České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů

Bakalářská práce

Paralelizace nástroje pro minimalizaci logických funkcí BOOM

Richard Fritsch

Vedoucí práce: Ing. Petr Fišer, Ph.D.

Studijní program:
Elektrotechnika a informatika, strukturovaný, bakalářský

Obor: Výpočetní technika

26. května 2011

Poděkování

Chtěl bych poděkovat vedoucímu práce Ing. Petru Fišerovi, PhD., za odborné vedení práce a za cenné rady při psaní textové části práce.
Dále bych rád poděkoval své rodině za trpělivou podporu, bez které by pro mne bylo psaní této práce mnohem obtížnější.

Prohlášení

Prohlašuji, že jsem svou bakalářskou 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 Černošicích dne 26. května 2011 ……

Abstract

This thesis focuses on parallelization of the BOOM Boolean minimizer. The content consists of a short background research on parallel programming and available platforms, parallelization of BOOM as a whole, parallelization of individual phases of minimization and tests of the final parallel program.

Abstrakt

V rámci této práce byl paralelizován nástroj pro minimalizaci logických funkcí BOOM. Obsahem práce je krátká úvodní rešerše paralelního programování a dostupných paralelních platforem, paralelizace nástroje BOOM jako celku, paralelizace jednotlivých fází a závěrem jsou provedeny testy výsledného paralelního nástroje.

Obsah

1 Úvod

S problémem minimalizace logických funkcí se setkáváme při návrhu PLA, víceúrovňové logiky, řídicích systémů, při návrhu built-in self-testů (BIST) a také v softwarovém inženýrství, umělé inteligenci apod. Kromě původní metody Quine-McCluskey a modernějšího algoritmu ESPRESSO byl vyvinut také BOOM (1), schopný dostatečně rychle řešit i problémy s velkým počtem vstupních proměnných a pomocí algoritmu FC-Min (2) i funkce s velkým počtem výstupů.

Tato práce má za cíl BOOM paralelizovat, tedy umožnit jeho vykonávání na více jádrech CPU. Úvodem je nastíněno paralelní programování, následně je provedena krátká rešerše dostupných paralelních platforem. Hlavní část práce tvoří paralelizace BOOMu jako celku a paralelizace jednotlivých kroků minimalizace. Závěrem je výsledný paralelní nástroj otestován na zkušebních úlohách.

V textu často uvádím *kurzívou* anglické termíny, je tomu tak proto, že buď kvůli skoro neexistující české literatuře či překladům často nejsou ustáleny jejich české ekvivalenty, nebo je uvádím čistě informativně jako klíčová slova pro hledání v anglické literatuře.

1.1 Paralelní programování

Technologické a fyzikální limity zastavily zdvojnásobování frekvencí procesorů každé dva roky během posledních tří desítek let. Aby následující generace procesorů přinesly očekávaný nárůst výkonu, začali výrobci procesorů vyrábět čipy s více jádry. Více procesorů na nižších frekvencích vytváří méně tepla a spotřebovává méně energie než jednoprocesorové čipy pokračující v trendu zdvojnásobování frekvence.

Jak využít tyto další jádra, potažmo procesory? Můžeme spustit více aplikací a každá může běžet na jiném jádře – tím získáme skutečně paralelní vykonávání práce. Takto ovšem můžeme provozovat jen omezený počet aplikací, a pokud nejsou výpočetně náročné, pravděpodobně plýtváme cykly procesoru.

Další možností je psát aplikace, které umějí využít další jádra pro spouštění náročných výpočtů, které mohou probíhat nezávisle na sobě. Psaní takových programů se nazývá paralelní programování (*concurrent programming*). S každým programovacím jazykem či metodologií jsou spojeny různé techniky, triky, pasti a nástroje pro implementaci takových programů.

V minulosti bylo paralelní programování doménou malého počtu programátorů, typickou aplikací byly vědecko-technické výpočty. S masivním rozvojem vícejádrových procesorů se paralelní programování stává mainstreamem.

Paralelní programování je o navzájem nezávislých výpočtech, které mohou být vykonány v jakémkoliv pořadí. Např. jednotlivé iterace cyklů mohou často být prováděny nezávisle na sobě. Jakákoliv práce, která lze vytáhnout ze sekvenčního kódu, může být přidělena vláknům či procesům a vykonána na dalších jádrech, která jsou k dispozici (nebo i pouze na jednom procesoru pomocí přepínání kontextu a vytváření iluze paralelního spouštění). Ne všechny výpočty v aplikaci jsou však nezávislé.

Při psaní paralelního kódu je obvykle nutné volat rutiny např. z knihoven pro práci s vlákny. Tato volání vždy znamenají určitou režii (*overhead*), která v původním sekvenčním kódu nebyla nutná. Do režie se počítají všechny rutiny pro manipulaci s vlákny, synchronizaci, signalizaci, nerovnoměrné rozdělení práce či neužitečné vytížení procesorů. Funkce zajišťující ideální rozdělení práce (maximální produktivní vytížení co největšího počtu procesorů), tzv. *load balancing* jsou také režií. Při psaní paralelního kódu je pochopitelně potřeba režii minimalizovat.

2 Dostupné paralelní platformy

2.1 Paralelní programovací modely

Paralelní programovací modely můžeme rozdělit na modely se sdílenou pamětí (*shared memory*) a modely s distribuovanou pamětí (*distributed memory*).

Jak jméno naznačuje, model se sdílenou pamětí předpokládá, že program poběží na jednom či více procesorech, které sdílejí buď jen část, nebo celou dostupnou paměť. Programy využívající sdílenou paměť jsou typicky tvořeny několika vlákny (*threads*, nezávislé entity vykonávající tok instrukcí), která sdílí adresní prostor, ale mohou mít navíc také vlastní (*private, thread-local*) data. Programovací modely musí poskytovat prostředky umožňující spouštění vláken a koordinaci jejich přístupu ke sdíleným datům, včetně těch, které zaručují, že některé operace může současně vykonávat jen jediné vlákno.

Pro systémy s distribuovanou pamětí existuje jiný způsob komunikace – zasílání zpráv (*message passing*). Tento model předpokládá, že programy budou vykonávány několika procesy, každý s vlastním adresním prostorem. Modely založené na zasílání zpráv musí poskytovat prostředky umožňující startování procesů, posílání a přijímání zpráv a také vykonávání operací na datech distribuovaných mezi procesy. Modely založené čistě na zasílání zpráv předpokládají, že procesy spolupracují na výměně zpráv kdykoliv některý potřebuje data produkovaná jiným. Novější modely ale využívají i jednostranné komunikace – procesy mohou přes síť pracovat i s pamětí v jiném uzlu.

2.2 Paralelní platformy

Existuje několik možností jak programovat paralelní aplikace pro systémy se sdílenou pamětí, tj. na architekturách se sdílenou pamětí u vícejádrových (*multicore*), SMP (*Symmetric Multi-Processing*), a cc-NUMA (*cache coherent Non-Uniform Memory Access*) systémů.

Typickou variantou jsou vlákna - UNIXové operační systémy podporují vlákna již mnoho let, což je i jeden z hlavních důvodů jejich úspěchu na poli serverů. Zejména GNU/Linux od verze 2.6 výrazně zlepšil efektivitu vláken novým

plánovačem a knihovnou Native POSIX Thread Library (NPTL), která nahradila dřívější Linux threads nativními POSIX vlákny. Druhou možností je na GNU/Linuxu i MS Windows API nazvané OpenMP.

2.2.1 POSIX vlákna (pthread)

Knihovna pthread je sada rutin pro koordinaci vláken vyvinutá IEEE v rámci standardu POSIX. Výrobci UNIXových systémů tuto knihovnu hromadně implementovali, její adopce v Linuxu a port na platformu MS Windows dále zlepšili její přenositelnost.

Pthread specifikuje potřebná rozhraní pro práci s vlákny – vytváření a ukončování vláken, jejich koordinaci a čekání na jejich dokončení. Veškerý přístup ke sdíleným datům je nutné explicitně synchronizovat – k tomu je k dispozici několik synchronizačních primitiv: mutexy, podmíněné proměnné, bariéry a semafory, ty však technicky nejsou součástí knihovny pthread, jsou ale konceptuálně spojené s vlákny a jsou k dispozici na všech systémech kde je pthread.

Na MS Windows je toto API k dispozici dvěma způsoby – pomocí portu v podobě knihovny pthreads-win32, která je postavena nad nativními Windows vlákny (tato knihovna je mj. použita v implementaci GNU pro Windows Mingw). To sice přináší jistou režii, ale lze je takto provozovat na všech verzích i edicích MS Windows. Druhý způsob je pomocí nativního POSIX subsystému ve Windows (ve verzi XP označován jako Windows Services for UNIX - SFU, ve verzích Vista a 7 jako Subsystem for Unix-based Applications - SUA). Tento subsystém je ale součástí pouze Ultimate a Enterprise edicí Windows a pro jeho využití je nutné programy speciálně kompilovat.

Při použití pthread je třeba psát kód specificky pro toto API – je třeba deklarovat pthread struktury a volat pthread-specifické funkce. Pthread je na UNIXových systémech slinkována s aplikací jako jiné knihovny.

Přestože je knihovna pthread poměrně komplexní a velmi portabilní, trpí vadou společnou všem nativním vláknovým knihovnám – je potřeba značné množství kódu specifického pro vlákna. Jinými slovy, programování s pthread neodvratně změní kód aplikace na vícevláknový model. Navíc např. počty vláken a další parametry se mohou stát pevnou součástí kódu (tzv. *hard-coded*). Toto je ale kompenzováno maximální kontrolou nad všemi operacemi

s vlákny – pthread je nízkoúrovňové (*low-level*) API, kde je obvykle potřeba několika kroků k provedení i poměrně jednoduché operace. Například pro paralelní procházení polem dat je potřeba deklarovat specifické struktury, vytvořit vlákna, vypočítat a předat jim meze pro procházení a nakonec vlákna ukončit – to vše musí být explicitně naprogramováno. Pokud je třeba víc než jen procházení, množství kódu značně poroste. Toto ale platí pro všechny nativní knihovny, ne jen pro pthread. Hledání snazších cest vyústilo ve vytvoření OpenMP.

2.2.2 OpenMP

V devadesátých letech byl společný problém vývojářů fakt, že každý operační systém jinak pojímal programování s vlákny – UNIX používal pthread, Sun Solaris threads, Windows své vlastní API a Linux tehdy Linux threads. Spolek výrobců pod záštitou Silicon Graphics přišel s API, které by beze změn mělo fungovat pod UNIXem/Linuxem i Windows.

Specifikace OpenMP obsahuje API a direktivy pro překladače (*pragma*). OpenMP prošlo několika revizemi a direktivy se ukázaly jako nejužitečnější. Rozumným použitím direktiv může být sekvenční program paralelizován bez použití API a toto je i preferovaný způsob.

Direktivy překladače (v kódu jako `#pragma`) umožňují překladači dát k dispozici specifické vlastnosti OS či hardwaru a současně stále dodržovat standard jazyka C/C++. Když překladač narazí na direktivu, kterou nezná, musí ji ignorovat, což je jedna z klíčových vlastností – kód bude stále fungovat i s překladačem, který nepodporuje OpenMP.

```
#pragma omp parallel for
for (int i = 0; i < N; i++) {
    printf("i = %d", i);
}
```

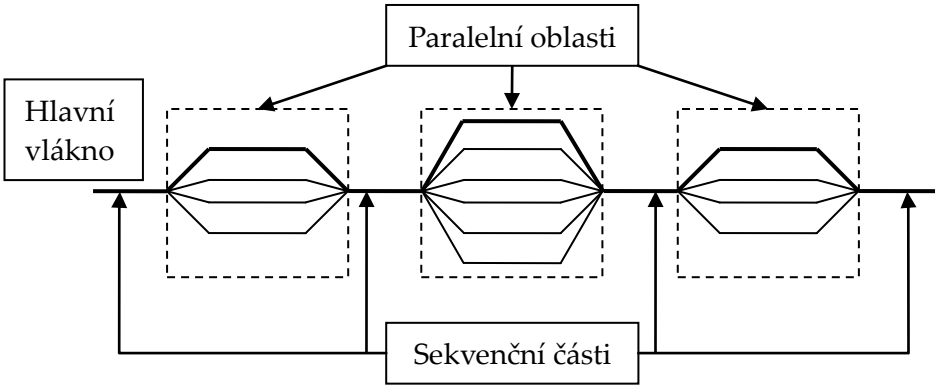
Direktiva `#pragma omp parallel for` v předchozí ukázce kódu sděluje překladači, že následující `for` cyklus má být prováděn paralelně. OpenMP se postará o vytváření vláken, přidělení mezí v poli a jejich ukončení. OpenMP negarantuje a priori, kolik vláken bude vytvořeno, obvykle je ale zvolen stejný počet jako počet exekučních jednotek (tj. procesorů, jader nebo HyperThreading jednotek). Nastavením proměnné prostředí lze počet vláken měnit.

S pomocí OpenMP ve verzi 3.0 je dokonce možné paralelizovat i průchod C++ STL kontejneru pomocí iterátorů:

```
#pragma omp parallel for
for (it = list1.begin(); it != list1.end(); it++) {
    it->compute();
}
```

K dispozici je mnoho dalších direktiv, např. k určení, zda má být proměnná sdílená či privátní pro každé vlákno, k synchronizaci vláken atd.

Průběh typického OpenMP programu můžeme naznačit následujícím diagramem:



Obrázek 1: Program paralelizovaný pomocí OpenMP

Mezi hlavní výhody OpenMP patří přenositelnost a minimální změny původního kódu. OpenMP poskytuje tzv. *medium-grained* (středně zrnitou) kontrolu nad paralelizací s poměrně vysokou mírou abstrakce, výborně se hodí pro paralelismus na úrovni dat (*data parallelism*) a je vhodné pro mnoho aplikací. OpenMP však neumožňuje paralelismus na úrovni úloh či funkcí (*task-level parallelism*).

2.3 Které API je vhodné?

OpenMP je výhodné, protože programu předem neurčuje pevný počet vláken. To může být velký problém pro vícevláknové aplikace používající *low-level* API jako pthread či nativní Windows vlákna – dosáhnout dobré škálovatelnosti (*scalability*) pokud je k dispozici více procesorů či jader není jednoduché.

Jedním z možných přístupů jsou tzv. *thread pools*, kdy je při spuštění vytvořena skupina vláken a úlohy jsou jim dynamicky přidělovány. Toto však vyžaduje množství kódu specifického pro vlákna a optimální škálování stále nemusí zaručeno.

Direktivy OpenMP mají také výhodu v možnosti vypnutí podpory OpenMP při překladu, program pak bude zkompilován jako sekvenční. To se může zdát jako nepodstatné, když cílem je paralelizace a ne udržení kompatibility, nesmírně to však usnadňuje ladění. Bez této možnosti je často velmi obtížné zjistit, zda je chybné chování komplexního kódu způsobeno nesprávnou prací s vlákny, či chybou nesouvisející s vlákny.

Pokud je potřeba s OpenMP docílit jemnější zrnitosti paralelismu (*fine-grained*), je možné použít kromě direktiv překladače i poskytované rutiny. Ty se dají rozdělit do skupin pro zjišťování vlastností prostředí a nastavování počtu vláken, používání zámků (*locks*) a pro časování. Používání těchto rutin není doporučováno, protože jsou jimi potlačeny výhody plynoucí z používání výhradně direktiv. S použitím rutin je OpenMP de facto jen malou podmnožinou funkcionality nabízené knihovnou pthread či Windows vlákny. Obě API jsou portabilní, nicméně pthread nabízí daleko větší paletu synchronizačních primitiv, která poskytují jemnější kontrolu nad operacemi s vlákny. V aplikacích, kde je třeba řídit vlákna individuálně, jsou tedy přirozenější volbou nativní vlákna jako pthread či Windows vlákna.

Vhodné vkládání direktiv OpenMP do sekvenčních programů umožní mnoha aplikacím využít paralelní architektury se sdílenou pamětí, a to jen s malými nebo žádnými změnami kódu. V praxi je v mnoha aplikacích značný paralelismus, který se dá snadno zužitkovat.

Úspěch OpenMP může být přisuzován několika faktorům – důraz na strukturované paralelní programování a poměrně jednoduché použití, kde překladač vyřeší spoustu věcí za programátora. Za OpenMP stojí skupina firem, které zároveň dodávají velkou část SMP počítačů na trh – jejich angažovanost v tomto standardu zaručuje, že jej bude možné dlouhodobě aplikovat na jejich architekturách. (3)

2.4 Volba platformy

Pro paralelizaci nástroje pro minimalizaci logických funkcí BOOM jsem nejdříve zvolil knihovnu pthread. Hlavními důvody byla možnost maximální kontroly nad paralelizací – možnost volby libovolné úrovně granularity a hlavně schopnost paralelismu na úrovni funkcí. Mezi další důvody patří přítomnost této knihovny na všech UNIXových systémech, její velmi snadné použití na MS Windows pomocí portu pthreads-win32, a také fakt že jsem toto API již používal a shledal ho plně vyhovujícím.

Schopnost paralelismu na úrovni úloh jsem při paralelizaci BOOMu jako celku plně využil. Při paralelizaci jednotlivých fází se však ukázalo, že zejména jednoduché průchody polí apod. vyžadují velké množství kódu, který se nedá použít nikde jinde kvůli své specifičnosti pro danou úlohu, navíc mohou být problémy s rovnoměrným rozdělením práce, pokud je např. v paralelním regionu vnořený cyklus závislý na hlavním. Dalším problémem je, pokud bychom chtěli navštěvovat paralelní region cyklicky – buď budeme pokaždé vytvářet vlákna znovu, což přinese znatelně velkou režii, nebo je nutné vyvinout poměrně velké implementační úsilí na korektní a „chytré“ organizování vláken, často opět specificky pro danou úlohu.

Jako nejlepším řešením těchto problémů se nakonec ukázalo OpenMP - jednoduché průchody polem jsou paralelizovány velmi snadno, rovnoměrné rozdělení zátěže lze zařídit použitím různých plánovacích algoritmů a cyklické návštěvy paralelních regionů nepředstavují problém, protože OpenMP spouští všechna vlákna jako perzistentní, tedy korektně čekající na další přísun práce.

3 Paralelizace a možné problémy

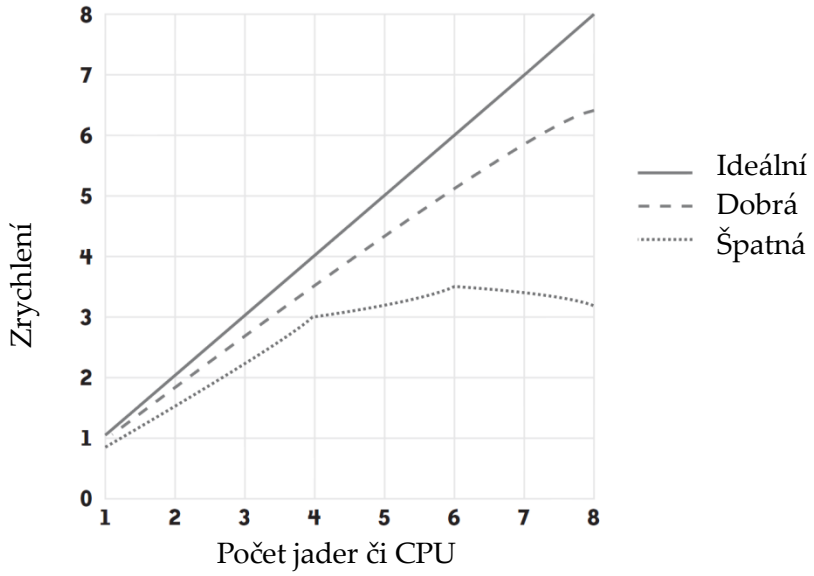
Při paralelním programování se mohou objevit problémy, které při sekvenčním programovaní nehrají žádnou roli, či vůbec nemohou nastat. Klasickým problémem je tzv. souběh (*race condition, data race*), kdy více vláken (či procesů) najednou čte a zapisuje do stejné části paměti a finální výsledek závisí na relativním časování práce jednotlivých vláken. Např. dvě vlákna A, B inkrementují společný čítač – načtou si jeho hodnotu do registru, inkrementují ji a zapíší zpátky do paměti. Přepínáním kontextu (*context switch*) však může být vlákno A uspáno těsně po načtení hodnoty čítače z paměti do registru. Vlákno B pak nějakou dobu dál inkrementuje čítač, poté je probuzeno vlákno A, inkrementuje hodnotu čítače v registru a zapíše ji do paměti, hodnota v registru však byla stará – vlákno B hodnotu čítače v paměti mezitím změnilo – do paměti se tak dostane nesprávná hodnota čítače. Podobné přístupy do paměti je třeba ošetřit synchronizačními primitivy – mutexy, semafory, podmíněnými proměnnými apod., jejich detailní popis a způsoby použití čtenář nalezne např. v páté kapitole v (4). Díky přepínání kontextu může souběh nastat i na jednoprocesorových počítačích.

Při použití synchronizačních primitiv je nutné dát pozor na uváznutí (*deadlock*), je to stav kdy vlákno A drží zamčený mutex 1 a zároveň je blokováno při čekání na uvolnění mutexu 2, vlákno B naopak drží zamčený mutex 2 a čeká na uvolnění mutexu 1. Vlákna se tak navzájem drží „v šachu“ a program tak nikdy neskončí. Uváznutí se dá předejít tím, že vlákna budou zamykat mutexy vždy ve stejném pořadí.

Opačným problémem k uváznutí je tzv. *livelock* – vlákna jsou uvězněna ve smyčce neustálého zamykání a odemykání mutexů. Často může nastat při pokusech o zabránění uváznutí, kdy pokud vlákno nemůže zamknout mutex 2, odemkne mutex 1 a poté se pokusí znovu zamknout oba. Nakonec se vláknu povede zamknout oba mutexy, může to však trvat neomezeně dlouho a do té doby budou vlákna neproduktivní.

Problémem při paralelizaci může být také špatná škálovatelnost – jader či procesorů k dispozici bude časem jen přibývat. Cílem by proto měla být aplikace efektivně využívající všech dostupných jader a s jejich přidáváním by měl výkon stále stejně narůstat. Obecně je ale těžké tohoto ideálu docílit, některé al-

goritmy a v krajním případě někdy i použitá paralelní platforma toto nedovolí. Podle (5) by měla být dobrá škálovatelnost primárním cílem paralelizace, efektivita na druhém místě a jednoduchost a přenositelnost až za nimi.

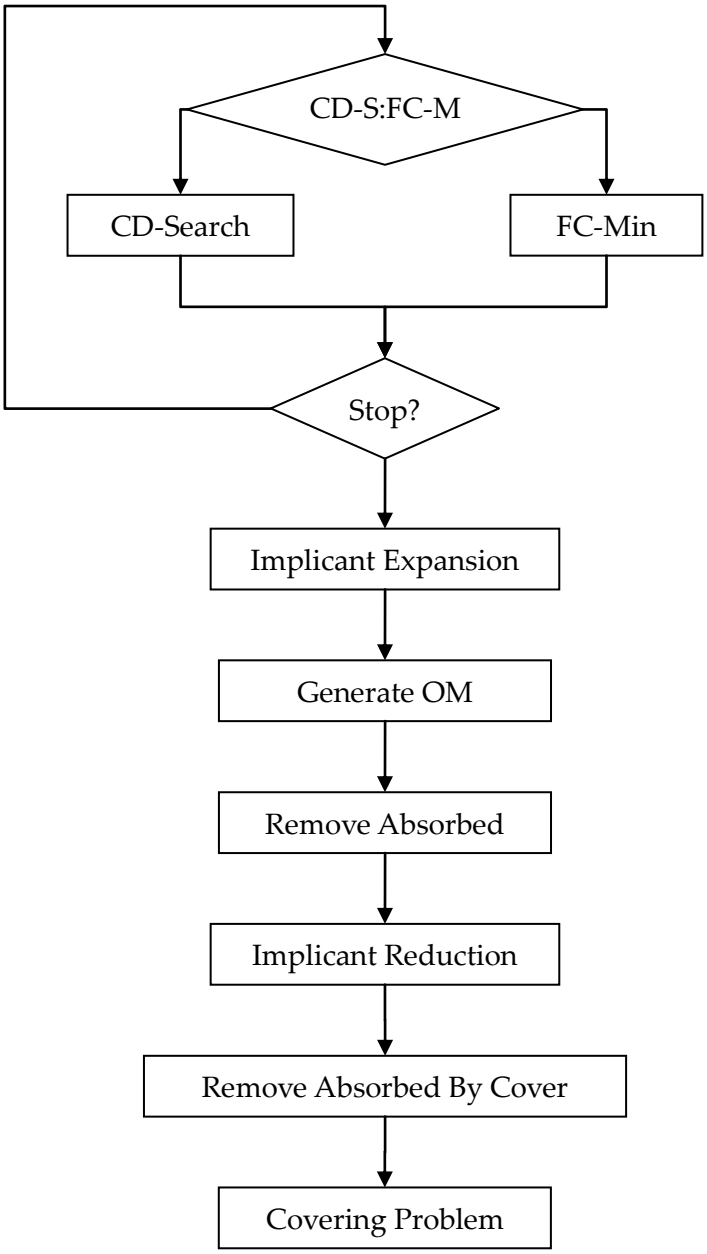


Obrázek 2: Znázornění kvality škálovatelnosti, (5)

Velký vliv na výkon paralelních aplikací má *cache* – velmi častým jevem, který znatelně snižuje výkon je „cache ping pong“ (*false sharing* či *cache thrashing*). Je to vedlejší efekt řádkové granularity a koherenčního mechanismu cache (*cache coherency*) v architekturách se sdílenou pamětí. Koherenční mechanismus označuje jednotlivé řádky cache příznakem *dirty*. Kdykoliv je řádek cache modifikován, koherenční mechanismus oznámí všem cache v ostatních jádrech se stejným řádkem, že řádek byl modifikován. V ostatních jádrech se musí řádek cache aktualizovat (buď z paměti, nebo přímo ze změněné cache). Příznak dirty se ale týká celých řádků a ne jednotlivých bytů v cache, jádra tedy nemohou vědět, že nezměněné byty v cache jsou stále bezpečné (aktuální) pro čtení. Namísto toho se musí znovu načíst celý řádek. Následně, pokud dvě vlákna modifikují různé prvky stejného řádku cache, navzájem si interferují. Pokud se to neděje často, nepředstavuje to žádný problém, pokud se to naopak děje neustále, výkonnost rapidně klesá – koherenční mechanismus je časově náročný. Někdy je proto vhodné vyplnit struktury extra *paddingem* či pouze vhodně proházet pořadí atributů tříd.

4 BOOM a jeho struktura

Algoritmus nástroje pro minimalizaci logických funkcí BOOM můžeme naznačit následujícím diagramem, poté si jej vysvětlíme a popíšeme použité datové struktury:



Obrázek 3: Schéma algoritmu BOOM

První fáze BOOMu je generování implikantů, mohou být generovány dvěma metodami – Coverage Directed Search (CD-Search) a FC-Min, které se spustí zvoleným počtem iterací a v zadaném poměru. Zatímco CD-Search generuje množinu implikantů potřebnou k pokrytí onsetu funkce (onset je množina termů logické funkce, jejichž výstupem je logická 1), FC-Min nejdříve nalezne pokrytí onsetu funkce a implikanty jsou poté generovány tak, aby pokrytí odpovídaly. Následná fáze – expanze – konvertuje implikanty na přímé. Generování výstupu (Generate OM) pouze přiřadí nedefinovaným výstupům definované hodnoty. Po odstranění redundantních implikantů (Remove Absorbed) následuje redukce (Implicant Reduction), při které jsou vytvářeny skupinové implikanty. Remove Absorbed By Cover je zobecněním Remove Absorbed, odstraňuje redundantní implikanty na základě vektorů pokrytí. Závěrem je vyřešen problém pokrytí (Covering Problem). Detailní popis algoritmu čtenář najde v (1) a (2).

4.1 Použité datové struktury

BOOM pracuje s obvody uloženými ve formátu PLA, v programu je pak obvod reprezentován třídou PLA. Obvod je tvořen seznamem (`std::list`) logických termů. Termy jsou reprezentovány třídou Term, a jejich data – vektory bitů – mají v BOOMu dvojí reprezentaci: jednak jako `std::string` s literály `'0'`, `'1'`, `'-'` a `'~'` (poslední dva jsou don't care resp. nedefinovaná hodnota) a pak také pomocí třídy bitvector s jednotlivými bity pakovanými v `unsigned int`, protože je ale potřebné uchovat čtyři stavy a ne jen dva, je nutné použít dvojici bitvectorů (`std::pair<bitvector, bitvector>`).

Při minimalizaci se vytvoří nová instance třídy PLA se stejným počtem vstupů i výstupů jako vstupní PLA a jednotlivé fáze BOOMu vždy pracují s jejím seznamem termů (implikantů), ten je na začátku pochopitelně prázdný. První fáze CDS a FCM přidávají do nového PLA implikanty vytvořené na základě vstupního PLA. Další fáze pak procházejí nové PLA a modifikují jednotlivé termy nebo přidávají nové či odstraňují redundantní.

4.1.1 Třída bitvector

Jak již bylo řečeno, reprezentace vektoru bitů je v BOOMu dvojí, hlavním důvodem je vyšší rychlost jednotlivých operací při použití první či druhé repre-

zentace. Implementace bitvectoru je také nutná kvůli nižší paměťové náročnosti, u velkých obvodů totiž může být při použití stringu problém s kapacitou paměti.

K vektoru bitů v termech se přistupuje při každé sebemenší operaci – pokud chceme znát hodnotu jednoho bitu, musí se při použití dvojice bitvectorů načíst dva bity z různých míst v paměti, velikost cache procesoru se potom může stát faktorem silně limitujícím výkon.

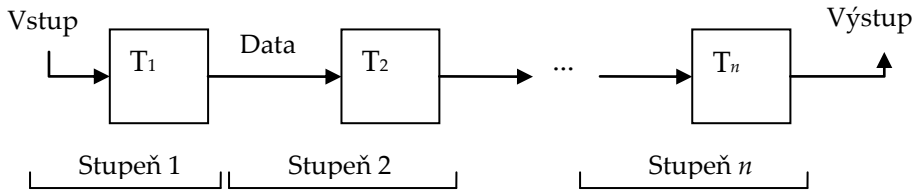
V rámci této práce jsem kvůli zlepšení výkonu implementoval bitový vektor jako jediné pole, ve kterém jsou bity uspořádány po dvojicích za sebou, načtení či modifikace jednoho bitu pak nevyžaduje přístup ke dvěma různým polím. Tato změna sice vyžaduje provedení více logických operací při čtení a modifikaci bitu než maskování jednoho bitu nutné předtím, přesto však přinesla na testovacím čtyřjádrovém stroji zhruba 30% nárůst výkonu.

Bitvector byl založen na `std::vector`, to velmi zjednodušilo běžné operace jako zvětšení velikosti a zautomatizovalo kopírující konstruktor, destruktor či operátor `=` a přineslo bezpečnost a stabilitu standardního kontejneru STL. Toto vše ale znamenalo jistou režii – přepsal jsem jej tedy na použití klasického pole s ruční alokací paměti, vlastním kopírujícím konstruktorem, operátorem `=` i destruktorem. Výsledkem byl další nárůst výkonu o zhruba 10%.

5 Paralelizace BOOMu

5.1 Proudové zpracování

Pohled na schéma algoritmu BOOM napovídá, že k paralelizaci by se dalo využít proudového zpracování (*parallelism by pipelining*). V proudovém zpracování je proud datových položek zpracováván po jednotlivých položkách sekvencí vláken T_1 až T_n, kde každé vlákno T_i vykonává nějakou specifickou operaci na každém prvku a předá ho vláknu T_{i+1}:



Obrázek 4: Proudové zpracování

Tento typ paralelního zpracování je uveden v (6), k implementaci jsou ale použity poměrně neelegantní podmíněné proměnné, navíc je ukázkový kód v neobjektovém C.

5.1.1 Třída ConcurrentQueue

Pro přenos dat mezi jednotlivými stupni pipeline využijeme frontu. Použitá fronta musí řešit klasický problém producentů a konzumentů – musí být bezpečná pro přístup více vláken. Elegantní řešení pomocí semaforů zmiňuje např. (4). Implementoval jsem tedy potřebnou strukturu pomocí semaforu a C++ STL fronty `std::queue`. Výsledná templatovaná třída ConcurrentQueue vypadá následovně:

```
template <typename T>
class ConcurrentQueue : public std::queue<T> {
private:
    pthread_mutex_t lock;
    sem_t sem;
public:
```

```
    ConcurrentQueue() {
        sem_init(&sem, 0, 0);
        pthread_mutex_init(&lock, NULL);
    }

    ~ConcurrentQueue() {
        pthread_mutex_destroy(&lock);
        sem_destroy(&sem);
    }

    void push(const T& t) {
        pthread_mutex_lock(&lock);  //mutex pro vyhradni pristup
        std::queue<T>::push(t);
        pthread_mutex_unlock(&lock);
        sem_post(&sem);    //zvysime hodnotu semaforu
    }

    void pop(T& t) {
        sem_wait(&sem);    //snizime hodnotu semaforu
        pthread_mutex_lock(&lock);  //mutex pro vyhradni pristup
        t = std::queue<T>::front();
        std::queue<T>::pop();    //puvodni front a pop sloucene
        pthread_mutex_unlock(&lock);
    }
};
```

Třída ConcurrentQueue má kromě atributů a metod zděděných od `std::queue` dva atributy – mutex `lock` zajišťující výhradní přístup k datům a semafor `sem`, který funguje jako atomické počítadlo prvků aktuálně přítomných ve frontě a zároveň blokuje vlákno, když chce vybírat z prázdné fronty. Je také potřeba si uvědomit, že čisté čtení z fronty nikdy neprobíhá, původní metody `std::queue front` a `pop` jsou sloučeny do nové `pop` (která strukturu pochopitelně modifikuje), tím je při více konzumentech zaručeno, že nebudou pracovat se stejným prvkem.

Konstruktor fronty inicializuje mutex a semafor, počáteční hodnota semaforu je nula (prázdná fronta); destruktor ruší mutex, semafor a implicitně volá destruktor nadtřídy `std::queue`. Metoda `push` zajistí výhradní přístup k frontě zamknutím mutexu, vloží data do fronty, odemkne mutex a následně zvýší hodnotu semaforu, což případně probudí jednoho z čekajících konzumentů. Metoda `pop` sníží hodnotu semaforu – v případě, kdy je po snížení hodnota semaforu menší než nula, je volající vlákno zablokováno – pokud je hodnota

semaforu větší nebo rovna nule, či je vlákno po nějaké době probuzeno, se zamknutím a následným odemknutím se odebere prvek z fronty.

5.2 Paralelizace BOOMu jako celku

V následujících kapitolách vždy prezentuji a vysvětluji původní sekvenční kód, jehož autorem je vedoucí práce. Poté s vysvětlením uvádím výsledný kód tak, jak jsem jej paralelizoval.

5.2.1 CD-Search a FC-Min

Algoritmus BOOM začíná spuštěním algoritmů CD-Search a FC-Min. Ty jsou pouštěny se zadaným počtem iterací v požadovaném poměru. Cílem je tyto fáze provádět současně jak vzájemně (pokud to poměr iterací dovoluje), tak i s dalšími fázemi.

Nově spuštěná vlákna budou pochopitelně nezávislá na hlavním vlákně, je proto potřeba jim předem vypočítat počty iterací. Celkový počet iterací označíme i, procentuální poměr metod r. Počet iterací CD-Search a FC-Min pak snadno spočítáme výrazem

$$i_{\mathrm{CD-S}} = \frac{i \times (100 - r)}{100} \quad \text{resp.} \quad i_{\mathrm{FC-M}} = \frac{i \times r}{100}$$

Následně můžeme spustit oba algoritmy, počty iterací jsou atributy tříd CD_Search a FC_Min:

```
if (options.CD_FC != 100) {    //pokud neni CDS zakazana
    pthread_create(&cdt, NULL, thread_wrapper<CD_Search,
        &CD_Search::Run>, CDS_alg);
}
if (options.CD_FC != 0) {    //pokud neni FCM zakazana
    pthread_create(&fcmt, NULL, thread_wrapper<FC_Min,
        &FC_Min::Run>, FC_alg);
    pthread_create(&fcmiet, NULL, thread_wrapper<Expand,
        &Expand::Run>, IE_FC_alg);
}
```

Vlákna jsou startována funkcí `pthread_create`, jejíž detailní popis může čtenář najít v (7). `Pthread_create` vyžaduje jako třetí argument ukazatel na funkci (konkrétně `void *(*thread_func)(void *)`), metody tříd však této signatuře

neodpovídají a je nutné je obalit do klasických funkcí. K tomuto účelu jsem vytvořil templatovanou funkci `thread_wrapper`, která zavolá požadovanou metodu objektů uvedených jako čtvrté argumenty. Všechna vlákna v paralelizovaném BOOMu jsou startována tímto způsobem.

```
template<class T, void(T::*member_function)()>
void * thread_wrapper(void * p) {
    (reinterpret_cast<T *>(p)->*member_function)();
    return NULL;
}
```

Detailní popis CD-Search a FC-Min je v (1), resp. (2), v sekvenční verzi jsou jejich výstupy (implikanty) přidávány do spojového seznamu, který je pak zpracováván v dalších fázích BOOMu. Místo do spojového seznamu však budeme implikanty přidávat do dvou front ConcurrentQueue – do generování výstupu (Generate OM) a do fáze Implicant Expansion. Proces tak bude ekvivalentní se sekvenčním, IE totiž vytváří nové implikanty na základě vstupních – do generování výstupu pak jdou dohromady se vstupními. Všechny fronty mezi jednotlivými fázemi jsou vytvořené na úrovni hlavního vlákna, vlákna jednotlivých fází k nim přistupují přes ukazatele. Kvůli paralelnímu provádění dalších fází je nutné v konstruktorech CDS a FCM vygenerovat kompletní onset a offset funkce.

5.2.2 Implicant Expansion

Fáze IE může probíhat několika metodami – Exhaustive, Fast, Multiple a Scatter Expand. První tři jmenované pracují přibližně na stejném principu, Scatter Expand je specifický pro FC-Min. Seznam implikantů ve třídě PLA je definován jako typ TermList, instance PLA v kódu jsou obvykle pojmenovány `implpool`. Ukazatel na samotný seznam získáme metodou `getData`. Podívejme se na velmi zjednodušený kód sekvenční verze prvních tří metod:

```
typedef std::list<Term> TermList;
TermList * data = implpool->getData();    //ziskani dat
for (ov = 0; ov < outputs; ov++) {    // pro kazdy vystup
    for (ti = data->begin(); ti != data->end(); ++ti) {
        t = *ti;   // a kazdy term
        ...
        for (i = 0; i < inputs; i++) {
            ...
            //po provedeni operaci pridame term do seznamu
```

```
            implpool->AppendTerm(t);
        }
    }
}
```

Cílem paralelizace BOOMu jako celku pomocí proudového zpracování je de facto odstranit z jednotlivých fází závislost na seznamu implikantů a transformovat ji na získání termu z ConcurrentQueue, zpracování a následné předání do další ConcurrentQueue. V předcházejícím úryvku vidíme, že operace se provádějí s každým výstupem a všemi termy. Všimněme si, že první dva `for` cykly lze navzájem vyměnit bez změny funkčnosti. Protože mezi jednotlivými iteracemi ani jednoho cyklu nejsou závislosti, transformoval jsem kód na použití ConcurrentQueue:

```
typedef ConcurrentQueue<Term> TermQueue;
TermQueue * tq_in, * tq_out;
while (true) {
    tq_in->pop(t);    //vyberu term z fronty
    if (t.invalid()) break;    //je term ukoncovaci?
    for (ov = 0; ov < outputs; ov++) {
        ...
        for (i = 0; i < inputs; i++) {
            ...
            // po zpracovani predam term do dalsiho stupne
            tq_out->push(t);
        }
    }
}
```

Zavedl jsem také definici typu TermQueue jako ConcurrentQueue<Term>, definice platí pro celý program a instance ve všech fázích mají obvykle jméno `tq`, `tq_in` či `tq_out`.

V kódu v nekonečné smyčce odebíráme prvky z fronty (do ní z druhé strany přidávají CD-Search a FC-Min), pokud je fronta prázdná, vlákno je blokováno. Expandované termy předáváme přes další ConcurrentQueue do generování výstupu. Po skončení CD-Search a FC-Min už do fronty nebudou přibývat další termy a vlákno provádějící IE by tedy trvale spalo. Tomu jednoduše zabráníme tak, že poté co hlavní vlákno provede `pthread_join` s vlákny CDS a FCM, přidá do fronty prázdný term s nastaveným ukončovacím příznakem `invalid`.

Speciálním případem je metoda expanze je Scatter Expand, použitá vždy společně s FC-Min. Její výstup jde vždy do generování výstupu, může ale také jít ještě do jedné ze tří standardních metod IE. Původní sekvenční kód:

```
vector<bool> tried;
unsigned int trials, n;
n = implpool->getTerms() * implpool->inputs();
tried.assign(n, false);
trials = n;
do {
    pos = rand() % n;    //nahodna pozice
    do {
        if (!tried[pos]) {    //byl jiz term testovan?
            ti = implpool->getTermByIndex(pos /
                implpool->inputs());  //vyber termu podle indexu
            termpos = pos % implpool->inputs();   //vyber vstupu
            tried[pos] = true; //oznacime jako jiz otestovany
            trials--;
            if (!ti->isLit_dc(termpos)) {
                c = ti->getLit(termpos);
                ti->setLit_dc(termpos);  //zkusime nastavit DC
                if (!source->isImplicant(*ti))
                    //vratime puvodni pokud neni implikantem
                    ti->setLit(termpos, c);
                else ti->origin |= 4;
            }
            break;
        } else pos = (pos + 1) % n;
    } while (trials > 0);
} while (trials > 0);
```

Vidíme, že vnitřní cyklus proběhne náhodným vybíráním pro každý term a každou vstupní proměnnou, celkový počet testů je počet termů × počet vstupních proměnných. V poli příznaků `tried` je zaznamenáno otestování daného vstupu u daného termu. Náhodné pořadí termů bude nově dáno tím, jak budou termy přicházet, testování implikace náhodným pořadím vstupních proměnných jsem zachoval:

```
while (true) {
    tq_in->pop(t);    //odebereme z fronty
    if (t.invalid()) break;    //konec?
    do {
        i = rand() % implpool->inputs();    //pro nahodny vstup
        if (!tried[i]) {
```

```
            ...    //stejne
        }
    } while (trials);
    tq_out_gom->push(t);   //predani do GenerateOM
    if (tq_out_ie)
        tq_out_ie->push(t);   //predani do IE
    trials = implpool->inputs();
    tried.assign(implpool->inputs(), false);
}
```

5.2.3 Generování výstupu

Při generování výstupu se pouze nedefinované výstupy nastaví na definované hodnoty. Transformace průchodu seznamem na proudové zpracování je triviální:

```
TermList::iterator ti = implpool->getData()->begin();
for ( ; ti != implpool->getData()->end(); ++ti) {
    for (i = 0; i < outputs; i++) {
        //kazdy vystup kazdeho termu
        //nastavime na 0 nebo 1 podle implikace
        if (ti->isOM_undef(i)) {
            if (source->isImplicant(*ti, i))
                ti->setOM_1(i);
            else
                ti->setOM_0(i);
        }
    }
}
```

K verzi využívající proudového zpracování není třeba nic dodávat:

```
while (true) {
    tq_in->pop(t);
    if (t.invalid()) break;
    for (i = 0; i < outputs; i++) ...    //stejne
    tq_out->push(t);
}
```

5.2.4 Remove Absorbed

Metoda Remove Absorbed probíhá pro vstupní PLA také ještě před CDS a FCM, v té chvíli ale nelze zakomponovat do proudového zpracování, protože

až do skončení metody není jasné, které termy v seznamu zůstanou. Remove Absorbed odstraňuje redundantní termy ze seznamu implikantů. V případě Remove Absorbed je to aplikace pravidla a + ab = a.

```
TermList::iterator ti = data.begin(), ti1;
do {
    ti1 = ti;    //testovany term
    ++ti1;       //zacneme testovat od nasledujiciho
    deleted = false;
    while (ti1 != data.end()) {
        if (ti->contains(*ti1))
            //je absorbovat -> vymazat
            ti1 = DeleteTerm(ti1);
        else if (ti1->contains(*ti)) {
            //testovany term je absorbovan
            ti = DeleteTerm(ti);
            deleted = true;
            break;    //znovu vnejsi cyklus
        } else ++ti1;    //pouze posunuti
    }
    if (!deleted) ++ti;
} while (ti != data.end());
```

V kódu již nevidíme `implpool`, Remove Absorbed je totiž metoda třídy PLA, metody `AppendTerm` a `DeleteTerm` tedy přímo pracují se seznamem uloženým v `data`.

Sekvenční kód pro každý term prochází seznam implikantů a kontroluje, zda term absorbuje jiné termy v seznamu a také zda termy v seznamu absorbují testovaný term. Termy absorbované testovaným termem jsou ze seznamu vyřazeny, stejně tak je vyřazen testovaný term, pokud je absorbován některým ze seznamu. Můžeme lehce spočítat, že v nejhorším případě (žádný term není vyřazen) je provedeno $n \times (n - 1) \times 2$ ověření absorpce.

Při proudovém zpracování ale žádný seznam implikantů k dispozici nebude, což se překvapivě ukáže jako velmi efektivní výhoda. V proudové verzi seznam termů od začátku vytvářím a ověření absorpce provádím jen pro nově příchozí termy. Tím se pro nejhorší případ sníží počet ověření přibližně na polovinu, přesněji na $n \times (n + 1)$:

```
while (true) {
    tq_in->pop(t);
    if (t.invalid()) break;
```

```
    deleted = false;
    for (ti = data.begin(); ti != data.end(); ) {
        if (t.contains(*ti)) ...
        ...    //analogicky
    if (!deleted)
        AppendTerm(t);
}
```

Za RA je nutné umístit bariéru, nefunguje totiž jako filtr, který výstup okamžitě po zpracování posílá dál – seznam „přeživších“ implikantů je finální až po ověření posledního příchozího termu, navíc se tak vyhnu masivní synchronizaci nutné pro souběh dalších fází. Zbylé fáze BOOMu tedy nebudou startovat, dokud neskončí všechny předchozí včetně Remove Absorbed.

5.2.5 Implicant Reduction

Fáze Implicant Reduction může probíhat třemi metodami – Fast Reduce, Multiple Reduce a Exhaustive Reduce. U všech tří jde o průchod spojovým seznamem:

```
TermList::iterator ti = implpool->getData()->begin();
for ( ; ti != implpool->getData()->end(); ++ti) {
    t = *ti;
    for (i = 0; i < implpool->inputs(); i++)
        ...
        if (...) {
            // po zpracovani pridame na konec
            implpool->AppendTerm(t);
        }
}
```

Pro transformaci na proudové zpracování je však potřeba zajistit dvě věci – v následující fázi Remove Absorbed By Cover se jednotlivým implikantům v seznamu vytvářejí vektory pokrytí (jsou to instance třídy bitvector) a velmi snadno by při manipulaci více vlákny nastaly problémy se souběhem. Navíc je nutné zajistit, aby byly vektory pokrytí vytvořeny zvlášť pro stávající a zredukované implikanty – vyřešil jsem to pomocí kopie celého PLA pro použití v Implicant Reduction. Kopie však musí proběhnout před vytvářením vektorů pokrytí, místo v další fázi tak spustím `GenerateCoverVectors` už v IR.

Vytvoření kopie celého PLA přinese určitou malou režii, nebude však nutné kopírovat vektory pokrytí do pipeline. Pokud bychom se ale chtěli kopírování

celého PLA vyhnout, tak jediným řešením by byla synchronizace na úrovni každého implikantu pomocí mutexů, což by přineslo ještě větší režii v podobě nemožnosti vláken pokračovat v práci, kopírování mutexů a de facto dvojího vytváření vektorů pokrytí. Obecně platí, že kopírování dat je velmi často lepším řešením než soupeření vláken o výhradní přístup.

```
PLA implpool2(*implpool);    //vytvoreni kopie celeho PLA
cp->GenerateCoverVectors();
for ( ti = implpool2->getData()->begin(); ti != ti1; ++ti ) {
    t = *ti;
    ...    //stejne
    tq->push(t);
}
```

5.2.6 Remove Absorbed By Cover

Fáze Remove Absorbed By Cover je zobecněním Remove Absorbed. V sekvenční verzi jde opět průchod seznamem implikantů, napřed jsou ale generovány vektory pokrytí. Při průchodu se pro každý implikant prochází seznam implikantů a kontroluje se, zda vektor pokrytí testovaného implikantu obsahuje vektory pokrytí ostatních implikantů v seznamu a vice versa. Pokud je vektor obsažen jiným, je implikant odstraněn ze seznamu a dojde k posunutí daného iterátoru. V případě, že se vektory obsahují navzájem, je odstraněn implikant s menším počtem don't care bitů na vstupu (ten se zjistí pomocí metody `getDim`):

```
TermList::iterator ti1, ti2;
bool abs1, abs2;
bool deleted;

ti1 = implpool->getData()->begin();
if (ti1->getCoverVector()->empty())
    GenerateCoverVectors();    //vytvoreni vektoru pokryti
do {
    deleted = false;
    ti2 = ti1;    //testovany term
    ti2++;        //kontroluji od nasledujiciho
    while (ti2 != implpool->getData()->end()) {
        abs1 = true;
        abs2 = true;
        //testovani a nastaveni flagu
        if (!ti1->getCoverVector()->contains(
```

```
                *ti2->getCoverVector()))
            abs2 = false;
        if (!ti2->getCoverVector()->contains(
                *ti1->getCoverVector()))
            abs1 = false;
        if (abs1 && abs2) {
            //vymazani termu s mene DC bity
            if (ti1->getDim() >= ti2->getDim())
                ti2 = implpool->DeleteTerm(ti2);
            else {
                ti1 = implpool->DeleteTerm(ti1);
                deleted = true;
                break;
            }
        }
        //vymazani termu podle flagu
        else if (abs1 && !abs2) {
            ti1 = implpool->DeleteTerm(ti1);
            deleted = true;
            break;
        }
        else if (!abs1 && abs2)
            ti2 = implpool->DeleteTerm(ti2);
        else ti2++;    //pouze posun
    }
    if (!deleted) ti1++;    //pouze posun pokud jsem nemazal
} while (ti1 != implpool->getData()->end());
```

Fázi Remove Absorbed By Cover je potřeba provádět pouze pro implikanty nově přicházející z Implicant Reduction. Přidávání a mazání implikantů ze seznamu nepředstavuje žádný synchronizační problém – bude tak činit pouze vlákno s Remove Absorbed By Cover, protože IR má vlastní kopii. Výsledný paralelní kód bude tedy vypadat následovně:

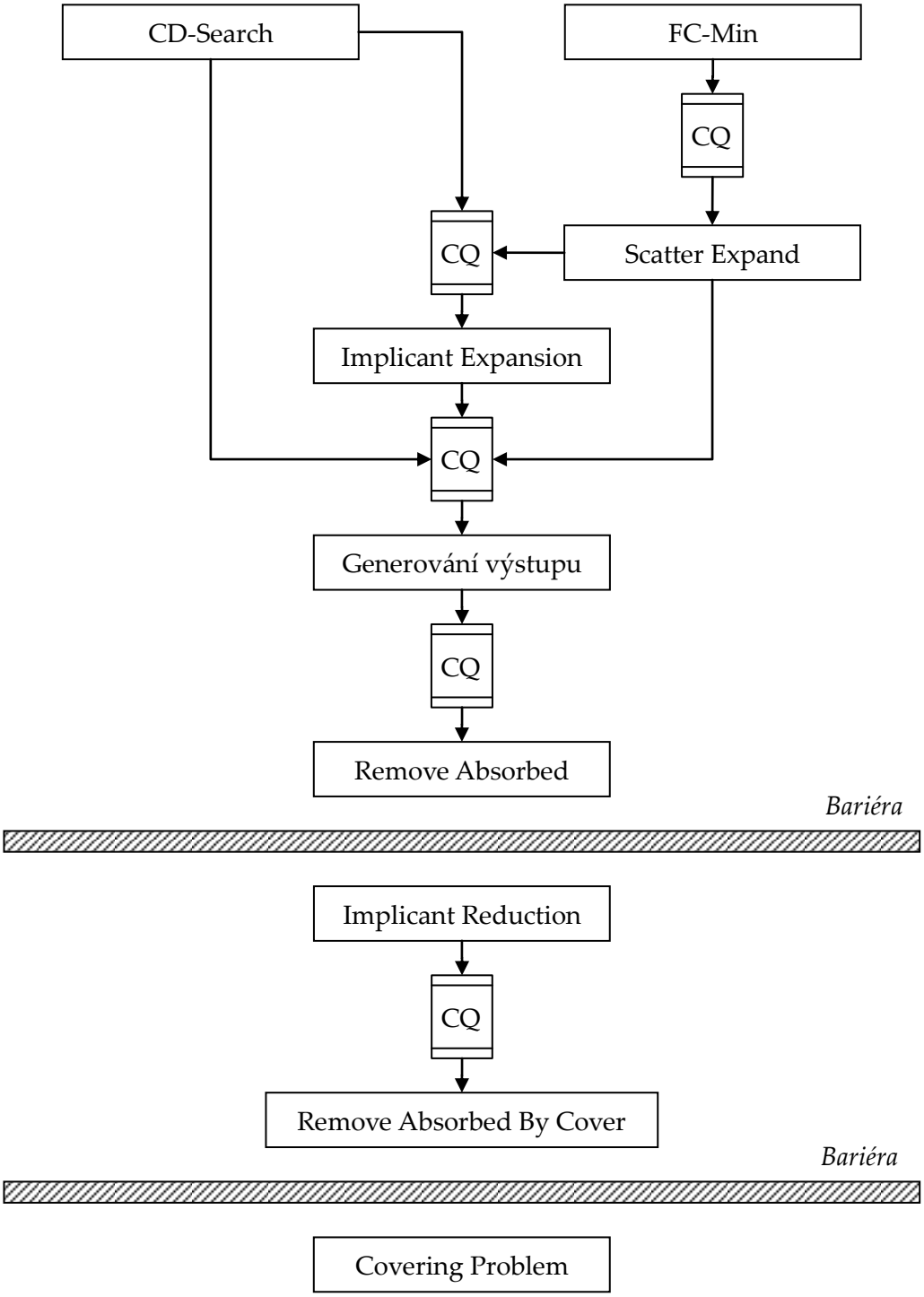
```
while (true) {
    tq_ir_rabc->pop(t);
    if (t.invalid()) break;
    GenerateCoverVector(t);
    deleted = false;
    ti = implpool->getData()->begin()
    while (ti != implpool->getData()->end()) {
        abs1 = abs2 = true;
        if (!t.getCoverVector()->contains(
                *ti->getCoverVector()))
            abs2 = false;
```

```
        if (!ti->getCoverVector()->contains(
                *t.getCoverVector()))
            abs1 = false;
        if (abs1 && abs2) {
            //vymazani termu s mene DC bity
            if (t.getDim() >= ti->getDim())
                ti = implpool->DeleteTerm(ti);
            else {
                //neni treba mazat, nastavim jen priznak
                //ze se nebude pridavat
                deleted = true;
                break;
            }
        }
        //vymazani termu podle flagu
        else if (abs1 && !abs2) {
            //opet neni treba mazat
            deleted = true;
            break;
        }
        else if (!abs1 && abs2)
            ti = implpool->DeleteTerm(ti);
        else ti++;    //pouze posun
    }

    }
    if (!deleted) {    //nebyl absorbovan, pripojim na konec
        implpool->AppendTerm(t);
    }
}
```

5.2.7 Celkové schéma

Následující fáze, tedy řešení problému pokrytí, již nemůže probíhat paralelně s předchozími fázemi, potřebuje znát celý finální seznam implikantů. Celkový průběh BOOMu paralelizovaného jako celek pomocí proudového zpracování ilustruji následujícím diagramem, jednotlivé fáze v obdélnících před bariérou běží paralelně, implikanty si předávají přes ConcurrentQueue (značené CQ), stejně tak běží paralelně dvě fáze mezi bariérami a nakonec je vyřešen problém pokrytí:



Obrázek 5: Schéma BOOMu paralelizovaného jako celku

5.3 Paralelizace jednotlivých fází

Jak již bylo řečeno při výběru platformy, původní záměr byl použít při paralelizaci pouze jednu vybranou paralelní platformu, nakonec se však pro paralelizace jednotlivých fází ukázalo použití OpenMP jako nejvhodnější – těla jednotlivých fází jsou veskrze snadno paralelizovatelné `for` cykly, či se na ně dají poměrně jednoduše transformovat.

5.3.1 Remove Absorbed

RA probíhá také ještě před začátkem CD-Search či FC-Min a v té chvíli nelze zakomponovat do proudového zpracování. K její paralelizaci jsem použil OpenMP, držel jsem se však normy 2.5, novější norma 3.0 totiž není podporována Microsoft C++ kompilátorem a takto využijeme paralelismu i s ním.

Podle normy 2.5 bohužel nelze jednoduše procházet `for` cyklem STL kontejnery, které nemají stejný přístup jako klasické pole, čili ani `std::list`, ve kterém je uložen seznam termů. Jak tedy projít seznam? Vytvořím si nový vektor s ukazateli na jednotlivé prvky seznamu, podobně jako v (8). Sekvenční kód byl již uveden dříve, ukážeme si tedy výsledný paralelní kód a rozebereme si jej:

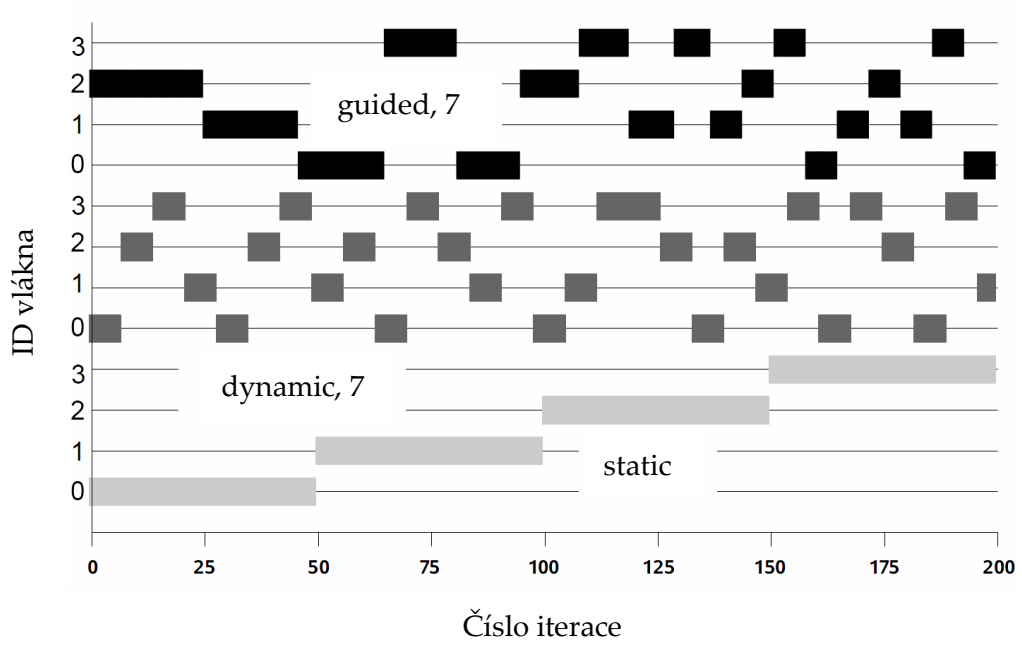
```
vector<Term *> v;    //vektor ukazatelu na termy v seznamu
v.reserve(data.size());    //rezervace potrebne velikosti
for (ti = data.begin(); ti != data.end(); ++ti)
    v.push_back(&(*ti));    //naplneni ukazateli
int n = v.size(), i, j;
//paralelni for cyklus se statickym planovanim po 10
#pragma omp parallel for schedule(static, 10) \
    shared(v, n) private(i, j)
for (i = 0; i < n; i++) {
    if (v[i]->invalid()) continue; //je platny? jinak dalsi kolo
    for (j = i + 1; j < n; j++) {  //od nasledujiciho do konce
        if (v[i]->invalid()) break;
        //testovany je neplatny? pak do vnejsiho cyklu
        if (v[j]->invalid()) continue;  //jedu dal
        if (v[i]->contains(*v[j]))
            v[j]->invalidate();    //zneplatneni jednoho
        else if (v[j]->contains(*v[i])) {
            v[i]->invalidate();    //nebo druheho
            break;    //pak skok do vnejsiho
        }
```

```
    }
}    //nasleduje fyzicke odstraneni neplatnych
for (i = 0, ti = data.begin(); i < n && ti != data.end(); ++i)
    if (v[i]->invalid()) ti = DeleteTerm(ti);
    else ++ti;
```

Po vytvoření vektoru a rezervování dostatečné velikosti je vektor naplněn ukazateli na jednotlivé prvky seznamu. Direktiva překladače `omp parallel for` zajistí paralelní průchod vektorem.

Následující klauzule jsou velmi důležité pro správnou funkčnost – pomocí klauzulí `shared` a `private` označíme jako proměnné jako sdílené resp. privátní. Sdílené proměnné nejsou chráněny žádnou implicitní synchronizací, tu musí zařídit programátor, privátní proměnné nemají při startu paralelního regionu definovanou hodnotu (v našem případě to nepředstavuje žádný problém). Velikost vektoru a vektor sám tedy označím jako sdílené a počítadla průchodů `i` a `j` jako privátní.

Klauzule `schedule` určuje, který plánovací algoritmus se použije. Na výběr jsou tři možnosti – `static` (implicitní), `dynamic` a `guided`. Následující obrázek ilustruje rozdělení 200 iterací podle plánovacích algoritmů pro 4 vlákna.



Obrázek 6: Rozdělení iterací mezi vlákna podle plánovacího algoritmu, (9)

U `static` je třeba rozlišovat použití s číselným parametrem a bez něj. Pokud jej neuvedeme, je vláknům přidělen podíl iterací proporcionální k počtu vytvořených vláken. Na obrázku vidíme, že čtyři vlákna si rozdělila 200 iterací po 50. Výhodou je téměř nulová režie, nevýhodou je vhodnost použití jen pro jednoduché průchody, ve kterých je snadné docílit rovnoměrného rozložení zátěže. V našem případě by ale byla zátěž rozložena velice nerovnoměrně, protože vnitřní cyklus je závislý na vnějším – `j` prochází od `i + 1` do konce pole, pro např. čtyři vlákna by první testovalo čtvrtinu prvků s celým polem, druhé s třemi čtvrtinami, třetí s polovinou a poslední s pouhou čtvrtinou prvků. Pokud ale použijeme parametr, např. 10, budou vláknům iterace přiděleny metodou *round robin* po 10 iteracích, čímž v našem případě docílíme rovnoměrného rozdělení práce.

Při použití `dynamic` si vlákna dynamicky odebírají iterace po zadaném počtu tak, jak jsou k dispozici. Po zpracování zadaného počtu iterací si vlákno vždy vyžádá další dávku iterací (*chunk of iterations*). Pokud není specifikován počet odebíraných iterací, odebírají se po jedné. Při `guided` se iterace odebírají ze začátku po větších dávkách, velikost dávky potom proporcionálně klesá až na hodnotu parametru.

Vraťme se ale k RA, princip metody zůstává stejný – v cyklu procházíme seznam implikantů (tentokrát ale paralelně) a odstraňujeme z něj redundantní. Jakým způsobem z něj ale budeme odstraňovat položky, když je ostatní vlákna mohou v tu chvíli zpracovávat? Vyřeším to pomocí tzv. *lazy* synchronizace, která spočívá v pouhém logickém odstranění ze seznamu, v tomto případě nastavením příznaku `invalid`. Fyzicky implikanty ze seznamu odstraním jediným průchodem seznamu po skončení paralelního procházení. O *lazy* synchronizaci se může čtenář dozvědět více např. v kapitole o spojových seznamech v (10).

Remove Absorbed chceme také paralelizovat i v proudovém zpracování. Princip bude opět stejný, s příchozím implikantům paralelně procházím seznam implikantů a při tom logicky odstraňuji absorbované:

```
Term t;
bool deleted;
deque<Term *> d;    //pouziti deque pro rychle pridavani
int n, i;
while (true) {
    tq_in->pop(t);
```

```
    if (t.invalid()) break;
    deleted = false;
    n = d.size();
    //vystacim si s defaultnim statickym planovanim
    #pragma omp parallel for \
        shared(n, d, deleted, t) private(i)
    for (i = 0; i < n; i++) {
        // flush aby byl priznak deleted aktualni
        #pragma omp flush(deleted)
        if (!deleted) { //podminka vykonavani cyklu
            if (d[i]->isUnwanted())
                continue;//pokud narazim na oznaceny, dalsi kolo
            if (t.contains(*d[i])) {
                d[i]->setUnwanted();   //logicke odstraneni
            } else if (d[i]->contains(t)) {
                //pokud testovany nebudu pridavat
                //nastavim priznak a provedu flush
                deleted = true;
                #pragma omp flush(deleted)
            }
        }
    }
    if (!deleted) {
        //mohu pridat - pripojim do seznamu i do pole ukazatelu
        TermList::iterator ti = AppendTerm(t);
        d.push_back(&(*ti));
    }
}
//fyzicke odstraneni oznacenych implikantu
for (i = 0, ti = data.begin(); i < n && ti != data.end(); i++)
    if (d[i]->isUnwanted()) ti = DeleteTerm(ti);
    else ++ti;
```

Jako pole ukazatelů jsem použil kontejner `std::deque`, umožňuje totiž časté přidávání na konec bez realokací (data nemusí být souvisle za sebou jako u `std::vector`) a poskytuje stejný přístup jako klasické pole pomocí operátoru `[]`. Nevýhodou OpenMP při `parallel for` je nemožnost používat `break` v paralelizovaném cyklu (ve vnořeném to už ale jde), pokud tedy zjistíme, že term do seznamu nepatří, nelze jednoduše přerušit vykonávání cyklu ve všech vláknech. Tento problém jsem vyřešil pomocí sdílené proměnné `deleted`, vykonávání těla cyklu jí podmíníme. Direktiva `flush` pak zajistí, že všechna vlákna budou mít stejnou (aktuální) hodnotu proměnné (překladač vynutí zápis dat z registru do paměti či vypláchnutí bufferů), ostatní vlákna si pak proměnnou načtou znovu z paměti a případně dokončí cyklu „naprázdno“.

5.3.2 CD-Search a FC-Min

Paralelizace CD-Search pomocí OpenMP je triviální – hlavní cyklus probíhá pro každý výstup nezávisle. Zdrojový kód nemá význam uvádět, jde opravdu jen o připsání `#pragma omp parallel for`. Paralelizace FC-Min by nebyla triviální, naprosto však postrádá smysl – z profilování vyplývá že FC-Min spotřebuje pod 1% procesorového času potřebného pro celou minimalizaci, je tak dost dobře možné že režie paralelizace by naopak způsobila zpomalení celého procesu.

5.3.3 Implicant Expansion

Fázi IE jsem paralelizoval velmi jednoduše – opět se jednalo pouze o přidání direktivy `parallel for` ke hlavnímu cyklu, který probíhá pro každý výstup nezávisle. Navíc jsem použil direktivu `firstprivate`, má stejný význam jako `private`, jen s tím rozdílem, že ve všech vláknech bude proměnná inicializovaná kopií z hlavního vlákna. IE by díky datovému paralelismu mohla být dobrým kandidátem na použití SIMD instrukcí. Některé operace (ne jen v IE) prováděné s bitvectory by mohly být pomocí SIMD rapidně urychleny. Pokud bychom ale v programu přistoupili k ruční optimalizaci v assembleru, přišli bychom o přenositelnost – z tohoto důvodu je lepší nechat použití SIMD instrukcí na kompilátoru, např. pro x86 jsou instrukční sady SSE a SSE2 obsahující skupinové logické operace kompilátory široce podporovány.

5.3.4 Generování výstupu

Fáze generování výstupu se sama o sobě již v podstatě nedá paralelizovat, granularita je již na nejjemnější úrovni (*fine-grained*). Tato fáze funguje čistě jako filtr, mohu tedy pustit libovolný počet vláken s touto metodou. I zde jsou podle mě dobré podmínky pro použití SIMD instrukcí.

5.3.5 Implicant Reduction

Všechny tři metody, kterými může redukce probíhat, jsem paralelizoval pomocí pomocného pole ukazatelů na prvky v seznamu implikantů a následného přidání kouzelné direktivy OpenMP `parallel for`. I zde jsou implikanty zpracovávány nezávisle na sobě a pro některé operace by se daly použít SIMD instrukce.

5.3.6 Remove Absorbed By Cover

Remove Absorbed By Cover jsem paralelizoval přesně analogicky jako Remove Absorbed v proudovém zpracování, včetně použití direktivy `flush`, opět nemá význam zde vypisovat výsledný kód, čtenář jej najde na přiloženém médiu. Paralelizoval jsem také vytváření vektorů pokrytí, opět pomocí pomocného pole ukazatelů a kouzelné direktivy `parallel for`.

5.3.7 Covering Problem

Řešení problému pokrytí je u některých vybraných obvodů velmi rychlé – v poměru s ostatními fázemi je jeho spotřeba CPU času téměř zanedbatelná. Pro většinu obvodů je to ovšem pro procesor suverénně nejintenzivnější část minimalizace. V BOOMu jsou implementovány tři heuristické metody pro řešení problému pokrytí: Aura-II, Lagrangeovská metoda větví a hranic a MCLC (*Most Covering, Least Covered*) heuristika. První dvě jmenované hledají exaktní řešení, jsou to složité rekurzivní algoritmy a jejich paralelizace je svým rozsahem mimo záběr této práce. Profilování metody MCLC ukázalo, že hlavním cílem paralelizace by měla být metoda `ColDominanceRemoval`. Metodu jsem paralelizoval velice podobně jako Remove Absorbed před pipeline, pouze s jedním rozdílem – v položkách seznamu není žádný příznak, kterým bychom mohli označovat odstraněné položky. Tento problém jsem vyřešil pomocí dvou polí ukazatelů, přičemž logické odstraňování probíhá nastavením položky na `NULL` vždy z jednoho pole a čtení (dereference) naopak z druhého – tím zaručím, že nikdy není dereferencován nulový ukazatel a přesto mohu označovat položky jako neplatné. Správného rozložení zátěže je opět dosaženo pomocí `schedule(static, 10)`. Zdrojový kód čtenář najde na přiloženém médiu.

6 Testování na zkušebních úlohách

6.1 Kompilace

Velký vliv na výkon BOOMu má použitý překladač a správné nastavení parametrů kompilace. Microsoft C++ kompilátor je defaultní pro Windows, nevyužívá makefile, ale parametry jsou nastaveny ve vlastnostech projektu Visual Studia (projekt je samozřejmě na přiloženém médiu). Na GNU/Linuxu jsem testoval tři překladače: GNU g++, open64 a Intel C++ Compiler. G++ je standardem, nedokáže však naplno využít vlastností procesoru jako open64 či Intel. Nejlepšího výkonu docílíme pomocí parametrů `-O3 -march=native -flto -fopenmp` při kompilaci a při linkování navíc `-lpthread -fwhole-program -fipa-pta`. Tyto parametry zajistí použití všech instrukcí podporovaných procesorem, link-time optimalizaci na úrovni celého programu (ne jen modulů), podporu OpenMP a interprocedurální analýzu. Překladač open64 vzniká pod záštitou AMD, na procesorech této firmy pak nabízí třeba i o třetinu rychlejší kód. Optimální parametry pro kompilaci jsou `-Ofast -march=auto -IPA:max_jobs=0 -apo -mso -mp`, bohužel ale implementace OpenMP má problém s OpenMP klauzulí `default(none)`. Intel C++ Compiler je na rozdíl od g++ a open64 komerční produkt, pro osobní nekomerční použití je ale k dispozici zdarma. Přirozeně nejvíce optimalizuje pro procesory Intel, nejlepší výkon dosáhneme pomocí flagů `-fast -openmp`. Na médiu jsou makefile pro všechny tři kompilátory.

6.2 Testy

Následující testy jsem provedl na čtyřjádrovém (HyperThreadingových jednotek je 8) Intel Xeon E5430, taktovaném na 2.66 GHz, s 16 GB RAM a 64 bit Linuxem 2.6.34.7. Jako kompilátor jsem použil Intel C++ Compiler verze 12.0.2.

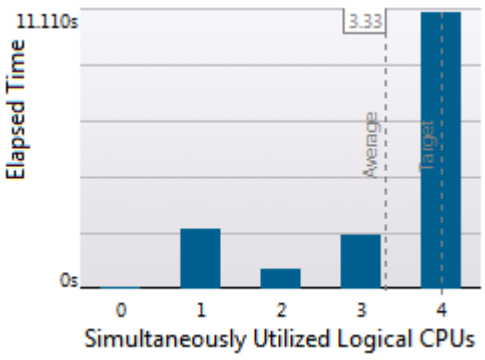
V následující tabulce jsou uvedeny jednotlivé testované obvody, jsou k nim připsány počty vstupů, výstupů a počet termů, kterými je daná funkce specifikována. Poslední tři sloupce jsou reálný (nikoliv CPU) čas dosažený původním sekvenčním BOOMem, reálný čas dosažený výsledným paralelizovaným nástrojem a koeficient zrychlení.

Obvod	i / o / n	Sekv. čas [s]	Par. čas [s]	zrychlení
1000x1000	1000/1/ 1000	38,023	8,881	4,28
100x100	100/1/100	0,022	0,405	0,05
10x10x600_30idc	10/10/600	4,337	2,011	2,16
20x5x200_10%	20/5/200	1,641	0,406	4,04
50x50x150_10%	50/50/150	323,711	18,826	17,19
apex5	117/88/2710	15,161	2,472	6,13
cordic	23/2/1206	17,410	3,000	5,80
pdc	16/40/2810	24,617	1,510	16,50
prom1	9/40/3136	84,809	9,634	8,80
50x10x1000_30%	50/10/1000	6 910,946	66,254	104,30
near-taut1	25/1/50000	1 138,933	186,985	6,09

Výsledky jsou na první pohled překvapující, existuje však pro ně poměrně jednoduché vysvětlení. Zrychlení větší než 4 se dá přisuzovat jednak Hyper-Threadingu – kdy na každém fyzickém jádře jsou dvě virtuální jádra, ty ovšem nepracují plnohodnotně paralelně, pouze při *stallu* jednoho z nich může druhé pracovat, nebo jej má na svědomí cache – upravený bitvector plní svou roli, jak má a např. termy se pak mohou celé vejít do jedné cache line. Evidentní je extrémní závislost zrychlení na parametrech vstupního obvodu.

Naopak v druhém řádku vidíme velmi špatný výsledek, je to způsobeno tím, že pro tak malé obvody jako 100x100 představuje režie paralelizace v podobě vytváření vláken a synchronizace ve frontě větší zátěž než samotná minimalizace, a logicky je pak rychlejší sekvenční BOOM.

Škálovatelnost mohu ilustrovat obrázkem z profileru pro obvod 50x50x150 na čtyřjádrovém procesoru AMD. Sloupce ukazují podíl času a počet současně využitých jader. Průměrné využití jader bylo 3,33, ideální by pochopitelně bylo 4. Svislá osa představuje celkový čas běhu – reálný, nikoliv CPU čas. Výsledný program se škáluje poměrně dobře.



Obrázek 7: Ilustrace škálovatelnosti

7 Závěr

V rámci této práce jsem si důkladně vyzkoušel paralelní programování na komplexním programu, jakým BOOM bezesporu je. S využitím knihovny pthread a pomocí proudového zpracování jsem paralelizoval BOOM jako celek a následně jsem pomocí OpenMP paralelizoval jednotlivé fáze. Zajímavou částí práce bylo i vytvoření nového bitvectoru s efektivnějším zaznamenáváním jednotlivých bitů, ve kterých je potřeba zaznamenat čtyři stavy namísto dvou.

Paralelní programování je velmi široká a náročná oblast informatiky – musel jsem prostudovat množství odborné literatury. Nejtěžší pro mě bylo vymýšlení celkového konceptu a následné přizpůsobování jednotlivých fází proudovému zpracování. Použití synchronizačních prostředků jsem se při tom snažil co nejvíce minimalizovat, to se nakonec povedlo až do té míry, že kromě joinování vláken je explicitní synchronizace pouze v ConcurrentQueue. Při paralelizaci jednotlivých fází pomocí OpenMP jsem využil datový paralelismus, musel jsem si však poradit s některými negativními vlastnostmi OpenMP, jako např. nemožnost použití `break` či záludnost plánovacích algoritmů, teprve poté se začaly projevovat výhody plynoucí z toho, že paralelní kód je generován překladačem.

Mimo samotné paralelní programování jsem se také do hloubky seznámil s knihovnou STL jazyka C++, zejména s výhodami a nevýhodami jednotlivých kontejnerů a naučil jsem se je efektivně využívat.

Výsledný paralelní nástroj jsem otestoval na zkušebních úlohách a myslím, že bylo dosaženo velmi dobrých výsledků. V budoucnu by se ale jistě dalo dosáhnout ještě lepšího výsledku pomocí chytřejšího rozložení zátěže, např. pomocí thread poolů, či pomocí přímého využití SIMD architektur.

8 Seznam použité literatury

1. **Fišer, Petr and Hlavička, Jan.** BOOM, A Heuristic Boolean Minimizer. *Computers and Informatics.* 2003, Vol. 22, 1, pp. 19-51.

2. *Flexible Two-Level Boolean Minimizer BOOM II and Its Applications.* **Fišer, Petr and Kubátová, Hana.** Cavtat (Croatia) : s.n., 2006. Proc. 9th Euromicro Conference on Digital Systems Design. pp. 369-376.

3. **Binstock, Andrew.** Threading Models for High-Performance Computing: Pthreads or OpenMP? *Intel® Software Network.* [Online] 2010. http://software.intel.com/en-us/articles/threading-models-for-high-performance-computing-pthreads-or-openmp/.

4. **Stallings, William.** *Operating Systems Internals and Design Principles.* s.l. : Prentice Hall, 2008.

5. **Breshears, Clay.** *The Art of Concurrency.* s.l. : O'Reilly, 2009.

6. **Rauber, Thomas and Rünger, Gudula.** *Parallel Programming for Multicore and Cluster Systems.* s.l. : Springer-Verlag, 2010.

7. **The IEEE and The Open Group.** Base Specifications Issue 7. [Online] 2008. http://pubs.opengroup.org/onlinepubs/9699919799/functions/contents.html.

8. **Terboven, Christian.** C++ and OpenMP. *cOMPunity.* [Online] 2007. http://www.compunity.org/events/pastevents/parco07/parco_cpp_openmp.pdf .

9. **Chapman, Barbara, Jost, Gabriele and van der Pas, Ruud.** *Using OpenMP: Portable Shared Memory Parallel Programming.* s.l. : The MIT Press, 2007.

10. **Herlihy, Maurice and Shavit, Nir.** *The Art of Multiprocessor Programming.* s.l. : Elsevier, 2008.

9 Seznam obrázků

10 Obsah přiloženého média

Součástí práce je také přiložené DVD, obsahuje následující soubory a adresáře:

- BOOM_paralelni\
  Složka se zdrojovými kódy paralelizovaného BOOMu. V podsložce Releases\MSVC\ se nachází soubor projektu pro MS Visual Studio 2010 Apps.sln. V podsložce Releases\Linux\ se nachází makefily pro různé překladače.
- BOOM_sekvencni\
  Složka se zdrojovými kódy původního sekvenčního BOOMu. Struktura je stejná jako u předchozí složky.
- BP.docx
  Zdrojový text bakalářské práce ve formátu Word 2007 (.docx).
- BP.pdf
  Platformově nezávislý text bakalářské práce ve formátu PDF.