Fakulta Informatiky, Masarykova Univerzita

Dynamicky rostoucí sdílená hašovací tabulka pro DiVinE

Bakalářská práce

Jiří Weiser

Brno, květen 2013

Prohlášení

Prohlašuji, že tato práce je mým původním autorským dílem, které jsem vypracoval samostatně. Všechny zdroje, prameny a literaturu, které jsem při vypracování používal nebo z nich čerpal, v práci řádně cituji s uvedením úplného odkazu na příslušný zdroj.

Vedoucí práce: doc. RNDr. Jiří Barnat PhD.

Poděkování

Chtěl bych poděkovat mému vedoucímu docentu RNDr. Jiřímu Barnatovi, PhD. za trpělivost a ochotu, se kterou mi pomáhal s bakalářskou prací, a RNDr. Petru Ročkaiovi za cenné rady a nápady. Mé díky jdou také lidem v laboratoři ParaDiSe. V neposlední řadě bych chtěl poděkovat rodině a Zuzce za podporu.

Shrnutí

V práci jsou prezentovány struktury, které umožňují procházení grafu ve sdílené paměti. Dál jsou v práci prezentovány různá řešení sdílené hašovací tabulky. Na závěr práce jsou provedena měření, která porovnávají stávající implementaci struktur oproti nové implementaci.

Klíčová slova

Ověřování modelů, Hašovací tabulka, Lock-free struktury, Datové struktury, Sdílená paměť

Obsah

1 Úvod **1**
1.1 DiVinE . . . 2
1.2 Visitor . . . 2
1.3 Cíl práce . . . 3

2 Současný stav **5**
2.1 Reprezentace vrcholů grafu . . . 5
2.2 Datová úložiště . . . 5
2.2.1 Hašovací tabulka . . . 6
2.2.2 Kompaktní hašovací tabulka . . . 6
2.3 Procházení grafu . . . 6
2.3.1 Třída `BFV` . . . 6
2.3.2 Třída `DFV` . . . 7
2.3.3 Třída `Partitioned` . . . 7

3 Nová implementace **9**
3.1 Třída `Shared` . . . 9
3.2 Datová fronta . . . 11
3.2.1 Třída `LockedQueue` . . . 11
3.2.2 Třída `SharedQueue` . . . 12
3.3 Sdílená tabulka . . . 12
3.3.1 Nerostoucí tabulka . . . 12
3.3.2 Rostoucí tabulka – spojový seznam . . . 14
3.3.3 Rostoucí tabulka – soustava tabulek . . . 20
3.3.4 Rostoucí tabulka – soustava tabulek s přehašováním . 21

4 Měření **25**
4.1 Výpočetní stroje . . . 25
4.2 Výběr varianty sdílené tabulky . . . 25
4.3 Porovnání procházení grafu . . . 28
4.4 Výsledky měření . . . 35

5 Závěr **37**
5.1 Plánovaná rozšíření . . . 37

Kapitola 1

Úvod

V rámci vývoje programů je běžnou praxí provádět testování, které má odhalit chyby jak v návrhu, tak v následném kódu. V případě paralelních aplikací ale kontrolní mechanismus testů selhává, protože vlákna mohou být mezi sebou libovolně proložena.

Jedna z možností, jak ověřit korektnost paralelního programu, nebo jeho modelu[1], je použít metodu explicitního ověření modelu, tzv. explicitní *model checking* [12]. Nástroje používající explicitní *model checking*[2] vyjmenují všechny možné stavy paměti, která mohou nastat v důsledku všech možných proložení vláken, čímž vytváří graf. Přechody mezi stavy paměti jsou realizovány jako provedení operace v jednom vlákně.

Mezi vyjmenovanými stavy nástroje hledají specifické stavy, nebo cykly v grafu, ve kterých se nachází specifické stavy. Označení specifických stavů je možné provést na základě vlastností, které definují požadavky na model.

Vyjmenování všech stavů paměti je exponenciální vzhledem k velikosti programu,[3] takže výsledné grafy jsou rozsáhlé. Je potřeba projít co nejrychleji jednotlivé stavy, které je možné řešit paralelním procházením grafu; velké nároky na paměť zároveň tvoří potřeba ukládat si prozkoumané stavy.

V dobách, kdy byly k dispozici výpočetní stroje s nejvýše 4 GB paměti, se nedostatek paměti řešil procházením grafu v distribuované paměti, které navíc umožňovalo paralelní procházení grafu. Výpočet v distribuované paměti je ale velmi závislý na rychlosti komunikace mezi jednotlivými výpočetními uzly, takže může docházet k výraznému zpomalení.

S příchodem architektury *x86_64*, která umožňuje mít na jednom výpočetním stroji více jak 4 GB paměti, a současně s příchodem vícejaderných procesorů začalo být rychlejší místo distribuované paměti využít sdílenou paměť mezi více vlákny na jednom výpočetním stroji.

[1] Dále bude používáno pouze označení model.

[2] Nástroj používající explicitní *model checking* se pak označuje jako explicitní *modelchecker*.

[3] Tomuto jevu se také říká stavová exploze.

1.1 DiVinE

DiVinE je explicitní model-checker, pomocí kterého lze verifikovat paralelní modely. Je tak konkurentem programů LTSmin[4] nebo SPIN.[5]

Pomocí nástroje DiVinE je možné verifikovat modely v několika vstupních formátech, například modely v jazyce *DVE* nebo *LLVM* bit-kódu, do něhož lze překládat kupříkladu aplikace psané v jazyce C a C++. Pro vstupní formát existuje generátor, který postupně vytvoří všechny dosažitelné stavy paměti, které může ověřovaný model mít.

Existují dva způsoby procházení grafu – do šířky a do hloubky. V nástroji DiVinE jsou zastoupeny oba dva. Procházení do hloubky používá algoritmus *Nested DFS* [4], zbylé algoritmy (*Metrics*[6], *Reachability*[7], *MAP* [6] a *OWCTY* [5]) používají procházení do šířky. V nástroji DiVinE se používá paralelní procházení grafu, které vychází z procházení do šířky. Paralelní varianta ale na rozdíl od sekvenční varianty postrádá jakkoliv definované pořadí procházení vrcholů grafu.

Obecně pro procházení grafu je potřeba mít datovou strukturu, do které vkládáme navštívené vrcholy a datovou strukturu, do které vkládáme vrcholy, které je potřeba zpracovat. Struktura pro nezpracované vrcholy se liší podle druhu procházení grafu. Pro procházení do hloubky je potřeba použít zásobník, kdežto procházení do šířky – sekvenční i paralelní varianta – vyžaduje frontu. Strukturu pro navštívené vrcholy lze použít stejnou pro oba druhy procházení grafu. Z důvodu vyšší rychlosti je ukládání navštívených stavů realizováno hašováním a následným uložením do tabulky.

1.2 Visitor

Visitor je koncept, který poskytuje rozhraní jednotlivým algoritmům pro procházení grafu. Pomocí generátoru vytváří stavy modelu. Navštívené stavy ukládá do hašovací tabulky, ještě nezpracované stavy ukládá v závislosti na druhu procházení grafu buď do fronty, nebo do zásobníku. Koncept algoritmům také poskytuje rozhraní pro zjištění vlastností stavů.

Některé implementace konceptu *visitor* umí procházet graf paralelně. Tyto implementace pak navíc obsahují mechanismy na komunikaci mezi svými instancemi, přičemž každá instance běží ve vlastním vlákně.

[4] `http://fmt.cs.utwente.nl/tools/ltsmin/`

[5] `http://spinroot.com/spin/whatispin.html`

[6] Algoritmus *Metrics* pouze vygeneruje celý stavový prostor.

[7] Algoritmus *Reachability* hledá ve stavovém prostoru specifické stavy; pokud je najde, procházení grafu se zastaví.

1.3 Cíl práce

Cílem práce je navrhnout a implementovat soubor tříd, které implementují koncept *visitor* a poskytují efektivní procházení grafu ve sdílené paměti, v nástroji DiVinE. Klíčový je návrh a implementace hašovací tabulky, která korektně pracuje ve sdílené paměti. Výstupem práce bude implementovaný kód a měření, která ověří škálování nově implementovaného kódu a zároveň ho porovnají oproti původnímu kódu.

Pojem škálování je v této práci použit jako vlastnost paralelního algoritmu, kdy navýšením počtu pracovních vláken při stejném rozsahu práce dojde k poměrnému snížení doby běhu algoritmu.

V rámci dalšího textu bude odkazováno na adresářovou strukturu, která je jako příloha poskytnuta v archivu práce. V adresářích jsou k dispozici zdrojové soubory DiVinE z doby, kdy vynikala tato práce, a implementace jednotlivých variant sdílených tabulek.

Pro spuštění přiloženého nástroje DiVinE je potřeba provést konfiguraci a přeložení[8]. Je potřeba upozornit, že nástroj DiVinE nepůjde přeložit, pokud se při konfiguraci zapne stromová komprese [11]. Při spuštění nástroje DiVinE lze vybrat třídu `Shared` pro procházení grafu přepínačem `-shared`, přičemž tato volba je dostupná pouze u algoritmů *Metrics* a *Reachability*.

[8] Podrobný postup je uveden na <`http://divine.fi.muni.cz/manual.html`>

Kapitola 2

Současný stav

Před započetím samotné implementace bylo nutné se seznámit se stávající programovou strukturou nástroje DiVinE a pochopit provázání jednotlivých tříd. Pro úspěšnou implementaci zadání bylo potřeba se věnovat hlavně části nástroje DiVinE, která implementuje procházení grafu.

2.1 Reprezentace vrcholů grafu

V nástroji DiVinE je několik generátorů stavů paměti a každý generátor vytváří stavy paměti s jinou strukturou, takže je potřeba stavy paměti jednotně reprezentovat jako vrcholy grafu. Tuto funkci zastává třída `Pointer` v souboru *divine/divine/toolkit/pool.h*.[1]

Většina algoritmů v nástroji DiVinE potřebuje ukládat některé dodatečné informace k vrcholům grafu. Z toho důvodu třída `Pointer` ukazuje na paměť, která může kromě stavu paměti obsahovat právě dodatečné informace.

2.2 Datová úložiště

Datová úložiště jsou rozdělena mezi několik zdrojových souborů. Samotné tabulky jsou zařazeny do složky *divine/divine/toolkit/*, protože mohou být snadno použity i mimo kontext procházení grafu. Protože třídy, které implementují koncept *visitor*, potřebují unifikovaně pracovat s tabulkami a potřebují, aby byly obecné tabulky rozšířeny o specifické metody, jsou v souboru *divine/divine/graph/store.h* definovány třídy, které požadovanou funkcionalitu tabulkám poskytují.

Práce s úložišti je v nástroji DiVinE specifická, protože je potřeba, aby obsahovala pouze metody pro vkládání a vyhledání vrcholů grafu. V prů-

[1] Protože je možné při překladu nástroje DiVinE určit, jestli bude pro alokace vrcholů grafu použitý standardní systémový alokátor, nebo alokátor implementovaný v nástroji DiVinE, je třída `Pointer` v souboru implementovaná dvakrát.

běhu psaní práce byla navíc upravena metodika přístupu k tabulce, takže v aktuální verzi je postačující, když tabulka umí pouze operaci vkládání.

2.2.1 Hašovací tabulka

Hašovací tabulka v nástroji DiVinE je implementována v souboru *divine/divine/toolkit/hashset.h*. Tabulka neumožňuje paralelní přístupy z více vláken.

Poskytuje vkládání vrcholů grafu a jejich vyhledávání. Při zaplnění z 75% dojde ke zvětšení, při kterém jsou již vložené vrcholy znovu vloženy do zvětšené tabulky. Pro rychlý přístup je ukládání realizováno pomocí hašování – každý vrchol grafu je do tabulky uložen se svým hašem.

2.2.2 Kompaktní hašovací tabulka

Kompaktní hašovací tabulka, prezentovaná v [10], je implementovaná jako nadstavba nad hašovací tabulkou. Využívá skutečnosti, že ukládané vrcholy grafu se skládají ze dvou částí, přičemž větší část – stav paměti – není v průběhu prohledávání grafu nijak měněna. Do tabulky se nezapisují celé vrcholy grafu s hašem, ale pouze haš s dodatečnými informacemi vrcholu grafu (ty se na rozdíl od stavu paměti mohou za běhu algoritmu měnit).

Uvedený způsob ukládání vrcholů má za následek snížení nároků na paměť, ale vyvstává riziko, že různé stavy paměti budou mít stejný haš. Pokud k tomu dojde, tabulka při vkládání mylně oznámí, že daný vrchol grafu byl již navštíven, takže může nastat situace, že nebudou navštíveny všechny vrcholy v grafu.

2.3 Procházení grafu

Implementace všech tříd, které používají koncept *visitor* se nacházejí v *divine/divine/graph/visitor.h*. Jednotlivé třídy, které se v souboru *visitor.h* nacházejí, jsou: `BFV`, `DFV` a `Partitioned`.

Protože výše uvedené třídy mají společné rozhraní, jsou obecné funkce implementovány ve třídě `Common`, která je ve stejném souboru a která poskytuje metody na zpracování vrcholu grafu, zaznamenává vrcholy, které již byly navštíveny, a umožňuje předčasné ukončení prohledávání grafu.

2.3.1 Třída `BFV`

Třída `BFV` reprezentuje sekvenční prohledávání grafu do šířky, takže ke své činnosti vyžaduje datové úložiště a frontu.

Třídu `BFV` využívá algoritmus *POR* [7], který eliminuje nedůležité vrcholy grafu. Modifikace třídy `BFV` je použitá ve třídě `Partitioned`.

2.3.2 Třída DFV

Třída DFV reprezentuje sekvenční prohledávání grafu do hloubky, proto ke své činnosti vyžaduje zásobník a frontu.

Třídu DFV používá algoritmus *Nested DFS*, a to jak v sekvenční variantě, tak v paralelní variantě s použitím dvou vláken.

2.3.3 Třída Partitioned

Třída Partitioned umožňuje paralelní procházení grafu. Paralelismus je v implementaci této třídy řešený tak, že každý vrchol grafu se podle hodnoty haše přiřadí jedné instanci třídy Partitioned. Každá instance má svoje identifikační číslo, počet vláken je n, takže se vlastník spočítá jako $vlastník \leftarrow haš \mod n$.

Posílání stavů mezi vlákny je řešeno pomocí sady front, kdy každé vlákno má frontu nezpracovaných vrcholů grafu od každého vlákna. V součtu je třeba k zasílání nezpracovaných stavů n^2 front. Fronty jsou implementovány jako N-zřetězený[2] spojový seznam, který podporuje čtení jedním vláknem za současného zapisování druhým vláknem. V souboru *divine/examples/llvm/fifo.cpp* je upravená implementace fronty, která byla pomocí DiVinE úspěšně verifikována.

Výhodou této implementace je správa vrcholů grafu. Každá instance třídy Partitioned může přistupovat pouze ke svým vrcholům grafu, takže není potřeba řešit sdílený přístup do tabulek, kam se ukládají zpracované vrcholy.

Další výhodou je dobré fungování v distribuovaném prostředí. Stačí, aby na každém výpočetním uzlu vznikla jedna upravená instance třídy Partitioned, která bude zajišťovat přeposílání stavů mezi výpočetními uzly.

Velkou nevýhodou je pak omezené škálování. Při generování nových vrcholů grafu totiž často dochází k vygenerování vrcholu, který byl již zpracován. Pro ověření je nutné vrchol grafu dopravit vláknu, které vrchol vlastní. Tím pádem se ve frontách vyskytují vrcholy, které již byly zpracovány. Zvyšuje se tak režie spojená s vkládáním prvků do fronty, která je předpokládaná, ale není nijak naměřená.

Protože vrcholy grafu jsou přidělovány jednotlivým vláknům na základě zbytku po dělení, může se stát, že se budou vytvářet takové haše, které sníží výkonnost nerovnoměrným rozložením zátěže mezi vlákna.

[2] Jedná se o spojový seznam, kdy v každém prvku seznamu není jedna hodnota, ale n hodnot v závislosti na parametrech.

Kapitola 3

Nová implementace

Pro efektivní běh ve sdílené paměti je potřeba vytvořit nové struktury pro procházení grafu. Toto zahrnuje novou třídu `Shared` implementující koncept *visitor*, sdílenou frontu a sdílenou tabulku. Dále je potřeba vytvořit ke všem třídám patřičné jednotkové testy.

Před započetím bakalářské práce bylo možné najít rozpracovaný kód v repozitáři DiVinE, ale tento kód nešlo přeložit. Po ukončení implementace se musely nové struktury zaintegrovat do DiVinE.

Následující části textu budou popisovat implementační záležitosti, které se dají najít ve složkách *divine/divine/graph/* a *divine/divine/toolkit/*. Během úprav bylo nutné částečně modifikovat kódy i v jiných částech DiVinE tak, aby bylo možné korektně instanciovat obě třídy pro procházení grafu – `Shared` i `Partitioned`.

3.1 Třída `Shared`

Idea třídy `Shared` je taková, že předem zvolený počet vláken bude odebírat nezpracované vrcholy grafu ze sdílené fronty. Z každého vybraného vrcholu se vygenerují následníci. Ti se pokusí vložit do sdílené tabulky, přičemž pokud se vložení podaří, je vygenerovaný vrchol grafu poslán do fronty na další zpracování.

Tento princip odebírání vrcholů grafu z fronty zatíží při dostatku vrcholů rovnoměrně všechna vlákna. Dál nebude docházet k duplikovaným výskytům vrcholů ve frontách. Problémem ale je umožnit výpočet v distribuovaném prostředí, který není zatím u třídy `Shared` implementován.

Detekce ukončení paralelního běhu

Jelikož třída `Shared` pracuje nad sdílenou frontou, bylo by možné pro zjištění ukončení uvažovat nad kontrolou neprázdnosti fronty. Tento přístup ale nebude fungovat, jelikož by mohl nastat případ, kdy $n - 1$ instancí třídy

`Shared` vyprázdní frontu, ale nestihnou vložit do fronty žádné nové vrcholy grafu. V tomto případě by instance číslo n zjistila, že fronta je prázdná, a skončila by.

To je důvod, proč je v *divine/divine/toolkit/shmem.h* implementovaná třída `ApproximateCounter`, která má za úkol počítat přibližný počet vrcholů grafu ve frontě. Počítadlo je navrženo tak, že existuje sdílená a lokální část. Standardně se manipuluje pouze s lokální hodnotou počítadla. Přístup ke sdílené části je při detekci ukončení a synchronizaci. Instance třídy `Shared` ukončí svoji činnost, pokud je hodnota sdíleného počítadla nulová nebo menší.

Vložením vrcholu grafu do fronty se počítadlo zvyšuje. Při výběru vrcholu z fronty se instance třídy `Shared` nejprve pokusí vygenerovat následníky, vložit je do fronty a až poté sníží počítadlo. Postup v tomto pořadí je důležitý, protože pokud by došlo nejprve ke snížení počítadla a teprve poté k vygenerování následníků a jejich vložení do fronty, mohla by nějaká další instance třídy `Shared` mylně usoudit, že byla všechna potřebná práce vykonána, a skončila by.

Synchronizace lokálního počítadla se sdíleným probíhá pouze za několika situací:

- Dojde k nulové hodnotě lokálního počítadla u přičítání. V tom případě se sdílená hodnota navýší o elementární krok (aktuálně 10.000). Stejná hodnota se pak vloží do lokální části počítadla, od které se následně odečítá.

- Instance třídy `Shared` zjistila, že je fronta prázdná. V tom případě si ověří, jestli nedošlo ke zpracování všech vrcholů grafu uzlů. Instance třídy `Shared` odešle své nově vygenerované stavy do fronty a provede synchronizaci.

Další možností synchronizace sdíleného počítadla nastává v případě, že generátor zjistí, že došlo k porušení či splnění vlastností definovaných v zadání modelu. V tom případě vynuluje sdílené počítadlo a vymaže obsah fronty, čímž přinutí ostatní instance třídy `Shared` skončit. Je nutné, aby mohla být v počítadle uložená záporná hodnota, protože další instance mohou ještě provádět synchronizaci, a mohlo by tak dojít k podtečení hodnoty sdíleného počítadla v případě, že by bylo implementované jako ne-znaménkový typ.

Ačkoliv je ve sdílené části mnohdy vyšší hodnota než počet prvků se frontě, nebrání to korektnímu fungování, neboť před ukončením dojde k synchronizaci s lokálním počítadlem, které obsahuje správnou hodnotu. Při předčasném ukončení je irelevantní vědět správný počet vrcholů ve frontách.

`ApproximateCounter` je rozdělený na dvě části, aby byly časté zápisy do sdílené části eliminovány. V případě více vláken by časté zápisy do stejné části paměti způsobovaly zpomalení běhu programu, protože by na úrovni *CPU cache* docházelo k neustálým zápisům do paměti, po kterých by bylo

nutné zneplatnit *CPU cache* na ostatních procesorech a stáhnout z *RAM* aktuální obraz paměti.

Detekce startu všech instancí třídy `Shared`

Instanciace třídy `Shared` je prováděna paralelně, stejně tak vložení iniciálních vrcholů grafu. Zde ale může nastat problém, protože instance třídy `Shared`, které se nepodaří vložit žádný iniciální stav do fronty, může dojít k detekci ukončení dřív, než jiné instance nastaví sdílené počítadlo na nenulový stav, čímž dojde k předčasnému ukončení běhu této instance třídy `Shared`.

Pro eliminaci předčasného ukončení je definována třída `StartDetector`, která počítá, kolik instancí třídy `Shared` se dostalo ke spuštění metody `run`. Kterékoliv instanci třídy `Shared` je pak dovoleno skončit pouze tehdy, jestliže všechny instance spustili metodu `run`. V metodě `run` následně probíhá zpracování vrcholů grafu, které se vytahují z fronty.

3.2 Datová fronta

Sdílená datová fronta slouží k přibližně rovnoměrně rozdělené práci mezi jednotlivé instance třídy `Shared`. Pro rychlý paralelní přístup je rovněž jako `ApproximateCounter` rozdělena na část lokální `SharedQueue` a sdílenou `LockedQueue`.

3.2.1 Třída `LockedQueue`

Sdílená část fronty je jednoduché rozšíření nad frontou `std::deque`. Pro vzájemně výlučný přístup používá zámek a redukuje počet metod.

Kromě políčka `empty` jsou dostupné metody

- `void push( const T & )`
- `void push( T && )`
- `T pop()`

První dvě metody mají shodnou funkčnost – vložení prvku –, pouze druhá jmenovaná využívá nové vlastnosti *C++11*, kterou je *move* sémantika. Metoda `pop` vyjme prvek z fronty, pokud je fronta neprázdná, jinak vrátí šablonový objekt `T` vytvořený bezparametrickým konstruktorem. Políčko `empty` lze využít na detekci prázdnosti fronty.

Jako zámek se využívá `SpinLock`, protože operace na vkládání a vyjímání jsou rychlé. Použití standardizovaného zámku[1], který vlákno uspí, pokud se nepodaří zámek uzamknout, by bylo plýtvání časem a zbytečně by docházelo ke zpomalení procházení grafu.

[1] [8], std::mutex – cppreference.com, url: <`http://en.cppreference.com/w/cpp/thread/mutex`>.

3.2.2 Třída `SharedQueue`

Lokální část dědí ze společného rozhraní `QueueFrontend` pro všechny fronty. Pro přístup ke stavům používá sdílenou část fronty.

Z důvodu rychlosti pracuje `SharedQueue` tak, že má dvě své vnitřní fronty – vstupní a výstupní. Pokud je výstupní fronta dostatečně veliká, pošle se do `LockedQueue`. Vstupní fronta se stahuje ze sdílené fronty, pokud je vstupní fronta prázdná. Pokud je aktuálně stažená vstupní fronta také prázdná, dojde k odeslání výstupní fronty a k synchronizaci sdíleného počítadla[2]

Aby i v počátku procházení grafu bylo využito co nejvíce vláken, je limit velikosti výstupní fronty nastaven na hodnotu 8. S každým odesláním výstupní fronty do `LockedQueue` se dvojnásobně zvyšuje limit až na horní hodnotu 32. To zaručí, že budou vygenerované stavy rovnoměrně rozložené mezi jednotlivé instance třídy `Shared` a zároveň nebude docházet k příliš častým přístupům k zámku ve třídě `LockedQueue`, které by zpomalovaly procházení grafu. V rámci vývoje bylo testováno, jaký je vliv limitu na rychlost běhu, a vyšlo najevo, že pro generátor přeloženého modelu v jazyce *DVE* je hodnota 32 optimální.

3.3 Sdílená tabulka

Stěžejní část bakalářské práce bylo navrhnout a implementovat efektivní hašovací tabulku pro sdílený přístup. V práci jsou uvedeny čtyři varianty tabulky, z nichž tři jsem implementoval.

Obecné hašovací tabulky pro práci ve sdíleném prostředí je velmi těžké dohledat, neboť jejich implementace bývají netriviální. Lze například najít `java.util.concurrent.ConcurrentHashMap`[3] nebo *TBB*[4] Concurrent Map. Implementace těchto dvou tříd je ale velmi komplikovaná a pro potřeby DiVinE zbytečně obsáhlé. Byly tudíž hned zpočátku práce zavrženy.

3.3.1 Nerostoucí tabulka

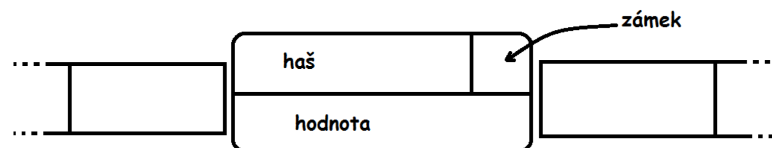
V DiVinE byla již implementována hašovací tabulka na základě [9]. Extrahovanou tabulku lze nalézt v *tables/nongrowing/*. Tabulka potřebuje před použitím nastavit na určitou velikost. Tato tabulka byla dle [9] verifikována pomocí nástroje *SPIN*. Její modifikace se používá v nástroji *LTSmin*. Ve složce *tables/nongrowing/verify* je upravená verze pro verifikaci pomocí DiVinE.

Princip tabulky je takový, že v každé buňce je zámek, který signalizuje, že se do buňky zapisuje. Pokud některé jiné vlákno, které prochází tabulku,

[2]Viz pasáž o detekci zastavení v sekci 4.1.

[3]Dokumentace na <`http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentHashMap.html`>

[4]Thread Building Blocks – <`http://threadingbuildingblocks.org/`>



Obrázek 3.1: Detail buňky tabulky.

aby našlo buď již vloženou hodnotu, nebo volnou buňku, narazí na uzamčený zámek, tak počká, dokud se zámek neuvolní. Po jeho uvolnění zkontroluje právě vložený obsah a pokračuje dál v běhu.

Každá buňka tabulky obsahuje tři hodnoty (viz Obrázek 3.1), které je třeba uložit:

- zámek
- haš hodnoty
- samotnou hodnotu

Zámek je kvůli zápisu v paralelním prostředí. Haš je uložený kvůli efektivitě porovnání, protože je jednodušší a časově méně nákladné porovnat nejprve dva haše a až při shodě porovnávat celé obsahy stavů, které jsou oproti velikosti čísla v paměti mnohonásobně delší. Hodnota je pak uložena právě kvůli tomu, že dva různé stavy mohou mít shodné haše.

V tabulce je definováno kvadratické skákání po buňkách, čímž se tabulka snaží eliminovat případné kolize. Dál je definovaný maximální počet kolizí pro vkládání. Pokud metoda vložení není schopná vložit prvek do tabulky po definovaném počtu kolizí, vypíše se chybová hláška a program skončí.

Kvůli paměťové úspoře a lepší efektivitě paralelního zápisu a vyhledávání je zámek přidán k hodnotě haše. Jelikož si stačí pamatovat pouze hodnoty zamčeno/odemčeno, je třeba odebrat z haše 1 bit. Ve výsledku se tak použije haš posunutý o 1 bit doleva. Tím zámek získá místo nejnižšího bitu, ale haš ztratí jeden nejvyšší bit. To ale není tolik na škodu, neboť pravděpodobnost shody hašů dvou různých hodnot je přibližně $\frac{1}{2^{31}}$.

Protože je výsledná velikost zámku a haše 2^{32} a standard *C++11* definuje atomické operace (nejen) nad elementárními typy[5], může dojít k současnému atomickému zápisu haše a zámku. To se využívá při zápisu, kdy dojde k zamčení, aby mohlo dojít k neatomickému zápisu hodnoty do buňky, po kterém dojde k odemčení. Po odemčení se hodnoty v buňce již nemění.

[5] [8], std::atomic – cppreference.com, url: <`http://en.cppreference.com/w/cpp/atomic/atomic`>.

```
 1 for pokus ← 0 ... maximum pokusů do
 2 │ buňka ← tabulka [ index(haš, pokus) ];
 3 │ if buňka je prázdná then
 4 │ │ if lze atomicky vložit haš a zamknout then
 5 │ │ │ zapiš hodnotu;
 6 │ │ │ odemkni;
 7 │ │ │ return ANO;
 8 │ │ end
 9 │ end
10 │ if buňka má stejný haš then
11 │ │ počkej na odemknutí;
12 │ │ if hodnoty se rovnají then
13 │ │ │ return NE;
14 │ │ end
15 │ end
16 end
17 zastavit program;
```

Algoritmus 1: Vkládání do tabulky

Algoritmus vkládání se může spolehnout na to, že se hodnota haše v libovolné buňce může změnit v tomto pořadí:

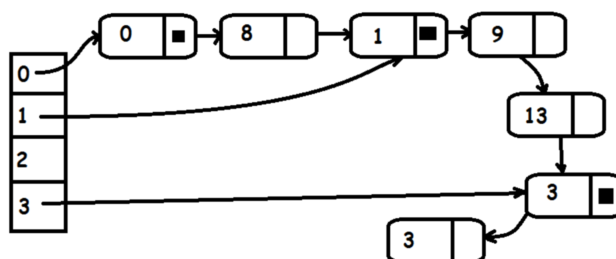
$$0 \rightarrow (hash << 1|1) \rightarrow (hash << 1|0)$$

Pokud tedy narazí při procházení na první variantu haše, může zkusit vložit hodnoty do buňky. Pokud narazí na druhou variantu haše, je potřeba počkat si do ukončení zápisu, a zkontrolovat rovnost hodnot. Třetí varianta je normálně obsazená buňka. Pro správné fungování rozpoznávání stavu buňky podle varianty haše je třeba zamezit, aby haš neměl nulovou hodnotu ani po bitovém posunu doleva.

Takto navržená tabulka velmi dobře škáluje [9] a má velmi rychlou dobu přístupu. Její problém ovšem je, že není zvětšovací, a tudíž je pro reálné nasazení špatně použitelná. V kontextu formální verifikace lze sice odhadnout, kolik se řádově vygeneruje stavů, ale vždy může být odhad chybný a vyžaduje zbytečné studování modelu.

3.3.2 Rostoucí tabulka – spojový seznam

Celá tato tabulka je implementace na základě publikace [1]. Extrahovanou tabulku lze najít v *tables/linkedlist/*. Jako jedinou zde uvedenou tabulku lze implementovat jako lock-free datovou strukturu. Lock-free datová struktura je struktura pro paralelní přístup, kdy není použitý žádný typ zámku.



Obrázek 3.2: Schéma tabulky. Vlevo je vektor zkratek. Prvky spojového seznamu s černým čtverečkem jsou falešné prvky.

Tabulka používá spojový seznam na uchování jednotlivých hodnot. Pro korektní chování se v tabulce nacházejí dva typy prvků – pravé, které obsahují haš a hodnotu, a falešné, které obsahují pouze haš. Jelikož je ale procházení celého spojového seznamu při hledání hodnoty časově nákladné, součástí tabulky je vektor zkratek do spojového seznamu. Vektor zkratek je pole ukazatelů na falešné prvky spojového seznamu, protože vložením zkratky nechceme do tabulky vkládat žádné hodnoty.

Aby bylo možné rovnoměrně rozmisťovat zkratky, je potřeba upravit práci s hašem. Pokud by totiž byly haše ve spojovém seznamu uloženy vzestupně, nebylo by možné implementovat rozumné vkládání zkratek. Proto jsou ve spojovém seznamu uloženy obrácené hodnoty hašů. Obrácenou hodnotou se myslí to, že v rozsahu délky haše – 32 bitů – bude každý i-tý bit přesunut na $(31 - i)$-tou pozici indexováno od 0.

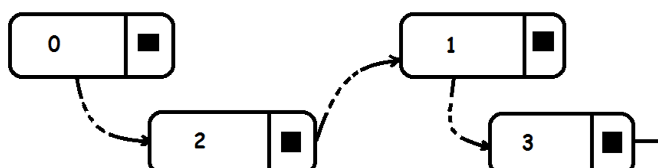
Protože je tabulka implementována pomocí spojového seznamu, není potřeba zadávat počáteční velikost. Je ale vhodné, aby počáteční velikost vektoru zkratek byla úměrná očekávané velikosti verifikovaného modelu.

Rekurzivně rozdělující řazení

Ve spojovém seznamu budou uloženy hodnoty seřazené vzestupně podle obrácené hodnoty haše. V následujícím textu bude ukázáno, jakým způsobem se zařazují falešné hodnoty do spojového seznamu. Pravé hodnoty jsou zařazovány shodně s tím rozdílem, že falešná hodnota musí být vždy umístěna před všemi pravými hodnotami stejného haše. Vektor zkratek pak ukazuje právě na falešné hodnoty.

Vlastnost tohoto uspořádání je taková, že pro dvě falešné hodnoty 0 a 1 je posloupnost následující:

Dvojnásobným navýšením falešných hodnot dojde k rozšíření o hodnoty 2 a 3. Ty se do stávající posloupnosti zařadí tak, že se postupně každá nová falešná hodnota zařadí za každou ve spojovém seznamu již obsaženou hodnotu. Po dvojnásobném navýšení tak bude seznam falešných hodnot vypadat následovně:



Z vlastností vkládání falešných hodnot vyplývá, že je nutné, aby tabulka zkratek měla velikost mocniny dvojky. Při každém zvětšení lze ihned vložit všech $\frac{n}{2}$ nových falešných hodnot, ale to je zbytečné, proto implementovaná tabulka používá líné vkládání falešných hodnot, kdy se falešná hodnota vloží až pokud je nutné jí použít jako zkratku. Je totiž možné jednoduše určit pro každou novou falešnou hodnotu, před kterou starou se má vložit. Líné vkládání se proto použije jak při vkládání, tak při vyhledávání hodnoty.

Pro novou falešnou hodnotu i je jejím předkem falešná hodnota j taková, že $j = i$, u kterého byl nejvyšší jedničkový bit nahrazen nulovým bitem. Tato rekurzivní metoda určení předka lze aplikovat na všechna i, kdy $n > i > 0$ a n je aktuální velikost vektoru zkratek.

Pro pravou hodnotu se zkratka na falešný předek určí tak, že se použije zkratka na indexu $hash \mod n$ vektoru zkratek, kdy n je aktuální velikost vektoru. Pokud tato zkratka neexistuje, vytvoří se rekurzivním určením, které je uvedeno výše.

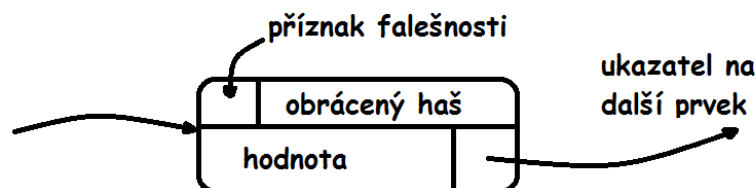
Pro detekci potřeby zvětšit vektor zkratek se v implementovaném algoritmu používá metodika, kdy pokud bylo potřeba při vkládání provést více jak 8 skoků po jednotlivých prvcích spojového seznamu. V rámci testování byly zkoušeny hodnoty 4, 8, 16 a 32. Hodnoty 16 a 32 způsobily výrazné zpomalení operací vkládání a vyhledávání. Použitím hodnoty 4 oproti hodnotě 8 se operace podstatně neurychlily ani nezpomalily. S ohledem na menší velikost vektoru zkratek je nakonec použitá hodnota 8.

Lock-free spojový seznam

Implementace spojového seznamu vychází z návrhu v [2], který převzali autoři v [1]. Uvedené implementace obsahují operaci odmazání prvku se spojového seznamu, která je ze zde prezentované implementace odstraněna. Zdrojový soubor je uložen jako *tables/linkedlist/LinkedList.h.*

Každý spojový seznam je složený z jednotlivých prvků. Každý prvek (viz Obrázek 3.3) obsahuje čtyři hodnoty:

- detekci falešného uzlu



Obrázek 3.3: Detail prvku ve spojovém seznamu.

- obrácený haš
- hodnotu
- ukazatel na další prvek

Prvek je možné zkonstruovat buď pomocí hodnoty a haše, tím vznikne pravý prvek, nebo jenom pomocí haše, tím vznikne falešný prvek. Pro potřeby lock-free sémantiky je potřeba, aby ukazatel na další prvek byl atomický. Jediné další metody prvku jsou porovnávací operátory, pomocí kterých lze definovat uspořádání prvků mezi sebou.

V popisu výše byly uvedeny termíny pravé a falešné hodnoty. Pro implementační potřeby je pravá hodnota reprezentovaná pravým prvkem, který má hodnotu i haš a nemá znak falešného uzlu. Falešná hodnota je pak reprezentovaná falešným uzlem, který má znak falešného uzlu, haš a nemá hodnotu.

Spojový seznam potřebuje mít definovány dvě různá porovnání: pro pravé prvky – `OrderRegular` – a pro falešné prvky – `OrderDummy` – kvůli podmínkám uvedeným výše.

Spojový seznam poskytuje celkem tři operace:

- vložení pravého prvku
- vložení falešného prvku
- nalezení prvku

Není poskytováno například zjištění počtu prvků v poli. Protože se očekává poměrně velké zatížení operace vkládání, mnoho vláken by tak soupeřilo o jedno paměťové místo a to by vedlo ke zbytečnému zpomalení.

Všechny tři metody jsou implementovány pomocí `protected` metody `Exists`[6], která na základě vybraného dočasného prvku, startovací pozice ve spojovém seznamu a porovnávací třídě vrátí tři hodnoty:

[6] V implementaci se první dvě veřejné metody jmenují `Insert` a třetí shodně `Exists`. Všechny metody mají rozdílné signatury, ale pro přehlednost uvádím veřejné metody českým opisem.

```
1  Function Exists(REF ukazatelNaPrvek, dočasnýPrvek, REF
   početSkoků) is
2  |   while ukazatelNaPrvek je validní do
3  |   |   if *ukazatelNaPrvek a dočasnýPrvek mají špatné pořadí then
4  |   |   |   return *ukazatelNaPrvek = dočasnýPrvek;
5  |   |   end
6  |   |   početSkoků++;
7  |   |   přesun na další prvek;
8  |   end
9  end
10 Function Insert(ukazatelNaPrvek, novýPrvek, početSkoků) is
11 |   repeat
12 |   |   if Exists(ukazatelNaPrvek, novýPrvek, početSkoků) then
13 |   |   |   return NE;
14 |   |   end
15 |   |   novýPrvek.další ← ukazatelNaPrvek.další;
16 |   until lze atomicky připojit novýPrvek za ukazatelNaPrvek;
17 |   return ANO;
18 end
```

Algoritmus 2: Vkládání nového prvku, `protected` metoda `Exists` a metoda `Insert`.

- zda se byl dočasný prvek nalezen
- referenci na ukazatel na poslední prvek v seznamu, který podle porovnávací třídy je v uspořádání před dočasným prvkem
- počet skoků, které metoda `Exists` provedla

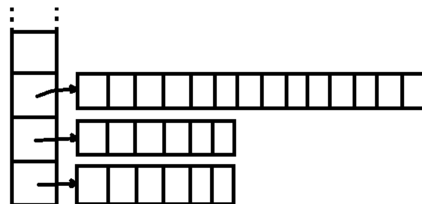
Implementace metod vložení se pokusí (pouze v případě, že nebyl nový prvek nalezen) atomicky pomocí `CAS` instrukce zařadit nový prvek za vrácený prvek seznamu. Pokud se jim to nepodaří, bylo jiné vlákno rychlejší a je potřeba opakovat proceduru vyhledání a vložení. Jako startovací pozice je zde pro urychlení procházení použit vrácený prvek seznamu.

Implementace metody nalezení prvku je pak triviální – použije se návratová hodnota metody `Exists`.

Lock-free rostoucí vektor

Implementace lock-free vektoru vznikla na základě studia publikace [3]. Zdrojový soubor je uložen jako *tables/linkedlist/Vector.h*.

Idea lock-free vektoru je taková, že existuje pole, které si pamatuje jednotlivé řádky tabulky. Protože je toto pole sdílené, není možné za běhu více vláken pole měnit. Proto má lock-free vektor nastavený horní limit velikosti.



Obrázek 3.4: Schéma tabulek v lock-free vektoru.

```
1  Function Grow(vektor) is
2  |  velikost ← vektor.velikost;
3  |  novýŘádek ← pole[velikost];
4  |  index ← MSB(velikost)-MSB(vektor.iniciálníVelikost)+1;
5  |  if CAS(vektor.řádky[index], 0, novýŘádek) then
6  |  |  vektor.velikost ← vektor.velikost + velikost;
7  |  |  return ANO;
8  |  else
9  |  |  smaž novýŘádek;
10 |  |  return NE;
11 |  end
12 end
```

Algoritmus 3: Metoda na zvětšení vektoru.

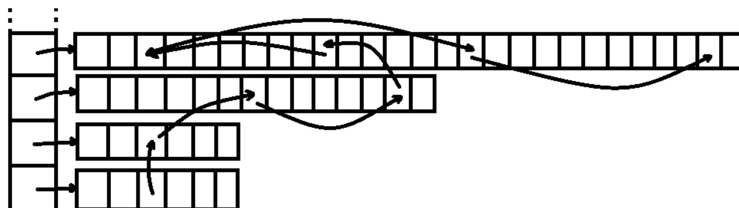
Protože ale vektor zvětšuje svoji kapacitu na dvojnásobek dosavadní velikosti, čímž roste exponenciálně, je horní limit kupříkladu 64 možných řádků – čili 63 možných zvětšení – více než dostačující na současných *x86_64* architekturách.

Pro potřeby hašovací tabulky je postačující, když se budou implementovat metody:

- zjištění velikosti
- žádosti o zvětšení
- náhodný přístup dle indexu

Velikost si lze uchovat v atomické proměnné. Na rozdíl od spojového seznamu zde dochází k méně častému zápisu do sdílené proměnné z více vláken. Hodnota velikosti se mění pouze po úspěšně provedeném zvětšení a těch je omezené množství.

Žádost o zvětšení pak pracuje tak, že alokuje paměť o velikosti dosavadního vektoru a tu zkusí přiřadit do posledního volného místa v poli. Pokud je vlákno předběhnuto, vytvořené pole dealokuje. Právě zde by se velmi hodil zámek, který by zamezil případné mnohonásobné alokaci, kdy se zbytečně



Obrázek 3.5: Nákres prohledávání v soustavě tabulek při vkládání.

může alokovat až $(t-1) \cdot n$ paměti – kdy t je počet vláken, n je velikost stávajícího vektoru –, které může vyústit v chybné chování, kdy bude program operačním systémem zabit kvůli žádosti o velmi mnoho paměti. Z demonstračních důvodů je ale implementace lock-free.

Zajímavým poznatkem pak je, že první dva řádky vektoru jsou stejně veliké. Toho se využívá při implementaci indexace do tabulky, kde je potřeba spočítat z jednoho indexu dva – výběr řádku a výběr políčka v řádku. Index na výběr řádku se spočítá jako

$$f(index) = \begin{cases} 0 & index < row[0].size \\ MSB(index) - MSB(row[0].size) + 1 & \text{jinak} \end{cases}$$

Index na výběr políčka lze pak triviálně dopočítat

$$h(index) = \begin{cases} index & \text{pokud } index < row[0].size \\ WithoutMSB(index) & \text{jinak} \end{cases}$$

Dvě výše uvedené funkce jsou implementovány na základě [13]. Funkce `MSB` vrátí index nejvyššího bitu. Funkce `WithoutMSB` vrátí číslo bez nejvyššího bitu.

3.3.3 Rostoucí tabulka – soustava tabulek

Extrahovanou tabulku lze najít v *tables/growing*.

Tabulka používá princip rostoucího pole podobně jako výše uvedený lock-free vektor. Pouze při zvětšovaní dochází k použití zámku, protože pro reálný běh je nebezpečné alokovat zbytečně mnoho paměti. Pro zápis se v tabulce používá stejná technika jako u nerostoucí tabulky.

Funkce pro skákání po řádku tabulky je zde oproti nezvětšovací tabulce vylepšena o vlastnost skákání po jednotlivých *cache-line*. S očekávanou velikostí 16 bytů na buňku a 64 bytů na velikost *cache-line* dojde k porovnání čtyř buněk na jeden paměťový dotaz. V rámci práce není nijak experimentálně ověřené, jestli má úprava skákání po tabulce vliv na rychlost.

U normální tabulky s rovnoměrným rozložením hašů dochází ke kolizím většinou pouze malé délky. U této tabulky ale dochází vždy k logaritmickému

prohledávání. Je to z toho důvodu, že v prvním a druhém patru se prohledává jedna *cache-line*. V každém dalším patře se pak prohledává tolik *cache-line*, jaký je index řádku, indexováno od nuly. Tabulka se tak plní jako binární strom, kdy se nové prvky zavěšují na spodek stromu.

Z těchto důvodu je již od počátku předpoklad, že tabulka bude s rostoucí zaplněností velmi rychle zpomalovat práci jednotlivých vláken. Tento fakt se projevil při experimentálním testování DiVinE. Naprosto nejvytíženější operací bylo porovnání haše v buňce oproti nule, protože to je první operace, která čeká na zaslání obsahu paměti z RAM do procesoru a protože k této operaci dochází průměrně $\log n$ často vzhledem k zaplněnosti tabulky. Výsledky měření v sekci 3.4 potvrzují tuto domněnku.

3.3.4 Rostoucí tabulka – soustava tabulek s přehašováním

Extrahovanou tabulku lze najít v *tables/growing-rehash*. Zápisy do řádku tabulky jsou řešené obdobně jako u nerostoucí varianty. Tato tabulka potřebuje ke správnému chodu vědět ještě před paralelním během počet vláken a je nutné, aby se po dobu paralelního běhu neměnil počet vláken, která pracují s tabulkou.

Tato tabulka se snaží eliminovat problém vzrůstající kolizní délky způsobem běžným pro sekvenční tabulky – přehašování. V sekvenčním přístupu je přehašování snadné, neboť nikdo jiný do tabulky během přehašování nezasahuje. Při paralelním přístupu je ale nezbytné, aby všechny nové i dobíhající zápisy proběhly korektně a aby tak byla zaručena unikátnost vložené hodnoty.

Pro nárůst kapacity se používá obdobný princip jako u tabulky v sekci 3.3.3. Používá se také totožná indexační funkce pro procházení řádku po jednotlivých *cache-line*. Jiná je ale detekce nárůstu – tato tabulka používá pro určení maximálního počtu akceptovatelných kolizí konstantu $16 \cdot |cache-line|$. Po jejím překročení dojde ke zvětšení tabulky. Z experimentů vyšlo najevo, že ke zvětšení dochází při zaplnění z 66% až 75%.

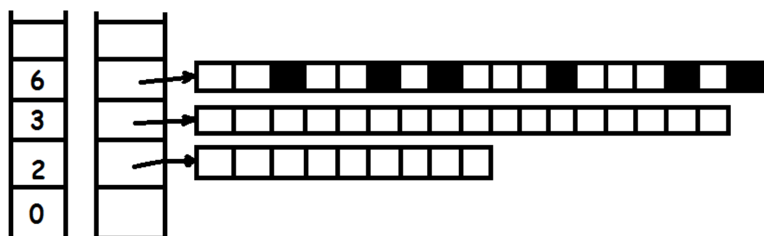
Zvětšování

Tabulka má oproti nezvětšovací verzi dva zámky. Jeden zámek se uzamkne při započetí procesu zvětšení. Každé nové přistoupivší vlákno tak musí počkat, dokud se tento zámek neodemkne. Po alokaci potřebné paměti dojde přehašování všech záznamů ze staršího řádku tabulky do nového řádku tabulky. Haš se v každé buňce může měnit v tomto pořadí:

$$(0) \rightarrow (hash << 1|1) \rightarrow (hash << 1|0) \rightarrow (0|1)$$

Poslední varianta hodnoty haše pak značí přesunutou buňku.

Dobíhající zápisy jsou řešeny tak, že každé vlákno, které vkládá, si po zamknutí zkontroluje, zda se zvětšuje, nebo zvětšení již proběhlo. Pokud se tak stane, vkládání se restartuje a použije se nový řádek řádek.



Obrázek 3.6: Nákres stavu tabulky při 6 vláknech. Zaplněné buňky jsou označeny černou výplní. Vlevo je tabulka počítadel.

Zápisy, které čekají na uvolnění zámku pro porovnání hodnoty jsou řešeny tak, že se restartují, pokud čekací algoritmus narazí na již přesunutou buňku. Tím se vyřeší všechny dobíhající zápisy.

Protože přehašování všech neprázdných buněk může být pro dlouhé řádky časově náročné, je starý řádek rozdělen na segmenty podle počtu jednotlivých vláken. Vlákno, které zvětšuje tabulku, uvolní po alokaci nového řádku druhý zámek. Ostatní vlákna, která čekají na dokončení zvětšení, si po uvolnění druhého zámku rozdělí jednotlivé segmenty a přehašují všechny neprázdné buňky do nového řádku. Po dokončení přehašování segmentu se daný segment označí za přehašovaný. Vlákno, které provede poslední přehašování segmentu, pak uvolní rostoucí zámek.

Odmazávání

Odmazávání starších řádků je v kontextu paralelního běhu problém. Ten je v této tabulce vyřešen tak, že tabulka je rozdělena na dvě části – sdílenou a lokální. Lokální část je pouze nadstavba nad sdílenou tabulkou, jediné, co si pamatuje, je svůj aktuálně používaný řádek. Ten se posílá v metodách do sdílené části tabulky, kde se porovná oproti indexu aktuálně používaného řádku.

Pro korektní uvolnění paměti je vedle tabulky řádků i tabulka počítadel. Každý řádek tak má své počítadlo, které indikuje počet vláken, které mohou řádek používat. Pokud je aktuálně používaný řádek vyšší než lokální, došlo mezi dvěma dotazy do tabulky ke zvětšení. Toto vlákno tedy sníží počítadlo u všech řádků mezi lokálně používaným řádkem a aktuálním řádkem. Pokud někde zůstane po snížení hodnota počítadla nulová, bylo toto vlákno poslední, které odmítlo daný řádek používat a je tedy možné řádek dealokovat.

```
1  Function ReleaseMemory(spodníIndex) is
2      for spodníIndex < aktuálníŘádek do
3          počítadlo[spodníIndex] ← počítadlo[spodníIndex] - 1;
4          if počítadlo[spodníIndex] = 0 then
5              dealokuj řádek;
6      end
7  end
8  Function Rehash() is
9      while je volný segment řádku do
10         for buňka ∈ segment do
11             uzamkni buňku;
12             if buňka není prázdná then
13                 vlož buňku do nového řádku poznač ji za přesunutou;
14         end
15     end
16     if jsem poslední vlákno, které dokončilo přehašování then
17         uvolni zámek pro zvětšování;
18 end
19 Function Grow(indexNovéhoŘádku) is
20     uzamkni zámek pro zvětšování;
21     if tabulka se zvětšila then
22         odemkni zámek pro zvětšování;
23         return;
24     end
25     tabulka[ indexNovéhoŘádku ] ← pole[ 2· velikost tabulky ];
26     počítadlo[ indexNovéhoŘádku ] ← početVláken;
27     povolit přehašování po segemntech;
28     Rehash();
29     ReleaseMemory(indexNovéhoŘádku - 1);
30 end
```

Algoritmus 4: Metody pro nárůst velikosti a uvolnění paměti u tabulky s přehašováním.

```
1  Function Insert(hodnota, haš, lokálníIndexŘádku) is
2      while zvětšuje se do
3          while jsou volné segmenty na přehašování do
4              Rehash();
5          end
6      end
7      if tabulka byla zvětšená then
8          ReleaseMemory(lokálníIndexŘádku);
9          lokálníIndexŘádku ← aktuálníŘádek;
10     end
11     for pokus ← 0 ... maximum pokusů do
12         buňka ← tabulka [lokálníIndexŘádku][ Index(haš, pokus) ];
13         if buňka je prázdná then
14             if lze atomicky vložit haš a zamknout then
15                 if zvětšuje se nebo se již zvětšila tabulka then
16                     odemknout buňku;
17                     restart vkládání;
18                 end
19                 zapiš hodnotu;
20                 odemkni;
21                 return ANO;
22             end
23         end
24         if buňka má stejný haš then
25             while buňka je zamčená do
26                 if buňka byla přesunuta then
27                     restart vkládání;
28                 end
29             end
30             if hodnoty se rovnají then
31                 return NE;
32             end
33         end
34     end
35     Grow(lokálníIndexŘádku+1);
36     restart vkládání;
37 end
```

Algoritmus 5: Metoda pro vkládání.

Kapitola 4

Měření

V této kapitole jsou zaznamenána dvoje měření. První měření je porovnání variant sdílených tabulek. Druhé měření je porovnání nové a staré implementace procházení grafu.

4.1 Výpočetní stroje

Testy byl realizovány na strojích *AURA* a *ANTEA*.

AURA má 64 výpočetních jader Intel Xeon o frekvenci 2,27 GHz a disponuje pamětí zhruba 460 GB. Velký počet procesorových jednotek je řešen architekturou *NUMA*,[1] přičemž jeden paměťový blok sdílí 8 výpočetních jader.

ANTEA má 8 fyzických výpočetních jader se zapnutou technologií *Hyper-Threading*, která přidá dalších 8 výpočetních jader. Procesory mají stejnou frekvenci jako na stroji *AURA*. Na stroji *ANTEA* je k dispozici paměť o velikosti přibližně 24 GB.

4.2 Výběr varianty sdílené tabulky

Před výběrem vhodné tabulky do DiVinE byly všechny navržené tabulky podrobeny testu na rychlost vkládání prvků.

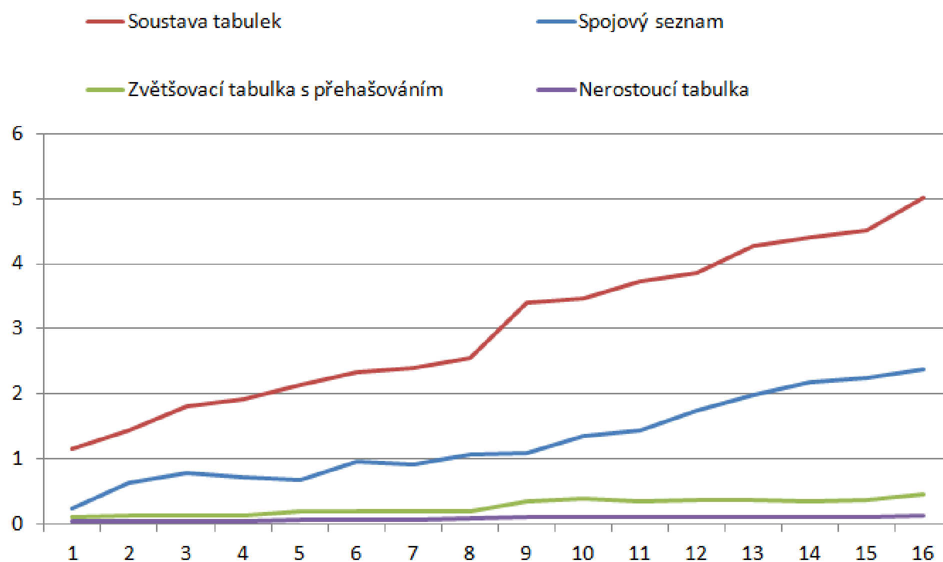
Následující tabulky ukazují časy extrahovaných sdílených tabulek pro 1 – 16 vláken. Každé vlákno vložilo 1.000.000 unikátních hodnot do sdílené tabulky. Tabulce implementované jako spojový seznam byla nastavena počáteční velikost vektoru zkratek na hodnotu 256. Nerostoucí tabulce byla nastavena dostatečná velikost, aby bylo možné uložit všechny hodnoty. Zbylým dvěma tabulkám byla nastavena počáteční velikost na hodnotu 1024.

Na vodorovné ose grafu je počet vláken, na svislé ose je měřený čas uvedený ve vteřinách.

[1] `http://cs.wikipedia.org/wiki/Non-Uniform_Memory_Access`

Stroj *ANTEA*.

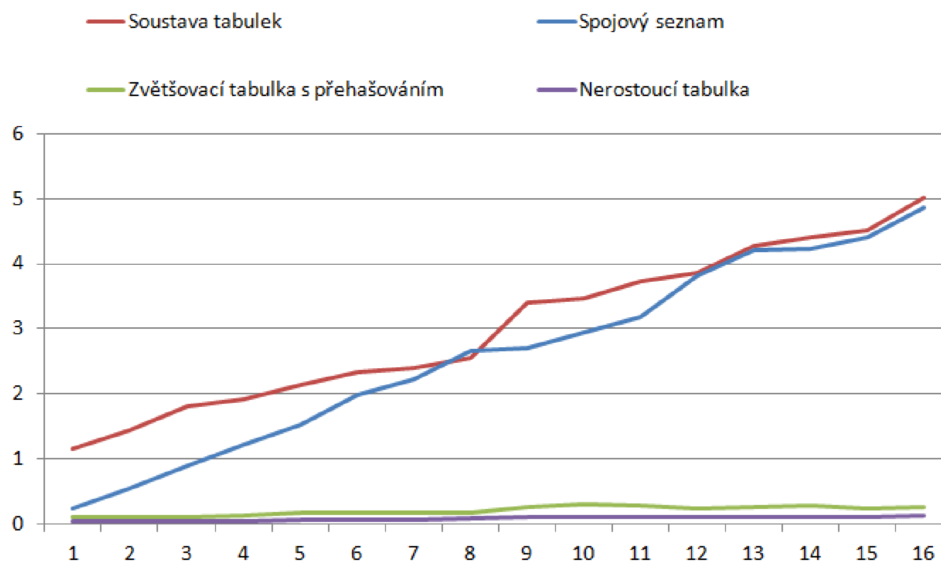
počet vláken	Tabulka Spojový seznam čas (s)	Nerostoucí tabulka čas (s)	Soustava tabulek čas (s)	Soustava tabulek s přehašováním čas (s)
1	0,243	0,034	1,156	0,107
2	0,624	0,041	1,440	0,122
3	0,788	0,050	1,805	0,129
4	0,728	0,051	1,912	0,131
5	0,679	0,071	2,134	0,199
6	0,955	0,070	2,342	0,190
7	0,923	0,070	2,400	0,191
8	1,061	0,081	2,553	0,192
9	1,100	0,118	3,410	0,341
10	1,342	0,113	3,469	0,401
11	1,434	0,113	3,728	0,351
12	1,733	0,110	3,859	0,379
13	1,987	0,111	4,274	0,369
14	2,184	0,110	4,409	0,338
15	2,256	0,110	4,513	0,359
16	2,371	0,129	5,011	0,452



Obrázek 4.1: Měření tabulek na stroji *ANTEA*.

Stroj *AURA*.

počet vláken	Tabulka Spojový seznam čas (s)	Nerostoucí tabulka čas (s)	Soustava tabulek čas (s)	Soustava tabulek s přehašováním čas (s)
1	0,243	0,034	1,156	0,100
2	0,542	0,041	1,440	0,114
3	0,895	0,050	1,805	0,118
4	1,215	0,051	1,912	0,120
5	1,516	0,071	2,134	0,176
6	1,981	0,070	2,342	0,167
7	2,222	0,070	2,400	0,166
8	2,668	0,081	2,553	0,170
9	2,710	0,118	3,410	0,268
10	2,935	0,113	3,469	0,301
11	3,174	0,113	3,728	0,274
12	3,807	0,110	3,859	0,246
13	4,206	0,111	4,274	0,267
14	4,241	0,110	4,409	0,281
15	4,413	0,110	4,513	0,236
16	4,859	0,129	5,011	0,268



Obrázek 4.2: Měření tabulek na stroji *AURA*.

Na základě měření byla pro implementaci v DiVinE vybrána poslední varianta sdílené tabulky – sdílená tabulka s přehašováním. Vkládání do tabulky není výrazně zpomaleno počtem vláken. Zajímavé je, že došlo k mírnému zpomalení při použití více jak 16 vláken. Předpokládám, že na stroji *ANTEA* k tomu došlo, že vlákna 9 – 16 neběžely na plnohodnotných jádrech procesorů, a na stroji *AURA* došlo ke zpomalení z důvodu architektury – jádra

Sdílená tabulka implementovaná pomocí spojového seznamu téměř neškálovala. patrně kvůli neustálým skokům na jednotlivé prvky spojového seznamu docházelo ke zdržení kvůli čekání na paměť.

Sdílená tabulka implementovaná pomocí soustavy tabulek byla rychlá, ale škálovala pouze do doby, než se začala zvětšovat. S každým nárůstem velikosti tabulky bylo nutné pro vložení hodnoty do tabulky projít více buněk v tabulce a to výrazně snižovalo rychlost práce s tabulkou.

4.3 Porovnání procházení grafu

Byly měřeny časy tříd `Partitioned` a `Shared` vzhledem k počtu vláken. Pro testování byly použity dva generátory – `Dummy` generátor a generátor kompilovaného *DVE* (*CESMI*[2]).

`Dummy` generátor generuje dvojce čísel, kdy počáteční hodnota je $(0, 0)$ a konečný stav je v tomto případě[3] $(2^{14}, 2^{14})$. Tento generátor je zahrnutý v testování, protože dosahuje nejvyšší rychlosti při generování – z každé dvojce čísel (x, y) vygeneruje další dvě dvojce $(x + 1, y)$ a $(x, y + 1)$. Další dvojce je vygenerována pouze pokud je x, respektive y menší jak 2^{14}. Tento generátor je testován nad algoritmem *Metrics*, který vždy vygeneruje celý stavový prostor, tedy všechny možné dvojce čísel od 0 po 2^{14} včetně; v součtu je vygenerováno 268.468.225 unikátních stavů.

DiVinE umožňuje verifikovat modely, které byly zkompilovány do dynamicky načítané knihovny. Tato knihovna musí mít rozhraní, které umožňuje získat iniciální stavy grafu a následně pro každý stav určit jeho následníky. Takto definované rozhraní se nazývá *CESMI*. V testu je použité *CESMI*, protože rychlost generování je vyšší než při použití *DVE* interpretru.

Byly provedeny následující sady testů tříd `Partitioned` a `Shared`:

- test nad `Dummy` generátorem pro 1 – 16 vláken
- test nad zkompilovným příkladem fischer.7.dve pro 1 – 32 vláken
- test nad zkompilovaným příkladem peterson.7.dve pro 1 – 32 vláken

[2] Více na `http://divine.fi.muni.cz/manual.html#the-cesmi-specification`

[3] Ve vydávaném DiVinE je konečný stav pouze $(2^9, 2^9)$. Je to z důvodu, že pro běžné otestování stačí řádově méně stavů. Pro testy třídy `Shared` je ale třeba větší zátěž.

Protože je časově náročné přehašovat tabulku, jsou testy třídy `Shared` spouštěny ve dvou konfiguracích – jedna s dostatečně velkou tabulkou, aby nedocházelo ke zvětšování, druhá s tabulkou o standardní velikosti tabulky[4], která se udává v nástroji DiVinE. Pro lepší přesnost byly testy spouštěny opakovaně. Výsledné hodnoty jsou minima z naměřených hodnot.

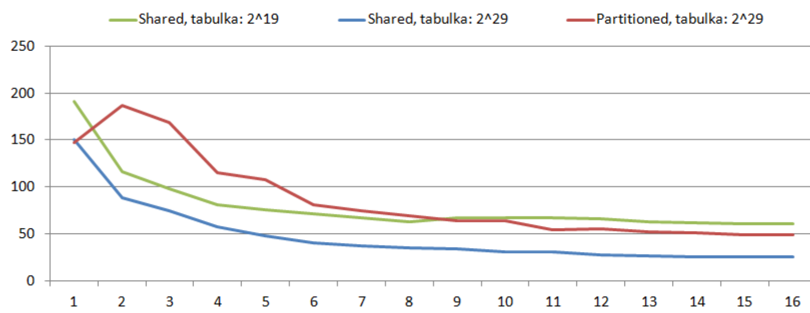
Popisky u grafů jsou následující:

- Na vodorovné ose je počet vláken.
- Pokud je graf měření rychlosti, je na svislé ose uveden čas ve vteřinách.
- Pokud je graf měření zrychlení, je na svislé ose uvedeno relativní zrychlení oproti času na jednom vlákně.

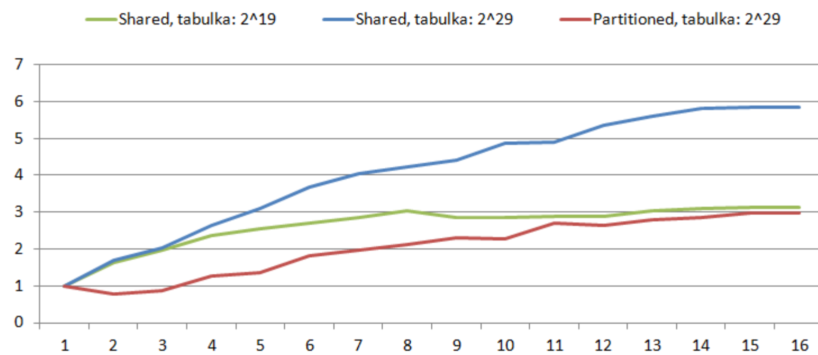
Test `Dummy` generátoru

	`Partitioned`	`Shared`		`Partitioned`	`Shared`	
	AURA			*ANTEA*		
iniciální velikost	2^{29}	2^{29}	2^{19}	2^{29}	2^{29}	2^{19}
počet vláken	čas (s)	čas (s)	čas (s)	čas (s)	čas (s)	čas (s)
1	164,1	209,4	233,132	147,018	150,3	191,5
2	346,9	180,7	215,267	187,197	88,7	116,5
3	342,3	139,2	148,89	168,41	74,3	97,8
4	313,2	117,5	134,206	115,019	57,2	81,3
5	286,2	94,4	108,728	107,936	48,3	75,5
6	239,3	82,1	106,608	81,1941	40,8	71,3
7	208,4	76,8	90,9066	74,8259	37,1	67,2
8	185,7	68,2	95,047	69,2778	35,6	63,1
9	181,1	63,6	91,4796	63,5211	34,1	67,4
10	161,9	59,7	100,517	64,3115	30,9	67,2
11	145,1	58,1	96,943	54,3175	30,6	66,7
12	138,9	59,2	116,381	55,96	28,1	66,4
13	127,7	75,1	166,803	52,5177	26,9	63,3
14	121,6	72,6	147,647	51,633	25,8	61,7
15	110,7	102,3	186,276	49,387	25,7	61,1
16	109,8	144,5	218,62	49,4241	25,7	61,2

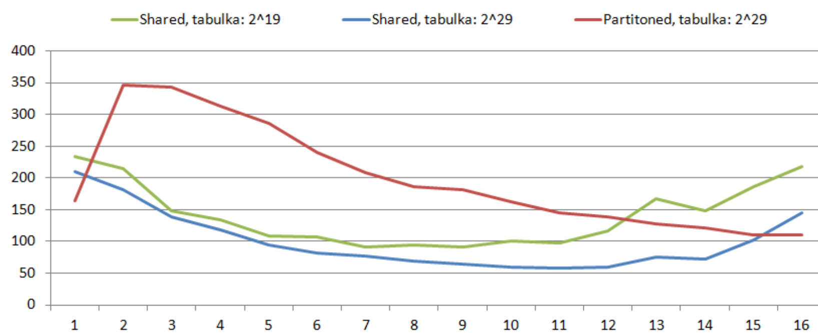
[4]Standardní velikost tabulky je 2^{19}.



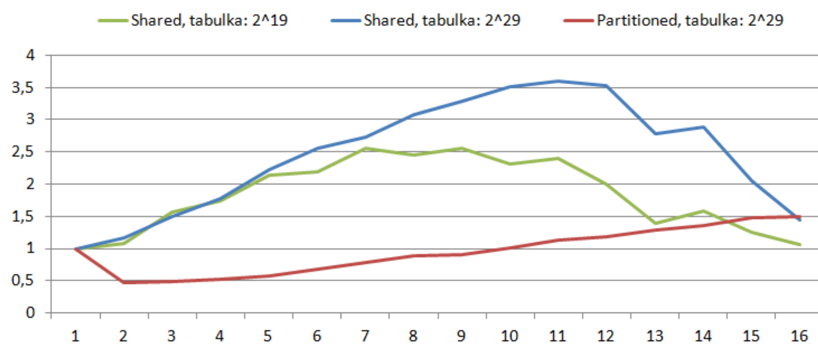
Obrázek 4.3: Čas `Dummy` na stroji *ANTEA*



Obrázek 4.4: Zrychlení `Dummy` na stroji *ANTEA*

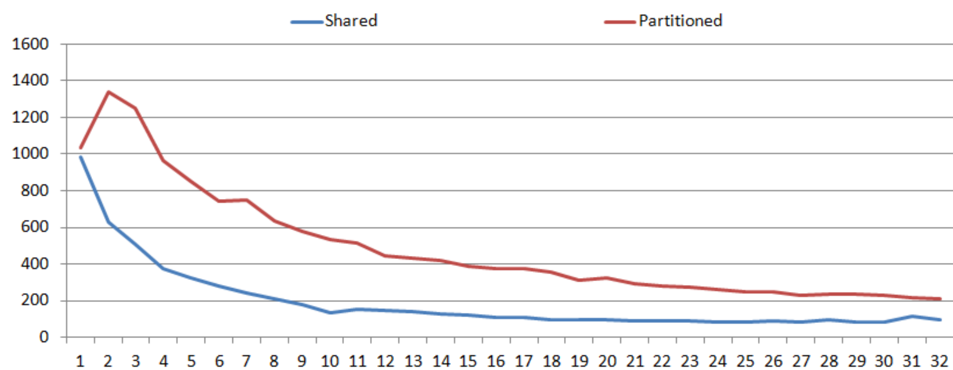


Obrázek 4.5: Čas `Dummy` na stroji *AURA*

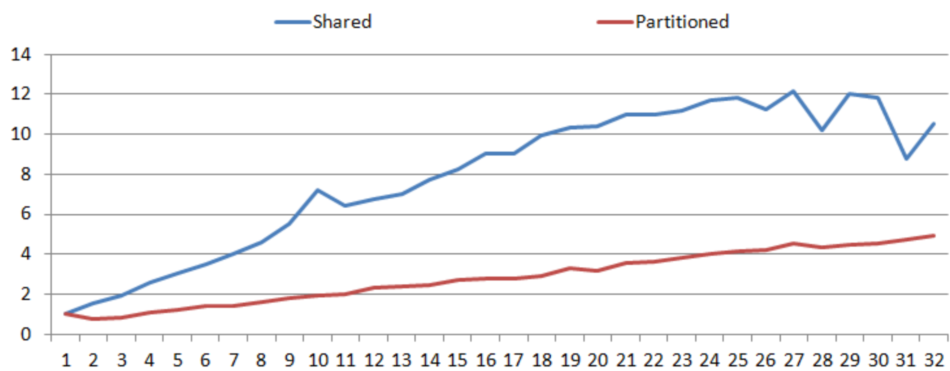


Obrázek 4.6: Zrychlení `Dummy` na stroji *AURA*

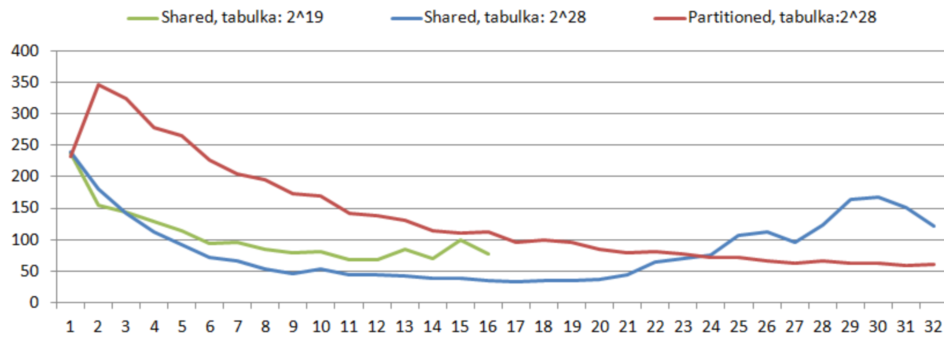
	AURA				
	fischer.7.dve		peterson.7.dve		
	`Partitioned`	`Shared`	`Partitioned`	`Shared`	`Shared`
iniciální velikost	2^{30}	2^{30}	2^{28}	2^{28}	2^{19}
počet vláken	čas (s)	čas (s)	čas (s)	čas (s)	čas (s)
1	1035,95	983,144	232,091	239,966	236,975
2	1336,68	629,615	345,81	180,58	158,875
3	1251,45	508,795	325,086	141,941	154,642
4	964,273	375,404	278,562	113,078	128,712
5	849,779	325,535	264,591	91,5153	114,268
6	744,554	278,542	226,63	71,3366	102,33
7	747,551	243,009	204,131	66,5818	95,5358
8	632,877	212,883	195,602	53,5381	109,403
9	580,991	177,33	173,014	46,1295	79,2253
10	532,168	136,265	169,359	52,9488	80,7311
11	512,398	152,302	142,06	43,5164	68,6084
12	443,779	145,869	138,814	44,3721	68,887
13	429,08	140,687	130,343	42,4346	105,81
14	417,78	127,082	114,476	39,4615	70,5686
15	385,339	119,38	110,73	37,7616	99,0822
16	375,813	109,202	111,611	34,9929	76,4766
17	374,885	108,52	96,7276	33,864	–
18	354,415	98,7033	98,6582	34,4461	–
19	311,76	95,2426	96,7206	34,314	–
20	324,433	94,4477	83,8605	37,3085	–
21	289,85	89,4652	79,7254	43,9693	–
22	283,052	89,7509	80,9331	64,4848	–
23	272,359	87,9431	77,6084	70,8436	–
24	259,148	84,0361	71,6155	74,9149	–
25	249,632	83,2049	71,2404	106,017	–
26	247,618	87,2548	65,9316	112,842	–
27	227,503	80,9135	63,1183	95,5293	–
28	237,233	96,4523	65,4531	123,03	–
29	232,904	81,7388	62,0635	164,391	–
30	227,235	83,3198	62,3151	168,066	–
31	219,592	112,352	58,673	151,23	–
32	208,692	93,6359	60,2475	121,874	–



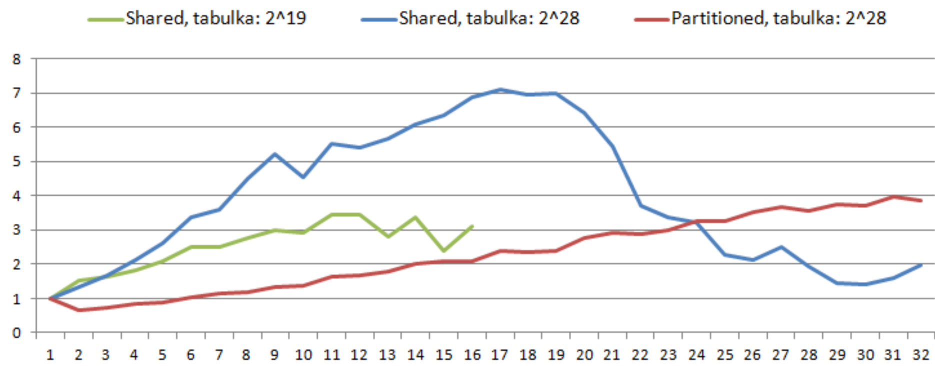
Obrázek 4.7: Čas modelu *fischer.7.dve*



Obrázek 4.8: Zrychlení modelu *fischer.7.dve*



Obrázek 4.9: Čas modelu *peterson.7.dve*



Obrázek 4.10: Zrychlení modelu *peterson.7.dve*

4.4 Výsledky měření

Z výsledků měření vyplývá, že ačkoliv třída `Shared` škáluje lépe než třída `Partitioned`, je rychlost výpočtu velmi závislá na stroji, jeho aktuálním zatížení a velikosti modelu, který je ověřován. Z naměřených dat také vyplývá, že pokud se sdílené tabulce nenastaví dostatečná počáteční, dojde při zvětšování ke konstantnímu zpomalení řádech několika málo desítek vteřin.

Příklad *peterson.7.dve* při použití třídy `Shared` na stroji *AURA* od 18 vláken neškáluje. Špatné škálování lze dát částečně za vinu malému modelu. Důležité ovšem je, že v době testování byl stroj *AURA* vytížen dalšími běžícími procesy, které vytěžovaly paměťové sběrnice.

Kapitola 5

Závěr

Výsledná implementace je zakomponovaná v nástroji DiVinE. Poněvadž třída `Shared` funguje pouze s algoritmy *Reachability* a *Metrics*, je nutné její použití vynutit přepínačem. Během vývoje třídy `Shared` byl vydán DiVinE 3.0 RC 1, kde je třída `Shared` obsažena, ale pouze s původní verzí nezvětšovací tabulky. Na webu[1] o nástroji DiVinE je na třídu `Shared` poukázáno s popisem, že jde o experimentální nasazení.

Podařilo se naplnit zadání práce. Byla úspěšně implementována třída `Shared` a sdílená tabulka, která se umí zvětšovat v paralelním prostředí a umožňuje rychlý přístup. V závěru práce byla prezentována experimentální měření, která potvrzují uvedené závěry.

5.1 Plánovaná rozšíření

Je v plánu rozšířit novou třídu `Shared` tak, aby uměla pracovat i v distribuovaném prostředí pomocí *MPI*. Toto vyžaduje netriviální úpravu třídy a zkoumání napojení stávajících struktur v DiVinE, které se mapují na volání knihovních funkcí MPI. Oproti třídě `Partitioned` používá nově implementovaná třída pro detekci zastavení a veškeré zasílání sdílenou paměť. Tedy bude nutné upravit práci s frontami, detekci zastavení a implementovat přiřazení uzlu jednotlivému *MPI* výpočetnímu uzlu.

Je v plánu umožnit použití třídy `Shared` s algoritmy *MAP* a *OWCTY*. Protože tyto algoritmy v průběhu procházení modifikují dodatečné informace k vrcholům grafu, je potřeba dodatečné informace uzamknout pro exkluzivní přístup. Tyto úpravy byly navrženy a částečně implementovány, ale protože během implementace došlo k rozhodnutí změnit struktury úložišť, alokace paměti a s tím související změny nad jednotlivými reprezentacemi stavů, je implementace zamykání dočasně pozastavena.

Jak bylo již výše zmíněno, souběžně s touto prací vznikala další práce zabývající se stromovou kompresí v tabulkách. Je žádoucí tyto dvě práce

[1] `http://divine.fi.muni.cz`

implementačně propojit, ovšem v čase, který byl v rámci bakalářské práce, nebylo možné provést spojení.

Celý vývoj směřuje k tomu, aby byla třída `Partitioned` nahrazena ve výchozím stavu třídou `Shared`, která je na více vláknech rychlejší.

Literatura

[1] *Split-Ordered Lists – Lock-free Resizable Hash Tables.* SHALEV, Ori a SHAVIT, Nir [online]. © 2006 [citováno 8. 11. 2012]. Dostupné z: <`http://www.cs.ucf.edu/~dcm/Teaching/COT4810-Spring2011/Literature/SplitOrderedLists.pdf`>

[2] *High performance dynamic lock-free hash tables and list-based sets.* MICHAEL, M. Maged [online]. 2002 [citováno 12. 11. 2012]. Dostupné z: <`http://www.research.ibm.com/people/m/michael/spaa-2002.pdf`>

[3] *Lock-free Dynamically Resizable Arrays.* DECHEV, Damian a PIRKELBAUER, Peter a BJARNE, Stroustrup [online]. 2006 [citováno: 8. 2. 2013]. Dostupné z: <`http://www.stroustrup.com/lock-free-vector.pdf`>

[4] *Multi-Threaded Nested DFS.* ROČKAI, Petr [online]. 2008 [citováno 12. 5. 2013]. Dostupné z: <`https://is.muni.cz/th/139761/fi_b/bc_thesis.pdf`>

[5] *Distributed explicit fair cycle detection.* ČERNÁ, Ivana a PELÁNEK, Radek [online]. 2003 [citováno 12. 5. 2013]. Dostupné z <`http://anna.fi.muni.cz/papers/src/public/0b12de9f069f5641c323d1602e3ba2a4.pdf`>

[6] *Accepting Predecessors are Better than Back Edges in Distributed LTL Model-Checking.* BRIM, Luboš a ČERNÁ, Ivana a MORAVEC, Pavel a ŠIMŠA, Jiří [online]. 2004 [citováno 12. 5. 2013]. Dostupné z <`http://www.fi.muni.cz/reports/files/2004/FIMU-RS-2004-09.pdf`>

[7] *Partial Order Reduction in Parallel Model Checking* ROČKAI, Petr [online]. 2010 [citováno 14. 5. 2013]. Dostupné z: <`https://is.muni.cz/th/139761/fi_m/main.pdf`>

[8] *Cppreference.com* [online]. 2000– [citováno 8. 11. 2012]. Dostupné z: <`http://en.cppreference.com/w/`>

[9] *Shared Hash Tables in Parallel Model Checking.* LAARMAN, Alfons a WEBER, Michael a POL, Jaco van der [online]. 23. 4. 2010 [citováno 2. 5. 2013]. Dostupné z: <`http://www.win.tue.nl/ipa/archive/springdays2010/Laarman.pdf`>

[10] *Distributed LTL Model Checking with Hash Compaction*, BARNAT, Jiří a HAVLÍČEK, Jan a ROČKAI, Petr. In *Proceedings of PASM/PDMC 2012*. 2013.

[11] *State space compression for the DiVinE model checker*. ŠTILL, Vladimír. 2013.

[12] *Model checking*. CLARKE, Edmund M. Jr., *Grumberg* Orna, and *Peled*, Doron A. Cambridge, MA, USA: MIT Press, 1999. isbn: 0-262-03270-8.

[13] *Bit Twiddling Hacks*. ANDERSON, Sean Eron. © 1997 – 2005. Dostupné z: <`http://graphics.stanford.edu/~seander/bithacks.html`>