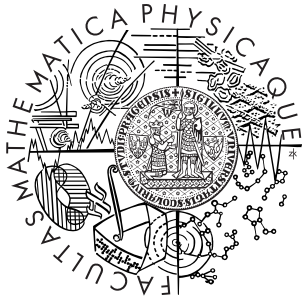
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Zbyněk Falt

Plánovač a paměťový alokátor pro systém Bobox

Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. Jakub Yaghob, Ph.D.
Studijní program: Informatika, obor Softwarové systémy

2010

Rád bych poděkoval RNDr. Jakubu Yaghobovi, Ph.D., za vedení diplomové práce, cenné rady a podnětné připomínky k této práci. Dále bych rád poděkoval Janu Bulánkovi za přínosné náměty k přemýšlení, Elišce Lacinové za jazykové korektury textu a v neposlední řadě i své rodině, která mě nejen během psaní této práce maximálně podporovala.

Prohlašuji, že jsem svou diplomovu práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne 3. srpna 2010 Zbyněk Falt

Obsah

Seznam obrázků

Název práce: Plánovač a paměťový alokátor pro systém Bobox
Autor: Zbyněk Falt
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí bakalářské práce: RNDr. Jakub Yaghob, Ph.D.
E-mail vedoucího: yaghob@ksi.ms.mff.cuni.cz

Abstrakt: Cílem této práce je stručný popis systému Bobox, což je prostředí pro paralelní zpracování dat vyvíjené Katedrou softwarového inženýrství Univerzity Karlovy, analýza výchozího plánovače a návrh nového efektivnějšího plánovače a paměťového alokátoru. Práce rovněž obsahuje experimentální porovnání různých přístupů k této problematice.

Klíčová slova: Bobox, paralelizace, plánovač, paměťový alokátor

Title: Scheduler and memory allocator for the Bobox system
Author: Zbyněk Falt
Department: Department of Software Engineering
Supervisor: RNDr. Jakub Yaghob, Ph.D.
Supervisor's e-mail address: yaghob@ksi.ms.mff.cuni.cz

Abstract: The content of this work is the description of the Bobox system, which is system for parallel data processing developed by the Department of Software Engineering of the Charles University, the analysis of its original scheduler and the design of a new, more efficient task scheduler and memory allocator of the system. The work also contains an experimental comparison of different aproaches to the problematic.

Keywords: Bobox, parallelization, scheduler, memory allocator

Kapitola 1

Úvod

Hlavní metodou, která byla donedávna používaná pro zvýšení výpočetního výkonu procesorů, bylo zvyšování jejich taktovací frekvence. Tento způsob bohužel začal narážet na fyzikální limity používaných technologií výroby. Navíc spolu se zvyšující se frekvencí rostla spotřeba a nároky na chlazení takového systému. Začaly se tedy hledat nové cesty, jak výkon počítačů dále zvyšovat, a přidávání dalších procesorů do systému je jedním z nejefektivnějších způsobů.

Ačkoliv výpočetní výkon systému se takto skutečně zvyšuje (přímo úměrně počtu procesorů), ve většině stávajících programů se toto zvýšení příliš neprojevilo. Zatímco urychlováním systému pomocí zvyšování jeho taktovací frekvence dojde k nárůstu rychlosti všech programů bez jakékoliv úpravy, v případě zvyšování počtu procesorů je bohužel často nutné programy explicitně upravit tak, aby tohoto potenciálu mohly využívat. Nutné úpravy však nemusí být zcela triviální a v některých případech mohou být zcela nemožné. Většinou je totiž nutné přeprogramovat použité algoritmy na jejich paralelní varianty. Ty ale ne vždy musí existovat, neboť některé problémy jsou čistě sériového charakteru[1].

Bohužel existuje mnoho faktorů, které proces implementace paralelního algoritmu značně zesložiťují. Prvním takovým faktorem je různorodost cílových platforem, kdy téměř každý operační systém má odlišné rozhraní pro vytváření vláken, jejich ukončování i jejich vzájemnou synchronizaci. Kromě faktorů softwarového charakteru existuje rovněž mnoho hardwarových, mezi něž patří např. velikost vyrovnávacích pamětí, počet procesorů, rychlost pamětí i celková topologie v případě NUMA[2] systémů, které rychlost již implementovaného algoritmu značně ovlivňují. Je tedy zřejmé, že vlastnosti systémů jsou velmi různorodé a závislé na hardwarové konfiguraci hostitelského systému.

Zahrnout všechny tyto faktory do návrhu a implementace algoritmu obnáší značné úsilí, které může programátora buď od paralelizace zcela odradit, způsobit, že je v případě faktorů hardwarových zanedbá (což se negativně projeví na efektivitě) nebo stráví nezanedbatelné množství času jejich implementací. Z tohoto důvodu existují hotová řešení, která mají za úkol implementaci paralelních algoritmů maximálně usnadnit. Tato řešení mají rozhraní nezávislé na hostitelské platformě, starají se

[1] Např. výpočet posloupnosti částečných součtů zadané posloupnosti.

[2] Non-Uniform Memory Access nebo také Non-Uniform Memory Architecture

o správu vláken i jejich synchronizaci a zároveň se snaží zohledňovat hardwarovou konfiguraci.

Mezi nejznámější a nejrozšířenější technologie patří *Threading building blocks* [13], *OpenMP*[3] [4] nebo *MPI*[4] [15]. Tyto technologie přirozeně neposkytují zcela identické funkce a jejich výběr ovlivňuje charakter implementovaného algoritmu (případně architektury systému). Platí ale, že jsou vhodnější spíše pro implementaci dílčích podúloh.

Naopak systém Bobox [2][3] poskytuje rozhraní pro zpracování dat v paralelním prostředí. Toto rozhraní je ale přirozeně trochu složitější, neboť má sloužit pro řešení složitějších problémů.

1.1 Cíle práce a její členění

Cílem této práce je analýza konkurenčních technologií a podrobná analýza systému Bobox z hlediska plánování a alokace paměti. Získané závěry jsou následně použity k tomu, aby mohl být navržen nový plánovač a alokátor pro Bobox s vlastnostmi, které by měly zefektivnit jeho chod na reálných systémech.

Práce je členěna do kapitol podle ucelených tematických celků. V následující kapitole jsou podrobně rozebrány konkurenční paralelizační technologie. Ve třetí kapitole následuje detailnější popis systému Bobox s důrazem na popis plánování jednotlivých úloh. Ve čtvrté kapitole je popsána metodika měření použitá v dalších kapitolách včetně popisu počítačů, na kterých byly výsledky získávány.

Pátá kapitola podrobně rozebírá a měří vliv jednotlivých faktorů, které je nutné vzít při návrhu plánovače v úvahu. Dále je v ní popsáno rozhraní plánovače a je stručně popsán plánovač výchozí. Ve zbytku kapitoly jsou rozebrány různé plánovací strategie pro různé typy úloh. Šestá kapitola pak popisuje samotnou implementaci nového plánovače. Tento popis se zaměřuje na použité programovací techniky a triky, které byly při implementaci použity.

Sedmá kapitola se zabývá vlivy, které je nutné vzít v úvahu při návrhu alokátoru paměti. Dále rozebírá specifické vlastnosti Boboxu z hlediska alokace pamětí. Kapitola osmá pak popisuje implementaci nového alokátoru, který využívá poznatků z předchozí kapitoly.

Devátá kapitola je věnována problematice, která sice s plánováním a paměťovou alokací příliš nesouvisí, ale souvisí úzce s tím, jak systém může těžit z vlastností plánovače a dosáhnout tak vyššího výkonu. Následně jsou zde teoreticky rozebrány možnosti, jak lze zvýšit stupeň paralelizace při zpracování požadavků a tím dále zvýšit výkon systému.

Poslední kapitola pak shrnuje celou práci tím, že srovnává efektivitu výchozího systému a systému s novým plánovačem a paměťovým alokátorem.

[3] Open Multi-Processing

[4] Message Passing Interface

Kapitola 2

Používané paralelizační technologie

Jak bylo uvedeno již v úvodu, je vývoj programu, který by měl být optimalizovaný pro běh na víceprocesorových systémech, poměrně náročnou úlohou. A to nejen z hlediska návrhu vhodných algoritmů, ale i z hlediska efektivní implementace. Rovněž bylo zmíněno, že existují již hotová řešení, která by implementační část vývoje měla maximálně usnadnit. Při použití těchto řešení pak programátor nemusí řešit, i když může stále ovlivňovat, následující aspekty:

- Zjišťování počtu výkonných jednotek v systému a tomu odpovídající počet vláken, ve kterých mají probíhat výpočty.
- Platformově závislý způsob spouštění těchto vláken, jejich synchronizaci a ukončování.
- Rozdělování problému na dílčí úlohy, prostorovou velikost těchto úloh a jejich přidělování běžícím vláknům.
- Potíže pramenící z toho, že se paralelizované funkce mohou volat rekurzivně.
- Pořadí v jakém jsou dílčí úlohy vykonávány, neboť to může značně ovlivnit efektivitu výpočtu.
- Správu paměti v paralelním prostředí.

V této kapitole je uveden stručný popis rozhraní nejpoužívanějších paralelizačních technologií, následovaný podrobnější popisem použitých plánovačů, případně alokátorů paměti. Technologie MPI v tomto seznamu zahrnutá není, neboť pro komunikaci používá zprávy místo sdílení adresového prostoru, což ji předurčuje pro nasazení spíše v distribuovaném prostředí, takže s cílem této práce příliš nesouvisí.

2.1 Threading building blocks

Threading building blocks [13] (dále jen *TBB*) je knihovna v jazyce C++ vyvinutá firmou Intel Corporation.

Knihovna poskytuje mnoho funkcí a tříd, které jsou určeny pro použití v paralelním prostředí. Jedná se zejména o synchronizační primitiva, různé atomické operace, paměťové alokátory optimalizované pro paralelní prostředí a vláknově bezpečné datové struktury. Kromě toho jsou k dispozici šablonové funkce, které mají za úkol usnadnit implementaci paralelních algoritmů. Pokud se programátor rozhodne použít TBB pro paralelizaci algoritmu, musí nejdříve vybrat takovou funkci, která nejvíce odpovídá charakteru algoritmu, a dále naimplementovat vyžadované třídy a metody.

K dispozici jsou tyto základní funkce:

- `parallel_for` – provede iteraci přes zadaný interval hodnot. Tato iterace může být paralelizována tak, že v jednom okamžiku se provádí několik iterací přes dílčí intervaly najednou. Je zřejmé, že operace pro jednotlivé hodnoty by měly být nezávislé na ostatních.

- `parallel_reduce` – spočítá výsledek funkce, která závisí na všech hodnotách v zadaném intervalu. Takto lze spočítat např. součet všech hodnot v dané posloupnosti. Výpočet je paralelizován tak, že zadaný interval je rozdělen na menší podintervaly, pro ně jsou výsledky počítány paralelně a tyto dílčí výsledky jsou následně (rovněž paralelně) skládány dohromady tak, aby byl získán výsledek operace pro celý interval.

- `parallel_scan` – tato funkce je vhodná, pokud výsledek operace pro jeden prvek závisí na výsledku pro prvek předchozí. Např. výpočet posloupnosti částečných součtů toto pravidlo splňuje. Paralelismu je tentokrát dosaženo tak, že samotný výpočet je rozdělen do dvou fází. Nejdříve je paralelně puštěn předvýpočet pro dílčí podintervaly. Ve druhé fázi je těmto předvýpočtům dodán výsledek pro prvek těsně před začátkem podintervalu. Z předpočítaných hodnot je poté dopočítán kompletní výsledek pro celý interval. Případného urychlení je tedy možné dosáhnout pouze paralelizací předvýpočtu, který ale pro některé operace nelze bez předchozího výsledku provést, resp. předvýpočet nepřináší buď žádné, nebo nepříliš výrazné urychlení.

- `parallel_do` a `parallel_for_each` – provede operaci pro každou zadanou vstupní hodnotu. Obě funkce jsou velmi podobné funkci `parallel_for`. Hlavní rozdíl je ale v tom, že vstupní hodnoty nejsou zadané jako číselný interval, ale pomocí iterátorů.

- `pipeline` – aplikuje posloupnost filtrů na proud vstupních dat. Filtr vždy dostane část vstupních dat, provede zadanou operaci a vrátí výsledná data, která jsou odeslána do dalšího filtru. Paralelizace funguje tak, že jednotlivé filtry jsou aplikovány paralelně.

- `parallel_sort` – setřídí zadanou sekvenci s tím, že pokud jsou k dispozici pomocná vlákna, jsou vytvořeny podúlohy, které jsou pro zvýšení výkonu vykonávány paralelně.

- `parallel_invoke` – spustí paralelně funkce, které dostane jako parametry.

Z tohoto seznamu je zřejmé, že celkový problém je vždy rozdělen na dílčí úlohy. Tyto úlohy jsou následně vykonávány paralelně a na konci je případně spočítán celkový výsledek. Protože počet vláken může být menší než počet úloh, které je třeba vykonat, je nutné, aby součástí TBB byl plánovač, který určuje, které úlohy budou kdy a na jakém vlákně vykonávány.

2.1.1 Plánovač

Algoritmus plánovaní úloh v TBB je v referenční dokumentaci [8] velmi dobře popsaný. Základním mechanismem je, že každá úloha může vygenerovat dílčí podúlohy, a teprve poté, co jsou tyto úlohy vykonány, je řízení vráceno do mateřské úlohy, a v ní je dokončeno zpracování. Je k dispozici i takový režim chování, kdy úloha vygeneruje dílčí podúlohy a navíc úlohu, která se má spustit poté, co všechny podúlohy byly zpracovány. Řízení se tak už nevrací zpět to mateřské úlohy, což umožňuje snížit paměťové nároky, neboť není nutné si mateřské úlohy pamatovat na zásobníku vlákna.

Plánovač má tedy k dispozici množinu naplánovaných úloh, ze které úlohy postupně odebírá a zpracovává je. Tato množina je udržovaná v těchto strukturách:

- Každé vlákno má oboustrannou frontu, která obsahuje ty úlohy, které vznikly na tomto vláknu (úloha může kromě samotného výpočtu vytvořit další úlohy).
- Sdílená fronta, která obsahuje ostatní úlohy. Do této fronty jsou vkládány iniciační úlohy paralelizovaných algoritmů.

V okamžiku, kdy vlákno ukončí zpracování přidělené úlohy a rozhoduje se, kterou další bude vykonávat, vybere první úlohu takovou, která odpovídá některé v následujícím seznamu (tento seznam je zpracováván podle uvedeného pořadí):

1. Úloha, která byla přímo vrácena právě skončenou úlohou.
2. Úloha, která se má spustit, pokud právě ukončená úloha byla poslední z určité skupiny dílčích podúloh.
3. Úloha z konce fronty patřící aktuálnímu vláknu.
4. Úloha, která má explicitně určeno, že má běžet v tomto vlákně
5. Úloha ze začátku sdílené fronty.
6. Úloha ze začátku fronty náhodného vlákna.

Pokud během zpracování úlohy dojde k vytvoření nové, je tato úloha rovnou vložena na konec fronty vlákna, ve kterém byla vytvořena.

Tento postup zajišťuje, že iniciální úlohy jsou plánovány stylem FIFO[1], takže ty jsou vyřizovány spravedlivě, zatímco úlohy, které vznikají při zpracování, jsou zpracovávané stylem LIFO[2], což způsobuje zvýšení vlivu principu lokality dat, a tím

[1] First in, first out
[2] Last in, first out

vyšší využití vyrovnávacích pamětí procesoru (viz část 5.1.1).

2.1.2 Alokátor paměti

Jak bylo zmíněno v úvodu kapitoly, poskytují TBB i vlastní paměťové alokátory. Jsou k dispozici čtyři typy alokátorů:

- `tbb_allocator` – alokátor, který převolává funkce `malloc`, `free` a `realloc` implementované v *TBB malloc* knihovně. V případě nedostupnosti této knihovny volá funkce poskytované runtime knihovnou.
- `scalable_allocator` – alokátor optimalizovaný pro použití ve vícevláknovém prostředí.
- `cache_aligned_allocator` – alokátor, který vrací bloky paměti zarovnané na násobek velikosti jedné cache line. To v důsledku způsobuje, že žádné dva vrácené bloky tímto alokátorem nesdílí cache line, a nedochází tak k tzv. falešnému sdílení dat mezi vlákny.
- `zero_allocator` – pouze dekorátor alokátorů, který vždy při požadavku na přidělení bloku paměti tuto paměť vynuluje.

Z hlediska této práce je nejdůležitější `scalable_allocator`, neboť jako jediný se stará pouze o samotné přidělování paměti. Chování tohoto alokátoru je podrobně popsáno v Intel Technology Journal [11].

Alokátor pracuje se superbloky velikosti 1MB, které alokuje přímo systémovým voláním. Tyto superbloky jsou dále rozděleny na 16KB bloky (zarovnané na adresu dělitelnou 16kB), které jsou vloženy do globálního seznamu volných bloků. V případě, že nastal požadavek na alokaci a není k dispozici žádný volný blok, je naalokovaný další superblok. Platí, že superbloky nejsou vraceny zpátky operačnímu systému. Při požadavku o alokaci paměti se pak podle velikosti zvolí jedna z těchto strategií:

- Požadavky do 8kB – požadavky jsou zpracovány alokátorem.
- Požadavky na 8-64kB – tyto požadavky jsou vyřizovány tak, že se volá přímo funkce operačního systému pro správu virtuální paměti. Tyto bloky jsou při uvolnění ukládány do vyrovnávací paměti pro případný další požadavek.
- Požadavky na více než 64kB - alokace probíhá stejným způsobem jako v předchozím bodu, při uvolnění paměti jsou ale rovnou vraceny operačnímu systému.

Je zřejmé, že nejzajímavější je první bod. Platí, že každé vlákno má pro určité velikosti požadavků (tyto velikosti jsou předem dané) svůj vlastní seznam 16KB bloků (v případě požadavku o přidělení paměti, která velikostí neodpovídá přesně žádné z velikostí, je použita nejbližší vyšší), ze kterého jsou vyřizovány požadavky o alokaci paměti této velikosti. Tyto bloky jsou zřetězeny v dvojsměrném kruhovém spojovém seznamu, kde na začátku jsou bloky s volným místem a na konci bloky zcela

zaplněné. Dále je k dispozici ukazatel na tzv. aktivní blok, což je blok, ze kterého byla naposledy přidělena paměť (ten se nachází mezi zcela plnými a částečně volnými).

Pokud není v seznamu blok, ze kterého je možné přidělit paměť, je převzat nový blok z globálního seznamu. Pokud se některý z bloků stane zcela prázdným, je vrácen zpět do globálního seznamu. Pokud se stane pouze částečně prázdným, je vložen do seznamu před aktivní blok. Díky tomuto mechanismu je možné vždy rychle najít blok, ve kterém je ještě volné místo – stačí se posunout o jeden blok k začátku seznamu.

Každý blok má dále svoji hlavičku (při uvolnění paměti ji lze snadno nalézt, neboť leží na adrese zarovnané na 16kB). V ní jsou uloženy ukazatele na předchůdce a následníka ve spojovém seznamu bloků, dále se zde nachází seznam volných míst v bloku. Tyto seznamy jsou dva – veřejný, do kterého se vrací volné úseky uvolněné v jiném vláknu, a soukromý, do kterého se vrací úseky uvolněné ve vlastnícím vlákně. Paměť se nejdříve přiděluje z privátního seznamu, ke kterému je přístup nesynchronizovaný, a teprve poté ze synchronizovaného veřejného.

2.2 OpenMP

OpenMP [4] je knihovna a zároveň rozšíření programovacího jazyka. Existuje specifikace pro jazyky `C`, `C++` a `Fortran` a v nejrozšířenějších překladačích těchto jazyků je OpenMP podporováno.

Zatímco v TBB je nutné implementovat metody různých tříd a tyto metody jsou spouštěny paralelně, v OpenMP se přímo do zdrojového kódu zapisuje, co a jak má být paralelizováno. Velkou výhodou je, že pro zápis je použita direktiva `#pragma`, která je určena právě pro dodatečná rozšíření překladačů. Překladače, které některá z rozšíření nepodporují tuto direktivu jednoduše ignorují. OpenMP je navrženo tak, že pokud jej překladač nepodporuje, stále může zdrojový kód bez problému přeložit. Výsledek akorát nebude paralelizovaný.

Při použití knihovních funkcí (např. pro zjištění počtu paralelních vláken) tato kompatibilita již nefunguje, ale zpravidla lze vystačit pouze s direktivami.

Existuje velké množství direktiv OpenMP a jejich možných parametrů. Kromě těch, které řídí samotnou paralelizaci, jsou k dispozici direktivy pro synchronizaci vláken, např. `#pragma omp barrier`, synchronizaci hodnot proměnných s hlavní pamětí, např. `#pragma omp flush`, pro sdílení proměnných mezi vlákny atd. Narozdíl od TBB ale nejsou k dispozici funkce pro alokaci a uvolnění paměti, které by byly optimalizované pro použití v paralelním prostředí.

Hlavní direktivy, které řídí paralelizaci, jsou tyto:

- `#pragma omp parallel` – následující blok příkazů bude vykonán paralelně, tj. bude spuštěn takový počet vláken, který odpovídá počtu výkonných jednotek v systému a všechna začnou najednou vykonávat tento blok. Aby bylo možné mezi těmito vlákny rozlišit a přidělit každému z vláken jiný úkol, je k dispozici funkce, která vrátí číslo vlákna, ze kterého byla tato funkce zavolána.

- `#pragma omp parallel for` – následující for-cyklus bude paralelizován, tzn. že bude rozdělen na více dílčích částí (na tolik částí, kolik je vláken), a tyto části budou spuštěny paralelně.

Implicitně se paralelizace for-cyklu provádí tak, že celý interval hodnot, přes který se iteruje, je rozdělen rovnoměrně na tolik částí, kolik je k dispozici vláken. Toto chování ale může být v některých situacích nevhodné. Na konci cyklu totiž dochází k synchronizaci vláken a mohlo by se stát, že zatímco některá vlákna již skončila, je stále nutné čekat, až svou práci dokončí i to poslední. To se může snadno stát tehdy, kdy operace pro každý prvek trvají různou dobu.

Existují dvě možnosti, jak dobu tohoto čekání snížit. První, kterou OpenMP nabízí, je, že lze explicitně zamezit, aby na sebe vlákna čekala. Tuto možnost lze použít např. tehdy, pokud za jedným paralelizovaným for-cyklem následuje další paralelizovaný a neexistují mezi nimi datové závislosti. Druhou možností je změnit způsob, kterým je interval rozdělován a kterým jsou úseky přidělované jednotlivým vláknům. Různé typy plánovačů jsou popsány v následující části.

2.2.1 Plánovač

V OpenMP existují následující typy plánovačů, které ovlivňují způsob rozdělování intervalů na jednotlivé části:

- *static* – interval je rozdělen na úseky zadané délky a takto vzniklé useky jsou metodou round-robin plánovány na jednotlivá vlákna. Je tedy dopředu určeno, které úseky bude které vlákno počítat.

- *dynamic* – vlákno si při ukončení požadavku samo řekne o další část. Rovněž je možné ovlivnit velikost úseku, který bude vláknům přidělen. Pokud se granularita zvolí relativně malá, sníží se doba čekání na poslední vlákno, neboť vlákna zpracovávají malé úseky místo velkých. Na druhou stranu čím menší úseky jsou, tím větší je režie na plánování a samotné spouštění požadavků.

- *guided* – stejné jako dynamic, velikost úseku je ale spočítána na základě zbývající části (vždy se vezme $\frac{1}{K}$-tina zbývajícího úseku, kde K je počet vláken) a úsek není nikdy kratší, než zadaná délka. Tento algoritmus odstraňuje vysokou režii a zároveň zachovává vlastnost, že ke konci již vlákna zpracovávají malé úseky, takže doba čekání na poslední vlákno je kratší.

- *auto* – plánovací algoritmus je určen konkrétní implementací.

Ve všech těchto případech je plánovač poměrně jednoduchý, neboť spolu úseky nijak nesouvisí a ani nemohou, neboť jinak by cyklus nešel takto jednoduše paralelizovat. Vždy se tedy jednoduše spustí úsek následující za posledním zpracovaným, resp. právě zpracovávaným.

Ve verzi 3.0 přibyla podpora tzv. *úloh* (tasks), což jsou operace, které mohou být vykonány paralelně, a pouze plánovač rozhoduje o tom, kdy budou vykonány.

Samozřejmě, že jsou k dispozici direktivy i pro synchronizaci těchto úloh, aby bylo možné zaručit, že požadované úlohy již byly vykonány.

V OpenMP může plánovač vykonávání úlohy pozastavit, pustit jinou a pak pokračovat dále. S touto vlastností souvisí pojem *vázané* (tied) a *nevázané* (untied) úlohy. Pro vázané úlohy platí, že pokud jsou pozastaveny, tak budou vždy spuštěny znovu na stejném vláknu. Pro nevázané úlohy toto pravidlo neplatí.

Je zřejmé, že rozšířením OpenMP o podporu úloh se změnily nároky na plánovač, který již nemůže být tak jednoduchý jako bez nich. Bohužel specifikace OpenMP nic neříká o tom, jak má být plánovač implementován, a algoritmus plánování ponechává na konkrétní implementaci. V článku [7] jsou ale navzájem porovnány různé plánovací algoritmy, které byly v OpenMP implementovány a testovány. Tyto algoritmy jsou rozděleny do dvou základních skupin:

- Tzv. breadth-first algoritmy, které obsahují sdílený seznam všech úloh k vykonání. Z tohoto seznamu si vlákna postupně odebírají úlohy, které mají vykonat. Kromě sdíleného seznamu má navíc každé vlákno svůj lokální seznam úloh, ve kterém jsou uloženy pozastavené vázané úlohy. Pokud vlákno dokončí svoji práci, zkouší nejdříve hledat úlohu ve svém lokálním seznamu a v případě neúspěchu v seznamu sdíleném. Pokud úloha vytvoří jinou úlohu, je tato automaticky vložena do sdíleného seznamu.

- Tzv. work-first algoritmy, jejichž cílem je snaha využít maximálně principu lokality dat (viz část 5.1.1) tak, že v okamžiku, kdy úloha vytvoří jinou úlohu, je pozastavena a řízení je předáno nově vytvořené úloze. Po jejím ukončení se řízení vrátí zpátky do původní úlohy. Pozastavené úlohy se podobně jako v předchozím případě vkládají do lokálního seznamu každého vlákna. Sdílený seznam v tomto případě neexistuje. Pokud vlákno dokončí úlohu, podívá se nejdříve do svého seznamu a pokud je tento seznam prázdný, zkouší ukrást úlohu z jiného vlákna. Vlákna, od nichž se snaží úlohu ukrást, zkouší metodou round-robin a platí, že pozastavené vázané úlohy ukrást nelze. Ještě nespuštěné vázané úlohy ale samozřejmě ukrást lze.

Aby délka seznamu s úlohami k vykonání nebyla příliš vysoká, jsou dále implementovány algoritmy, které mají za úkol celkový počet úloh omezovat. Hlavně u prvního popsaného algoritmu totiž hrozí, že počet úloh bude vysoký, neboť se do seznamu vkládají všechny nově vytvořené úlohy. Ve druhém případě se do seznamu vkládají pouze úlohy pozastavené, takže nadměrný nárůst velikosti seznamu není tak pravděpodobný.

První algoritmus omezuje velikost seznamu tak, že pokud obsahuje více úloh, než určité číslo (konstanta vynásobená počtem vláken), jsou nově vytvářené úlohy rovnou vykonávány přímým voláním. Druhý algoritmus postupuje způsobem, že úloha je vykonána rovnou, pokud má více než předem zvolený počet předků. Takto rovnou vykonávané úlohy již logicky nemohou být pozastaveny, neboť se nejedná o úlohy v pravém slova smyslu, ale o přímá volání.

Tyto algoritmy jsou navzájem otestovány, přičemž zmíněné seznamy jsou implementovány jednou jako zásobník a podruhé jako fronty. Článek v závěru doporučuje

používat jako implicitní breadth-first plánovač, který ačkoliv je méně efektivní než work-first, nabízí vyšší výkon při práci s vázanými úlohami, za které jsou implicitně považovány všechny úlohy v OpenMP (nevázané úlohy musí být explicitně označeny).

Kapitola 3

Bobox

Bobox je systém vyvíjený katedrou softwarového inženýrství na Matematicko-fyzikální fakultě Univerzity Karlovy. Cílem tohoto systému je, narozdíl od předchozích technologií, poskytnout prostředí nikoliv pro implementaci jednotlivých částí programů, nýbrž prostředí pro paralelní zpracování dat. Z předchozích technologií je Bobox nejvíce podobný funkci `pipeline` z knihovny TBB. Narozdíl od této funkce ale jednotlivé filtry nemusí být uspořádány lineárně. Jeden filtr tak může mít více vstupů i více výstupů.

3.1 Architektura systému

Celý systém je rozdělen na dvě základní části. První část, *back-end*, distribuuje a provádí samotné výpočty, zatímco druhá část, *front-end*, poskytuje programy (v dalším textu označované jako *požadavky*), které mají být vykonávány. Zatímco back-end je pouze jeden a tvoří jádro celého systému, tak front-endů může být neomezeně mnoho. V současné době jsou implementovány tři. Jeden, který převádí dotazy v jazyku XQuery [18] na programy pro Bobox, druhý překládá dotazy jazyka TriQ [6] a třetí totéž pro jazyk SPARQL [17].

Požadavek pro Bobox, který front-end předává back-endu k vyhodnocení, je tvořen orientovaným grafem, v jehož vrcholech se nacházejí tzv. *krabičky*, které zapouzdřují dílčí operace s daty. Tyto krabičky přijímají vstupní data, která přicházejí po vstupních hranách, vykonávají nad nimi implementovanou operaci a výsledek posílají do dalších krabiček, které jsou na konci jejich výstupních hran. Vstupním bodem programu je tzv. iniciační krabička, která nemá žádný vstup a je logicky první v topologickém uspořádání grafu. Na konci je pak tzv. terminační krabička, která sděluje systému, že výpočet byl ukončen.

Důležité je, že krabičky jsou programovány proti veřejnému rozhraní, takže front-end nemusí být závislý pouze na předprogramovaných krabičkách, ale může disponovat vlastní sadou implementací. To činí systém snadno rozšiřitelným o další možné front-endy.

Ve skutečnosti back-end od front-endu nedostává graf, v jehož uzlech jsou již instanciované krabičky, ale dostává tzv. *model*. Tento model je rovněž tvořen gra-

fem. V jeho uzlech jsou ale instance továrních tříd, které krabičky před spuštěním požadavku teprve instanciují a pospojují hranami. Navíc je graf požadavku mírně komplikovanější, než bylo zmíněno. Jsou v něm totiž zastoupeny dva druhy krabiček. Jeden druh byl již uveden – jsou to krabičky, které provádějí operace s daty. Tyto krabičky jsou potomky třídy `box` a dále budou označovány jako *výkonné krabičky*. Druhý typ krabiček jsou tzv. *směrovací krabičky*, což jsou instance třídy `via`.

Hlavní rozdíl mezi těmito druhy je v omezení na počet vstupních a výstupních hran. Zatímco výkonné krabičky mohou mít libovolný počet vstupů, tak směrovací krabičky mohou mít pouze jeden vstup. S výstupy je to přesně naopak. Každá výkonná krabička má vždy pouze jeden výstup (kromě krabičky terminační, která žádný výstup nemá), zatímco směrovací mohou mít výstupů libovolný počet. Z výše uvedeného plyne i jejich úloha v celém systému. Výkonné krabičky zpracovávají vstupní data a směrovací krabičky data pouze distribuují do ostatních výkonných. V grafu vždy platí, že hrana z výkonné krabičky vede do směrovací a hrany ze směrovacích vedou do výkonných.

Příklad grafu vygenerovaného pro XQuery dotaz:

```
let $auction := doc("input.xml") return|
  for $person in $auction/site/people/person[@id = "person0"]
    return $person/name/text()
```

je na obrázku 3.1. Kvůli úspoře místa v grafu nejsou zakresleny směrovací krabičky samostatně, ale jsou sloučeny s odpovídajícími výkonnými.

Veškerá data, která proudí mezi krabičkami, jsou zapouzdřena v tzv. obálkách, což jsou instance třídy `envelope`. Struktura dat v jedné obálce je taková, že se jedná o vektor sdílených ukazatelů na tzv. sloupce. Každý sloupec se pak skládá z několika záznamů. Tyto záznamy jsou z pohledu back-endu dále nedělitelná binární data. Platí, že každý sloupec v obálce obsahuje stejný počet těchto záznamů. Množina i-tých záznamů všech sloupců se nazývá i-tým *řádkem* obálky.

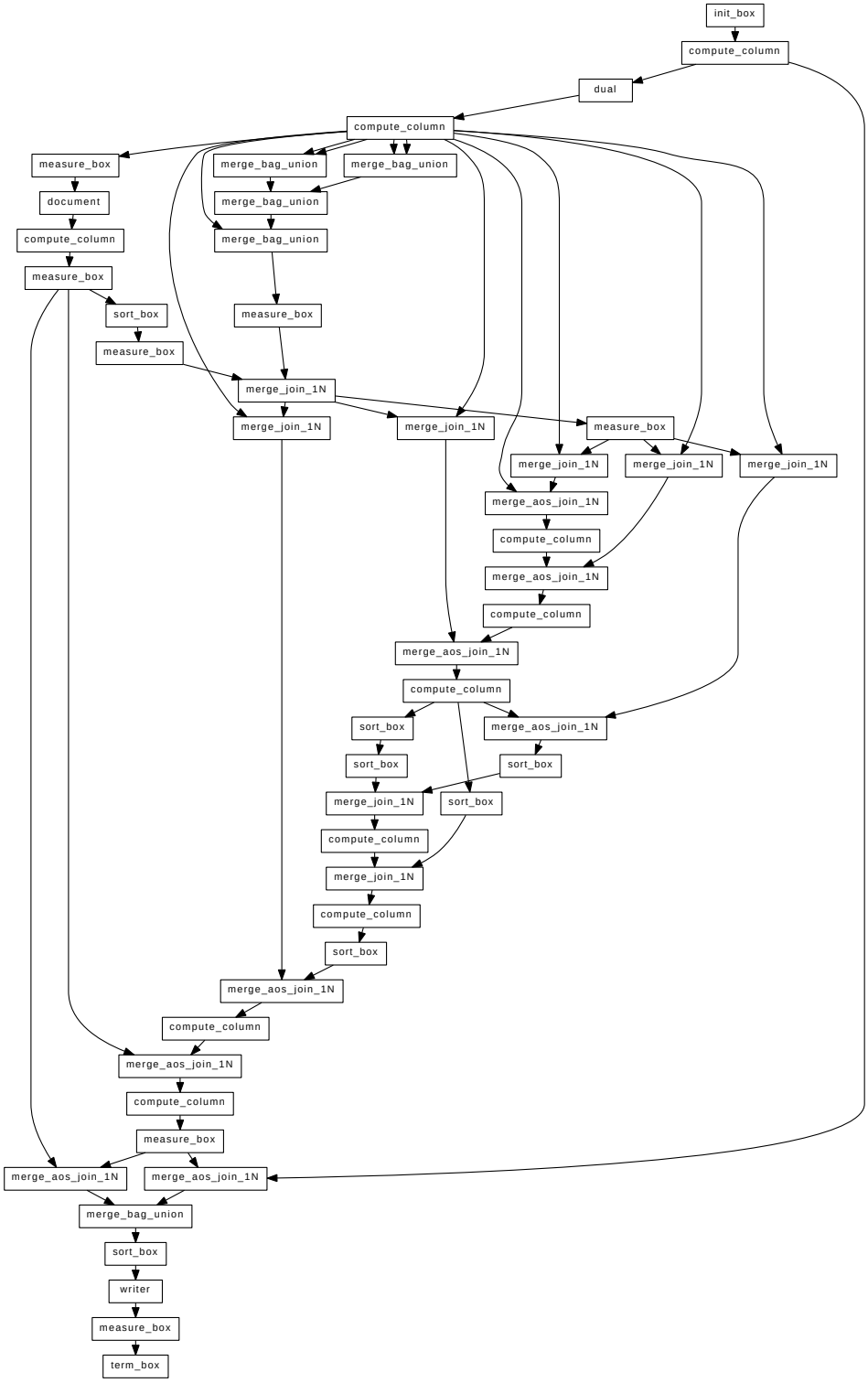
Kromě řádků může obálka obsahovat tzv. *skalární data*, jejichž formát je nezávislý na formátu jednotlivých sloupců. Na tato data se dá pohlížet i tak, že se jedná o další sloupce, které ale obsahují pouze jeden záznam.

Záleží pouze na front-endu, jaká data jsou v obálkách uložena. Může se jednat o obyčejná binární data, o řetězce nebo např. o ukazatele na jiné struktury. Z hlediska plánovače je důležité, že interpretaci těchto dat nemusí řešit. Nemusí ani řešit synchronizaci dat, pokud s daty v obálce pracuje více krabiček. Synchronizace přístupu k datům v obálkách je tedy záležitostí krabiček, nikoliv plánovače.

Za zmínku stojí fakt, že sloupce mohou být sdílené mezi více obálkami a že se může měnit počet sloupců v obálce. Nelze spoléhat ani na to, že počet řádků v obálce nebude krabičkou změněn.

3.2 Plánování krabiček

Vyhodnocování programu probíhá zjednodušeně tak, že se spustí iniciační krabička. Ta vygeneruje data, která jsou rozeslána krabičkám, do nichž z ní vede hrana. Ná-



Obrázek 3.1: Příklad jednoho požadavku v systému Bobox

sledně jsou spuštěny všechny krabičky, které mají nějaká data ke zpracovaní. Tyto krabičky pak opět rozešlou data dalším krabičkám atd., dokud nejsou všechna data zpracována a terminační krabička neoznámí konec výpočtu.

Ve skutečnosti není krabička při příjmu obálky okamžitě spuštěna. Je pouze *naplánována*, což pro systém znamená, že krabička má nějaká data připravená ke zpracování, a systém až se rozhodne, může spustit její kód, který data zpracuje. Tento fakt je klíčový, neboť takto je možné spouštět krabičky na různých logických procesorech počítače, a tím dosáhnout paralelizace výpočtu.

Pro plánování a spouštění krabiček platí několik důležitých pravidel. Jedním z nejdůležitějších je to, že každá krabička může v jednom okamžiku běžet v nejvýše jednom vlákně. Není tak nutné řešit synchronizaci přístupu k vlastním datům krabičky a v případě, že obálky neobsahují sdílená data, není nutné řešit ani synchronizaci přístupu k datům v obálce.

Dalším pravidlem je, že každá krabička je naplánovaná v jednom okamžiku nejvýše jednou. Pokud přijde více požadavků na naplánování krabičky a ta je již naplánovaná, nic se nestane. Pokud přijde požadavek na naplánování ve chvíli, kdy je krabička zrovna spuštěna, je tento požadavek uložen, a v okamžiku ukončení výpočtu krabičky je ihned znovu naplánována.

Krabičky nejsou jediné objekty, které mohou zažádat o přidělení výpočetního času. Naplánovaná a spuštěná může být libovolná instance třídy `task`. Ve výchozí implementaci ale od této třídy dědí pouze zmíněné `box` a `via`.

3.3 Manipulace s daty

Data ve skutečnosti nejsou posílána celá najednou, ale jsou rozdělena na menší části. Tyto části jsou pak zabaleny do obálek a v této podobě dochází k přenosu dat. Každá datová obálka tak obsahuje určité celočíselné množství řádků (navíc může volitelně obsahovat skalární data). Počet řádků v obálkách není nijak explicitně určen, takže se může lišit a záleží jenom na krabičkách, kolik řádků v obálce se rozhodnou odeslat.

Aby krabičky mohly snadno poznat, že nemají očekávat další příchozí data, odesílá vždy každá krabička poté, co již nehodlá další data odesílat, tzv. *otrávenou obálku*. Příjem této obálky tak signalizuje, že do krabičky již žádná další data nepřijdou. Např. zmíněná terminační krabička na otrávenou obálku reaguje tak, že systému oznámí, že výpočet skončil.

Díky zpracování dat po částech mohou být data zpracovávaná proudově. Tento přístup je vhodný pro určité typy operací, jako např. slévání. Není tak nutné mít v jediném okamžiku všechna data v paměti.

Na druhou stranu jsou operace, např. třídění, které potřebují veškerá data najednou. Tyto operace nejsou z hlediska implementace problematické. Stačí přijímat příchozí obálky a někam je ukládat. Při příjmu otrávené obálky pak lze spustit požadovanou operaci.

Důležité pravidlo, které dále platí, je, že krabička odesílá obálky jednotlivě. To znamená, že operace odeslání odešle nejvýše jednu obálku. Není ale zakázané odeslat několik jednotlivých obálek za sebou. Tato vlastnost nepřináší žádná omezení, neboť

pokud chce krabička odeslat větší množství dat najednou, může data spojit do jedné větší obálky. Dále má každá krabička vlastní vyrovnávací paměť pro příchozí a odchozí obálky. Pokud je vyrovnávací paměť příchozích obálek zaplněná, má krabička právo další příchozí obálku odmítnout.

3.4 Souvislost plánování s tokem dat v systému

Je zřejmé, že předchozí dvě části textu spolu velmi úzce souvisí, neboť plánování krabiček je závislé na tom, jak krabičky odesílají a přijímají obálky.

Hlavní metoda, která se volá při požadavku na naplánování libovolné úlohy, je `task::schedule(bool immediate)`. Tato metoda přidá úlohu do seznamu naplánovaných úloh, případně vyřeší případy, kdy je úloha již naplánována nebo právě běží. Parametr `immediate` určuje, zda má být úloha spuštěna pokud možno co nejdříve.

Z pohledu krabiček a dat tento parametr určuje příčinu naplánování krabičky. Systém totiž dodržuje pravidlo, že pokud byla krabička naplánovaná z důvodu příjmu obálky, je zavolána funkce `schedule` s tímto parametrem rovným `true`. Taková úloha bude dále označovaná jako *datově vázaná úloha*. Pokud byla naplánována z jiného důvodu, je tento parametr roven `false`. Taková úloha bude označována jako *datově nevázaná úloha*.

Motivace tohoto pravidla je taková, že když dojde k odeslání/příjmu obálky, je tato obálka uložena v cache procesoru, takže další manipulace s ní bude tím rychlejší, čím dřív se s ní začne pracovat. Další situace, kdy je krabička naplánována s požadavkem co nejrychlejšího spuštění, nastává tehdy, kdy se některé krabičce (směrovací nebo výkonné) uvolní místo ve vstupní vyrovnávací paměti pro příchozí obálky poté, co odmítla přijmout další obálku. Tehdy je naplánována zdrojová krabička této odmítnuté obálky. Toto chování je ale v nové implementaci změněno a takto vzniklá úloha je plánována jako datově nevázaná, neboť se ve skutečnosti nejedná o odeslání obálky (viz 5.4.1).

Ve všech ostatních případech nemá požadavek na okamžité spuštění žádné opodstatnění. Jmenovitě se jedná o tyto situace:

- Pokud má výkonná krabička zaplněnou výstupní vyrovnávací paměť pro obálky, pak nelze pustit samotnou operaci s příchozími daty, neboť by nebylo kam uložit výsledek. Jestliže se ale podaří tuto paměť aspoň částečně uvolnit, je krabička opět naplánována, aby mohla pokračovat dál ve své práci.

- Pokud výkonná krabička zpracovala obálku ze vstupu, ale ve vstupní vyrovnávací paměti stále zůstávají obálky ke zpracování.

- Jestliže směrovací krabička přeposlala obálku ze své vstupní vyrovnávací paměti a tato paměť stále není prázdná.

Je vidět, že tyto úlohy nesouvisí s příjmem obálky, ale pouze řeší okrajové situace. Je tedy jedno, kdo takovou úlohu vykoná. Ale právě tyto úlohy jsou klíčové pro paralelizaci výpočtu, neboť např. zdrojová krabička dokud neodešle všechna data

takto stále vytváří datově nevázané úlohy a při každém jejím spuštění krabičky je odeslána další obálka. Takto mohou obálky zpracovávat paralelně různá vlákna.

Záleží samozřejmě pouze na plánovači, jak hodnoty parametru `immediate` bude interpretovat. Na druhou stranu je to jediná informace, kterou plánovač o charakteru úloh dostává, a její respektování může výkon systému zvýšit.

Kapitola 4

Metodika měření a testování

Protože ve zbývající části práce budou uváděny výsledky různých měření a experimentů, je nutné popsat, jakým způsobem byly uvedené výsledky získány.

Téměř všechna konkrétní čísla uvedená při porovnávání různých přístupů znamenají dobu trvání měřené operace. Měření doby běhu programu je ale poměrně složité, neboť během měření neběží na počítači pouze měřený program, ale i operační systém a různé jeho služby. Protože se může během měření stát, že operační systém musí obsloužit např. přerušení, načítat data z disku, vyprazdňovat vyrovnávací paměti pro diskové operace nebo některá ze služeb začne spontánně vyvíjet činnost, jsou měření vždy zatížena chybou.

Proto je každá operace spuštěna 10× a celkový výsledek je získán na základě jednotlivých výsledků. Protože občas některý ze zmíněných vlivů způsobí poměrně velkou odchylku některého z dílčích výsledků, není počítán průměr těchto hodnot, ale jejich medián, který je vůči takovým chybám mnohem odolnější. Znamená to tedy, že všechny hodnoty vznikly tak, že byla 10× změřena doba běhu operace, výsledných 10 hodnot bylo setříděno podle velikosti a průměr prostředních dvou pak tvoří uvedený výsledek.

Pokud je uvedena hodnota, která byla změřena na systému Bobox, pak je vždy uvedena pouze celková doba běhu požadavků. Do tohoto času tak není zahrnut překlad požadavků, jejich instanciace a parsování vstupního souboru. To umožňuje porovnávat skutečně jenom vliv plánovače a alokátoru. Navíc se do měření nezapočítávají diskové operace, které by chybu měření dále zvyšovaly.

Požadavky pro systém Bobox jsou vygenerovány pomocí front-endu, který překládá jazyk XQuery do programu pro Bobox. Ve všech měřeních je rovněž použita alternativní implementace třídící krabičky (viz 9.1), neboť ta lépe pracuje s vyrovnávací pamětí, a mohou se tak lépe projevit rozdílné plánovací strategie.

Veškeré časové údaje jsou uvedené v sekundách.

4.1 Testovací hardware

Většina měření byla provedena na čtyřech různých počítačích. Jejich konfigurace je shrnuta v tabulce 4.1.

Označení	OS	CPU	Logických procesorů	Uzlů	Cache	Paměť
Windows-2	MS Windows 7	Intel Core2 Duo 2,80Ghz	2	1	L1: 32kB+32kB L2: 6MB sdílená	4GB
Linux-8	Linux	2x Intel Xeon E5310 1,60Ghz	8	1	L1: 32kB+32kB L2: 2x4MB sdílená	8GB
Linux-4x6	Linux	4x AMD Opteron 8431 2,40Ghz	24	4	L1: 64kB+64kB L2: 512kB L3: 6MB sdílená	2x32GB 2x16GB
Linux-4x6HT	Linux	4x Intel Xeon E7540 2,00Ghz	48	4	L1: 32kB+32kB L2: 256kB L3: 18MB sdílená	4x32GB

Tabulka 4.1: Hardwarová konfigurace testovacích počítačů

Položka *Označení* udává jméno počítače, pod kterým se na něj v textu bude odvolávat. Položka *Uzlů* určuje počet uzlů z pohledu NUMA systémů. U velikosti L1 cache je vždy zvlášť uvedena velikost vyrovnávací paměti pro data a instrukce. Velikost paměti pak určuje velikost paměti jednotlivých uzlů.

Vzhledem k tomu, že pouze některé počítače disponují určitými vlastnostmi, budou některá měření provedena pouze na nich. Např. vlastnost NUMA má pouze *Linux-4x6* a *Linux-4x6HT*. Technologií Hyper-Threading pak disponuje pouze *Linux-4x6HT*.

Počítače s operačním systémem Linux používají distribuci *Red Hat Enterprise Linux Server release 5.5 (Tikanga)*. Všechny programy na Linuxu byly přeloženy kompilátorem GCC verze 4.1.2 pro 64b systém s přepínači `-O3 -DNDEBUG`. V systému Windows byl použit překladač, který je součástí *Microsoft Visual Studio 2008 Professional* v Release konfiguraci rovněž pro 64b systém.

Kapitola 5

Plánovač

Je zřejmé, že plánování krabiček je faktor, který ovlivňuje výkon celého systému. Na první pohled by se ale mohlo zdát, že tomu tak není. Platí totiž, že ať je plánování jakékoliv, každá krabička musí vzhledem k determinističnosti systému vykonat vždy stejné množství práce (musí zpracovat všechna příchozí data), takže celkové množství práce je vždy stejné přes všechny plánovací algoritmy. To by pak znamenalo, že nejlepší plánovací algoritmus je takový, který pouze zajišťuje, aby mělo každé vlákno dostatek práce.

Takto jednoduchá situace však není. Je nutné vzít v úvahu vliv mnoha dalších faktorů, které sice nesnižují množství potřebné práce, ale zvyšují rychlost jejího vykonání. Některé tyto faktory byly stručně zmíněny již v úvodu práce. Rovněž je nutné zajistit, aby byly požadavky vyřizovány pokud možno spravedlivě.

5.1 Faktory ovlivňující efektivitu plánovače

5.1.1 Vliv vyrovnávacích pamětí

Téměř každý moderní procesor má tzv. *cache* neboli vyrovnávací paměť. Cache je velmi rychlá paměť umístěná blízko procesoru. Při výpočtu si procesor ukládá kopie požadovaných dat právě do této paměti, takže práce s nimi je mnohem rychlejší než kdyby přistupoval k datům přímo do operační paměti, která bývá až řádově pomalejší.

Protože cena paměti je přímo úměrná její rychlosti a velikosti, bývá vyrovnávací paměť obvykle několikaúrovňová. To znamená, že je v systému několik vyrovnávacích pamětí najednou, od nejmenších a nejrychlejších až po pomalejší, ale s vyšší kapacitou. Tím je možné dosáhnout rozumného kompromisu mezi náklady na výrobu a výkonem systému.

Když pak procesor s několikaúrovňovou cache potřebuje přistupovat k datům z hlavní paměti, zkouší postupně hledat jejich kopie v nejrychlejší vyrovnávací paměti. Pokud v ní k dispozici jsou, pokračuje procesor v práci – nastává tzv. *cache hit*. V opačném případě se zkouší hledat v pomalejší paměti (tzv. *cache miss*) atd., až jsou v nejhorším případě zkopírována z hlavní paměti.

Protože vyrovnávací paměť má pouze omezenou kapacitu, je nutné řešit i situaci, kdy je cache zaplněná a procesor chce pracovat s dalšími daty. V tento okamžik se změněná data začnu zapisovat zpátky do hlavní paměti a uvolněná data jsou nahrazena kopiemi požadovaných dat. Existuje mnoho různých algoritmů, které určují, která data budou z cache uvolněna, ale jejich popis by byl nad rámec této práce.

V moderních procesorech jsou typicky dvě úrovně vyrovnávacích pamětí. Tzv. *L1 cache* je nejrychlejší a její velikost je řádově desítky kilobytů. Tato paměť bývá většinou rozdělena na dvě části. Jedna uchovává data a druhá instrukce[1]. *L2 cache* je o něco pomalejší, ale její velikost je řádově větší než velikost L1 cache a její kapacita se pohybujeme ve stovkách kB až jednotkách MB. Některé systémy mají navíc ještě *L3 cache*, jejíž kapacita může být řádově až desítky MB. Úkolem této paměti je dále vyrovnávat rozdílnou rychlost L2 cache a hlavní paměti.

Další důležitou charakteristikou je, zda je vyrovnávací paměť sdílena mezi více výpočetními jednotkami systému. Zatímco L1 cache sdílená nebývá, L2 resp. L3 sdílené bývají poměrně často. Sdílená vyrovnávací paměť přináší některé další výhody. Např. při přesunu výpočtu z jednoho logického procesoru na jiný nemusí cílový procesor znovu kopírovat data do své cache. Navíc pokud některý ze sdílejících procesorů není vytížen, mají ostatní k dispozici navíc jeho část vyrovnávací paměti.

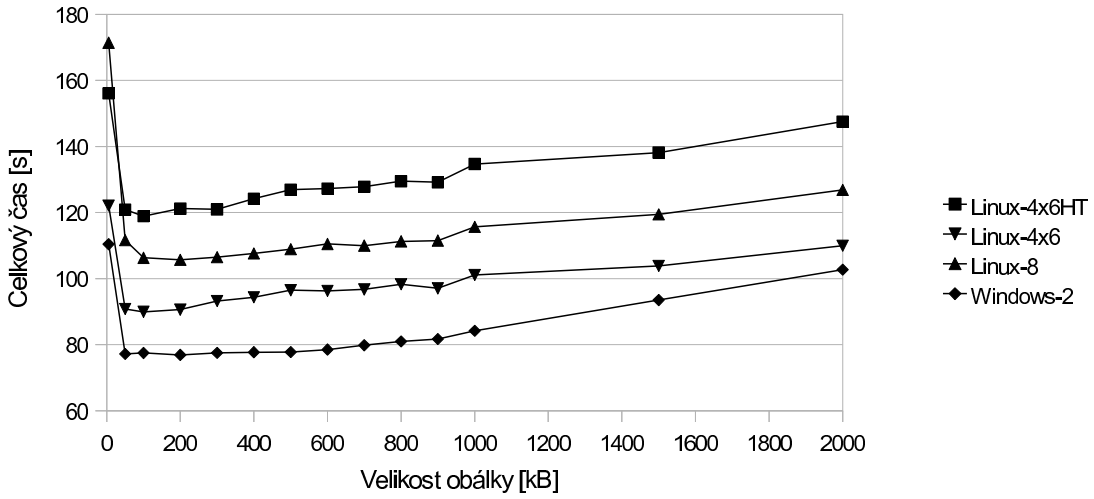
Jak bylo uvedeno výše, procesor nikdy nepřistupuje přímo do hlavní paměti. Vždy přistupuje do L1 cache, pokud v ní data nejsou, jsou zkopírována z L2 cache a pokud se ani v této paměti data nenachází, jsou kopírována z dalších úrovní vyrovnávacích pamětí případně až z hlavní paměti. Tento mechanismus by za normálních okolností příliš dobře nefungoval, neboť všechna požadovaná data stejně nakonec musí být načtena z hlavní paměti a putování dat skrz hierarchii vyrovnávacích pamětí navíc může dále zdržovat.

Naštěstí platí poměrně důležité pravidlo pro téměř všechny programy. Toto pravidlo říká, že pokud program pracuje s pamětí, pak přístupy do této paměti nejsou zcela náhodné. Naopak platí, že pokud nastal přístup do nějakého místa v paměti, je velmi pravděpodobné, že budou ta samá data nebo alespoň data z blízkého okolí toho místa, požadována brzy znovu. Toto pravidlo je známé pod názvem *princip lokality*.

Tento princip je důsledkem toho, že globální data programu bývají uložena na společném místě v paměti. Rovněž i lokální data, se kterými se pracuje, bývají pohromadě na vrcholu zásobníku programu. Navíc se k těmto datům, mezi něž patří i proměnné programu, zpravidla nepřistupuje pouze jednou, ale intenzivně se s nimi pracuje. Takže se jejich hodnoty načtou jednou a dále se s nimi pracuje v cache. Navíc se při vytváření kopie dat nekopírují jenom data požadovaná, ale vždy se načte určitá minimální velikost tzv. *cache line*, která má velikost 16 až 256B, takže při jednom přístupu se rovnou zkopíruje i blízké okolí požadovaných dat.

Vyrovnávací paměti urychlují běh programu také proto, že v případech, kdy princip lokality neplatí, což se stává např. při zpracování většího množství dat, přistupuje program k těmto datům zpravidla sekvenčně. Protože se s daty načítá i jejich blízké okolí, jsou rovnou k dispozici i data pro další kroky algoritmu.

[1] U některých existuje tzv. *trace cache*, do které se ukládají již dekódované instrukce.



Obrázek 5.1: Vliv velikosti obálky na výkon systému – jeden požadavek

Je tedy zřejmé, že vyrovnávací paměti mohou výkon programu značně urychlit. Je ale nutné při jeho návrhu dodržovat určitá pravidla, aby se tento potenciál maximálně využil. Jedná se zejména o tato pravidla:

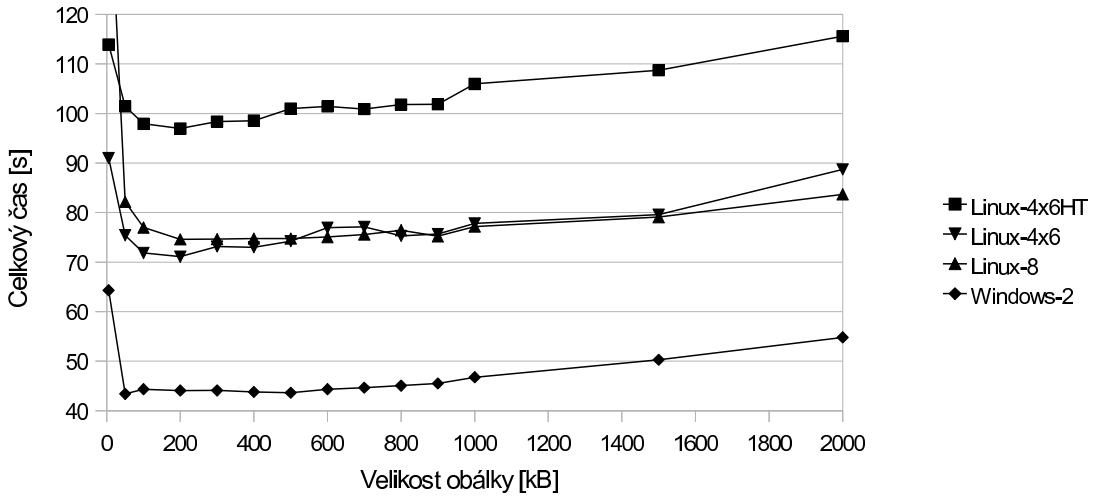
- Snažit se data co nejdéle udržet data ve vyrovnávací paměti během této doby s nimi co nejintenzivněji pracovat.
- Velká data zpracovávat sekvenčně.

Nelze jednoduše zaručit splnění druhého pravidla, neboť záleží skutečně jenom na krabičkách, jak s daty v obálkách pracují. Druhé pravidlo ale může plánovač zkusit zajistit alespoň maximalizací pravděpodobnosti toho, že když je krabička spuštěná, tak obálka, se kterou bude pracovat, je uložená v cache procesoru.

Velikost obálek

Velikost obálky má zřejmě velký vliv na to, zda práce s příchozí obálkou bude probíhat v cache, nebo nikoliv. Vztah mezi velikostí obálky a rychlostí systému je znázorněn v grafech 5.1 a 5.2. Probíhaly dva druhy měření. Při jednom byl spuštěn pouze jeden požadavek v jednom čase a těchto požadavků bylo spuštěno celkem 16 po sobě. Při druhém měření bylo udržováno vždy tolik paralelních požadavků, kolik vláken běželo. Takto bylo zpracováno celkem osmkrát tolik požadavků, kolik bylo vláken. V případě NUMA systémů byl výchozí plánovač (viz část 5.3) před měřením upraven tak, aby vlákna běžela pouze v rámci jednoho uzlu a nemohlo tak docházet k přesouvání úloh mezi uzly. Každé měření bylo provedeno několikrát, vždy s jinou výchozí velikostí obálky. Nejmenší měřená velikost obálky je 5kB.

Interpretace grafů je poměrně náročná, neboť jsou zatíženy jednak chybami měření (i přesto, že každé bylo provedeno desetkrát) a mnoha jinými vlivy, které s vyrovnávacími paměťmi příliš nesouvisí. Jedná se zejména o alokátor paměti, který pro



Obrázek 5.2: Vliv velikosti obálky na výkon systému – paralení požadavky

různé velikosti objektů může volit různé alokační strategie, takže různě velké bloky paměti mohou být přidělovány různě rychle.

Počítač *Windows-2* se chová téměř ukázkově. Je vidět, že při nižší velikosti obálek je systém mírně zdržován režií, která souvisí s odesláním obálky, naplánováním a spuštěním přijímací krabičky. Čím jsou totiž obálky menší, tím více jich pro stejná data musí být a tím více úloh je nutné vykonat. Pokud je velikost menší než určitá hranice, začne tato režie výrazně převládat a chod systému se značně zpomalí. Na druhou stranu při zvětšování velikosti obálek začne hrát roli omezená velikost vyrovnávací paměti, takže se systém začne zpomalovat a tento trend dále zůstává. Počítač *Linux-8* se chová velmi podobně, i když graf není tak rovnoměrný.

Ačkoliv by se mohlo zdát, že obálka má optimální velikost tehdy, kdy se akorát vejde do vyrovnávací paměti, není tomu tak, a hlavně v případě grafu s jedním požadavkem výkon s klesající velikostí obálky dále roste. Nakonec ale opět převládne režie spojená s odesíláním obálek a systém se značně zpomalí. Otázkou je, proč by zmenšování velikosti obálek mělo mít pozitivní vliv na výkon systému.

Platí, že data v rámci jednoho požadavku jsou odesílána sekvenčně, takže jsou vlákny zpracovávána data, která jsou k sobě poměrně „blízko“. Z toho plynou dva důsledky. Za prvé takovéto uspořádání způsobuje, že v okamžiku, kdy dorazí data do krabičky, je pravděpodobné (při malé velikosti obálky), že budou v cache k dispozici stále data z jejího dřívějšího spuštění. Pokud se jedná např. o třídící krabičku, která nově příchozí data zatřiďuje mezi předchozí, projeví se přítomnost předchozích obálek ve vyrovnávací paměti pozitivně na rychlosti tohoto zatřídění.

Za druhé platí, že když na požadavku pracuje několik vláken najednou, tak v okamžiku, kdy je obálka zatřiďována mezi ostatní, musí být do vyrovnávací paměti načteny obálky, které do krabičky dorazily z jiných vláken. Tímto způsobem dochází k určité synchronizaci obsahu vyrovnávacích pamětí, neboť obálky, které dorazily z jiných vláken jsou pravděpodobně rovněž uložené v jejich cache. Tato synchronizace má pozitivní vliv na výkon tehdy, když je velikost obálky dostatečně malá.

Pak totiž platí, že práce s jinými obálkami s menší pravděpodobností vytlačí obálky z poslední takové synchronizace. Když poté obálka dorazí opět do třídící krabičky, je k dispozici větší množství potřebných dat ve vyrovnávací paměti, a zatřídění obálky je tak rychlejší.

V případě paralelních požadavků popsaný jev není tak výrazný, neboť vlákna zpracovávají různé požadavky a ty neobsahují společná data. Data požadavků se tak navzájem vytlačují z vyrovnávací paměti a ke zmíněné synchronizaci obsahu cache tolik nedochází.

Podobné chování jako v případě počítače *Linux-8* lze pozorovat i u zbývajících počítačů. Při velikosti zhruba 900kB dochází u všech počítačů s operačním systémem Linux k mírnému poklesu. Vzhledem k tomu, že tato hodnota je stejná pro počítače s velmi rozdílnou hardwarovou konfigurací, lze tento pokles připsat právě některému ze softwarových vlivů.

Vliv sdílení vyrovnávací paměti

Již bylo zmíněno, že skutečnost, zda je vyrovnávací paměť sdílená nebo ne, může mít vliv tehdy, kdy některé jádro je méně vytížené a ostatní tak mohou používat jeho část vyrovnávací paměti, a tehdy, kdy se výpočet přesune z jednoho jádra na jiné.

První případ nelze příliš ovlivnit ani nijak využít, neboť cílem systému je, aby pokud možno vytěžoval rovnoměrně celý systém. Zrychlení přesunu výpočtu mezi jádry ale jistý vliv mít může. Tento jev ale lze simulovat pouze na počítači *Linux-8*, neboť se jedná o SMP systém, kde některá jádra cache sdílejí a některá ne. U ostatních počítačů sdílejí vyrovnávací paměť celé uzly, takže u počítače *Windows-2* by měření nemělo smysl a u *Linux-4x6* a *Linux-4x6HT* by výsledky byly silně zatížené faktorem NUMA.

Pro změření grafu 5.3 byl výchozí plánovač mírně upraven tak, aby každá úloha musela být ukradena od jiného vlákna. V první případě se prioritně vykrádalo vlákno, se kterým vykrádající cache nesdílí. Ve druhém případě se prioritně vykrádalo vlákno, se kterým cache sdílí. Ačkoliv je změřený rozdíl na hranici přesnosti měření, i mnohanásobné přeměření výsledku potvrdilo, že první případ je mírně pomalejší.

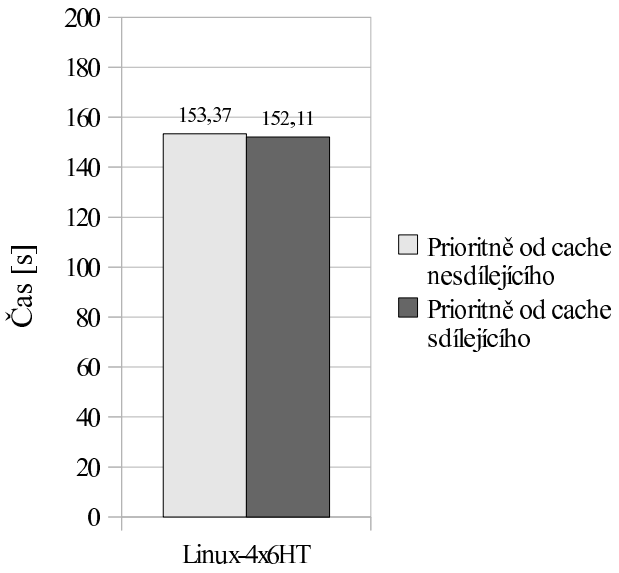
5.1.2 Systémy NUMA

Většina počítačů, které mají více procesorů, ja tvořena tzv. *SMP*[2] systémy. Takové systémy obsahují několik stejných procesorů, jejichž postavení v rámci systému je identické. To znamená, že všechny mají rovný přístup ke systémovým prostředkům, jako je např. operační paměť.

Bohužel se zvyšujícím se počtem procesorů se ze sdílených prostředků stává úzké hrdlo celého systému, neboť např. zmíněná paměť musí poskytovat data všem procesorům, což přirozeně snižuje rychlost přístupu k ní.

Systémy NUMA tento problém odstraňují tak, že seskupují procesory to tzv. *uzlů*, přičemž každý uzel má svojí vlastní paměť, se kterou přímo komunikuje. Pokud chce procesor z jednoho uzlu přistupovat na paměť jiného (což se běžně může stát, neboť

[2]Symmetric multiprocessing



Obrázek 5.3: Vliv sdílených cache na výkon systému

všechny procesory stále sdílí adresový prostor), musí k ní přistupovat přes paměťový řadič příslušného uzlu. To se samozřejmě projeví tím, že přístup do takové paměti je pomalejší.

Kromě zpomalení přístupu k paměti, ale nastává další problém. Někdo, komu paměť nenáleží, totiž může změnit její obsah. Pokud žádný z ostatních procesorů nemá uloženou kopii změněné části této paměti ve své cache, nic se neděje. V opačném případě je nutné tuto situaci nějak řešit. Existují dva různé přístupy:

- Tuto situaci nijak neřešit – ačkoliv je toto řešení velmi jednoduché z výrobního hlediska, z programátorského pohledu je komplikované pro takové systémy vyvíjet jakékoliv aplikace. Z tohoto důvodu se tyto systémy téměř nevyrábí.

- Zajistit, že změna bude propagována do všech kopií těchto dat ve vyrovnávacích pamětích procesorů. Tento model se z programátorského hlediska chová stejně jako SMP. Takové systémy se nazývají *cc-NUMA*[3] a ačkoliv jsou náročnější na výrobu (je nutné řešit vzájemnou synchronizaci vyrovnávacích pamětí), vyrábí se téměř výhradně právě tyto systémy.

Tato práce se zabývá pouze systémy cc-NUMA a následující část je věnována podrobnějším informacím o těchto systémech.

Paměť NUMA systémů je tedy disjunktně rozdělená mezi jeho uzly. To je realizováno tak, že fyzický adresový prostor je rozdělen na úseky a každý úsek odpovídá určitému uzlu. Toto rozdělení fyzického prostoru je důležité pouze pro operační systém. Programátor aplikačního programu se toto rozdělení příliš jednoduše nedozví. Navíc takovou znalost stejně nemůže nijak využít, neboť program pracuje pouze

[3] cache-coherent NUMA

s virtuálními adresami. Je ale možné pomocí určitých systémových volání ovlivnit mapování virtuálních adres na fyzické. Lze totiž požádat o namapování virtuálních stránek na fyzické vlastněné zadaným uzlem.

Implicitně se systém chová tak, že při přístupu na alokovanou, ale nenamapovanou stránku, je tato namapována na fyzickou stránku toho uzlu, který na ní přistoupil. To zaručuje poměrně rozumné chování, neboť tento uzel pak s pamětí pracuje rychleji. Existují ale aplikace (v případě Linuxu např. `numactl`), které toto chování mohou ovlivnit. Lze např. požádat o to, aby veškerá paměť byla přidělována pouze z určitých uzlů, nebo aby byla přidělována metodou round-robin z množiny zadaných uzlů. Rovněž lze měnit chování systémového plánovače tak, aby vlákna pouštěl pouze na zvolených uzlech. Toto chování lze ovlivňovat také přímo z programu pomocí knihovny `libnuma` v případě systému Linux nebo přímo systémovými funkcemi v případě systému Windows.

S NUMA systémy dále souvisí pojem tzv. *vzdálenosti uzlů*. Tato vzdálenost vyjadřuje, jak rychlý je přístup k paměti nějakého uzlu z jiného uzlu. Operační systém Linux poskytuje funkce, které pro libovolné dva uzly tuto vzdálenost umí vrátit. Pro vrácené hodnoty platí, že vzdálenost k sobě samému (tj. k vlastní paměti) je rovná číslu 10. Ostatní vzdálenosti jsou vztaženy k tomuto číslu, takže pokud je např. práce s pamětí jiného uzlu dvakrát pomalejší, bude tato vzdálenost rovná 20. V tabulce 5.1 je vidět matice všech vzdáleností tak, jak ji vrací operační systém pro počítače *Linux-4x6* a *Linux-4x6HT*.

	0	1	2	3
0	10	20	20	20
1	20	10	20	20
2	20	20	10	20
3	20	20	20	10

Tabulka 5.1: Matice vzdáleností počítače *Linux-4x6* a *Linux-4x6HT* – podle operačního systému

Systém Windows žádnou funkci pro zjišťování vzdáleností mezi uzly neposkytuje. Nejen z tohoto důvodu byla v rámci této práce implementována funkce, která tyto vzdálenosti měří sama. Funkce postupuje jednoduše tak, že v každém uzlu provede operaci s pamětí, která byla postupně naalokována na různých uzlech. Tuto operaci tvoří obrácení pořadí čísel ve velmi velkém poli, aby měření bylo co nejméně ovlivněno vyrovnávacími paměťmi.

Výsledek pro počítač *Linux-4x6*, pro nějž byla uvedena tabulka 5.1, je v tabulce 5.2, kde hodnotě 10 odpovídá nejmenší naměřená vzdálenost a ostatní čísla jsou vůči ní relativně přepočítaná. Už na první pohled je vidět, že tato matice vypadá trochu jinak, než ta, kterou poskytuje systém. Dokonce nejsou ani všechna čísla na diagonále stejná. Výsledky spíše napovídají, že uzel 0 a uzel 3 jsou osazeny pomalejšími paměťmi než uzly 1 a 2. Těchto přesnějších výsledků lze samozřejmě využít pro zlepšení plánovacího algoritmu.

Z tabulky je zřejmé, že vlastnosti NUMA systémů mohou efektivitu programů značně ovlivnit. Protože tabulka zachycuje rychlostní rozdíly v algoritmu, který byl

	0	1	2	3
0	13	16	16	26
1	19	10	16	23
2	23	16	10	19
3	26	16	16	13

Tabulka 5.2: Matice vzdáleností počítače Linux-4x6 – reálně změřená čísla

	0	1	2	3
0	10	14	15	14
1	15	10	14	14
2	14	15	10	15
3	15	15	15	10

Tabulka 5.3: Matice vzdáleností počítače Linux-4x6HT – reálně změřená čísla

navržen schválně tak, aby tyto rozdíly byly co nejvyšší, byl experimentálně změřen vliv rozdílné rychlosti pamětí i na systém Bobox. Výsledky jsou uvedeny v grafu 5.4, který byl získán trojnásobným spuštěním identických požadavků. Poprvé veškeré výpočty probíhaly pouze na uzlu 0 a alokovaly paměť pouze z tohoto uzlu. Po druhé byly výpočty prováděny na uzlu 1 a paměť byla alokována rovněž z uzlu 1. Třetí test pak zachycuje dobu výpočtu, když výpočty běžely na uzlu 0, ale paměť byla alokována na uzlu 3.

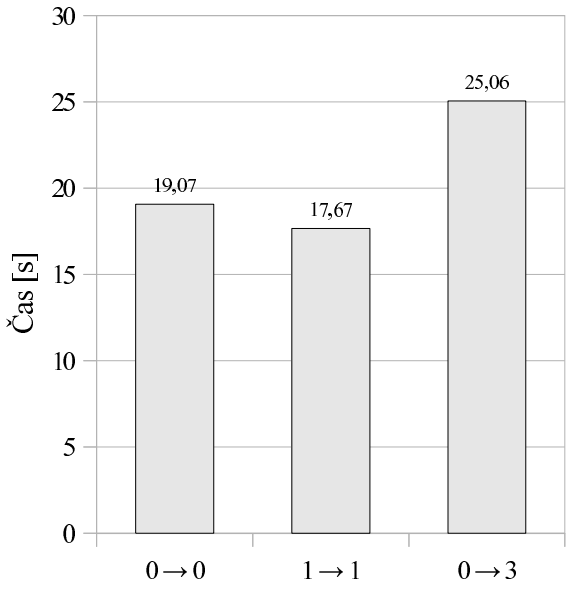
Z grafu vyplývá, že praktické výsledky potvrzují experimentální měření. Ačkoliv vliv rozdílných rychlostí paměti je redukovaný tím, že jednak nedochází k tak intenzivní práci s pamětí, jako v případě syntetického testu, a tím, že cache pomalost hlavní paměti mírně vyrovnává.

Další věc, která byla na testovacím systému pozorována, je, že práce procesoru s cizí pamětí zpomaluje nejen tento procesor, ale i vlastnící uzel. To je způsobeno tím, že paměťový řadič, který je zodpovědný za tuto paměť, musí vyřizovat další požadavky navíc.

Matice vzdáleností, která byla stejným algoritmem změřena pro počítač *Linux-4x6HT* je uvedena v tabulce 5.3. Ačkoliv jsou hodnoty zřejmě mírně zatížené nepřesností měření, je vidět, že u tohoto počítače mají všechny uzly stejně rychlou paměť a že vliv NUMA není tak silný jako v případě *Linux-4x6*.

5.1.3 Vliv Hyper-Threadingu

Během jednoho výpočtu zpravidla nebývají vytížené veškeré výpočetní jednotky procesoru. Například při práci s celými čísly nejsou využívány jednotky pro práci s čísly reálnými. Navíc moderní procesory nemívají od každého druhu pouze jednu jednotku, ale vyskytuje se v nich hned několik identických jednotek (např. pro zmíněnou práci s celými čísly) najednou. Tyto jednotky navíc využívá procesor pro paralelní zpracování instrukcí. Ačkoliv platí, že se procesor snaží tyto jednotky vytížit co možná nejvíce, ne vždy se mu to podaří. Procesor během výpočtu např. musí čekat na data z operační paměti, čekat až se vyhodnotí podmínka, která určí směr dalšího výpočtu



Obrázek 5.4: Vliv systému NUMA na výkon Boboxu

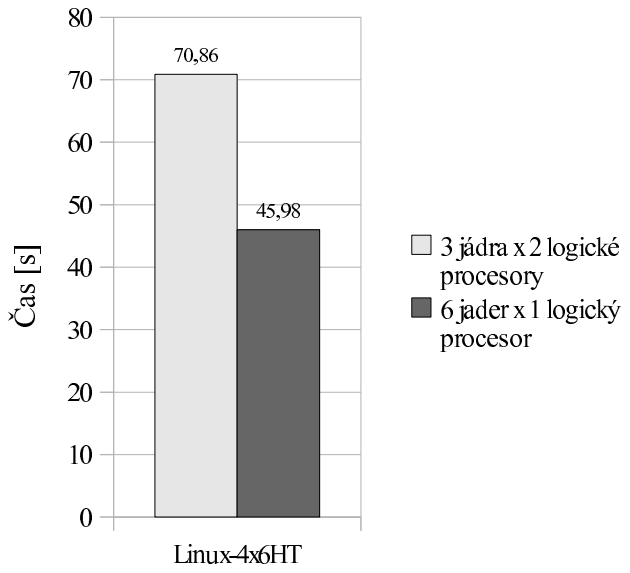
atd. Ve výsledku to znamená, že jenom zřídka jsou veškeré jednotky vytíženy.

Nabízí se tedy otázka, zda by nebylo možné tyto nevyužité jednotky nějak využít. Jedno z řešení je právě Hyper-Threading, který k nim jednoduše přidává další rozhraní. Procesor s podporou Hyper-Threadingu tedy vypadá jako obyčejný procesor se dvěma logickými procesory. Tyto procesory ale sdílí společné jednotky. Je tedy zřejmé, že v extrémní situaci je urychlení prakticky nulové, neboť pokud jeden logický procesor vytíží veškeré jednotky, na druhý se žádné další nedostanou. Toto však z výše zmíněných důvodů zpravidla nenastává, takže i druhé jádro má k dispozici výpočetní jednotky k tomu, aby mohlo provádět vlastní výpočty.

Z výše uvedeného popisu vyplývá, že výkon jednoho logického procesoru v jádře s podporou Hyper-Threadingu je silně ovlivněn vytížeností procesoru druhého a naopak. Platí, že výkon obou procesorů vytížených najednou je nižší než dvojnásobek výkonu jednoho z nich v případě, že druhý je nevytížený. Rovněž ale platí, že výkon obou dohromady bývá vyšší než výkon jednoho z nich, což je právě motivace k použití Hyper-Threadingu.

Pokud má systém N jader, přičemž každé disponuje technologií Hyper-Threading, je výpočetní výkon prvního logického procesoru prvního jádra a výkon prvního logického procesoru druhého jádra z výše uvedeného důvodu vyšší než výkon prvního a druhého logického procesoru prvního jádra. Právě toto pravidlo je návodem k tomu, jak by se měl plánovač ideálně chovat. Měl by se tedy nejdříve snažit vytížit jádra, jejichž oba logické procesory neprovádějí žádné výpočty, a teprve poté se snažit vytěžovat logické procesory, které sdílí výpočetní jednotky s logickými procesory, které jsou již vytížené. Pro nízký počet paralelních požadavků je tedy možné dosáhnout vyššího výpočetního výkonu pro jejich zpracování.

Graf 5.5 dokládá, že taková úprava má skutečný vliv na výkon systému. Naměřené hodnoty byly získány jednoduše tak, že byl výchozí plánovač puštěn na 6 logických



Obrázek 5.5: Vliv Hyper-Threadingu na výkon systému

procesorech jednoho uzlu počítače *Linux-4x6HT*. V prvním případě se jednalo o 3 jádra, kdy byly použity oba jeho logické procesory, ve druhém případě se jednalo o 6 jader, přičemž z každého byl použit pouze jediný logický procesor. Bylo puštěno vždy 6 paralelních požadavků najednou, aby byla všechna výpočetní vlákna maximálně vytížena.

Je zjevné, že Hyper-Threading je faktor, který se značně podílí na výkonnosti systému, a plánovač by s ním měl počítat. Na ostatních systémech toto měření nebylo provedeno, neboť technologií Hyper-Threading nedisponují.

5.2 Rozhraní plánovače

Jak bylo zmíněno v kapitole 3, musí všechny úlohy v Boboxu být oddděděné od třídy `task`. Mezi dvě nejdůležitější metody, které tato třída poskytuje, patří metoda `void schedule(bool immediate)`, která způsobí naplánování úlohy, a dále metoda `void run()`, kterou volá plánovač ve chvíli, kdy se rozhodne úlohu spustit.

Každá úloha se může nacházet v jednom z těchto stavů:

- `TS_SCHEDULED`, což znamená, že úloha je naplánovaná, ale ještě nebyla spuštěná
- `TS_NOTHING`, kdy úloha není ani naplánovaná ani spuštěná
- `TS_RUNNING`, kdy je úloha spuštěná
- `TS_RUNNING_SCHEDULED`, což je stav, kdy úloha běží a během toho nastal požadavek na její naplánování

Při požadavku na naplánování se postupuje tak, že úloha ve stavu `TS_NOTHING` se přidá do seznamu naplánovaných úloh a zároveň se přepne do stavu `TS_SCHEDULED`.

Pokud je ve stavu `TS_RUNNING`, přepne se do stavu `TS_RUNNING_SCHEDULED` a uloží se hodnota parametru `immediate`. Ve stavech `TS_SCHEDULED` a `TS_RUNNING_SCHEDULED` se při požadavku o naplánování nic neděje, neboť je úloha již naplánovaná.

Tyto stavy hrají svoji roli i ve chvíli, kdy krabička již splnila svůj úkol. Pokud se v tento okamžik nachází ve stavu `TS_RUNNING_SCHEDULED`, znamená to, že během výpočtu nastal požadavek na naplánování a je tedy nutné krabičku znovu naplánovat a přepnout do stavu `TS_SCHEDULED`. Při tomto znovunaplánování se použije uložená hodnota parametru `immediate`. Znamená to tedy, že pokud byla hodnota parametru `true`, je krabička naplánovaná na to vlákno, které právě ukončilo úlohu, a nikoliv na to, které o naplánování požádalo.

Těmito pravidly je dosaženo splnění všech podmínek zmíněných v části 3.2.

Přechod mezi stavy si instance třídy řídí sama, přičemž tyto přechody jsou implementovány jako atomické operace. Pokud úloha má být naplánována, zavolá metodu `void schedule(task *the_task, bool immediate)` na příslušné instanci plánovače. Těchto instancí je tolik, kolik vláken v systému běží paralelně a úloha zavolá metodu na té instanci, která odpovídá vláknu, ze kterého byl požadavek na naplánování vytvořen.

Z výše uvedeného tedy vyplývá, že jako plánovač může fungovat jakákoliv třída, která implementuje zmíněnou metodu `schedule` a která se postará o to, že v konečném čase bude zavolána metoda `run` na instanci naplánované úlohy. Ve skutečnosti je rozhraní mírně složitější, ale pro pochopení dalšího textu jsou tyto informace dostatečné a detailnější popis by byl nad rámec této práce.

Samotný požadavek, který má Bobox vyhodnotit, je spuštěn jednoduše vybráním některé z instancí plánovače a na něj se naplánuje iniciační krabička tohoto požadavku.

5.3 Výchozí plánovač

Systém Bobox měl již v době zadání vlastní implementaci plánovače úloh a v této části je stručně rozebrána jeho funkčnost.

Každý plánovač si udržuje dva seznamy úloh, které má vykonat. První seznam lze označit jako *soukromý* a druhý jako *veřejný*. Při vytvoření další úlohy (požadavku o naplánování další úlohy) je tato úloha vložena na začátek některého ze seznamů. Přesný výběr seznamu závisí na hodnotě parametru `immediate` metody `schedule`. Pokud je tento parametr roven `true`, je vložen na začátek soukromého seznamu, v opačném případě na konec veřejného seznamu. V okamžiku, kdy plánovač dokončí úlohu, podívá se na začátek svého soukromého seznamu, případnou úlohu odebere a spustí.

Pokud je tento seznam prázdný, zkouší si nějaké úlohy ukrást s tím, že nejdříve zkouší okrást sebe a pak teprve ostatní. Kradení probíhá tak, že zkouší metodou round-robin najít první plánovač, jehož veřejný seznam není prázdný. Pokud takový plánovač najde, přesune celý obsah jeho veřejného seznamu do svého soukromého a dále pokračuje stejným způsobem. Z výše uvedeného plyne hlavní rozdíl mezi oběma seznamy – ze soukromého seznamu nelze ukrást žádnou úlohu.

Algoritmus je navržen tak, aby zohledňoval vliv vyrovnávacích pamětí, neboť pokud krabička pošle obálku jiné, je cílová krabička (krabička je zároveň úlohou) vložena na začátek soukromého seznamu, takže cizí plánovač tuto úlohu neukradne a zároveň bude tato úloha spuštěna ihned po dokončení aktuální úlohy, takže příchozí obálka pravděpodobně bude stále ve vyrovnávací paměti.

V novější verzi tohoto plánovače je implementován mechanismus uspávání vláken. Ten funguje tak, že existuje jeden semafor, který má v každém okamžiku hodnotu, odpovídají počtu plánovačů, které mají neprázdný veřejný seznam. Když má plánovač prázdný soukromý seznam, pokusí se snížit hodnotu tohoto semaforu. Pokud se to povede, najde plánovač takový veřejný seznam, ze kterého může úlohy ukrást (takový určitě existuje, neboť před snížením hodnoty semaforu musela být jeho hodnota nenulová). Pokud se to nepovede, je vlákno uspáno, dokud se některý z veřejný seznamů opět nestane neprázdným.

Tento plánovač má ale i několik nevýhod. První nevýhoda je zřejmá již z jeho popisu. Pokud se do soukromého seznamu dostane několik časově náročných úloh, nemůže žádné volné vlákno některou z těchto úloh vzít a zpracovat místo vlastnícího vlákna. Rovněž je nevhodný způsob, kterým jsou spouštěny nové požadavky – jejich iniciační krabičky jsou metodou round-robin předávány jednotlivým plánovačům, které si je zařazují do svého soukromého seznamu. Toto chování ale trpí stejným nedostatkem, který je zmíněn v části 5.5.1, tj. že velikost vyžadované paměti může neomezeně růst. Dále výchozí plánovač nezohledňuje některé z již zmíněných faktorů. Jedná se zejména o podporu NUMA systémů, Hyper-Threadingu a velikostí a uspořádání vyrovnávacích pamětí, neboť implicitně jsou obálky vytvářeny s konstantním množstvím řádků a tato konstanta nijak nezávisí na hostitelském systému ani na samotném formátu obálky.

5.4 Plánování datově vázaných úloh

V této části budou rozebrány různé situace, kdy dochází k naplánování datově vázaných úloh. Tj. těch, které vznikly příjmem obálky. Na základě rozboru dílčích situací bude v závěru navržena strategie pro spuštění těchto úloh. Ta bude určena pouze pro SMP systém. Algoritmus pro NUMA systémy pak ze zde navržené strategie bude vycházet.

Ostatní úlohy, tj. datově nevázané, není třeba řešit rozborem případů, neboť takové krabičky mohou být spuštěny kdykoliv bez ohledu na tvar grafu. Na druhou stranu ani spouštění těchto úloh nemůže být zcela náhodné, neboť ačkoliv toto pořadí příliš neovlivňuje výkon systému, ovlivňuje výrazně práci systému s pamětí a spravedlnost Boboxu vůči příchozím požadavkům. Strategie plánování datově nevázaných úloh je diskutována v části 5.5.

Vzhledem k tomu, že u SMP systémů není nutné brát v úvahu rozdílnou rychlost pamětí, bude prioritou plánovače jediná věc – udržet každou obálku ve vyrovnávací paměti co nejdéle a snažit se, aby se s obálkou pracovalo co nejintenzivněji právě tehdy, dokud je uložená v cache procesoru. Jediná možnost, jak tohoto chování může plánovač dosáhnout je ta, že se bude snažit, aby když je krabička spuštěná, měla

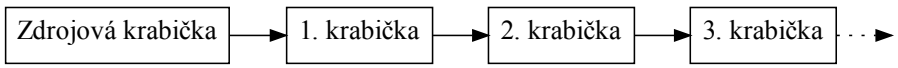
obálky, se kterými bude pracovat pokud možno ve vyrovnávací paměti.

V dalším textu budou krabičky rozděleny na čtyři typy podle toho, jaké je jejich vnitřní chování z hlediska plánovače. Tyto typy jsou:

- *Proudová krabička* – taková krabička, která přijatou obálku zpracuje a výsledek ihned odešle.
- *Blokující krabička* – tato krabička neposílá výsledek ihned, ale postupně si ukládá příchozí obálky a když jich má dostatečné množství, data zpracuje a výsledné obálky rozešle dál. Takové chování má např. třídící krabička, která začne posílat obálky až poté, co přijme všechna příchozí data.
- *Zdrojová krabička* – krabička, která posílá vstupní data, přičemž tato data odesílá, když je spuštěná. Po odeslání dat dojde k opětovnému naplánování a tento postup se provádí dokud má nějaká data k odeslání, resp. dokud neodešle otrávenou obálku.
- *Cílová krabička* – krabička, která pouze přijímá obálky, ale žádné neodesílá. Např. terminační krabička má přesně takové chování.

5.4.1 Lineární graf bez blokujících krabiček

Tato část se zabývá situací, kdy je graf izomorfní orientované cestě (viz obrázek 5.6) a každá krabička v tomto grafu, kromě první, která musí být logicky zdrojová, je proudová.



Obrázek 5.6: Lineární graf

Jednoprocesorový systém

Pokud by se jednalo o jednoprocesorový systém, byla by situace jednoduchá, neboť existují prakticky pouze dvě krajní strategie.

První přiděluje výpočetní čas popořadě jednotlivým krabičkám, takže nejdříve naplánuje první krabičku, aby zpracovala všechna data. Poté bude plánovat druhou krabičku, až ta zpracuje všechna data, tak třetí atd. Takový algoritmus je poměrně nefunkční, neboť naráží na omezenou velikost vstupních bufferů krabiček a navíc je poměrně nešetrný ke cache, neboť pokud už je obálka ve vyrovnávací paměti, tak místo aby se s ní pracovalo dále, je zahozena a začne se zpracovávat další obálka v posloupnosti.

Druhý způsob funguje tak, že se nepřiděluje čas po krabičkách, ale po datech. To znamená, že se přidělí čas první krabičce, až tato krabička odešle výstupní data, je spuštěna druhá krabička, aby zpracovala přijatá data, následně třetí atd. V okamžiku, kdy jsou data zapsaná do poslední krabičky, je opět spuštěna první krabička a tento postup se opakuje, dokud nejsou zpracovaná všechna data. Druhý způsob odstraňuje obě nevýhody prvního způsobu. Typicky se totiž data předávají pouze v cache procesoru a není nutné je zapisovat do hlavní paměti. Navíc nehrozí, že by se někdy zaplnila vstupní vyrovnávací paměť krabiček – vždy totiž jedna obálka přijde a jedna odejde.

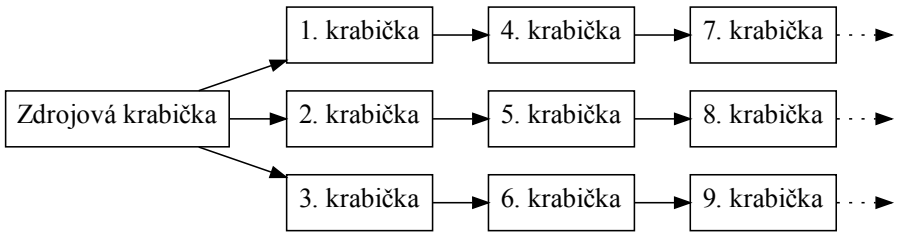
Víceprocesorový systém

Pokud má systém více procesorů, je situace trochu složitější. Samozřejmě, že ideální situace by byla taková, že by se paralelně aplikovala druhá z výše zmíněných strategií. První vlákno by spustilo první krabičku, jakmile by práce první krabičky skončila, pokračovalo by první vlákno druhou krabičkou, zatímco druhé vlákno by spustilo opět první krabičku. Pokud by výpočet jednoho vlákna skončil, vrátilo by se opět k první krabičce a začalo zpracovávat další obálku.

Bohužel takto by systém mohl fungovat pouze za předpokladu, že všechny krabičky pracují stejně dlouho. To ale zaručeno není. Problém tedy nastává ve chvíli, kdy některé krabičky pracují déle než jiné. Pak totiž vlákno nemůže jednoduše po ukončení zpracovávání jedné krabičky pokračovat další, neboť ta může být stále spuštěna předchozím vláknem.

Tento problém je v Boboxu řešen tím, že každá krabička má svoji vstupní vyrovnávací paměť obálek, do které je možné posílat obálky i tehdy, kdy je spuštěná. Na druhou stranu velikost takové paměti je omezená, takže to zmíněný problém zcela neřeší, ale pouze oddaluje. Navíc nelze nechat vlákno čekat na to, až předchozí vlákno doběhne a uvolní vyrovnávací paměť, neboť není jasné, jak dlouho ještě poběží. To by v případě např. třídící krabičky mohlo způsobit zamrznutí celého systému, ačkoliv by vlákna mezitím mohla vykonat spoustu jiné práce. Ať už v jiné části grafu nebo v jiném požadavku.

Zde tedy nezbývá nic jiného, než přijmout fakt, že vlákno v takovém případě sice pošle obálku krabičce, ale místo aby pokračovalo v jejím zpracováním, musí začít zpracovávat jinou a třeba i zcela nesouvisející úlohu. Na druhou stranu je dobré zajistit, aby poté, až zpracování krabičky skončí, bylo toto vlákno hlavním kandidátem pro další spuštění krabičky. To ale ve výchozí architektuře není možné, neboť historie požadavků o naplánování neexistuje a krabička je spuštěna stejným vláknem, které právě ukončilo výpočet této krabičky (viz část 5.2) navíc jako datově vázanou úlohu, což je nevhodné i z toho důvodu, že by vlákno raději mělo spustit tu krabičku, do které právě krabička poslala data. Právě z tohoto důvodu je toto výchozí chování změněno a krabička je znovu naplánována jako datově nevázaná. Dle měření tato úprava o 1-2% urychluje výpočet systému.



Obrázek 5.7: Příklad rozvětvení

5.4.2 Lineární graf s blokujícími krabičkami

Pokud je graf izomorfní orientované cestě, ale některé krabičky na ní jsou blokující, je situace mírně odlišná. Nelze totiž nechat kompletně zpracovat jednu obálku, poté druhou atd., neboť blokující krabička sice obálku přijme, ale dokud se sama nerozhodne, že nějakou obálku odešle, tak ji bude držet u sebe.

Lze ale jednoduše použít strategii, která vychází z toho, že blokující krabička se ze strany vstupu chová jako cílová (negeneruje žádné obálky), ale ve chvíli, kdy začne posílat obálky, začne se ze strany výstupu chovat jako zdrojová krabička. Algoritmus je tedy stejný jako pro několik lineárních grafů bez blokujících krabiček spojených vedle sebe.

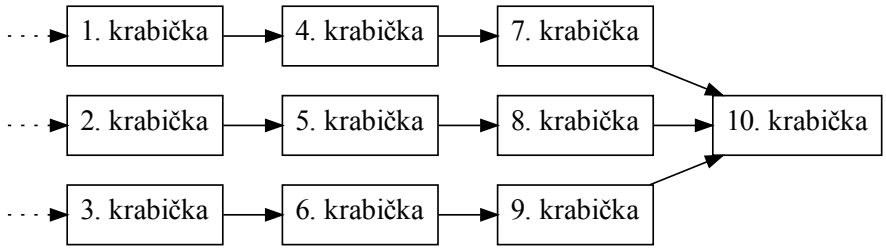
5.4.3 Rozvětvení cesty

Obtížnější situace nastává, pokud má jedna krabička více výstupních hran (příklad takového rozvětvení je na obrázku 5.7).

Z hlediska plánovače dojde k tomu, že přijdou požadavky na okamžité naplánování tří krabiček. Otázkou je, v jakém pořadí krabičky pouštět. Lze je buď pouštět v pořadí 1, 2, 3, 4, 5, 6, … (což vlastně odpovídá ukládání úloh do fronty), nebo v pořadí 1, 4, 7, …, 2, 5, 8, …, 3, 6, 9 … (což odpovídá použití zásobníku).

První přístup nemá téměř žádnou výhodu. Mohlo by se sice zdát, že krabičky 1, 2 a 3 budou mít vstupní obálku ve vyrovnávací paměti, ale ani to nemusí být pravda, neboť obálky vygenerované krabičkou 1 mohou původní obálku z této paměti zcela vytlačit.

Proto je lepší druhá strategie, kdy se plánovač zachová, jako kdyby se jednalo o několik lineárních grafů vedle sebe, každý se svojí zdrojovou krabičkou (i přesto, že se jedná o jednu a tu samou). Sice ve chvíli, kdy se spouští krabička č. 2 již pravděpodobně nebude mít vstupní obálku ve vyrovnávací paměti, ale to je vykoupeno tím, že obálky v jednotlivých větvích budou předávány ve vyrovnávací paměti.



Obrázek 5.8: Příklad spojení větví

5.4.4 Spojení několika větví

Zde je situace zcela opačná, neboť několik rozdílných cest se spojuje do jedné (viz obrázek 5.8. Zde může docházet k situaci, kdy přijde požadavek o naplánování krabičky, ta bude spuštěna, ale protože nebude mít data ze všech vstupů, žádnou práci nevykoná. S tímto spuštěním „naprázdno“ ale plánovač nemůže nic udělat, neboť nelze dopředu určit, zda krabička má nějakou práci, či ne. Dále není možné jednoduše zajistit, aby všechny příchozí obálky byly ve vyrovnávací paměti procesoru. Jednak se tolik obálek do ní vůbec nemusí vejít a navíc cesty do této krabičky mohou být velmi odlišné a může po nich přijít rozdílný počet obálek, takže by to bylo zcela nemožné.

Proto přímočaré řešení, které se ke krabičce bude chovat, jako by neměla více vstupů, je přijatelné, neboť tímto způsobem se plánovač pokusí zajistit, aby alespoň jedna z obálek ve vyrovnávací paměti byla.

5.4.5 Vyhodnocení strategií

Z výše uvedeného rozboru případů vyplývá, že strategie, která nově vzniklé datově vázané úlohy ukládá na zásobník a spouští je tak, že vždy odebere nejnovější úlohu ze zásobníku a spustí, splňuje všechny požadavky, které byly zjištěny. Neboť pro lineární graf bez blokujících krabiček a pro rozvětvení se chová tak, jak je požadováno a ostatní případy se z hlediska plánovače dle rozboru případů řeší stejně jako lineární graf.

Na druhou stranu se takto může snadno stát, že jedno vlákno bude mít hodně práce (v případě rozvětvení), zatímco jiná vlákna nebudou mít práci žádnou. Proto je nutné zahrnout i vykrádání úloh z tohoto zásobníku.

5.5 Plánování datově nevázaných úloh

Plánování datově nevázaných úloh, nemá příliš velký vliv na výkon systému. Výrazně ale ovlivňuje jeho jiné vlastnosti.

První důležitou vlastností je spravedlnost systému. Je totiž důležité, aby požadavky byly vyřizovány pokud možno spravedlivě a nemohlo se stát to, že některý požadavek bude upřednostněn na úkor jiného. Protože iniciační úlohy jsou pro plánovač datově nevázané, souvisí právě plánování těchto úloh velmi úzce se spravedlností systému vůči požadavkům. Dále strategie plánování iniciačních a datově nevázaných úloh ovlivňuje nároky Boboxu na velikost operační paměti.

5.5.1 Řešení pomocí fronty

Spravedlnosti je zřejmě možné dosáhnout tím, že datově nevázané úlohy budou vkládány do seznamu typu FIFO. První úloha, která se objeví, bude vykonána nejdříve. V širším kontextu to způsobuje jak spravedlivé spouštění jednotlivých požadavků (iniciační krabičky budou spuštěny v tom pořadí v jakém přicházely odpovídající požadavky do systému), tak spravedlivé vyřizování úloh, které vznikly během vyhodnocování požadavku. Experimentální měření potvrzuje, že požadavky jsou skutečně vyřizovány v pořadí, které odpovídá času zařazení do systému a jejich výpočetní složitosti. Na druhou stranu v případě, že je požadavků do systému zařazeno mnoho, jsou vyhodnocovány všechny najednou, což způsobuje, že se výrazně zvyšuje spotřeba paměti a tato spotřeba není ničím omezená (pouze množstvím požadavků, ale to je faktor, který plánovač nemůže ovlivnit). Pokud by se vyskytlo dostatečné množství požadavků najednou, může dokonce dojít až k úplnému vyčerpání prostředků a ke zhroucení systému.

5.5.2 Řešení pomocí zásobníku

Druhým řešením je použít seznam typu LIFO. Takový postup minimalizuje spotřebu paměti, neboť má tendenci intenzivně pracovat pouze na jednom požadavku a teprve, když jej dokončí, tak začít pracovat na jiném. Pokud by nastala situace zmíněná výše (velké množství požadavků najednou), pak by se sice všechny iniciační úlohy vložily na zásobník. Následně by se ale odebrala ta, která je na vrcholu, a spustila by se. Veškeré datově nevázané úlohy, které by během zpracování vznikly, by se rovněž vkládaly na zásobník a blokovaly by dále spouštění iniciačních úloh ostatních požadavků. Samozřejmě, že v paralelním prostředí pravděpodobně nebude vyhodnocován pouze jeden požadavek v jednom čase (každé vlákno může ze zásobníku odebrat jednu iniciační úlohu), ale počet paralelně zpracovávaných požadavků jistě nebude více než je počet vláken v systému. Zdůvodnění tohoto tvrzení je obdobné jako v části 5.5.3.

Z hlediska spotřeby paměti se tedy zdá, že zásobník pro datově nevázané úlohy vychází lépe. To ale nemusí být pravda ve chvíli, kdy požadavky do systému budou přicházet s časovými odstupy. Pak se může stát, že bude rozpracováno několik požadavků, když přijde požadavek nový. Z předchozího odstavce plyne, že tento požadavek začne být okamžitě vykonáván. Když bude částečně rozpracován i tento, přijde požadavek další atd. Takto budou budou v zásobníku stárnout požadavky, přičemž každé bude mít naalokované určité množství paměti. Tímto způsobem může rovněž

dojít k vyčerpání prostředků systému. Navíc je takový systém zjevně nespravedlivý, takže v tomto kontextu vychází lépe řešení s frontou.

5.5.3 Řešení pomocí dvou front

Další řešení, ze kterého již vychází implementace nového plánovače, je založeno na použití seznamu typu FIFO. Tyto seznamy jsou ale dva. Jedna pro iniciační úlohy požadavků a druhá pro datově nevázané úlohy vzniklé při vyhodnocování požadavků. Z první fronty se odebírá pouze tehdy, když neexistuje jiná úloha, kterou by vlákno mohlo vykonat. Rozdíl oproti řešení s jednou frontou spočívá v tom, že při tomto uspořádání je vždy najednou zpracováváno maximálně tolik požadavků, kolik je k dispozici vláken.

Zdůvodnění tohoto tvrzení je snadné. Platí totiž, že dokud není požadavek ukončen (tj. terminační krabička jej neukončila), vždy buď existuje alespoň jedno vlákno, které vyhodnocuje jeho úlohu nebo alespoň jedna jeho úloha je naplánovaná (vázaná nebo nevázaná). V opačném případě by neexistoval způsob, jak by mohl požadavek být dokončen. Pokud nastala první situace, pak je toto vlákno zaměstnané a nemá důvod požadovat další iniciační úlohu. Pokud je nějaká úloha pouze naplánovaná, pak bude spuštěna tato úloha místo toho, aby se vzala další úloha z fronty iniciačních úloh. To vyplývá z toho, že z fronty iniciačních úloh se odebírá až ve chvíli, kdy jiná úloha v systému není.

Platí tedy, že počet požadavků, které jsou paralelně vyhodnocovány, je omezen počtem vláken a vyčerpání prostředků zahlcením systému velkým počtem požadavků nehrozí. Zároveň je systém maximálně vytížen, neboť v okamžiku, kdy se vlákno uvolní a nemá jinou práci, začne zpracovávat další požadavek. Platí tedy, že N paralelních požadavků bude vyhodnoceno za stejnou dobu jako v případě prostého řešení s frontou, neboť se v obou případech vykoná stejné množství práce.

5.5.4 Řešení s frontou požadavků

Předchozí řešení trpí jedním nedostatkem. Z části 5.1.1 vyplývá, že je dobré, pokud vlákna pracují pokud možno nad jedním požadavkem. Dochází pak k tomu, že vyrovnávací paměti procesorů obsahují podobná data, což urychluje zpracování požadavku. Logicky ale nelze zajistit bez negativního vlivu na paralelizaci, aby vlákna zpracovávala pouze jeden požadavek. Na druhou stranu řešení se zásobníkem se alespoň snažilo tomuto chování přiblížit. Pokud totiž při zpracování požadavku vznikla datově nevázaná úloha, byla vložena na vrchol zásobníku, takže vlákno, které si tuto úlohu vyzvedlo, pracovalo nad stejným požadavkem. Navíc je vyšší pravděpodobnost, že příště vlákno bude pracovat s tím samým požadavkem – jiné vlákno umístilo jeho úlohu opět na vrchol. Na druhou stranu fronta takovou vlastnost nemá a každé vlákno (při úplném vytížení) pravděpodobně vyhodnocuje úlohu jiného požadavku.

Proto finální řešení používá uspořádání, kdy jsou úlohy rozděleny podle požadavků, kterým náleží. Každý požadavek tak obsahuje frontu, na jejíž konec jsou vkládány jeho datově nevázané úlohy a z jejíhož začátku jsou úlohy odebírány ke zpracování. V samotném systému jsou pak dvě fronty požadavků – jedna *vstupní*, do

které jsou vkládány požadavky ke zpracování, a druhá *pracovní* ve které jsou požadavky, které se právě vyhodnocují. Vytvoření nového požadavku pak obnáší pouze vytvořit jeho vlastní frontu datově nevázaných úloh, do této fronty vložit iniciační úlohu a požadavek vložit do vstupní fronty.

Když je požadována úloha pro zpracování, najde se nejstarší požadavek v pracovní frontě takový, který má neprázdnou frontu úloh. Ze začátku této fronty se pak úloha odebere. Pokud žádný takový požadavek v pracovní frontě není, je do ní přidán nejstarší požadavek ze vstupní fronty a je vrácena jeho iniciační úloha. Nová datově nevázaná úloha požadavku se vkládá jednoduše na konec jeho fronty. Pokud je požadavek ukončen, je jednoduše odstraněn z pracovní fronty.

Tento systém zachovává výhodu zásobníku - nejvíce se pracuje s nejstarším požadavkem v pracovní frontě. Ale zároveň zachovává spravedlnost, neboť požadavky jsou vyřizovány v tom pořadí, ve kterém vstupovaly do systému. Navíc stále platí, že délka pracovní fronty je shora omezená počtem výpočetních vláken.

Experimentálním měřením bylo zjištěno, že tato úprava zrychluje zpracování paralelních požadavků o 2-3%.

Kapitola 6

Implementace plánovače

6.1 Architektura plánovače

Architektura plánovače (viz obrázek 6.1) je třívrstvá. V nejvyšší úrovni je jediná instance třídy `scheduling_manager`, která přijímá požadavky k vyhodnocování a tyto požadavky distribuuje do nižší vrstvy.

Ve druhé vrstvě jsou instance třídy `node_manager`, přičemž pro každý uzel NUMA systému existuje právě jedna instance. Tato třída spravuje vstupní i pracovní frontu požadavků, neboť tyto fronty jsou sdílené všemi vlákny v rámci jednoho uzlu. Dále třída řídí uspávání a probouzení vláken, i když ve skutečnosti tuto práci pouze deleguje do třídy `task_manager`, která je popsána v části 6.2.

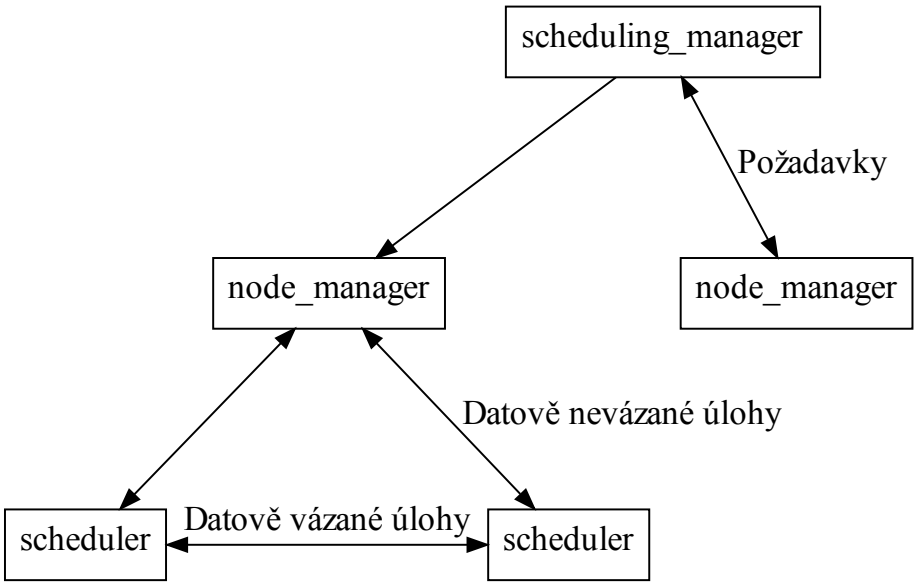
Ve třetí vrstvě jsou pak plánovače pro jednotlivá vlákna. Každý plánovač udržuje vlastní seznam datově vázaných úloh. Tento seznam se ze strany vlastnícího plánovače chová jako zásobník, zatímco pro ostatní plánovače se chová jako fronta. To zajišťuje, že vláknu nebudou kradeny úlohy, u nichž je nejpravděpodobnější, že jejich vstupní data jsou uložena ve vyrovnávací paměti, neboť takové úlohy se nacházejí ve vrcholu zásobníku, resp. na začátku fronty.

Toto uspořádání je zvoleno tak, aby odráželo fyzickou strukturu hostitelského počítače. Díky tomu je možné lépe implementovat podporu různých hardwarových faktorů.

Dále jsou detailněji rozebrány jednotlivé funkční části plánovače.

6.2 Uspávání vláken

Uspávání vláken v případě nízkého vytížení systému je velmi důležitou vlastností plánovače. Pokud by vlákna měla v nekonečné smyčce aktivně čekat na to, až bude k dispozici úloha, kterou by mohla vykonat, byl by hostitelský systém trvale, bez ohledu na zatížení Boboxu, zcela vytížen. To samozřejmě přináší nejen výkonnostní (obzvláště v případě Hyper-Threadingu čekání v aktivní smyčce značně zpomaluje druhý logický procesor v páru), ale spíše ekonomické problémy. Vytížený systém totiž spotřebuje více energie, má vyšší nároky na chlazení a je nepoužitelný k jiným účelům.



Obrázek 6.1: Architektura nového plánovače

Je tedy třeba zajistit, aby vlákna na úlohy nečekala aktivně, ale pasivně. Je několik různých přístupů, jak tuto funkčnost implementovat. Od poměrně jednoduchých až po více sofistikovaná. Důležitým faktem, který by měl brán v úvahu, je to, že uspání a znovuprobuzení vlákna je časově poměrně náročné. Při aktivním čekání vlákno úlohu začne vykonávat ihned, zatímco při pasivním musí být nejdříve probuzeno. Snahou by tak mělo být to, aby k uspávání vláken nedocházelo pokud možno zbytečně.

Uspávání vláken na určitou dobu

Zcela nejjednodušší způsob, jak snížit zátěž systému při nízkém vytížení Boboxu, je jednoduše pasivně uspat vlákno na zvolenou dobu. Operační systém se pak sám postará o probuzení vlákna, které se podívá, zda se neobjevila další úloha k vykonání. Pokud ano, tak ji vykoná, v opačném případě se znovu uspí. Ačkoliv je taková úprava velmi snadná na implementaci, trpí velkým problémem. Ten vyplývá z toho, že pokud se objeví nová úloha, je nutné počkat, až se některé z vláken probudí, aby ji mohl začít vykonávat. To zřejmě může snižovat rychlost odezvy systému. Řešením by bylo snížit časový interval, na který bude vlákno uspáno, ale taková úprava by byla na úkor vytíženosti operačního systému.

Další potíž spočívá v tom, že i v případě, že Bobox nemá žádný požadavek k vyhodnocení, je systém stále mírně vytěžován. To by sice šlo vyřešit jednoduše uspáváním vláken na stále delší a delší dobu, přičemž délka by se odvíjela od toho, kolikrát

se vlákno probudilo zbytečně. Ale o to déle by musel čekat nově příchozí požadavek na spuštění.

Tento způsob uspávání vláken nebyl ze zmíněných důvodů při implementaci použit, ačkoliv byl experimentálně testován a přes svojí jednoduchost vykazoval poměrně dobré chování. Dalším důvodem pro odmítnutí tohoto způsobu bylo, že takto nelze regulovat zátěž jednotlivých vláken.

Uspávání vláken pomocí podmínkových proměnných

Synchronizační primitivum *podmínková proměnná*[1] je určeno právě pro tento účel, neboť slouží pro pasivní čekání na určitou podmínku (v tomto případě na výskyt nové úlohy). Pokud by každé vlákno mělo vlastní podmínkovou proměnnou, bylo by možné vlákna snadno notifikovat o tom, že mají úlohu ke spuštění. Bylo by tedy možné snadno implementovat rovněž vyvažování zátěže vláken.

Tento způsob ale nebyl použit ze dvou důvodů. Použití podmínkové proměnné obnáší velké množství systémových volání jak na straně toho, kdo úlohu vytvořil (zamknutí mutexu, probuzení vlákna, odemknutí mutexu), tak na straně vlákna (zamknutí mutexu, kontrola/čekání na podmínku, odemknutí mutexu). Navíc operační systém Windows neposkytuje ve starších verzích [5] toto synchronizační primitivum a jeho korektní implementace pomocí jiných dostupných je náročná a ve výsledku neefektivní [14].

Uspávání vláken pomocí semaforu

Zcela přímočaré řešení spočívá v tom, že bude k dispozici jeden semafor, jehož hodnota bude vždy rovná počtu naplánovaných úloh. Při vytvoření úlohy se zvýší hodnota semaforu. Pokud vlákno žádá o další úlohu, pokusí se nejprve snížit hodnotu semaforu. To se buď se povede a vlákno nějakou z úloh vykoná, nebo nepovede. V tom případě je vlákno uspáno a probuzeno až ve chvíli, kdy se vytvoří další úloha. Toto řešení je podobné tomu, které je implementováno ve výchozím plánovači. Zde má ale semafor hodnotu odpovídající počtu úloh a nikoliv počtu vláken s neprázdným veřejným seznamem úloh.

Centralizovanost přístupu a nemožnost vyvažovat zátěž vláken jsou důvody, proč tato implementace nebyla zvolena.

Finální implementace

Zvolené řešení se snaží vyvarovat všech nedostatků zmíněných v předchozích částech. Snaží se tedy minimalizovat počet uspání vláken a implementuje vyvažování zátěže tím, že umožňuje cíleně probouzet určité vlákno, resp. je probouzet podle priorit.

Klíčová myšlenka spočívá v tom, že se průběžně počítají hodnoty tří proměnných:

- Počet *běžících vláken*, tj. vláken, která nejsou uspaná.

[1] Condition variable

- Počet *volných vláken*, tedy takových, která sice běží, ale nevykonávají žádnou úlohu.
- Počet naplánovaných úloh.

Platí, že volná vlákna se snaží aktivně vyhledávat úlohu ke zpracování. Pokud ji najde, zůstane sice běžícím vláknem, ale přestane být volným. Jakmile úlohu dokončí, stane se opět volným.

Volné vlákno má právo se kdykoliv rozhodnout přestat vyhledávat úlohy (např. po určitém počtu pokusů) a požádat o uspání. Této žádosti může a nemusí být vyhověno. Aby nemohlo nikdy dojít k tomu, že všechna vlákna budou uspaná, zatímco jsou k dispozici úlohy ke zpracování, je zachováván vždy tento invariant: pokud existuje alespoň jedna naplánovaná úloha, musí vždy existovat alespoň jedno běžící vlákno. Tímto invariantem je zajištěno, že systém nikdy nemůže přestat reagovat na příchozí požadavky, neboť dokud existuje alespoň jedna úloha, nemůže být poslední vlákno uspané.

Na druhou stranu může zřejmě dojít k situaci, kdy všechna vlákna budou uspaná, kromě jednoho, které provádí veškerou práci. Aby k této situaci nedocházelo, je v okamžiku vzniku úlohy provedena kontrola, zda existuje volné vlákno, které by tuto úlohu mohlo začít vykonávat. Pokud volné vlákno není, ale existuje nějaké uspané vlákno, pak je probuzeno, čímž se z něj stane běžící volné vlákno, které tuto úlohu začne zpracovávat. Samozřejmě je možné, že úloha bude nakonec získána někým jiným – např. tehdy, kdy některé jiné vlákno v tuto chvíli dokončí svojí úlohu a stihne si přivlastnit nově vytvořenou úlohu dříve než právě probuzené. To ale nevadí, neboť po několika neúspěšných pokusech takové vlákno může požádat opět o uspání.

Takovýto systém má velkou výhodu v tom, že není příliš striktní. Vlákno může bez rizika zamrznutí systému zažádat kdykoliv o uspání i přesto, že může existovat úloha, kterou by mohlo vykonat. Tento fakt výrazně zjednodušuje implementaci plánovače.

Samotná vlákna jsou uspávána pomocí semaforů, přičemž každé má k dispozici svůj vlastní semafor. Díky tomu je možné regulovat zátěž vláken. Lze tak např. probudit určité vlákno, či zajistit, aby byla prioritně probouzena vlákna v určitém pořadí.

Výše popsané chování implementuje třída `task_manager`, která poskytuje metody, pomocí kterých je objekt informován o tom, že přibyla nová úloha, že vlákno začalo vykonávat úlohu (přestalo být volným), že se snížil počet úloh a že vlákno dokončilo úlohu (stalo se opět volným). Dále poskytuje metodu, kterou vlákna volají, pokud se rozhodla zažádat o uspání. Tato metoda buď volajícího uspí nebo se okamžitě vrátí (v případě, že by uspáním byl porušen zmíněný variant).

Pro korektní implementaci je nutné zajistit, že všechny tři proměnné budou měněny atomicky najednou, aby se nemohlo stát např. to, co je naznačeno v programu 6.1. Pro snažší pochopení příkladu je nutné poznamenat, že o uspání žádá vždy pouze volné vlákno, což je důsledkem předchozích odstavců. Na začátku je jedno volné vlákno a žádná úloha. Na konci je jedna úloha a žádné běžící vlákno, což je spor se zmíněným invariantem.

pracovní vlákno žádá o uspání

```
if (tasks == 0
    || running_threads > 1) {

  --free_threads;
  --running_threads;
  sleep();
}
```

`scheduling_manager` přidává iniciační úlohu

```
++tasks;
if (free_threads == 0) {
  wakeup(some_thread);
}
```

Program 6.1: Zamrznutí systému kvůli špatné synchronizaci

Zřejmě je tedy nutné atomicky provést operaci: „Pokud může být vlákno uspáno, sniž počet běžících a volných vláken.“ Samotné uspání již atomické být nemusí. Je ale důležité zajistit, aby za každé snížení počtu běžících vláken bylo skutečně nějaké vlákno uspáno a za každé zvýšení bylo nějaké vlákno probuzeno. To je pouze technický detail a lze řešit např. polem atomických proměnných typu `bool`, kde každý prvek odpovídá jednomu vláknu a určuje, zda vlákno běží či nikoliv. Úspěšná atomická změna z `false` na `true` pak znamená, že je možné snížit hodnotu semaforu odpovídajícího vlákna a tím probudit vlákno, změna z `true` na `false` odpovídá zvýšení hodnoty semaforu a tím uspání vlákna. Pokud se cyklicky prochází tímto polem, dokud se nepodaří jedno z vláken probudit, je zachována konzistence mezi hodnotou proměnných a skutečným počtem běžících vláken.

Atomičnost byla experimentálně implementována dvěma způsoby. První používal pro synchronizaci přístupu k proměnným spin-lock. Ukázalo se, že je ale úzkým hrdlem systému, neboť tento spin-lock je sdílen všemi vlákny v rámci jednoho uzlu.

Druhý způsob používá atomické instrukce poskytované instrukční sadou procesoru. Protože ale je možné atomicky měnit pouze jednu proměnnou v jeden okamžik, je při implementaci použit následující trik: Jedna 64b atomická proměnná je rozdělena tři části – 16b pro počet volných vláken, 16b pro počet běžících vláken a 32b pro počet úloh. Experimentálně je tato bezzámková implementace o 3 až 4% rychlejší než při synchronizaci pomocí spin-locku.

6.3 Podpora NUMA systémů

Podpora NUMA systémů je velmi důležitá, neboť nezohlednění vlivu pomalejších a rychlejších pamětí může mít výrazný negativní vliv na výkon systému. Z měření v části 5.1.2 vyplývá, že je klíčové, aby procesory jednoho uzlu pracovaly pokud

možno výhradně s pamětí, která patří tomuto uzlu.

Přímočaré řešení tohoto problému spočívá v zajištění, aby jeden požadavek mohl běžet pouze na jednom uzlu a všechny jeho úlohy tak spouštět pouze na procesorech tohoto uzlu. Tento postup jistě splní podmínku zmíněnou v předchozím odstavci. V souvislosti s tím se ale objevují dva velké problémy. První potíž spočívá v tom, že pokud má požadavek běžet pouze na jednom uzlu, tak není jisté, podle čeho při vzniku nového požadavku vybrat tento cílový uzel. Druhý problém souvisí s tím, že pokud strategie pro výběr uzlu, na kterém bude požadavek spuštěn, selže, budou některé uzly budou vytíženy mnohem více než jiné. To se samozřejmě může stát velmi snadno, neboť požadavek nenese žádnou informaci o tom, jak moc časově a paměťově je náročný. Takže jakákoliv strategie, která se rozhoduje pouze při vkládání nového požadavku a dále nezohledňuje další chod systému, může zklamat. Navíc to zabraňuje využití více než jednoho uzlu pro výpočet jednoho požadavku.

Ačkoliv je implementován mechanismus vyvažování zátěže během chodu systému, může tento algoritmus přinášet určité výkonnostní problémy, neboť přesun rozpracovaného požadavku z jednoho uzlu na druhý znamená, že v cílovém uzlu bude pracováno s pamětí, která byla alokována na původním uzlu. Jistou optimalizací by bylo, přemapovat takto alokovanou paměť z původního uzlu na jiný tak, jak je to řešeno např. v plánovači pro systém VMWare ESX [16], to ale nelze na úrovni aplikačního programu řešit, neboť podporované operační systémy pro tuto operaci neposkytují funkce.

Zřejmě je tedy důležité, aby strategie pro výběr uzlu pro spuštění požadavku byla co nejvíce efektivní, a algoritmus pro vyvažování zátěže tak musel přesouvat požadavky mezi uzly co nejméně.

6.3.1 Výběr uzlu pro spuštění požadavku

Při výběru uzlu, na kterém bude nový požadavek spuštěn, je nutné vzít v úvahu následující faktory:

- Zatížení jednotlivých uzlů.
- Velikost dostupné paměti uzlů.
- Rychlost uzlů, neboť jak bylo změřeno v grafu 5.4, nemusí být všechny uzly stejně rychlé.

Zatížení uzlu lze snadno spočítat jako počet vláken, která právě vyvíjí nějakou činnost. Toto číslo lze rovněž vyjádřit jako poměr mezi počtem běžících vláken a celkovým počtem vláken uzlu.

Velikost dostupné paměti uzlů je snadno zjistitelná, neboť operační systémy přímo nabízejí funkce pro zjištění této hodnoty. Tento údaj je rovněž vyjádřen jako poměr mezi dostupnou pamětí a celkovou velikostí jeho paměti.

Za rychlosti uzlů lze považovat čísla na diagonále v matici vzdáleností mezi uzly systému (viz tabulka 5.1). I tento faktor je vyjádřen jako poměr mezi rychlostí nejrychlejšího a rychlostí zkoumaného uzlu.

Při výběru uzlu je postupováno v následujících krocích:

1. Nejdříve jsou vybráni tzv. kandidáti, což jsou uzly takové, které mají nejkratší vstupní frontu požadavků. Zřejmě nemá smysl předávat nový požadavek uzlu, který má neprázdnou vstupní frontu, když existuje uzel, který má tuto frontu kratší resp. prázdnou.

2. Z těchto kandidátů jsou vyloučeny ty uzly, u kterých není pravděpodobné, že by jejich systémové prostředky mohly stačit na vyřízení požadavku. Aby uzel tuto podmínku splnil, musí být splněna nerovnost:

$$1 - \text{vytížení} > 2 \times \frac{\text{vytížení}}{\text{počet nezpracovaných požadavků}}$$

   kde

$$\text{vytížení} = \max(\text{zjištěné vytížení}, \frac{\text{počet nezpracovaných požadavků}}{\text{počet vláken}})$$

   Vytížení je jednou bráno jako paměťové a podruhé jako výpočetní. Pro splnění kritéria musí být splněny nerovnosti pro oba druhy. Hodnota, kterou vrací funkce *max*, se dá označit jako odhadované vytížení. Pokud totiž přijde několik požadavků najednou, může se stát, že při vyčíslování tohoto vztahu ještě není vyhodnocování zcela rozběhnuté, takže ačkoliv uzel vyřizuje mnoho požadavků, může být jeho zjištěné vytížení poměrně nízké a naprosto neodpovídající realitě v blízké budoucnosti. Proto se používá maximum aktuálního vytížení a předpokládaného vytížení, které se spočítá na základě jednoduchého předpokladu, že jeden požadavek vytíží jedno vlákno. Toto vytížení je vyděleno počtem požadavků, čímž se získá kolik prostředků průměrně spotřebuje jeden požadavek. Výraz $1 -$ vytížení pak říká, kolik prostředků uzlu ještě zbývá. Nerovnost tedy platí, pokud zbývá více prostředků než dvojnásobek toho, kolik spotřebuje průměrný požadavek.

3. Pokud jsou vyloučeny všechny uzly, vybere se jednoduše ten, který má k dispozici nejvíce dostupné paměti. V opačném případě se vybere náhodný nejrychlejší uzel, který předchozí kritérium splnil.

Tímto způsobem se tedy při novém požadavku vybere cílový uzel a do jeho vstupní fronty se vloží nový požadavek, v jehož frontě se nachází iniciační úloha.

6.3.2 Vyvažování zátěže

Již bylo zmíněno v úvodu této části, že výběr uzlu nemůže být dokonalý. Předpokládá totiž, že nový požadavek se bude chovat průměrně. Takový předpoklad ale jistě nemusí platit a systém se může stát velmi nerovnoměrně zatížený. Dále bylo uvedeno, že takto nemůže být jeden požadavek spuštěn na více než jednom uzlu, ačkoliv by to mohlo urychlit jeho vyřízení.

Je tedy zřejmé, že by plánovač měl implementovat mechanismus pro přesun a sdílení požadavků mezi jednotlivými uzly. Takový mechanismus by umožnil přesunout

celý požadavek z přetíženého uzlu na méně vytížený, případně provádět výpočet jednoho požadavku paralelně na několika uzlech najednou.

Jak bylo popsáno v části 5.5.4 jsou datově nevázané úlohy jednotlivých požadavků umístěné v navzájem oddělených frontách. Díky této skutečnosti je možné popsaný mechanismus poměrně snadno implementovat, neboť přesun požadavku odpovídá přesunutí požadavku ze vstupní/pracovní fronty jednoho uzlu do vstupní fronty druhého uzlu. Sdílení požadavku znamená sdílení tohoto požadavku mezi několika uzly.

Z hlediska implementace je nutné vyřešit mnoho technických detailů – jako např. synchronizaci přístupu k jednotlivým datovým strukturám, zodpovědnost za ukončení požadavku a smazání jeho pomocných dat. Navíc je nutné tento mechanismus naimplementovat tak, aby byl kompatibilní s mechanismem uspávání vláken.

Synchronizace fronty úloh jednotlivých požadavků je poměrně jednoduchá, neboť stačí každou frontu spojit se zámkem, který bude synchronizovat přístup k ní. Synchronizace pracovní i vstupní fronty požadavků by šla snadno řešit stejným způsobem, ale to by znamenalo, že pro odebrání jedné datově nevázané úlohy je nutné získat dva zámky, což se negativně projeví na výkonu systému.

Proto je pracovní fronta (do které je intenzivně přistupováno) implementována způsobem, kdy je nutné získat zámek pouze tehdy, když je do ní vkládán nebo odebírán požadavek. Při implementaci je využito toho, že operace zápisu a čtení ukazatele je atomická a že nevadí, když se na chvilku v zásobníku ocitne jeden požadavek několikrát. Protože se úloha vyhledává tak, že se najde nejstarší požadavek takový, který má neprázdnou frontu, ze které je odebrána úloha, dojde tak pouze k tomu, že jeden požadavek bude zkoumán několikrát.

Znamená to tedy, že jediný zámek, který je nutné pro nalezení datově nevázané úlohy získat, je ten, který synchronizuje přístup k frontě úloh jednoho požadavku. Navíc před požadavkem na zamknutí tohoto zámku je ověřeno, zda tato fronta není prázdná. Tato implementace experimentálně zvyšuje výkon o 3-4%, neboť není zamykaná žádná globální datová struktura, ale až teprve dílčí a to jen tehdy, kdy to má smysl (fronta úloh není prázdná). Struktura uchovávající datově nevázané úlohy se tak nestává úzkým hrdlem systému.

Požadavek je ukončen tím uzlem, který vykonal jeho poslední úlohu. Odpovídající instance třídy `node_manager` pak oznámí instanci `scheduling_manager`, že požadavek byl ukončen a tato instance se postará vymazání požadavku ze všech ostatních uzlů.

Kompatibilita s mechanismem uspávání vláken je zajištěna tak, aby vždy platilo, že za každou úlohu je zodpovědný nějaký uzel. Nemůže se tedy stát, že by všechna vlákna jednoho uzlu, který je za nějakou z nezpracovaných úloh zodpovědný, byla uspána.

Při přesunutí požadavku z jednoho na druhý je jednoduše zodpovědnost za všechny úlohy přenesena na cílový uzel. Drobná potíž nastává při sdílení fronty, kdy nemusí být jasné, který uzel je zodpovědný za kterou úlohu. Proto je každá datově nevázaná úloha doplněna o informaci, který uzel je za ni zodpovědný. Pokud pak vlákno uzlu zpracuje úlohu, která patří jinému uzlu, je majitel této úlohy notifikován, že úloha již byla zpracována a že se o ní nemusí dále starat.

Další potíž, která může nastat, je ta, že při sdílení určitého požadavku, není vy-

loučeno, že některé uzly tento požadavek vůbec nezačnou zpracovávat a všechna jeho vlákna se uspí. To se může stát tehdy, pokud se požadavek sice tomuto uzlu předá, ale nebude v něm existovat úloha, za kterou by měl uzel zodpovědnost. Aby k tomuto nedocházelo a požadavek byl rovnoměrně zpracováván všemi sdílejícími uzly, je implementován algoritmus, že zodpovědnost za každou novou úlohu je přidělována metodou round-robin jednotlivým uzlům.

Algoritmus vyvažování zátěže

V okamžiku, kdy je implementován mechanismus pro přesun a sdílení požadavků mezi vlákny, zbývá implementovat logiku, která bude těchto funkcí využívat pro zajištění vyváženosti systému.

Vyvažování zátěže má na starosti speciální vlákno, které každých 100ms analyzuje zatížení jednotlivých uzlů a aplikuje algoritmus na jejich vyvážení. Aby toto vlákno nezdržovalo v případě zcela vytíženého systému, platí pravidlo, že toto vlákno běží jenom tehdy, pokud je alespoň jedno výpočetní vlákno uspané.

Algoritmus pro vyvážení funguje v následujících krocích:

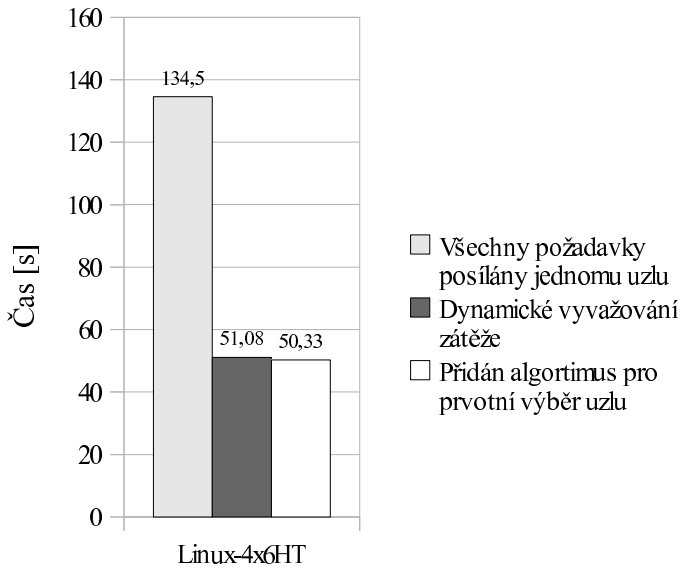
1. Nejdříve jsou vybrány množiny *vytížených* a *volných uzlů*. Podmínka pro to, aby uzel mohl být prohlášen za vytížený, je, že jeho výpočetní nebo paměťové vytížení je větší než 90% nebo má neprázdnou frontu iniciačních úloh. Volný uzel pak musí splnit podmínku, že má tuto frontu prázdnou a zároveň je jeho výpočetní a rovněž i paměťové vytížení menší než 50%. Protože se výpočetní vytížení uzlu může měnit velmi dynamicky, je uzel prohlášen za vytížený resp. volný teprve tehdy, kdy splní výše uvedenou podmínku 4krát za sebou, tj. v průběhu 400ms.

2. Pokud je alespoň jedna z těchto množin prázdná, žádné vyvažování zátěže se nekoná. V opačném případě se postupně procházejí přetížené uzly a dokud existuje neprázdná množina volných uzlů, provádí se kroky 3 až 6.

3. Vybere se nejméně rozpracovaný požadavek z přetíženého uzlu. Takový lze nalézt velmi snadno, neboť je umístěn buď ve vstupní frontě nebo je to nejmladší požadavek v pracovní frontě.

4. Pokud uzlu stále zbývají nějaké požadavky, je tento požadavek z uzlu zcela odstraněn, pokud je to poslední požadavek, pak tento požadavek odstraněn není a bude sdílen (takový požadavek vytěžuje celý uzel).

5. Následně je vybrán volný uzel, který je aktuálnímu přetíženému uzlu nejblíže ve smyslu vzdálenosti mezi uzly NUMA systému. Pokud vybraný požadavek není uzlem zpracováván (to se může stát tehdy, kdy je tento požadavek již sdílen a přesto uzel vytěžuje), a zároveň požadavek na tomto uzlu ještě neběžel, je požadavek vložen do vstupní fronty cílového uzlu. V opačném případě se zkouší najít další volný uzel. Pokud se tento požadavek nepodaří nikomu předat, je vrácen zpět původnímu uzlu. Požadavek není předán uzlu, ve kterém již běžel,



Obrázek 6.2: Vyvažování zátěže mezi uzly na počítači *Linux-4x6HT*

proto, aby nemohlo dojít k situaci, kdy je jeden požadavek neustále přesouván mezi jednotlivými uzly.

6. Uzel, kterému je předán požadavek, je odstraněn z množiny volných uzlů.

Účinnost tohoto algoritmu ukazuje graf 6.2, který byl získán tak, že bylo najednou spuštěno 200 požadavků na počítači *Linux-4x6HT*. V prvním případě byly všechny přidělovány pouze jednomu uzlu a algoritmus pro vyvažování zátěže byl potlačen. Ve druhém případě již vyvažování zátěže bylo zapnutné a ve třetím byl aktivní rovněž algoritmus pro prvotní výběr uzlu. Z grafu je vyplývá, že popsaný algoritmus je poměrně efektivní a že algoritmus pro prvotní výběr uzlu výkon dále zvyšuje.

6.4 Afinita vláken

Nastavení správné afinity vláken jednotlivých plánovačů je velmi důležitý krok při inicializaci plánovače. Na SMP se lze spolehnout na to, že operační systém bude vlákna pouštět na všech procesorech rovnoměrně a že je mezi procesory nebude přesouvat. Na druhou stranu toto již nemusí zcela platit tehdy, pokud se vlákno uspí a pak opět probudí. Ačkoliv na SMP to není větší problém, na systémech NUMA by přesunutí vlákna z jednoho uzlu na jiný mohlo mít velmi negativní důsledky.

Nastavení afinity jednotlivých vláken tak, aby každé vlákno běželo na svém přiděleném logickém procesoru, zajistí, že žádné nečekané potíže se změnou plánování vlákna nemohou nastat.

6.5 Podpora Hyper-Threadingu

Z výsledků měření vlivu Hyper-Threadingu na výkonnost systému vyplývalo (viz část 5.1.3), že vliv této technologie nelze rozhodně zanedbat. Vzhledem k použité implementaci snižování zátěže systému pomocí uspávání vláken je velmi snadné regulovat vytíženost jednotlivých vláken. Vlákna jsou probouzena buď explicitním probuzením vlákna, o které bylo požádáno, nebo je vybráno první takové ze seznamu vláken, které je uspané. Znamená to tedy, že vlákna ze začátku seznamu vláken jsou probouzena prioritně před vlákny z konce seznamu. Vhodným setříděním tohoto seznamu je možné regulovat vytíženost jednotlivých vláken a vzhledem k pevně nastavené afinitě každého vlákna tedy i vytíženost jednotlivých logických procesorů.

Podpora Hyper-Threadingu tedy obnáší pouze to, aby byl tento seznam setříděn tak, že v první půlce seznamu budou vždy první procesory z páru a ve druhé půlce pak druhé z páru.

6.6 Vliv vyrovnávacích pamětí

Jak bylo uvedeno v části 5.1.1, velikost a uspořádání vyrovnávacích pamětí má nezanedbatelný vliv na výkon systému. Hlavně velikost obálek úzce souvisí s tímto faktorem.

Z té samé části pak vyplývá, že prioritou není posílat co největší obálky, které se ještě vejdou do cache, ale naopak obálky co nejmenší, aby docházelo k efektu synchronizace vyrovnávacích pamětí. Na druhou stranu zmenšení velikosti obálky zvětšuje režii spojenou s posíláním obálek a plánováním krabiček. Jsou tedy dva faktory, které optimální velikost obálky ovlivňují, a jejich požadavky na velikost obálky jsou zcela opačné.

Optimální velikost obálky by měla být zvolena tak, aby nastal rozumný kompromis mezi těmito faktory. Platí ale, že vliv synchronizací vyrovnávacích pamětí je poměrně nespolehlivý a záleží na tvaru grafu požadavku, typu krabiček i počtu paralelně zpracovávaných požadavků. Tento faktor není možné jakkoliv dopředu odhadnout a prioritu by tak mělo hrát omezení zmíněné režie.

Řešení, které bylo nakonec zvoleno, se snaží nastavit velikost doporučené obálky tak, aby obálka byla co nejmenší možná, ale aby režie zůstala zanedbatelná. Pokud menší obálka urychlí výpočet, pozitivně se to projeví na výkonu systému. V opačném případě je alespoň zaručeno, že systém nebude režií zbytečně zatěžován.

Velikost obálky je v souladu s měřením zvolena tak, aby byla nejvýše 200kB a zároveň nebyla vyšší než velikost L2 cache. Pokud je L2 cache sdílená, je její velikost ještě vydělena počtem logických procesorů, které ji sdílí. Pro všechny testovací počítače podle tohoto pravidla vychází velikost obálky 200kB. Experimentálně je ale ověřeno, že při této velikosti má systém stabilně dobrý výkon na všech počítačích bez ohledu na počet požadavků.

Protože systém ve výchozí implementaci poskytoval pouze funkci pro vytvoření obálky se zadaným počtem řádků a nikoliv zadané velikosti v bytech, je tato funkce rozšířena o chování, že pokud je požadovaný počet řádků nula, je vytvořena obálka

s doporučenou velikostí. Tato úprava nevyžaduje změnu zdrojových kódů, které funkci volají, a lze se tak rozhodnout, zda vytvořit obálku s přesným počtem řádků, či doporučeným. Konečný počet řádků je spočítán podle vztahu:

$$\text{počet řádků} = \max(\lfloor \frac{\text{velikost obálky} - \text{velikost skalárních dat}}{\text{celková velikost jednoho řádku}} \rfloor, 1)$$

6.6.1 Zohlednění sdílení vyrovnávacích pamětí

Velikost vyrovnávacích pamětí určuje, jak veliké obálky se budou posílat. V předchozí části byl navržen jednoduchý vztah mezi velikostí a sdíleností vyrovnávacích pamětí a velikostí obálek. Sdílenost v něm hraje pouze tu roli, že je spočítáno jak velká část této paměti připadá na jeden logický procesor, aby si procesory navzájem nevytlačovaly obálky z paměti.

Jak bylo uvedeno v části 5.1.1, je režie nižší při přesunu výpočtu z jednoho logického procesoru na jiný, pokud spolu sdílí vyrovnávací paměť. I s tímto faktorem nový plánovač počítá, ačkoliv to má téměř neměřitelný vliv na výkon.

Plánovač sdílení vyrovnávnávacích pamětí zohledňuje tím, že každé vlákno si při inicializaci připraví pořadí, ve kterém se bude pokoušet okrádat jiná vlákna o úlohy. Tento seznam je sestaven tak, že se postupně prochází vyrovnávací paměti od nejvyšší úrovně (L1) po nejnižší. Pro každou úroveň jsou na konec seznamu přidány logické procesory, které s procesorem, pro který je seznam vytvářen, cache dané úrovně sdílí a ještě v seznamu není. Nakonec jsou do seznamu přidány dosud nepřidané procesory. Aby byl systém vykrádání spravedlivý jsou úseky v seznamu odpovídající jednotlivým úrovním cache včetně úseku odpovídající ostatním procesorům randomizovány.

Kapitola 7

Paměťový alokátor

Implementace efektivního paměťového alokátoru pro systém Bobox je druhým cílem této práce. Ve výchozí implementaci používá Bobox alokátor poskytovaný jazykem C++, tj. instanci třídy `std::allocator`. Například v případě operačního systému Linux je tento alokátor napsaný velmi efektivně a s ohledem na použití ve vícevláknovém prostředí [12]. Navíc alokátory v obou podporovaných operačních systémech obsahují elementární podporu systémů NUMA (viz část 7.1.3).

Na druhou stranu je tento alokátor napsaný poměrně obecně, neboť je určen pro použití ve všech možných (a tím pádem velmi různorodých) aplikacích. V systému Bobox jsou požadavky na alokátor trochu specifičtější a některých jeho vlastností lze využít k implementaci, která bude nakonec v závislosti na systému a překladači více či méně efektivnější než standardní alokátor.

V této i následující kapitole bude dodržována konvence, že alokátor přiděluje tzv. *objekty* určité velikosti bez ohledu na to, zda se jedná o objekt nebo např. o pole. *Bloky* pak označují větší úseky, přičemž např. objekty mohou být přidělovány z takových bloků. Termínem *superbloky* budou označovány ještě větší úseky paměti stejné velikosti.

7.1 Faktory ovlivňující efektivitu alokátoru

Stejně jako v plánovači bylo mnoho okolností, které ovlivňovaly jeho efektivitu, tak i v případě alokátoru je nutné vzít v úvahu některé faktory. Ty jsou přitom podobné těm, které byly uvedeny v případě plánovače. Tentokrát se ale na efektivitě podílejí jiným způsobem.

7.1.1 Vyrovnávací paměť procesorů

V případě plánovače ovlivňovala vyrovnávací paměť rychlost zpracovávání příchozích obálek. Jinými slovy se plánovač snažil optimalizovat práci s přidělenými bloky paměti. Alokátor na druhou stranu nemá žádnou možnost, jak ovlivňovat práci s již přidělenými bloky, což ale neznamená, že by vliv vyrovnávacích pamětí neměl brát v úvahu. Platí totiž, že v okamžiku, kdy se přidělený objekt vrací alokátoru, je

zvýšená pravděpodobnost, že je tento objekt uložen v cache, neboť se s ním pravděpodobně nedávno pracovalo. Pokud takový blok bude co nejdříve opět přidělen, bude práce s ním rychlejší.

Takže zatímco alokátor optimalizuje alokátor práci s volnými bloky paměti, plánovač s těmi obsazenými, což je výhodné, neboť se tyto komponenty systému navzájem nijak neovlivňují, pouze doplňují.

7.1.2 Vícevláknové prostředí

Vzhledem k tomu, že volné a obsazené bloky v paměti jsou spravovány určitými datovými strukturami, a zároveň k tomu, že požadavky na alokaci a uvolnění bloků paměti mohou přicházet paralelně, je nutné nějakým způsobem řešit synchronizaci přístupu ke zmíněným datovým strukturám.

Nejjednodušeji to lze samozřejmě řešit globálním synchronizačním primitivem, které zajistí vzájemnou výlučnost přístupů k těmto strukturám. Efektivita toho přístupu ale klesá s tím, jak roste počet konkurenčních vláken, které k paměti přistupují. Při větším počtu vláken se tento globální zámek stává úzkým hrdlem systému, takže v efektivním alokátoru je nutné se takového přístupu maximálně vyvarovat.

Další možností je použít vláknově bezpečné datové struktury, ty jsou ale buď rovněž synchronizovány společným primitivem, nebo jsou řešeny bezzámkově. Tento přístup je ale implementačně poměrně náročný a navíc takové struktury bývají méně efektivní než jednovláknové varianty.

Rovněž výše zmíněný vliv vyrovnávacích pamětí souvisí s problematikou vícevláknového prostředí. Platí, že různá vlákna pravděpodobně (v Boboxu zcela jistě) běží na rozdílných procesorech pro dosažení maximální paralelizace. Pokud je objekt vrácen jedním vláknem, bude pravděpodobně ve vyrovnávací paměti odpovídajícího procesoru. Výše popsaná optimalizace tak nebude fungovat, pokud tento objekt bude vrácen při požadavku na alokaci z druhého vlákna (pokud tedy oba procesory nesdílejí vyrovnávací paměť). Z tohoto důvodu je třeba rozlišovat mezi jednotlivými vlákny.

7.1.3 Systémy NUMA

Vlastnosti systémů NUMA byly již popsány v kapitole 5.1.2. Důležité u nich je tedy to, že všechna paměť není z pohledu procesoru stejná, ale že přístup k některým částem je rychlejší a k některým pomalejší. Je tedy zřejmé, že alokátor by měl důsledně kontrolovat, z jakých částí paměti se kterému vláknu paměť přiděluje.

Na tomto místě je vhodné rovněž zmínit, že standardní alokátory, které poskytuje operační systém resp. samotný programovací jazyk, podporují NUMA systémy pouze částečně. Tento faktor zohledňují pouze ve chvíli, kdy vlákno poprvé přistoupí na virtuální stránku, která ještě nebyla namapovaná na fyzickou. V tu chvíli se použije fyzická stránka, která odpovídá uzlu, ze kterého bylo na stránku přistoupeno [9]. Pokud je veškerá paměť uzlu již obsazená, použije se paměť jiného uzlu. Takové chování sice způsobuje, že nově přidělená paměť bude patřit tomu, kdo s ní poprvé

pracoval, ale pokud je taková paměť vrácena a cacheována, už není při opětovném přidělení rozlišováno, jakému uzlu ve skutečnosti patří.

7.1.4 Vliv stránkování

Alokátory zpravidla pro alokaci větších bloků (řádově od stovek kilobytů) přidělují paměť přímo voláním funkce systému pro správu virtuální paměti[1]. Tyto funkce se chovají téměř identicky - najdou oblast požadované velikosti ve virtuální paměti a zarezervují pro ni fyzickou paměť. Při prvním přístupu na některou z takto přidělených stránek dojde k vytvoření mapování mezi touto virtuální stránkou a fyzickou. Při uvolnění přiděleného bloku je pak mapování opět zrušeno, fyzické stránky uvolněny a odpovídající virtuální stránky jsou označené jako volné.

Tento mechanismus způsobuje to, že alokace paměti je poměrně rychlá, ale první přístup do takové paměti je poměrně pomalý.

Nejlépe tento jev lze demonstrovat programy 7.1 a 7.2:

```
for (int i=0; i<ITERATIONS; i++) {
        int *buf = (int*)malloc(SIZE*sizeof(int));
                for (int j=0; j<SIZE; j++)
                        buf[j] = j+i;
        free(buf);
}
```

Program 7.1: Přístup na nově alokované bloky

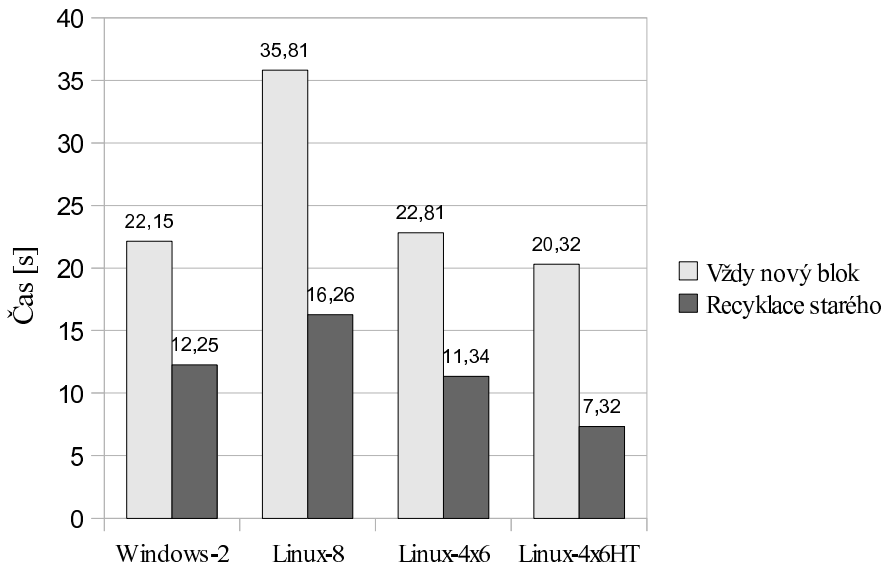
```
int *buf = (int*)malloc(SIZE*sizeof(int));
for (int i=0; i<ITERATIONS; i++) {
        int *tmp = (int*)malloc(SIZE*sizeof(int));
        for (int j=0; j<SIZE; j++)
                buf[j] = j+i;
        free(tmp);
}
free(buf);
```

Program 7.2: Přístup na stále stejný blok

Platí, že for-cyklus ve druhém kódu je rychlejší než v prvním uvedeném, ačkoliv se uvnitř cyklu cyklu provádějí identické operace. Jenom zápis probíhá jednou do vždy nově alokovaných oblastí, zatímco podruhé do stále stejné paměti.

Rozdíl v rychlostí samozřejmě závisí na velikosti konstanty `SIZE`. Pokud je hodnota zvolená tak, že celková velikost alokované paměti se vejde do vyrovnávací paměti procesoru, je to poměrně očekávaný výsledek, neboť celé pole bude uložené v této paměti, do které je přístup rychlý.

[1] `mmap` resp. `munmap` v Linuxu a `VirtualAlloc` resp. `VirtualFree` ve Windows



Obrázek 7.1: Vliv stránkování na rychlost práce s alokovanými bloky

Pokud ale hodnota `SIZE` bude řádově větší než velikost vyrovnávací paměti, pak cache přestane hrát roli. Přesto je druhý for-cyklus stále rychlejší, jak dokazuje graf 7.1 a je to právě ze zmíněného důvodu. Pro účely měření bylo použito 512MB paměti a 50 iterací.

Je tedy zřejmé, že recyklací větších alokovaných bloků lze dosáhnout lepšího výkonu alokátoru. Resp. výkon alokátoru se zřejmě nezvýší, ale zvýší se rychlost programu, který s přidělenými bloky bude pracovat.

7.2 Rozhraní alokátoru

Jazyk C++ má určité specifické požadavky na třídu, aby mohla být použita jako alokátor paměti. V zásadě se jedná o šablonovou třídu, která je schopná alokovat souvislé místo v paměti, do kterého se vejde požadované množství elementů, toto místo opět uvolnit a vrátit maximální počet elementů, pro něž lze najít místo v paměti. Přesné požadavky na třídu lze nalézt ve specifikaci jazyka C++ [1].

Tyto požadavky vyžadované normou C++ splňuje třída `thr_allocator`, která je implementovaná jako adaptér nad třídou `internal_thr_allocator`. Ta má mnohem jednodušší rozhraní, neboť musí poskytovat pouze tři metody. První je metoda `void *allocate(size_t size)`, která vrací souvislý úsek požadované délky v bytech, dále metodu `void deallocate(void *ptr, size_t size)`, která uvolní úsek paměti přidělený metodou `allocate`. Vždy platí, že hodnota parametru `size` při uvolnění úsek je stejná jako při jeho přidělování. A metodu `size_t max_size()`, která má vracet délku největšího možného úseku, který lze přidělit. Není vyžadováno, aby toto číslo bylo přesné. To znamená, že není vyžadováno, aby se následná alokace bloku velikosti, která byla vrácená metodou `max_size`, musela povést.

Další důležitý fakt je, že každé vlákno v systému je spřaženo s nějakou instancí třídy `internal_thr_allocator`, takže jich může být v celém systému více. Třída `thr_allocator` pak převolává metody té instance, která odpovídá vláknu, ze kterého byl vznesen požadavek na alokaci/uvolnění paměti. Cílem tedy je naimplementovat všechny tři metody třídy `internal_thr_allocator` a zajistit distribuci instancí této třídy mezi jednotlivá vlákna.

7.3 Specifické vlastnosti systému Bobox

Jak bylo zmíněno v úvodu, systém Bobox na rozdíl od jiných programů pracuje s pamětí poněkud specifickým způsobem. Systém vyhodnocuje jednotlivé požadavky, přičemž během zpracování jednoho požadavku dochází k alokaci a uvolňování paměti. Pokud je ale práce s touto pamětí korektní a nedochází k únikům paměti, je veškerá paměť naalokovaná požadavkem uvolněná při jeho ukončení.

Samozřejmě, že během instanciace grafu požadavku je také alokována paměť, ale ta je uvolněna rovněž při jeho ukončení, takže i tyto alokace jsou v souladu s tím, co bylo napsáno v předchozím odstavci.

Z výše zmíněného a z dalších vlastností systému tak plynou následující vlastnosti:

1. Programy se zpravidla z hlediska alokace chovají jako jeden celek. To znamená, že mezi jednotlivými požadavky na alokaci paměti nelze nijak rozlišovat. V Boboxu se alokace dají striktně rozdělit do skupin podle toho, ke kterému požadavku přísluší. Platí, že paměť přidělená při zpracování jednoho požadavku bude vždy patřit tomuto požadavku.

2. Při alokaci v běžném programu není jisté, zda bude přidělená paměť vůbec někdy uvolněna a pokud ano, nelze vytvořit žádný předpoklad o tom, kdy se tak stane. Naproti tomu zde platí, že každá alokovaná paměť (kromě inicializace systému) bude vždy uvolněna zároveň s ukončením požadavku. Navíc takto bude určitě uvolněna celá skupina zmíněná v prvním bodu.

3. V běžném programu nelze předpokládat nic o počtu vláken, ze kterých bude paměť alokovaná. V Boboxu je počet těchto vláken pevně daný při spuštění systému.

4. Za normálních okolností program používá dvojici funkcí `malloc` a `free` resp. jejich sémantické ekvivalenty. Důležité je, že první funkce vyžaduje velikost potřebné paměti a druhá pouze ukazatel na uvolňovaný blok. V alokátoru pro Bobox ale i metoda pro uvolnění paměti dostane velikost bloku. To vyplývá z rozhraní popsaného v části 7.3.

5. Bobox není určený pro použití v realtimových systémech, ale pouze pro zpracování dat. Znamená to tedy, že lze používat datové struktury, které mají zajištěnou určitou amortizovanou časovou složitost na operaci a nikoliv pouze složitost v nejhorším případě.

Otázkou ale stále zůstává, jak konkrétně tyto vlastnosti mohou pomoct. První dvě vlastnosti samy o sobě mnoho neříkají. Pokud se ale spojí dohromady, platí důležité pravidlo – existují disjunktní skupiny alokací, přičemž veškerá alokovaná paměť z jedné skupiny bude v konečném čase uvolněna. Pokud by tedy paměť různým skupinám byla přidělovaná z různých částí paměti, je jisté, že s ukončením odpovídajícího požadavku budou všechny bloky z odpovídající části uvolněny.

Toto pravidlo by bylo možné použít pro snížení fragmentace volné paměti. Na druhou stranu naprostá separace paměti jednotlivých požadavků přináší jisté nevýhody. Zřejmě největší je ta, že optimalizace popsaná v 7.1.1 by musela být omezená nejen na stejné vlákno, ale i na stejný požadavek. Toto by způsobovalo další její omezení. Další nevýhoda plyne z toho, že není snadné zcela oddělit jednotlivé požadavky, pokud není dopředu známé, kolik paměti budou potřebovat. Takže paměť by se musela alokovat po určitých superblocích (stejné velikosti, aby se zachovala výhoda, že se paměť nebude fragmentovat), a přidělovat paměť z nich. To ale přináší další nevýhodu, neboť požadavky na paměť větší než jeden superblok se musí řešit zvlášť. Navíc pokud by požadavek naalokoval jediný objekt v superbloku a tento objekt držel až do skončení požadavku, nemohl by tento superblok být použit jiným požadavkem, což by způsobilo zbytečné plýtvání pamětí.

Využití prvních dvou bodů pro snížení fragmentace paměti tedy přináší více nevýhod než výhod, takže nový alokátor při požadavku o alokaci mezi požadavky nijak nerozlišuje.

Třetí zmíněná vlastnost je poměrně důležitá při zohledňování faktoru vícevláknového prostředí zmíněného v části 7.1.2. Je totiž možné vytvořit přesný počet alokátorů podle počtu vláken a není třeba řešit potíže související s dynamickým vytvářením a rušením vláken během chodu systému. Dále je možné např. vytvořit veřejný seznam uvolněných objektů[2] pro každé vlákno, čímž se dále snižuje množství vláken, které sdílí určitou datovou strukturu, která se tak s menší pravděpodobností stane úzkým hrdlem.

Čtvrtá vlastnost umožňuje snížit velikost potřebné paměti na správu volných a obsazených úseků v paměti. Pokud by alokátor nedostával informace o velikosti uvolňovaných objektů, bylo by nutné implementovat mechanismus, který z adresy objektu určí jeho velikost, což může přinášet určité komplikace, případně zvýšenou paměťovou režii.

Další výhoda, kterou čtvrtá vlastnost přináší, je, že je možné zvolit naprosto rozdílné a navzájem zcela nezávislé strategie pro alokaci objektů určitých velikostí. Při uvolnění objektu pak není nutné jednotným způsobem zjišťovat jeho velikost, aby se požadavek mohl předat části zodpovědné pro zpracování bloků této velikosti.

Pátá vlastnost pak říká, že je možné navrhovat např. takové struktury, kdy je možné např. některé operace odkládat a až nastane příhodná situace, bude je možné zpracovat všechny najednou. Takové zdržení se sice projeví chvilkovým pozastavením vlákna, ale v celkovém čase takový způsob zpracování může vyjít lépe. Navíc je možné takto odložené operace provádět ve chvílích, kdy vlákno nemá nic jiného na práci, a tak případně zvýšit výkon.

[2] V tom smyslu, který byl uveden v části 2.1.2, která popisovala alokátor v TBB.

Kapitola 8

Implementace alokátoru

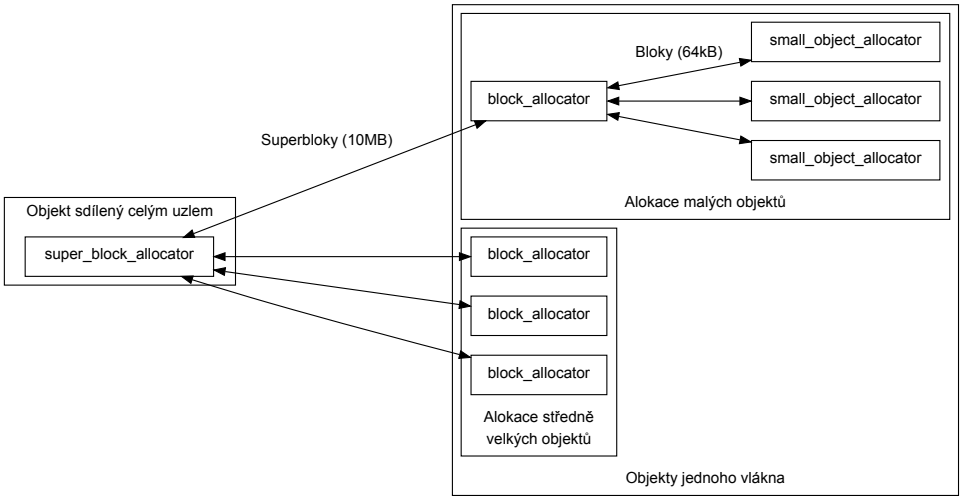
Podle velikosti požadovaného objektu alokátor používá následující tři alokační strategie:

- Objekty do velikosti 8kB nebo 16kB – takové požadavky zpracovávají tzv. alokátory malých objektů (viz následující část). Platí, že pro 32b systémy se používá hranice 8kB a pro 64b systémy 16kB. Tento rozdíl je poměrně žádoucí, neboť obecně platí, že stejné třídy jsou v 64b systému o něco větší než v systému 32b. V obou systémech tak bude přibližně stejná množina objektů alokována stejným způsobem.
- Objekty do velikosti 512kB – tyto požadavky jsou zpracovávány tzv. alokátory bloků (viz část 8.1.2). Horní hranice této velikosti je zvolena s ohledem na velikost obálek i s ohledem na úsporu paměti. Detailní informace jsou uvedeny v části 8.2.
- Objekty větší než 512kB – takové požadavky jsou vyřizovány jednoduše voláním funkcí pro správu virtuální paměti. Výskyt takových alokací by měl být velmi ojedinělý (viz část 8.4), takže není nutné mít pro tyto požadavky zvláštní strategii.

Z tohoto rozdělení pak vyplývá celková architektura alokátoru, která je naznačena na obrázku 8.1. Je na něm vidět uspořádání tříd a jejich vzájemná komunikace.

8.1 Alokace malých objektů

Alokátor malých objektů slouží, jak je již patrné z jeho názvu, pro alokaci objektů, jejichž velikost je omezená 8kB v případě 32b systému, resp. 16kB v případě 64b systému. Důvod, proč jsou malé objekty řešeny zvláštním alokátorem, je ten, že se předpokládá vysoké množství požadavků na alokaci a uvolnění objektů s touto velikostí. Tento předpoklad vychází z toho, že všechny třídy resp. struktury nemívají velikost větší než uvedená horní hranice, a pokud má třída uchovávat větší množství dat, alokuje pro ně zpravidla paměť dynamicky a nikoliv staticky, takže velikost těchto dat se neprojeví při alokaci paměti pro její instanci. Samozřejmě, že do tohoto



Obrázek 8.1: Architektura alokátoru

intervalu se vejdou i menší pole, s nimiž je práce zpravidla rovněž intenzivnější (z pohledu alokací a uvolňování paměti) než s velkými.

Navíc právě u takto malých objektů je vliv vyrovnávacích pamětí výraznější, a to ze dvou důvodu:

- Jejich velikost dostatečně malá na to, aby se celé vešly do vyrovnávací paměti, takže má smysl snažit se uvolněnou instanci co nejdříve recyklovat.
- Lze předpokládat, že před uvolněním objektu se s ním pracovalo (minimálně destruktor objektu k jeho datům mohl přistupovat), takže při uvolnění bude pravděpodobně ve vyrovnávací paměti uložen.

Cílem tohoto alokátoru pro malé objekty je tedy poskytovat co nejvyšší výkon s ohledem na vyrovnávací paměť procesoru, tzn. přidělovat pokud možno takové objekty, které byly nedávno uvolněny.

Protože celkový počet možných různých velikostí těchto objektů není vysoký, je možné vytvořit jakési přihrádky pro alokaci objektů určité velikosti a při alokaci objektu přiřadit paměť z přihrádky pro nejbližší vyšší nebo stejnou velikost.

Alokace pro malé objekty funguje právě tímto způsobem. Existují totiž přihrádky pro určité velikosti, přičemž každá přihrádka je spravována jednou instancí třídy `small_object_allocator`. Tato třída umí přidělovat a uvolňovat pouze objekty stejné, v konstruktoru zadané, velikosti. Navíc platí, že každé vlákno obsahuje vlastní množinu přihrádek, takže není nutné při alokaci řešit synchronizaci přístupu k alokátoru.

Velikost přihrádek neroste zcela rovnoměrně, ale podle funkce $1,07^i$ s tím, že velikosti jsou zaokrouhleny tak, aby byly vždy dělitelné minimální velikostí objektu. Tato velikost je součtem velikosti jednoho ukazatele a velikosti typu `size_t`, což vychází 8B pro 32b architekturu a 16B pro 64b architekturu. Číslo $1,07$ bylo zvoleno tak, aby se minimalizovala vnitřní fragmentace (každý objekt zabere maximálně o 7%

místa více, než kolik skutečně potřebuje) a zároveň minimalizovaly paměťové nároky alokátoru, neboť každá instance třídy `small_object_allocator` potřebuje paměť pro své interní struktury (viz část 8.1.1).

Při alokaci objektu tedy není nutné řešit synchronizaci, neboť každé vlákno má vlastní sadu přihrádek. Toto ale neplatí při uvolňování objektů, neboť objekt může být uvolněn z vlákna, v němž nebyl naalokovaný. Nelze tedy bez synchronizace zavolat metodu alokátoru v příslušné přihrádce. Tento problém je řešen tak, že objekty vracené nevlastnícím vláknům jsou pouze připojovány do veřejného spojového seznamu v alokátoru vlastnícího vlákna. Tento spojový seznam je synchronizovaný, takže konkureční přístup k němu nevadí. Vlastnící vlákno pak z tohoto seznamu volné objekty odebírá a uvolňuje, takže v tomto případě není skutečně nutné vůbec nijak synchronizovat přístup do alokátorů v přihrádkách.

Dále je využito bodů 3 a 5 z části 7.3. Bod 3 je využit tak, že seznam není jeden sdílený všemi ostatními vlákny, ale je vytvořen seznam pro každé vlákno (to je snadné proto, že je v době konstrukce objektu známý jejich počet) a každý takový je synchronizovaný vlastním primitivem. To samozřejmě zvyšuje efektivitu, neboť vlákno na takový zámek nemusí nijak čekat, neboť je určen přímo pro něj. Důvod proč seznamy musí být stále synchronizované i v tomto případě je ten, že stále do nich musí přistupovat i vlastnící vlákno, aby mohlo objekty z něj skutečně odalokovat.

Dle bodu 5 je možné tyto seznamy vyprázdnit pouze jednou za čas a objekty skutečně vrátit s tím, že případné zdržení nevadí. Tato operace se provádí tehdy, kdy celková velikost objektů ve veřejných seznamech je větší než 1MB, nebo těsně předtím než vlákno požádá o uspání (viz část 6.2), kdy stejně nemá žádnou jinou úlohu ke zpracování. Hranice 1MB je zvolena tak, aby se čištění seznamů neprovádělo zbytečně často, ale zároveň aby se v seznamech zbytečně nehromadila již uvolněná paměť. Kontrola, zda velikost nepřekročila tuto hranici je provedena vždy při každé alokaci případě uvolnění.

Objekty z těchto seznamů jsou uvolňovány trochu jiným způsobem než objekty vrácené vlastnícím vláknem. Platí totiž, že takové objekty nejsou uloženy v cache procesoru a zablokovaly by tak objekty, které byly vráceny vlastnícím vláknem a které mají šanci, že ve vyrovnávací paměti procesoru jsou. Tato úprava zvyšuje výkon alokátoru zhruba o 1-2%.

Rovněž bylo testováno chování, kdy objekty odalokované ve vláknech, které s vlastnícím vláknem sdílí cache, byly vraceny stejným způsobem jako v případě objektů vrácené vlastnícím vláknem. Tato úprava ale nepatrně výkon snížila, neboť takové objekty nejsou vraceny ihned a pravděpodobně v době vrácení již v cache uložené nejsou.

8.1.1 Alokátor malých objektů

V tuto chvíli je zřejmé, že samotný alokátor pouze vytváří v jednotlivých velikostních přihrádkách instance třídy `small_object_allocator` a s veškerými požadavky se obrací na instanci zodpovědnou za požadovanou velikost. Navíc platí, že tyto požadavky jsou již jednovláknové (o pročištění veřejných seznamů volných objektů se stará správce přihrádek), takže v těchto alokátorech již není nutné nijak řešit

synchronizaci.

Ve stručnosti funguje tento alokátor tak, že přialokovává 64kB bloky paměti, tyto bloky štěpí na objekty požadované velikosti (každý alokátor alokuje objekty stejné velikosti) a tyto bloky nejsou mezi stejnými alokátory sdílené. Zároveň si udržuje seznam vrácených objektů a při požadavku se nejdříve snaží vzít objekt z tohoto seznamu. Pokud je prázdný, tak naalokuje další blok (viz část o alokátoru bloků 8.1.2). Aby se alokátor choval dobře z hlediska vyrovnávacích pamětí, jsou objekty do tohoto seznamu vkládány a odebírány metodou LIFO, jedná se tedy o zásobník.

Tento postup ale přináší mnoho technických komplikací, které je nutné vyřešit. Téměř všechny tyto komplikace vyplývají z toho, že je třeba zajistit nejen efektivní alokování objektů, ale že je třeba vracet volné bloky zpět systému, resp. do alokátoru bloků.

Hledání vlastnícího bloku

První problém, který je nutný vyřešit, je detekce toho, že se blok stal zcela volným. Lze samozřejmě u každého bloku počítat, kolik objektů v něm je volných, a pokud je toto číslo rovné celkovému počtu objektů v bloku, tak jej uvolnit. Problém je ale s tím, že při uvolnění objektu dostane alokátor pouze ukazatel na začátek tohoto objektu. Z něj ale nelze jednoduchým způsobem odvodit adresu vlastnícího bloku, takže inkrementaci čítače nelze provést.

Jedním z možných řešení by mohlo být to, že před každým objektem bude vždy ukazatel na začátek vlastnícího bloku. To ale přináší jistou paměťovou režii a ztratila by se tak výhoda plynoucí z bodu 5 v části 7.3. Další řešení, které rovněž používá i alokátor v TBB, spočívá v tom, že všechny začátky bloků budou zarovnány na adresy, které jsou násobkem jeho velikosti. Protože všechny bloky jsou stejně velké, lze velmi snadno z adresy objektu spočítat adresu vlastnícího bloku a dále s ním provádět patřičné operace. Toto zarovnávání řeší již alokátor bloků, takže alokátor malých objektů se jím nemusí zabývat.

Uvolnění prázdného bloku

Když už se zdetekuje skutečnost, že se nějaký blok uvolnil, nastává další komplikace, neboť je nutné odstranit všechny jeho objekty ze zásobníku volných objektů. To ale není vůbec jednoduchá operace, neboť zásobník takovou operaci nepodporuje. Existují dvě přímočará řešení tohoto problému. Buď celý zásobník projít a odstranit z něj tyto bloky (to může být značně neefektivní), nebo udržovat zásobník jako dvojsměrný spojový seznam, projít všechny objekty v bloku a ze zásobníku odstranit. Druhá možnost je efektivnější, ale v případě objektů velikosti 16B by to obnášelo projít zhruba 4000 objektů a všechny postupně uvolnit.

Je tedy zřejmé, že jednoduchý zásobník použít nelze. Navíc měření ukázala, že uvolnění bloku ze zásobníku má nepříznivý vliv na výkonnost alokátoru. K uvolnění bloku totiž dochází právě tehdy, když do něj byl vrácen poslední objekt. Jak bude zdůvodněno dále, bývají zpravidla najednou vráceny celé skupiny objektů, takže vrácením takového bloku se způsobí, že případná skupina objektů, která byla blízko

vrcholu zásobníku a mohla tak být ve vyrovnávací paměti, se dále nepoužije a následné alokace bude vracet objekty z jiného bloku, hlouběji v zásobníku, které v cache již být nemusí.

Vysoká spotřeba paměti

Poslední velká potíž je způsobena tím, že pokud se volné bloky přidělují výlučně metodou LIFO, tak může docházet k plýtvání pamětí, neboť může existovat mnoho bloků, které jsou téměř prázdné a přitom při jiné strategii přidělování objektů by mohly být již dávno uvolněné. Extrémní situace může vypadat tak, že v bloku budou používané pouze dva objekty. Když se uvolní jeden, je okamžitě naalokován někým jiným. Následně, když se uvolní druhý objekt, tak jej naalokuje opět někdo jiný. Dojde tedy k tomu, že tyto objekty nebudou současně volnými a jejich vlastnící blok nebude uvolněn. Tato situace může reálně nastat např. tak, že požadavek naalokuje vysoké množství objektů. Poté je začne postupně vracet s tím, že do tohoto procesu budou vstupovat jiné požadavky na alokaci těchto objektů. Dojde tedy k tomu, že mnoho bloků bude téměř volných a z každého bude naalokováno pouze několik objektů. Vůbec se tedy neprojeví skutečnost, že mnoho objektů již bylo vráceno a paměť zůstane nevyužita.

Aby nedocházelo k plýtvání paměti, je nutné přidělovat objekty jiným způsobem. Protože není dopředu jasné, jak přesně budou objekty vraceny, nabízí se jediná rozumná heuristika - přidělovat objekty vždy z nejzaplněnějšího bloku. Tím se v případě téměř volných bloků zvyšuje pravděpodobnost, že se stihnou dříve zcela uvolnit, než z nich bude přidělen další objekt. Tuto pravděpodobnost dále zvyšuje platnost bodu 2 z části 7.3, takže je jisté (samozřejmě jenom tehdy pokud je korektně implementováno uvolňování paměti alokované krabičkami a při instanciaci požadavku), že takový poloprázdný blok bude v konečném čase buď zcela uvolněn, nebo použit pro další alokace.

Bohužel postup popsaný v předešlém odstavci je zcela v rozporu s tím, aby objekty byly přidělovány metodou LIFO, neboť obě zmíněné strategie definují navzájem zcela jiné pořadí přidělování objektů.

Omezení velikosti zásobníku

Je tedy nutné najít kompromis mezi těmito přístupy. Jednou z možností je omezit velikost zásobníku. Pokud je v zásobníku dostatek objektů, tak přidělovat z něj. Pokud je prázdný, přidělovat z nejzaplněnějšího bloku. Tímto způsobem se zachovají výhody obou a maximální velikostí zásobníku lze regulovat poměr mezi nimi. Čím větší je totiž zásobník, tím více se uplatňuje vliv vyrovnávacích pamětí. Naopak čím je menší, tím více lze očekávat uvolňování volných bloků.

Otázkou je, jakým číslem by měla být velikost zásobníku omezena. Za rozumný horní odhad tohoto čísla lze považovat velikost cache procesoru vydělenou velikostí objektu. Pro větší čísla totiž použití zásobníku ztrácí význam. Toto číslo ale může být poměrně vysoké (řádově až 10^5 pro nejmenší bloky). Navíc je nutné počítat i s tak extrémní situací, že v zásobníku budou takové volné objekty, že každý objekt pochází

z jiného bloku. V praxi by to pak znamenalo, že na jeden objekt je alokovaný jeden blok.

Přílišným zmenšením zásobníku se sice situace částečně zachrání (může stále dojít k výše popsané situaci, ale takových objektů už bude určitě méně), ale je to na úkor výkonnosti alokátoru.

Zvolené řešení

Řešení, které je použito při implementaci je založeno na následujícím předpokladu: dva objekty stejné velikosti vrácené za sebou (mezi nimi mohou být vráceny i jiné objekty, ale musí být odlišné velikosti) pravděpodobně patří do stejného bloku. Teoreticky toto pravidlo vyplývá z toho, že pokud se najednou alokuje několik objektů, jsou tyto objekty pravděpodobně rovněž vráceny ve stejnou chvíli. Posloupnost objektů, které byly vráceny bezprostředně za sebou a patří do stejného bloku, bude dále označována jako tzv. *řetězec*.

V tabulce 8.1 je vidět závislost délky takových řetězců na velikosti objektu po spuštění velkého množství paralelních požadavků. To, že je délka řetězce menší u větších objektů je způsobeno tím, že takových objektů se do bloku vejde méně, a délka řetězce tedy nemůže být nikdy delší než velikost bloku dělená velikostí objektu. Měření bylo provedeno několikrát přičemž počet požadavků byl postupně zvyšován. Čísla v tabulce odpovídají situaci, kdy další zvýšení počtu požadavků již nemělo vliv na průměrnou délku řetězců. I přes to, že např. přihrádka pro objekty do velikosti 528B byla skutečně intenzivně používána, tak délka řetězce se ustálila na velikosti, která odpovídá zhruba 2kB. Ostatní velikosti buď nebyly vůbec alokovány, nebo se jednalo o alokace, kdy byla veškerá paměť alokována pouze z jednoho bloku (to se může snadno stát tehdy, kdy životnost objektů je velmi krátká), takže průměrnou délku řetězců u nich nemá smysl zkoumat.

Velikost objektu [B]	Počet alokací	Průměrná délka řetězce
48	2307737	17,4802
128	372653	9,00359
448	328220	12,8666
528	29543417	4,04333
5024	6136	4,70782
8112	4629	2,68503
10656	4564	4,61538
13072	8002	2,52087
16048	19163	2,03401

Tabulka 8.1: Průměrné délky řetězců

Této vlastnosti je pak využito pro jiný způsob omezení zásobníku vrácených objektů. Ten totiž není omezen počtem objektů, ale počtem takových řetězců obsažených v zásobníku. Platí důležité pravidlo, že počet bloků obsažených v zásobníku nemůže být nikdy vyšší než počet řetězců v něm (ve skutečnosti bývá menší, neboť jeden blok může mít v zásobníku i několik řetězců), omezuje tento způsob jednoznačně

maximální počet bloků v zásobníku. Maximální povolený počet řetězců v zásobníku navíc může být poměrně nízký, neboť skutečný počet objektů v zásobníku je podle výše zmíněné vlastnosti mnohem větší.

Algoritmus tedy funguje tak, že pokud se vrací objekt, který pochází ze stejného bloku, ze kterého pochází objekt ve vrcholu zásobníku, je prodloužen řetězec na vrcholu zásobníku o tento objekt. Pokud pochází z bloku jiného, stane se počátkem nově vznikajícího řetězce a pokud zásobník začne obsahovat více, než povolený počet řetězců, je nejstarší řetězec ze zásobníku odstraněn.

Pokud je takto odstraněn ze zásobníku blok, který v něm již nemá žádný řetězec, provede se jedna z následujících akcí (blok nemůže být zcela zaplněný, jinak by nemohl být v zásobníku):

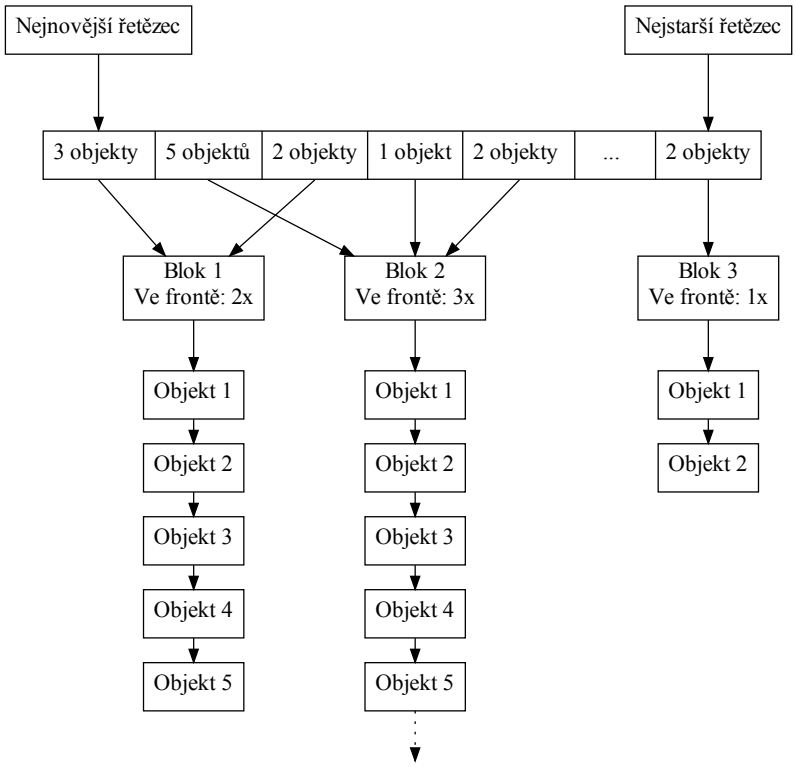
- Zcela prázdný blok se vrátí alokátoru bloků.
- Více než z poloviny prázdný blok se vloží do seznamu téměř prázdných bloků.
- Ostatní bloky se vkládají do seznamu téměř plných bloků.

Bloky nejsou udržované setříděné podle zaplněnosti, ale jsou udržované v několika seznamech. Tyto seznamy jsou čtyři - seznam zcela zaplněných bloků, seznam téměř plných bloků (víc než polovina obsazených objektů), seznam téměř prázdných bloků (méně než polovina obsazených bloků) a seznam bloků zastoupených v zásobníku.

To, že jsou bloky takto rozděleny, hraje důležitou roli tehdy, kdy je požadován další objekt, ale zásobník je prázdný. Prioritně se totiž vezme téměř zaplněný blok, a všechny volné objekty v něm jsou vloženy jako nový řetězec na vrchol zásobníku. To, že se vždy nebere striktně nejzaplněnější blok, způsobuje, že takto vytvořený řetězec nebude pravděpodobně úplně krátký. Důsledkem toho je, že se řetězce při uvolňování objektů vytvářejí i poté, co je s alokátorem intenzivně pracováno. Navíc lze téměř prázdné bloky takto nechat stranou čekat, až se buď celé uvolní, nebo budou použity pro alokaci. Tím je simulována strategie popsaná v části popisující vysokou spotřebu paměti.

Implementovat popsanou datovou strukturu je možné několika způsoby. Zásobník může být implementován např. pomocí obousměrného spojového seznamu. Je pak velmi snadné odstranit nejstarší řetězec - postupně se uvolňují objekty od konce, dokud se nezmění jejich vlastnící blok. Tento postup je ale mírně neefektivní, neboť se skutečně musí projít částí spojového seznamu, což prakticky znamená přistupovat na místa v paměti, která už pravděpodobně nejsou ve vyrovnávací paměti (jsou nejstarší v zásobníku). Navíc složitost takové operace není konstantní, ale lineární vzhledem k délce řetězce.

Byla proto zvolena jiná implementace – pomocí statické obousměrné fronty řetězců. Do této fronty se přidává pouze na konec. Pokud obsahuje příliš mnoho řetězců, jsou odstraňovány ze začátku. Pokud je požadován objekt, je vrácen nejmladší objekt z řetězce na konci fronty. Každý blok si udržuje vlastní jednosměrný spojový seznam (zásobník) volných bloků, který nijak se samotnými řetězci nesouvisí. Jedna položka fronty pak odpovídá jednomu řetězci a obsahuje ukazatel na blok, kterému



Obrázek 8.2: Zásobník vrácených objektů

řetězec odpovídá, a dále počet objektů v řetězci, což počtu objektů ze zásobníku volných objektů v bloku, které jsou součástí tohoto řetězce.

Přidání objektu do nejmladšího řetězce tak obnáší přidat objekt do seznamu v bloku a inkrementovat počet objektů v řetězci, který je uložen v nejmladším prvku fronty. Pro přidání nového řetězce je nutné vytvořit nový záznam na konci fronty. Odebrání objektu znamená pouhou dekrementaci čísla v nejnovějším záznamu fronty a případné odstranění tohoto záznamu (pokud se řetězec vyčerpal). Celou situaci znázorňuje obrázek 8.2.

Dále si každý blok pamatuje, kolikrát je zastoupen ve frontě, takže je možné snadno určit, kdy se blok dostal z fronty, a je možné jej zařadit do jednoho ze seznamů, jak bylo zmíněno výše. Vyšší efektivita toho postupu oproti obousměrnému spojovému seznamu spočívá v tom, že odstranění nejstaršího řetězce obnáší pouze odstranění nejstaršího záznamu z fronty, dekrementace počtu řetězců bloku ve frontě a případné přemístění bloku do vybraného seznamu. V každém případě se jedná o operaci s konstantním časovou složitostí.

Pokud je fronta prázdná, je odebrán blok buď z téměř zaplněného seznamu, případně z téměř prázdného a pokud i tento seznam je prázdný, je vytvořen blok nový. V každém případě je ze všech volných objektů v tomto bloku vytvořen jeden řetězec,

který je vložen na konec fronty a alokace pak může probíhat jako obvykle.

Zbývá zmínit, jakým číslem je počet řetězců ve frontě omezen. Toto číslo totiž není konstantní, ale mění se podle počtu naalokovaných objektů. Je udržováno tak, aby maximální počet řetězců byl roven dvojnásobku minimálního počtu bloků, které jsou potřeba pro všechny alokované objekty. Tj.

$$\frac{2 \times \text{celkový počet alokovaných objektů}}{\text{počet objektů v jednom bloku}}$$

To zaručí, že v zásobníku bude maximálně 50% volného místa blokováno. Toto případné místo navíc se samozřejmě nikam neztrácí a bude použito pro další alokace. Hlavní důsledek je spíše ten, že i kdyby byl použit pouze jeden objekt z každého bloku, pak případná paměť, která zůstane nevyužitá, není příliš velká.

Zároveň je toto číslo vždy udržováno v intervalu $< 16,64\text{kB}/\text{velikost objektu} >$. Hranice 64kB byla zvolena jako kompromis mezi počtem objektů, které se ukládají do fronty, a paměťovými nároky jednoho alokátoru, neboť fronta je alokována vždy při vytvoření alokátoru a toto místo zabírá až do destrukce alokátoru.

Jeden alokátor tak pro jednu frontu zabere

$$\frac{64\text{kB}}{\text{velikost objektu}} * (\texttt{sizeof(size\_t)} + \texttt{sizeof(void*)})$$

, což dohromady pro všechny alokátory malých objektů jednoho vlákna vychází méně než 2MB, což je přijatelná paměťová režie.

Vracení objektů uvolněných v jiných vláknech

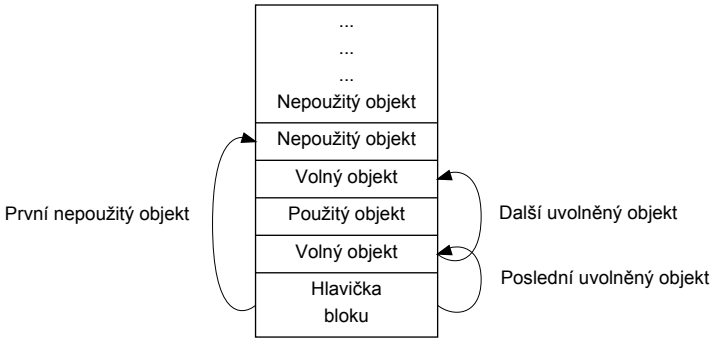
Jak bylo uvedeno již v části 8.1, objekty uvolněné v jiných vláknech jsou vraceny tak, že se pouze obchází ukládání uvolněných objektů do fronty/zásobníku. Objekt je přímo vrácen do seznamu volných objektů vlastnícího bloku a je pouze zkontrolováno, zda se tím nestal zcela nebo téměř prázdným.

Štěpení bloků na objekty

Seznam volným objektů v bloku je ve skutečnosti udržován dvojím způsobem. První již zmíněný způsob je ten, že součástí bloku je spojový seznam všech jeho volných objektů. Aby ale nebylo nutné při alokaci nového bloku vytvářet vždy celý spojový seznam, který by obsahoval všechny volné objekty, je tento způsob doplněn o druhý. Každý blok tak navíc obsahuje adresu prvního volného objektu, který ještě nikdy nebyl přidělen. V případě požadavku na další objekt se nejdříve berou objekty ze spojového seznamu. Pokud je seznam prázdný, je použit onen první, který ještě nebyl přidělen, a adresa je posunuta na další objekt. Obrázek 8.3 zobrazuje zjednodušenou vnitřní strukturu jednoho bloku.

8.1.2 Alokátor bloků

Alokaci bloků má na starosti třída `block_allocator`. Tato třída poskytuje pouze dvě metody. Jedna pro alokaci bloku a druhá pro jeho uvolnění. Alokace probíhá tak,



Obrázek 8.3: Vnitřní struktura bloku

že jsou od třídy `super_block_allocator` požadovány superbloky o velikosti 10MB. Z těchto superbloků jsou pak přidělovány jednotlivé bloky. Implementace této třídy je optimalizovaná pouze na maximální úsporu paměti alokované přímo od operačního systému. Při navrácení bloku se ihned kontroluje, zda se některý ze superbloků nestal volným a v případě, že ano, je vrácen do alokátoru superbloků.

Toto chování se může jevit jako nevýhodné, neboť přidělení a uvolnění superbloku je časově náročná operace (paměť je nutné nastránkovat, viz 7.1.4). V extrémním případě pak může dojít ke střídavému přidělování a uvolňování bloku, které se bude projevovat střídavým přidělováním a uvolňováním superbloku. Jistě by tedy bylo vhodné mít vyrovnávací paměť na uvolněné bloky a při případném požadavku vracet prioritně blok z této paměti, aby se omezilo výše zmíněné nežádoucí chování.

Zodpovědnost za tuto činnost je ale přenechána klientům této třídy. Ve finální implementaci toto zajišťuje třída `cached_block_allocator`, což je dekorátor této třídy, který zmíněnou logiku implementuje.

Alokátor tedy přímo od operačního systému získává superbloky a ty dále štěpí na jednotlivé bloky. Aby se maximálně snížilo množství superbloků, jsou bloky přidělovány vždy z co nejvíce zaplněného superbloku. To znamená, že méně zaplněné superbloky čekají na to, až se buď zcela uvolní, nebo na to, až nebude k dispozici blok, který by byl více zaplněný.

Klíčovou datovou strukturou pro zajištění zmíněného pravidla je binární minimální halda, jejímiž prvky jsou superbloky a klíčem je množství volných bloků v něm. Ve vrcholu haldy se tak vždy nachází nejvíce zaplněný superblok, neboť má nejméně volných bloků. Superbloky, které už nemají žádný volný blok, v haldě umístěné nejsou.

Při každé alokaci, resp. uvolnění, je tato halda aktualizována. Tato aktualizace ale není příliš složitá. V případě alokace platí, že počet volných bloků v superbloku, který je umístěn ve vrcholu haldy (z něj byla alokace provedena), se dále sníží. Takže superblok blok zůstává ve vrcholu haldy. Pokud již superblok po této operaci žádný další volný blok neobsahuje, je vrchol z haldy odstraněn a halda je zmenšena o jeden superblok. Do vrcholu se tak dostane další nejvíce zaplněný superblok. Pokud je halda zcela prázdná, je naalokován další superblok.

V případě uvolnění bloku, je nalezen příslušný superblok. To lze provést jednoduše, neboť každý blok si v hlavičce pamatuje ukazatel na vlastnící superblok. Pokud

se nachází v haldě (tj. obsahoval alespoň jeden volný blok), je halda opravena tak, aby stále byla haldou. Tato oprava znamená ve většině případů pouze konstantní množství práce, neboť hodnota klíče se zvýší pouze o jedničku. Pokud byl superblok zcela zaplněný, je po uvolnění bloku přidán do haldy.

Použití haldy zaručuje, že každá operace bude trvat maximálně $O(\log(n))$ operací, kde n je počet částečně zaplněných superbloků. Jak ale bylo zmíněno, většina operací bude trvat konstantní množství operací. Střídavé odstraňování a vracení superbloku do vrcholu haldy (tyto operace trvají $O(\log(n))$ vždy) je redukováno vyrovnávacími paměťmi klientů třídy.

Protože tento algoritmus vyžaduje rozšířenou sadu operací nad haldou, než které poskytuje třída `std::priority_queue`, jako např. aktualizaci záznamu uprostřed haldy (při uvolnění bloku), je použita vlastní implementace. Každý superblok si pamatuje svoji pozici v haldě, takže taková aktualizace je velmi rychlá. Jediná komplikace je, že tyto pozice superbloků je nutné udržovat upravovat při změnách haldy. To je ale pouze technický detail.

Platí, že bloky poskytované tímto alokátorem mohou volitelně začínat vždy na adrese, která je násobkem velikosti bloku. Toho dále využívá alokátor malých objektů.

Seznam volných bloků v rámci superbloku je udržován stejným způsobem jako seznam volných objektů v rámci bloku. Vytvoření nového superbloku tak trvá rovněž konstantní množství práce.

8.1.3 Vyrovnávací paměť pro bloky

Třída `cached_blok_allocator` slouží jako dekorátor třídy pro alokaci bloků a přidává funkčnost pro cacheování bloků. Bohužel v této souvislosti se objevují podobné problémy jako v případě alokátorů malých objektů. Pokud by se totiž uchovávalo vždy určité množství bloků, mohlo by se snadno stát, že každý blok z této vyrovnávací paměti bude odpovídat jinému superbloku. Ve výsledku by to znamenalo, že veškeré optimalizace na vracení paměti by mohly být nefunkční, a ačkoliv bloky by byly vraceny správně, tak superbloky by byly blokované právě v této vyrovnávací paměti.

Proto není velikost vyrovnávací paměti omezená počtem bloků, ale počtem superbloků v ní. Využívá se tak podobného jevu, jako je tvoření řetězců vracených objektů v případě alokátoru malých objektů.

Pokud je vrácen blok, který připadá superbloku, který již je ve vyrovnávací paměti, je tento blok uchován. Pokud nikoliv, ale počet superbloků ještě nedosáhl maxima, je počet superbloků zvýšen o jedna, a blok je uchován. V opačném případě je blok vrácen přímo alokátoru bloků.

V případě alokace se nejdříve vrací blok z vyrovnávací paměti a teprve, když je tato paměť prázdná, je zavolána příslušná metoda na instanci alokátoru bloků.

Při alokaci bloků pro alokátor malých objektů se takto uchovávají bloky maximálně z 8 superbloků. Limit je zvolen takto nízký proto, aby ani v nejhorším případě v tomto alokátoru nezůstávalo více, než 80MB paměti. Navíc i pro superbloky existuje vyrovnávací paměť, takže tato vyrovnávací paměť nemusí být příliš veliká.

8.2 Alokace středně velkých objektů

Do této kategorie spadají všechny alokace objektů od horní hranice pro použití strategie pro alokaci malých objektů do velikosti 512kB. Vzhledem k doporučené velikosti obálky, která je maximálně 200kB (viz část 6.6), by do tohoto intervalu měly spadnout veškeré alokace při vytváření obálek.

Zároveň se jedná o poměrně velké objekty, u kterých již nemá velký význam brát v úvahu vliv vyrovnávacích pamětí. Začíná se u nich ale uplatňovat vliv stránkování.

Tento fakt značně usnadňuje situaci, neboť není nutné zajišťovat, aby nejpozději odalokovaný objekt byl opět přidělen co nejdříve, takže je možné optimalizovat pouze na úsporu paměti a není nutné hledat žádný kompromis. Algoritmus pro uvolňování a přidělování objektů tak může být zcela stejný jako v případě alokace bloků.

Právě toho je při implementaci využito a pro alokaci středně velkých objektů se používá právě alokátor bloků. Protože tento alokátor umí přidělovat pouze bloky stejné velikosti, je použit stejný postup jako v případě alokací malých objektů. Existují rovněž přihrádky, které řeší alokaci a uvolňování objektů pouze těch velikostí, za které je přihrádka zodpovědná. Jediná změna v chování alokátoru bloků je ta, že přidělované bloky nejsou zarovnávané na adresu dělitelnou jeho velikostí. Tato úprava snižuje vnitřní fragmentaci superbloku. Dále nejsou volány přímo metody alokátoru bloků, ale je využita třída `cached_block_allocator`, ve které se uchová maximálně 1 superblok.

Velikost těchto přihrádek rovněž neroste rovnoměrně, ale stejně jako v případě přihrádek pro alokátory malých bloků, roste stejně rychle jako funkce $1,07^i$. Minimální velikost přihrádek je 512B a každá velikost je dělitelná tímto číslem. Velikost vnitřní fragmentace tedy není nikdy vyšší než 10%.

Horní limit pro alokaci středně velkých objektů je ve skutečnosti o něco menší než 512kB (přesná hranice je 510kB), aby se nejednalo o číslo které dělí velikost superbloku, která je 10MB. Protože super blok obsahuje určité množství pomocných informací, při velikosti přesně 512kB by zůstal vždy prakticky celý jeden blok nevyužitý.

8.3 Alokace superbloků

Alokaci superbloků má na starosti třída `super_block_allocator`. Tato třída je jedinou třídou v paměťovém alokátoru, která je sdílená více vlákny, takže přístup k ní musí být synchronizovaný. Na druhou stranu alokace superbloků by měly nastávat s poměrně nízkou frekvencí (z jednoho superbloku je přiděleno velké množství objektů), takže by se tento alokátor neměl stát úzkým hrdlem. Navíc není sdílený všemi vlákny, ale pouze vlákny v rámci jednoho uzlu NUMA systému, což dále snižuje množství vláken, které sdílí jeden zámek.

Důvod, proč je instance sdílená, je ten, že se nestará pouze o alokaci a uvolňování superbloků, ale i o jejich cacheování. Využívá se tak poznatku získaného v 7.1.4, že alokace velkého bloku paměti a první přístup k takovému bloku jsou poměrně pomalé operace. Vyplácí se tedy takové bloky hned neuvolňovat, ale uchovávat pro

budoucí použití. Skutečnost, že všechny alokované superbloky jsou stejné velikosti správu cache značně usnadňuje. Dále je zaručeno, že všechny alokované i vrácené superbloky odpovídají jednomu uzlu, takže není nutné nijak řešit otázku vlastnictví při jejich zařazování do vyrovnávací paměti.

Superbloky jsou přidělovány systémovými funkcemi pro správu virtuální paměti s podporou NUMA systémů. Je tedy možné alokovat superblok tak, aby bylo jisté, že přidělená paměť bude patřit danému uzlu a nemohlo se stát, že časem se paměť jednotlivých uzlů promíchá, jak se to může stát v případě použití funkce `malloc`.

Velikost vyrovnávací paměti superbloků je nastavena tak, aby byla vždy pokud možno volná jedna čtvrtina dostupné paměti uzlu. Počítá se celkové množství paměti alokované alokátorem a pokud by mělo toto množství přesáhnout 75% dostupné paměti, jsou již bloky přímo vraceny systému. Do této velikosti se počítají i objekty alokované alokátorem velkých objektů. Důvod, proč je stále ponecháváno volné místo v paměti je ten, že je nutné zajistit dostatek prostoru standardnímu alokátor paměti pro případ, že jej krabičky používají pro alokaci paměti místo třídy `thr_allocator`.

8.4 Alokace velkých objektů

Pro objekty větší než 512kB se volají přímo funkce pro správu virtuální paměti. Při uvolnění objektu je paměť ihned uvolněna. Právě proto je možné volat funkce pro alokaci obecné paměti (`VirtualAlloc` a `VirtualFree` v operačním systému Windows, resp. `mmap` a `munmap`) a nikoliv pro alokaci paměti na daném uzlu. Hlavním důvodem pro toto rozhodnutí je, že funkce pro alokaci paměti v knihovně `libnuma` jsou pomalé [10] a navíc bylo experimentálně zjištěno, že v jednom okamžiku může být alokováno nejvýše zhruba 65500 bloků. Poté, bez ohledu na velikost zbývající paměti, začnou alokace selhávat. Toto nedokumentované chování bylo pozorované na všech testovácích počítačích s operačním systémem Linux.

Použití obecných funkcí pro alokaci a uvolnění virtuální paměti je ale v tomto případě v pořádku, neboť stránky jsou, jak již bylo zmíněno, mapovány na paměť toho uzlu, ze kterého na ní bylo poprvé přistoupeno. Při uvolnění je toto mapování zapomenuto, takže nedochází k tomu, že by tento objekt byl příště přidělený uzlu, kterému fyzická paměť objektu nepatří.

Dále platí, že pokud selže alokace virtuální paměti při alokaci objektu většího než 512kB, je alokátor superbloků požádán, aby se pokusil příslušnou velikost paměti uvolnit a alokace je poté ještě jednou zopakována.

Kapitola 9

Další možnosti urychlení

Ačkoliv plánovač i alokátor mají vliv na výkonnost systému, jeho efektivitu samozřejmě ovlivňují i jiné části. Jedná se zejména o samotnou implementaci operací prováděných v krabičkách.

Přestože cílem práce není měnit jejich implementaci, je v této kapitole popsána efektivnější implementace třídící krabičky, která vznikla (jak bylo zmíněno v kapitole 4) hlavně z důvodu snazšího testování různých plánovačů. Dále jsou zde shrnuty některé poznatky získané při experimentech během vývoje plánovače a alokátoru, které by mohly v budoucnu dále zvýšit efektivitu Boboxu.

9.1 Implementace třídící krabičky

Je zřejmé, že práce plánovače končí ve chvíli, kdy spustí krabičku s tím, že příchozí data jsou pokud možno celá ve vyrovnávací paměti. Nyní záleží pouze na krabičce, jak této skutečnosti využije. Jsou operace, které ze své vlastní podstaty této skutečnosti využívají – např. nezávislá operace nad každým příchozím řádkem nebo v určité míře např. krabičky, které provádí množinové operace nad příchozími setříděnými daty (sjednocení, rozdíl nebo průnik).

Na druhou stranu jsou krabičky, které toho nativně nevyužívají, a je třeba algoritmus trochu upravit tak, aby přítomnosti dat v cache využil. Třídící krabička je právě jedním z příkladů.

Původní implementace si jednoduše ukládá veškerá příchozí data do pomocné paměti, takže ve chvíli, kdy obdrží otrávenou obálku, může použít některý z klasických algoritmů na třídění řádků.

Otázkou tedy je, jak implementaci krabičky změnit tak, aby vliv vyrovnávacích pamětí procesoru zohledňovala. Je zřejmé, že správný postup je takový, který se bude snažit s daty pracovat ihned při příjmu obálky.

Jako první nápad se nabízí udržovat všechny dosud přijaté řádky setříděné a při příchodu další obálky její řádky setřídit a posléze zatřídit do setříděné posloupnosti dosud přijatých řádků. Bohužel takový algoritmus zvyšuje časovou složitost na kvadratickou vzhledem k celkovému počtu řádek. Tato složitost vyplývá z toho, že zatřídění n setříděných řádků mezi m jiných setříděných trvá $O(n + m)$.

Zpracování první obálky pak trvá $O(n+0)$, druhé $O(n+n)$, třetí $O(n+2n)$ atd. i-tá příchozí obálka tedy vyžaduje $O(in)$ a celková složitost je $O(n\sum_{i=1}^{k} i)$, kde k je celkový počet obálek. Suma vychází $O(nk(1+k)/2)$ tj. $O(nk^2)$. Protože počet řádků v obálkách je stále stejný, platí že $k = \ell/n$, kde ℓ je celkový počet řádek. Celková složitost tedy vychází $O(\ell^2/n + \ell\log(n))$, kde člen $\ell\log(n)$ odpovídá prvotnímu setřídění přijatých obálek.

Protože n je ve skutečnosti konstanta (obálky mají počet řádků shora omezený konstantou), je celková složitost $O(\ell^2)$.

Tímto způsobem tedy algoritmus urychlit nelze. Z analýzy časové složitosti ale vyplývá, že neefektivita se skrývá ve slévání dlouhé posloupnosti s krátkou. Takže kdyby se podařilo zachovat pravidlo, že se budou slévat pouze posloupnosti podobné (nebo ještě lépe stejné) délky, mohla by časová složitost vyjít lépe a zároveň by se zachovala původní myšlenka – data by byla zpracována ihned při příchodu.

Pokud má být ale splněna požadovaná podmínka, je zřejmé, že již nelze vystačit s pouze jednou setříděnou posloupností, ale je nutné udržovat celou množinu takových posloupností.

Protože sléváním dvou stejně dlouhých posloupností vzniká posloupnost dvojnásobné délky, nabízí se udržovat tuto množinu tak, aby všechny posloupnosti v ní měly délku 2^i. Zároveň v množině nebudou existovat dvě posloupnosti stejné délky.

Do takové množiny je velmi snadné zatřídit setříděnou posloupnost, jejíž délka je mocninou čísla 2. Při této operaci totiž mohou nastat pouze dva případy:

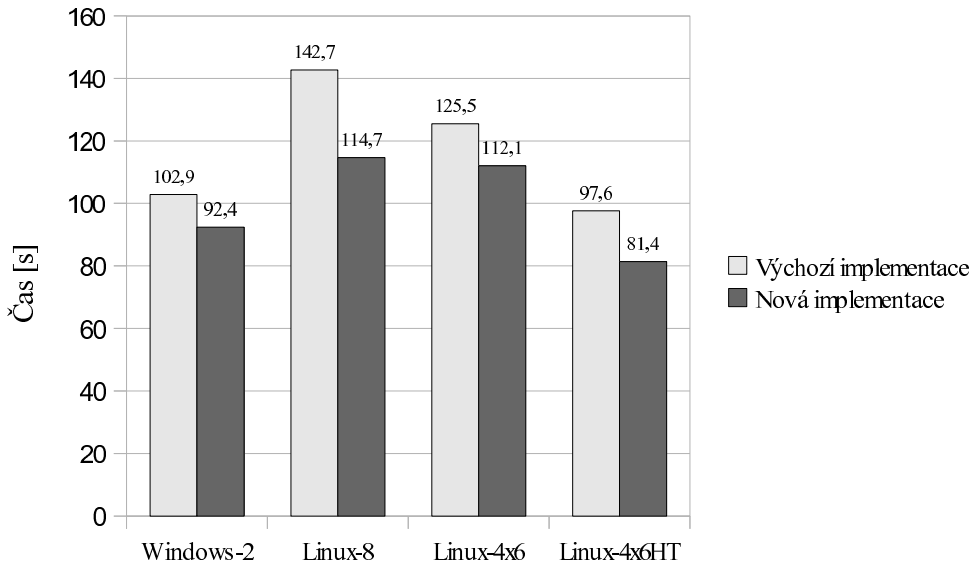
- V množině posloupnost takové délky ještě neexistuje – pak je situace jednoduchá, neboť stačí tuto posloupnost jednoduše přidat do množiny.
- V množině posloupnost se stejnou délkou již existuje – v tomto případě se obě posloupnosti slijí do jedné posloupnosti dvojnásobné délky a výsledná posloupnost se pak stejným postupem opět přidá do množiny.

Tento algoritmus je jistě konečný, neboť délka posloupnosti, která se takto rekurzivně vkládá do množiny, exponenciálně roste a zároveň je shora omezená celkovým počtem řádků.

Drobná potíž nastává ve chvíli, kdy obálka neobsahuje přesně 2^i řádků. Ta se ale snadno dá řešit, neboť platí, že každé nezáporné celé číslo lze vyjádřit součtem $\sum_{i=0}^{\infty} a_i 2^i$, kde a_i je buď nula, nebo jedna (tato operace odpovídá převodu čísla do binární soustavy). Takto lze tedy obálku rozdělit na několik posloupností, jejichž délka splňuje výše uvedené pravidlo, a tyto posloupnosti lze výše popsaným způsobem zatřídit do průběžně udržované množiny. Samozřejmě, že prvky v posloupnostech musí být setříděné, což lze provést snadno např. pomocí funkce `std::sort` – buď pro celou obálku před rozdělením, nebo pro jednotlivé posloupnosti.

Při příjmu otrávené obálky pak pouze zbývá slít všechny posloupnosti ve výsledné množině. Toto slévání je prováděno od nejkratších posloupností po nejdelší, aby bylo co nejvíce splněno pravidlo, že se slévají posloupnosti podobné délky.

Celý algoritmus je podobný sčítání binárních čísel, kde jednotlivé bity odpovídají posloupnostem a jejich váha odpovídá délce posloupností.



Obrázek 9.1: Účinnost nové implementace třídící krabičky

Odvodit časovou složitost je jednoduché, protože je možné snadno spočítat, kolikrát se každý prvek zúčastní operace slévání. Při každé této operaci se na každý prvek spotřebuje konstantní množství práce (porovnání a překopírování). Dále platí, že pokud se prvek zúčastní slévání, dostane se do posloupnosti dvojnásobné délky. Protože existuje maximálně $O(\log(\ell))$ posloupností, kde ℓ je opět celkový počet všech prvků, zúčastní se každý prvek maximálně $O(\log(\ell))$ operací slévání. Celková časová složitost tak je $O(\ell \log(\ell))$.

Popsaný algoritmus tedy zachovává výhody prvního navrhovaného (data jsou průběžně zpracovávána při jejich příjmu) a zároveň má optimální časovou složitost, neboť $O(\ell \log(\ell))$ je zároveň dolní odhad složitosti třídění.

Ačkoliv je tento postup skutečně jednoduchý, je z grafu 9.1 zřejmé, že nezanedbatelně zvyšuje rychlost zpracování požadavků. Graf vznikl tak, že byly dvakrát puštěny stejné požadavky – jednou, kdy se používala původní implementace a podruhé, kdy se použila implementace nová. Měření probíhala s výchozí implementací plánovače.

9.2 Další možné úpravy

Protože je systém Bobox na počátku svého vývoje, je pochopitelné, že existuje prostor pro jeho vylepšení. Během vzniku této práce byla nalezena některá místa, jejichž úpravou by mohl být výkon systému dále zvýšen. Ty, které souvisí s tématem této práce, jsou včetně návrhu jejich řešení stručně rozebrány v následujících částech této kapitoly.

9.2.1 Instanciace grafu jako úloha

V současnosti existuje jedno hlavní vlákno, které přijímá požadavky, grafy těchto požadavků instanciuje a postará se o zařazení iniciační krabičky do plánovače.

Instanciace grafu ale není časově nezanedbatelná operace, což se může projevit zvýšenou zátěží toho procesoru, na kterém je hlavní vlákno spuštěno. To může způsobit chvilkové snížení výkonu, neboť se musí po tuto dobu přerušit výpočetní vlákno, a tím se může změnit jeho obsah vyrovnávacích pamětí. Toto snížení výkonu ale pravděpodobně nebude vysoké.

Větší potíž ale nastává tím, že veškerá data tohoto požadavku jsou alokována na tom uzlu sytému NUMA, ve kterém běží hlavní vlákno. Pokud je tento uzel jiný, než na kterém nakonec bude požadavek spuštěn, může to dále způsobit výkonnostní propad, který pravděpodobně bude trochu citelnější.

Řešení problému by mohlo být např. takové, že instanciace požadavku by byla zapouzdřena jako samostatná úloha a taková úloha by se vložila do plánovače místo iniciační krabičky požadavku. Naplánování iniciační krabičky by proběhlo z této iniciální úlohy a plánovač by už sám automaticky zajistil, že ta bude spuštěná na správném uzlu vzhledem k tomu, na kterém byl požadavek vytvořen.

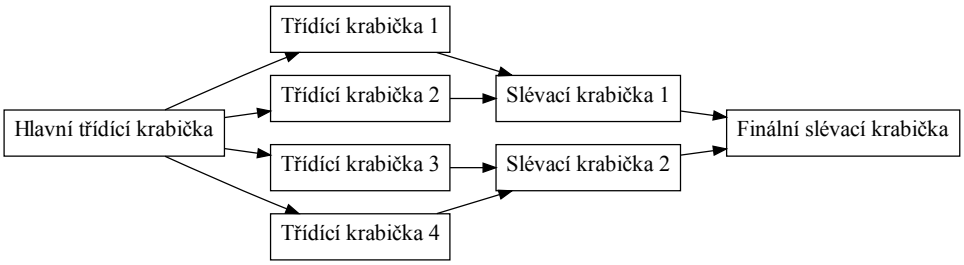
9.2.2 Paralelní spuštění jedné krabičky

Každá krabička může být v současné implementaci v jeden okamžik spuštěna nejvýše v jednom vlákně. To sice přináší usnadnění implementace některých krabiček, neboť není nutné se zabývat synchronizací, na druhou stranu to může obnášet výkonnostní omezení.

Při analýze optimální plánovací strategie pro lineární graf byl zmíněn fakt, že rozdílná rychlost zpracování krabiček způsobuje, že vlákno A, které právě ukončilo zpracování jedné krabičky může narazit na to, že následující krabička je stále zpracovávaná předchozím vláknem B. Vlákno A tak musí začít vykonávat jinou práci a nemůže využít toho, že má data uložená ve vyrovnávací paměti. Přitom existují operace, které nemusí být nutně zpracovávány sériově, resp. lze zpracování jednotlivých částí paralelizovat (ve smyslu funkce `parallel_reduce` uvedené v popisu technologie TBB (část 2.1) – tedy např. různé agregační funkce, odfiltrování určitých řádků atp.

Možným řešením by bylo explicitně rozlišovat mezi krabičkami, zda mohou být v jeden okamžik spuštěny nejvýše jednou nebo libovolněkrát. Tato úprava by nijak nevyžadovala úpravu plánovače, neboť ten spouští to, co bylo naplánováno, a maximální počet spuštění si řídí krabičky/úlohy samy (viz část 5.2). Na druhou stranu je nutné zajistit, aby, pokud záleží na pořadí výstupních dat, bylo toto zachováno.

Další možnost, jak tento problém řešit, je podobná jako v případě následující části, tj. úpravou grafu. Pravidlo, že jedna krabička může být spuštěna v jeden okamžik pouze jednou, by mohlo zůstat zachováno na úkor toho, že by takových krabiček bylo více s tím, že by před nimi byla krabička, která příchozí data pouze přepošle do takové krabičky, která zrovna neprovádí žádný výpočet.



Obrázek 9.2: Paralelizace štěpením krabiček

9.2.3 Paralelizace krabiček

V tuto chvíli nemůže krabička používat žádnou jinou paralelizační technologii než tu, kterou poskytuje systém Bobox. Například třídící krabička třídí data sériově, ačkoliv existují paralelní algoritmy, které by třídění mohly značně urychlit. Samozřejmě používat paralelní algoritmus pro třídění má význam jenom tehdy, kdy není uzel zcela vytížen a kdy zbývají vlákna, která by mohla třídit paralelně.

Řešení takového problému je trochu náročnější, ale v zásadě se nabízí tyto přístupy:

- Změnit dynamicky graf tak, aby paralelizace algoritmu proběhla na úrovni krabiček. Tzn., že by se např. pro třídící krabičku vytvořilo několik podkrabiček, přičemž každá by setřídila určitou část dat, a tyto krabičky by posílaly setříděná data dále do slévacích krabiček, ty do dalších slévacích, až nakonec do finální krabičky, která by rovnou odesílala setříděná data. Příklad takového uspořádání je na obrázku 9.2. Dynamická změna grafu je zde zmíněna proto, že by počet dílčích třídících krabiček závisel na skutečném počtu vláken systému (resp. by bylo možné vytvořit dostatečný počet krabiček staticky).

- Použít plánovač v existující podobě (případně jej trochu upravit) a nechat do něj krabičky vkládat vlastní úlohy. O paralelizaci by se tak musely starat explicitně samy.

- Přizpůsobit plánovač Boboxu tak, aby mohl být použit jako hlavní plánovač TBB nebo OpenMP, a tím krabičkám umožnit používat některou z těchto technologií přímo v jejich implementacích.

Kapitola 10

Závěr

Grafy 10.1, 10.2, 10.3 a 10.4 zachycují, jak nový plánovač a alokátor paměti zvýšil výkon systému na jednotlivých testovacích počítačích v závislosti na počtu paralelních požadavků. Zápis 16x4 znamená, že bylo spuštěno celkem 64 požadavků, ale ty byly back-endu předávány tak, aby běžely paralelně maximálně čtyři. Výsledky byly naměřeny v souladu s popisem v kapitole 4, takže všechna měření proběhla s použitím nové třídící krabičky. Ta se tedy nijak nepodílí na zrychlení, které z naměřených hodnot vyplývá. Měření nejsou všechna stejná, neboť počet paralelních požadavků je v případě výchozí implementace omezen dostupnou operační pamětí. Nový plánovač naopak (díky postupnému zpracovávání požadavků) umožňuje zařadit do systému libovolný počet požadavků najednou, aniž by hrozilo vyčerpání paměti. Je ale zřejmý trend, že čím více je spuštěno paralelních požadavků, tím vyšší je účinnost nového plánovače.

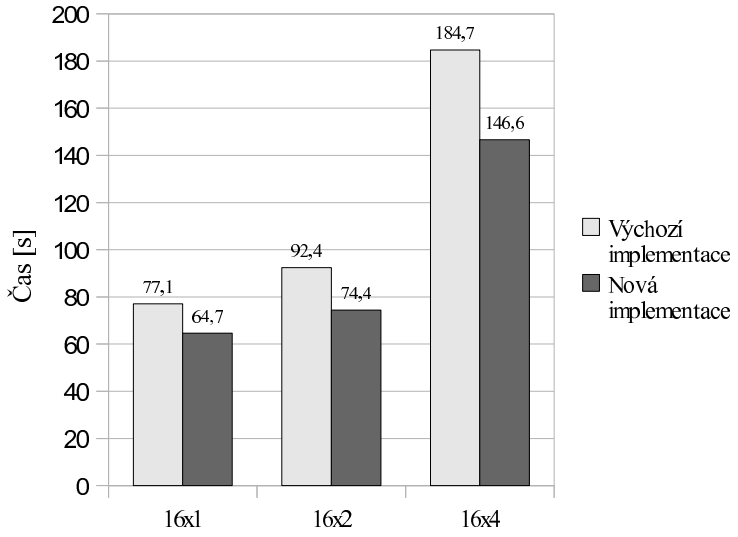
Jako výchozí implementace je použita novější s uspáváním vláken (viz část 5.3), která poskytuje mnohem vyšší výkon než předchozí bez uspávání, neboť aktivní čekání vláken na úlohy zatěžovalo výrazně i ostatní procesory, které prováděly výpočet. Nejvíce se tento jev samozřejmě projevil na počítači *Linux-4x6HT*, ale i u ostatních počítačů nebyl zanedbatelný.

Z grafů je zřejmé, že práce svůj cíl splnila, neboť zrychlení se v závislosti na počítači a počtu paralelních požadavků pohybuje od 6 do 25%. Rovněž je v nové implementaci kladen důraz na to, aby ani přidávání dalších procesorů a uzlů do systému nezpůsobilo, že se nějaká část systému stane jeho úzkým hrdlem.

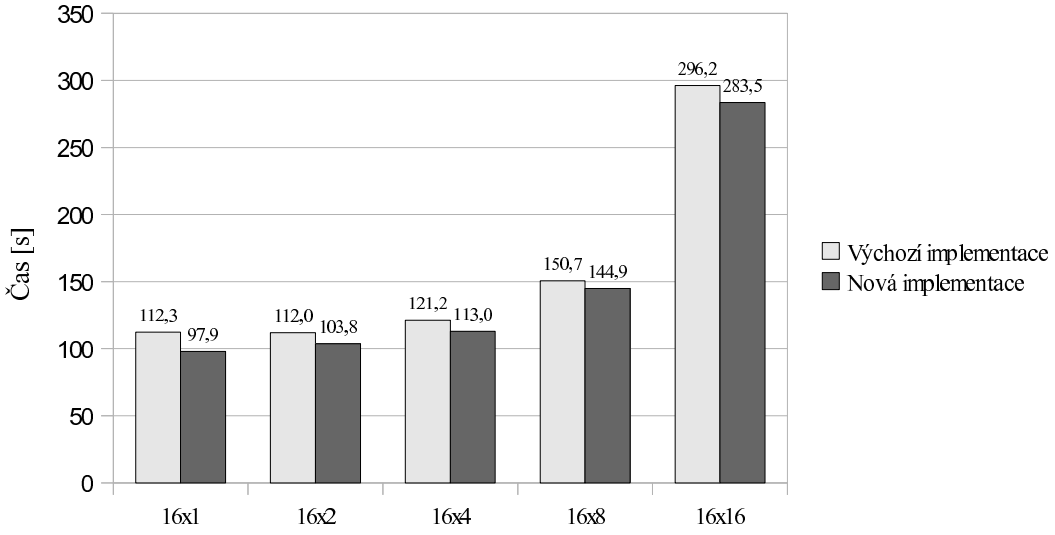
10.1 Budoucí práce

Na druhou stranu existuje několik oblastí, v jejichž zkoumání lze pokračovat. Jedná se zejména o vliv vyrovnávacích pamětí na rychlost systému, neboť existuje velmi mnoho faktorů, které optimální velikost obálky přímo i nepřímo ovlivňují. Tyto faktory většinou mají na velikost obálky protichůdné požadavky. Navíc bylo měření provedeno na malém vzorku různých hardwarových konfigurací, takže získané závěry v této oblasti nejsou zcela přesné.

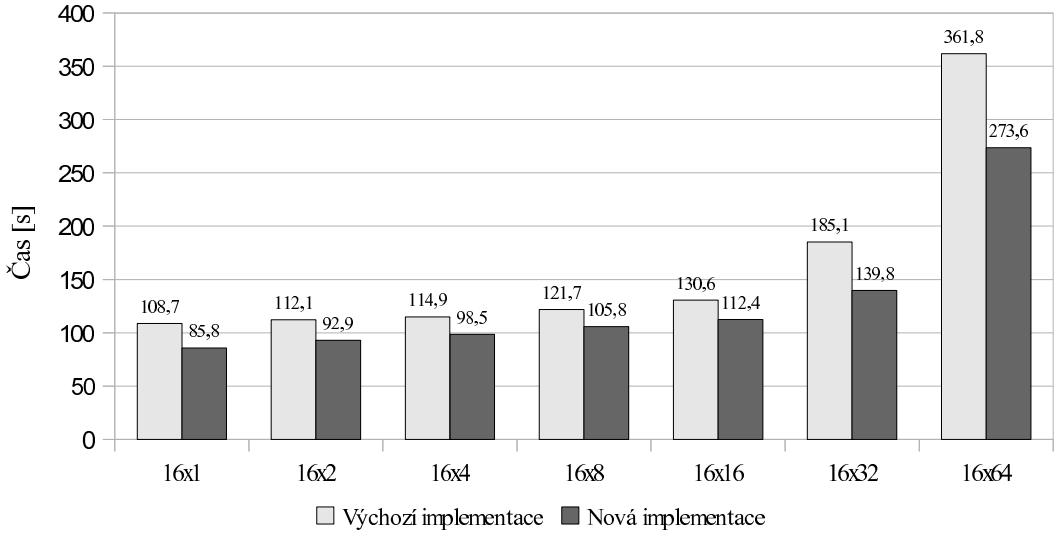
Dále byl plánovač i alokátor testován pouze s jedním front-endem, takže v pří-



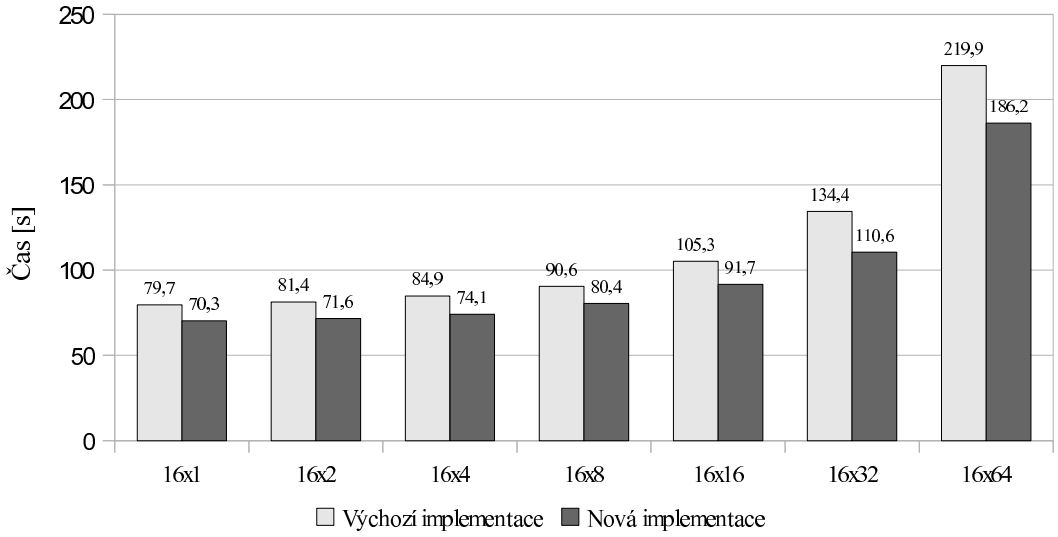
Obrázek 10.1: Účinnost plánovače a alokátoru na počítači *Windows-2*



Obrázek 10.2: Účinnost plánovače a alokátoru na počítači *Linux-8*



Obrázek 10.3: Účinnost plánovače a alokátoru na počítači *Linux-4x6*



Obrázek 10.4: Účinnost plánovače a alokátoru na počítači *Linux-4x6HT*

padě jiných front-endů může systém vykazovat menší či větší zrychlení. Není tedy vyloučené, že některé z uvedených rozhodnutí bude v závislosti na dalším vývoji Boboxu třeba mírně modifikovat. Samotná implementace je proto navržena tak, aby případné změny byly snadno proveditelné.

10.2 Obsah přiloženého CD

Kompaktní disk přiložený k této práci obsahuje kromě tohoto textu v elektronické podobě dvě složky: `original` a `final`. Ve složce `original` jsou k dispozici výchozí zdrojové kódy Boboxu, ve složce `final` jsou zdrojové kódy Boboxu s finální implementací plánovače a alokátoru, se kterým byly změřeny závěrečné výsledky.

Literatura

[1] Becker, P. *Working Draft, Standard for Programming Language C++* [online].
URL: <http://www.open-std.org/JTC1/SC22/WG21/docs/
papers/2007/n2461.pdf>
Verze z 22. října 2007 [citováno 29. července 2010]

[2] Bednárek, D., Dokulil, J., Yaghob, J., Zavoral, F. The Bobox Project - A Parallel Native Repository for Semi-structured Data and the Semantic Web.
IX. Informačné technológie – aplikácie a teória
Seňa: PONT s.r.o., 2009. s. 44-59. ISBN: 978-80-970179-1-0

[3] Bednárek, D., Dokulil, J., Yaghob, J., Zavoral, F. Using Methods of Parallel Semi-structured Data Processing for Semantic Web
Proceedings of the 2009 Third International Conference on Advances in Semantic Processing, Washington, DC, USA: IEEE Computer Society, 2009. s. 44-49. ISBN:978-0-7695-3833-4

[4] Chandra, R., et al. *Parallel Programming in OpenMP*
San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001. 230 s. ISBN:1-55860-671-8

[5] *Condition Variables (Windows)* [online]
URL: <http://msdn.microsoft.com/en-us/library/ms682052(VS.85).aspx>
Verze z 5. května 2008 [citováno 18. července 2010]

[6] Dokulil, J., Yaghob, J., Zavoral, F. Trisolda: The environment for semantic data processing.
International Journal On Advances in Software, vol. 1, no. 1, 2008. s. 43-58
URL: <http://iariajournals.org/software/>

[7] Duran, A., Corbalán, J., Ayguadé, E. Evaluation of OpenMP Task Scheduling Strategies
Proceedings of the 4th international conference on OpenMP in a new era of parallelism, Berlin, Heidelberg: Springer-Verlag, 2008. s. 100-110.
ISBN-ISSN:0302-9743, 978-3-540-79560-5

[8] Intel® Corporation. *Intel® Threading Building Blocks – Reference Manual* [online, PDF].
Revize 1.20 z 28. května 2010 (číslo dokumentu 315415-007), [cit. 2010-07-18]

URL: <http://www.threadingbuildingblocks.org/uploads/81/91/
Latest%20Open%20Source%20Documentation/Reference.pdf>

[9] Kaminski, P. *NUMA aware heap memory manager* [online, PDF].
URL: <http://developer.amd.com/Assets/
NUMA_aware_heap_memory_manager_article_final.pdf>
Verze z roku 2009 [citováno 31. července 2010]

[10] Kleen, A. *An NUMA API for Linux* [online, PDF].
URL: <http://www.halobates.de/numaapi3.pdf>
Verze ze srpna 2004 [citováno 28. července 2010]

[11] Kukanov, A., Voss, M. J.: *The Foundations for Scalable Multi-Core Software in Intel® Threading Building Blocks.*, Intel Technology Journal [online].
URL: <http://www.intel.com/technology/itj/2007/v11i4/
5-foundations/1-abstract.htm>
Verze z listopadu 2007 [citováno 18. července 2010]

[12] Lever, C., Boreham, D. malloc() performance in a multithreaded Linux environment
Proceedings of the annual conference on USENIX Annual Technical Conference
Berkeley, CA, USA: USENIX Association, 2000. s. 56-56.

[13] Reinders, J.: *Intel threading building blocks*
Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2007. 332 s.
ISBN:9780596514808

[14] Schmidt, D. C., Pyarali, I. *Strategies for Implementing POSIX Condition Variables on Win32* [online].
URL: <http://www.cs.wustl.edu/ schmidt/win32-cv-1.html> [online]
[citováno 26. července 2010]

[15] The Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 2.2*
Stuttgart: High Performance Computing Center, 2009. 647 s.

[16] VMware®. *The CPU Scheduler in VMware ESX® 4.1*
URL: <http://www.vmware.com/resources/techresources/10131>
Revize z 13. června 2010 [citováno 31. července 2010]

[17] W3C®. *SPARQL Query Language for RDF* [online].
URL: <http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/>
Verze z 15. ledna 2008 [citováno 14. července 2010]

[18] W3C®. *XQuery 1.0: An XML Query Language* [online].
URL: <http://www.w3.org/TR/2007/REC-xquery-20070123/>
Verze z 23. ledna 2007 [citováno 14. července 2010]