {
    state = IDLE;
    TOKEN = 0;

    if(myid != 0 && numprocs > 1)
    {
      requestJob();
    }
    if(myid == 0 && iteration == MAX_ITERATION)
    {
      sendTokenFirstTime();
    }
  }
}
```

Základem je nekonečný while cyklus, který je ukončen při obdržení tagu *MSG_FINISH*. V cyklu se vždy po určitém počtu průchodů , určeném proměnnou *CHECK_MSG_AMOUNT*, testuje funkcí **MPI_Iprobe()**, jestli proces neobdržel nějakou řídící zprávu.

Pokud proces obdržel zprávu, provede na základě tagu zprávy příslušnou operaci. Možné

tagy a operace jsou následující.

- **MSG_WORK_REQUEST** - Pokud není proveden maximální počet iterací, zašle řídící proces pomocí funkce **generateRandomNetAndSend()** 3.2.1 náhodnou síť a příkaz, který se nad ní provede. V jiném případě zašle informaci pomocí funkce **sendNoWork()** 3.2.2.
- **MSG_WORK_SENT** - Podřízený proces obdrží od řídícího obvod a provede nad ním příkaz pomocí funkce **doNetworkAndSendResult()** 3.2.3.
- **MSG_RESULT** - Řídící proces přijme výsledky a zpracuje pomocí funkce **receiveResultAndWrite()** 3.2.4.
- **MSG_WORK_NOWORK** - Obdrží podřízený proces, pokud není co vykonávat, poté již jen čeká na ukončovací token.
- **MSG_TOKEN** - Proces zpracuje obdržený token a zašle dál pomocí funkce **sendToken()** 3.2.5
- **MSG_FINISH** - Proces obdrží ukončovací tag a pomocí funkce **writeResultsAndFinalize()** 3.2.6 provede ukončování a ukončí MPI.

Na konci cyklu se pak procesy nastaví na stav *IDLE* a nastaví token na W. Proces s rankem různým od 0 pomocí funkce **requestJob()** zašle žádost o práci. Řídící proces pomocí funkce **sendTokenFirstTime()** zašle poprvé token W, pokud již byly vykonány všechny iterace.

V následujících sekcích pak jsou stručně popsány jednotlivé funkce, které zajišťují vykonávání aplikace:

3.2.1 generateRandomNetAndSend()

Řídící proces vybere příkaz, který byl zatím nejméně vykonáván. Příkaz se spolu s obvodem, který se náhodně vybere ze složky obvodů, zabalí do zprávy *MPI*. Ta se zašle pomocí funkce *MPI* **MPI_Isend()** žádajícímu procesu.

Zároveň si řídící proces zvýší proměnnou o počtu provedených iterací a informaci o tom, kolikrát byl daný příkaz vykonán. Tím se zajistí jak rovnoměrné vykonávání příkazů, tak omezení počtu iterací.

V této částí programu se implementuje varianta programu s tzv. elitismem. Pokud je program **MPI_abc** spuštěn s přepínačem *-e*, dojde k rozhodnutí, zda se zašle náhodný výstupní obvod, nebo prozatím nejlepší dosažený.

Rozhodnutí je určeno na základě generování náhodných čísel. Při startu programu je určeno náhodné číslo, které určuje jestli se bude propagovat nejlepší dosažené řešení. V této části programu je pak vygenerováno náhodné číslo, a pokud se rovná číslu vygenerovanému na začátku programu, je zasláno místo náhodného obvodu nejlepší dosažené řešení.

Rozmezí generování náhodných čísel je určeno počtem příkazů *ABC*, které program využívá.

3.2.2 sendNoWork()

Tato funkce zasílá pomocí funkce *MPI* **MPI_Isend()** procesu, který žádal o práci, informaci, že již není co zpracovávat.

3.2.3 doNetworkAndSendResult()

Podřízený proces přijme od řídícího zaslané údaje pro vykonávání. Poté vykoná obdržený příkaz na přijatém obvodu, který byl náhodně vybrán ze složky obvodů. Po provedení zpracuje výstup z nástroje **ABC**, a to předem připraveným parserem 3.6.

Nakonec všechny nutné záznamy, jako jsou např. informace o obvodu, název výstupního obvodu, provedený příkaz, apod. zašle řídícímu procesu. Ten tyto údaje shromažďuje pro pozdější zpracování.

3.2.4 receiveResultAndWrite()

Řídící proces přijme výsledky od podřízeného procesu. Tyto výsledky uloží do struktury *array_net*. Díky této struktuře se provede zpětná rekonstrukce sekvence příkazů, pomocí kterých byly tyto výsledky dosaženy.

Poté řídící proces všechny tyto údaje uloží do souboru obsahujícího záznamy o provádění. Tento soubor se pak používá při vyhodnocení výsledků. V neposlední řadě si pamatujeme index prozatím nejlepšího výsledku v datové struktuře. Pokud je právě dosažený výsledek prozatím nejlepším možným, updatujeme tuto hodnotu.

3.2.5 sendToken()

Tato funkce se využívá pro zasílání tzv. pešků (token), které se vyměňují při ukončování programu. U paralelního programu končí výpočet až poté, co všechny procesy dokončily svou práci.

Pro identifikaci stavu, že všem procesům došla práce, se používá metoda *ADUV*. U této metody může být každý proces obarven dvěma barvami: B (Black) nebo W (White). Na počátku mají procesy barvu W. Samotný algoritmus ukončení funguje takto:

- Jakmile se první proces P_0 dostane do stavu *IDLE*, nastaví svou barvu na W a pošle peška barvy W procesu P_1.
- Jestliže proces P_i pošle práci procesu P_j, kde i > j, pak P_i nastaví svou barvu na B.
- Jestliže P_i obdrží peška, pak má-li P_i barvu B, nastaví barvu peška na B. Jakmile se P_i stane *IDLE*, pošle peška po kružnici procesoru P_{i+1} a nastaví svoji barvu na W.
- Pokud P_0 obdrží zpět peška barvy W, můžeme výpočet ukončit. Pokud má pešek barvu B, P_0 zahájí nové kolo vysláním barvy W procesoru P_1.
- Ukončení je detekováno, jakmile P_0 obdrží zpět peška v barvě W.

Samotná funkce **sendToken()** pak má následující strukturu:

- Pokud se jedná o proces s rankem jiným než 0 a je ve stavu *IDLE*, a pokud je jeho token W přepošle dál token, který obdržel. Pokud je jeho token B, přepošle dál token B, a nastaví si svůj token na W.
- Pokud se jedná o proces s rankem jiným než 0 a je ve stavu *ACTIVE*, přepošle dál token B.
- Pokud se jedná o proces s rankem 0, je ve stavu *IDLE* a obdrží token W, nastaví si sám tag na *MSG_FINISH* a všem ostatním pošle ukončovací tag *MSG_FINISH*.
- Pokud se jedná o proces s rankem 0, je ve stavu *IDLE* a obdrží token B, začne znovu se zasílání tokenu a zašle procesu s rankem 1 token W.

3.2.6 writeResultsAndFinalize()

Při ukončování programu je nutno zapsat výsledná data a ukončit aplikaci. U řídícího procesu musíme zjistit nejlepší dosažený výsledek, který se zapíše do souboru i s údaji o sekvenci příkazů, kterými se k němu došlo. U podřízených procesů je pak nutno ukončit instanci nástroje *ABC*.

Při provádění se kvůli vykonávání nad různými daty ukládají veškeré výstupní obvody. Z tohoto důvodu je při větším počtu iterací a větších vstupních obvodech velká paměťová náročnost. Jelikož bylo při testování spouštění programu opakováno nad různými obvody, bylo nutno vytvořit funkci, která po vykonání smaže veškeré dočasné soubory.

Při ukončování programu se volá funkce **deleteFilesFromTempDir()**, která vymaže veškeré soubory obsažené ve složce.

3.3 Paralelní spouštění skriptů ABC

Tato verze aplikace, která je popsána v kapitole 2.2.2 je obměněnou variantou implementovanou v předchozí kapitole. Její spuštění je podmíněno zadáním přepínače *-n* v programu **MPI_abc**. Rozdíly v implementaci jsou pak jiné vstupní parametry, rozdílná inicializace a různé obměny ve volaných funkcích.

Z hlediska inicializace došlo ke změně v tom smyslu, že se načítání příkazů přesunulo na podřízené procesy. To bylo učiněno z toho důvodu, že každý podproces vykonává stejné příkazy, a odpadá tak nutnost zasílání příkazů pomocí *MPI*.

Obměněné funkce se pak volají na stejných místech jako původní a jsou označeny příponou NRP (Not Real Parallelism). Pokud některá z funkcí nemá ekvivalent s příponou NRP, nijak neliší od původní verze, která se volá. Jejich seznam a funkčnost pak vypadá následovně.

- **generateRandomNetAndSendNRP()** - ze zadané složky obvodů vybere jeden, který zašle podřízenému procesu k vykonávání
- **doNetworkAndSendResultNRP()** - pomocí funkce **runABC()** provede nad obdrženým obvodem cyklicky volání jednotlivých příkazů, odpovídající počtu iterací, a odešle výsledky řídícímu procesu

- **receiveResultAndWriteNRP()** - přijme od podřízeného procesu výsledky provedení příkazu a zapíše je do výstupního souboru
- **writeResultsAndFinalizeNRP()** - pouze ukončení *ABC*

3.4 Úprava nástroje ABC

Aby bylo možno komunikovat s nástrojem *ABC*, jak bylo popsáno v kapitole 2.1.3, musíme udělat následující změny.

1. Vytvořit proměnnou pro výstupní soubor v souboru *main.h* v sekci *Global variable*.

```
////////////////////////////////////////////////////////////////////////
///                       GLOBAL VARIABLES                           ///
////////////////////////////////////////////////////////////////////////

FILE *forwardSTDOUT;
```

2. Vytvořit deklarace externích funkcí pro otevírání a zavírání výstupního souboru v souboru *main.h* v sekci *Function definition*.

```
////////////////////////////////////////////////////////////////////////
///                     FUNCTION DEFINITIONS                         ///
////////////////////////////////////////////////////////////////////////

extern FILE *              File_Open(char[]);
extern void                File_Close();
```

3. Vytvořit definici funkcí v souboru *mainLib.c* v sekci *Function definition*.

```
////////////////////////////////////////////////////////////////////////
///                     FUNCTION DEFINITIONS                         ///
////////////////////////////////////////////////////////////////////////

/**Function*************************************************************

  Synopsis    [Open file for forward stdout.]

  Description []

  SideEffects []

  SeeAlso     []

***********************************************************************/
```

```
FILE * File_Open(char path[])
{
  forwardSTDOUT = fopen(path,"a+");
  return forwardSTDOUT;
}

/**Function*************************************************************

  Synopsis    [Close file for forward stdout.]

  Description []

  SideEffects []

  SeeAlso     []

***********************************************************************/
void File_Close()
{
  fclose(forwardSTDOUT);
}
```

4. Upravit funkci **Abc_NtkPrintStats** v souboru *abcPrint.c* následujícím způsobem.

```
void Abc_NtkPrintStats()

  //FILE * pFile = stdout;
  FILE * pFile = forwardSTDOUT;
```

3.5 Komunikace aplikace s ABC

Komunikace aplikace s *ABC* byla zprovozněna přes upravenou statickou knihovnu, jak je popsáno v 2.1.3, a to následujícím způsobem.

```
RESULT doCommand(int iteration)
{
  RESULT result;

  file = File_Open(cesta);

  Cmd_CommandExecute( pAbc, Command )

  result.result_stats = readLine(cesta);
```

```
  File_Close();

  return result;
}

char * readLine(char path[])
{
  File_Close();
  file = File_Open(path);

  while(fgets(line,sizeof(line),file) != NULL);

  file = fopen(path,"w");
  fclose(file);

  File_Close();
  file = File_Open(path);

  return line;
}
```

3.6 Parsovaní vysledků ze souboru

Podle analýzy v kapitole 2.1.4 byl pro tuto aplikaci vytvořen následující parser na zpracování statistik z výstupního souboru.

Použité funkce **append_char()** a **append_char_backwards()** pak slouží ke skládání řetězců ať už pro načítání řetězce do znaku „=“, tak pro zpětnou konstrukci parametrů a hodnot při zpětném procházení.

Funkce **inicializovatPromenne()** zajistí, aby se proměnné po načtení parametrů a hodnot opět nastavily na výchozí hodnoty a bylo možno provést další načítání. Pro přiřazení jednotlivých hodnot jejich parametrů a jejich zapsání do výstupní struktury slouží funkce **priraditParametrum()**.

```
for(unsigned int i = 0; i < strlen(stats); i++)
{
  if(stats[i] != '=')
  {
    temp = append_char(temp, stats[i]);
  }
  else
  {
    for(int j = strlen(temp) - 1; j >= 0 ; j--)
    {
      if(temp[j] != ' ')
```

```
      {
        if(space < 2)
        {
          parameter = append_char_backwards(parameter, temp[j]);
        }
        else if (space >= 2 && isValue)
        {
          value = append_char_backwards(value, temp[j]);
        }
      }
      else
      {
        space++;
      }
    }
    if(isValue)
    {
      priraditParametrum();
    }
    inicializovatPromenne();
  }
  if(i == strlen(stats) - 1)
  {
    for(int j = strlen(temp) - 1; j >= 0 ; j--)
    {
      if(temp[j] != ' ')
      {
        value = append_char_backwards(value, temp[j]);
      }
    }
    priraditParametrum();
  }
}
```

Kapitola 4

Testování

4.1 Průběh testování

Testování probíhalo pomocí programu **MPI_abc**, který byl vytvořen pro tuto práci. Testování se skládalo ze čtyř fází:

- Měření sekvenčního vykonávání viz. 4.1.3
- Měření paralelního vykonávání viz. 4.1.4
- Zpracování naměřených dat viz. 4.2
- Vyhodnocení výsledků viz. 4.3

Pro testováno byly zvoleny dva stroje. Prvním z nich byl výpočetní cluster *STAR*, na kterém byl pro tuto práci vytvořen účet. Druhým strojem, použitým pro testování, byl linuxový stroj *MAGNUM4*. Konfigurace těchto strojů je uvedena v následujících kapitolách.

I přes to, že bylo testováno na výše uvedených strojích, je možno aplikaci používat i na stroji s operačním systémem Windows. Vytvoření statické knihovny *ABC* B.1.1, instalace MPI B.1.2 a kompilace aplikace B.1.3 pod operační systémem Windows jsou uvedeny v instalační a uživatelské příručce.

4.1.1 Testování na stroji STAR

Jedná se o výpočetní svazek, který je založen na "blade"architektuře firmy IBM. Výpočetní svazek se skládá z jednoho centrálního počítače *star.fit.cvut.cz* a celkem 14 výpočetních uzlů.

- node-001, node-003, node-005, node-007, node-008, node-010, node-012, node-014 - uzly obsahují dva dvou-jádrové procesory
- node-002, node-004, node-006, node-009, node-011, node-013 - uzly obsahují dva šesti-jádrové procesory

Podrobnější informace o výpočetním clusteru se nacházejí na stránkách předmětu Paralelní systémy [10] nebo na wiki výpočetního clusteru *STAR* [11].

Na výpočetním clusteru *STAR* je již nainstalována knihovna *MPI*, takže není problém se zprovozněním aplikace. Pro samotné zprovoznění tak stačí zdrojové kódy zkompilovat pomocí následujícího Makefile.

```
CC = mpiCC

LIBS = -L/home/<USERNAME>/lib/ lib/libabc.a

all: <PROGRAM\_NAME>
default: <PROGRAM\_NAME>

<PROGRAM\_NAME>:<PROGRAM\_NAME>.cpp
$(CC) $(CFLAGS) -o <PROGRAM\_NAME> <PROGRAM\_NAME>.cpp $(LIBS)
```

4.1.2 Testování na stroji MAGNUM4

Server *MAGNUM4* je osmi-jádrový linuxový stroj. Server je osazen dvěma procesory Intel Xeon E5430 s frekvencí 2.66GHz s kapacitou paměti RAM 16 GB.

Na tomto stroji není nainstalováno paralelní prostředí *MPI*. Je tedy nutno *MPI* nainstalovat podle návodu v kapitole B.2.2. Kompilace samotné aplikace je tedy náročnější a její postup je popsán v kapitole B.2.3. Pro kompilaci aplikace je nutno zkompilovat statickou knihovnu *ABC* B.2.1.

4.1.3 Měření sekvenčního vykonávání

Měření sekvenčního vykonávání logické syntézy bylo testováno na stroji *MAGNUM4*. Důvodem bylo, že na clusteru *STAR* je omezen, kvůli vytíženosti, běh jednotlivých úloh na maximálně 40 minut. Na tomto stroji mohl být program spuštěn na celou složku obvodů, a bylo tak usnadněno spouštění.

Výsledky tohoto testování budou v následující kapitolách prezentovány jako výsledky skriptů *resyn*. Testování probíhalo pomocí vytvořeného programu **MPI_abc**. Spouštění programu bylo vykonáváno pomocí přepínače **-n** a syntézních skriptů *resyn*. Více o spouštění programu je možno zjistit v kapitole B.3.

Syntézní skripty jsou různé posloupnosti základních syntézních příkazů. Skripty resyn vypadají následovně:

- *resyn* - balance; rewrite; rewrite -z; balance; rewrite -z; balance
- *resyn2* - balance; rewrite; refactor; balance; rewrite; rewrite -z; balance; refactor -z; rewrite -z; balance
- *resyn2a* - balance; rewrite; balance; rewrite; rewrite -z; balance; rewrite -z; balance

Spouštění programu bylo prováděno na obvodech ISCAS'85 [1], ISCAS'89 [2] a IWLS'93 [3]. Výsledky tohoto měření byly zaznamenány do souborů *.csv, které se nacházejí na přiloženém CD ve složce Naměřené výsledky.

Výsledky z tohoto měření by se daly shrnout tak, že všechny obvody jsou náchylné na opakované volání skriptů, ale pouze po omezený počet iterací. Po určitý počet iterací jsou výsledky monotónně klesající, poté již k žádnému zlepšení nedochází.

Toto měření bylo prováděno hlavně proto, aby bylo možno porovnat výsledky měření paralelního provádění s tímto druhem provádění. Jelikož se jednotlivé syntézní skripty skládají z několika základních syntézních kroků, byl omezen počet iterací u tohoto měření.

Omezení bylo uzpůsobeno tak, aby spuštění skriptů odpovídalo počtu iterací paralelního měření. Paralelní měření se provádělo po 10 000 iterací, tudíž například skript *resyn* obsahující 6 základních příkazů se prováděl jen 1666 iterací, což odpovídá šestině celkového počtu.

4.1.4 Měření paralelního vykonávání

Pro měření paralelního vykonávání logické syntézy byl zvolen výpočetní svazek *STAR*. Při měření paralelního vykonávání byly použity základní syntézní příkazy balance, rewrite, refactor, rewrite -z a refactor -z. U měření paralelního vykonávání byly měřeny dvě varianty.

První z nich je randomizované paralelní spouštění logické syntézy, které je v dalších kapitolách označováno jako výsledky varianty *-r*. V této variantě byly spouštěny následující příkazy: balance, rewrite, refactor, rewrite -z, refactor -z.

Druhou variantou je pak randomizované paralelní spouštění logické syntézy se zabudovaným elitismem, které je v dalších kapitolách označováno jako výsledky varianty *-e*. Jediným rozdílem mezi těmito variantami je ten, že se u varianty *-e* náhodně zasílá místo náhodného obvodu nejlepší dosažený obvod. Tím se zaručí, že nejlepší výsledky není možno opomenout v průběhu vykonávání. Pravděpodobnost zaslání nejlepšího dosaženého obvodu se odvíjí od počtu syntézním kroků, předaných při startu aplikace. Výsledná pravděpodobnost je pak 1 : Počet syntézních kroků.

Pro spouštění na tomto výpočetním svazku je nutno jednotlivé spouštění programu provádět přes plánovač.

Je tedy třeba si vytvořit skripty, které určují, jak se program spustí, a předají se plánovači. Aby odpadlo ruční vytváření skriptů, byl vytvořen program, který generuje skripty automaticky viz. C.1. Tento program vytvoří skripty pro všechny zadané obvody a zároveň vytvoří příkazy pro zařazení do plánovače.

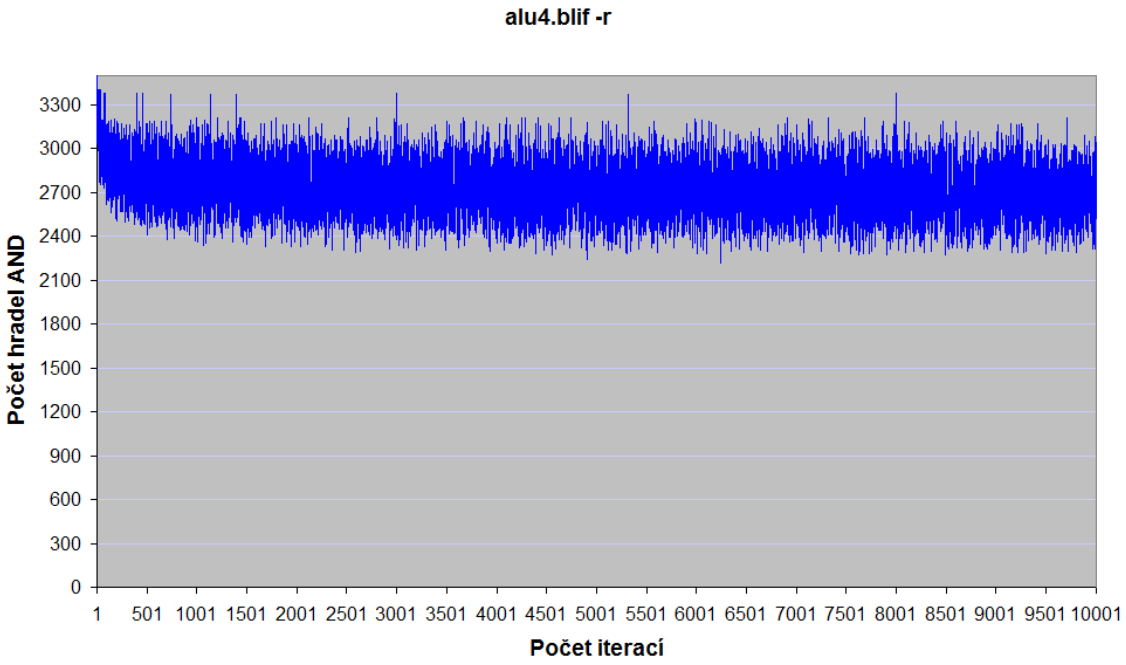
Ve výsledku pak stačilo vytvořené skripty nakopírovat na výpočetní svazek *STAR* do složky uživatele a pomocí vygenerovaných příkazů spustit tyto skripty. Spouštěcí příkaz pak vypadá například takto.

```
qrun.sh 24 long parallel_job000.sh
```

Samotný program **MPI_abc** pak vytvořil soubory results.csv, které obsahují výsledky měření. Tyto soubory se pak předaly dále ke zpracování.

4.2 Zpracování naměřených dat

U jednotlivých obvodů byl měřen počet hradel AND a počet úrovní obvodu. Po naměření výsledků bylo nutno zpracovat samotná data. Jelikož paralelní provádění logické syntézy je založeno na náhodném vybírání sítě, vypadají výsledky jako na obrázku 4.1 u varianty *-r* (zobrazeny výsledky pro obvod alu4.blif), na obrázku 4.2 u varianty *-e* (zobrazeny výsledky pro obvod alu4.blif). Grafy znázorňují, jak se v průběhu iterací měnil, při vykonávání různých syntézních kroků, počet hradel AND obvodu. Na ose x je uveden počet iterací provádění. Na ose y je uvedeno kolik hradel AND obvod po provedení syntézního kroku obsahoval.



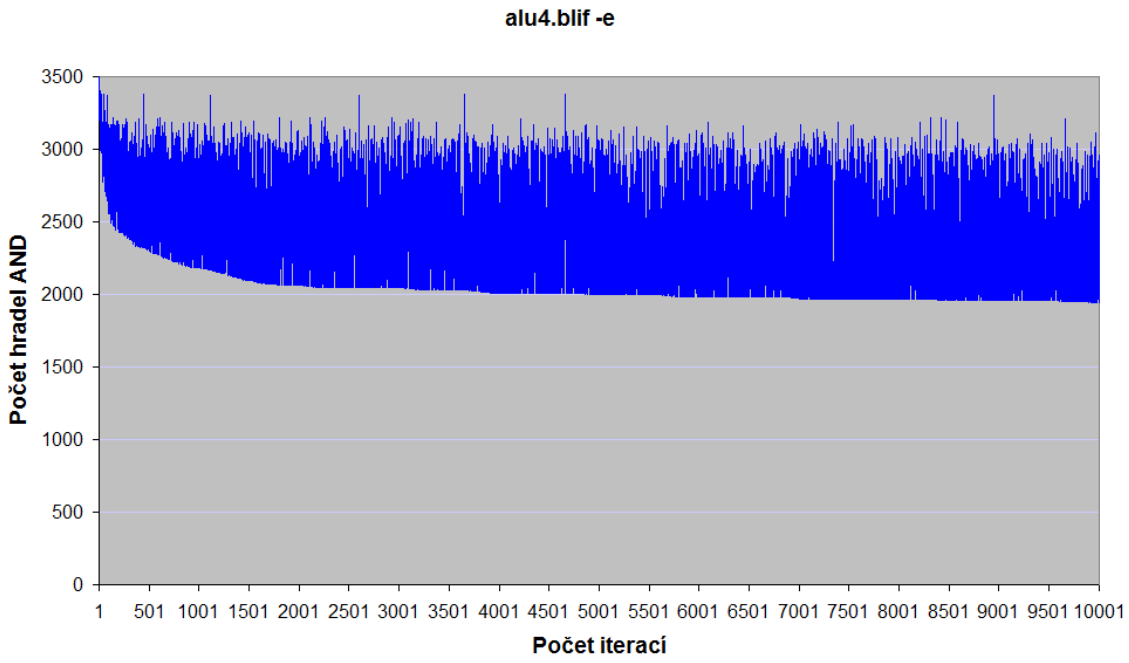
Obrázek 4.1: Paralelní vykonávání obvodu alu4.blif při přepínači -r

Ať se jedná o jakoukoliv iteraci, může proces obdržet náhodně obvod, který má špatný výsledek, a nepodaří se mu ho výrazně zlepšit. Nicméně i přesto je vidět, že se výsledky průběžně zlepšují. U varianty *-e* je pak vidět, že propagování nejlepší sítě opravdu funguje a zlepšování je vidět patrněji.

4.3 Vyhodnocení výsledků testování

4.3.1 Zpracování výsledků

Pro zpracování dat a porovnávání se sekvenčním měřením jsou výše uvedené grafy nevhodné a je nutno je upravit. Jelikož nás zajímá zlepšování výsledků obvodu v průběhu iterací, vytvořily se dolní konvexní obálky těchto grafů.



Obrázek 4.2: Paralelní vykonávání obvodu alu4.blif při přepínači -e

Dále by bylo přínosem vytvořit ze všech dat naměřených pro jeden obvod graf, který porovná všechny varianty měření mezi sebou.

Posledním význačným přínosem pro zpracování dat by bylo vytvořit tabulku, ve které by byly poznamenány statistické údaje pro každý obvod. V této tabulce by se vyskytovaly nejlepší dosažené výsledky, porovnání výsledků jednotlivých variant a sekvence příkazů, pomocí kterých se k těmto datům dospělo.

4.3.2 Program na zpracování výsledků

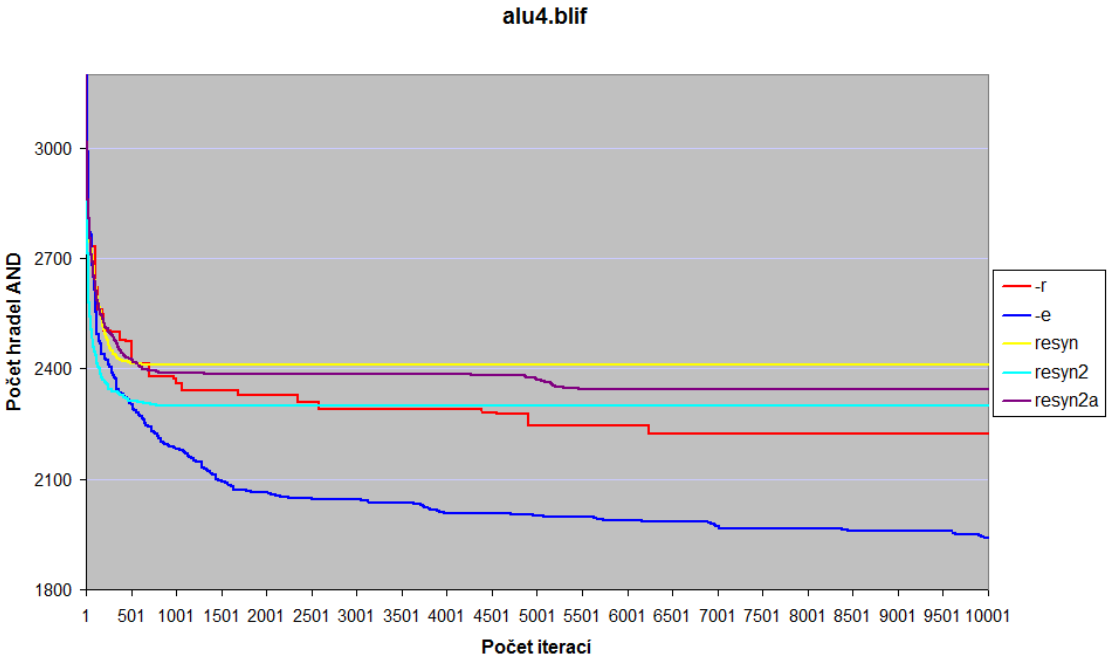
Jelikož by bylo velice obtížné zpracovávat data tímto způsobem ručně, byl vytvořen program C.2, který to zajišťuje. Tento program obdrží veškeré naměřené výsledky a zpracuje je. U sítí vytvořených variantami *-r* a *-e* vytvoří dolní konvexní obálku.

U dat *resyn* zase rozšíří výsledky tak, aby odpovídaly počtu iterací. To znamená, že například pro skript *resyn* zkopíruje výsledek první iterace do prvních šesti iterací variant *-r* a *-e*. Důvod je takový, že varianty *-r* a *-e* dosáhly výsledku v šesté iteraci provedením šesti základních příkazů logické syntézy, kdežto skript *resyn* již v první iteraci, neboť sám obsahuje šest příkazů.

Po provedení všech těchto úprav je pak na výstupu programu pro každý obvod jeden soubor *.xls. Ten obsahuje jak počet logických hradel AND pro každou variantu, tak graf, který zobrazuje porovnání těchto variant.

Výsledný graf pak vypadá například takto 4.3. Z tohoto grafu vyplývá, že variata *-e* dosahuje velmi dobrých výsledků. Z hlediska omezeného prostoru této práce zde nebudou

uvedeny jednotlivé grafy pro všechny zpracované obvody, nicméně je možno je nalézt na přiloženém CD ve složce Zpracované výsledky.



Obrázek 4.3: Porovnání jednotlivých variant logické syntézy na obvodu alu4.blif

4.3.3 Výběr údajů pro vyhodnocení

Po zpracování výsledků pro jednotlivé sítě nám program na zpracování na výstupu poskytl tabulku obsahující podrobné informace o obvodech a měřených variantách. V této tabulce se nacházejí následující údaje:

- ORIGAND - určující původní počet hradel AND obsažených v obvodu
- AND- počet hradel AND dosažených po provedení varianty *-r*
- ANDRESYN - počet hradel AND dosažených po provedení sekvenčního volání jednotlivých skriptů *resyn*
- ANDELIT- počet hradel AND dosažených po provedení varianty *-e*
- DIFF - jednotlivé rozdíly variant v počtu hradel AND ať už vůči původnímu stavu, nebo jednotlivým skriptům, např.

```
DIFF = ANDRESYN - AND
```

- %DIFF - jednotlivé rozdíly variant v počtu hradel AND vyjádřených v procentech ať už vůči původnímu stavu, nebo jednotlivým skriptům, např.

  ```
  %DIFF = ((ANDRESYN - AND) / ANDRESYN) * 100
  ```

- SEKV - sekvence příkazů dosažených po provedení varianty *-r*
- SEKVELIT - sekvence příkazů dosažených po provedení varianty *-e*

Ze všech těchto údajů byly pro zobrazení do následujících tabulek vybrány hodnoty ORIGAND a %DIFF. Tyto hodnoty nejvíce znázorňují účinnost jednotlivých variant a je možno z nich dopočítat ostatní uvedené hodnoty.

Výsledky tak byly rozděleny do dvou sekcí. Zvlášť byly uvedeny výsledky pro variantu -r a pro variantu -e.

4.3.4 Vyhodnocení paralelismu logické syntézy

U paralelizace logické syntézy je v níže uvedených grafech vidět, že ve většině případů je výhodnější než skripty *resyn*.

U paralelizace logické syntézy s přepínačem *-r* jsou výsledky horší jen u 5,18 % testovaných obvodů než při sekvenčním spouštění skriptu *resyn*. U paralelizace dochází k průměrnému zlepšení 9,22 % na obvod oproti skriptu *resyn*. Oproti skriptu *resyn2a* je zhoršení u 4,78 % obvodů a průměrné zlepšení je 9,24 % na obvod.

U obou skriptů se pak zhoršení týká stejných obvodů, a je tedy možno podotknout, že tyto obvody nejsou na tento způsob zpracování náchylné. Pokud se podíváme na grafy jednotlivých obvodů, např. na obrázku 4.3, je z nich patrno následující. Zatímco se při provádění sekvenčního volání skriptů *resyn* rychle ustálí na výsledných hodnotách, tak varianta *-r* průběžně stále nachází zlepšení.

Pokud by tedy tento průběh grafu pokračoval při zvýšení počtu iterací, je možno, že bychom dosáhli zlepšení u všech testovaných obvodů. Tyto výsledky by byly samozřejmě vykoupeny větší časovou náročností, avšak pokud by se aplikovaly vždy na jeden požadovaný obvod, nebyla by časová náročnost tak vysoká a byla by vykoupena podstatným zlepšením výsledku.

Jedná-li se o porovnání se skriptem *resyn2*, je situace poněkud horší. Zde je vidět, že tento skript je po zásluze prozatím nejpoužívanějším skriptem a naše varianta *-r* dosahuje horších výsledků u 25,5 % obvodů a průměrné zlepšení je pouze 2,7 % na obvod oproti skriptu *resyn2*.

I přesto dochází touto variantou u většiny obvodů ke zlepšení, a tato varianta se pro některé obvody rozhodně vyplatí.

Výsledky pro variantu *-r* jsou uvedeny v tabulkách 4.3, 4.4, 4.5, 4.6, 4.7 a 4.8.

Popis sloupců tabulek je následující:

- Název - název testovaného obvodu
- AND - celkový počet hradel AND testovaného obvodu

- diff(all) - rozdíl (vyjádřený v procentech) výsledného počtu hradel AND varianty -r oproti původnímu obvodu

  ```
  diff(all) = ((AND - AND_varianta_-r) / AND) * 100
  ```

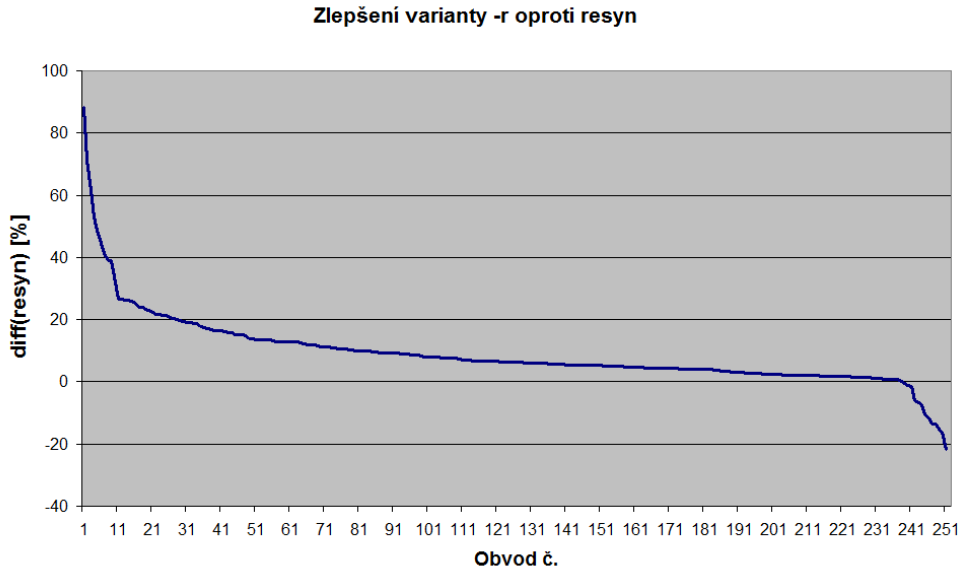
- diff(resyn)- rozdíl (vyjádřený v procentech) výsledného počtu hradel AND varianty -r oproti skriptu *resyn* (obdobně u skriptů *resyn2* a *resyn2a*)

  ```
  diff(resyn) = ((AND_resyn - AND_varianta_-r) / AND_resyn) * 100
  ```

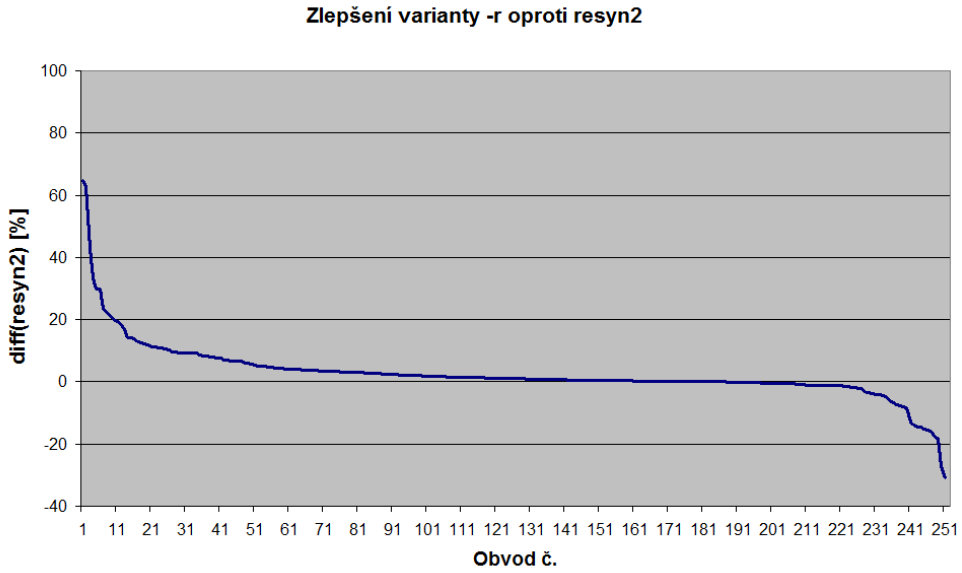
Grafické vyjádření výsledků pro oproti jednotlivým skriptů je zobrazeno na obrázcích 4.4, 4.5 a 4.6. V grafech jsou na ose y uvedeny jednotlivé rozdíly (diff) v procentech. Ty jsou pro zpřehlednění seřazeny podle velikosti. Osa x pak zastupuje jednotlivé testované obvody. Kladné hodnoty rozdílů (diff) znamenají zlepšení, záporné zhoršení.

Abychom mohli porovnat jestli má velikost obvodu vliv na výsledné zlepšení, byly vytvořeny grafy na obrázcích 4.7, 4.8 a 4.9. Pro zobrazení byly výsledky seřazeny podle velikost obvodu. Na ose x je zobrazena velikost obvodu, na ose y je uveden rozdíl (diff) v procentech.

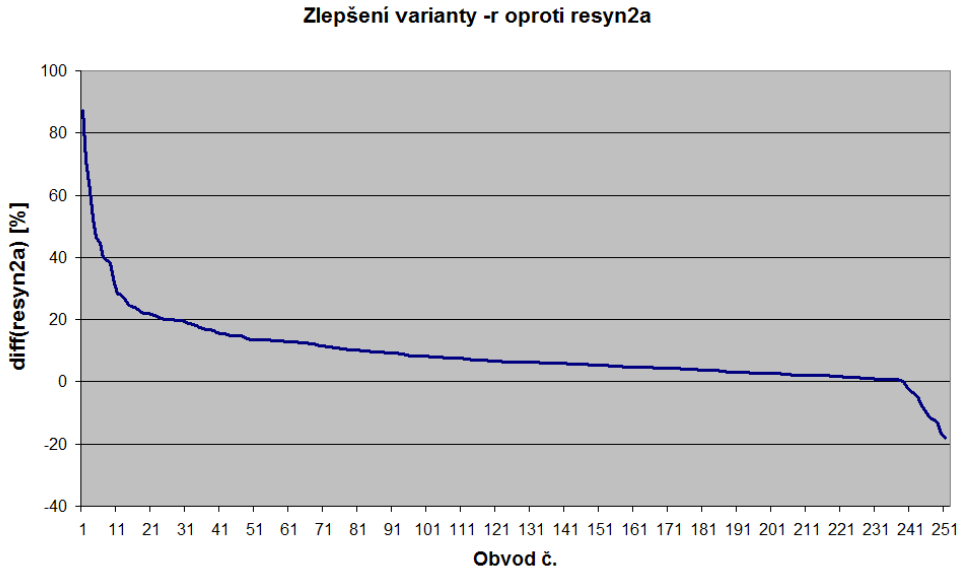
Z těchto grafů můžeme odvodit, že paralizace dosahuje lepších výsledků zejména pro menší a středně velké obvody. Velké zlepšení se však může vyskytovat i u velkých obvodů. Na celkové zlepšení má vliv spíše vnitřní uspořádání než jeho velikost.



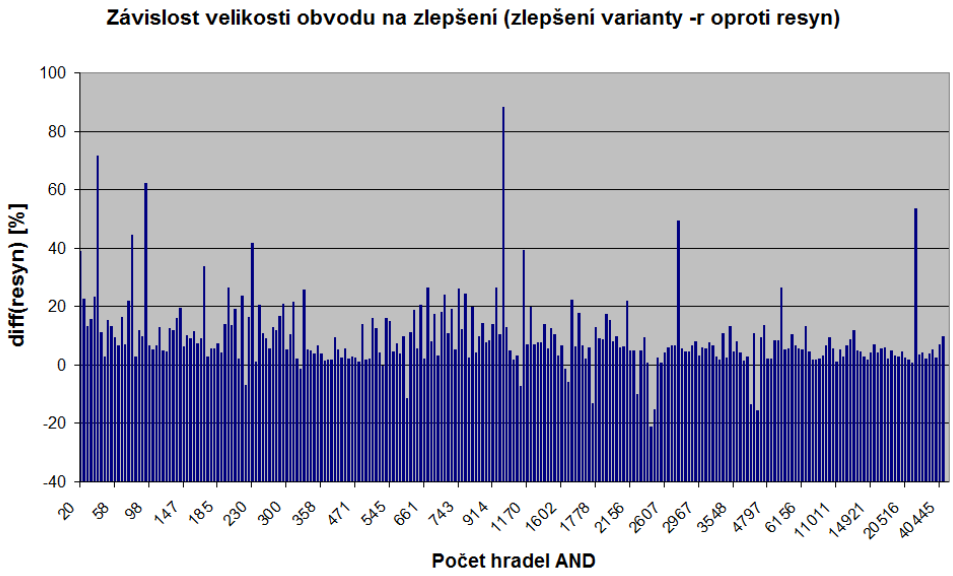
Obrázek 4.4: Zlepšení varianty -r oproti skriptu resyn



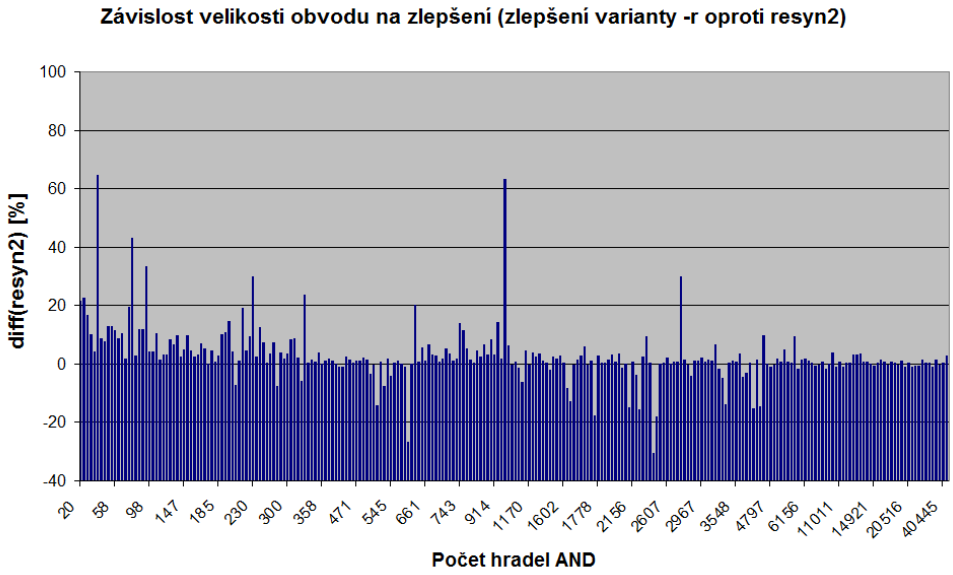
Obrázek 4.5: Zlepšení varianty -r oproti skriptu resyn2



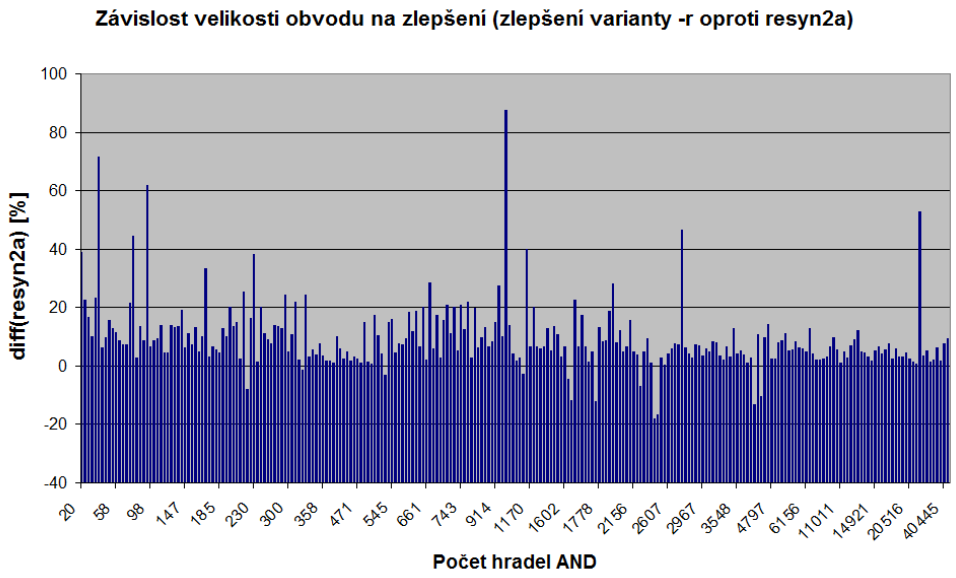
Obrázek 4.6: Zlepšení varianty -r oproti skriptu resyn2a



Obrázek 4.7: Závislost velikosti obvodu na zlepšení (zlepšení varianty -r oproti skriptu resyn)



Obrázek 4.8: Závislost velikosti obvodu na zlepšení (zlepšení varianty -r oproti skriptu resyn2)



Obrázek 4.9: Závislost velikosti obvodu na zlepšení (zlepšení varianty -r oproti skriptu resyn2a)

4.3.5 Vyhodnocení paralelního logické syntézy s elitismem

Pokud se jedná o variantu *-e* se zabudovaným elitismem, je vidět, že dochází k podstatně většímu zlepšení. Je to dáno tím, že se zde náhodně propaguje nejlepší dosažené řešení viz. 3.1. Tím se tato varianta více blíží sekvenčnímu volání skriptů *resyn*, které vždy berou výsledek předchozí operace.

Varianta *resyn* však časem narazí na obvod, který již dále nedokáže zlepšit, kdežto varianta *-e* může v tomto případě obdržet náhodnou předchozí síť, kterou si zpracováním může otevřít další cesty ke zlepšení.

Samotné výsledky pak vypadají následovně. U paralelizace logické syntézy s přepínačem *-e* jsou výsledky horší pouze u 0,8 % obvodů než při sekvenčním spouštění skriptu *resyn*. U paralelizace dochází k průměrnému zlepšení 11,61 % na obvod oproti skriptu *resyn*. Oproti skriptu *resyn2a* je zhoršení u 0,8 % obvodů a průměrné zlepšení je 11,58 % na obvod. Oproti skriptu *resyn2* je zhoršení u 7,17 % obvodů a průměrné zlepšení je 5,37 % na obvod. Oproti tomuto skriptu je pak zhoršení větší než 2 % jen u 2,4 % obvodů.

U této varianty se dají výsledky považovat za úspěch a je možno konstatovat, že tato varianta je podstatně lepší než sekvenční volání skriptů *resyn*. Zlepšení varianty *-e* oproti skriptů *resyn* je možno pozorovat například na grafu 4.3.

Výsledky pro variantu *-e* jsou uvedeny v tabulkách 4.9, 4.10, 4.11, 4.12, 4.13 a 4.14

Popis sloupců tabulek je následující:

- Název - název testovaného obvodu
- AND - celkový počet hradel AND testovaného obvodu

- diff(all) - rozdíl (vyjádřený v procentech) výsledného počtu hradel AND varianty -e oproti původnímu obvodu

  ```
  diff(all) = ((AND - AND\_varianta_-r) / AND) * 100
  ```

- diff(resyn)- rozdíl (vyjádřený v procentech) výsledného počtu hradel AND varianty -e oproti skriptu *resyn* (obdobně u skriptů *resyn2* a *resyn2a*)

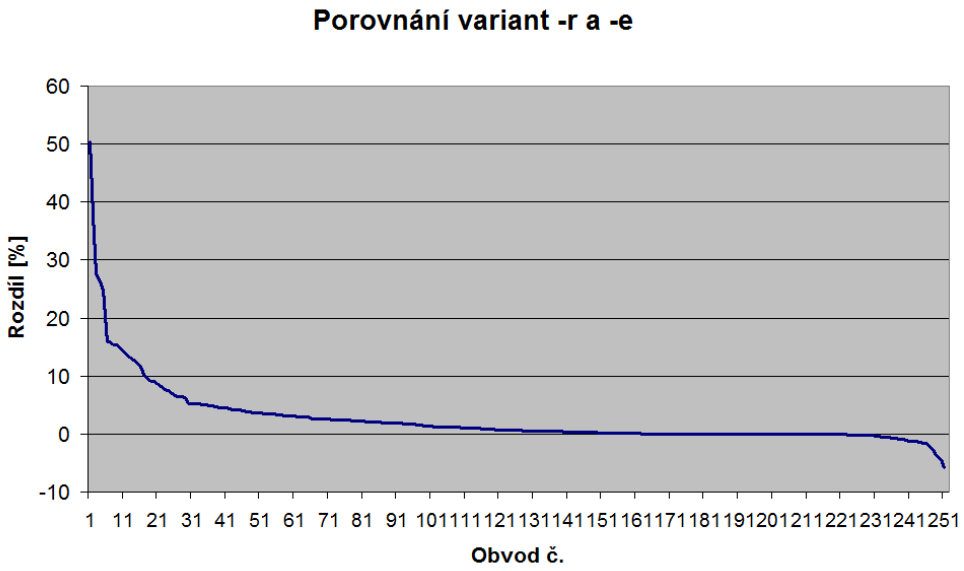
  ```
  diff(resyn) = ((AND_resyn - AND_varianta_-e) / AND_resyn) * 100
  ```

Grafické vyjádření výsledků pro oproti jednotlivým skriptů je zobrazeno na obrázcích 4.11, 4.12 a 4.13. V grafech jsou na ose y uvedeny jednotlivé rozdíly (diff) v procentech. Ty jsou pro zpřehlednění seřazeny podle velikost. Osa x pak zastupuje jednotlivé testované obvody. Kladné hodnoty rozdílů (diff) znamenají zlepšení, záporné zhoršení.

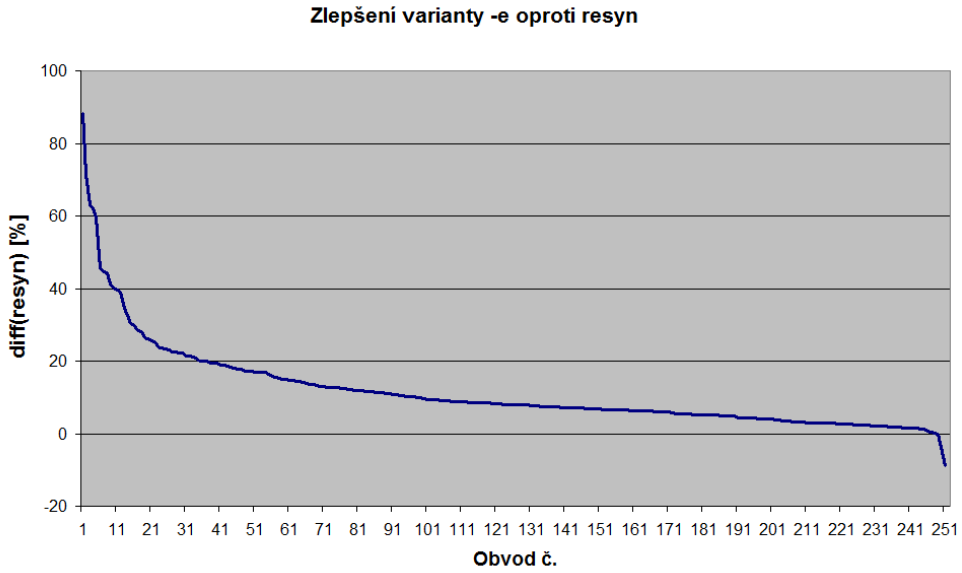
Abychom mohli porovnat jestli má velikost obvodu vliv na výsledné zlepšení, byly vytvořeny grafy na obrázcích 4.14, 4.15 a 4.16. Pro zobrazení byly výsledky seřazeny podle velikost obvodu. Na ose x je zobrazena velikost obvodu, na ose y je uveden rozdíl (diff) v procentech.

Z těchto grafů můžeme odvodit, že paralizace dosahuje lepších výsledků zejména pro menší a středně velké obvody. Velké zlepšení se však může vyskytovat i u velkých obvodů. Na celkové zlepšení má vliv spíše vnitřní uspořádání než jeho velikost.

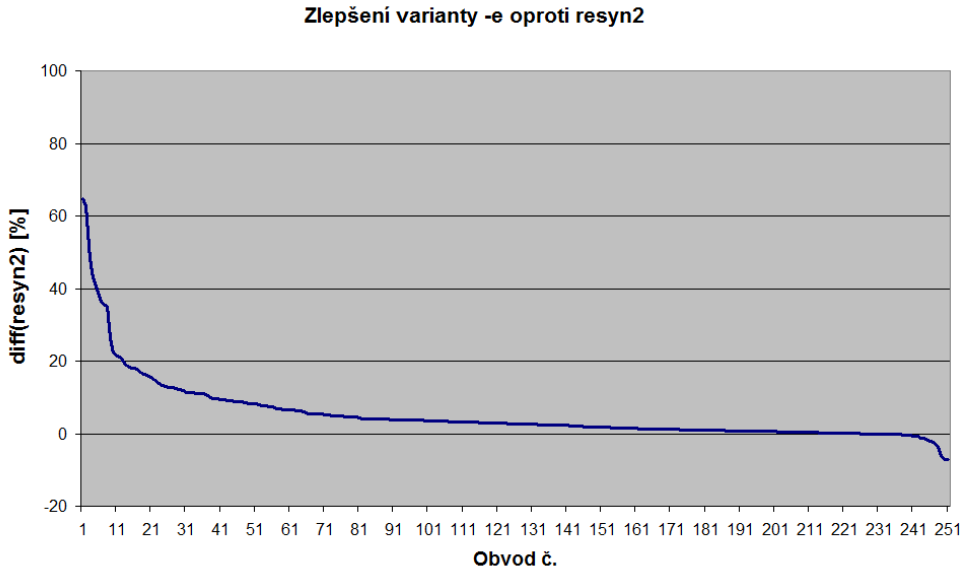
Pokud se jedná o porovnání variant *-r* a *-e* mezi sebou, je toto možno vidět na grafu 4.10, který vyjadřuje, o kolik procent je varianta *-e* lepší než varianta *-r* u jednotlivých obvodů.



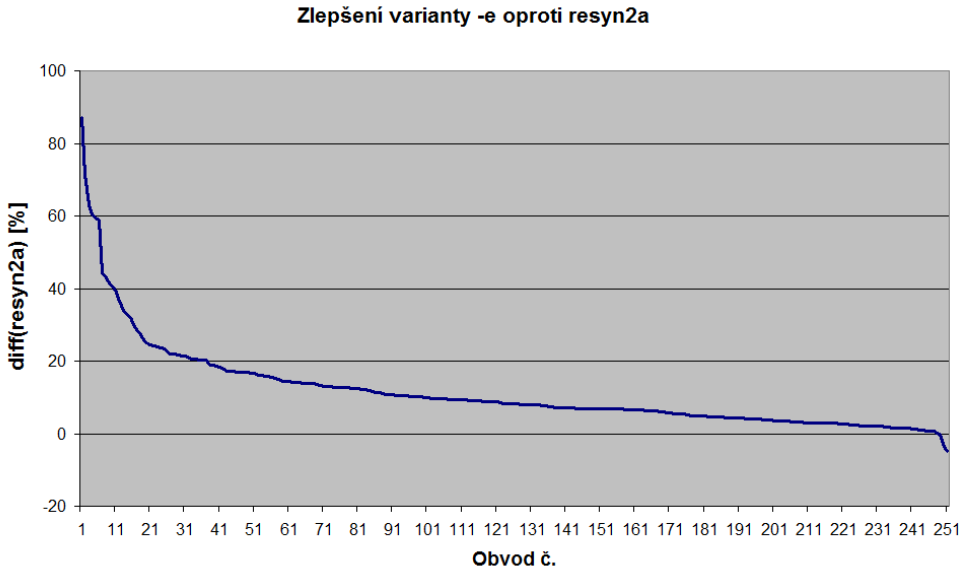
Obrázek 4.10: Porovnání dosažených výsledků variant -r a -e



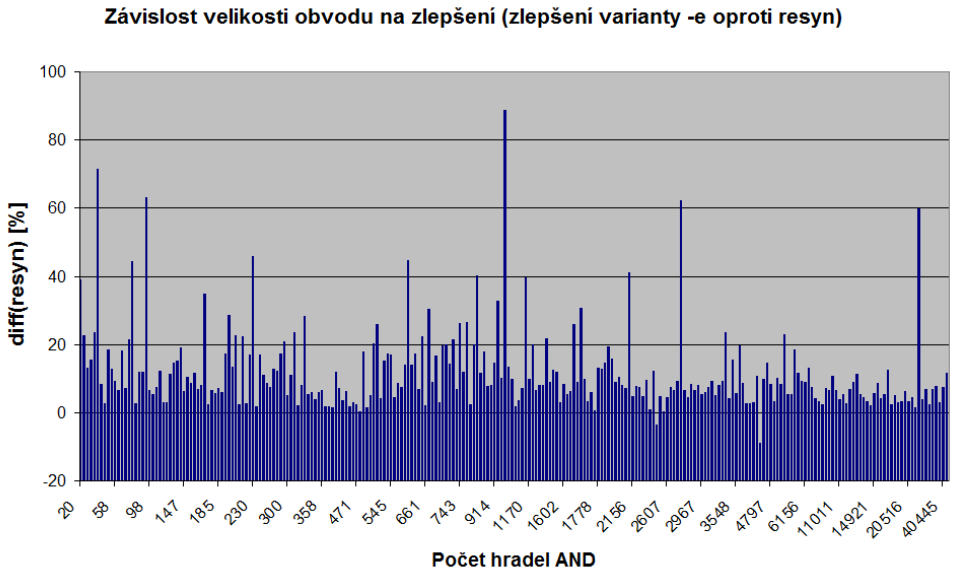
Obrázek 4.11: Zlepšení varianty -e oproti skriptu resyn



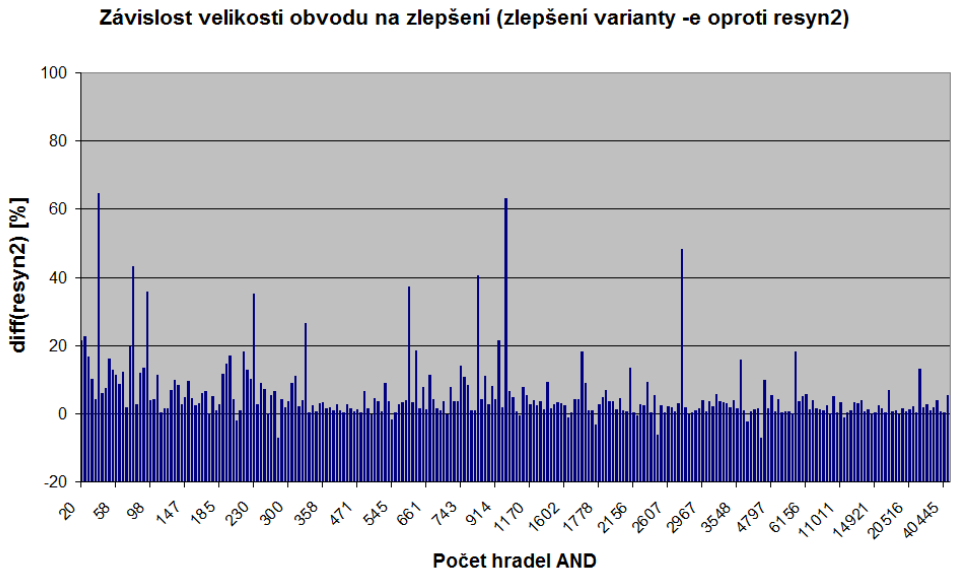
Obrázek 4.12: Zlepšení varianty -e oproti skriptu resyn2



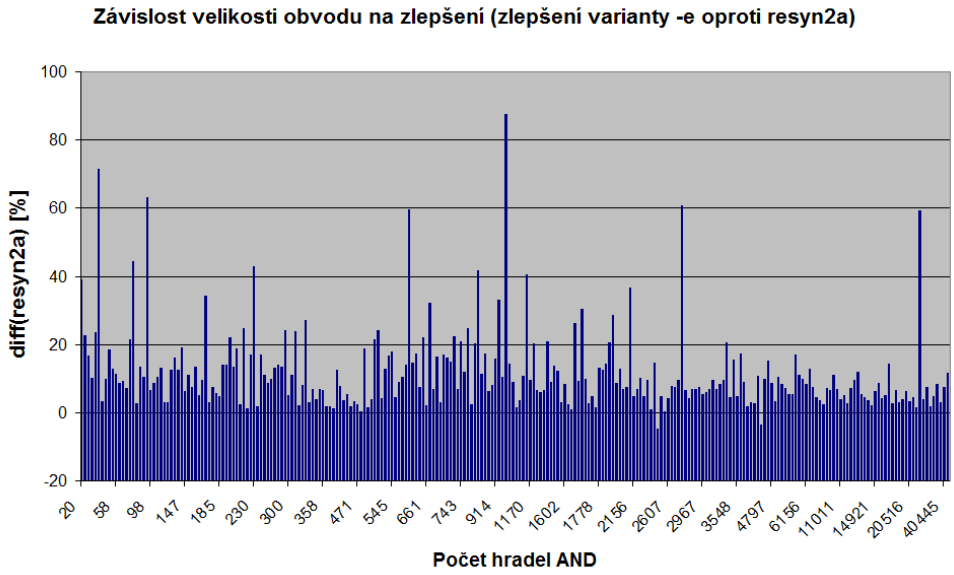
Obrázek 4.13: Zlepšení varianty -e oproti skriptu resyn2a



Obrázek 4.14: Závislost velikosti obvodu na zlepšení (zlepšení varianty -e oproti skriptu resyn)



Obrázek 4.15: Závislost velikosti obvodu na zlepšení (zlepšení varianty -e oproti skriptu resyn2)



Obrázek 4.16: Závislost velikosti obvodu na zlepšení (zlepšení varianty -e oproti skriptu resyn2a)

4.3.6 Hledání optimální sekvence příkazů

V závěru zpracovávání výsledků, byla hledána optimální sekvence příkazů. Pro každý obvod byla nalezena sekvence, která vedla k nejlepšímu výsledku. Motivací je nalezení obecné sekvence příkazů. Ta by při sekvenčním volání předčila skripty *resyn*.

Abychom tuto sekvenci nalezli, musíme znát opakující se sekvence. Pro tento případ by vytvořen program, který zjistí nejdelší společnou sekvenci viz. C.3. Tento program porovná sekvence příkazů systémem každý s každým a poznamená jejich nejdelší společné sekvence. Na výstupu programu je soubor obsahující tyto sekvence i s jejich četností.

V tabulce 4.1 jsou uvedeny nejčastěji se opakující sekvence příkazů. Sekvence jsou vyjádřeny číselně. Jednotlivá čísla odpovídají základním syntézním příkazům (0 - balance, 1 - rewrite, 2 - refactor, 3 - rewrite -z, 4 - refactor -z).

Tabulka 4.1: Seznam nejčastějších sekvencí s jejich četnostmi.

sekvence	četnost
131	150
333	126
3341	121
3343	121
33341	103
434	100
3333	88
3334	86
3342	76
3331	76
33431	75
3303	72
4333	72
1433	67
3311	66
3032	66
3433	65
33432	64
4330	63
4331	62
33433	62
303	61
433	60

Z důvodu, že se obsah jednotlivých sekvencí opakuje, jsem se rozhodl využít je všechny. Pro získání výsledné posloupnosti byl postup následující:

- odstranit posloupnosti, které jsou již obsažené v některé delší posloupnosti, např. 33341 obsahuje 3341, 333 apod.

- rozšířit stávající posloupnosti tak, aby obsahovaly více posloupností, např. 433032 obsahuje 3303, 3032, 4330, 303, 433 apod.
- spojit posloupnosti, které jsme vytvořili 333341433032 a doplnit příkazy tak, aby byly všechny posloupnosti zastoupeny (ty sice nejsou v přesné podobě a příkazy nenasledují hned za sebou, ale jejich posloupnost je dodržena)

Po provedení těchto úprav dostaneme následující novou posloupnost příkazů (skript):

```
33334143303214

rewrite -z; rewrite -z; rewrite -z; rewrite -z; refactor -z; rewrite;
refactor -z; rewrite -z; rewrite -z; balance; rewrite -z; refactor;
rewrite; refactor -z;
```

4.3.7 Vyhodnocení optimální sekvence příkazů

Aby se dalo potvrdit, že je tato sekvence opravdu výhodnější než skripty *resyn*, musíme provést měření. Pro měření byl stejně jako u skriptů *resyn* využit stroj **MAGNUM4** a program **MPI_abc** za využití přepínače *-n*.

Jelikož sekvence obsahuje více základních syntézních příkazů, byl opět omezen počet iterací. Nalezená sekvence obsahuje 14 základních syntézních příkazů. Měření tedy probíhalo pouze po 715 iterací. To odpovídá čtrnáctině z 10 000 iterací.

Poté bylo provedeno porovnání mezi těmito variantami. Byl vytvořen program viz. C.4, který obdrží soubor *results.csv*, který je výstupem programu na zpracování grafů C.2, a naměřené výsledky pro zvolenou sekvenci.

Samotné výsledky pak vypadají následovně. Oproti skriptu *resyn* dosahuje nalezená sekvence zhoršení pouze u 4,76 % obvodů a dosahuje průměrného zlepšení 10,09 % na obvod. Oproti skriptu *resyn2a* je zhoršení u 3,17 % obvodů a průměrné zlepšení je 10,05 % na obvod. Oproti skriptu *resyn2* je zhoršení u 23,8 % obvodů a průměrné zlepšení je 3,50 % na obvod. Oproti tomuto skriptu je zhoršení větší než 2 % pouze u 7,94 % obvodů.

Výsledky pro sekvenci příkazů jsou uvedeny v tabulkách 4.15, 4.16, 4.17, 4.18, 4.19 a 4.20

Popis sloupců tabulek je následující:

- Název - název testovaného obvodu
- AND - celkový počet hradel AND testovaného obvodu
- diff(resyn)- rozdíl (vyjádřený v procentech) výsledného počtu hradel AND varianty -r oproti skriptu *resyn* (obdobně u skriptů *resyn2* a *resyn2a*)

  ```
  diff(resyn) = ((AND_resyn - AND_varianta_-r) / AND_resyn) * 100
  ```

4.3.8 Testování sekvenčního náhodného spouštění syntézních příkazů

Na závěr testování bylo provedeno měření, které zjišťuje, jaký vliv na výsledky má posloupnost příkazů. Abychom mohli testování provést musíme si určit jak budeme postupovat.

Aby se projevila závislost pouze na posloupnosti příkazů byla v této variantě zcela vypuštěna paralizace. Testování pak probíhalo následovně:

- Bylo vybráno několik reprezentujících obvodů, které budeme testovat.
- Tyto obvody byly spouštěny sekvenčně. Při každé iteraci byl vybrán náhodně jeden z předem zadaných příkazů. Náhodně vybraný příkaz byl proveden nad zadaným obvodem.
- Při každé další iteraci se příkazy provádí nad obvodem, který byl vytvořen v předchozí iteraci.
- Jako příkazy byly zvoleny základní syntézní příkazy.
- Tento postup se prováděl iterativně po 1000 iterací.
- Celé testování se opakovaně spouštělo, aby byly odhaleny rozdílné výsledky pro různé sekvence příkazů. Spuštění celého testování bylo v tomto případě prováděno 100 krát.

Pro porovnání výsledků s nějakou referenční hodnotou byly zvoleny výsledky skriptů *resyn2*. Jelikož bylo testování prováděno po 1000 iterací (spuštění 1000 příkazů), byl skript *resyn2*, obsahující 10 základních syntézních příkazů, spuštěny sekvenčně po 100 iterací.

Měření bylo provedeno pomocí programu **MPI_abc** s přepínačem *-n*. Kvůli nutnosti náhodné volby vykonávacího příkazu byla upravena funkce **Run_Abc** následujícím způsobem:

```
for(int i = 0; i < ITERATION_COUNT; i++ ) //po zadany pocet iteraci
{
  //for(int j = 1; j <= commandCount_int; j++){
  //proved nad obovodem vsechny zadane prikazy
  int j = rand() % (commandCount_int);
  j = j + 1;

  stats_temp = doCommandNRP(Command_Array[j - 1], WriteNet);
  parser_stats = parser(stats_temp.result_stats);
  //}
}
```

Tato úprava zajišťuje, že nejsou vykonávány sekvenčně všechny zadané příkazy, ale náhodně se vybere jeden z nich.

Výsledky obdržené z tohoto měření pak byly zpracovány pomocí programu C.5. Výsledky měření jsou pak zpracovány v tabulce 4.2.

Popis sloupců tabulek je následující:

- Název - název testovaného obvodu
- AND - celkový počet hradel AND testovaného obvodu
- resyn2 - počet hradel AND dosažený při sekvenčním spouštění skriptu *resyn2*
- min- minimální počet hradel AND dosažený při sekvenčním náhodném spouštění syntézních příkazů
- max- maximální počet hradel AND dosažený při sekvenčním náhodném spouštění syntézních příkazů
- avg- průměrný počet hradel AND dosažený při sekvenčním náhodném spouštění syntézních příkazů

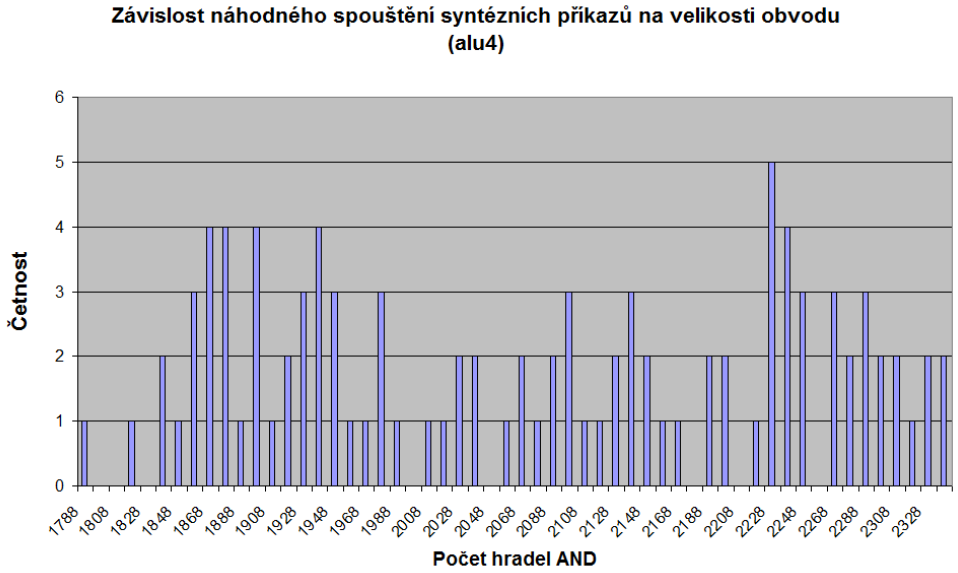
Z výsledků uvedených v tabulce 4.2 vidíme, že posloupnost příkazů spuštěných na obvod a její vnitřní uspořádání má značný vliv na dosažený výsledek. Hlavním zjištěním je že skript *resyn2* dává téměř vždy horší výsledek, než je průmerný výsledek dosažený sekvenčním náhodným spouštěním syntézních příkazů.

Důvodem je, že sekvenční spouštění náhodných příkazů nám zabraní uvaznutí v lokálním minimu. Pokud mohu provést hodně iterací, je lepší vykonávat sekvenční spouštění náhodných příkazů, než spouštět syntézní skripty.

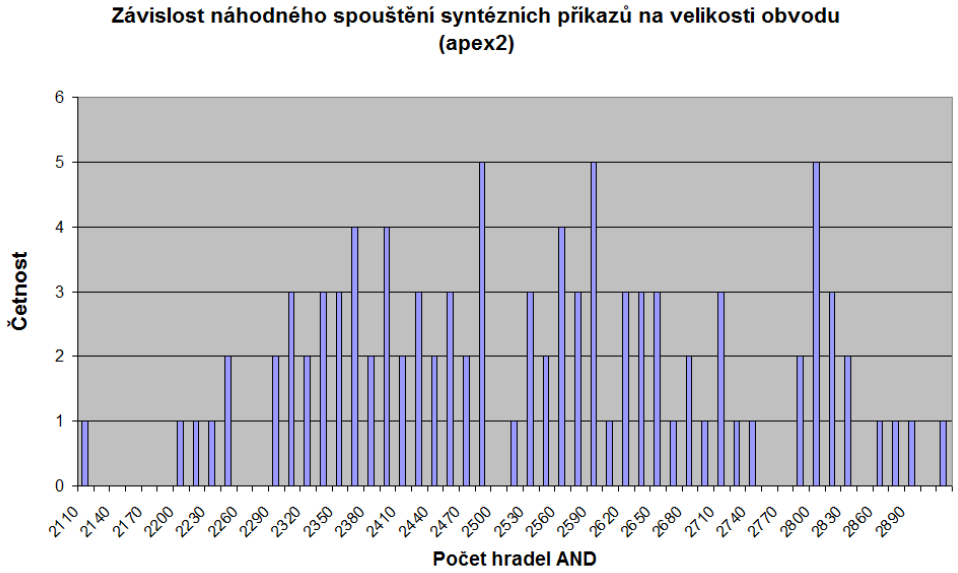
Pro přehlednost v jakém rozmezí se pohybují hodnoty při náhodném volání syntézních příkazů, byly vytvořeny grafy na obrázcích 4.17, 4.18, 4.19 a 4.20. Grafy byly po přehlednost seřazeny podle počtu hradel AND. Na ose x jsou uvedena jednotlivá měření. Osa y obsahuje počet hradel AND dosažených při daném měření.

Tabulka 4.2: Výsledky sekvenčního náhodného spouštění syntézních příkazů

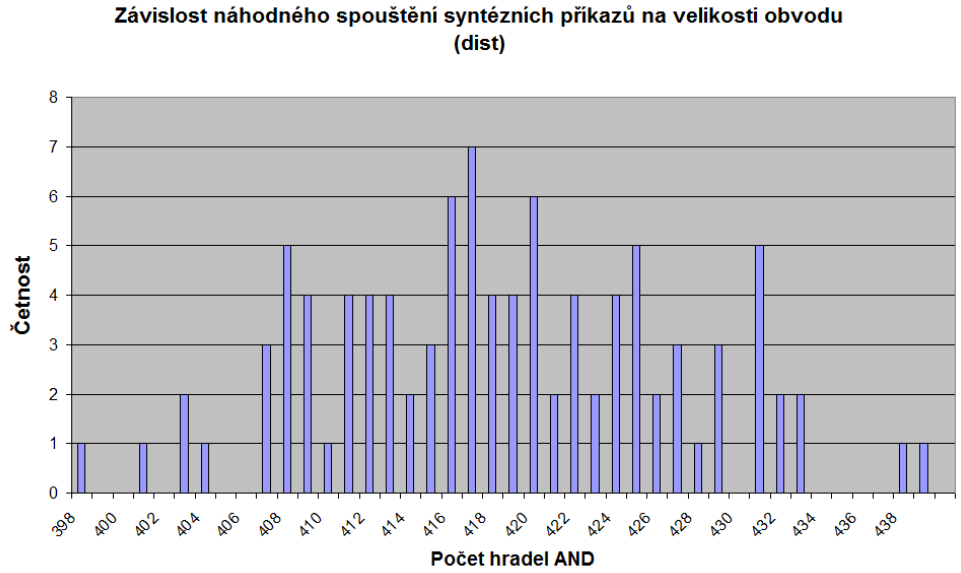
Název	AND	resyn2	min	max	avg
alu4	3603	2301	1788	2335	2071
apex2	4545	2714	2110	2907	2526
apex4	2454	1758	1727	1798	1762
dist	564	434	398	439	419
ex5p	2394	1357	894	1249	1084
frg1	617	243	155	269	199
max512	743	560	513	552	533
misex3	5485	2921	2054	2984	2575
seq	4233	2236	1786	2324	2024
table3	1124	756	616	738	688



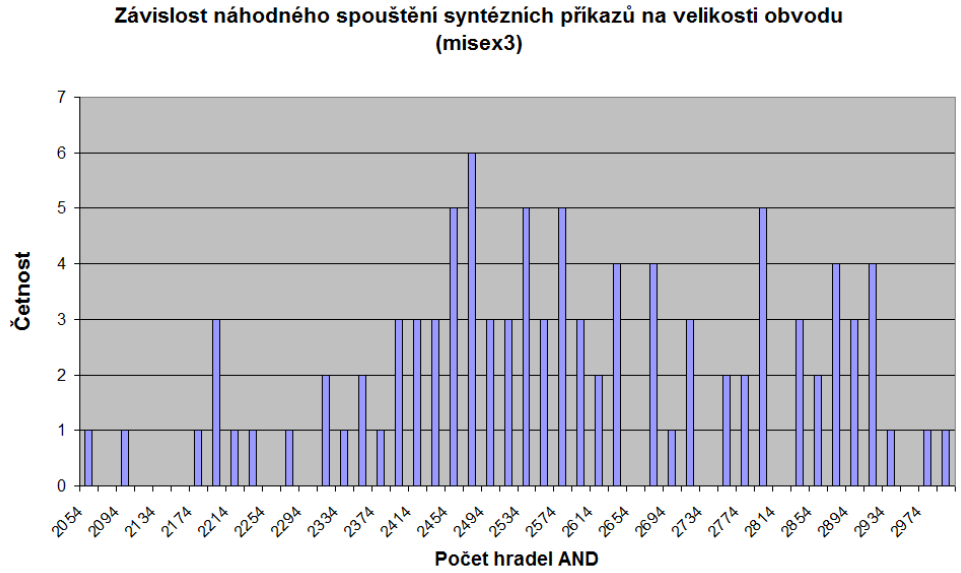
Obrázek 4.17: Závislost náhodného spouštění syntézních příkazů na velikosti obvodu



Obrázek 4.18: Závislost náhodného spouštění syntézních příkazů na velikosti obvodu



Obrázek 4.19: Závislost náhodného spouštění syntézních příkazů na velikosti obvodu



Obrázek 4.20: Závislost náhodného spouštění syntézních příkazů na velikosti obvodu

Tabulka 4.3: Prorovnání varianty -r oproti skriptům resyn v procentech část 1.

Název	AND	diff(all)	diff(resyn)	diff(resyn2)	diff(resyn2a)
5xp1	206	49,51	18,75	-7,22	14,75
9sym	333	14,71	-1,43	-5,97	-1,43
9symml	206	7,77	2,06	0,52	2,06
al2	119	26,05	6,38	10,20	9,28
Altera_ac97_ctrl	14272	27,74	1,72	-0,16	1,76
Altera_aes_core	21217	8,81	1,33	-1,25	1,40
Altera_barrel16	701	53,21	17,38	2,38	16,96
Altera_barrel16a	712	49,16	17,91	1,63	15,42
Altera_barrel32	1778	52,02	12,87	2,40	12,78
Altera_barrel64	4284	49,98	10,75	1,34	10,60
Altera_cfft	13838	37,62	2,43	0,42	2,74
Altera_cord1	11846	32,58	2,59	0,21	2,61
Altera_cord2	15773	34,24	2,06	0,14	2,10
Altera_des_area	4852	10,70	2,17	-0,44	2,26
Altera_ether	10820	24,16	5,52	-1,08	5,66
Altera_fip_cordic_cla	3033	53,18	7,73	0,56	8,03
Altera_fip_cordic_rca	2937	49,81	6,53	0,74	6,94
Altera_fip_risc8	11011	33,65	0,79	0,27	0,73
Altera_i2c	1170	23,50	6,58	-0,9	6,28
Altera_mem	16727	19,11	2,93	-0,38	2,92
Altera_mem_ctrl	15648	49,82	5,92	-0,26	7,74
Altera_mux8_128bit	4609	47,17	9,45	9,41	9,45
Altera_mux8_64bit	2305	47,11	9,37	9,30	9,37
Altera_nut_000	1851	58,83	16,90	0,91	18,24
Altera_nut_001	6237	53,17	12,99	1,02	12,68
Altera_nut_002	1207	51,12	19,73	3,75	20,16
Altera_nut_004	781	54,42	19,46	0,28	19,82
Altera_oc_aes_core	12429	40,56	8,55	2,90	8,99
Altera_oc_aes_core_inv	14978	43,04	6,45	0,20	6,61
Altera_oc_aquarius	34800	43,32	1,89	-0,51	1,82
Altera_oc_ata_ocidec1	2156	33,63	4,66	0,35	4,54
Altera_oc_ata_ocidec2	2607	37,02	4,03	1,62	3,86
Altera_oc_ata_ocidec3	5327	37,56	5,51	0,15	5,48
Altera_oc_ata_v	1252	41,85	7,61	3,06	5,45
Altera_oc_ata_vhd_3	5258	36,80	4,92	0,24	4,87
Altera_oc_cfft_1024x12	15199	44,69	5,25	0,17	4,99
Altera_oc_cordic_p2r	13269	39,77	4,49	0,47	4,54
Altera_oc_cordic_r2p	17047	39,30	2,26	0,61	2,77
Altera_oc_correlator	2956	40,09	7,71	0,95	6,94
Altera_oc_dct_slow	1515	42,11	10,05	1,24	10,42
Altera_oc_des_area_opt	4678	46,96	13,43	0,12	14,06
Altera_oc_des_des3area	8011	47,70	4,05	0,26	3,97

Tabulka 4.4: Prorovnání varianty -r oproti skriptům resyn v procentech část 2.

Název	AND	diff(all)	diff(resyn)	diff(resyn2)	diff(resyn2a)
Altera_oc_des_perf_opt	34325	39,86	5,12	1,06	5,80
Altera_oc_ethernet	14921	43,92	4,18	-1,17	4,66
Altera_oc_fcmp	770	54,16	23,92	5,11	21,90
Altera_oc_fpu	34175	52,97	3,92	-1,25	1,86
Altera_oc_gpio	971	33,16	9,99	1,67	10,11
Altera_oc_hdlc	2998	41,39	5,23	1,29	4,82
Altera_oc_i2c	1383	40,64	7,75	0,85	6,39
Altera_oc_mem_ctrl	20516	34,82	2,23	0,05	2,20
Altera_oc_minirisc	2983	43,78	5,84	0,53	5,73
Altera_oc_miniuart	914	50,11	13,64	2,98	14,93
Altera_oc_mips	25167	41,64	1,72	0,39	1,28
Altera_oc_oc8051	17361	47,23	4,53	-1,19	4,51
Altera_oc_pavr	24582	43,02	4,06	-0,06	4,69
Altera_oc_pci	12189	36,67	6,16	0,12	6,41
Altera_oc_rtc	2064	53,83	6,02	-0,42	6,29
Altera_oc_sdram	1431	42,63	12,38	2,49	13,58
Altera_oc_simple_fm_r	4907	53,60	8,30	0,13	8,26
Altera_oc_vga_lcd	11185	39,79	5,17	-1,25	4,89
Altera_oc_video_c_s_h_d	2931	46,88	4,30	-4,08	2,81
Altera_oc_video_c_s_h_e	3497	49,16	12,84	0,89	12,97
Altera_oc_wb_dma	24170	44,32	3,53	1,51	3,59
Altera_os_blowfish	12676	49,42	4,74	3,40	4,81
Altera_os_sdram16	1997	49,02	9,43	3,51	11,94
Altera_pci_bridge32	22776	21,87	0,68	-0,59	0,61
Altera_pci_conf_c_a_d	87	4,60	2,35	2,35	2,35
Altera_pci_spoci_ctrl	1394	43,47	13,60	-0,13	12,54
Altera_radar12	40445	40,20	6,99	-0,08	7,11
Altera_sasc	775	21,94	2,26	0,82	2,26
Altera_simple_spi	1036	22,78	2,79	-1,27	2,79
Altera_spi	3845	17,32	1,46	-3,48	0,69
Altera_ss_pcm	405	4,20	1,27	0,51	0,77
Altera_steppermotordrive	188	35,64	3,97	9,70	12,32
Altera_systemcaes	12609	28,38	11,29	2,89	11,71
Altera_systemcdes	2967	18,17	3,04	1,54	2,92
Altera_ts_mike_fsm	84	60,71	21,43	19,51	21,43
Altera_tv80	9764	26,37	3,18	-1,86	2,92
Altera_usb_funct	16051	19,62	4,45	0,23	5,94
Altera_usb_phy	468	25,43	2,79	0,29	3,06
Altera_wb_dma	4074	17,84	2,36	0,15	2,70
Altera_xbar_16x16	1232	42,86	6,38	2,22	6,38
alu2	519	29,67	2,14	-3,4	0,82
alu3	78	14,10	6,94	1,47	6,94

Tabulka 4.5: Prorovnání varianty -r oproti skriptům resyn v procentech část 3.

Název	AND	diff(all)	diff(resyn)	diff(resyn2)	diff(resyn2a)
alu4	3603	38,33	7,84	3,43	5,20
am2910	879	23,09	7,52	2,73	6,24
amd	665	51,28	26,03	6,09	28,16
apex1	1622	24,78	-6,09	-12,75	-11,93
apex2	4545	31,64	-16,19	-14,48	-10,53
apex3	1757	23,45	-13,41	-17,78	-12,27
apex4	2454	16,34	-15,21	-18,06	-16,65
apex5	3325	74,59	10,30	-14,04	6,63
apex6	661	9,98	2,14	1,00	2,14
apex7	276	41,30	12,90	6,90	13,37
apla	277	39,71	11,64	-7,74	13,47
b10	655	43,97	20,22	5,41	20,04
b11	123	46,34	4,35	2,94	4,35
b12	978	94,58	88,25	62,68	87,17
b2	1793	33,97	8,71	0,34	8,43
b3	525	45,14	16,03	-0,35	17,48
b4	346	30,35	4,74	1,23	5,49
b7	123	46,34	4,35	2,94	4,35
b9	117	33,33	4,88	3,70	8,24
bbtas	45	48,89	23,33	4,17	23,33
bc0	1626	43,85	21,83	-0,66	22,17
bca	3830	34,39	3,64	-4,58	3,83
bcb	3319	34,89	1,77	-4,8	2,00
bcc	3189	35,31	2,73	-1,98	3,10
bcd	2205	35,46	4,62	-3,79	3,79
beecount	54	38,89	10,81	8,33	5,71
bigkey	5149	42,86	26,28	4,76	10,98
br1	209	50,72	23,13	18,90	25,36
br2	192	59,38	26,42	14,29	19,59
bw	180	23,33	5,48	0,72	5,48
c1355	510	23,53	1,27	1,02	1,02
c1908	457	22,54	5,60	2,21	4,84
c2670	709	22,85	2,67	0,55	2,67
c3540	1033	10,84	1,81	0,65	1,60
c432	209	33,01	-6,87	4,11	-8,53
c499	400	3,25	1,28	0,77	1,28
c5315	1667	20,52	1,92	-0,53	1,19
c6288	2337	19,98	0,64	0,00	0,64
c7552	2003	27,06	5,86	-1,39	4,76
c8	278	61,51	20,74	1,83	24,11
c880	329	6,69	1,60	1,60	1,60
chkn	444	23,42	5,29	-1,19	5,82

Tabulka 4.6: Prorovnání varianty -r oproti skriptům resyn v procentech část 4.

Název	AND	diff(all)	diff(resyn)	diff(resyn2)	diff(resyn2a)
cht	336	55,95	5,13	0,00	2,63
clip	532	45,11	12,05	-14,51	9,88
clma	1155	79,91	39,11	4,53	40,05
clmb	23490	72,54	52,95	-0,99	52,45
cm152a	31	32,26	22,22	22,22	22,22
cmb	54	31,48	2,63	7,50	9,76
comp	125	31,20	11,34	6,52	13,13
cordic	2785	80,14	48,80	29,82	46,62
cps	1607	28,31	-1,32	-8,37	-4,63
cu	55	38,18	12,82	12,82	12,82
dalu	1663	41,07	6,67	5,77	6,49
dc1	42	33,33	15,15	9,68	9,68
dc2	130	39,23	15,96	9,20	13,19
dekoder	54	48,15	15,15	12,50	15,15
des	4797	33,17	1,78	-1,58	2,05
dist	564	23,94	7,34	1,15	7,54
div16	755	29,93	11,98	10,94	12,13
dk14	124	42,74	12,35	7,79	13,41
dk15	70	27,14	16,39	10,53	7,27
dk17	177	32,20	5,51	4,00	6,25
dk27	90	42,22	11,86	11,86	13,33
dk48	259	44,79	5,30	3,38	7,74
dk512	58	32,76	9,30	11,36	11,36
dsip	2523	0,48	0,16	0,16	0,16
duke2	535	28,41	-0,52	-7,89	-3,51
e64	790	27,59	4,03	4,51	6,08
ex1010	3340	23,41	2,25	-0,2	2,59
ex4	471	16,14	2,23	1,00	2,23
ex4p	2108	47,49	21,77	-15,07	15,56
ex5	951	57,41	26,50	14,01	26,90
ex5p	2394	25,90	-21,59	-30,73	-18,19
ex7	119	25,21	12,75	1,11	13,59
example2	321	28,97	10,24	8,06	10,24
exep	867	50,87	13,77	6,58	13,06
exp	439	31,89	9,39	0,00	9,94
exps	1628	27,46	6,12	1,34	6,42
f51m	230	60,00	41,40	29,77	38,26
frg1	617	68,56	18,83	20,16	18,83
frg2	1951	65,61	15,28	3,03	28,24
g125	2250	28,89	4,36	2,08	4,36
g216	6156	43,24	4,98	1,41	4,40
g25	300	28,33	4,87	3,59	4,87

Tabulka 4.7: Prorovnání varianty -r oproti skriptům resyn v procentech část 5.

Název	AND	diff(all)	diff(resyn)	diff(resyn2)	diff(resyn2a)
g36	684	43,71	7,89	3,02	5,87
g625	15000	29,17	4,08	1,36	4,08
gary	606	30,53	10,81	-0,24	11,55
i10	2512	29,78	2,22	-0,23	2,27
i2	230	8,70	0,94	1,87	0,94
i4	245	17,55	10,62	6,91	10,62
i5	335	52,54	25,35	23,56	24,29
i6	402	3,73	1,78	1,78	1,78
i8	3090	69,03	6,18	6,63	7,89
i9	883	43,71	8,13	8,13	8,13
ibm	229	24,45	16,43	9,42	16,43
idxlat02n_m12	2237	28,12	-10,29	-15,85	-6,99
in0	726	42,98	18,98	0,48	19,77
in1	1793	34,02	8,79	0,42	8,51
in2	544	32,72	15,67	1,61	14,88
in3	574	41,29	9,41	-1,2	9,41
in4	545	44,77	14,97	-4,15	15,92
in5	506	38,54	13,85	2,20	14,79
in6	355	32,39	6,61	3,61	7,69
in7	166	25,30	8,82	6,77	10,14
inc	158	39,24	6,80	3,03	4,95
intb	1425	21,96	5,52	-2,21	5,28
ITC_b04	451	13,75	2,02	-1,3	2,02
ITC_b05	792	43,18	9,64	1,96	9,46
ITC_b06	20	45,00	38,89	21,43	38,89
ITC_b07	351	6,55	3,53	0,30	3,53
ITC_b08	153	16,34	8,57	4,48	7,25
ITC_b09	84	46,43	44,44	43,04	44,44
ITC_b10	174	8,05	2,44	0,00	3,03
ITC_b11	621	22,38	5,49	0,21	6,04
ITC_b12	1005	24,58	12,87	6,07	13,77
ITC_b13	257	15,95	8,86	0,46	8,86
ITC_b14	6063	28,24	5,47	1,18	5,94
ITC_b14_1	4901	28,87	7,92	1,58	8,02
ITC_b15	8400	11,01	1,58	-1,16	1,59
ITC_b15_1	8956	22,99	1,60	-0,48	1,96
ITC_b21_1	10325	27,82	9,12	3,46	9,43
jbp	566	34,63	3,65	-0,54	6,80
ldd	90	38,89	9,84	11,29	8,33
m2	328	42,07	21,49	8,65	21,81
mainpla	5552	33,95	6,50	-2,03	5,97
max1024	1021	21,06	4,50	-0,88	3,70

Tabulka 4.8: Prorovnání varianty -r oproti skriptům resyn v procentech část 6.

Název	AND	diff(all)	diff(resyn)	diff(resyn2)	diff(resyn2a)
max512	743	25,84	4,84	1,61	5,00
Mentor_1_01	2707	32,95	6,68	0,77	7,49
Mentor_1_05	2883	32,29	5,61	1,01	5,79
Mentor_1_07	9942	30,32	6,67	0,06	6,63
Mentor_1_08	2704	31,40	5,60	-0,22	5,98
Mentor_1_09	1602	32,58	6,17	0,09	6,17
Mentor_1_11	2723	31,91	6,51	0,32	6,83
misex1	58	22,41	6,25	8,16	8,16
misex3	5485	52,54	9,90	9,43	8,31
mm30a	1576	38,20	2,99	2,89	2,99
mm9a	472	37,08	0,67	0,67	0,67
mult16	560	22,50	4,19	0,00	4,19
newapla1	39	48,72	13,04	16,67	16,67
newcpla1	132	41,67	18,95	2,53	18,95
newxcpla1	166	53,61	33,62	4,94	33,04
pair	1740	28,16	5,59	0,64	4,36
pcont2	2914	35,21	4,26	-0,48	4,02
pope	722	36,43	10,53	3,57	11,22
rd53	95	70,53	61,64	33,33	61,64
root	277	37,18	16,35	3,33	12,56
s1238	532	10,34	4,02	0,63	4,02
s1423	462	5,41	1,58	1,13	1,35
s15850.1	3548	22,77	4,46	0,44	3,76
s298	98	25,51	6,41	3,95	6,41
s38417	9158	11,60	1,84	0,31	1,96
s444	151	34,44	10,00	9,17	10,81
s526	199	41,71	13,43	4,13	13,43
s641	147	18,37	6,25	4,76	6,25
s832	358	30,17	3,47	0,00	3,47
s9234	1962	32,01	7,94	0,22	7,68
seq	4233	38,98	-13,54	-15,52	-13,69
spla	1657	52,81	17,25	2,25	16,90
sqrt8ml	154	69,48	11,32	2,08	12,96
t2	230	44,35	20,00	12,33	20,00
table3	1124	28,29	-7,32	-6,61	-3,2
term1	712	80,90	24,02	4,90	20,47
vg2	579	55,09	-11,59	-26,83	18,50
x1dn	190	38,95	13,43	10,77	10,08
x4	743	63,53	26,16	13,97	20,76
xor5	47	74,47	71,43	64,71	71,43
Z9sym	185	21,08	7,01	2,67	4,58
Průměr		36,22	9,22	2,70	9,24

Tabulka 4.9: Prorovnání varianty -e oproti skriptům resyn v procentech část 1.

Název	AND	diff(all)	diff(resyn)	diff(resyn2)	diff(resyn2a)
5xp1	206	51,94	22,66	-2,06	18,85
9sym	333	22,52	7,86	3,73	7,86
9symml	206	7,77	2,06	0,52	2,06
al2	119	26,89	7,45	11,22	10,31
Altera_ac97_ctrl	14272	27,90	1,94	0,07	1,98
Altera_aes_core	21217	11,65	4,40	1,89	4,46
Altera_barrel16	701	52,64	16,37	1,19	15,95
Altera_barrel16a	712	50,14	19,50	3,53	17,06
Altera_barrel32	1778	51,97	12,77	2,29	12,68
Altera_barrel64	4284	49,98	10,75	1,34	10,60
Altera_cfft	13838	37,85	2,79	0,78	3,10
Altera_cord1	11846	32,56	2,55	0,17	2,57
Altera_cord2	15773	34,32	2,18	0,27	2,22
Altera_des_area	4852	11,62	3,18	0,60	3,27
Altera_ether	10820	25,06	6,63	0,11	6,77
Altera_fip_cordic_cla	3033	53,71	8,77	1,68	9,07
Altera_fip_cordic_rca	2937	49,64	6,21	0,40	6,63
Altera_fip_risc8	11011	35,43	3,45	2,95	3,40
Altera_i2c	1170	25,98	9,60	2,37	9,32
Altera_mem	16727	19,18	3,03	-0,28	3,01
Altera_mem_ctrl	15648	53,28	12,40	6,65	14,10
Altera_mux8_128bit	4609	47,17	9,45	9,41	9,45
Altera_mux8_64bit	2305	47,11	9,37	9,30	9,37
Altera_nut_000	1851	59,81	18,87	3,25	20,17
Altera_nut_001	6237	53,25	13,14	1,19	12,83
Altera_nut_002	1207	51,20	19,86	3,92	20,30
Altera_nut_004	781	54,67	19,91	0,84	20,27
Altera_oc_aes_core	12429	40,79	8,91	3,29	9,35
Altera_oc_aes_core_inv	14978	44,33	8,55	2,45	8,71
Altera_oc_aquarius	34800	43,95	2,96	0,59	2,90
Altera_oc_ata_ocidec1	2156	33,44	4,40	0,07	4,27
Altera_oc_ata_ocidec2	2607	37,02	4,03	1,62	3,86
Altera_oc_ata_ocidec3	5327	37,53	5,45	0,09	5,43
Altera_oc_ata_v	1252	41,93	7,74	3,20	5,58
Altera_oc_ata_vhd_3	5258	36,88	5,04	0,36	4,98
Altera_oc_cfft_1024x12	15199	44,54	5,00	-0,1	4,75
Altera_oc_cordic_p2r	13269	39,75	4,46	0,44	4,50
Altera_oc_cordic_r2p	17047	39,71	2,92	1,28	3,43
Altera_oc_correlator	2956	40,32	8,08	1,34	7,30
Altera_oc_dct_slow	1515	43,10	11,59	2,93	11,95
Altera_oc_des_area_opt	4678	47,63	14,52	1,37	15,14
Altera_oc_des_des3area	8011	49,49	7,35	3,69	7,27

Tabulka 4.10: Prorovnání varianty -e oproti skriptům resyn v procentech část 2.

Název	AND	diff(all)	diff(resyn)	diff(resyn2)	diff(resyn2a)
Altera_oc_des_perf_opt	34325	41,28	7,36	3,40	8,03
Altera_oc_ethernet	14921	44,58	5,31	0,02	5,79
Altera_oc_fcmp	770	55,71	26,51	8,33	24,56
Altera_oc_fpu	34175	54,35	6,73	1,71	4,73
Altera_oc_gpio	971	33,26	10,12	1,82	10,25
Altera_oc_hdlc	2998	42,70	7,34	3,48	6,93
Altera_oc_i2c	1383	40,78	7,98	1,09	6,61
Altera_oc_mem_ctrl	20516	35,28	2,92	0,76	2,90
Altera_oc_minirisc	2983	43,85	5,95	0,65	5,85
Altera_oc_miniuart	914	50,66	14,58	4,04	15,86
Altera_oc_mips	25167	41,95	2,24	0,92	1,80
Altera_oc_oc8051	17361	48,14	6,17	0,54	6,15
Altera_oc_pavr	24582	44,46	6,49	2,47	7,11
Altera_oc_pci	12189	36,97	6,60	0,58	6,85
Altera_oc_rtc	2064	54,17	6,71	0,32	6,98
Altera_oc_sdram	1431	42,77	12,59	2,73	13,79
Altera_oc_simple_fm_r	4907	53,50	8,10	-0,09	8,06
Altera_oc_vga_lcd	11185	39,83	5,24	-1,17	4,96
Altera_oc_video_c_s_h_d	2931	49,10	8,30	0,27	6,87
Altera_oc_video_c_s_h_e	3497	50,64	15,39	3,79	15,52
Altera_oc_wb_dma	24170	44,48	3,79	1,78	3,85
Altera_os_blowfish	12676	49,66	5,20	3,87	5,27
Altera_os_sdram16	1997	49,57	10,41	4,55	12,89
Altera_pci_bridge32	22776	22,46	1,43	0,18	1,36
Altera_pci_conf_cyc_a_d	87	4,60	2,35	2,35	2,35
Altera_pci_spoci_ctrl	1394	48,64	21,49	9,02	20,53
Altera_radar12	40445	40,26	7,08	0,02	7,20
Altera_sasc	775	21,81	2,10	0,66	2,10
Altera_simple_spi	1036	23,36	3,52	-0,51	3,52
Altera_spi	3845	18,02	2,29	-2,6	1,53
Altera_ss_pcm	405	4,20	1,27	0,51	0,77
Altera_steppermotordrive	188	36,70	5,56	11,19	13,77
Altera_systemcaes	12609	28,19	11,04	2,62	11,47
Altera_systemcdes	2967	19,92	5,11	3,65	5,00
Altera_ts_mike_fsm	84	60,71	21,43	19,51	21,43
Altera_tv80	9764	29,43	7,21	2,38	6,95
Altera_usb_funct	16051	20,07	4,98	0,78	6,45
Altera_usb_phy	468	25,64	3,06	0,57	3,33
Altera_wb_dma	4074	18,11	2,68	0,48	3,02
Altera_xbar_16x16	1232	42,86	6,38	2,22	6,38
alu2	519	31,79	5,09	-0,28	3,80
alu3	78	14,10	6,94	1,47	6,94

Tabulka 4.11: Prorovnání varianty -e oproti skriptům resyn v procentech část 3.

Název	AND	diff(all)	diff(resyn)	diff(resyn2)	diff(resyn2a)
alu4	3603	46,16	19,54	15,69	17,24
am2910	879	22,87	7,25	2,45	5,96
amd	665	53,83	29,91	11,01	31,93
apex1	1622	33,42	6,09	0,18	0,92
apex2	4545	35,95	-8,86	-7,26	-3,56
apex3	1757	32,78	0,42	-3,42	1,42
apex4	2454	24,74	-3,65	-6,21	-4,94
apex5	3325	78,35	23,57	2,83	20,44
apex6	661	9,98	2,14	1,00	2,14
apex7	276	40,94	12,37	6,32	12,83
apla	277	40,07	12,17	-7,1	13,99
b10	655	45,34	22,17	7,73	22,00
b11	123	45,53	2,90	1,47	2,90
b12	978	94,58	88,25	62,68	87,17
b2	1793	36,92	12,80	4,80	12,53
b3	525	47,81	20,12	4,53	21,49
b4	346	31,21	5,93	2,46	6,67
b7	123	45,53	2,90	1,47	2,90
b9	117	33,33	4,88	3,70	8,24
bbtas	45	48,89	23,33	4,17	23,33
bc0	1626	46,43	25,43	3,97	25,75
bca	3830	37,78	8,63	0,83	8,80
bcb	3319	39,80	9,18	3,10	9,39
bcc	3189	38,63	7,73	3,26	8,08
bcd	2205	37,46	7,57	-0,58	6,76
beecount	54	37,04	8,11	5,56	2,86
bigkey	5149	40,30	22,98	0,49	6,99
br1	209	50,24	22,39	18,11	24,64
br2	192	60,42	28,30	16,48	21,65
bw	180	23,33	5,48	0,72	5,48
c1355	510	23,53	1,27	1,02	1,02
c1908	457	22,98	6,13	2,76	5,38
c2670	709	23,13	3,02	0,91	3,02
c3540	1033	10,75	1,71	0,54	1,50
c432	209	38,76	2,29	12,33	0,78
c499	400	3,50	1,53	1,03	1,53
c5315	1667	21,66	3,33	0,91	2,61
c6288	2337	19,98	0,64	0,00	0,64
c7552	2003	28,66	7,93	0,83	6,84
c8	278	61,51	20,74	1,83	24,11
c880	329	6,69	1,60	1,60	1,60
chkn	444	25,00	7,24	0,89	7,76

Tabulka 4.12: Prorovnání varianty -e oproti skriptům resyn v procentech část 4.

Název	AND	diff(all)	diff(resyn)	diff(resyn2)	diff(resyn2a)
cht	336	55,95	5,13	0,00	2,63
clip	532	53,57	25,60	3,14	23,77
clma	1155	80,09	39,63	5,35	40,57
clmb	23490	76,30	59,39	12,84	58,96
cm152a	31	32,26	22,22	22,22	22,22
cmb	54	31,48	2,63	7,50	9,76
comp	125	33,60	14,43	9,78	16,16
cordic	2785	85,35	62,22	48,22	60,62
cps	1607	33,04	5,36	-1,22	2,27
cu	55	38,18	12,82	12,82	12,82
dalu	1663	43,05	9,81	8,94	9,64
dc1	42	33,33	15,15	9,68	9,68
dc2	130	38,46	14,89	8,05	12,09
dekoder	54	50,00	18,18	15,63	18,18
des	4797	37,48	8,11	4,97	8,37
dist	564	25,00	8,64	2,53	8,84
div16	755	29,54	11,48	10,44	11,63
dk14	124	41,94	11,11	6,49	12,20
dk15	70	28,57	18,03	12,28	9,09
dk17	177	32,77	6,30	4,80	7,03
dk27	90	42,22	11,86	11,86	13,33
dk48	259	45,95	7,28	5,41	9,68
dk512	58	32,76	9,30	11,36	11,36
dsip	2523	0,48	0,16	0,16	0,16
duke2	535	39,44	14,96	8,73	12,43
e64	790	54,94	40,27	40,57	41,54
ex1010	3340	24,55	3,71	1,29	4,04
ex4	471	16,14	2,23	1,00	2,23
ex4p	2108	60,48	41,13	13,41	36,46
ex5	951	60,88	32,49	21,02	32,85
ex5p	2394	46,41	12,06	5,45	14,52
ex7	119	24,37	11,76	0	12,62
example2	321	29,28	10,63	8,47	10,63
exep	867	53,17	17,81	10,96	17,14
exp	439	33,71	11,82	2,68	12,35
exps	1628	29,55	8,82	4,18	9,11
f51m	230	63,04	45,86	35,11	42,95
frg1	617	67,91	17,15	18,52	17,15
frg2	1951	65,81	15,78	3,61	28,66
g125	2250	28,89	4,36	2,08	4,36
g216	6156	45,42	8,62	5,19	8,07
g25	300	28,33	4,87	3,59	4,87

Tabulka 4.13: Prorovnání varianty -e oproti skriptům resyn v procentech část 5.

Název	AND	diff(all)	diff(resyn)	diff(resyn2)	diff(resyn2a)
g36	684	44,30	8,85	4,03	6,85
g625	15000	29,17	4,08	1,36	4,08
gary	606	32,67	13,56	2,86	14,29
i10	2512	31,29	4,32	1,93	4,38
i2	230	9,13	1,42	2,34	1,42
i4	245	17,55	10,62	6,91	10,62
i5	335	54,33	28,17	26,44	27,14
i6	402	3,73	1,78	1,78	1,78
i8	3090	68,64	5,00	5,46	6,74
i9	883	43,71	8,13	8,13	8,13
ibm	229	24,89	16,91	9,95	16,91
idxlat02n_m12	2237	39,56	7,27	2,59	10,05
in0	726	44,49	21,14	3,13	21,90
in1	1793	38,26	14,65	6,82	14,39
in2	544	34,01	17,28	3,49	16,51
in3	574	44,25	13,98	3,90	13,98
in4	545	46,06	16,95	-1,73	17,88
in5	506	41,30	17,73	6,60	18,63
in6	355	31,83	5,84	2,81	6,92
in7	166	24,70	8,09	6,02	9,42
inc	158	39,24	6,80	3,03	4,95
intb	1425	24,84	9,01	1,56	8,77
ITC_b04	451	14,86	3,27	0,00	3,27
ITC_b05	792	44,19	11,24	3,70	11,07
ITC_b06	20	45,00	38,89	21,43	38,89
ITC_b07	351	6,55	3,53	0,30	3,53
ITC_b08	153	16,34	8,57	4,48	7,25
ITC_b09	84	46,43	44,44	43,04	44,44
ITC_b10	174	8,05	2,44	0,00	3,03
ITC_b11	621	23,19	6,47	1,24	7,02
ITC_b12	1005	24,98	13,33	6,57	14,22
ITC_b13	257	15,56	8,44	0,00	8,44
ITC_b14	6063	31,07	9,21	5,09	9,66
ITC_b14_1	4901	30,63	10,20	4,01	10,29
ITC_b15	8400	13,17	3,96	1,29	3,98
ITC_b15_1	8956	24,16	3,10	1,05	3,45
ITC_b21_1	10325	28,73	10,27	4,68	10,57
jbp	566	36,93	7,03	2,99	10,08
ldd	90	40,00	11,48	12,90	10,00
m2	328	43,60	23,55	11,06	23,87
mainpla	5552	37,30	11,24	3,14	10,74
max1024	1021	25,17	9,48	4,38	8,72

Tabulka 4.14: Prorovnání varianty -e oproti skriptům resyn v procentech část 6.

Název	AND	diff(all)	diff(resyn)	diff(resyn2)	diff(resyn2a)
max512	743	27,32	6,74	3,57	6,90
Mentor_1_01	2707	32,84	6,53	0,60	7,34
Mentor_1_05	2883	32,64	6,09	1,52	6,27
Mentor_1_07	9942	30,22	6,53	-0,09	6,50
Mentor_1_08	2704	32,66	7,33	1,62	7,70
Mentor_1_09	1602	33,90	7,99	2,04	7,99
Mentor_1_11	2723	33,57	8,77	2,74	9,10
misex1	58	22,41	6,25	8,16	8,16
misex3	5485	57,06	18,48	18,06	17,05
mm30a	1576	38,20	2,99	2,89	2,99
mm9a	472	36,65	0,00	0,00	0,00
mult16	560	22,50	4,19	0,00	4,19
newapla1	39	48,72	13,04	16,67	16,67
newcpla1	132	41,67	18,95	2,53	18,95
newxcpla1	166	54,22	34,48	6,17	33,91
pair	1740	28,16	5,59	0,64	4,36
pcont2	2914	35,35	4,46	-0,27	4,22
pope	722	39,06	14,23	7,56	14,89
rd53	95	71,58	63,01	35,71	63,01
root	277	37,55	16,83	3,89	13,07
s1238	532	10,34	4,02	0,63	4,02
s1423	462	5,41	1,58	1,13	1,35
s15850.1	3548	23,37	5,20	1,20	4,50
s298	98	25,51	6,41	3,95	6,41
s38417	9158	11,68	1,94	0,41	2,06
s444	151	34,44	10,00	9,17	10,81
s526	199	41,71	13,43	4,13	13,43
s641	147	18,37	6,25	4,76	6,25
s832	358	32,40	6,56	3,20	6,56
s9234	1962	32,47	8,56	0,90	8,30
seq	4233	47,65	2,59	0,89	2,46
spla	1657	60,29	30,37	17,75	30,07
sqrt8ml	154	69,48	11,32	2,08	12,96
t2	230	42,17	16,88	8,90	16,88
table3	1124	37,72	6,79	7,41	10,37
term1	712	79,78	19,55	-0,7	15,79
vg2	579	77,72	44,64	37,07	59,56
x1dn	190	41,58	17,16	14,62	13,95
x4	743	63,53	26,16	13,97	20,76
xor5	47	74,47	71,43	64,71	71,43
Z9sym	185	21,08	7,01	2,67	4,58
Průměr		37,75	11,61	5,37	11,58

Tabulka 4.15: Prorovnání nového skriptu oproti skriptům resyn v procentech část 1.

Název	AND	diff(resyn)	diff(resyn2)	diff(resyn2a)
5xp1	206	13,28	-14,43	9,02
9sym	333	-0,71	-5,22	-0,71
9symml	206	1,55	0	1,55
al2	119	6,38	10,2	9,28
Altera_ac97_ctrl	14272	1,81	-0,07	1,85
Altera_aes_core	21217	4,89	2,4	4,96
Altera_barrel16	701	15,37	0	14,94
Altera_barrel16a	712	16,55	0	14,02
Altera_barrel32	1778	11,34	0,69	11,25
Altera_barrel64	4284	9,41	-0,14	9,26
Altera_cfft	13838	2,15	0,13	2,46
Altera_cord1	11846	2,54	0,16	2,56
Altera_cord2	15773	2,7	0,8	2,74
Altera_des_area	4852	3,54	0,97	3,63
Altera_ether	10820	7,24	0,76	7,38
Altera_fip_cordic_cla	3033	6,43	-0,84	6,74
Altera_fip_cordic_rca	2937	7,55	1,82	7,95
Altera_fip_risc8	11011	-2,91	-3,44	-2,96
Altera_i2c	1170	8,98	1,69	8,69
Altera_mem	16727	3,41	0,11	3,39
Altera_mem_ctrl	15648	13,37	7,69	15,05
Altera_mux8_128bit	4609	0,04	0	0,04
Altera_mux8_64bit	2305	0,07	0	0,07
Altera_nut_000	1851	15,16	-1,17	16,52
Altera_nut_001	6237	11,35	-0,85	11,03
Altera_nut_002	1207	17,01	0,49	17,46
Altera_nut_004	781	18,78	-0,56	19,14
Altera_oc_aes_core	12429	7,92	2,23	8,36
Altera_oc_aes_core_inv	14978	9,9	3,88	10,06
Altera_oc_aquarius	34800	1,49	-0,92	1,42
Altera_oc_ata_ocidec1	2156	3,46	-0,91	3,34
Altera_oc_ata_ocidec2	2607	3,86	1,44	3,69
Altera_oc_ata_ocidec3	5327	5,4	0,03	5,37
Altera_oc_ata_v	1252	7,74	3,2	5,58
Altera_oc_ata_vhd_3	5258	4,75	0,06	4,7
Altera_oc_cfft_1024x12	15199	4,54	-0,58	4,28
Altera_oc_cordic_p2r	13269	2,94	-1,15	2,99
Altera_oc_cordic_r2p	17047	2,19	0,54	2,71
Altera_oc_correlator	2956	8,49	1,79	7,72
Altera_oc_dct_slow	1515	9,64	0,79	10,01
Altera_oc_des_area_opt	4678	14,48	1,33	15,1
Altera_oc_des_des3area	8011	3,62	-0,19	3,53

Tabulka 4.16: Prorovnání nového skriptu oproti skriptům resyn v procentech část 2.

Název	AND	diff(resyn)	diff(resyn2)	diff(resyn2a)
Altera_oc_des_perf_opt	34325	2,93	-1,22	3,63
Altera_oc_ethernet	14921	5,58	0,3	6,05
Altera_oc_fcmp	770	25,22	6,72	23,23
Altera_oc_fpu	34175	4,37	-0,78	2,32
Altera_oc_gpio	971	8,74	0,3	8,86
Altera_oc_hdlc	2998	6,85	2,98	6,45
Altera_oc_i2c	1383	6,97	0	5,59
Altera_oc_mem_ctrl	20516	1,24	-0,96	1,21
Altera_oc_minirisc	2983	5,22	-0,12	5,12
Altera_oc_miniuart	914	11,74	0,85	13,06
Altera_oc_mips	25167	-1,18	-2,54	-1,63
Altera_oc_oc8051	17361	4,46	-1,27	4,44
Altera_oc_pavr	24582	4,04	-0,09	4,67
Altera_oc_pci	12189	5,77	-0,3	6,03
Altera_oc_rtc	2064	2,37	-4,32	2,65
Altera_oc_sdram	1431	11,74	1,78	12,95
Altera_oc_simple_fm_r	4907	7,49	-0,75	7,45
Altera_oc_vga_lcd	11185	5,05	-1,37	4,77
Altera_oc_v_c_s_h_d	2931	11,43	3,68	10,05
Altera_oc_v_c_s_h_e	3497	14,12	2,34	14,24
Altera_oc_wb_dma	24170	-0,04	-2,13	0,03
Altera_os_blowfish	12676	4,52	3,18	4,59
Altera_os_sdram16	1997	12,01	6,26	14,45
Altera_pci_bridge32	22776	1,37	0,11	1,3
Altera_pci_c_c_a_d	87	2,35	2,35	2,35
Altera_pci_spoci_ctrl	1394	16,67	3,43	15,65
Altera_radar12	40445	6,74	-0,35	6,86
Altera_sasc	775	2,1	0,66	2,1
Altera_simple_spi	1036	4,5	0,51	4,5
Altera_spi	3845	4,28	-0,52	3,53
Altera_ss_pcm	405	0,76	0	0,26
Altera_steppermotordrive	188	-2,38	3,73	6,52
Altera_systemcaes	12609	9,41	0,84	9,85
Altera_systemcdes	2967	4,75	3,28	4,64
Altera_ts_mike_fsm	84	0	-2,44	0
Altera_tv80	9764	7,25	2,42	7
Altera_usb_funct	16051	6,07	1,93	7,53
Altera_usb_phy	468	1,67	-0,86	1,94
Altera_wb_dma	4074	2,86	0,66	3,2
Altera_xbar_16x16	1232	4,26	0	4,26
alu2	519	2,68	-2,83	1,36
alu3	78	4,17	-1,47	4,17

Tabulka 4.17: Prorovnání nového skriptu oproti skriptům resyn v procentech část 3.

Název	AND	diff(resyn)	diff(resyn2)	diff(resyn2a)
alu4	3603	11,61	7,39	9,09
am2910	879	8,07	3,31	6,8
amd	665	24,2	3,77	26,39
apex1	1622	23,39	18,58	19,17
apex2	4545	18,44	19,64	22,41
apex3	1757	5,65	2,01	6,59
apex4	2454	4,43	2,07	3,24
apex5	3325	25,58	5,4	22,54
apex6	661	2,14	1	2,14
apex7	276	11,29	5,17	11,76
apla	277	80,95	76,77	81,35
b10	655	21,09	6,44	20,92
b11	123	2,9	1,47	2,9
b12	978	78,94	33,1	77
b2	1793	14,49	6,65	14,23
b3	525	15,45	-1,05	16,91
b4	346	1,98	-1,64	2,75
b7	123	2,9	1,47	2,9
b9	117	3,66	2,47	7,06
bbtas	45	23,33	4,17	23,33
bc0	1626	25,09	3,53	25,4
bca	3830	8,78	1	8,96
bcb	3319	6,86	0,63	7,07
bcc	3189	8,63	4,2	8,97
bcd	2205	11,33	3,5	10,55
beecount	54	10,81	8,33	5,71
bigkey	5149	28,69	7,87	13,89
br1	209	7,46	2,36	10,14
br2	192	24,53	12,09	17,53
bw	180	8,9	4,32	8,9
c1355	510	1,77	1,52	1,52
c1908	457	3,47	0	2,69
c2670	709	0,53	-1,64	0,53
c3540	1033	1,07	-0,11	0,85
c432	209	-12,21	-0,68	-13,95
c499	400	0,51	0	0,51
c5315	1667	2,44	0	1,72
c6288	2337	0,64	0	0,64
c7552	2003	7,93	0,83	6,84
c8	278	20,74	1,83	24,11
c880	329	0,96	0,96	0,96
chkn	444	10,31	4,17	10,8

Tabulka 4.18: Prorovnání nového skriptu oproti skriptům resyn v procentech část 4.

Název	AND	diff(resyn)	diff(resyn2)	diff(resyn2a)
cht	336	5,13	0	2,63
clip	532	24,1	1,18	22,22
clma	1155	36,48	0,41	37,47
clmb	23490	50,45	-6,36	49,92
cm152a	31	7,41	7,41	7,41
cmb	54	-5,26	0	2,44
comp	125	16,49	11,96	18,18
cordic	2785	68,06	56,22	66,7
cps	1607	13,72	7,71	10,9
cu	55	5,13	5,13	5,13
dalu	1663	12,86	12,02	12,69
dc1	42	9,09	3,23	3,23
dc2	130	14,89	8,05	12,09
dekoder	54	6,06	3,13	6,06
des	4797	9,43	6,34	9,69
dist	564	6,05	-0,23	6,25
div16	755	11,48	10,44	11,63
dk14	124	2,47	-2,6	3,66
dk15	70	14,75	8,77	5,45
dk17	177	-0,79	-2,4	0
dk27	90	10,17	10,17	11,67
dk48	259	7,28	5,41	9,68
dk512	58	2,33	4,55	4,55
dsip	2523	0,08	0,08	0,08
duke2	535	13,91	7,61	11,35
e64	790	35,57	35,89	36,95
ex1010	3340	3,82	1,41	4,15
ex4	471	0,74	-0,50	0,74
ex4p	2108	37,39	7,9	32,42
ex5	951	17,60	3,61	18,05
ex5p	2394	48,87	45,03	50,3
ex7	119	8,82	-3,33	9,71
example2	321	3,54	1,21	3,54
exep	867	21,05	14,47	20,41
exp	439	8,18	-1,34	8,73
exps	1628	7,15	2,42	7,45
f51m	230	18,47	2,29	14,09
frg1	617	9,62	11,11	9,62
frg2	1951	15,53	3,32	28,45
g125	2250	2,87	0,55	2,87
g216	6156	5,93	2,4	5,36
g25	300	2,65	1,35	2,65

Tabulka 4.19: Prorovnání nového skriptu oproti skriptům resyn v procentech část 5.

Název	AND	diff(resyn)	diff(resyn2)	diff(resyn2a)
g36	684	7,66	2,77	5,62
g625	15000	2,95	0,19	2,95
gary	606	14,19	3,57	14,92
i10	2512	5,38	3,01	5,43
i2	230	0,47	1,4	0,47
i4	245	9,29	5,53	9,29
i5	335	23	21,15	21,9
i6	402	1,78	1,78	1,78
i8	3090	7,35	7,8	9,05
i9	883	5,91	5,91	5,91
ibm	229	11,11	3,66	11,11
idxlat02n_m12	2237	25,03	21,25	27,28
in0	726	18,59	0	19,38
in1	1793	14,49	6,65	14,23
in2	544	18,2	4,57	17,44
in3	574	9,68	-0,9	9,68
in4	545	14,12	-5,19	15,08
in5	506	9,7	-2,52	10,68
in6	355	2,33	-0,8	3,46
in7	166	5,88	3,76	7,25
inc	158	-0,97	-5,05	-2,97
intb	1425	8,41	0,92	8,18
ITC_b04	451	3,27	0	3,27
ITC_b05	792	10,84	3,27	10,66
ITC_b06	20	27,78	7,14	27,78
ITC_b07	351	3,53	0,3	3,53
ITC_b08	153	7,14	2,99	5,8
ITC_b09	84	44,44	43,04	44,44
ITC_b10	174	0	-2,5	0,61
ITC_b11	621	4,71	-0,62	5,26
ITC_b12	1005	8,05	0,87	8,99
ITC_b13	257	8,44	0	8,44
ITC_b14	6063	6,26	2	6,72
ITC_b14_1	4901	10,41	4,23	10,5
ITC_b15	8400	4,48	1,81	4,49
ITC_b15_1	8956	3,44	1,4	3,8
ITC_b21_1	10325	9,17	3,51	9,48
jbp	566	3,39	-0,82	6,55
ldd	90	4,92	6,45	3,33
m2	328	18,18	4,81	18,52
mainpla	5552	10,3	2,11	9,79
max1024	1021	11,14	6,13	10,39

Tabulka 4.20: Prorovnání nového skriptu oproti skriptům resyn v procentech část 6.

Název	AND	diff(resyn)	diff(resyn2)	diff(resyn2a)
max512	743	4,84	1,61	5
Mentor_1_01	2707	7,61	1,75	8,41
Mentor_1_05	2883	6,67	2,13	6,85
Mentor_1_07	9942	8,29	1,79	8,25
Mentor_1_08	2704	7,23	1,51	7,6
Mentor_1_09	1602	7,3	1,3	7,3
Mentor_1_11	2723	7,61	1,51	7,94
misex1	58	2,08	4,08	4,08
misex3	5485	12,7	12,25	11,17
mm30a	1576	-0,2	-0,3	-0,2
mm9a	472	-0,67	-0,67	-0,67
mult16	560	4,19	0	4,19
newapla1	39	8,7	12,5	12,5
newcpla1	132	16,84	0	16,84
newxcpla1	166	30,17	0	29,57
pair	1740	3,1	-1,99	1,84
pcont2	2914	4,77	0,05	4,52
pope	722	10,33	3,36	11,03
rd53	95	17,81	-42,86	17,81
root	277	12,5	-1,11	8,54
s1238	532	3,42	0	3,42
s1423	462	0,45	0	0,23
s15850.1	3548	4,36	0,33	3,65
s298	98	-3,85	-6,58	-3,85
s38417	9158	1,88	0,34	2
s444	151	3,64	2,75	4,5
s526	199	6,72	-3,31	6,72
s641	147	1,56	0	1,56
s832	358	6,56	3,2	6,56
s9234	1962	8,07	0,37	7,82
seq	4233	20,92	19,54	20,82
spla	1657	18,52	3,75	18,17
sqrt8ml	154	3,77	-6,25	5,56
t2	230	14,38	6,16	14,38
table3	1124	19,71	20,24	22,79
term1	712	29,05	11,19	25,73
vg2	579	31,76	22,44	50,16
x1dn	190	17,16	14,62	13,95
x4	743	19,89	6,67	14,04
xor5	47	54,76	44,12	54,76
Z9sym	185	6,37	2	3,92
Průměr		10,09	3,50	10,05

Kapitola 5

Závěr

Zadáním diplomové práce bylo vytvořit program pro paralelizaci logické syntézy, naměření výsledků provádění a srovnání se sekvenčním vykonáváním. Všechny tyto body byly úspěšně splněny. Nad rámec zadaní byla v rámci aplikace vytvořena i varianta na sekvenční spouštění syntézy za pomoci paralelismu.

Při zpracování výsledků měření bylo dosaženo uspokojivých výsledků a bylo prokázáno, že provádění paralelizace logické syntézy dosahuje ve většině případů lepších výsledků než klasické sekvenční spouštění. Navíc bylo odhaleno, že pokud se při provádění zabuduje do kódu zvýhodněné propagování prozatím nejlepšího dosaženého výsledku, výsledky vykonávání se razantně zlepší.

Ve výsledku bylo potvrzeno, že zatímco sekvenční vykonávání logické syntézy již po několika desítkách iterací nedokáže přeskupit obvod tak, aby bylo možno dosáhnout zlepšení, paralelizace logické syntézy je schopna tuto cestu najít.

Důvodem je to, že program náhodně vybírá již zpracované obvody a snaží se nad nimi vykonat jiný příkaz. Zatímco se tedy sekvenční vykonávání zastaví, paralelní vykonávání může pokračovat z jiného bodu, a dosáhnout zlepšení.

Z těchto důvodů se dá potvrdit, že paralelizace logické syntézy je správným krokem a umožňuje dosáhnout lepších výsledků než standardní sekvenční metoda.

V této práci byla taktéž nalezena sekvence příkazů, která dosahuje u většiny obvodů lepších výsledků než skripty resyn. Bylo tedy dokázáno, že je možno, pomocí paralelizace logické syntézy, odhalit nové skripty. Nalezené skripty můžeme využít při sekvenčním provádění logické syntézy.

Poslední důležitým objevem této práce je, že sekvenční náhodné spouštění syntézních příkazů dosahuje lepších výsledků než syntézní skripty. Pokud mohu provést hodně iterací, je lepší vykonávat sekvenční spouštění náhodných příkazů, než spouštět syntézní skripty.

Tuto práci je možno dále rozšiřovat, neboť veškeré programové vybavení je dostupno, a je tedy možno rekonstruovat měření nebo vytvářet nové měření s jinými příkazy a parametry. Další výhodou pro nadcházející pokračování v této práci je programově automatizované zpracování výsledků, které usnadňuje budoucí práci.

Pokračování této práce by se mohlo zaobírat zakomponováním příkazů collapse a dsd do měření. Tím by se dalo určit, jestli není možno vylepšit výsledky i u obvodů, u kterých

prozatím ke zlepšení nedošlo. Taktéž je možno pokračovat v odhalování nových sekvencí příkazů.

Literatura

[1] BRGLEZ, F. – FUJIWARA, H. A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortan. *Proc. of International Symposium on Circuits and Systems.* 1985, s. 663 – 698.

[2] BRGLEZ, F. – BRYAN, D. – KOZMINSKI, K. Combinational Pro_les of Sequential Bench-mark Circuits. *Proc. of International Symposium on Circuits and Systems.* 1989, s. 1929 – 1934.

[3] MCELVAIN, K. LGSynth93 Benchmark Set: Version 4.0. *Mentor Graphics.* 1993, s. 1929 – 1934.

[4] MISHCHENKO, A. – BRAYTON, R. Scalable Logic Synthesis using a Simple Circuit Structure. *Proc. IWLS '06. http://www.eecs.berkeley.edu/ alanmi/publications/2006/iwls06_ sls.pdf.* 2006.

[5] MISHCHENKO, A. – CHATTERJEE, S. – BRAYTON, R. DAG-aware AIG rewriting: A fresh look at combinational logic synthesis. *Proc. DAC '06. http://www.eecs.berkeley.edu/ alanmi/publications/2006/dac06_ rwr.pdf.* 2006.

[6] MISHCHENKO, A. ABC : A System for Sequential Synthesis and Verification, . `http://www.eecs.berkeley.edu/~alanmi/abc/`, stav z 3. 12. 2011.

[7] MISHCHENKO, A. ABC : A System for Sequential Synthesis and Verification Repository, . `https://bitbucket.org/alanmi/abc`, stav z 3. 12. 2011.

[8] VANDERWIEL, S. P. – NATHANSON, D. – LILJA, D. J. Complexity and Performance in Parallel Programming Languages. *IEEE High-Level Programming Models and Supportive Environments, 1997. Proceedings., Second International Workshop on.* 1997, 10, 1109, s. 3 – 12.

[9] web:mpi. MPICH2 : High-performance and Widely Portable MPI. `http://www.mcs.anl.gov/research/projects/mpich2/`, stav ze 3. 12. 2011.

[10] web:par. - - - + + + + - = | S t a r C l u s t e r | = - + + + + - - -. `http://service.felk.cvut.cz/courses/X36PAR/`, stav ze 6. 12. 2011.

[11] web:starwiki. start - Star. `http://star.fit.cvut.cz/dokuwiki/`, stav ze 6. 12. 2011.

Příloha A

Seznam použitých zkratek

MPI Message Passing Interface - protokol pro podporu paralelního řešení výpočetních problémů v počítačových clusterech

ABC nástroj pro logickou syntézu

PVM Paralel Virtual Machine - nástroj pro podporu paralelního řešení výpočetních problémů v počítačových clusterech

AIG And-Invertor Graf

⋮

Příloha B

Instalační a uživatelská příručka

B.1 Operační systém Windows

B.1.1 Kompilace statické knihovny ABC

V operačním systému Windows je kompilace statické knihovny velice jednoduchá a dá se učinit v následujících krocích.

- stáhnout zdrojové kódy nástroje *ABC* (nejlépe z úložiště tohoto nástroje [7])
- otevřít projekt nástroje v Microsoft Visual Studiu 2010 (2008) přes soubor *abcspace.dsw*
- spustit kompilaci přes Build -> Rebuild Solution
- po provedení kompilace bude soubor statické knihovny *abcd.lib* ve složce lib

Soubor statické knihovny je pak možno použít v aplikaci, jak je popsáno v kapitole B.1.3.

B.1.2 Instalace MPI

I když je aplikace primárně určena pro práci na výpočetním clusteru, je možno ji spustit na lokálním počítači s operačním systémem Windows. Pro spuštění aplikace je nutné mít nainstalovanou knihovnu *MPI*.

Nejdříve je nutno stáhnout instalační balíček z následujícího odkazu [9] v záložce download. Odtud stáhneme nejnovější stabilní uvolněnou verzi s označením MPICH2 Windows IA32 (binary). Instalace pak probíhá následujícím způsobem:

- spuštění instalačního balíčku mpich2-<version>-win-ia32.msi, tím se spustí služba Windows Installer
- kliknutí na tlačítko Next
- pročtení informací o knihovně MPICH2 a kliknutí na Next

- přečtení a odsouhlasení licenční podmínky zvolení volby I Agree a poté kliknutí na Next
- nyní instalace upozorňuje, že je nutno být při instalaci přihlášen jako uživatel v administrátorské skupině, dále se nainstaluje služba Smpd, která zajišťuje spouštění MPI procesů, pro spouštění je nutno si zvolit heslo
- zvolit si adresář pro instalaci MPICH2 a to, jestli bude knihovna dostupná jen přihlášenému uživateli nebo všem uživatelům počítače
- kliknutí na Next provede samotnou instalaci
- dokončení instalace a její ukončení kliknutím na Close

Nyní je možno *MPI* používat na počítači se systémem Windows. Pro spuštění aplikace naprogramované pro *MPI* je nutno si do složky obsahující spouštěcí soubor zkopírovat soubor mpiexec.exe z následujícího adresáře.

```
c:\Program Files\MPICH2\bin\
```

Aplikace využívající *MPI* se spouští následujícím příkazem.

```
mpiexec -n <number> ./<PROGRAM\_NAME>.exe
```

B.1.3 Kompilace aplikace

Ke zkompilování aplikace na systému Windows je nejlépe mít nainstalované prostření Visual Studio 2010 od firmy Microsoft. Pomocí tohoto prostředí je možno si otevřít již vytvořený Projekt MPI_abc přes soubor *MPI_ abc.sln*, který se nachází na přiloženém CD. Po otevření projektu je nutno nastavit cesty k prostředí *MPI* podle následujících bodů.

- Project Properties - v Solution Explorer pravým tlačítkem kliknout na projekt MPI_abc a zvolit možnost Properties
- Additional Include Directories - v Project Properties zvolit položku Configuration Properties -> C/C++ -> General -> Additional Include Directories, zde zvolit <Edit...> a přidat složku include ze složky MPICH2

  ```
  C:\Program Files\MPICH2\include
  ```

- Additional Library Directories - v Project Properties zvolit položku Configuration Properties -> Linker -> General -> Additional Library Directories, zde zvolit <Edit...> a přidat složku lib ze složky MPICH2 a cestu ke statické knihovně *ABC*

  ```
  C:\Program Files\MPICH2\lib
  C:\<PROJECT_PATH>\MPI_abc\MPI_abc\abclib
  ```

Po těchto změnách je možno aplikaci zkompilovat přes Build -> Rebuilt MPI_abc. Výsledný spustitelný soubor *MPI_ abc.exe* pak nalezneme v následující složce.

```
C:\<PROJECT_PATH>\MPI_abc\Debug
```

Před spuštěním aplikace je nutno do složky aplikace zkopírovat soubor mpiexec.exe ze složky

```
C:\Program Files\MPICH2\bin\
```

Spuštění aplikace pak probíhá přes příkaz

```
mpiexec -n <number> ./<PROGRAM\_NAME>.exe
```

B.2 Operační systém Linux

B.2.1 Kompilace statické knihovny

Kompilace statické knihovny pod systémem Linux je trochu složitější. Je nutno udělat několik změn jak v aplikaci, tak v souboru Makefile. Popis provedených změn, které zajistí úspěšné zkompilování, je popsán na následujících řádcích.

- nejprve převede některé ze souborů do unixového tvaru a nastavíme práva pomocí následujících příkazů

  ```
  dos2unix Makefile Makefile
  dos2unix depends.sh depends.sh
  chmod 755 depends.sh
  ```

- dále provedeme tyto změny v souboru Makefile

  ```
  #all: $(PROG)
  #default: $(PROG)
  all: lib$(PROG).a
  default: lib$(PROG).a
  ```

- poté upravíme přímo některé soubory *ABC*
- v souboru *main.c*, který se nachází ve složce ./src/base/main, je třeba odkomentovat řádek

  ```
  #define ABC_LIB
  ```

  poté upravíme následující část kódu, aby byla uzavřena v #ifndef ABC_LIB

```
#ifndef ABC_LIB
  return /*ABC_NAMESPACE_PREFIX*/ Abc_RealMain(argc, argv);
#endif
```

- poslední úpravou je zakomentování příkazů **readline()** a **add_history()** ve funkci **Abc_UtilsGetUsersInput()**, kterou nalezneme v souboru *mainUtils.c*, výsledná funkce pak vypadá následovně

```
char * Abc_UtilsGetUsersInput( Abc_Frame_t * pAbc )
{
    static char Prompt[1000];
#ifndef _WIN32
    static char * line = NULL;
#endif

    sprintf( Prompt, "abc %02d> ", pAbc->nSteps );
#ifdef _WIN32
    fprintf( pAbc->Out, "%s", Prompt );
    fgets( Prompt, 999, stdin );
    return Prompt;
#else
    if (line != NULL) ABC_FREE(line);
    //line = readline(Prompt);
    if (line == NULL){ printf("***EOF***\n"); exit(0); }
    //add_history(line);
    return line;
#endif
}
```

- samotnou kompilaci statické knihovny *ABC* provedeme příkazem

```
make
```

Soubor statické knihovny můžeme použít v aplikaci, jak je popsáno v kapitole B.2.3.

B.2.2 Instalace MPI

Pro zprovoznění knihovny **MPI** je třeba stáhnout zdrojové kódy tohoto paralelního prostředí. V této práci byla zvolena nejrozšířenější verze, a to MPICH2. Stažení je možno z následujícího odkazu [9] v záložce download. Odtud stáhneme nejnovější stabilní uvolněnou verzi s označením MPICH2 Source (UNIX and Windows). Postup zprovoznění jsem převzal ze souboru readme, který je obsažen v instalačním balíku.

- zkopírovat stáhnutý balík zdrojových kódů MPICH2 na stroj s operačním systémem Linux

- rozbalit balík přímo v /home/<USERNAME> pomocí příkazu

  ```
  tar xzf mpich2-1.4.1p1.tar.gz
  ```

  vytvoří se složka mpich2-1.4.1p1, do které přistoupíme pomocí příkazu

  ```
  cd mpich2-1.4.1p1
  ```

  pokud nelze rozbalit instalační balík tímto způsobem zkusíme následující příkazy

  ```
  gunzip mpich2-1.4.1p1.tar.gz
  tar xf mpich2-1.4.1p1.tar
  cd mpich2-1.4.1p1
  ```

- zvolit si adresář pro instalaci (buď prázdný nebo neexistující)

  ```
  /home/<USERNAME>/mpich2-install
  ```

- konfigurovat instalační adresář

  ```
  ./configure --prefix=/home/<USERNAME>/mpich2-install 2>&1 | tee c.txt
  ```

- zkompilovat MPICH2:

  ```
  make 2>&1 | tee m.txt
  ```

- instalovat MPICH2

  ```
  make install 2>&1 | tee mi.txt
  ```

- přidat podadresář bin instalačního adresáře do PATH

  ```
  PATH=/home/<USERNAME>/mpich2-install/bin:$PATH ; export PATH
  ```

- zkontrolovat, jestli vše prozatím proběhlo v pořádku příkazy

  ```
  which mpicc
  which mpiexec
  ```

  Při každém spuštění pomocí *which mpicc* zkontrolovat, jestli je podadresář bin nastaven do proměnné PATH. Pokud tomu tak není, je nutno to provést přes výše uvedený příkaz, jinak spuštění programu zahlásí chybu.

- otestovat funkčnost spuštěním ukázkové úlohy CPI s 'n' procesy na lokálním počítači

  ```
  mpiexec -n <number> ./examples/cpi
  ```

B.2.3 Kompilace aplikace

Pro zkompilování zdrojového kódu pod operačním systémem Linux je nejlépe použít Makefile z ./mpich2-1.4.1p1/examples. V tomto Makefile se samozřejmě musí udělat příslušné úpravy.

Aby bylo možno úpravy brát univerzálně pro jakoukoliv aplikaci, je název aplikace nahrazen následujícím řetězcem <PROGRAM_NAME>.

- abs_builddir - nastavit na adresář, kam se bude kompilovat - /home/<USERNAME>/
- abs_srcdir - nastavit na adresář se zdrojovými kódy - /home/<USERNAME>/
- MPICHDIR - nastavit na adresář MPICH2 - /home/<USERNAME>/mpich2-install
- CC - nastavit na hodnotu

  ```
  $(MPICHDIR)/bin/mpic++
  ```

- CFLAGS - nastavit na hodnotu

  ```
  -I$(MPICHDIR)/include
  ```

- INCLUDES - nastavit na hodnotu

  ```
  -I$(MPICHDIR)/include -I$(MPICHDIR)/include
  ```

- LIBS - nastavit na hodnotu

  ```
  -L/home/<USERNAME>/lib/ lib/libabc.a -L$(MPICHDIR)/lib
  ```

- EXTRA_PROGRAMS - přepsat na název aplikace - <PROGRAM_NAME>
- .SUFFIXES - nastavit podle programovacího jazyka, v tomto případě na .cpp
- clean - přepsat na název aplikace - <PROGRAM_NAME>
- all-redirect - nastavit na název aplikace - <PROGRAM_NAME>
- vytvořit pravidlo pro kompilaci aplikace.

  ```
  <PROGRAM\_NAME>: <PROGRAM\_NAME>.o $(MPICHDIR)/lib/lib${MPILIBNAME}.a
    $(C\_LINK) $(CFLAGS) $(LDFLAGS) -o <PROGRAM\_NAME> <PROGRAM\_NAME>.o
    -lm ${LIBS}
  ```

- SOURCES - doplnit zdrojové kódy aplikace <PROGRAM_NAME>.cpp
- all-programs - nastavit na název aplikace - <PROGRAM_NAME>

Po těchto úpravách v Makefile lze aplikaci v pořádku zkompilovat příkazem

```
make
```

Poté lze aplikaci spustit následujícím příkazem s tím, že je nutno zadat spouštění parametry, pokud nějaké jsou.

```
mpiexec -n <number> ./<PROGRAM\_NAME>
```

B.3 Spouštění programu MPI_abc

Samotný program MPI_abc je dostupný ve třech variantách. Jednotlivé varianty jsou dostupné přes přepínač, který označuje, která z variant programu se má spustit. Jednotlivé varianty a přepínače jsou pak následující:

- **-r** - randomizovaná verze, u které se vybírají obvody pro vykonávání náhodně
- **-e** - randomizovaná verze s vestavěným elitismem, to spočívá v tom, že obvody vybírá náhodně jak u předchozí varianty, avšak při vybírání vybírá náhodně, jestli předá náhodný obvod, nebo prozatím nejlepší dosažený
- **-n** - nerandomizovaná verze spouští předem dané příkazy nad obvody po určený počet iterací

Spuštění samotné aplikace pak probíhá následujícím příkazem.

```
mpiexec -n <number> ./MPI\_abc
```

Jednotlivé varianty programu se spouštějí s různými argumenty, proto je spouštění každé varianty popsáno zvlášť.

B.3.1 Randomizované verze

Tyto verze programu jsou označeny přepínači **-r** a **-e**. Jedná se o hlavní varianty této práce, které spouští různé základní syntézní příkazy nad různými daty. Pro spuštění programu je nutno mu předat následující argumenty.

- <prepinac> - **-r** nebo **-e**
- <pocet_iteraci> - udává, po kolik iterací se budou provádět předané syntézní příkazy.
- <nazev_obvodu> - tato verze programu se spouští vždy pro jeden obvod, tento argument pak udává název obvodu umístěného ve složce ./Networks
- <unikatni_cislo> - slouží pro rozlišení dočasných složek pro jednotlivé obvody

Aby bylo možno program v této verzi úspěšně spustit, je nutno mít ve složce aplikace složku ./Network, která obsahuje všechny obvody, jež chceme vykonávat. Poslední nutností ke spuštění je mít ve složce programu textový soubor commands.txt, který obsahuje seznam základních syntézních příkazů, které se budou spouštět.

Příklad spuštění programu v těchto variantách:

```
mpiexec -n 4 ./MPI_abc.exe -r 10000 apex4.blif 1
mpiexec -n 4 ./MPI_abc.exe -e 10000 apex4.blif 1
```

V těchto příkazech mpiexec udává agenta *MPI* a přepínač *-n* udává na kolika procesech bude program spuštěn.

B.3.2 Nerandomizovaná verze

Tato verze programu je označena přepínačem **-n**. Tato varianta slouží převážně pro sekvenční spouštění syntézních příkazů. Pro samotné spuštění je nutno spustit program s těmito parametry.

- <prepinac> - **-n**
- <slozka_obvodu> - cesta ke složce obvodů, které se mají vykonat
- <soubor_prikazu> - cesta k souboru obsahující syntézní příkazy pro vykonávání
- <pocet_iteraci> - udává, kolikrát se má každý příkaz nad každým obvodem vykonat

Příklad spuštění programu v této variantě:

```
mpiexec -n 4 ./MPI_abc.exe -n ./Networks ./resyn.txt 10000
```

Příloha C

Pomocné programy

Tato kapitola obsahuje seznam a stručný popis všech pomocných programů, které byly vytvořeny pro usnadnění měření a zpracování výsledků. Všechny tyto programy byly vytvořeny v prostředí Visual Studio 2010 a v programovacím jazyku C#. Tím je jejich použití omezeno na operační systém Windows. K jejich spouštění je dále nutno mít nainstalovanou knihovnu .NET Framework. Veškeré informace o zprovoznění a spouštění aplikací je popsáno v souborech readme.txt, které se nacházejí u každého programu na přiloženém CD.

C.1 Generování spouštěcích skriptů

Aby u spouštění aplikace na výpočetním clusteru *STAR* odpadla nutnost vytvářet spouštěcí skripty, byl vytvořen následující program. Tento program podle zadaných argumentů vytvoří spouštěcí skripty pro všechny zadané obvody. Program se nachází na přiloženém CD ve složce GenerateScripts.

Jeho spuštění vyžaduje následující argumenty.

- name - uživatelské jméno na clusteru STAR pro vytvoření cest
- prepinac - určuje, jaká verze aplikace se spustí: -r náhodné vybírání, -e elitismus
- numberIteration - počet iterací, které se provedou nad obvodem
- numberProceses - počet procesů, na kterých se úloha spustí (mělo by korespondovat s volbou fronty pro spuštění)
- fronta - volba fronty pro spouštění na výpočetním clusteru STAR

Na výpočetním clusteru se nachází několik možností volby fronty. Jejich stručná charakteristika je následující.

- default - program je možno spustit maximálně na osmi procesech a vykonávání bude probíhat maximálně 30 minut

- fast - program je možno spustit maximálně na osmi procesech a vykonávání bude probíhat maximálně 1 minutu
- long - program je možno spustit maximálně na 32 procesech a vykonávání bude probíhat maximálně 40 minut
- serial - program je možno spustit maximálně na jednom procesu a vykonávání bude probíhat maximálně 60 minut
- short - program je možno spustit maximálně na 16 procesech a vykonávání bude probíhat maximálně 20 minut

Při spuštění programu je nutno mít složku ./Networks, kde se nacházejí obvody, pro něž mají být vytvořeny skripty.

Na výstupu programu pak obdržíme složku ./Scripts, která obsahuje vygenerované skripty a soubor runFile.txt, ve kterém jsou příkazy pro spuštění skriptů.

C.2 Zpracování výsledků měření

Tato aplikace přijme naměřená data ze všech měření. Jejich hodnoty naskládá do jednoho souboru a vytvoří graf znázorňující porovnání jednotlivých variant.

Programu jsou data zadána do tří složek.

- složka obsahující naměřené výsledky varianty *-r*

  `c:\<PROJECT_PATH>\ExcelInCSharp\bin\Debug\Vysledky\`

- složka obsahující naměřené výsledky varianty *-e*

  `c:\<PROJECT_PATH>\ExcelInCSharp\bin\Debug\Elitism\`

- složka obsahující naměřené výsledky varianty *resyn*

  `c:\<PROJECT_PATH>\ExcelInCSharp\bin\Debug\Resyn\`

Aplikace načte soubory ze složky Vysledky. Poté zjistí, jestli k nim existují příslušná naměřená data ve složce Resyn. Pokud ano, vezme tyto statistiky a zároveň načte i příslušný soubor ve složce Elitism. Všechny výsledky se zaznamenají do výstupního souboru *.xls, který je pro každou síť unikátní. V tomto souboru je pak automaticky vygenerovaný graf porovnávající jednotlivé varianty. Tyto soubory se uloží do složky Grafy. Grafy zobrazují dolní konvexní obálky jednotlivých výsledků.

Pokud jsou data připravena v příslušných složkách, stačí spustit spustitelný soubor **ExcelInCSharp.exe** a kliknout na tlačítko Start.

Informace označena jako *Zpracování souboru* upozorňuje, který soubor je zpracováván. *Průběh zpracování* pak informuje o fázi zpracovávání a počtu zpracovaných řádků souboru.

C.3 Nalezení nejdelší společné sekvence

Tato aplikace přijme naměřené sekvence ze všech měření. Provede porovnání každý s každým a nalezne nejdelší společné sekvence. Z těchto údajů vytvoří tabulku obsahující nejdelší společné sekvence a jejich četnosti výskytů. Program se nachází na přiloženém CD ve složce LCS.

Programu jsou zadána data o sekvencích příkazů pomocí souboru *results.csv*, který je výstupem programu C.2.

Na výstupu programu je soubor *sekvence.csv*, obsahující nejdelší společné sekvence a jejich četnosti.

C.4 Porovnání nového skriptu se skripty resyn

Tato aplikace přijme naměřené výsledky pro novou sekvenci příkazů a provede porovnání těchto výsledků s výsledky skriptů *resyn*. Program se nachází na přiloženém CD ve složce LCSResynCompare.

Programu jsou zadána tato data.

- složka obsahující naměřené výsledky nově vytvořené sekvence příkazů

  `c:\<PROJECT_PATH>\LCSResynCompare\bin\Debug\NewScript\`

- soubor obsahující naměřené výsledky varianty *resyn*

  `c:\<PROJECT_PATH>\LCSResynCompare\bin\Debug\results.csv`

Aplikace načte názvy obvodů ze souboru *results.csv*. Poté zjistí, jestli k nim existují příslušná naměřená data ve složce NewScript. Pokud ano, vezme tyto statistiky a zapisuje jednotlivé výsledky do souboru *newscriptcompare.csv*.

C.5 Porovnání volání náhodných příkazů se skripty resyn

Tato aplikace přijme naměřené výsledky pro sekvenční volání náhodných příkazů a provede porovnání těchto výsledků s výsledky skriptů *resyn*. Program se nachází na přiloženém CD ve složce SekvenceResynCompare.

Programu jsou zadána tato data.

- složka obsahující naměřené výsledky sekvenčního volání náhodných příkazů

  `c:\<PROJECT_PATH>\SekvenceResynCompare\bin\Debug\Sekvence\`

- složka obsahující naměřené výsledky syntézních skriptů *resyn*

  `c:\<PROJECT_PATH>\SekvenceResynCompare\bin\Debug\Resyn\`

Aplikace načte postupně výsledky pro jednotlivé obvody ze složky Sekvence. Poté zjistí, jestli k nim existují příslušná naměřená data ve složce Resyn. Pokud ano, vezme tyto statistiky a zapisuje jednotlivé výsledky do souboru *sekvcompare.csv*.

Příloha D

Obsah přiloženého CD

```
\
|   readme.txt
|
+---.NET_Framework_3.5
|       dotNetFx35setup.exe
|
+---ExcelInCSharp
|   |   readme.txt
|   |
|   \---ExcelInCSharp
|
+---GenerateScripts
|   |   readme.txt
|   |
|   +---GenerateScripts
|   |
|   \---GenerateTimeList
|
+---LCS
|   |   readme.txt
|   |
|   +---LCS
|   |
|   +---LCSResynCompare
|   |
|   \---SekvenceResynCompare
|
+---MPICH2
|   |   mpich2-1.3.2p1-win-ia32.msi
|   |   readme.txt
|   |
|   \---mpich2-1.4.1p1
```

```
|
|
+---Namerene_vysledky
|   +---Paralelni_vykonavani
|   |   +---10000_iteraci_-e
|   |   |
|   |   \---10000_iteraci_-r
|   |
|   +---Sekvencni_vykonavani
|   |   +---ResultsResyn2a_1
|   |   |
|   |   +---ResultsResyn2_1
|   |   |
|   |   \---ResultsResyn_1
|   |
|   \---Testovani_noveho_skriptu
|
+---Nastroj_ABC
|   |   readme.txt
|   |
|   \---ABC
|
+---Program_MPI_abc
|   |   readme.txt
|   |
|   +---Unix
|   |   |
|   |   \---lib
|   |
|   \---Windows
|       \---MPI_abc
|
+---Text
|   |   sirekond.pdf
|   |   sirekond.tex
|
\---Zpracovane_vysledky
    |   newscriptcompare.csv
    |   results.csv
    |   sekvence.csv
    |
    \---Grafy
```