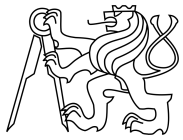
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Paralelní optimalizace logických obvodů
Student:	Bc. Lukáš Rusin
Vedoucí:	doc. Ing. Petr Fišer, Ph.D.
Studijní program:	Informatika
Studijní obor:	Návrh a programování vestavných systémů
Katedra:	Katedra číslicového návrhu
Platnost zadání:	Do konce zimního semestru 2017/18

Pokyny pro vypracování

Vytvořte nástroj pro optimalizaci logických obvodů s možností využití více CPU jader. Logický obvod (netlist) bude rozdělen na části, které budou paralelně optimalizovány (resyntetizovány) jedním, případně více optimalizačními algoritmy. Výsledný obvod pak bude vytvořen opětovným složením těchto částí. Uvažujte paralelizmus ve smyslu rozdělení netlistu a paralelizmus ve smyslu aplikace různých optimalizačních algoritmů. Nástroj důkladně otestujte na standardních zkušebních úlohách a ověřte jeho efektivitu a škálovatelnost.

Doporučený jazyk: C++, MPI.

Seznam odborné literatury

Dodá vedoucí práce.

doc. Ing. Hana Kubátová, CSc.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 9. září 2016

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra číslicového návrhu

Diplomová práce

Paralelní optimalizace logických obvodů

Bc. Lukáš Rusin

Vedoucí práce: doc. Ing. Petr Fišer, Ph.D.

28. června 2017

Poděkování

Na tomto místě bych rád poděkoval vedoucímu své práce doc. Ing. Petru Fišerovi, Ph.D. za zpětnou vazbu, trpělivost i přivedení na správný přístup u testování práce.

Tato práce vznikla za podpory projektů CERIT Scientific Cloud (LM2015085) a CESNET (LM2015042) financovaných z programu MŠMT Projekty velkých infrastruktur pro VaVaI.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V Praze dne 28. června 2017 .....

Odkaz na tuto práci

Rusin, Lukáš. *Paralelní optimalizace logických obvodů.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2017.

Abstrakt

Tato práce se zabývá rozdělováním logického obvodu na vhodné části, které následně paralelně resyntetizuje za účelem dosažení lepšího konečného výsledku, než kdyby se obvod resyntetizoval vcelku. Představen je vlastní algoritmus pro rozdělování. Výstupy obou variant zpracování obvodu jsou porovnány řadou experimentů. Vyhodnocen je i časový přínos, který do resyntézy vnáší rozdělování.

Klíčová slova okno obvodu, iterační rozdělování, paralelní resyntéza, kvalita obvodu, rychlost zpracování

Abstract

This thesis introduces the logic circuit division into appropriate parts, which are subsequently resynthesised in parallel. There is a purpose to get better final result than the whole circuit resynthesis. The own algorithm of circuit division is introduced. The results of both optimizing methods are compared by a large set of experiments. Time improving brought by division is also evaluated.

Keywords circuit window, division by iterations, parallel resynthesis, circuit quality, processing speed

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Optimalizace logického obvodu představuje proces, ve kterém se různými postupy dosahuje minimalizace jeho zvolených vlastností. Nejčastější motivací je snížit počet hradel nebo signálů v obvodu, nebo převést jeho strukturu do požadovaného tvaru (mapování na technologii). Pro mnohé algoritmy optimalizace platí, že provádějí tzv. resyntézu obvodu, vcelku nebo v určitých jeho částech. Explicitně vyjádřit, které části obvodu resyntetizovat, nelze do té doby, dokud není obvod předem na ně rozdělen. Myšlenka, že obvod rozdělený na části má potenciál optimalizovat se lépe, než za použití stejného algoritmu vcelku, tvoří základ pro tuto práci. Zároveň se zde otvírá cesta k paralelizaci zpracování jednotlivých částí, protože ty mohou být disjunktní, tedy na sobě nezávislé. Je také možné tímto vyzkoušet na stejné části více optimalizačních algoritmů.

Jak se daný obvod bude rozdělovat konkrétně, to je hlavním předmětem této práce. Existují měřítka, kterých pro jednotlivé jeho části docílit, aby dávala co největší prostor na zlepšení vlastností. Podstatný je také čas, který se při resyntéze zjednodušené struktury může pohybovat v nižších řádech.

Kapitola 1

Teorie

1.1 Definice pojmů

V rozsahu této práce používám pojmy:

- **Netlist** je orientovaný graf popisující logický obvod, sestávající z uzlů (hradel) a orientovaných hran (signálů). Budeme uvažovat pouze kombinační obvody, v příslušném grafu tedy neexistují orientované cykly.
- **Počet h-vstupů** v obvodu je zde zkratkou pro souhrnný počet vstupů u všech jeho hradel, bez ohledu na propojení signály. Měřítko, které oproti prostému počtu hradel více popisuje míru složitosti obvodu.
- **Počet úrovní** obvodu. Mějme množinu všech cest mezi libovolným primárním vstupem a výstupem. Každou z těchto cest ohodnotíme číslem, které vyjadřuje počet hradel na cestě. Počet úrovní obvodu je maximum z těchto ohodnocení.
- **Okno** v obvodu je podmnožina jeho hradel a všech signálů s nimi sousedících. Toto označení je převzato z [1]. Vstupy, resp. výstupy okna jsou takové jeho signály, které sousedí s hradlem nepatřícím do okna, nebo jsou primárními vstupy/výstupy obvodu.
- **Fragmentace obvodu**. Pokud netlist převedeme na neorientovaný graf, označuje fragmentace obvodu počet komponent tohoto grafu.
- **Resyntéza obvodu** je zde chápána jako atomický proces, výsledek aplikace tzv. resyntézní metody na daný obvod. Resyntetizuje se za účelem snížení složitosti obvodu (počet hradel, h-vstupů), nebo jiných jeho charakteristik (počet úrovní).
- **Optimalizace obvodu** je v této práci proces složený z jednotlivých resyntéz spolu s vyhodnocováním jejich výstupů. Z výstupů se poté vybírá ten s nejlepšími danými vlastnostmi.

1.2 Algoritmus

Cílem navrženého algoritmu je optimalizovat obvod po částech (oknech). Vstupem algoritmu je jeden obvod, výstupem funkčně ekvivalentní optimalizovaný obvod. Top-level vypadá následovně:

1. Začíná se **rozdělením obvodu na okna**, což je navrženo sekvenčně.
2. Následuje **paralelní resyntéza** těchto oken, zvolenými metodami, z nichž se vybírají nejlepší výsledky podle nastavených kritérií.
3. Závěrečná část je sekvenční, kde se tyto výsledky opět spojí do jednoho obvodu.

Optimalizace obvodu se docílí optimalizací oken, na která byl rozdělen. Schéma algoritmu je na obrázku 1.1.

1.2.1 Výběr oken

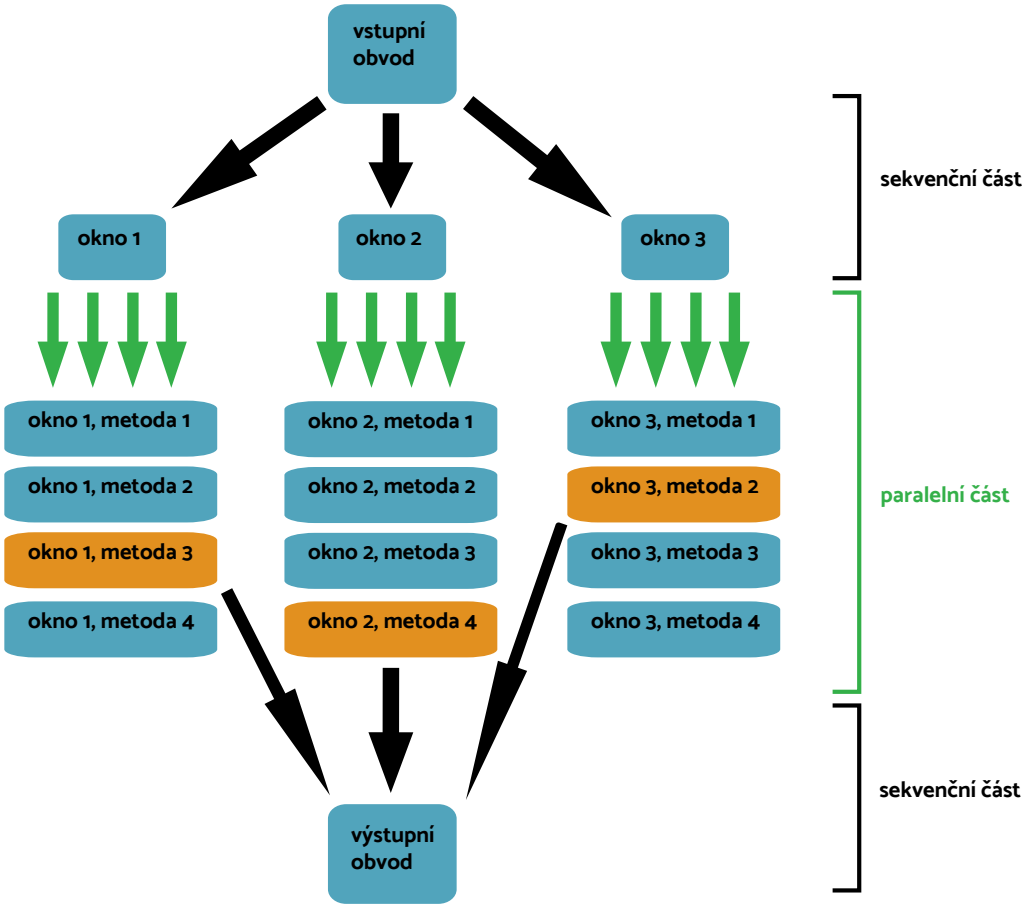
Atomickou částí obvodu je **hradlo**. Rozdělení obvodu na stanovený počet oken znamená přiřadit každé hradlo k nějakému oknu. Okna jsou si navzájem disjunktní.

Podle [1], obsahujícím různé metody pro výběr jednoho okna, je pro jeho resyntézní potenciál podstatné:

- Nezanedbatelný počet hradel vůči zbytku obvodu (zde ostatním oknům). Z důvodu, aby resyntéza mohla přinést nějaké významné zlepšení.
- Minimalizace počtu vstupů/výstupů okna vůči okolí. Důležitým důsledkem je, že okno se vybere tak, aby nebylo fragmentované. Přidáním nesousední části do okna nemůžeme docílit minimalizace.

Problémem, který je třeba v průběhu výběru stále řešit, je udržení těchto vlastností u oken - pokud vycházíme z předpokladu, že budou vznikat iterativní cestou.

Základní myšlenkou mého algoritmu je zde postupovat „odzdola“ - tj. zprvu rozdělit obvod na maximální možný počet oken, což je počet hradel v obvodu. Každé hradlo tak bude umístěno ve vlastním okně. Toto rozdělení má tu triviální vlastnost, že je jediné možné a tudíž nelze nalézt rozdělení s lepšími vlastnostmi oken. Za hypotetické situace, kdy by byl stanovený počet oken roven tomuto maximu, je toto řešením. Pokud by bylo požadováno o jedno okno méně, k řešení bude potřeba vykonat iterační krok: **spojit dvě sousední okna do jednoho**. Možností je tolik, kolik je sousedních dvojic. Vybereme tu, kde nově vzniknuvší okno bude mít ze všech dvojic nejlepší vlastnosti, tj. minimální počet vstupů, výstupů a hradel ve stanoveném poměru. Takto



Obrázek 1.1: Základní schéma algoritmu.

se dají realizovat ty metody výběru okna, které pracují s uvedenými třemi vlastnostmi.

Uvedeným postupem vždy můžeme snížit aktuální počet oken o jedno, a libovolně iterovat až k hranici fragmentace obvodu. Lze tak dosáhnout rozdělení pro jakýkoli stanovený počet oken (od fragmentace po počet hradel). U oken jsou zachovány požadované vlastnosti.

1.2.2 Paralelní část

Protože jsou okna disjunktní, lze provádět jejich resyntézy nezávisle na sobě, tedy i **paralelizovat**. Pokud stanovíme pořadí jednotlivých prací, vybere si procesor, který právě skončil, za další práci tu aktuální v pořadí. Práce se tedy přiděluje dynamicky. Je také potřeba **synchronizace** přístupu k těmto

sdíleným informacím.

Jakmile jsou hotovy všechny resyntézní metody pro dané okno, vznikne úkol vybrat z výsledků nejlepší obvod dle zadaných kritérií (minimum počtu hradel, h-vstupů, úrovní). Ten se pak použije jako výstupní okno při spojování obvodu.

1.3 Cíle práce

- Rozdělování obvodu by mělo být **rychlé**, netvořit časově nejvíce náročnou část výpočtu.
- Vyřešit synchronizaci paralelní části na bázi MPI.
- Univerzálnost v použití resyntézních metod. Umožnit jejich uživatelské **externí volání** a snadnou editaci.

Kapitola 2

Framework k výpočtu

V této kapitole představím prvky, na kterých je postavena efektivita implementace.

Definice užívaných pojmů:

- **Seznamem** prvků je v této práci označováno rozšiřitelné pole prvků stejného typu. Indexem prvku je jeho aktuální pozice v tomto poli. V jazyce `C++` tuto funkcionalitu pokrývá datový typ `vector`.
- **T-seznam** je zde seznam s přidruženou trií pro vyhledávání prvků v konstantním čase.
- **Hlavní proces** je proces s MPI rankem 0 [2].

Pro potřebu dosažení cílů práce jsem realizoval následující:

1. **Upravená binární halda** je použita pro rychlý iterační krok při rozdělování obvodu (sekce 2.1).
2. **T-seznam** je použit obecně pro rychlou práci s obvodem, načítání i rozdělování (sekce 2.2).
3. **Mutex nad MPI-2** slouží k synchronizaci přidělování práce u paralelní části algoritmu (sekce 2.3).

2.1 Upravená binární halda

Uvažujme binární haldu vybavenou operacemi `min`, `insert`, `delete` a `make` [3]. Prvky haldy jsou zde ukazatele na objekty s binární operací `isLower`, tedy které je možné mezi sebou porovnat a rozhodnout, který je menší. To se realizuje pomocí přímého porovnání proměnné `eval` v objektech. V případě rovnosti rozhodne náhodné číslo.

Na rozdíl od standardní operace `delete` s indexem mazaného prvku je zde za parametr přímo prvek, který má být z haldy smazán. Je tomu tak z potřeby realizovat tuto operaci zvenčí u kteréhokoliv objektu, když známe pouze hodnotu ukazatele. Aktuální index, kde se prvek v haldě nachází, je zapisován přímo do příslušného objektu. Halda se ve všech svých operacích stará o jeho aktuálnost, beze změny v časových složitostech. Smazání kteréhokoliv prvku z haldy má tak složitost $O(log(n))$, kde n je počet prvků haldy.

Pro využití funkce make má halda proměnnou `valid`. Pokud je `true`, má halda platnou strukturu. Lze ji kdykoli nastavit na `false`, což umožňuje poté vkládat nové prvky v konstantním čase. Obnovuje se automatickým voláním `make`, když vznikne požadavek na správnost haldy u ostatních operací.

2.2 T-seznam

Uvažujme datovou strukturu zvanou `trie`. Oproštěno od konkrétní implementace, jedná se o n-ární strom, který uchovává klíče coby řetězce znaků nad n-ární abecedou. Pokud budeme klíče interpretovat jako posloupnosti bitů, dostáváme $n = 2$ a definici pro **binární trie** [4]. Délka klíče je implementačně omezena pouze na celé byty.

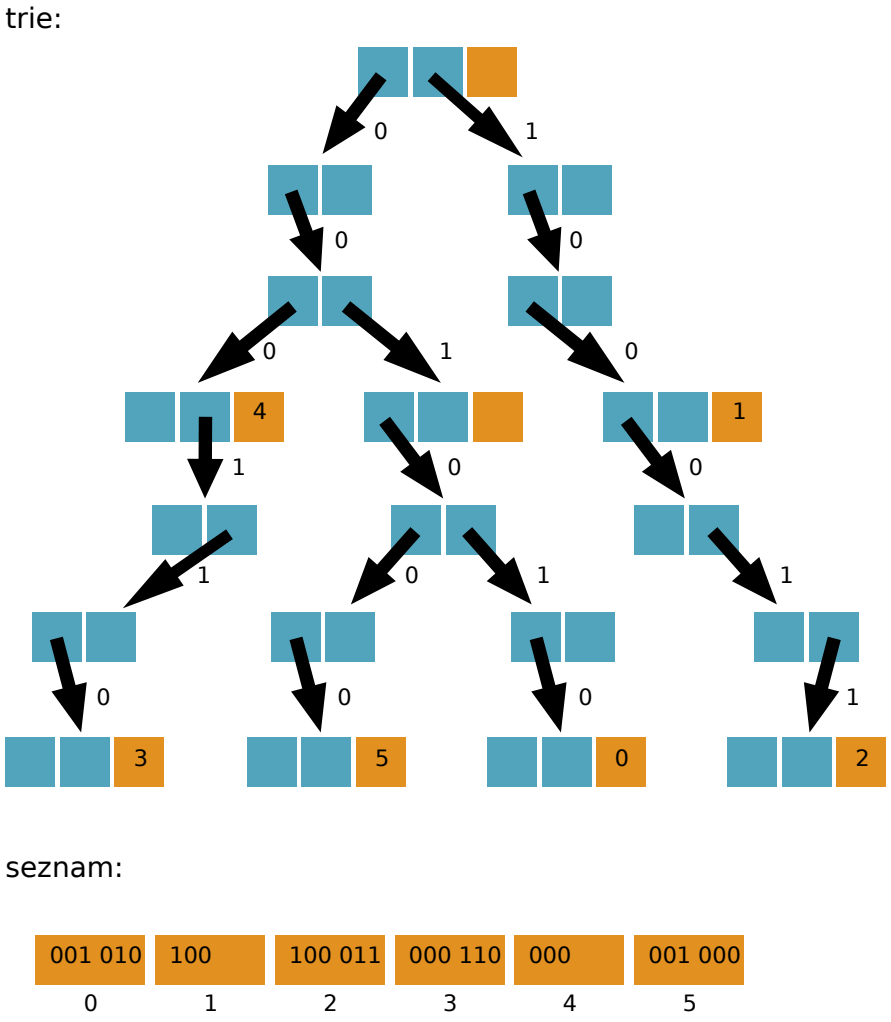
Dále definujme **T-seznam** jako seznam, který má k sobě přidruženou binární trie, jejíž klíče jsou výsledkem nějaké bezkolizní (obvykle triviální) hashovací funkce jeho prvků. T-seznam neuchovává duplicitní prvky, stejně jako binární trie implicitně neobsahuje tytéž klíče. Prvky zde považujeme za shodné, pokud se rovnají jejich klíče.

2.2.1 Implementace a význam

Seznam prvků je v T-seznamu uložen standardní cestou. Trie je zde implementována jako seznam indexů - číselných odkazů do sebe sama, které tak tvoří strukturu jejího stromu. Každý vnitřní uzel je tvořen dvěma po sobě jdoucími indexy pro aktuální bit klíče - pro hodnoty `0` a `1`. Pokud je index platný, odkazuje v seznamu na další uzel stromu, jinak se jedná o slepou větev. Listy a vnitřní uzly, které reprezentují konec příslušného klíče, obsahují navíc i položku pro aktuální index (pozici) jeho prvku v seznamu. Celý T-seznam tak může sloužit jako **bezkolizní dynamická hashovací tabulka**, s hashi vždy v rozsahu $0..(n-1)$, kde n je aktuální počet prvků.

Hlavní výhodou, pro kterou byla tato struktura navržena, je **rychlost základních operací**. Vkládání, mazání, hledání hashe k prvku, to vše má složitost $\Theta(m)$, kde m je délka klíče prvku. Z hlediska počtu prvků jsou tedy všechny tyto operace **konstantní**.

Příklad T-seznamu je na obrázku 2.1.



Obrázek 2.1: T-seznam v příkladu pro binární řetězce.

2.2.2 Hledání indexu prvku a konstrukce trie

Jako vstup máme referenci na prvek. Podle typu z něj získáme klíč v podobě posloupnosti bytů. Tuto posloupnost procházíme po bitech, v rámci bytu pak big-endian. Pohybujeme se po trii od kořene, směr každého kroku je adresován příslušným bitem. Narazíme-li na slepou větev, prvek v seznamu není přítomen. Pokud se dostaneme až na koncový uzel, přečteme z něho index prvku. V případě neplatného indexu také není prvek přítomen, byl smazán.

Velmi podobně se postupuje i v případě vkládání nového klíče při konstrukci trie. Ta se děje přidáváním nových uzlů ke slepé větvi, podle hodnot zbytku bitů v klíči. Do koncového uzlu zapíšeme index vkládaného prvku.

2.2.3 Vkládání prvku

Nejdříve je třeba zkotrolovat, zda se prvek se stejným klíčem v seznamu již nenachází. To zjistíme, pokud pro jeho klíč nalezneme v trii platný index. V případě, že klíč je v trii přítomen, prvek se nepřidá a i trie zůstane beze změny. Jinak se prvek přidá na konec seznamu, a tento index se zapíše do trie.

2.2.4 Mazání prvku

Pro daný prvek si zjistíme jeho index v seznamu. Pokud se v něm nachází, nahradíme ho prvkem z konce seznamu a seznam zkrátíme o jednu položku. Pro oba prvky musíme poté aktualizovat indexy v trii - smazanému zapsat neplatný, přesunutému jeho nový.

Některé uzly v trii se po těchto operacích stanou nepotřebnými, ale velikost trie zůstává stejná.

2.2.5 Průsečík

Vzhledem k absenci duplicit se na dva T-seznamy můžeme dívat také jako na množiny svých prvků, a jejich průsečík definovat jako standardní množinovou operaci.

Mějme prázdný T-seznam jako výstup. Ve výsledku bude obsahovat prvky nacházející se v obou T-seznamech. Toho lze docílit dvěma vzájemně analogickými způsoby: procházet jeden z operandů a testovat přítomnost jeho prvků ve druhém, nebo naopak. Zde můžeme využít toho, že časová složitost procházení závisí na počtu prvků, ale u hledání prvku tomu už tak není (sekce 2.2.1). Pro procházení tedy zvolíme ten kratší ze seznamů, kvůli rychlejšímu provedení.

2.2.6 Specializace pro různé typy prvků

Obecné pravidlo pro odvození klíče je následující: jako klíč se uvažuje celá hodnota prvku, tj. všechny jeho byty, čtené little-endian. Pro daný typ má

klíč pevnou velikost, proto jsou položky pro index umístěné jen v listech trie, všechny ve stejné hloubce. Tento model se konkrétně využívá u celých čísel a ukazatelů. Zejména pro posloupnosti celých čísel má význam čtení littleendian (tak, jak jsou také obvykle uložena v paměti), v trii se vytváří dlouhý společný kořen a čísla jsou zapisována s úsporností.

U řetězců je klíčem celý řetězec, čtený popředu. Ze stejného důvodu jako v předchozím případě je to úsporné např. pro číslované názvy signálů. Struktura trie je vzhledem k nestejným velikostem klíčů odlišná - položky pro index se nacházejí v každém uzlu hloubky dělitelné b, kde b je počtem bitů na byte/znak na dané architektuře (kořen má hloubku 0). Tímto je zajištěno, aby se do trie v jakémkoli stavu dal zapsat libovolný další řetězec.

2.2.7 Velikost trie a rezervace místa

Pokud budeme pracovat s T-seznamem s předem známým počtem prvků, můžeme kromě rezervace místa na seznam i rezervovat místo pro trie. Protože obojí se v průběhu vkládání při nedostatku místa realokuje, v rámci co nejlepšího výkonu se tomuto můžeme vyhnout, při znalosti rozumné horní meze paměťové složitosti trie.

Určení takové meze závisí jak na typu i hodnotách vkládaných prvků, tak odvozeně i na specializaci struktury trie. Pro konkrétní potřeby programu byly experimentálně zjištěny (soubor `trie_coliru.hpp`) koeficienty pro dva typy - vzestupná celá čísla a totéž ve formě řetězce.

Pro množinu celých čísel, která je tvořena malým (vzhledem k celkovému počtu čísel) počtem aritmetických posloupností s diferencí 1, platí následující: poměr počtu položek trie a prvků v T-seznamu konverguje k hodnotě `3,0`.

Pro řetězce ve tvaru obvyklém pro označování signálů v obvodu (písmenný prefix a číslo) je tentýž koeficient `5,8`. Platí tomu tak za stejných podmínek jako pro celá čísla výše, jen zde jsou aritmetické posloupnosti explicitně odděleny prefixy. Vzhledem k dopřednému směru ukládání řetězce v trii nemá tvar ani délka těchto prefixů na stanovený koeficient vliv.

2.3 Mutex nad MPI-2

Úkolem mutexu je vytvořit tzv. **kritickou sekci** - část programu, kterou může v daném okamžiku vykonávat nejvýše jeden proces. Její začátek a konec je vymezen pomocí dvou funkcí - zamčení (`lock`) a odemčení (`unlock`). Proces, který do sekce vstoupil, musí také sekci korektně opustit (zavolat `unlock`). Mutex je zde tvořen samostatnou třídou, která tyto funkce sdružuje.

Knihovna MPI-2 nezajišťuje potřebnou atomicitu sdružených operací, která by vedla k jednoduchému řešení mutexu (např. funkce `MPI_Get_accumulate`). Funkcionalita prezentovaného algoritmu tedy využívá **sdílenou paměť** (RMA) [5] a **posílání zpráv**.

Sdílená paměť je umístěna v paměti procesu s nejnižším rankem (0). Pro práci s ní je užito funkcí `MPI_Get` a `MPI_Put`. K eliminaci *data races* se při tom dodržují tato obecná pravidla:

1. Sdílená paměť je členěna na buňky, jejichž velikost je v takových jednotkách (bytech), které na dané architektuře umožňují provádění operací čtení a zápis pro každou buňku samostatně a atomicky.

2. Nejsou používány jiné operace než čtení nebo zápis. Jejich kombinace nelze používat jako atomické operace.

3. Každá hromadná (neatomická) operace čtení nebo zápisu více buněk musí být v algoritmu ekvivalentem jejich postupného zpracování v libovolném pořadí.

4. Pokud do buňky zapisuje více procesů, je algoritmem zaručeno, že je zapisována stejná hodnota.

5. Daný proces provádějící operaci vždy vyčká jejího dokončení.

Konkrétní struktura paměti zahrnuje $n+1$ buněk, kde n je počet procesů. Buňka je tvořena jedním bytem, ale používají se jen hodnoty `0` a `1`, je tedy používána jako bit. Každý proces má dle ranku přidělenu jednu buňku, kam má výhradní právo zápisu. Hodnota v ní určuje, zda mutex proces zrovna používá, tj. vykonává některou z funkcí `lock`, `unlock`, nebo je v kritické sekci. Poslední buňka s indexem n je sdílená a značí, zda je mutex nějakým procesem používán.

K posílání zpráv se využívají funkce `MPI_Isend` a `MPI_Recv`. Jde tedy o model neblokujícího posílání a blokujícího příjmu. Obsahem posílaných zpráv je vždy jen jedna číselná hodnota. Ta určuje druh zprávy, její sekundární tag. Těch je celkem 6: `PASSIVE`, `ACTIVE`, `PROBE`, `WINNER`, `TICKET`, `POKE`. Jejich použití je popsáno v dalších částech kapitoly.

2.3.1 Inicializace mutexu

Inicializace probíhá v konstruktoru. Předá se komunikační tag pro posílání zpráv, který plní důležitou funkci filtru. Procesy reagují pouze na zprávy vytvořené v rámci činnosti mutexu, a komunikace je tak odstíněna od té vnější a nehrozí dezinterpretace zpráv.

Dále všechny procesy inicializují RMA. Proces, u něhož má být založena sdílená paměť, ji tak založí a hromadným zápisem inicializuje buňky na `0`. Všechny ostatní procesy počkají na dokončení této operace na bariéře.

2.3.2 Zamykání

Každý proces, který zavolá funkci `lock`, si ze všeho nejdříve zaregistruje svou přítomnost v mutexu - do své buňky zapíše `1`. Účelem je jistota pro ostatní procesy, že **není zaneprázdněný jinde** a dočkají se od něj v průběhu algoritmu komunikační odezvy, pokud mu pošlou zprávu. Žádný proces nikdy nepošle zprávu nepřítomnému procesu, v jehož buňce si přečte `0`. Znalost aktuální hodnoty `1` v buňce adresáta předchází posílání každé zprávy.

Poté si přečte hodnotu ze sdílené buňky. Hodnota `0` znamená, že je v mutexu volno a proces může pokračovat dále. Pokud ne, stává se tzv. **pasivním** a zůstává čekat na zprávu, která může být typu `POKE`, `PROBE`, `WINNER` nebo `TICKET`. Na zprávu `POKE` nikterak nereaguje. Na `PROBE` nebo `WINNER` pošle volajícímu procesu odpověď `PASSIVE`. Pokud obdrží `TICKET`, opouští funkci `lock` a ihned se dostává do kritické sekce. Po vyřízení ostatních druhů zpráv opět kontroluje sdílenou buňku jako na začátku.

Hned po úspěšném čtení sdílené buňky do ní proces zapíše `1` a stane se tzv. **aktivním**. Mezi čtením `0` a zápisem `1` mohou tak učinit i některé ostatní procesy, takže se jich v tomto místě algoritmu může sejít i více aktivních. Jistotou je, že po prvním dokončení tohoto zápisu je množina aktivních **již uzavřena**, další příchozí se po přečtení `1` stanou pasivními. Aktivní procesy tedy od místa provedení zápisu pokračují v pevném počtu.

Každý aktivní proces si dále hromadně přečte buňky ostatních procesů. Jeho lokální kopie tak obsahuje hodnoty `1` v buňkách všech aktivních procesů. Obsahovat totéž může i u některých pasivních, což ovšem nemá na strukturu výpočtu vliv.

Principem zbytku výpočtu je rozhodnout, který z aktivních procesů bude pokračovat do kritické sekce (vítěz) a tuto informaci mezi ně **distribuovat**. První z úkolů se dá řešit triviálně - bude to ten s nejnižším rankem. Problém distribuce informace, zda pokračovat, je komplexnější a zahrnuje několik fází.

Proces posílá postupně „doleva“ (od svého ranku směrem k nižším, co mají v buňce `1`) zprávu `PROBE` a čeká na odpověď. Pokud od někoho obdrží `ACTIVE`, ví, že není vítězem, a po vyřízení zbytku komunikace se stane pasivním.

Předtím, než se prohravší stane pasivním, musí ještě počkat na `PROBE`, kterou mu pošle aktivní soused zprava nebo vítěz. V odpověď generuje zprávu `ACTIVE`. Tím odblokuje řetěz závislosti o jeden článek směrem doprava, protože všechny aktivní procesy vpravo stále čekají odpověď na svou sondu.

Pokud proces nenalezne nalevo od sebe aktivního souseda, stává se vítězem. Nestačí pouze se dozvědět tuto skutečnost, ale také postarat o ukončení komunikace ostatních procesů, pokud jsou přítomny.

Jednodušší situace nastává, pokud již během předchozího výpočtu vítěz obdrží `PROBE`. Znamená to přítomnost jeho pravého souseda, kterému tak zašle zprávu `WINNER`. Ta plní stejnou funkci jako `ACTIVE`. V opačném případě mu zpráva `PROBE` může přijít později, ale také nemusí, pokud je sám. Proto musí aktivně začít prohledávat prostor postupně napravo od sebe, každému s `1`

zaslat `WINNER` a počkat na odpověď. V případě pravého aktivního souseda je zaslaný `WINNER` jím interpretován jako odpověď na jeho sondu.

Že je sám, zjistí vítěz v předchozí fázi. Pokud ano, může přejít do kritické sekce. V opačném případě musí ještě najít poslední aktivní proces vpravo, který je stále blokován čekáním na `PROBE` od svého neexistujícího pravého souseda. Proto mu ji musí poslat vítěz sám. Učiní to analogickým způsobem jako v předchozí fázi - postupným zasíláním `PROBE` od nejvyššího ranku směrem doleva. Vyčká na zprávu `ACTIVE`, která je zároveň i poslední zaslanou zprávou. Pak teprve může funkci `lock` opustit, až když je **veškerá komunikace vyřízena**.

Schéma zamykání je na obrázku 2.2.

2.3.3 Odemykání

Funkce `unlock` je výrazně jednodušší, hlavně z důvodu, že je prováděna jediným procesem po kritické sekci, **sekvenčně**. Obdobně jako u zamykání, i zde je první činností procesu zapsání `0` do své buňky. Ostatní průběh se odvíjí od přítomnosti dalších procesů v mutexu, pasivních a čekajících na zprávu.

Proces si přečte stav všech buněk. Pokud je někde hodnota `1`, jediné, co zbývá udělat, je poslat příslušnému procesu `TICKET`. Tím se obejde celý složitý mechanismus zamykání, který vybraný proces spolu s ostatními nemusí absolvovat a vystřídá odemykající proces v kritické sekci.

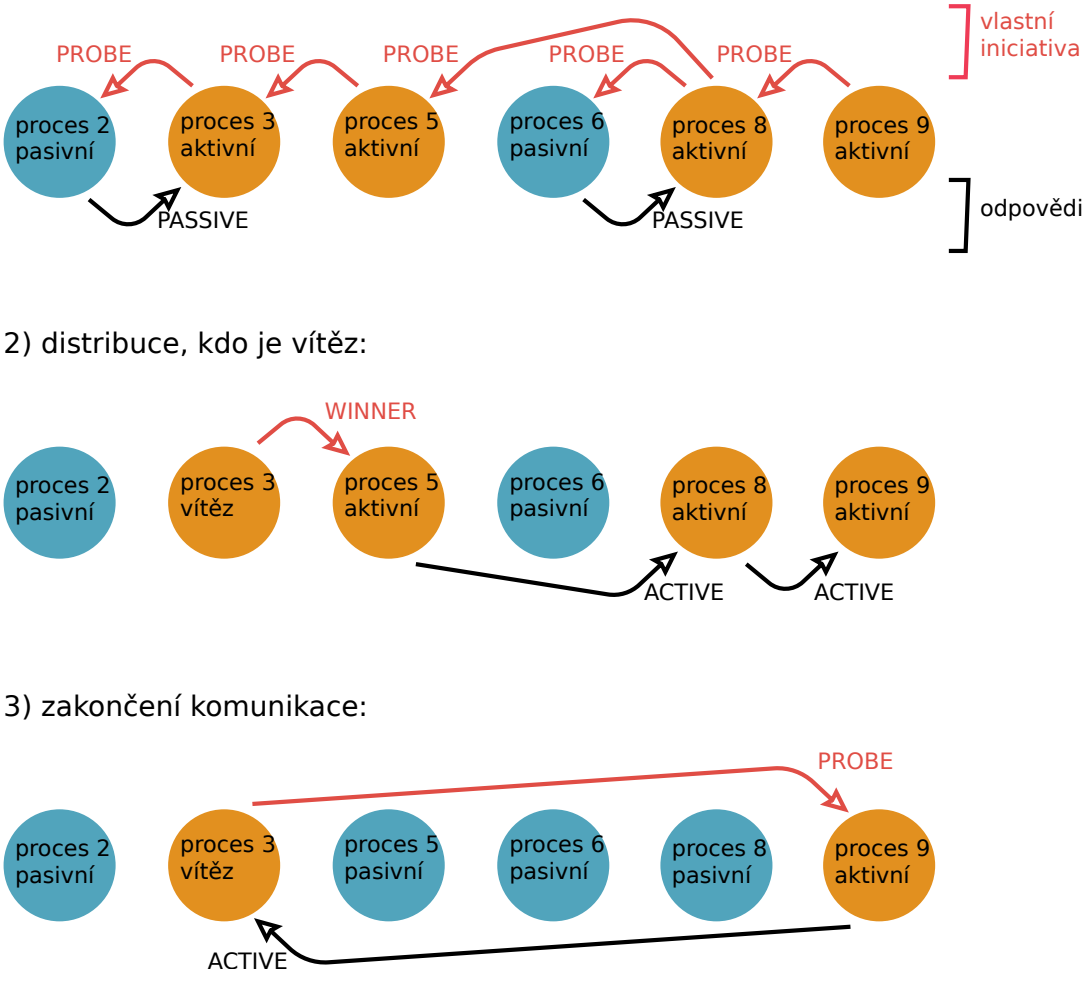
V případě, že jsou všude hodnoty `0`, přistoupí proces k samotnému odemčení - zapíše `0` do sdílené buňky. Skončit odemykání zde by ovšem mohlo vést k **neošetřenému deadlocku**. Mezi přečtením prázdných buněk a fyzickým zápisem `0` do sdílené buňky si ji mohou ještě jako obsazenou přečíst právě příchozí procesy, které na její následné uvolnění už nikterak nezareagují. Čekají na zprávu a jsou tak závislými na dalších příchozích. Pokud by byli ve výpočtu poslední, zůstanou v deadlocku.

K ošetření výše zmíněného slouží zpráva `POKE`. Po uvolnění sdílené buňky si proces ještě jednou přečte stav všech buněk. Pokud nějaký další proces dorazil v inkriminovanou dobu, bude tam mít hodnotu `1`. Po přijatém `POKE` si tento proces přečte již uvolněnou sdílenou buňku, což mu umožní stát se aktivním.

2.3.4 Detaily ke korektnosti komunikace

Užitý komunikační model není triviální a zprávy procesům mohou dorazit v každém pořadí, které nevylučuje jejich vzájemná závislost. Je třeba umět reagovat na všechny tyto možné situace, aby funkce mutexu nebyla narušena (nečekaná zpráva, deadlock, více procesů v kritické sekci, ...). Každý proces si proto uchovává u každého z typů zpráv `ACTIVE`, `WINNER` a `PROBE` rank odesílatele, aby uměl rozlišit zprávy, které mu přišly bez jeho iniciativy.

Ve fázi určování vítěze se může stát, že pravý soused obdrží od vítěze zprávu `WINNER` dříve, než mu stihne poslat `PROBE`. Proto porovnává zjištěný rank vítěze s procesem, od nějž právě očekává odpověď. Pokud se ranky



Obrázek 2.2: Schéma zamykání mutexu.

neshodují, pošle samostatně vítězi `PROBE` a končí s prohledáváním. V opačném případě končí rovnou, jako by obdržel `ACTIVE`.

Výše uvedená komunikace výměny zpráv mezi vítězem a sousedem je symetrická, nemá pevného iniciátora. Obdobně postupovat musí tedy i vítěz. Pokud obdrží `PROBE`, musí si podle ranku zkontrolovat, zda na ni odpovídat, nebo zda již zprávu ve smyslu odpovědi poslal.

Zpracování zprávy `POKE` se liší podle toho, zda je proces aktivní. U pasivního zpráva splní svůj účel, tj. probudí proces k další kontrole sdílené buňky. Pokud zprávu proces obdrží jako aktivní, vyfiltruje ji a bezprostředně začne čekat na další.

2.4 Generátor pseudonáhodných čísel

Knihovní funkce MPI používají ve své implementaci standardní `C` funkci `rand`, přičemž počet těchto volání se mění v závislosti na komunikační situaci [6]. Tento fakt je překážkou v možnosti zajistit programu **determinističnost**, opakovatelnost. Proto má program vlastní generátor, který pracuje na základě modulární aritmetiky. Pomocí funkce `seed` se dá nastavit pevnými hodnotami (deterministicky), nebo s využitím systémového času (nedeterministicky).

Jádro generátoru je převzato z [7].

Kapitola 3

Implementace

Byl použit jazyk `C++` a funkcionalita knihovny `MPI-2`. MPI je tu zde pro snadnou komunikaci mezi procesy. Z hlediska paralelního výpočetního modelu také všechny používají i sdílené úložiště souborů.

3.1 Popis formátu BLIF

Pracuje se s tímto souborovým formátem. BLIF je formát pro zápis logických obvodů vyvinutý na univerzitě v Berkeley [8]. Všechny obvody i výsledky resyntézy, které jsem v práci použil, byly zapsány pouze v základní podobě tohoto formátu, kterou teď popíši:

- V souboru je na začátku uvedeno klíčové slovo `.model` s názvem obvodu.
- Poté následuje řádek uvedený `.inputs` s názvy primárních vstupů oddělených mezerami, analogicky pak řádek `.outputs`.
- Hradlo je uvedeno `.names`, po němž následují názvy signálů představujících jeho vstupy, poslední z nich je pak jeho výstup. Hradlo je vždy jednovýstupové. Další řádky popisují chování hradla, a to až do následujícího klíčového slova.
- Na konci je klíčové slovo `.end`.

Příklad je uveden v tabulce 3.1.

3.2 Top-level algoritmus

Z globálního pohledu vzato, celý algoritmus je orientován úzce **souborově**. Vše je ukládáno na sdíleném úložišti, všechny vytvořené pracovní soubory představují platné BLIF obvody.

Algoritmus je následující:

```
.model myExample1
.inputs x1 x2 x3
.outputs y1 y2 y3
.names x2 n1
0 1
.names x3 n1 n2
11 1
.names n1 n3
0 1
.names x1 n3 n5 n4
010 1
101 1
.names n3 n5
0 1
.names n4 n6
1 1
.names n5 n6 n10 n7
0-1 1
10- 1
.names n2 n8
1 1
.names n8 n9
0 1
.names n8 n9 n10
00 1
.names n8 n9 n11
10 1
.names n6 y1
1 1
.names n7 y2
1 1
.names n11 y3
1 1
.end
```

Tabulka 3.1: Příklad formátu BLIF.

1. Všechny procesy si načtou parametry z příkazové řádky.

2. Procesy si hromadně načtou soubor s resyntézními metodami.

3. Hlavní proces načte obvod ze vstupního souboru, vyhodnotí ho a rozdělí. Každé okno pak uloží do vlastního souboru. Z každého tohoto souboru pak udělá tolik kopií, kolik je resyntézních metod. Je to proto, že metody soubory modifikují, a také k předejití možným kolizím mezi procesy při hromadném načítání stejného souboru. Pro pojmenovávání používá v obou případech číslované přípony. Ostatní procesy zatím čekají na bariéře.

4. V rámci optimalizační části hlavní proces vyrobí **plánovač** jednotlivých dílů práce, který si ostatní procesy od něj načtou pomocí RMA [5]. Práce je dvojího druhu: resyntéza konkrétního okna konkrétní metodou, a vyhodnocení celého okna. Vyhodnocení je podmíněno dělat až v okamžiku, kdy je okno resyntetizováno již všemi metodami, a spočívá v přejmenování souboru s nejlepším obvodem. Plánovač v sobě uchovává tyto informace, na základě kterých práce přiděluje. Přístup k němu je řízen pomocí **mutexu**, který kolektivně založily i používají všechny procesy. Přístup probíhá tak, že daný proces zamkne mutex, načte plánovač, odevzdá práci, přijme novou, uloží plánovač a odemkne mutex. Do okamžiku, kdy mají všechny procesy hotovo, se čeká na bariéře.

5. Hlavní proces provede spojení obvodu z přejmenovaných oken, a posléze smaže pracovní soubory. Výsledek pak ještě načte za účelem vyhodnocení.

3.3 Načítání obvodu

Pro práci s obvodem je v programu samostatná třída. Obsahuje:

- T-seznam všech názvů signálů, včetně primárních vstupů a výstupů. Názvy jsou fyzicky uloženy pouze zde, všude dále se už používají jen jejich hashe (indexy v tomto seznamu).
- T-seznamy hashů primárních vstupů a výstupů.
- Seznam ukazatelů na vytvořená okna obvodu, počet těchto oken.
- Vypočtená měřítka kvality obvodu: počet hradel, h-vstupů, úrovní obvodu.

Obvod se načte z BLIF souboru a zároveň se vyhodnotí (vypočtou se měřítka kvality).

3.3.1 Základní struktury

Každé okno má své id, které odpovídá jeho indexu v seznamu oken. Dále obsahuje své vstupy, výstupy, ukazatele na hrany sousednosti s ostatními okny (vše T-seznamy) a hradla (spojový seznam).

Hrana sousednosti je složena ze dvou ukazatelů na své uzly (okna). Jedná se zároveň o potomka třídy, s jejímiž objekty se dá pracovat v haldě (sekce 2.1). Obsahuje tedy i proměnnou `eval`, která zde představuje vyhodnocení příslušné dvojice oken při rozdělování obvodu.

Hradlo obsahuje seznam řetězců v takové podobě, jak je zapsáno na příslušných řádcích v souboru. Podstatná je první řádka obsahující jména signálů, s dalšími řádky popisujícími chování **se nepracuje**, jsou uloženy pouze pro pozdější zapsání. Pro realizaci spojového seznamu je také přítomen ukazatel na další hradlo.

3.3.2 Načtení netlistu

Plnohodnotné načtení má 6 fází. Pouze v první se čte ze souboru, v dalších se vytvářejí metadata.

V první fázi se ze souboru načtou řádky se jménem modelu, primárními vstupy a výstupy. Načtou se jednotlivá hradla, ke každému se vytvoří nové okno, do něhož se začlení. Oknu se přidělí id, podle pozice v seznamu. Ekvivalence hradla a okna platí až do doby, než se okna začnou spojovat dohromady.

Protože známe počty primárních vstupů/výstupů, zarezervujeme pomocí odhadu ze sekce 2.2.7 místo pro jejich T-seznamy. Stejně tak celkový počet signálů bude $h + i$, kde h je (opět známý) počet hradel a i počet primárních vstupů. Jiné signály se v obvodu nevyskytují. Podle tohoto čísla tedy zarezervujeme místo pro T-seznam signálů.

Ve druhé fázi se procházejí hradla, parsují se jejich výstupy a přidávají do T-seznamu signálů. Hashem příslušného názvu je vždy jeho aktuálně koncový index (sekce 2.2.3). Tento hash se přidá do T-seznamu výstupů okna, jehož je hradlo součástí. Kontrolují se duplicity.

Ve třetí fázi se procházejí naparsované primární výstupy obvodu. Kontroluje se, zda je daný signál přítomen ve stávajícím T-seznamu signálů, ve kterém jsou zatím výstupy všech hradel. Signál, který není řízen žádným hradlem, je ignorován. Vytváří se T-seznam primárních výstupů.

Čtvrtá fáze vytvoří analogickým způsobem T-seznam primárních vstupů. Signál se zapíše do T-seznamu signálů. Pokud jde o signál nový, zapíše se jako hash i do primárních vstupů. Tento hash se kontroluje na hodnotu, která nesmí být nižší než počet hradel, protože jinak by se jednalo o výstup nějakého hradla, a tudíž vadnou strukturu obvodu.

Po čtvrté fázi je T-seznam signálů již kompletní, žádný jiný se nesmí kdekoli v obvodu nacházet. Byl postupně rozšiřován přesně v tom pořadí, aby byla splněna důležitá vlastnost: hashe z rozsahu $0..(h-1)$, kde h je počet

hradel, jsou zároveň i indexy do seznamu k příslušným oknům (sekce 3.3). Toho je využíváno k přímému přístupu a pohybu po netlistu. Zbytek signálů jsou primární vstupy obvodu.

V páté fázi se znovu procházejí hradla, parsují se jejich vstupy a přidávají do T-seznamu vstupů příslušného okna. Nyní již jsou všechny T-seznamy oken i obvodu kompletní.

Šestá fáze se stará o vytvoření **hran sousednosti**. K tomu stačí projít okna a z každého jejich neprimárního vstupu hranu vytvořit. Protože se hrana vytváří na základě jediného signálu, zohledňuje se i její orientace - pořadí zapsání indexů oken v hraně, na první pozici zdroj, na druhé cíl. Ukazatel na hranu se přidá do T-seznamu zdrojového i cílového okna. Případné duplicity (multihrany) pozdějším výpočtům nevadí. Hrany nepostihují přítomnost primárních vstupů/výstupů.

3.3.3 Určení počtu úrovní

Je užito prohledávání do hloubky a dynamického programování. Vytvoří se pomocný seznam hradel, do kterých vedou primární vstupy, a algoritmus startuje postupně a nezávisle z těchto hradel. Dále seznam lokálních výsledků pro každé hradlo. Toto číslo říká počet úrovní podobvodu všech hradel tranzitivně řízených z tohoto hradla, tj. délku cesty k nejvzdálenějšímu výstupu, pokud existuje. Algoritmus při průchodu agreguje maximum z těchto výsledků, což je počet úrovní celého obvodu.

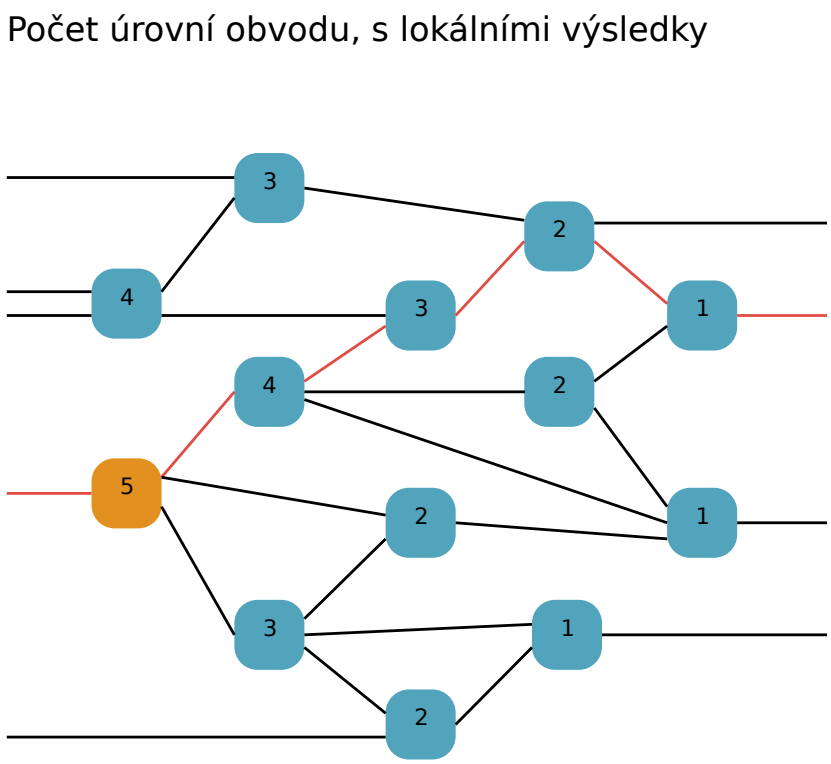
Samotný průchod probíhá pomocí zásobníku, jehož položky označují indexy oken po cestě a index právě zpracovávané hrany z T-seznamu. Přeskakují se hrany představující vstupy, index okna musí být v hraně na první pozici. Na začátku mají výsledky všech oken speciální hodnotu `FRESH`, což odpovídá dosud nevypočteným. Výpočet proběhne těsně před opuštěním okna, při návratu o jednu úroveň zpět. Má rekurzivní charakter, výsledkem je $m + 1$, kde m je maximum z platných hodnot v oknech o jednu úroveň dále. Za platnou hodnotu považujeme konečné celé číslo větší než 0. V případě, že m neexistuje a z okna vede nějaký primární výstup, generujeme v okně hodnotu 1 (triviální podobvod o 1 hradle). Pokud není ani primární výstup, výsledkem je neplatná hodnota 0 (slepá větev).

Koncept dynamického programování se zde uplatňuje ve znovupoužitelnosti lokálních výsledků. Jakmile je výsledek u okna vypočtený, při dalších návštěvách se jen přečte a celý tranzitivní podobvod se už do konce výpočtu neprochází.

Příklad výsledku výpočtu je na obrázku 3.1.

3.4 Rozdělování obvodu

Název této podkapitoly se může jevit jako zavádějící, protože obvod je již načítán jako rozdělený - na maximální počet oken po jednotlivých hradlech.



Obrázek 3.1: Počet úrovní obvodu, výsledek výpočtu.

V souladu s teorií se naopak budou tato okna spojovat dohromady (sekce 1.2.1). K rozdělování obvodu dochází z globálního pohledu, kdy na záčátku obvod chápeme jako celek. Výsledkem této části algoritmu je na konci určitý počet oken, na která je obvod rozdělený.

Cílem je rozdělit obvod na takový počet oken, který je zadán. Dodržení tohoto však není zaručeno, protože výsledný počet je zdola limitován fragmentací obvodu (a shora počtem hradel). Je tu i speciání hodnota 0, která značí, aby se obvod nerozděloval a dál chápal jako celek, jediné okno.

Algoritmus je následující:

1. Vytvořit haldu ze všech hran sousednosti (jejich ukazatelů).

2. V cyklu vždy spojit aktuálně nejvhodnější dvojici oken, iterovat dokud není dosaženo jejich požadovaného počtu nebo fragmentace obvodu.

3. Zapsat výsledná okna do souborů.

Haldu vytvoříme průchodem oken a jejich hran sousednosti. Aby v haldě nebyla každá hrana dvakrát, bereme z daného okna jen hrany představující výstupy. Hranu nelze v haldě umístit bez vyhodnocení.

Vyhodnocení hrany proběhne bezprostředně před přidáním do haldy, jako neoddělitelný krok. Vypočte se jako vážený součet počtu vstupů, výstupů a hradel okna, které by vzniklo spojením dvojice oken propojených touto hranou (váhy zadány).

V jedné iteraci cyklu se spojí dvojice oken na základě hrany s nejlepším (nejnižším) vyhodnocením. Tuto hranu získáme z vrcholu haldy. Pokud je halda prázdná, je počet oken právě roven fragmentaci obvodu, není už co spojovat a cyklus je ukončen. První okno z dvojice asimiluje druhé. Vedlejším, ale důležitým produktem této operace je T-seznam sousedů této dvojice oken (kromě nich samotných). Následně smažeme všechny hrany, které se spojovanými okny sousedí. Týká se to i hrany mezi nimi. Mažeme s nimi i ukazatele ve všech zainteresovaných T-seznamech, zároveň i v haldě. Díky funkčnímu rozšíření ji při tom **nemusíme procházet** (sekce 2.1). Nyní už můžeme smazat i samotné druhé okno, a na příslušné místo v hlavním seznamu oken (podle jeho id) zapsat `NULL`. Posledním krokem je přidání aktuálních hran mezi výsledné okno a všechny jeho sousedy, k čemuž využijeme T-seznam sousedů, který máme k dispozici. Napřed takto obnovíme strukturu grafu, a až pak tyto hrany vyhodnotíme a přidáme do haldy.

Schéma jedné iterace je na obrázku 3.2.

Po procesu spojování se projde seznam oken, a všude kde není `NULL`, zapíše se okno do BLIF souboru jako samostatný obvod. Vstupy a výstupy se získají přímo z okna, předchozím výpočtem byly udržovány aktuální. Hradla se zapíší v nezměněné podobě.

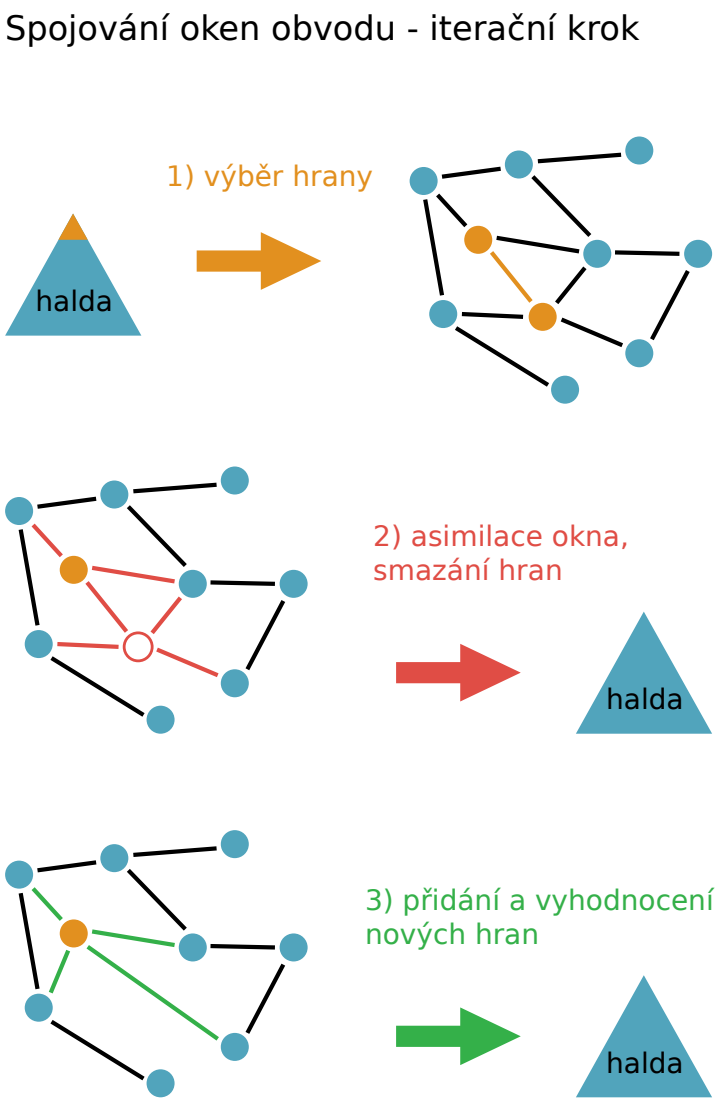
V případě, že se obvod nerozděluje, vyřeší se zápis okna prostým zkopírováním vstupního BLIF souboru.

3.4.1 Spojování dvojice oken

Binární operace nad okny, jejíž výsledek je zapsán do jednoho z nich, je definována z hlediska vstupů, výstupů a hradel - nikoli hran, ty jsou v režii dříve popsaných výpočtů. Tyto prvky okna se nastaví tak, aby definovaly spojené okno. Spojové seznamy hradel se jednoduše sloučí. Určení T-seznamů vstupů a výstupů je komplexnější operace, která se používá i při přidávání hran do haldy (tedy spojování „nanečisto“).

Prvním krokem je získání T-seznamu sousedů od obou oken. Protože jde o T-seznam, duplicity jsou řešeny implicitně. Dodatečně z něj také musíme odstranit obě okna, která jsou si samy vzájemnými sousedy.

K výpočtu potřebujeme dále založit 3 nové T-seznamy: pracovní vstupy/výstupy a průsečík. Pracovní vstupy a výstupy budou na konci tvořit výsledky u spojeného okna. Pracovní vstupy inicializujeme sjednocením vstupů z obou oken,



Obrázek 3.2: Schéma spojení dvojice oken v obvodu.

stejně tak u výstupů. Poté zapíšeme průsečík pracovních vstupů a výstupů, tedy jejich společné signály.

Pracovní vstupy zpracujeme tak, že od nich průsečík odečteme. Jsou to vzájemné signály obou oken, a každý má původ v jednom z nich. Nemohou to tedy být vstupy zvnějšku.

U pracovních výstupů budeme také odečítat průsečík, ale napřed jej musíme projít a zredukovat. Cílem bude odstranit z něj signály, které se větví i jinam než jen do druhého okna, a jsou tak součástí výsledných výstupů. To nastává ve dvou situacích: je to buď výstup primární, nebo je nalezen ve vstupech některého sousedního okna, k čemuž využijeme T-seznam sousedů. V redukovaném průsečíku tak zbydou jen vnitřní signály mezi spojovanými okny, které odečteme.

3.5 Paralelní optimalizace

Optimalizační část se skládá ze zpracování každého okna každou resyntézní metodou, spolu s vyhodnocením těchto výsledků v rámci jednoho okna. Všechny jednotky resyntézní práce jsou na sobě nezávislé, lze je tedy bez omezení dělat paralelně. Jednotka vyhodnocovací práce má jako prerekvizitu dokončení všech resyntéz na daném okně, pak je lze také dělat paralelně.

Na začátku všechny procesy založí plánovač prací, sdílený v paměti hlavního procesu, a také jeho lokální kopii u sebe v paměti. Plánovač zajišťuje korektní přístup k jednotkám práce, a plánuje tak jejich paralelizaci. Ke své funkci potřebuje informaci o počtu resyntézních metod i výsledném počtu oken. Druhé z čísel je ovšem známo jen hlavnímu procesu, tak musí dojít k jeho **distribuci**. Toto je zajištěno v rámci synchronizace plánovače (zde ve smyslu udržování jeho aktuálnosti). Hlavní proces založil lokální kopii plánovače s těmito oběma čísly, je tak jediná funkční. Tu zapíše do sdílené paměti, odkud si ji načtou ostatní procesy. V tomto bodě mají všechny procesy již k dispozici plánovač ve stejném stavu.

Poté proběhne **prvotní rozdělení práce**, k čemuž každý proces využije svůj plánovač, a opakovaně iteruje převzetí práce podle svého MPI ranku. Protože plánovač přiděluje práce v pořadí za sebou, tak musí proces, který iteroval nejvíce, zapsat svůj plánovač do sdílené paměti, aby tak jeho stav odpovídal převzetí poslední práce.

V hlavní smyčce jsou další přístupy k plánovači chráněny mutexem. Podle druhu obdržené práce proces buď spustí **externí resyntézu**, nebo **vyhodnocuje výsledky**. V případě vyhodnocování se postupně načítají obvody představující dané okno po resyntéze jednotlivými metodami, včetně původního okna bez resyntézy. Výsledkem vyhodnocení je vážený součet počtu hradel, h-vstupů, úrovní (váhy zadány) - všechna tato měřítka se vypočtou v rámci načítání. Obvod, který má tuto hodnotu minimální, se použije jako výsledek optimalizace tohoto okna. Zahrnutí původního okna do vyhodnocení

zaručuje, že optimalizace **nezhoršuje kvalitu okna** a v konečném důsledku ani celého obvodu.

3.5.1 Resyntézní metody

Resyntézní metody jsou definovány v samostatném skriptu. Je vytvořen formát pro jejich uživatelskou editaci. Stavebními kameny metod jsou **resyntézní příkazy**. Ty jsou uvedeny na neprázdných řádcích od začátku souboru, až do klíčového slova `METHODS`. Představují externí příkaz, kterým se má dané okno zpracovat. Pro uvedení souboru s oknem slouží proměnná `IO_FILE`, za kterou si program příslušně dosazuje.

Volání příkazů se děje pomocí standardní funkce `system` [9]. Ta uspí stávající proces a spustí příkazový interpret v nově vytvořeném procesu, po jehož skončení se původní proces probudí a pokračuje dále.

Konkrétní použité resyntézní příkazy jsou voláním nástroje ABC [10] s inline skriptem k provedení. Jejich společná část spočívá v načtení obvodu s převedením na AIG formu (vyjádření netlistu pouze pomocí hradel AND a invertorů) a provedení ABC příkazů `strash` a `dch`. Ve druhém a třetím se použije ještě namapování hradel na jednu z `.genlib` knihoven a příkaz `mfs`.

0. *abc -q 'read_blif -a IO_FILE; strash; dch; write_blif IO_FILE'*
1. *abc -q 'read_library ../src/mcnc.genlib; read_blif -a IO_FILE; strash; dch; map; mfs; write_blif IO_FILE'*
2. *abc -q 'read_library ../src/2-gates.genlib; read_blif -a IO_FILE; strash; dch; map; mfs; write_blif IO_FILE'*

Resyntézní metoda sestává z resyntézních příkazů, které se provedou za sebou. Zapisuje se tak posloupností *id* příkazů, indexovaných od 0 podle pořadí uvedení. Metody jsou použité tyto: `(0)`, `(1)`, `(2)`, `(0 1)`, `(0 2)`, `(1 2)`, `(2 1)`, `(0 0 0)`, `(1 1 1)`, `(2 2 2)`.

Použitím modifikace vstupně-výstupního souboru se také program stává **odolným vůči pádu** resyntézního příkazu. Pokud se tak stane, jediným důsledkem je, že soubor zůstane nezměněn. Odolnost pro resyntézní metody se dá nastavit v parametrech, zda pokračovat ve zbylých příkazech, pokud v průběhu dojde k pádu nějakého příkazu.

3.5.2 Obsluha RMA operací

Předchozí model výpočtu je třeba ještě rozšířit, s ohledem na zajištění obsluhy RMA operací hlavním procesem. Je zde třeba obsluhovat mutex a sdílenou paměť plánovače, a to neprodleně po vzniku požadavku. Způsob této obsluhy je v MPI **implementačně závislý**. Může být realizován samostatným vláknem v režii knihovny, na základě přerušení, nebo jiným specifickým způsobem [11].

V každém případě, pokud by byl hlavní proces uspán po dobu běhu procesu externí resyntézy, vzniklé RMA požadavky by obsloužil nejdříve až po probuzení, což by nepřístojně blokovalo ostatní procesy a pozdržovalo výpočet. Proto způsob, kterým hlavní proces provádí resyntézu, musí být odlišný. Základem řešení je pustit resyntézní příkaz **na pozadí** a průběžně kontrolovat jeho dokončení. To se děje pomocí dočasného souboru s exit codem resyntézy, který vytvoří volaný proces na konci. Samotné toto řešení ovšem ještě nestačí, s ohledem na výše zmíněná implementační specifika MPI. Obsluhu požadavků na RMA operace je třeba v nejhorším případě vyvolávat **aktivně**, tj. předávat řízení MPI voláním nějaké vhodné funkce, v důsledku čehož se všechny čekající požadavky obslouží. Takovými funkcemi jsou např. `MPI_Win_lock` a `MPI_Win_unlock`, které vymezují dobu přístupu k dané sdílené paměti, a nemající navzdory názvu žádný synchronizační charakter [5]. Budou se volat pro sdílenou paměť plánovače (nebo kteroukoli jinou), a to v podobě **průběžného zamykání a odemykání**, které jinak nemá žádný další efekt. Tím je problém obsluhy RMA požadavků během resyntézy vyřešen.

Stejný způsob aktivní obsluhy ovšem nelze rozumně použít u vyhodnocovacích prací, kde by v rámci původního procesu došlo ke snížení čistoty kódu a nepředpokládaně i výkonu. Proto je plánovač upraven tak, aby **nepřidělo-val hlavnímu procesu žádné vyhodnocovací práce**, pokud je procesů více.

3.6 Spojování obvodu

Výstupem resyntézy jsou jednotlivá okna obvodu, každé v samostatném souboru. Jedinými signály, které jsou u okna platné i po resyntéze, jsou jeho vstupy a výstupy. Vše ostatní jsou nová hradla i signály s novými jmény. Také u výsledného obvodu jsou vstupy a výstupy pořád stejné, zbývá do něj zapsat hradla ze všech oken. Prostý zápis by však vedl ke **kolizi jmen vnitřních signálů**, které se (zpravidla v číslované podobě) objevují v každém okně. Je nutné je přejmenovat tak, aby nekolidovaly.

Vytvoříme T-seznam vstupů a výstupů ze všech oken. To jsou signály mezi okny, ty se přejmenovávat nebudou. Dále zpracujeme každé okno odděleně.

Myšlenka je taková, aby každý vnitřní signál byl přejmenován do tvaru „n“ + *id*, kde *id* bude jeho unikátní index, číslovaný od 0 a přeskakující případné kolize se signály mezi okny.

V rámci okna potřebujeme ještě T-seznam pro jeho vnitřní signály a seznam použitých id pro tyto signály (přidružená tabulka). Procházíme hradla a parsujeme jeho řádek se jmény signálů. Zároveň vytváříme nový řádek, který ho v hradle nahradí. Procházíme pak jednotlivé signály. Pokud je to signál mezi okny, zapíšeme ho v nezměněné podobě. Jinak si zkontrolujeme jeho přítomnost ve vnitřních signálech. Pokud je nalezen, je přejmenován podle id na místě jeho indexu v T-seznamu. Pokud není, je do něj automaticky přidán

a id se mu musí přidělit. Nové id inkrementujeme tak dlouho, dokud výsledné jméno koliduje s některým ze signálů mezi okny. Po přejmenování všech signálů hradlo aktualizujeme a zapíšeme ho do souboru.

3.7 Paralelní čas

Paralelní čas určím vzhledem k modelovému obvodu, majícím v netlistu přibližně konstantní stupeň uzlů, tedy počet sousedů každého hradla. Předpokládejme dále, že při rozdělování se tato vlastnost měnit nebude. Měřítkem velikosti n zde bude počet hradel. Počet vstupů a výstupů bude rovněž konstantní, na n nezávislý. Obvod bude rozdělen na k oken ($k << n$), všechny přibližné velikosti $\frac{n}{k}$.

Výsledek bude sestávat z dílčích částí: načítání obvodu, určování počtu úrovní, rozdělování obvodu, paralelní optimalizace. Režii bariér a mutexu zanedbám.

3.7.1 Složitost načítání obvodu

U načítání obvodu je hlavní složkou procházení n hradel za účelem vytvoření netlistu. K získání ukazatele na sousední hradlo z názvu signálu slouží T-seznam a tabulka ukazatelů. K určení hashe je potřeba $\Theta(k)$ čas, kde k je délka názvu signálu. Zde je na místě položit si otázku, zda k na n nějak nezávisí. Obecně ne, ale v tomto případě je třeba vzít v potaz spodní mez k, tedy střední délku číslovaných názvů hradel. K rozlišení n hradel je minimálně zapotřebí $log(n)$ cifer, což zde dosadíme za k. Čas pro načtení obvodu je:

$$\Theta(n \cdot log(n))$$

3.7.2 Složitost určování počtu úrovní

Jediným netriviálním výpočtem u měřítek kvality je určení počtu úrovní. Protože jde o procházení do hloubky za pomoci dynamického programování, u každého hradla se vypočte lokální výsledek právě jednou. Složitost tohoto výpočtu je $\Theta(1)$, podle předpokladu konstantní sousednosti u modelu. Celkem:

$$\Theta(n)$$

3.7.3 Složitost rozdělování obvodu

Jde o iterativní výpočet, s počtem iterací $n-k$, což odpovídá počtu spojených dvojic oken. Podle předpokladu modelu, že $k << n$, je to asymptoticky $\Theta(n)$ iterací.

Složitost jedné iterace zahrnuje: spojení oken, smazání hran, přidání a výpočet nových hran.

Spojení oken má stejnou složitost jako výpočet hrany, jelikož mají společnou hlavní část výpočtu (sekce 3.4.1).

Počet mazaných i přidávaných hran je $\Theta(s)$, kde s je počet sousedů, podle předpokladu o sousednosti tedy $\Theta(1)$. Výpočet hrany pracuje také s okruhem sousedů, jde o operaci se složitostí $\Theta(s)$, výsledně opět $\Theta(1)$. Je přitom nutno podotknout, že v praxi tato část ve více propojeném obvodu může tvořit hlavní díl složitosti celé iterace. Počet hran v haldě je:

$$\Theta(s \cdot n) = \Theta(n)$$

Složitost mazání i přidávání do haldy je tak $O(log(n))$, což je i celková složitost jedné iterace.

Rozdělování obvodu má celkovou složitost:

$$\Theta(n \cdot log(n))$$

3.7.4 Složitost paralelní optimalizace

Je třeba provést pro každé z k oken m resyntézních metod, zároveň také každý výsledek resyntézy načíst a určit počet úrovní (spočítat měřítka kvality). Paralelizováno je $k \cdot m$ jednotek resyntézní práce a k jednotek vyhodnocovací práce. Předpokládejme, že počet hradel zůstane po resyntéze asymptoticky stejný.

Složitosti jednotlivých resyntézních metod jsou dopředu neznámé, těch konkrétních u ABC také neuváděné. Dále tedy předpokládejme, že asymptotická složitost každé resyntézní metody je stejná, označme ji $\Theta(R(n))$, kde n je počet hradel zpracovávaného okna. Pak je složitost celé resyntézní práce:

$$\Theta(k \cdot m \cdot R(\frac{n}{k}))$$

U jednotky vyhodnocovací práce se zpracuje $m+1$ obvodů stejných charakteristik, tj. výstupy ze všech metod a původní okno. Určování počtu úrovní můžeme vzhledem k načítání obvodu asymptoticky zanedbat. Složitost celé vyhodnocovací práce je:

$$\Theta(k \cdot (m+1) \cdot \frac{n}{k} \cdot log(\frac{n}{k})) = \Theta((m+1) \cdot n \cdot log(n))$$

Za předpokladu ideální rovnoměrné paralelizace je výsledná složitost optimalizace součtem obou prací, dělených počtem procesorů p:

$$\Theta(\frac{k \cdot m \cdot R(\frac{n}{k}) + (m+1) \cdot n \cdot log(n)}{p})$$

3.7.5 Celkový paralelní čas

Souhrnná asymptotická složitost sekvenční části je $\Theta(n \cdot log(n))$. Celkový paralelní čas:

$$T(n,p) = (\alpha + \beta) \cdot n \cdot log(n) + \frac{\gamma \cdot k \cdot m \cdot R(\frac{n}{k}) + \alpha \cdot (m+1) \cdot n \cdot log(n)}{p}$$

kde α je konstanta pro načítání obvodu, β pro rozdělování, γ pro resyntézu.

3.7.6 Efektivita, škálovatelnost

Efektivita je definována následovně: [12]

$$E(n,p) = \frac{SU(n)}{p \cdot T(n,p)}$$

$SU(n)$ představuje **nejlepší známý sekvenční algoritmus** na daný problém. V tomto případě je to $T(n,1)$, protože stávající algoritmus s rozdělováním představuje jedinou alternativu - varianta bez rozdělování dává výsledek s rozdílnou kvalitou. Po dosazení je efektivita:

$$E(n,p) = \frac{(\alpha \cdot (m+2) + \beta) \cdot n \cdot log(n) + \gamma \cdot k \cdot m \cdot R(\frac{n}{k})}{(\alpha + \beta) \cdot p \cdot n \cdot log(n) + \gamma \cdot k \cdot m \cdot R(\frac{n}{k}) + \alpha \cdot (m+1) \cdot n \cdot log(n)}$$

Protože je ve vzorci pro efektivitu neznámá funkce představující resyntézu, má smysl efektivitu i škálovatelnost určovat až vzhledem ke konkrétnímu zpracovávanému souboru. Efektivita se bude odvíjet od poměru rozdělení na sekvenční a paralelní část tohoto výpočtu. Pokud vyjdeme z **Amdahlova zákona** [12], bude pro daný počet procesorů její horní mez:

$$\frac{t_S + t_P}{p \cdot t_S + t_P}$$

kde t_S je čas sekvenční části, t_P paralelní části.

Kapitola 4

Experimenty

Testování probíhalo na dvou platformách - výpočetním clusteru `Star` [13] na FIT ČVUT a virtuální organizaci `MetaCentrum` [14], která sdružuje výpočetní uzly různých institucí pro celou akademickou obec.

4.1 Testy kvality

Pojem kvalita zde zahrnuje 4 charakteristiky, týkající se výstupního obvodu a výpočtu: **výsledný počet hradel, h-vstupů, úrovní, a sekvenční čas**. Všechny testy kvality se počítaly na Staru a 1 procesoru, aby výsledný čas nebyl zkreslován např. nevyužitým výkonem. Sekvenční čas zde tedy vystupuje jako souhrn konstantní (rozdělovací) části a variabilní (optimalizační) části. Vedle srovnání s referenčním řešením (bez rozdělování) tak bude i vyjadřovat variabilitu použitých metod vzhledem ke složitosti oken.

Vybral jsem 12 obvodů ze sbírky `BLIF.7z`, a to podle následujících kritérií: nízká fragmentace (řádově jednotky), odpovídající složitost (tisíce až desetitisíce hradel). Všechny dosažené výsledky jsou zkontrolovány na funkční ekvivalenci s originálem pomocí ABC příkazu `cec` [10].

Je použit totožný optimalizační skript se všemi metodami, jaké byly popsány v sekci 3.5.1.

Jediným měněným parametrem je počet oken. Pro každý ze souborů proběhl referenční test s 1 oknem, představujícím celý obvod nezávisle na fragmentaci. Dále sada testů postihujících rozsah počtu oken $f..(f+10)$, kde f je fragmentace obvodu.

Další parametry společné všem testům:

- váhy při rozdělování: 1 (vstupy), 1 (výstupy), 0 (hradla)
- váhy při optimalizaci: 0 (hradla), 1 (h-vstupy), 0 (úrovně)
- random seed: 0, 0

soubor	hradel	h-vstupů	úrovní	fragmentace
Other___radar___part6.blif	20658	41316	88	2
Other___fpu___part1.blif	17012	33990	3580	3
LEKO___g625.blif	14000	28000	24	1
IWLS93___bigkey.blif	2281	10865	6	3
IWLS2005___systemcaes.blif	6445	16262	30	1
ITC99___b17.blif	26303	57311	80	6
EPFL___voter.blif	13758	27516	70	1
EPFL___square.blif	18484	36968	250	1
EPFL___sqrt.blif	24618	49236	5058	1
EPFL___multiplier.blif	27062	54124	274	1
EPFL___log2.blif	32060	64120	444	1
EPFL___arbiter.blif	11839	23678	87	1

Tabulka 4.1: Testy kvality, základní údaje.

V souladu se zvolenými vahami, podle kterých program dělá optimalizaci oken, bude pro určování kvality prioritní počet h-vstupů. Jak se projevilo, počet hradel s ním do značné míry koresponduje.

oken	hradel	h-vstupů	úrovní	čas [s]
x	20658	41316	88	x
1	11727	22769	54	273
2	11708	22485	62	260
3	11722	22583	63	238
4	11773	22695	66	219
5	11548	22601	66	216
6	11512	22549	68	213
7	11543	22622	68	215
8	11381	22543	65	215
9	11229	22457	66	215
10	11219	22438	66	217
11	11260	22509	68	218
12	11282	22548	68	222

Tabulka 4.2: Testy kvality, Other___radar___part6.blif, 1 CPU.

Z tohoto testování vyplynulo, že:

- Všechny testy, bez ohledu na počet oken, přinesly **výrazné zlepšení** ve všech měřítkách kvality oproti originálu.
- Globální porovnání kvality rozdělovaných a nerozdělovaných obvodů: vždy srovnatelné v řádu jednotek procent.

oken	hradel	h-vstupů	úrovní	čas [s]
x	17012	33990	3580	x
1	7409	16621	1725	149
3	8680	16891	2647	149
4	8348	16746	2638	154
5	8372	16686	2639	161
6	8386	16726	2639	166
7	8379	16737	2639	167
8	8324	16620	2639	167
9	8411	16764	2664	168
10	8489	16848	2678	162
11	8380	16832	2667	165
12	8459	16950	2713	162
13	8488	16986	2718	163

Tabulka 4.3: Testy kvality, Other___fpu___part1.blif, 1 CPU.

oken	hradel	h-vstupů	úrovní	čas [s]
x	14000	28000	24	x
1	9787	18592	28	179
2	9745	18523	31	170
3	9704	18456	30	172
4	9710	18488	30	173
5	9660	18431	30	179
6	9642	18411	31	190
7	9610	18365	31	194
8	9589	18341	31	198
9	9585	18336	31	198
10	9585	18343	31	201
11	9497	18308	31	201

Tabulka 4.4: Testy kvality, LEKO___g625.blif, 1 CPU.

oken	hradel	h-vstupů	úrovní	čas [s]
x	2281	10865	6	x
1	2272	5852	6	50
3	2233	5713	7	88
4	2268	5710	7	91
5	2381	5923	10	93
6	2262	6053	8	96
7	2283	5945	8	98
8	2238	5782	9	99
9	2226	5728	9	102
10	2251	5811	8	105
11	2262	5871	8	106
12	2304	5761	8	108
13	2319	5873	8	112

Tabulka 4.5: Testy kvality, IWLS93___bigkey.blif, 1 CPU.

oken	hradel	h-vstupů	úrovní	čas [s]
x	6445	16262	30	x
1	5190	12708	21	128
2	5571	13966	26	151
3	5702	14221	30	146
4	5702	14221	30	150
5	6254	14933	28	155
6	6254	14933	28	155
7	6489	15017	29	153
8	6491	14912	28	155
9	6473	14885	28	153
10	6657	15260	28	155
11	6656	15249	28	149

Tabulka 4.6: Testy kvality, IWLS2005___systemcaes.blif, 1 CPU.

oken	hradel	h-vstupů	úrovní	čas [s]
x	26303	57311	80	x
1	16915	39875	56	335
6	17092	40428	56	385
7	17071	40632	67	388
8	17118	40823	77	383
9	17164	40807	78	380
10	17220	41100	78	378
11	17283	41329	78	392
12	17664	41895	85	414
13	17621	41895	85	413
14	17641	42022	85	436
15	17667	42188	85	435
16	17777	42370	85	424

Tabulka 4.7: Testy kvality, ITC99___b17.blif, 1 CPU.

oken	hradel	h-vstupů	úrovní	čas [s]
x	13758	27516	70	x
1	6580	13004	44	216
2	6556	12891	48	192
3	6418	12733	48	198
4	6458	12774	48	194
5	6510	12869	48	195
6	6503	12845	47	195
7	6509	12869	47	200
8	6538	12918	49	201
9	6513	12873	49	204
10	6504	12853	49	208
11	6483	12815	48	209

Tabulka 4.8: Testy kvality, EPFL___voter.blif, 1 CPU.

oken	hradel	h-vstupů	úrovní	čas [s]
x	18484	36968	250	x
1	13196	25948	246	265
2	12965	28417	134	237
3	13714	28809	161	221
4	13980	29283	164	220
5	14514	29725	178	220
6	14636	29904	178	220
7	14837	29997	181	221
8	14661	29912	181	224
9	14695	29949	181	224
10	14801	29900	187	225
11	15184	29981	221	227

Tabulka 4.9: Testy kvality, EPFL___square.blif, 1 CPU.

oken	hradel	h-vstupů	úrovní	čas [s]
x	24618	49236	5058	x
1	14940	29662	5839	210
2	15093	30969	5137	201
3	15258	31124	5257	175
4	15329	31215	5322	171
5	15282	31134	5340	170
6	15295	31224	5324	179
7	15342	31297	5319	184
8	15478	31257	5437	186
9	15450	31193	5446	193
10	15449	31202	5434	197
11	16205	31472	5937	195

Tabulka 4.10: Testy kvality, EPFL___sqrt.blif, 1 CPU.

- Nejlepších výsledků pro rozdělované obvody je dosaženo při **rozdělení na jednotky oken**, vyšší počty nemají smysl.

- Zvyšování počtu oken vede rovněž ke zvyšování počtu úrovní u výsledku. Je to z důvodu, že optimalizace počtu úrovní okna má své limity, a délky cest přes sousedící optimalizovaná okna mají tendenci se sčítat.

- Rozdělování povětšinou vede ke **snížení celkového času**, a to i oproti nerozdělovaným obvodům (kde je čas na toto ušetřen). Minimálního času je dosaženo při počtu v řádu jednotek oken, při vyšším začíná opět vzrůstat. Toto chování závisí na časové složitosti použitých resyntézních metod (např. podle počtu hradel). Čím větší než lineární, tím

oken	hradel	h-vstupů	úrovní	čas [s]
x	27062	54124	274	x
1	18816	37408	258	229
2	17628	37637	244	249
3	18860	37835	260	265
4	17850	37227	238	281
5	17270	36967	225	283
6	17435	37335	226	287
7	17798	37587	226	294
8	17828	37686	219	300
9	17872	37732	226	305
10	18476	38031	227	307
11	18320	37965	227	310

Tabulka 4.11: Testy kvality, EPFL___multiplier.blif, 1 CPU.

oken	hradel	h-vstupů	úrovní	čas [s]
x	32060	64120	444	x
1	24744	48441	290	482
2	19975	47080	254	525
3	20201	47289	256	485
4	20280	47347	256	478
5	20390	47483	258	474
6	20755	47941	253	369
7	20761	47944	255	370
8	20780	47992	255	372
9	23306	47938	340	394
10	23329	47970	340	397
11	23322	47973	340	399

Tabulka 4.12: Testy kvality, EPFL___log2.blif, 1 CPU.

bude snížení času výraznější, s jeho minimem při větším počtu oken.

4.1.1 Iterace

Testování těchtéž obvodů pokračuje částí, která by měla odhalit potenciál pro iterační zlepšování. Iterace je jedno nezávislé spuštění programu, kde se jako vstupní soubor bere výstup z iterace předchozí (mají stejný název).

Pro každý soubor se provede 1..10 iterací, s použitím parametru takového počtu oken (min. 2), pro který bylo v předchozím testování dosaženo výsledku s nejméně h-vstupy (přesně podle zadaných vah). Referenční řešení se pro srovnání iteruje také.

Z iteračního testování vyplynulo, že:

oken	hradel	h-vstupů	úrovní	čas [s]
x	11839	23678	87	x
1	4300	9997	20	188
2	5019	11263	28	194
3	5177	11611	28	189
4	5160	11631	28	174
5	5158	11663	29	168
6	5288	11998	29	167
7	5398	12272	31	169
8	5464	12428	31	168
9	5549	12613	30	168
10	5618	12765	30	169
11	5704	13018	31	169

Tabulka 4.13: Testy kvality, EPFL___arbiter.blif, 1 CPU.

iterace	hradel	h-vstupů	úrovní	bez rozd.:	hradel	h-vstupů	úrovní
0	20658	41316	88		20658	41316	88
1	11219	22438	66		11727	22769	54
2	11150	22395	66		11727	22769	54
3	11128	22356	66		11727	22769	54
4	11119	22330	66		11727	22769	54
5	11119	22330	66		11727	22769	54
6	11115	22318	66		11727	22769	54
7	11201	22288	66		11727	22769	54
8	11068	22213	69		11727	22769	54
9	10872	22176	65		11727	22769	54
10	10851	22155	65		11727	22769	54

Tabulka 4.14: Iterační testy, Other___radar___part6.blif, 1 CPU, 10 oken.

iterace	hradel	h-vstupů	úrovní	bez rozd.:	hradel	h-vstupů	úrovní
0	17012	33990	3580		17012	33990	3580
1	8324	16620	2639		7409	16621	1725
2	8133	16521	2600		7336	16480	1717
3	8133	16521	2600		7312	16466	1715
4	8133	16521	2600		7290	16419	1715
5	8133	16521	2600		7297	16395	1715
6	8133	16521	2600		7275	16377	1714
7	8133	16521	2600		7297	16376	1715
8	8056	16502	2590		7281	16372	1715
9	8056	16502	2590		7281	16372	1715
10	7986	16492	2553		7281	16372	1715

Tabulka 4.15: Iterační testy, Other___fpu___part1.blif, 1 CPU, 8 oken.

iterace	hradel	h-vstupů	úrovní	bez rozd.:	hradel	h-vstupů	úrovní
0	14000	28000	24		14000	28000	24
1	9497	18308	31		9787	18592	28
2	9407	18184	31		9787	18574	28
3	8983	17999	31		9787	18574	28
4	8983	17999	31		9787	18574	28
5	8846	17983	31		9787	18574	28
6	8780	17980	29		9787	18574	28
7	8780	17980	29		9787	18574	28
8	8732	17965	29		9787	18574	28
9	8732	17965	29		9787	18574	28
10	8732	17965	29		9787	18574	28

Tabulka 4.16: Iterační testy, LEKO___g625.blif, 1 CPU, 11 oken.

iterace	hradel	h-vstupů	úrovní	bez rozd.:	hradel	h-vstupů	úrovní
0	2281	10865	6		2281	10865	6
1	2268	5710	7		2272	5852	6
2	2143	5499	10		2272	5852	6
3	2133	5332	10		2272	5852	6
4	1961	5029	10		2272	5852	6
5	1963	4879	9		2272	5852	6
6	1969	4870	10		2272	5852	6
7	1921	4836	10		2272	5852	6
8	1854	4767	10		2272	5852	6
9	1875	4754	9		2272	5852	6
10	1801	4677	10		2272	5852	6
20	1776	4653	10		x	x	x
30	1756	4612	9		x	x	x
40	1756	4612	9		x	x	x
50	1744	4608	9		x	x	x

Tabulka 4.17: Iterační testy, IWLS93___bigkey.blif, 1 CPU, 4 okna.

iterace	hradel	h-vstupů	úrovní	bez rozd.:	hradel	h-vstupů	úrovní
0	6445	16262	30		6445	16262	30
1	5571	13966	26		5190	12708	21
2	5415	13406	24		4860	12009	23
3	5183	12757	28		4860	12009	23
4	4864	12064	25		4860	12009	23
5	4718	11716	25		4860	12009	23
6	4718	11716	25		4860	12009	23
7	4718	11716	25		4860	12009	23
8	4718	11716	25		4860	12009	23
9	4718	11716	26		4860	12009	23
10	4691	11669	26		4860	12009	23

Tabulka 4.18: Iterační testy, IWLS2005___systemcaes.blif, 1 CPU, 2 okna.

iterace	hradel	h-vstupů	úrovní	bez rozd.:	hradel	h-vstupů	úrovní
0	26303	57311	80		26303	57311	80
1	17092	40428	56		16915	39875	56
2	17021	39951	56		16704	39758	51
3	17002	39810	55		16704	39758	51
4	16764	39718	52		16704	39758	51
5	16768	39432	54		16704	39758	51
6	16668	39343	52		16704	39758	51
7	16563	39219	54		16704	39758	51
8	16563	39219	55		16704	39758	51
9	16563	39219	56		16704	39758	51
10	16564	39155	54		16704	39758	51

Tabulka 4.19: Iterační testy, ITC99___b17.blif, 1 CPU, 6 oken.

iterace	hradel	h-vstupů	úrovní	bez rozd.:	hradel	h-vstupů	úrovní
0	13758	27516	70		13758	27516	70
1	6418	12733	48		6580	13004	44
2	6399	12693	48		6600	12981	44
3	6387	12664	48		6600	12981	44
4	6387	12664	48		6600	12981	44
5	6387	12664	48		6600	12981	44
6	6387	12664	48		6600	12981	44
7	6387	12664	48		6600	12981	44
8	6387	12664	48		6600	12981	44
9	6387	12664	48		6600	12981	44
10	6377	12635	48		6600	12981	44

Tabulka 4.20: Iterační testy, EPFL___voter.blif, 1 CPU, 3 okna.

iterace	hradel	h-vstupů	úrovní	bez rozd.:	hradel	h-vstupů	úrovní
0	18484	36968	250		18484	36968	250
1	12965	28417	134		13196	25948	246
2	12681	27774	157		11956	25591	126
3	12681	27774	157		11956	25591	126
4	12681	27774	157		11956	25591	126
5	12665	27703	157		11956	25591	126
6	12604	27563	163		11956	25591	126
7	12405	26985	128		11956	25591	126
8	12393	26807	127		11956	25591	126
9	12412	26706	127		11956	25591	126
10	12412	26706	127		11956	25591	126

Tabulka 4.21: Iterační testy, EPFL___square.blif, 1 CPU, 2 okna.

iterace	hradel	h-vstupů	úrovní	bez rozd.:	hradel	h-vstupů	úrovní
0	24618	49236	5058		24618	49236	5058
1	15093	30969	5137		14940	29662	5839
2	15873	30611	5966		14788	29342	5892
3	15909	30597	6039		14794	29325	5888
4	15816	30515	6113		14757	29244	5906
5	15816	30515	6113		14730	29205	5905
6	15692	30262	6012		14730	29205	5905
7	15692	30262	6012		14730	29205	5905
8	15274	29789	5967		14730	29205	5905
9	14905	29378	5933		14730	29205	5905
10	14905	29378	5933		14730	29205	5905

Tabulka 4.22: Iterační testy, EPFL___sqrt.blif, 1 CPU, 2 okna.

iterace	hradel	h-vstupů	úrovní	bez rozd.:	hradel	h-vstupů	úrovní
0	27062	54124	274		27062	54124	274
1	17270	36967	225		18816	37408	258
2	17270	36967	225		18816	37408	258
3	17270	36967	225		18816	37408	258
4	17270	36967	225		18816	37408	258
5	17270	36967	225		18816	37408	258
6	17270	36967	225		18816	37408	258
7	17270	36967	225		18816	37408	258
8	17270	36967	226		18816	37408	258
9	17270	36967	225		18816	37408	258
10	17270	36967	225		18816	37408	258

Tabulka 4.23: Iterační testy, EPFL___multiplier.blif, 1 CPU, 5 oken.

iterace	hradel	h-vstupů	úrovní	bez rozd.:	hradel	h-vstupů	úrovní
0	32060	64120	444		32060	64120	444
1	19975	47080	254		24744	48441	290
2	19975	47080	254		24759	48429	290
3	19975	47080	254		24737	48392	290
4	19975	47080	254		24737	48392	290
5	19975	47080	254		24737	48392	290
6	19975	47080	254		24737	48392	290
7	19975	47080	254		24737	48392	290
8	19975	47080	254		24737	48392	290
9	19975	47080	255		24737	48392	290
10	19975	47080	254		24737	48392	290

Tabulka 4.24: Iterační testy, EPFL___log2.blif, 1 CPU, 2 okna.

iterace	hradel	h-vstupů	úrovní	bez rozd.:	hradel	h-vstupů	úrovní
0	11839	23678	87		11839	23678	87
1	5019	11263	28		4300	9997	20
2	4750	10422	26		3968	8668	18
3	4561	9921	24		3864	8474	18
4	4473	9813	23		3864	8474	18
5	4421	9669	23		3864	8474	18
6	4379	9599	21		3864	8474	18
7	4344	9421	23		3864	8474	18
8	4288	9334	23		3864	8474	18
9	4286	9215	22		3864	8474	18
10	4286	9215	23		3864	8474	18

Tabulka 4.25: Iterační testy, EPFL___arbiter.blif, 1 CPU, 2 okna.

- Nerozdělované obvody se rychle ustálí, zatímco rozdělované se s každou iterací mírně zlepšují.
- Konvergence rozdělovaných obvodů je pomalá, mnohdy nedosáhne ve stanoveném rozsahu iterací na kvalitu nerozdělovaného. Pokud je tomu naopak, lze postupně dosáhnout významnějšího rozdílu - jako je tomu u obvodů Other___radar___part6 (tabulka 4.14), LEKO___g625 (tabulka 4.16), IWLS93___bigkey (tabulka 4.17), ITC99___b17 (tabulka 4.19).
- Smysl má iterovat v řádu jednotek až desítek - obvod IWLS93___bigkey (tabulka 4.17).

4.1.2 Větší vzorek

Poslední test kvality není tak detailní jako předchozí, ale má za úkol zpra-

covat více souborů z výběru, kde ubyl požadavek na fragmentaci, tudíž jde o výběr takřka náhodný, jen s omezením na přibližnou velikost. Každý obvod se opět duálně zpracuje v 10 iteracích, s rozdělováním i bez. Protože dopředu nevíme strukturu ani přesný počet oken pro nejlepší výsledek, zvolil jsem po předchozích zkušenostech hodnotu 10 oken pro každý obvod. Pokud bude fragmentace vyšší, použije se toto číslo, čímž test také odpoví na otázku, zda má takovéto rozdělování pro dané resyntézní metody smysl.

Z tabulek 4.26 a 4.27 je vidět, že:

- I v širším měřítku platí, že rozdíly v kvalitě ohledně rozdělování jsou v řádu jednotek procent.
- Obvody s počtem oken rovným fragmentaci mají obvykle **horší kvalitu** oproti zpracování vcelku. Naopak obvody rozdělené na 10 oken jsou optimalizované lépe.

Obvody rozdělené na 10 oken mají pravděpodobně fragmentaci nižší. Zopakoval jsem pro ně stejný test s rozdělením na 5 oken.

Z tabulky 4.28 je vidět, že:

- Redukce na 5 oken přinesla výraznější zlepšení např. pro obvody ITC99___b14_opt, IWLS93___des, IWLS93___misex3. Naopak zhoršení vykázaly obvody ITC99___b20___part0, ITC99___b21___part0, ITC99___b22___part0.
- Ukazuje se, že vzhledem ke struktuře obvodu lze používat jak minimální počty oken (5), tak i vyšší (10). Pro obě hodnoty parametru existují vhodné obvody. U každého obvodu je to třeba individuálně vyzkoušet.

4.2 Testy výkonu

Testy výkonu mají vyjádřit, jak rychle proběhne rozdělení velkého obvodu a jak obecně funguje paralelizace výpočtu. Pro tyto účely jsem vybral obvod LEKU___CD pro Star a LEKO___g1296 pro MetaCentrum.

LEKU___CD byl vybrán jako zátěžový test, LEKO___g1296 z důvodu malého poměru sekvenční a paralelní části, kde má smysl přidávat více procesorů.

U obvodu LEKU___CD (tabulka 4.30) probíhalo jeho rozdělení za cca. 285 s. Z tabulky je vidět, že více času zabírala optimalizace, paralelizací se tak dosáhlo **významného snížení celkového času**. Efektivita přitom klesá pozvolna, má smysl tedy použít na tento obvod řádově jednotky procesorů.

U obvodu LEKO___g1296 (tabulka 4.31) probíhalo jeho rozdělení za cca. 25 s. Teoreticky by si měl udržovat vyšší míru efektivity než předchozí obvod, ale výsledky dopadly oproti očekávání hůře. Souvisí to s několika **problémy**, které toto testování potkaly, a které se pokusím rozebrat níže:

soubor	oken	hradel	h-vstupů	úrovní
Cookbook___gearbox_40_66.blif	150	3124	9194	10
Cookbook___gearbox_66_40.blif	108	2933	8242	10
Cookbook___lane_tx.blif	274	2697	7148	11
Cookbook___rgb_to_hue___part2.blif	10	1805	4455	15
Cookbook___rx_buffer_fifo_2.blif	804	2420	6441	5
Cookbook___tx_4channel_arbiter.blif	21	2161	6430	12
EPFL___sin.blif	10	4179	8662	152
Illinois___piir8.blif	49	1272	2417	59
Illinois___piir8o.blif	49	1459	2765	69
ITC99___b14_opt.blif	10	3275	7507	73
ITC99___b20___part0.blif	10	6317	14472	81
ITC99___b21___part0.blif	10	6107	14025	79
ITC99___b22___part0.blif	10	9448	21713	68
IWLS2005___aes_core.blif	138	13293	32899	17
IWLS2005___des_area.blif	10	3310	7991	31
IWLS2005___DMA.blif	54	13483	33088	15
IWLS2005___DSP___part0.blif	95	24957	60540	34
IWLS2005___ethernet___part0.blif	53	25548	67162	17
IWLS2005___pci_bridge32.blif	85	10334	28028	23
IWLS2005___wb_conmax.blif	17	17420	54350	16
IWLS93___clma___part0.blif	54	3143	7638	21
IWLS93___des.blif	10	2523	5574	15
IWLS93___misex3.blif	10	2249	5387	28
LEKO___g1296.blif	10	28899	55899	37
Other___aqua___part0.blif	23	13331	31880	117
Other___carpat___part22.blif	10	4119	8137	165
Other___carpat___part32.blif	10	3008	5842	61
Other___cfft___part0.blif	18	1939	4477	29
Other___cord2.blif	150	7105	16051	58
Other___fp_operators___part3.blif	10	11221	27823	330
Other___mem.blif	107	7888	19331	15
Other___ray___part0.blif	899	18693	44296	76
Other___ray___part7.blif	10	1881	3811	80
Other___video___part1.blif	10	11167	22102	60
QUIP___fip_risc8.blif	13	2413	8158	24
QUIP___oc_aes_core_inv.blif	185	1967	7108	9
QUIP___oc_cordic_p2r.blif	145	2081	7326	17
QUIP___oc_fpu___part0.blif	18	2570	8994	72
QUIP___oc_pci___part0.blif	144	2663	8833	18
QUIP___oc_vga_lcd.blif	249	2368	7856	21
QUIP___os_blowfish.blif	150	2247	7561	34
QUIP___uoft_raytracer___part7.blif	10	15864	51379	72

Tabulka 4.26: Testy kvality na větším vzorku, min. 10 oken, 10 iterací.

soubor	hradel	h-vstupů	úrovní
Cookbook___gearbox_40_66.blif	3006	7497	9
Cookbook___gearbox_66_40.blif	2530	6261	9
Cookbook___lane_tx.blif	1720	4072	11
Cookbook___rgb_to_hue___part2.blif	1310	3254	13
Cookbook___rx_buffer_fifo_2.blif	2420	7242	6
Cookbook___tx_4channel_arbiter.blif	2167	6436	12
EPFL___sin.blif	4610	8897	145
Illinois___piir8.blif	1270	2413	58
Illinois___piir8o.blif	1436	2717	64
ITC99___b14_opt.blif	3234	7554	31
ITC99___b20___part0.blif	6765	15549	44
ITC99___b21___part0.blif	6701	15446	41
ITC99___b22___part0.blif	10166	23065	45
IWLS2005___aes_core.blif	13576	33361	16
IWLS2005___des_area.blif	3616	8382	25
IWLS2005___DMA.blif	13220	32813	15
IWLS2005___DSP___part0.blif	24679	59839	33
IWLS2005___ethernet___part0.blif	25761	67418	16
IWLS2005___pci_bridge32.blif	10334	28028	23
IWLS2005___wb_conmax.blif	18645	55038	13
IWLS93___clma___part0.blif	3247	7900	20
IWLS93___des.blif	2127	5552	12
IWLS93___misex3.blif	1410	3523	10
LEKO___g1296.blif	30120	58517	28
Other___aqua___part0.blif	13340	31962	119
Other___carpat___part22.blif	4300	8240	142
Other___carpat___part32.blif	3108	5908	59
Other___cfft___part0.blif	1778	4067	25
Other___cord2.blif	7111	15941	41
Other___fp_operators___part3.blif	11315	28972	287
Other___mem.blif	7967	19396	17
Other___ray___part0.blif	18556	45011	39
Other___ray___part7.blif	2035	3955	76
Other___video___part1.blif	11097	21349	53
QUIP___fip_risc8.blif	2413	8158	24
QUIP___oc_aes_core_inv.blif	1964	7115	7
QUIP___oc_cordic_p2r.blif	2081	7326	17
QUIP___oc_fpu___part0.blif	2527	9088	72
QUIP___oc_pci___part0.blif	2663	8833	17
QUIP___oc_vga_lcd.blif	2367	7863	16
QUIP___os_blowfish.blif	2240	7564	34
QUIP___uoft_raytracer___part7.blif	15864	51379	71

Tabulka 4.27: Testy kvality na větším vzorku, bez rozdělování, 10 iterací.

soubor	oken	hradel	h-vstupů	úrovní
Cookbook___rgb_to_hue___part2.blif	6	1805	4449	14
EPFL___sin.blif	5	4252	8739	154
ITC99___b14_opt.blif	5	3025	6868	74
ITC99___b20___part0.blif	6	6881	15742	40
ITC99___b21___part0.blif	6	6735	15477	42
ITC99___b22___part0.blif	6	10315	23367	46
IWLS2005___des_area.blif	5	3220	7960	31
IWLS93___des.blif	5	2109	5454	12
IWLS93___misex3.blif	5	2086	4966	29
LEKO___g1296.blif	5	29595	56228	38
Other___carpat___part22.blif	5	4203	8141	170
Other___carpat___part32.blif	5	2949	5787	60
Other___fp_operators___part3.blif	5	10969	27946	314
Other___ray___part7.blif	5	1898	3842	77
Other___video___part1.blif	5	11437	22211	62
QUIP___uoft_raytracer___part7.blif	5	15864	51379	73

Tabulka 4.28: Testy kvality na větším vzorku, min. 5 oken, 10 iterací.

soubor	hradel	h-vstupů	úrovní	sekv. čas bez rozd. [s]
LEKU___CD.blif	1167052	2334104	19	x
LEKO___g1296.blif	52704	105408	52	449

Tabulka 4.29: Testy výkonu, základní údaje.

CPU	čas [s]	efektivita
1	2579	1.00
2	1579	0.82
3	1090	0.79
4	898	0.72
5	782	0.66
6	714	0.60
7	656	0.56
8	614	0.53
9	587	0.49
10	559	0.46
11	550	0.43
12	533	0.40

Tabulka 4.30: Testy výkonu, Star, LEKU___CD.blif, 25 oken.

CPU	čas [s]	efektivita
1	429	1.00
2	237	0.91
3	171	0.84
4	149	0.72
8	101	0.53
16	88	0.30
32	77	0.17
64	171	x

Tabulka 4.31: Testy výkonu, MetaCentrum, LEKO___g1296.blif, 10 oken.

1. 64 procesorů byl maximální počet, pro který se mi podařilo program alespoň jednou úspěšně spustit. Příčinou vždy bylo násilné ukončení plánovačem z důvodu překročení počtu procesorů, na začátku přiděleného pro úlohu. To jedině může souviset se základní architekturou programu, který volá externí resyntézu pomocí funkce `system` [9]. Nově vytvořený proces je pak namísto setrvání na domovském procesoru zřejmě někdy v rámci uzlu přeplánován na jiný procesor s cizí úlohou, což vede k chybě.

2. Pokud mají procesy úlohy na každém z procesorů k dispozici plný výkon (základní předpoklad), mohla by v bodu `1` uvedená skutečnost stát i za výrazně zhoršenými časy, co se týče resyntézy. Např. pokud je několik takto vytvořených procesů umístěno na stejný, zrovna volný procesor.

3. I při opakovaném spouštění na uzlech o stejném výkonu výrazně fluktuoval i čas ze sekvenční části, což se začalo znatelně projevovat už od 4 procesorů. Způsob měření času je pomocí časových razítek na různých místech programu, což zajišťuje MPI. Na hlavním procesoru, kde sekvenční výpočet probíhá, není v programu mezi těmito dvěma razítky **žádná komunikace s ostatními procesy**, a po celou dobu běží (kromě operačního systému) jen tento jediný proces. Hypotézu z bodu `2` zde tedy nelze uplatnit. Možnou odpozorovanou příčinou je negativní interakce s úlohami běžícími na ostatních procesorech v uzlu, což by stálo proti předpokladu nezávislosti přidělených procesorů (spekulace přítomnosti podpůrných procesů pro režii uzlu).

Pokud odhlédneme od těchto specifických problémů, tak lze program spouštět na neomezeném množství procesorů, na dané architektuře s takovým plánováním nových procesů, jako je nastíněno bodem `1`.

Závěr

Nástroj slouží k libovolně zvolené externí resyntéze obvodu, přičemž metody lze kombinovat. Bylo dosaženo všech základních cílů práce. Zpracování obvodu, vedoucí k jeho rozdělení, mělo u všech provedených experimentů použitelnou časovou složitost, která často vedla ke snížení celkového času optimalizace obvodu, než při zpracování vcelku.

Rozdělení obvodu podle stanoveného klíče může vést ke zlepšení kvality, výsledkem ovšem pořád srovnatelně s variantou bez rozdělování. Přínos se otvírá při iterování výpočtu, kde opakované rozdělování vede k mírnému zlepšování kvality v každé iteraci.

V testu kvality, kde se „naslepo“ rozdělovaly obvody na 10 oken (bez ohledu na skutečnou fragmentaci, ale naopak s ohledem na výpočet v praxi) vykázalo snížení počtu hradel 18 ze 42 obvodů celkem. Toto snížení nejvíce dosáhlo hodnoty -6 %. Výsledky dosažené individuálním zkoušením počtu oken u jiných obvodů byly i -20 % (tabulka 4.12). Tímto je nastíněn potenciál experimentů i s různě nastavenými parametry vah, na které v této práci z důvodu jednotnosti testování nedošlo, a získat tak optimální výsledky v rámci nástroje.

Zkoušení nástroje na velkém počtu procesorů selhalo pro počty větší než 64, patrně z důvodu migrace procesů s resyntézou na jiné procesory. To se negativně projevovalo i v časové rovině. Teoreticky nic nebrání efektivní paralelizaci paralelní části, což bylo tak dokázáno pouze do 12 procesorů.

Literatura

[1] Fišer, P.; Schmidt, J.: It Is Better to Run Iterative Resynthesis on Parts of the Circuit. 2010. Dostupné z: http://users.fit.cvut.cz/~fiserp/papers/iwls10.pdf

[2] University of Tennessee: *MPI-2: Extensions to the Message-Passing Interface*. Dokumentace k MPI-2. Dostupné z: http://users.fit.cvut.cz/~soch/mi-par/mpi_home/docs/mpi-20-html/mpi2-report.html

[3] Černý, J.: Základní grafové algoritmy. 2010, kapitola Halda. Dostupné z: http://kam.mff.cuni.cz/~kuba/ka/halda.pdf

[4] Berezovský, M.; Mařík, R.: Search trees, binary trie, patricia trie. 2012, binární trie. Dostupné z: https://cw.fel.cvut.cz/wiki/_media/courses/a4m33pal/paska13trie.pdf

[5] Gropp, W.: Lecture 35: More on One Sided Communication. 2015, dokumentace k RMA. Dostupné z: http://wgropp.cs.illinois.edu/courses/cs598-s16/lectures/lecture35.pdf

[6] GitHub: *Libevent perturbs application calls to rand()*. Rušivá MPI volání rand(). Dostupné z: https://github.com/libevent/libevent/issues/384

[7] Cook, J. D.: Simple Random Number Generation. 2014. Dostupné z: http://www.codeproject.com/KB/recipes/SimpleRNG.aspx

[8] University of California, Berkeley: *Berkeley Logic Interchange Format (BLIF)*. Dostupné z: https://www.cse.iitb.ac.in/~supratik/courses/cs226/spr16/blif.pdf

[9] die.net: *system(3) - Linux man page*. Funkce system() v prostředí UNIX. Dostupné z: https://linux.die.net/man/3/system

[10] Berkeley Logic Synthesis and Verification Group: *ABC - A System for Sequential Synthesis and Verification*. Dokumentace k ABC. Dostupné z: `https://people.eecs.berkeley.edu/~alanmi/abc/`

[11] Stack Overflow: *MPI with C: Passive RMA synchronization*. Implementační specifika MPI u RMA. Dostupné z: `https://stackoverflow.com/a/18767421`

[12] Tvrdík, P.: *Parallel Algorithms and Computing*. ČVUT, 2003.

[13] ČVUT FIT: *Výpočetní svazek STAR*. Architektura STARu. Dostupné z: `https://edux.fit.cvut.cz/oppa/MI-PAR/cviceni/cv5.pdf`

[14] MetaCentrum VO: *O MetaCentru VO*. Dostupné z: `https://metavo.metacentrum.cz/cs/about/index.html`

Příloha **A**

Použití programu

Program byl odladěn v prostředí serveru `star.fit.cvut.cz` [13], pro něž také předkládám postup pro spuštění:

1. Nakopírovat na server adresář `src` se zdrojovými kódy programu.
2. V adresáři, kde je umístěn `src`, založit ještě adresář `test` a nakopírovat do něj vybrané `.blif` soubory.
3. Z adresáře `test` spustit překlad programu: `../src/preklad.sh`
4. Upravit v `src` dle potřeby skripty `spusteni.sh` (pro počet procesorů) a `diplomka.sh` (pro parametry programu).
5. Upravit soubor `methods.txt` na vlastní zvolené resyntézní metody (formát popsán v sekci 3.5.1), doinstalovat potřebné externí nástroje (ABC, ...).
6. Spustit program z adresáře test: `../src/spusteni.sh`

Všechny parametry programu jsou nepovinné, kromě specifikace vstupního souboru. Pokud je i ta vynechána, program vypíše seznam všech svých parametrů a ukončí se. Popis parametrů:

- **-o** ... Jméno výstupního souboru. Pokud není uvedeno, použije se jméno vstupního s příponou „_o“.
- **-n** ... Počet oken, na který chceme obvod rozdělit. Pokud je menší nebo rovno fragmentaci obvodu, rozdělí se obvod podle fragmentace. Podobně pokud je větší nebo rovno počtu hradel, povede to k rozdělení na jednotlivá hradla. Je tu také speciální hodnota 0, která značí, že se obvod ponechá vcelku. To lze použít např. na srovnání kvality s výsledkem, kde bylo rozdělování použito.

- **-w** ... Váhy užívané při rozdělování obvodu, které ovlivní podobu výsledných oken. Váhy jsou 3: pro počet vstupů, výstupů, hradel. Vyjadřují, na jaké vlastnosti u vznikajících oken chceme dávat důraz, aby byly minimální.
- **-r** ... Váhy pro výsledky resyntézy, podle nichž se vybírá ten nejlepší. Jsou také 3: pro počet hradel, h-vstupů, úrovní. Vyjadřují důraz na vlastnosti okna jako samostatného obvodu, který prošel resyntézou.
- **-m** ... Jméno souboru s resyntézními metodami.
- **-force** ... Značí, pokud je metoda tvořena více příkazy a jeden z nich skončí s chybou, zda provádět i zbylé příkazy.
- **-rand** ... Nastavení seedu pro generátor pseudonáhodných čísel. Zde jsou to 2 celá čísla. Pokud není nastaven, generátor se inicializuje náhodně (podle systémového času).
- **-nclear** ... Pokud je uveden, nebudou se po proběhnutí výpočtu mazat žádné pracovní soubory. Lze si tak např. prohlédnout, jak byl obvod rozdělen, a soubory pak smazat ručně.
- **-log** ... Podrobné logování všech kroků výpočtu. Každý procesor loguje do svého souboru.
- **-nwarn** ... Nebudou se logovat žádná varování, např. pokud resyntézní příkaz skončí s chybou.

Příloha B

Seznam použitých zkratek

ABC název nástroje, zde používaného na resyntézu

AIG and-inverter graph

BLIF Berkeley Logic Interchange Format

MPI Message Passing Interface

RMA Remote Memory Access

Příloha C

Obsah přiloženého CD

- `src` ..... zdrojové kódy
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF
- `experimenty` ..... záznamy ze všech experimentů
  - `star-k1` ..... Star - podrobné testy kvality
  - `star-k2` ..... Star - rozsáhlejší testy kvality
  - `star-meta-v1` ..... Star, MetaCentrum - testy výkonu
- `blif` ..... kompletní sbírka obvodů k testování