MASARYKOVA UNIVERZITA
FAKULTA INFORMATIKY

Semisynchronní transkoding multimediálních dat

DIPLOMOVÁ PRÁCE

Roman Vaníček

Brno, 2007

Prohlášení

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

Vedoucí práce: RNDr. Eva Hladká, Ph.D.

Poděkování

Děkuji především vedoucímu své diplomové práce, RNDr. Evě Hladké, za všestrannou pomoc a připomínky. Dále všem těm, kteří mi pomohli odstranit chyby, jichž jsem se při psaní této diplomové práce dopustil.

Shrnutí

Semisynchronní transkoding multimediálních dat představuje takový výpočetní problém, kde rychlost produkce výstupních dat kódovacího algoritmu musí přibližně odpovídat rychlosti jejich přehrávání. Při použití běžného hardware není možné semisynchronního zpracování dosáhnout s jediným výpočetním strojem.

Práce se zabývá obecnějším problémem – plánováním proudově propojených procesů na výpočetním gridu. Dělitelnost multimediálních dat umožňuje distribuovat části vstupních dat výpočetním uzlům gridu. Tato forma paralelizace umožní semisynchronní zpracování dat i pomocí běžně dostupného hardware.

Součástí navrženého řešení jsou: běhový systém Grid Streaming Runtime, který zajišťuje provádění naplánovaných proudových úloh, a dva plánovače, jeden globální na úrovni celého gridu a druhý na úrovni každého výpočetního uzlu. Globální plánovač je propojen s plánovacím systémem PBS a problém plánování vyjadřuje formou lineárního programování s reálnými proměnnými.

Platformou implementace je virtuální stroj Common Language Infrastructure (CLI). Virtualita stroje zajišťuje přenositelnost programů v binární podobě v heterogenním prostředí gridu. Důraz je kladen na dvě hlavní implementace CLI – CLR a Mono.

Klíčová slova

plánování, plánovač, proudové zpracování, grid, lineární programování, LP, CLI, PBS, MPI, CLR, Mono

Obsah

Seznam obrázků

Seznam kódu

Kapitola 1

Úvod

Rychlost produkce výstupních dat algoritmu, který provádí semisynchronní transkoding multimediálních dat, musí přibližně odpovídat rychlosti jejich přehrávání. Triviálním řešením takového výpočetního problému je použití stroje s dostatečnou výpočetní kapacitou. Pokud stroj takové síly nelze sestrojit nebo tomu brání vysoké pořizovací náklady, pak je nutné provést výpočet na několika slabších strojích paralelně.

Navrhovaný výpočetní model musí sledovat několik cílů. Jednak přidání nových strojů do současného výpočetního systému sníží výpočetní dobu dané množiny úloh téměř lineárně v poměru k přidanému výkonu – škálovatelnost[1]. A pak také programy nesmí být monolitické, ale explicitně strukturované do částí tak, aby plánovač měl informaci o návaznostech těchto částí. Semisynchronnost dále vyžaduje vstupní data, která jsou dělitelná a zpracovatelná paralelně. V neposlední řadě musí tento výpočetní model klást minimální požadavky při implementaci. V opačném případě bude tvorba programů složitá, pomalá a s vysokou chybovostí.

Práce se zabývá výpočetním modelem proudově propojených procesů v prostředí výpočetního gridu a navrhuje a implementuje infrastrukturu od nejvyšší úrovně plánovače až po vlastní spuštění proudových procesů.

1.1 Grid

Výpočetní systém je označován pojmem **grid** [8] pokud má následující vlastnosti:

- hardware
  - skládá se z **výpočetních uzlů**, což je stroj (stroje) spravovaný právě jedním operačním systémem. Samotný hardware výpočetního uzlu není nijak omezen a může být realizován od běžně

1. scalability

dostupných stolních strojů až po superpočítače (případně může být i virtualizován).

- operační systém výpočetního uzlu nabízí služby správy paměti, plánování procesů (obvykle preemptivní), synchronizaci mezi procesy, izolaci procesů na úrovni bezpečnosti a síťové služby.
- jednotlivé výpočetní uzly jsou propojeny síťovými kapacitami, obvykle s garantovanou propustností a latencí.

- software
  - nabízí služby rezervace výpočetních kapacit (plánovač) s omezením na požadované vlastnosti jako operační systém, platforma, instalovaný software, připojený hardware atd. pro současně nebo později dodaný program a/nebo data.
  - zajišťuje autorizaci operací uživatelů
  - umožňuje komunikaci zasíláním zpráv
  - software gridu je decentralizovaný, ovšem uživateli se jeví jako centralizovaná služba.
  - poskytuje netriviální kvalitu služeb
  - očekává velkou granularitu problémů

Minimální definice gridu [4] vyžaduje decentralizované řízení zdrojů, použití standardizovaných a otevřených protokolů a netriviální kvalitu služeb.

Ke složitosti gridu jako celku přispívá fakt, že je spravován více nezávislými organizacemi. To činí problémy jak s kompatibilitou software[2] tak s bezpečností. Dále výpočetní uzly jsou nespolehlivé prvky ať již vlastním selháním nebo síťovou nedosažitelností, na kterých musí grid poskytovat netriviální kvalitu služeb.

1.2 Výpočetní modely gridu

Základní vlastností gridu je rezervace výpočetních kapacit a následné spuštění programu v příslušném čase na příslušném stroji (strojích) ve formě procesů. Na základě propojení těchto procesů se dělí do několika **výpočetních modelů** [5, strana 121].

2. Například MPICH2 (implementace MPI) vyžaduje instalaci identické verze na všech strojích.

1.2.1 Volně spojené procesy

Volně spojené procesy jsou logicky spjaté procesy (loosely coupled processes), které po svém spuštění nijak nekomunikují. Typicky se jedná o případy, kdy se velký problém nejprve rozdělí do menších podčástí, které se zpracovávají samostatně. Na závěr se spustí úloha, která výsledky jednotlivých podčástí spojí do konečného výsledku.

Tento jednoduchý model je přímo podporován ve většině gridových software.

1.2.2 Workflow

Workflow je koordinované spouštění závislých úloh, kde úloha je spuštěna až poté, co skončí výpočet všech jejích předchůdců. Má hlavně význam pro workflow, která nejsou ve tvaru řetězce, protože umožní spuštění několika nezávislých úloh paralelně (obvykle na různých výpočetních uzlech) v rámci jednoho workflow.

Obecně tento model není podporován současným gridovým softwarem, ale existují doplňující nadstavby [3].

1.2.3 Rourou spojené procesy

Nevýhodou workflow jsou potenciálně velké mezivýpočty, které je nutné uložit minimálně do dokončení všech přímo závislých úloh, v horším případě i přenést mezi výpočetními uzly. Další potenciální nevýhodou je omezený stupeň paralelizace, protože následník je spuštěn až po úplném dokončení předchůdců.

Existuje řada problémů, které lze řešit pomocí lokální znalosti dat, tzv. datového okna (pojem je detailně diskutován v kapitole 5). Jedná se o většinu kódovacích/dekódovacích problémů (kódování, kryptografie, zpracování multimédií), vyhledávání informací v databázích, filtrování dat v reálném čase nebo i prostý přenos dat ze zdroje k cíli. Datové okno je v mnoha případech dokonce konstantní velikosti, a tedy obvykle nevyžaduje velké paměťové nároky. Dále existují polynomiální algoritmy, které minimalizují množství vyrovnávací paměti na koncích roury [6].

Při spouštění úlohy v modelu s rourou spojenými procesy spustí grid procesy dané úlohy na naplánovaných výpočetních uzlech a propojí je rourou, kterou potečou výstupní data předchůdců k následníkům. Grid spravuje roury transparentně a vlastní procesy neví odkud data přichází a kam odchází. Kritické je tedy současné spuštění procesů a synchronní výpočet na obvykle mnoha výpočetních uzlech.

Výhodou tohoto modelu oproti workflow je možnost efektivnější paralelizace za cenu náročnějšího plánování. Nevýhodou je omezení třídy problémů, které je model schopen řešit.

Současný gridový software tento model nepodporuje, nicméně lze jej realizovat pomocí samostatného plánovače (metaplánovače) a modelu s volně spojenými procesy [11, 14].

1.3 Rozvržení kapitol

Práce je dělena následovně. Kapitola 2 popisuje a srovnává používané přístupy při řešení obecných problému plánování. V návaznosti na to kapitola 3 přidává fakta specifická pro výpočetní grid.

Kapitola 4 se zabývá vlastním návrhem infrastruktury nutné k plánování na úrovni výpočetních uzlů na platformě virtuálního stroje CLI[2]. Obdobně kapitola 5 popisuje návrh proudového zpracování dat v CLI.

Kapitola 6 popisuje problém plánování proudově propojených procesů na gridu jako problém lineárního programování (LP) a jeho napojení na správce úloh PBS.

Kapitola 2

Plánování

Přiřazení zdrojů úlohám v čase s minimalizací účelové funkce a v rámci daných výkonnostních omezení zdrojů a dalších omezení, se nazývá **plánování** (scheduling) [15, 16]. V české literatuře se také používá termín rozvrhování. Dále termíny plánování, rozvrhování a plán, rozvrh budou vždy znamenat scheduling a schedule.

2.1 Grahamova klasifikace

Pro popis problémů plánování se používá Grahamova klasifikace [1]

$$\alpha|\beta|\gamma$$

Dále nechť $M_0, ..., M_m$ jsou stroje, $P_0, ..., P_p$ operace a $r_{i,j}$ délka zpracování operace i na stroji j.

- α prostředí strojů. Dělí se na **identické paralelní stroje**, pokud $\exists r_i \forall j : r_i = r_{i,j}$. V případě, že stroje jsou různě rychlé, ale pro všechny operace stejně $(\exists s_j \forall i_1, i_2)(\frac{r_{i_1,j}}{r_{i_2,j}} = s_j)$, se jedná o **uniformní paralelní stroje**. Pokud je délka zpracování závislá na operaci i stroji, tak mluvíme o **nezávislých paralelních strojích**.

- β vlastnosti (omezení) úloh. Například preempce (přerušitelnost), **precedence** (závislosti úloh), precedence ve tvaru stromů, řetězců aj.

- γ optimalizační kritéria (účelová funkce)

Za předpokladu, že zdroje znemožňují vícenásobné použití, lze problém plánování zjednodušit. Na jednom stroji probíhá v každý okamžik maximálně jedna operace. Například se jedná o stroje na výrobní lince, odbavovací přepážky a jiné objekty, které ve své podstatě znemožňují vícenásobné použití. Nicméně stroje mohou obsahovat více nástrojů, kterými mohou postupně vykonávat různé operace – mluvíme o **víceúčelových strojích**.

2.2 Plánovací algoritmus

Obecný **plánovací algoritmus** lze formálně popsat jako parciální funkci

$$\begin{aligned} f &: (\alpha, \beta, \gamma) \rightarrow P \\ f &: (M, P, \gamma) \rightarrow M \times P \times T \times T \times R, \end{aligned}$$

kde

- M jsou stroje z prostředí strojů
- P jsou plánované operace z vlastnosti úloh
- $T \subseteq \mathbb{R}$ je množina časových okamžiků
- $R \subseteq \mathbb{R}$ je procentuální vytížení,

která počítá časový interval ($T \times T$), kdy daná úloha (P) je počítána daným strojem (M) daným procentem (R) svého maximálního využití. Dále řešení by mělo být globálním minimem (maximem) vzhledem k γ a musí splňovat omezení β.

Pojem stroj v definici nemusí nutně znamenat jeden počítač, ale spíše jeden jeho procesor (jádro).

2.3 Statické a dynamické plánování

Pokud se prostředí strojů a vlastnosti úloh nemění během plánovaného horizontu, pak mluvíme o statickém plánování. V opačném případě je nutné řešit změny během provádění již existujícího plánu – **dynamické plánování**[1].

Dynamické plánování ve své podstatě nemůže zaručit optimalitu z pohledu finálního prostředí strojů, ale sleduje několik kritérií, která vyjadřují schopnost plánovacího algoritmu vyrovnat se s nastalými změnami [9]. Hlavním kritériem je schopnost nalezení nového plánu s využitím plánu existujícího tak, aby nový rozvrh obsahoval minimum změn ve srovnání s plánem původním[2] [7]. Toto kritérium je konfliktní s kritériem optimality. Hledání řešení u dynamického plánování je dále omezeno množstvím času, který má výpočet k dispozici.

1. on-line plánování
2. minimal perturbation problem

2.4 Algoritmy plánovacích problémů

Plánovací problémy obecně patří mezi NP-těžké problémy, proto existuje několik technik a algoritmů k jejich řešení. Řešením problému plánování je nalezení plánu, který je nejlepší vzhledem k optimalizačním kritériím – **globální optimum**.

Lokálním optimem nazýváme řešení, které se nachází v lokálním extrému optimalizační funkce (množina lokálních optim obsahuje i globální optimum). Algoritmus, který začíná s libovolným platným řešením (inicíální řešení) od něhož postupně v iteracích přechází k okolním řešením až skončí v lokálním optimu účelové funkce, se nazývá **algoritmus lokálního prohledávání**.

Nejdůležitější částí lokálního prohledávání je funkce **okolí** $o : 2^R \rightarrow 2^R$, pomocí které se přechází od jednoho (nebo menšího počtu) řešení k sousedním řešením. Sousedství by mělo být dostatečné malé, ale pestré. Malé z toho důvodu, že obvykle pro všechna řešení z daného okolí se vyhodnotí účelová funkce a na základě **rozhodovací funkce** se vybere jedno (nebo malé množství) do další iterace. Pestrost okolí zajistí vyšší pravděpodobnost perspektivního řešení, které vede k optimu a v neposlední řadě omezí pravděpodobnost nekonečného cyklu.

Návrh okolí pro složitější problémy plánování je netriviální. První problém spočívá v určení všech základních transformací, pomocí kterých lze prohledat celý prostor řešení. Často se jedná o permutace, případně základní aritmetiku (sčítání, násobení). Druhým problémem je, že elementární transformace u složitějších problémů může porušit jedno nebo více omezení. Buď se takový produkt zamítne jako řešení s nebezpečím neprohledání části prostoru řešení, nebo je nutné propagovat změnu iterativně, dokud produkt nevyhoví omezením a nestane se řešením.

Druhou důležitou částí je rozhodovací funkce, která vybírá kandidáty z okolí pro vstup do další iterace. Většina algoritmů umožňuje místy přechod i k horšímu řešení (vzhledem k účelové funkci), aby se zamezilo uvíznutí v lokálním minimu.

2.5 Deterministické algoritmy

Deterministické algoritmy pro daný vstup vypočítají (pokud existuje) vždy stejné řešení.

Mezi deterministické algoritmy patří **přesné řešící algoritmy**, které pro daný problém naleznou vždy globální optimum. Pro jejich praktické použití je vhodná polynomiální složitost. Teorie plánování zkoumá třídy plánova-

cích problémů za účelem nalezení polynomiálních algoritmů [15, 16].

Pokud algoritmus vždy nalezne řešení r takové, že

$$k \geq 1 \wedge o \cdot k \geq r$$

kde o je optimální řešení, mluvíme o **k-optimálním algoritmu**. k-optimální algoritmy nalézají řešení, která mají garantovanou odchylku od optimálního řešení. Hlavním představitelem jsou hladové algoritmy.

2.5.1 Programování s omezujícími podmínkami

Základem programování s omezujícími podmínkami (constraint programming, CP) je splňování podmínek (constraint satisfaction, CS), ze kterého se CP [13] vyvinulo. Na rozdíl od lineárního programování (LP) je mnohem pružnější ve struktuře vyjadřování omezení. Lze v něm přímočaře vyjádřit výrazy výrokové logiky, polynomy vyšších řádu než lineární apod.

Model CP obsahuje vektor proměnných $\mathbf{x}$ a omezení, která je nutné splnit. Pro každou proměnnou x_i je navíc sledována doména D_i (obor hodnot, díváme-li se na proměnné jako na nulární funkce). Během vyhodnocování CS se postupně přiřazují hodnoty proměnných z jejich oboru hodnot, a ihned se omezují obory hodnot všech proměnných závislých na právě přiřazené proměnné přes podmínky, tzv. **šíření omezení**. Díky tomu není zdaleka nutné vyzkoušet všechna přiřazení iniciálního problému. Vyřešením CS získáme iniciální řešení x^*.

Zavedením účelové funkce $\min o(\mathbf{x})$ do CS jako podmínku $o(\mathbf{x}) < x^*$, kde x^* je dosud nejlepší nalezené řešení (v první iteraci iniciální), získáme optimalizační problém **programování s omezujícími podmínkami**. V každé iteraci se spustí CS, v případě nalezení řešení se nahradí x^* nově nalezeným řešením a pokračuje se. V opačném případě bylo dosaženo globálního optima. V případě velmi mnoha řešení účelové funkce je možné v případě znalosti přibližné dolní meze postupovat binárním půlením intervalu k získávání nových x^*. K získávání těchto mezí se používá LP.

Z postupu optimalizace CP je vidět, že nejefektivněji řeší problémy s diskrétními hodnotami s malými doménami. V opačném případě CS iteruje přes příliš mnoho hodnot ve chvíli, kdy volnost problému leží právě v proměnných s velkou doménou. Z tohoto důvodu je snaha v budoucnu vytvořit hybridní implementace CP a LP, které využijí nejlepší vlastnosti obou [16, 17].

2.5.2 Lineární programování

Lze-li problém reprezentovat pomocí n proměnných $\{x_i | x_i \in \mathbb{R} \wedge 0 \leq i < n\}$, jejichž hodnoty lze omezit lineárními nerovnostmi $\sum_i c_i \cdot x_i \leq b$ a $o : x_0 \times \ldots \times x_{i-1} \rightarrow \mathbb{R}$ lineární účelovou funkcí, pomocí které se vybírá mezi možnými řešeními vymezenými lineárními nerovnostmi, pak se jedná o lineární optimalizační problém, **lineární programování** (LP).

Definuje se jako matice A, kde řádky jsou jednotlivá lineární omezení

$$\begin{array}{rcl} & & A \cdot \mathbf{x} \leq \mathbf{b} \\ \begin{pmatrix} c_{0,0} & \cdots & c_{0,i-1} \\ \vdots & & \vdots \\ c_{j,0} & \cdots & c_{j,i-1} \end{pmatrix} \cdot \mathbf{x} & \leq & \begin{pmatrix} b_0 \\ \vdots \\ b_j \end{pmatrix} \end{array}$$

a množina všech řešení je

$$\{\mathbf{x} | \mathbf{x} \geq 0\}$$

s účelovou funkcí

$$\begin{array}{rcl} \mathbf{o} & = & (d_0, \ldots, d_{i-1}) \\ o(\mathbf{x}) & = & \sum_i d_i \cdot x_i \end{array}$$

Pokud jsou veškeré proměnné nad doménou reálných čísel, pak existují polynomiální algoritmy vůči počtu proměnných[3], které vždy naleznou globální minimum pokud existuje. V případě, že proměnné jsou celočíselné, pak mluvíme o **celočíselném programování** (ILP). V obecném případě ILP není polynomiální, protože v binárním případě by řešilo problém splnitelnosti výrokových formulí (SAT). Nicméně ILP se často používá v plánování [19, 12, 10, 18].

Podmínka linearity formulace LP si vynucuje před návrhem transformaci z obecného matematického zápisu. Výsledkem je znepřehlednění celého modelu. Například k zavedení proměnné a s celým rozsahem $(-\infty; \infty)$ jsou vyžadovány dvě proměnné a_0, a_1 takové, že pro původní hodnotu platí $a = a_0 - a_1$. Po obdržení výsledku lineárního programování je nutné provést zpětnou transformaci k získání cílových hodnot.

LP je vhodné na řešení problémů s reálnými čísly a s limitovanou vyjadřovací schopností omezení. Bez použití binárních proměnných není možné vyjádřit disjunkci. Ve speciálním případě, kdy jsou splněny podmínky unimodularity platí, že řešení LP je i řešením ILP. Tento případ je často využíván v plánování [14].

---

3. simplexový algoritmus není polynomiální, ale dlouho nebyl překonán při reálném použití

Algoritmy, které řeší lineární problémy, jsou iterativního charakteru. Protože pracují nad reálnými čísly a z důvodu rychlosti využívají pevný počet bitů k reprezentaci čísel (přímo podporované procesory), tak nutně dochází k zaokrouhlovacím chybám, jejichž akumulace může i znemožnit najít existující globální optimum.

2.5.3 Hladové algoritmy

V teorii plánování se hladové algoritmy nazývají **řídícími pravidly**. Řídící pravidla určují výběr úlohy ve chvíli, kdy se uvolní nějaký zdroj [15, 16]. Jejich výhodou je jednoduchá implementace, rychlost a možnost použití při dynamickém plánování. Nevýhodou je až na speciální případy neoptimalita řešení.

Mezi základní pravidla se řadí FCFS (First Come First Served), EDD (Earliest Due Date), ERD (Earliest Release Date), LPT (Longest Processing Time first) a SPT (Shortest Processing Time first).

2.6 Stochastické algoritmy

Přechodová funkce stochastických (náhodnostních) algoritmů obsahuje alespoň jeden stav, ze kterého může s danými pravděpodobnostmi přejít do jiných stavů. Z pohledu řešení těžkých problémů jsou zajímavé ty stochastické algortimy, které pro daný vstup vypočítají daný výsledek s pravděpodobností méně než jedna – **heuristické algoritmy**.

2.6.1 Simulované žíhání

Simulované žíhání[4] je algoritmus lokálního prohledávání s náhodnostní rozhodovací funkcí zvanou Metropolisovo kritérium. Jedná se o model fyzikálního jevu ochlazování kovů, kde vyšší teplota umožní atomům přejít do energeticky náročnějších stavů a během postupného ochlazování postupně přejde do energetického lokálního minima, o kterém se očekává, že je blízko globálnímu minimu. Vysoká teplota (energie) umožňuje potenciálně přechod k libovolnému řešení díky náhodnému pohybu atomů.

Definice 2.6.1 (Metropolisovo kritérium) *Pravděpodobnost přijetí řešení R_{i+1} z řešení R_i při účelové funkci C je určeno funkcí M v rovnici 2.1.*

4. simulated annealing

$$M(R_i, R_{i+1}) = \begin{cases} 1 & \text{pro } C(R_{i+1}) \leq C(R_i) \\ e^{\frac{C(R_i)-C(R_{i+1})}{T}} & \text{pro } C(R_{i+1}) > C(R_i) \end{cases} \tag{2.1}$$

Lepší řešení jsou vždy přijata, zatímco horší pouze při dostatečné teplotě. Čím horší řešení, tím vyšší teplota je nutná pro reálnou pravděpodobnost přijetí. Návrh okolí simulované žíhání nijak neomezuje a tedy ani nijak neusnadňuje.

2.6.2 Tabu prohledávání

Problémem algoritmů lokálního prohledávání je nebezpečí uváznutí v nekonečném cyklu. Tabu prohledávání může nekonečným cyklům zamezit, častěji však jen sníží pravděpodobnost, že nastanou.

Definice 2.6.2 *Nechť A je algoritmus lokálního prohledávání, který používá transformační funkce $F = \{T : R \to R\}$ k nalezení řešení v rámci okolí. Dále pro každou transformaci $t \in F$ existuje její inverze $t^{-1} \in F$. Pokud rozhodovací funkce zamítne řešení nalezené transformací t, kde t^{-1} byla transformace jednoho z posledních n přijatých řešení, pak A je algoritmus* ***tabu prohledávání****.*

Posloupnost posledních několika transformací, obvykle v řádu jednotek, slouží jako obrana proti krátkým cyklům. Oproti simulovanému žíhání je proces přijetí/odmítnutí deterministický, nicméně výběr okolního řešení omezen není, a je obvykle stochastický.

Návrh okolí je opět zcela neomezen a tedy neusnadněn.

2.6.3 Genetické algoritmy

Zatímco v posledních dvou algoritmech přecházelo do další iterace lokálního prohledávání právě jedno řešení, u **genetických algoritmů** přechází hned několik řešení – generace [20].

Transformační funkce okolí se kategorizují do dvou skupin, **rozmnožovací funkce** $f : R \times R \to R$ a **mutační funkce** $g : R \times \mathbb{R} \to R$. Rozmnožovací funkce kombinuje dvě platná řešení v jedno, potenciálně nové. Protože velikost a pestrost generace je řádově menší než prohledávaný prostor, rozmnožovací funkce samotné ho pokrýt nedokáží. Řešením jsou tedy mutační funkce, které s jistou pravděpodobností (druhý argument) provedou svou transformaci, a tím přidají vlastnost dosud neprohledávatelnou.

Z pohledu genetických algoritmů obsahují simulované žíhání a tabu prohledávání pouze mutační funkce. Kromě deklarace typu transformačních funkcí není jejich návrh opět nijak usnadněn.

2.7 Reprezentace času v plánovacích problémech

Při formulaci plánovacího problému pomocí některého z výše popsaných algoritmů je nutné se nejdříve rozhodnout, jak reprezentovat čas. Výstupem plánovacího algoritmu je nutně časový interval ($T \times T$).

V přímočarém případě je interval reprezentován dvojicí proměnných. Pro stochastické algoritmy (tabu, žíhání, genetické) to znamená zvětšit již tak obrovský prostor řešení o další dvě dimenze. Programování s omezujícími podmínkami zatěžuje naopak velká doména těchto proměnných, která brání efektivní eliminaci domén závislých proměnných [13]. Lineární programování je obecně vhodné pro řešení kontinuálních proměnných [13], jenže intervalové proměnné znemožňují například formulaci omezení na přetížení zdrojů, ve kterém je třeba počítat sumy přes průniky časových intervalů. Bez použití binárních proměnných podmínky poruší linearitu; v případě jejich použití způsobí obvykle exponenciální složitost.

Implementace programování s omezujícími podmínkami směřuje k hybridní kombinaci s lineárním programováním, které významně sníží rozsahy proměnných [13].

Formulace čistě v LP musí přistoupit k alternativním reprezentacím času. Jednou z možností je rozdělení času na intervaly pevné délky, **kvanta**, a veškeré plánování provádět pro jednotlivá kvanta. Přirozeným problémem je výběr jeho délky, které buď vede k přílišné fragmentaci využití zdrojů (příliš dlouhé) nebo velké výpočetní náročnosti (příliš krátké).

Další možností je předvybrat pouze potenciálně vhodné intervaly a přiřadit jim binární proměnné. Počet binárních proměnných roste sice velmi rychle, ale řešení pomocí generování sloupců může být realizovatelné [18]. Předvýběr intervalů může být také omezen dalšími podmínkami, v případě lidí například pracovními smlouvami a zákoníkem práce, čímž dojde k dalšímu omezení potenciálních intervalů.

Kapitola 3

Plánování úloh na gridu

Při plánování úloh pro výpočetní grid je výhodné přihlédnout ke specifickým vlastnostem, které je možné využít k zjednodušení podmínek plánovacího problému. V ideálním případě natolik, že algoritmus bude mít polynomiální složitost s přijatelnou odchylkou od optimálního plánu.

Počítače se od většiny plánovaných zdrojů – lidí, strojů na lince, odbavovacích přepážek – liší tím, že mají velmi malou cenu za změnu. Jsou tedy velmi efektivní v přerušování operací jinými operacemi (za předpokladu, že mají dostatek operační paměti). Díky tomu lze současně přidělit jednomu stroji n operací a vektor $\{(p_1, \ldots, p_n) | 0 \leq p_i \leq 1 \wedge \sum_i^n p_i = 1\}$, který určuje kolik procent výpočetního času se má v průměru vykonávat příslušná operace [14]. Pokud n není natolik velké, aby cenu za změnu přesáhla únosnou mez (řádově desetiny procent), a časový interval je dostatečně dlouhý, aby došlo ke zprůměrování (řádově $n \cdot \frac{1}{\min_i p_i}$, sekundy), pak jsou soudobé operační systémy schopny dané zadání splnit.

3.1 Nespolehlivost výpočetních zdrojů

Gridy se z definice skládají z nespolehlivých výpočetních uzlů a spojů. Úkolem gridového software je tedy i sledování již běžících naplánovaných operací a jejich případné okamžité přeplánování v případě výpadku.

Grid obvykle spravuje několik nezávislých organizací s různou kvalitou správy strojů, rozmístěných ve značných vzdálenostech. Pravděpodobnost výpadků strojů a spojení je spíše vyšší, než v plánovacích problémech výrobní linky nebo odbavovacích terminálů. Provoz výrobní linky sice omezuje spolehlivost subdodavatelů, nicméně skladové zásoby mohou přeplánování oddálit. V případě gridu při výpadku dojde k nutnosti okamžitého, případně velmi brzkého přeplánování.

Pro plánování úloh na gridu není možné použít statický plánovač.

3.2 Decentralizované plánování

Vzhledem k nespolehlivosti strojů není vhodné vybrat právě jeden stroj, jehož jediným úkolem bude plánovat provoz celého gridu. Je sice uskutečnitelné určovat plánovací stroj volbou všech výpočetních uzlů gridu, jenže tento přístup zamezuje efektivními růstu gridu, protože s růstem počtu strojů rostou velmi rychle i požadavky na výkon stroje. I komunikační zpoždění napříč gridem přestane být zanedbatelné.

Plánovací algoritmus je vhodné spouštět decentralizovaně, buď na každém výpočetním uzlu [14], nebo na jednom uzlu v rámci skupiny strojů (tzv. cluster). **Cluster** je skupina strojů propojených datovými spoji s malým zpožděním a velkou kapacitou spravovaných jednou organizací. Jeho maximální velikost je obvykle do sta strojů. Na každý okolní přímo dosažitelný cluster pak může plánovač nahlížet jako na jediný výpočetní uzel – obvykle značně výkonný.

3.3 Plánovače operačních systémů

Gridy běží na běžně dostupných a používaných operačních systémech (Windows, Linux a další UNIXové systémy), jejichž plánovače procesů provádí obvykle přerušitelné plánování procesů. Každý plán vytvořený na úrovni celého gridu musí být v konečném důsledku realizován spuštěním jednoho nebo více procesů operačního systému.

Při přerušitelném plánování procesů je běžící proces (vlákno) přidělen právě jednomu procesoru stroje. Po době pevné délky, **časovém kvantu**, je přerušen. K přerušení může dojít dříve, pokud se proces při své činnosti sám zablokuje (synchronizací s jinými procesy nebo interakcí s externím zařízením).

Plánovač gridu může dále využít faktu, že některé úlohy mohou intenzivněji využívat procesor, a jiné externí zařízení. Souběh takových úloh pak může i přesáhnout teoretickou rychlost stroje, protože výpočetní úloha může být spuštěna v době čekání na odezvu externího zařízení pro druhou úlohu, která se tím zablokovala, a přerušena pouze během zpracování odezvy. Externími zařízeními mohou být disky, síťová rozhraní aj., které v mnoha případech umožňují přenos dat přímo z operační paměti bez zatěžování procesoru (DMA).

Zcela nezbytné je, aby plánovač zajistil, že veškeré souběžné úlohy a jejich data se v době přerušení vždy vydají do operační paměti stroje. V opačném případě musí operační systém během přerušování úloh odkládat/číst virtuální paměť do/ze sekundární paměti (disku). Takové chování způso-

buje velkou režii a je časově zcela neodhadnutelné.

Časové kvantum se obvykle pohybuje v řádech jednotek až desítek milisekund[1]. Výběr dalšího procesu, který bude spuštěn, má zanedbatelnou složitost, pokud n je v řádech, na které jsou operační systémy stavěny a požadovaná doba průměrování zatížení stroje se pohybuje minimálně v sekundách, což jsou volnější požadavky než jaké mají interaktivní programy. Plánovače operačních systému předpokládají maximálně stovky procesů. Poté poměr režie vyhledání dalšího procesu a délky časového kvanta začne být nepřijatelný. Pokud na přelomu časového kvanta dojde k výměně procesu, tak není zanedbatelné ani zpoždění na úrovní hardware způsobené vyprázdněním vyrovnávacích pamětí[2].

Možnost změny plánovače operačního systému je velmi nízká. Brání jí jak technické, tak administrativní překážky. Mezi technické patří nutnost programovat v rámci jádra OS. Takové prostředí neumožňuje přenositelnost a vyžaduje naprostou bezchybnost programu. S tím souvisí i administrativní překážka ve formě nedůvěry provozovatelů k takovému programu.

3.4 Reprezentace času

Obdobně jako při obecném plánování přináší prostředí gridu další možnosti jak reprezentovat čas v plánovacím problému.

Při použití výpočetního modelu s rourou spojenými procesy (1.2.3) lze formulovat problém jako hledání **ustáleného plánu** (steady-state schedule) pro každý výpočetní uzel, tj. najít zlomky času, které budou stráveny komunikací a výpočtem pro každou operaci počítanou výpočetním uzlem tak, aby množství dat vyprodukovaných/konzumovaných operací odpovídalo množství dat konzumovaných/produkovaných u přímo závislých operací [14, 10]. V této formulaci je čas reprezentován implicitně jako jeden obrovský interval. Nevýhodou je, že při velkém množství plánovaných operací dojde k přidělování velmi malých zlomků času každé operaci v rámci jednotlivých výpočetních uzlů. Tím stoupne celková suma ceny za změnu a může dojít k vyčerpání operační paměti.

Další alternativou je uvažovat několik intervalů proměnné délky, v rámci

1. 2-20 intervalů ve Windows NT (PspJobSchedulingClasses[]) kde jeden interval je 10 až 15 milisekund (záleží na hardware)
100 ms Windows CE (http://msdn2.microsoft.com/en-us/library/ms918017.aspx)
10 až 300 ms Linux (run(1) od Concurrent Computer Corporation)
2. jiný proces vyžaduje jiné mapování virtuální paměti (znehodnocení TLB) a pravděpodobnost použití dat v ostatních fyzicky adresovatelných vyrovnávacích pamětech (L1, L2) je malá

kterých bude tvořen ustálený plán. Počet takových intervalů je závislý na počtu tříd relace ekvivalence závislostí mezi operacemi (nezávislé úlohy). Vzhledem k nespolehlivosti prostředí gridu je však účelné dále tento počet omezit tak, aby délka plánovaných intervalu nepřesáhla čas, za který je nanejvýš pravděpodobné, že dojde k přidání nové úlohy nebo k výpadku zdrojů v gridu. Změna prostředí strojů (α) nutně vyžaduje přeplánování.

3.5 Odhadování výpočetní a komunikační náročnosti úloh

Vstupem plánovacího algoritmu je i délka zpracování operace pro každý výpočetní uzel. Ekvivalentním vstupem může být rychlost strojů v počtech instrukcí za sekundu a požadovaný počet instrukcí pro každou operaci v případě uniformních paralelních strojů. Pro nezávislé paralelní stroje je nutné přidat váhový koeficient pro každou implementaci operace pro příslušný výpočetní uzel.

V případě rourou spojených procesů je možné odvodit komunikační nároky s minimem informací o implementaci každé operace (poskytnuté programátorem), a to poměry vstupně-výstupních operací.

Odhady by měly být záležitostí plánovače, s dostatečnou podporou metadat od implementací operací, a nikoliv uživatele. Většina prací, týkajících se plánování znalost těchto nároků jednoduše předpokládá [14] nebo délka operací je předem známa [12] (délka zpracování instrukcí). Jednou z možností, jak požadované informace získat, je provést kontrolované spuštění každé operace [10], tzv. **referenční běh**.

Kapitola 4

Kooperativní plánování nad CLI

Pokud procesy[1] přerušují svůj běh na procesoru vlastním zablokováním na synchronizačním objektu (příp. speciální funkcí k tomu určené), tak mluvíme o **kooperativním plánování**.

Common Language Infrasturcture (CLI) [2] je specifikace virtuálního stroje, spustitelných programů, metadat a datových typů. Instrukční sada programů v rámci CLI se nazývá Common Intermediate Language (CIL). CIL je výstupem kompilátorů z mnoha programovacích jazyků (C#, VB, C, C++, Pascal, ...). V současnosti existují dvě implementace CLI. Vůdčí implementací je Common Language Runtime (CLR)[2] od firmy Microsoft následovaný Mono projektem[3]. CLR běží nad operačním systémem Windows, Mono nad OS Linux, Mac OS X, Sun Solaris, OpenBSD, FreeBSD, NetBSD a Microsoft Windows. Průnik obou implementací je větší než CLI, nicméně tato rozšíření nebudeme dále uvažovat.

V této kapitole je popsána vlastní implementace infrastruktury pro kooperativní plánování nad CLI, tedy bez použití implementačních rozšíření.

4.1 Producent – konzument

Existují algoritmické problémy, jejichž implementaci lze modelovat buď jako **model zasílání zpráv mezi procesy** nebo jako **volání procedur řízené konečným automatem** (VPKA). Výsledná volba má vliv na rychlost implementace a to jak po stránce spouštění tak doby implementace.

Při modelování vztahu producent-konzument formou zasíláním zpráv se na producenta nahlíží jako na proces. Obdobně na konzumenta. Rozhraní producenta je omezeno na funkci Write, kterou zasílá produkovaná data. Proces konzumenta má naopak jedinou funkci Read, která čte data zaslaná producentem. Obě funkce jsou blokující, mohou tedy pozastavit vo-

1. V této kapitole bude slovo proces označovat sekvenční zpracování instrukcí.
2. známější pod jménem .NET Framework
3. http://www.mono-project.com

lající proces. Na implementaci programu producenta/konzumenta nejsou kladeny žádné další požadavky ani žádná omezení.

Z pohledu VPKA bude problém producenta-konzumenta tvořit jediný proces. Implementace se bude skládat ze tří částí – kódu **stavové jednotky**, producenta a konzumenta (výpisy 4.1, 4.2, 4.3). Stavová jednotka zajišťuje přechody mezi stavy konečného automatu, pro náš problém pouze dvou stavů – produkování a konzumování.

Na rozdíl od modelu zasílání zpráv, má VPKA požadavky na implementaci funkcí. Ty musí být schopné zpracovat část vstupních dat, vytvořit částečná data výstupní, a předat řízení stavové jednotce. Předání řízení může být požadováno do určitého časového okamžiku, anebo množství zpracovaných dat.

Implementace pomocí VPKA explicitně kontroluje okamžiky přechodu ze stavu do stavu, protože jeho stav je přímo dostupný formou proměnných programu. Oproti tomu množina procesů, které si zasílají zprávy, také obsahuje stav, ovšem nedostupný jako celek pro jednotlivé procesy – část stavu je obsažena v množině synchronizačních objektů. Je to právě plánovač procesů, který rozhoduje na jejich základě, zda proces může běžet, a zda poběží. Plánovač procesů spolu se synchronizačními objekty je obdobou stavové jednotky u VPKA.

Z pohledu náročnosti implementace je problém producent-konzument jednodušší pomocí zasílání zpráv, protože část stavu je možné držet v zásobníku volaných funkcí[4] v rámci procesu. Totéž není možné u VPKA, kde před každým přechodem stavu je zásobník volání prázdný. Navíc většina operačních systémů obsahuje již obecný plánovač, procesy i synchronizační objekty. Cenou za tuto obecnost je rychlost běhu programu a většinou nemožnost i minimálních změn v programu plánovače.

4.2 Udržovatelnost programového kódu

Hlavním kladem modelu zasílání zpráv je schopnost držet část stavu v zásobníku volání. V důsledku toho program obsahuje méně větvení, protože není nutné kontrolovat návratové hodnoty funkcí, které by mohly indikovat požadavek na návrat do stavové jednotky. Tím se zvyšuje přehlednost programu a snižuje chybovost. Současně se jedná i o nevýhodu. V žádném běžném programovacím jazyku není možné volně přistupovat k proměnným volajících funkcí. Tím je ztíženo uložení úplného stavu procesu (množiny procesů) na disk za účelem pozdějšího spuštění. Ještě komplikovanější je

4. call stack

Výpis kódu 4.1: Funkce stavové jednotky

```
state = S_Produce;
finished = false;
while (!finished) {
  switch (state) {
    case S_Produce:
      finished = Produce();
      state = S_Consume;
      break;
    case S_Consume:
      finished = Consume();
      state = S_Produce;
      break;
  }
}
```

Výpis kódu 4.2: Funkce producenta

```
bool Produce() {
  /* vypočítej část dat */
  Process();
  /* odešli data */
  Write();
  /* rozhodnutí o ukončení */
  return Finished();
}
```

Výpis kódu 4.3: Funkce konzumenta

```
bool Consume() {
  /* přijmi část dat */
  Read();
  /* zpracuj část dat */
  Process();
  /* rozhodnutí o ukončení */
  return Finished();
}
```

pak přesunutí a opětovné spuštění takto uloženého stavu na počítač s jinou architekturou.

4.3 Jiné implementace zasílání zpráv

Operační systémy v kategorii soft-realtime obsahují základní a uživatelem neměnný systémový plánovač spolu s danou sadou synchronizačních objektů. Až na jednoduché příznaky ve formě priorit nemá plánovač v jádře žádnou představu o interakci procesů, které soutěží o procesory, které jsou k dispozici. Plánovač je obecný s cílem zamezení stárnutí a prosazení férovosti. U procesů se očekává délka nepřerušeného běhu na úrovni časových kvant v řádu milisekund.

Pro úzce spolupracující procesy, které vyžadují časté pozastavování a spouštění, se mohou režijní náklady systémového plánovače na přepínání mezi nimi stát nepřijatelnými. Reakcí jsou pak programová rozhraní [5], která umožní vlastní implementací subprocesů – kooperativní plánování. Vůči operačnímu systému vystupují jako jediný proces a jejich plánovače obvykle nezabraňují stárnutí procesů, protože množství a povaha všech procesů je dopředu známa.

Mnohé specializované programy, jejichž rysem je vysoký stupeň paralelizace jako relační databázové stroje, spravují a plánují samy své procesy. Základním předpokladem jejich plánovačů je dominantní zatěžování stroje, na kterém běží. Spustitelných pak udržují jen tolik procesů, kolik je procesorů a systémovému plánovači nedají jinou volbu, než tyto procesy spustit. I bez možnosti úpravy systémového plánovače tak prosadí svou plánovací politiku.

4.4 Plánování procesů nad CLI

Common Language Runtime (implementace CLI) umožňuje kooperativní plánování na úrovni hostování celého virtuálního stroje v rámci procesu operačního systému[6]. Existují plánovače[7], které provádí striktní kontrolu běhu procesů a prevenci jejich zablokování. Program běžící uvnitř ovšem nemá možnost volby vlastního plánovače. Navíc toto řešení není přenositelné na ostatní implementace CLI.

Programové rozhraní CLI nabízí jen omezenou možnost manipulovat

5. Fibers ve Windows a Linuxu, Green Threads v JVM (Java Virtual Machine)
6. Ovládání se provádí pomocí ICorRuntimeHost
7. Microsoft SQL Server 2005

se zásobníkem volání. Navíc přístup k těmto funkcím vyžaduje speciální privilegia[8], která mnohdy správci systému nebo hostující programem neudělují.

Z výše uvedených důvodů byla pro implementaci zvolena cesta úpravy instrukcí zkompilovaného programu – postprodukce programů. Úprava kódu je zcela ve shodě se specifikací CLI, výsledný kód zachovává verifikovatelnost CIL a je plně přenositelný.

4.5 Blokující a přerušitelné funkce

Základním rysem funkcí v modelu zasílání zpráv je, že mohou být blokující. Sekveční zpracování programu se zablokuje ve chvíli, kdy blokující funkce nemůže dokončit svůj výpočet, protože nemůže získat veškerá vstupní data. Funkci budeme nazývat **přerušitelnou**, pokud není pozorovatelné, že se její volání ukončilo, pokud se ekvivalentní blokující funkce zablokuje a není pozorovatelné, že byla opět zavolána ve chvíli, kdy se ekvivalentní blokující funkce odblokuje.

Pro každou přerušitelnou funkci f existuje parciální funkce $r_f : I_f \rightarrow I_f$ kde I_f jsou instrukce funkce f, která vrací instrukci, kde dojde ke znovuspuštění funkce pokud daná instrukce způsobí přerušení.

Cílem přerušitelných funkcí je umožnit psát program, který se jeví jako komunikující procesy a pomocí postprodukce jej transformovat na model konečného automatu. Stavová jednotka bude nutně běžet na prázdném zásobníků volání, nicméně důležité části se z něj nejdříve překopírují do struktur na haldě. Nebude tedy nepodobna ani plánovači procesů.

Přerušitelné funkce bez přímé i nepřímé rekruze lze implementovat postprodukcí binárního kódu.

4.6 Přerušitelné funkce nad CLI

Výsledkem kompilace programovacích jazyků, které mají být spuštěny na virtuálním stroji CLI je CIL. Jedná se o přenositelný jazyk, který musí CIL-to-native[9] kompilátor přeložit do strojového jazyka příslušné architektury. CIL popisuje přechody zásobníkového stroje, jehož chování je ekvivalentní s původně zkompilovaným programem. Existence pouze šesti základních datových typů a absence registrů usnadňuje postprodukci programů.

Aby byla možná postprodukce programů s přerušitelnými funkcemi, je

8. SecurityPermissionFlag.Infrastructure
9. Obvykle Just-In-Time (JIT)

nutné takové funkce rozeznat. Atribut Threading.InterruptibleAttribute slouží právě takovému účelu. Provádění funkce se přeruší ve chvíli, kdy se jí volaná přerušitelná funkce přeruší nebo sama způsobí přerušení.

Přerušení jsou způsobena voláním služeb v třídě Threading.Interrupt (výpis 4.4). Služby samotné žádný kód neobsahují, slouží pouze jako značky pro postprodukci. V místě volání služby Now se začne s přerušením, pokud je její parametr **true**. V opačném případě se pokračuje sekvenčně dále. Místo volání funkce PreCheckBarrier udává místo v kódu před voláním Now kde se má pozorovatelně začít se znovuspuštěním funkce po návratu z přerušení. Jediným úkolem takto vymezeného bloku kódu je znovu vypočítat parametr pro Now a buď opětovně způsobit přerušení nebo pokračovat dále.

Výpis kódu 4.4: Služby nutné ke způsobení přerušení

```
1 public sealed class Interrupt {
2   public static void Now(bool interrupt);
3   public static void PreCheckBarrier();
4   public static void ResumeBarrier();
5 }
```

Při znovuspouštění přerušené funkce f mohou nastat dva případy. V prvním případě funkce f způsobila přerušení přímo voláním služby Interrupt.Now instrukcí $a \in I_f$. Pak znovuspuštění začíná instrukcí $r_f(a)$. V druhém případě přerušení způsobila funkce g volaná z těla f instrukcí b. Znovuspuštění začíná instrukcí $r_f(b)$ a úkolem kódu před dosažením instrukce b je předat parametry funkci g, která bude následně také znovuspuštěna. Funkce g musí být volána s identickými hodnotami parametrů, jinak by byla porušena podmínka nezpozorovatelnosti přerušení. Nutně tedy musí být blok kódu vykonaný mezi instrukcemi $r(b)$ a b deterministický a bez vedlejších efektů. Protože funkce r_f se počítá během postprodukce, tak může programátor vložit volání služby Interrupt.ResumeBarrier, aby vyznačil místo, před kterým dochází k nedeterministickým výpočtům, nebo tvorbě vedlejších efektů. V případě, že by toto volání bylo mezi instrukcemi $r(b)$ a b, posptrodukce ohlásí chybu.

Dále je nutné zamezit ztrátě hodnot lokálních proměnných L_f přerušitelné funkce f. Nechť C je množina instrukcí, které mohou způsobit přerušení a R je množina instrukcí ldloc.a, které čtou lokální proměnnou a. Pak množina $S_f \subseteq L_f$ všech proměnných, které musí být uloženy před přerušením, obsahuje $s \in S_f$ právě tehdy, když existuje instrukce ldloc.s

$\in R$, která je dosažitelná z $r_f(C)$, aniž by proběhla instrukce stloc.s uložení hodnoty do proměnné[10]. Lokální proměnné $L_f \setminus S_f$ není nutné ukládat, protože ztráta jejich hodnoty není zpozorovatelná.

Během postprodukce je pro každou přerušitelnou funkci f vytvořen nový datový typ zvaný **stavová struktura**[11], který obsahuje všechny původně lokální proměnné S_f. Funkci f je přidán nový parametr nově vytvořeného typu, který je předávaný odkazem, a pro všechna $s \in S_f$ jsou všechny instrukce ldloc.s a stloc.s nahrazeny čtením a zápisem z nového parametru.

Stavová struktura kromě S_f obsahuje i stavové struktury všech přímo volaných přerušitelných funkcí z funkce f. Ukazatele na tyto proměnné jsou pak předávány v rámci nového parametru. Protože je nutné zachovat typovou správnost a verifikovatelnost, není možné použít ekvivalent union z jazyka C. Použití struktur nevyžaduje alokaci paměti při každém volání, ovšem znemožňuje rekurzi přerušitelných funkcí.

Nakonec je nutné při znovuspuštění funkce po přerušení na instrukci a zajistit skok na $r_f(a)$. Indikace příslušného $r_f(a)$ je uložena ve stavové struktuře a na začátek přerušitelné funkce je přidána instrukce **switch**, která obsahuje tabulku skoků do všech $r_f(I_f)$.

4.7 Alokace stavových struktur

Stavovou strukturu přijme přerušitelná funkce jako odkaz posledním ze svých parametrů. Ta obsahuje i stavové struktury všech potenciálně volaných přerušitelných funkcí. Kdo tedy alokuje tuto pamět? Zdálo by se, že nepřerušitelná funkce f, která zavolala první přerušitelnou funkci g. Fakticky ano, ale funkce f nemůže být psaná programátorem, protože v době kompilace není ještě známa stavová struktura. Ta vzniká až při postprodukci, stejně jako funkce f zvaná **brána**.

Jak ale volat funkci, která vzniká až v postprodukci? Kód brány je vložen do funkce, která má název a parametry jako měla přerušitelná funkce před postprodukcí. Stavová struktura je přidána do datového typu, ve kterém je přerušitelná funkce obsažena. Její alokace probíhá jako součást alokace instance tohoto typu. Protože stavová struktura může být značně velká, postprodukce implicitně bránu nevytváří, pouze na požádání pomocí atributu Threading.InterruptibleAttribute(**true**).

---

10. Modifikace problému známeho jako Definite Assignment
11. Struktury nejsou alokovány samostatně, ale jsou celé obsaženy buď v mateřském typu nebo na zásobníku stejně jako v jazyku C

V relacích jazyka C by bylo správné namítnout, že stavová struktura nám ztíží nebo téměř znemožní zpětnou kompatibilitu přerušitelných funkcí. Přidání či odstranění volání jiné přerušitelné funkce povede ke změně stavové struktury. Obdobně lokální proměnné, u kterých dokonce stačí, aby se změnila dosažitelnost jejich čtení z místa přerušení. Jazyk CIL nepoužívá offset k přístupu k proměnným struktury, ale metadat. Proto ani velikost struktury není známa až do doby kompilace do strojového kódu, kdy je k dispozici CIL ze všech knihoven. Postprodukce neovlivňuje zpětnou kompatibilitu přerušitelných funkcí.

4.8 Výsledná implementace a její omezení

Program s implementací postprodukce se jmenuje interruptible.exe. Pro práci s binárními soubory byla použita knihovna Mono.Cecil z projektu Mono. Při jejím použití docházelo ke dvěma typům problémů: úplná absence dokumentace a vyšší chybovost v oblasti nové technologie zvané generics. Další sada problémů souvisela s úpravou ladicích informací pro CLR uložených v PDB souborech. Klíčová funkce ISymUnmanagedWriter.DefineLocalVariable nesprávně zpracovává datové typy proměnných. Naštěstí v poslední verzi CLR byla přidána nová funkce s obdobným chováním ISymUnmanagedWriter2.DefineLocalVariable2. Ladění programů nad CLR je tedy bezproblémové. Na ostatních implementacích CLI není ladění možné, protože při postprodukci dojde k přidání instrukcí do různých míst těla funkce. Dojde tedy ke změně offsetu instrukcí, který se počítá od počátku příslušné funkce. Následně se ladící informace pro přerušitelné funkce stanou neplatnými.

Vlastní plánovače kooperativních procesů, které využívají přerušitelné funkce, jsou popsány v kapitole 5, kde je zadefinována speciální podoba procesů.

V úhrnu klade postprodukce přerušitelných funkcí následující omezení na jejich tvar a použití.

Veškerá možná posloupnost volání musí být známá v době kompilace. Proto rekurzivní volání (ať již přímé nebo nepřímé) přerušitelných funkcí nejsou možná. Ze stejného důvodu přerušitelné funkce nemohou být virtuální, protože cíl volání funkce musí být známý.

Současná implementace dále nedovoluje, aby místo volání přerušitelné funkce bylo v rámci bloku, který zpracovává výjimky[12]. Vlastní vyvolá-

12. Platí pro **try** ... **catch** i **try** ... **finally**

vání[13] výjimek není omezeno. Naopak je možné automaticky serializovat úplný stav procesu na disk, pokud program neobsahuje otevřené soubory (příp. síťová připojení). V jejich případě je nutná spolupráce programu. Obnovení běhu programu je následně možné i na jiné architektuře, než na které byl původně spuštěn. Podmínkou je pouze kompatibilní implementace virtuálního stroje CLI.

13. throw

Kapitola 5

Proudové zpracování dat

V této kapitole bude formalizován pojem proudového zpracování dat a bude popsána vlastní implementace běhového prostředí pro proudové úlohy nad platformou CLI.

5.1 Proudové zpracování a operátory

Klasické výpočetní modely odvozené od RAM (Random Access Machine) nekladou žádná omezení na přístup k datům v paměti. Proto program a jeho data musí být umístěn a spuštěn na právě jednom stroji implementujícím klasický výpočetní model. Jedinou cestou jak zkrátit dobu výpočtu, je urychlit běh výpočetního stroje. Vzhledem k tomu, že urychlování strojů naráží jak na fyzikální, tak ekonomické limity, zkracování výpočetního času je nutné docílit použitím více strojů – paralelizací.

Aby běh paralelizovaného programu skončil vždy stejným a korektním výsledkem, je nutné zavést do výpočetního modelu nová omezení. Ta mohou, ale nemusí zachovat ekvivalentní sílu výpočetního modelu.

Jednou z možností, jak zrychlit běh výpočetního stroje, je omezit náhodný přístup programu do paměti. V případě, že omezení bude v časové rovině, mluvíme o **paměťově slabých modelech**. Pokud klademe omezení na vzdálenost po sobě následujících přístupů do paměti, jedná se o **paměťovou lokalitu**.

Definice 5.1.1 *Nechť P je program, P_I množina instancí problému počítaného programem P a $M_r(i \in P_I) = (a_0, a_1, \ldots, a_n)$ jsou paměťové adresy postupně čtené při výpočtu programu P nad instancí problému i. Obdobně M_w pro zápisy. Pokud existuje konečné $w_r \in \mathbb{N}$ takové, že $\forall i \in P_I : \forall a, b \in \mathbb{N} : a \leq b : M_r(i)[b] - M_r(i)[a] \geq -w_r$ (a obdobně pro zápis w_w), pak P splňuje model **proudového zpracování dat**.*

Model proudového zpracování dat nařizuje programům dopřednou pa-

měťovou lokalitu s konstantním omezením vracení se. Povoluje přístup pouze v rámci **datového okna** pro čtení w_r, respektive zápis w_w. V praktickém případě je nutné, aby datové okno bylo menší než operační paměť výpočetního stroje. Obdobně množina P_I by měla obsahovat nekonečně velké vstupy. V opačném případě by datové okno mohlo mít až velikost nejdelšího vstupu.

Proudové zpracování dat samo o sobě neumožňuje paralelizaci běhu programu P, nicméně pokud je program P součástí větší struktury Q vstupně/výstupně provázaných programů $P_1, \ldots, P_n$, kde každý splňuje model proudového zpracování dat, pak lze Q spustit až na n strojích současně. Mezi stroji pak dochází k přenosu výstupních dat ihned poté, co pro producentský program opustí datové okno. Navíc program Q lze efektivně spustit i na méně strojích, pokud pro každý stroj platí, že součet datových oken všech na něm spuštěných programů, nepřesahuje dostupnou operační paměť.

Kromě toho struktury vstupně/výstupně provázaných programů proudového zpracování dat explicitně vyjadřují nejvyšší úroveň abstrakce výpočtu. Ta může dále sloužit plánovači k odhadu celkové výpočetní náročnosti a případně postupu běžícího výpočtu na základě množství dat, které opustí datové okna programů.

V praxi používanými programy proudového zpracování dat je například zpracování multimédií, kde datovým oknem je posloupnost několika málo obrazů u videa nebo vzorků u audia, nebo relační databáze, kde datovým oknem je n-tice.

Tvrzení 5.1.1 *Nechť program P řeší problém s instancemi I_P a splňuje model proudového zpracování dat, pak existuje algoritmus Q, který řeší problém s instancemi I_Q, kde $I_Q = 2^{I_P}$, a splňuje model proudového zpracování dat.*

Důkaz Opakovaným spouštěním programu P pro každé $i \in I_Q$.

Problémy se strukturou zavedenou v tvrzení 5.1.1 umožňují další zvýšení paralelizace, protože opakované spouštění podproblému může probíhat souběžně na několika strojích.

V praxi se opět jedná o zpracování multimédií, kde se několik stovek obrazů může zpracovávat zcela nezávisle. I na úrovni relačních databází dochází při některých plánech k paralelizaci v rámci jednoho dotazu, pokud plánovač dokáže určit distribuční kritérium pro n-tice vzhledem k následujícím operacím.

Definice 5.1.2 *Nechť P je program, který splňuje model proudového zpracování dat. Dále nechť I_P i O_P jsou množiny programů, které splňují model proudového zpracování dat. Pak P nazýváme* ***proudovým operátorem*** *pokud svá vstupní data přijímá z výstupů programů I_P a zasílá svá výstupní data programům O_P.*

Proudový operátor lze chápat jako uzel orientovaného grafu, kde hrany reprezentují datové toky.

5.2 Paměťová lokalita

Jedním z důsledků růstu objemu operačních pamětí v soudobých počítačích je rostoucí rozdíl mezi rychlostí náhodného přístupu do paměti a rychlostí procesoru. To nutí výrobce přidávat mezi procesor a operační paměť několik úrovní vyrovnávacích pamětí s rostoucí velikostí, ale i rostoucí dobou odezvy. Adresově zůstává sice operační paměť homogenní, ale doba přístupu nikoliv. Dalším krokem jsou systémy, které poskytují programům informace o nehomogenní struktuře své paměti, s názvem NUMA (Non-Uniform Memory Access). Pokud chce program dosáhnout maximální efektivity u systémů NUMA, pak už v době alokace paměti musí zvažovat poměr mezi rychlostí a množstvím.

Při proudovém zpracování dat pracuje program s pamětí vstupu i výstupu téměř sekvenčně. Výjimkou je relativně malé datové okno, které umožňuje návrat zpět. Takové schéma přístupu je z pohledu vyrovnávacích pamětí procesoru ideální, protože veškerá spekulativně získaná data z operační paměti jsou použita.

5.3 Struktura předávaných dat

Data mezi proudovými operátory je vhodné strukturovat do podoby pole. Operátor do něj nebude mít zcela náhodný přístup, ale jen v rámci svého datového okna. Nejvyšší dimenze pole pak reprezentuje homogenní bloky dat (např. celé číslo, data jednoho obrázku, textový řetězec). Pro různé operátory může homogenita znamenat něco jiného. Relační operátor bude pracovat se strukturou tří dimenzí soubor(relace)/n-tice/atribut zatímco operátor, který kopíruje soubor, vystačí s jedinou dimenzí soubor.

Struktura navíc umožní nespouštět operátor pro každou instancí problému zvlášť, ale pouze pomocí struktury vyjádřit počátek nové instance. Implementace operátorů pak implicitně řeší podmnožinu množiny všech problémů v rámci jediného běhu popsaného v tvrzení 5.1.1. Zamezí se tak

ztrátě výkonu ve chvíli, kdy by režie spojená se zastavením operátoru, vytvořením a spuštěním nového byla srovnatelná s výpočetní náročností vlastního problému. Například operátor kopírování souborů bude stejně efektivní pro malé i velké soubory, protože data malých souborů budou při zpracování v paměti následovat ihned za sebou.

Výhodná je i výpočetní úspora, protože každý operátor nebude muset složitě analyzovat vstup. To bude práce pro specializované operátory na počátku. Příkladem je načtení relace ve formátu CSV[1]. Všechny další relační operátory už budou pracovat s dohodnutým významem dimenzí.

5.4 Grid Streaming Runtime

Programování proudových úloh s sebou nese několik obtíží. Tou první je efektivní výměna dat, jejímž úkolem je zpřístupňovat paměť jako souvislý blok paměti a současně jako oddělené kusy, které lze předat následujícímu programu (konzumentovi). Výměna dat by měla umožnit předávání velkých částí vstupu přímo na výstup bez nutnosti kopírování. Cena za kopírování 1 MB, pokud mineme vyrovnávací paměť, se pohybuje až v řádu desítek milisekund. Pokud trefíme vyrovnávací paměť, tak o řád rychleji – v řádu milisekund. Nakonec by měla umožnit definovat na proudu dat strukturu v podobě pole.

Druhou obtíží při programování proudových úloh je přepínání mezi proudovými operátory. Pokud se operátory implementují jako VPKA (viz 4.1), obsahují minimálně čtyři stavy[2] pro každý vstup. Programování operátorů je pak únavné, zhoršuje čitelnost kódu a je náchylné k chybám. Navíc je problémem zavést jednotnou politiku pro přepínání mezi proudovými operátory, protože každá implementace bude považovat různé množství zpracovaných dat (příp. uběhnutého času) za vhodné k přepnutí.

Vlastní implementace pro CLI, nazvaná Grid Streaming Runtime (GSR), využívá modelu zasílání zpráv, kde každý proudový operátor běží ve vlastním procesu. Procesy jsou řízeny vlastním plánovačem, který určuje intervaly přepínání i s ohledem na paměťovou lokalitu. Kód implementace je umístěn v knihovně IO.dll a programu gsr.exe. Zde je nutné zdůraznit, že termín proces označuje sekvenční zpracování a nikoliv proces operačního systému. Jediným procesem OS je jediná instance gsr.exe, ve které běží veškeré proudové operátory.

---

1. Comma Separated Values
2. začátek nové instance, potřebná data nedočtena, potřebná data dočtena, potřebná data dočtena a poslední instance

5.4.1 Výměna dat

Proudové operátory zpracovávají velká množství dat. V každém okamžiku mohou sice přistupovat pouze k datům v rámci datového okna, ale celkové množství dat vyměněných mezi jednotlivými operátory zůstává nezměněno.

Výměnu dat je možné realizovat dvěma způsoby. Prvním způsobem je **půjčování paměti**, kdy konzument předá prázdný blok paměti producentovi k naplnění. Konzument však zůstává vlastníkem této paměti. Vlastnictvím paměti rozumíme povinnost jejího uvolnění. Při druhém způsobu, **předávání paměti**, naopak producent předá konzumentovi již naplněný blok paměti a spolu s ním i její vlastnictví.

Výhodou půjčování paměti je snadnost implementace, protože vlastnictví paměti se nemění a vlastníkem je přímo každý konzument. Nevýhodou je naopak nutnost neustálého kopírování paměti v každém producentovi. Producent by sice mohl půjčenou paměť půjčit dále ve vztahu kde je konzumentem (tranzitivní půjčení), ale až na triviální případy, kdy poměr vstupu a výstupu je jedna ku jedné, takové chování vyžaduje mnoho přepínání mezi producentem a konzumentem k naplnění celé paměti. Případ produkce identických dat na více než jeden výstup nevyřeší ani tranzitivní půjčování.

Předáváním paměti lze řešit všechny nedostatky uvedené u půjčování paměti. Cenou za tuto výhodu je složitější implementace. Většinu obtíží lze však izolovat do množiny funkcí nazvaných **správa paměti** proudových operátorů.

Potenciální nevýhodou obou způsobů výměny dat je nejednotnost velikostí paměťových bloků a možnost volby nějaké „vhodné“ velikosti každým konzumentem nebo producentem. Ve výsledku tak každý producent nebo konzument musí řešit možnou změnu velikostí bloků během své existence.

5.4.2 Správa paměti

Úkolem správy paměti je zbavit konzumenty a producenty problémů s vlastnictvím paměti. Proto základem implementace správy paměti je třída IO.DataHandle<T> (výpis 5.1), která reprezentuje vlastnictví paměti, která je v něm umístěna – držadlo[3]. Při návrhu proudového operátoru je většinou možné dopředu určit potřebný počet držadel. Ty jsou pak vytvořeny spolu s operátorem, spolu s ním zanikají, a jejich alokace a uvolnění není problém.

Na povrchu musí správa paměti nabízet data jako souvislý blok paměti.

3. handle

Výpis kódu 5.1: Rozhraní přístupu k datovým stránkám

```
1  public sealed class DataHandle<T> {
2    public void Add(DataHandle<T> data)
3    public void Add(DataHandle<T> data, int offset, int
     count)
4    public void Clear();
5    public DataCollection Data { get; }
6    public BufferCollection Buffers { get; }
7
8    public sealed class BufferCollection : ICollection,
     IEnumerable<BufferRun<T>> { }
9
10   public sealed class DataCollection : ICollection,
     IEnumerable<T> { }
11 }
```

Uvnitř však pracuje s bloky paměti pevné délky nepřesahující 64K položek[4] zvané **stránky**. Stránky jsou reprezentovány třídou IO.DataPage<T>, která obsahuje čítač odkazů[5], který vyjadřuje počet uživatelů této stránky – sdílené vlastnictví.

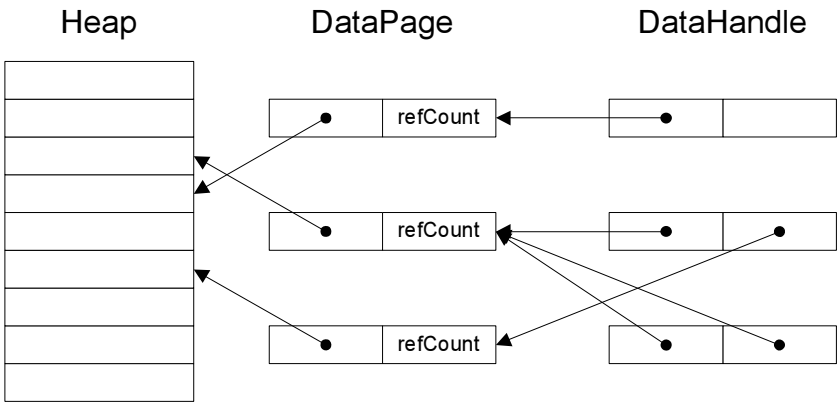
Velikost stránek má několik důsledků. Za prvé, paměť je předávána mezi producentem a konzumentem pouze po stránkách. Ta sice nemusí být plná, ale obvykle tomu tak je. Její velikost přímo určuje minimální interval mezi přepnutím operátorů. Za druhé, dříve či později je paměť předána nějaké I/O službě operačního systému, aby ji uložil na disk nebo zaslal po síti. Při tomto předání může dojít ke dvěma situacím. Paměť je příliš malá nebo fragmentovaná, a její obsah je nejdříve překopírován do paměti uvnitř jádra, nebo je dostatečně velká, plná dat a je pomocí virtuální paměti namapována na adresy určené pro jádro. V druhém případě se zamezí zbytečnému kopírování, tzv. zero-copy. Operační systémy chápou velkou paměť obvykle od několika desítek KB.

Pro maximální efektivnost je vhodné obejít automatickou správu paměti v CLI. Pro velké bloky paměti je méně efektivní a spotřeba stránek v proudových operátorech je velká. Z těchto důvodů existuje třída IO.DataPool <T>, která recykluje již nepoužívané stránky ve chvíli, kdy čítač odkazů v DataPage klesne na nulu.

Komplikovanost vlastnictví paměti je ve třídě DataPage – v čítači odkazů. Z tohoto důvodu je DataPage neveřejná, a veškeré změny vlastnictví vznikají

4. v případě datového typu **byte** 64 KB, pro **char** 128 KB
5. reference counting

Obrázek 5.1: Ukazatelé mezi objekty správy paměti



jako vedlejší operace při volání funkcí v DataHandle. Pokud by byla tato povinnost přenechána na programech jednotlivých operátorů, hrozilo by, že se nejenom stránky nebudou recyklovat včas, ale naopak příliš brzy. Tím by došlo k poškození dat. Základní paměťovou jednotkou je tzv. **datový běh**[6], který představuje posloupnost položek v rámci jedné stránky. Jeho délka je tedy shora omezena délkou stránky. DataHandle obsahuje posloupnost datových běhů a stará se o správné počítání odkazů příslušných stránek. Příklad vztahu mezi instancemi objektů správy paměti je uveden na obrázku 5.1.

Vlastní výměna dat mezi proudovými operátory je rozdělena mezi tři třídy: IO.ArrayWriter<T>, IO.ArrayExchange a IO.ArrayReader<T>, kde T je typ předávaných dat (obvykle **byte** nebo **char**). Každý operátor vlastní ArrayWriter pro každý svůj výstup a obdobně ArrayReader pro každý svůj vstup. Třída ArrayExchange pak zajišťuje předávání dat mezi výstupy a vstupy. Jako jediná je implementována jako thread-safe, protože dva propojené operátory mohou běžet současně, každý na různém vlákně operačního systému[7].

Úkolem třídy IO.ArrayExchange je předávat datové bloky a strukturu dat od producenta ke konzumentovi. Aby se omezilo blokování mezi producentem a konzumentem, ale udržela konzistence a snadnost čtení, předávají se stránky i struktura v cyklických frontách pouze na explicitní žádost. Funkce IO.ArrayRWExchange<T>.PublishWrites() zveřejní nová data zapsaná přes ArrayWriter. Naopak IO.ArrayRWExchange<T>.ReadPublished()

6. data run
7. jeden operátor vždy běží na maximálně jednom vlákně

zpřístupní zveřejněná data ke čtení.

Přístup k vlastním datům realizují proudové operátory přes už zmíněnou třídu DataHandle. Ta umožňuje jak souvislý přístup po jednotlivých položkách[8], který zakryje existenci datových běhů, tak na úrovni datových běhů[9]. Druhý přístup je vhodnější, pokud operátor předává data další funkci f, která má parametry f(T[] buffer, **int** offset, **int** count). Takových funkcí je v CLI značné množství, zvláště v System.IO. Každopádně se musí v druhém případě postarat o potenciální zalomení dat na úrovni datových běhů.

Jednotná správa paměti určuje velikost datových stránek pro každý datový typ a během zpracování se tato velikost nemění. Protože proudové operátory žádnou paměť, kterou chtějí vyměňovat, nealokují, je možné změnit strategii alokace datových stránek i v budoucnu, bez nutnosti změn programů jednotlivých operátorů. Příkladem může být zohlednění NUMA systémů.

5.4.3 Operátory

Proudové operátory musí implementovat minimálně rozhraní IO.Processing.IIoOperator (výpis 5.2). Předpokládá se, že funkce Dispatch bude přerušitelná (4.6), proto je návratová hodnota sémanticky kompatibilní s bránou přerušitelné funkce.

Výpis kódu 5.2: Povinné rozhraní proudového operátoru

```
1 public interface IIoOperator {
2   bool Dispatch();
3 }
```

Typický operátor se skládá z vnořených cyklů, které odpovídají struktuře vstupních dat. Výstupní strukturu definuje voláním funkcí BeginRank a EndRank. Operátor na výpisu 5.3 čte vstupní data z jediného vstupu s hierarchickou strukturou soubory/n-tice/atributy[10] a na výstupu eliminuje úroveň souboru. Cykly na řádcích 5, 6 a 8 navozují dojem iterace přes třídimenzionální pole, které je celé k dispozici v paměti. Ve skutečnosti tento operátor vyžaduje v každý okamžik hodnotu jen jednoho atributu (řádek 10). Jedná se o datové okno proudového zpracování dat. Operátor obsahuje

8. IO.DataHandle<T>.DataCollection
9. IO.DataHandle<T>.BufferCollection
10. obecná reprezentace relace předchozím operátorem načtená například z DBF nebo CSV formátu

pět volání přerušitelných funkcí. Na řádcích 5, 6 a 8 jsou to MoveNext skryté za syntaxí **foreach** a na řádcích 10 a 11 čtení a zápis dat.

Další ukázkový operátor (výpis 5.4) očekává dva vstupy stejné délky v první dimenzi, dimenze n-tic. Na výstup pak zapisuje sjednocení atributů z obou vstupních n-tic. Ve chvíli, kdy má operátor více vstupů, je nutné nahradit syntaktickou zkratku **foreach** jazyka C# obecným cyklem (řádky 5 až 8). Zde jsou již na řádku 8 viditelná volání MoveNext, u kterých je dobré si povšimnout, že k přerušení může dojít u každého z nich. Pokud by přerušení nastalo u volání tuples1.MoveNext(), pak při znovuspuštění funkce se tímto voláním opět začne – nikoliv voláním tuples0.MoveNext(). V opačném případě by iterátor prvního vstupu chybně postoupil k další n-tici.

5.4.4 Plánovač proudových operátorů

Úkolem plánovače proudových operátorů je volat IIoOperator.Dispatch jednotlivých operátorů. Volání této funkce skončí vždy, když se operátor dobrovolně (kooperativně) přeruší. Jediným nástrojem plánovače na ovlivnění délky dílčích běhů je omezení počtu stránek na straně producentů. Při překročení limitu je operátor přerušen. Toto přerušení však může nastat dříve, než producent zapsal, z pohledu konzumenta, alespoň elementární jednotku dat, kterou je schopen číst. Plánovač musí sledovat zda konzument pokročil ve výpočtu a v opačném případě přikročí ke zvětšení limitu počtu výstupních stránek.

Kromě výběru operátoru, který má dále běžet, poté co se současně běžící operátor dobrovolně přerušil, musí plánovač rozhodnout o mapování procesů-operátorů na vlákna operačního systému. Vlákna jsou v CLI reprezentována pomocí System.Threading.Thread, které CLI mapuje na vlákna operačního systému. Pro plné vytížení stroje musí plánovač použít až tolik vláken, kolik má stroj procesorů. Další zvyšování počtu vláken vede spíše k degradaci, protože systémový plánovač dostane možnost volby a jeho přepínání vláken bude stát dodatečnou režii.

Je vhodné, aby plánovač proudových operátorů implementoval rozhraní IIoScheduler (výpis 5.5) s jedinou funkcí Execute, která spustí plánovač nad daným grafem precedencí proudových operátorů.

V rámci práce byly implementovány dva plánovače. První se jmenuje ChainScheduler a umí spouštět pouze graf precedencí $G = (V, E)$ ve formě jediného lineárního řetězce. Platí tedy, že $|V| = |E| + 1$ a graf je souvislý. Operátory jsou spouštěny na jediném vlákně, na vlákně volající funkci IIoScheduler.Execute.

Výpis kódu 5.3: Operátor upravující pouze strukturu dat

```
1  [Interruptible(true)]
2  public bool Dispatch() {
3    DataHandle<byte> data = new DataHandle<byte>();
4
5    foreach (Rank<byte> file in input.Root) {
6      foreach (Rank<byte> tuple in file) {
7        output.BeginRank();
8        foreach (Rank<byte> attribute in tuple) {
9          output.BeginRank();
10         attribute.GetData(data);
11         output.Write(data);
12         output.EndRank();
13         data.Clear();
14       }
15       output.EndRank();
16     }
17   }
18 }
```

Výpis kódu 5.4: Operátor s více vstupy

```
1  [Interruptible(true)]
2  public bool Dispatch() {
3    DataHandle<byte> data = new DataHandle<byte>();
4
5    for (ArrayReader<byte>.RankEnumerator
6      tuples0 = input0.Root.GetEnumerator(),
7      tuples1 = input1.Root.GetEnumerator();
8      tuples0.MoveNext() && tuples1.MoveNext(); ) {
9      output.BeginRank();
10     for (int i = 0; i < 2; i++)
11       Rank<byte> t = (i == 0 ? tuples0 : tuples1).Current
         ;
12       foreach (Rank<byte> attribute in t) {
13         output.BeginRank();
14         attribute.GetData(data);
15         output.Write(data);
16         output.EndRank();
17         data.Clear();
18       }
19     output.EndRank();
20   }
21 }
```

Výpis kódu 5.5: Doporučené rozhraní plánovače proudových operátorů

```
1 public interface IIoScheduler<TNode, TEdge> {
2   void Execute(PrecedenceGraph<TNode, TEdge> precedence);
3 }
```

ChainScheduler[11] se hodí ke spouštění úloh, které lze vytvořit zřetězením několika operací a bez požadavků na paralelizaci. Plánovač používá pouze dvě datové stránky k výměně dat mezi producentem a konzumentem. Výhodou je pak malá spotřeba paměti celé úlohy, protože dochází k častému přerušování jednotlivých operátorů. Hodí se i k ladění operátorů, protože je jeho průběh z důvodu běhu na jediném vláknu deterministický.

Druhý implementovaný plánovač je WeightedScheduler[12], který dokáže plánovat libovolný graf precedencí. Jako vstup je dále ke každému operátoru $o \in O$ přidělena váha $w_o \in \mathbb{R}$, která určuje podíl výpočetního výkonu stroje, který má operátor o v průměru získat. Pro váhy platí $\sum_{o \in O} w_o = 1$. Plánovač dále očekává, že bude jediný, kdo významně zatěžuje stroj, na kterém běží. Pro každý procesor totiž vytvoří jedno vlákno, které bude provádět plánování, a očekává, že plánovač operačního systému nebude mít jinou možnost, než každé vlákno přidělit na jednotlivé procesory paralelně. Použit byl algoritmus Weighted Round Robin, který má složitost $O(\log n)$, kde n je počet operátorů (procesů).

Kód plánovače je napsán thread-safe, aby nedošlo k poškození dat nutných k plánování jednotlivými vlákny plánovače. Vlastní proudové operátory takový požadavek u IIoOperator.Dispatch záměrně nemají, aby se zjednodušilo jejich programování. Jeden operátor tak nemůže být volán paralelně. K dosažení maximálního výkonu je tedy nutné alespoň tolik operátorů, kolik je procesorů.

11. v knihovně IO.dll
12. v programu gsr.exe

Kapitola 6

Plánování gridu jako problém LP

Požadavky a limity distribuovaného plánovače a prostředí, ve kterém funguje, byly uvedeny v předchozích kapitolách. Nyní následuje návrh plánovače pro specifickou kategorii programů.

Výpočetním modelem gridu byly zvoleny rourou spojené procesy (1.2.3), protože umožňují velkou úroveň paralelismu, a tím umožňují maximálně zatížit grid i při menším počtu úloh, a ve výsledku zpracují jednotlivé úlohy v krátké době. Dále implementace jednotlivých operací pracují jen nad oknem vstupních a výstupních dat malé velikosti, a mají i samotné paměťovou složitost v polynomiální třídě. Tato vlastnost je velmi důležitá, protože umožní souběžné spuštění většího množství procesů (operací) na jednom procesoru (3.3). Navržený plánovač bude pracovat nad Portable Batch System (PBS) – správcem gridu. Oproti původnímu záměru není spolupráce s jinými plánovači PBS, které implementují volně spojené procesy (1.2.1), možná.

Většina současných gridových plánovačů neřeší transport vstupních a výstupních dat, případně mezivýsledků, na stroje, kde bude spuštěn proces, který je zpracovává. Grid je z definice distribuovaný systém, kde nelze předpokládat, že datové spoje budou vždy dostatečné. Navíc úloha, jejímž úkolem bude pouze transport ze zdroje(ů) k cíli je naprosto legální a s rostoucím rozměrem gridů i častější. Z tohoto důvodu musí být i přenosové kapacity předmětem plánování.

Množina výpočetních uzlů není nijak omezena, zvláště pak homogenita procesorů v ní obsažených (1.1). Jejich rychlost může záviset na typu operací. Je tedy nutné předpokládat, že se jedná o nezávislé paralelní stroje (2.1). Na druhou stranu není nutné omezovat jeden výpočetní uzel na maximálně jednu operaci v každý okamžik. Obdobně počítače jsou velmi efektivní ve výměně nástrojů, jedná se o víceúčelové stroje.

Při rourou spojených procesech se obvykle hledá ustálený plán [14, 10] pro jediný globální časový interval. Tím je znemožněno přidat omezení, které vyžaduje dokončení úlohy do daného časového okamžiku (deadline). Při globálním intervalu by došlo k nevyužití výpočetní kapacity po tomto

okamžiku. Tyto výpočetní ztráty nelze nijak ohraničit a vyžadují tedy zavedení několika časových intervalů, v rámci kterých se bude hledat ustálený plán.

6.1 Operace, precedence a operátory

Operace v rourou spojených procesech reprezentuje výpočet ve vstupně/-výstupním vztahu k dalším operacím. Je implementována proudovým operátorem (5.4.3). Vzájemný vztah operací je určen precedencemi (β v 2.1).

$$P = \{(a,b)|a,b \in O) \tag{6.1}$$

$$P_u = \{(a,b)|(a,b) \in P \vee (b,a) \in P) \cup \{(a,a)|a \in O\} \tag{6.2}$$

$$T = O/P_u = \{[o]_{P_u}|o \in O\} \tag{6.3}$$

Množina precedencí P vyjadřuje antisymetrickou tranzitivní relaci nad množinou operací O (6.1). Rozšířením relace P na symetrickou a reflexivní relaci získáme relaci ekvivalence P_u (6.2). Rozkladem množiny operací podle posledně získané relace dostaneme třídy (6.3), které nazýváme **úlohy** T.

Úlohy mají v proudovém zpracování dat význam v tom, že určují operace, které musí být zpracovávány synchronně – výstup producenta je ihned po opuštění datového okna zpracován konzumentem. V opačném případě, při ukládání celých mezivýpočtů, by systém neměl předvídatelné paměťové nároky obdobně jako workflow.

Operace reprezentují specifikaci funkčního vztahu mezi vstupem a výstupem. Při plánování konkrétního stroje ale potřebujeme pracovat i s implementací této operace, operátorem. Zavedeme parciální funkci $Impl$, která vrací operátor (implementaci) dané operace na daném výpočetním uzlu.

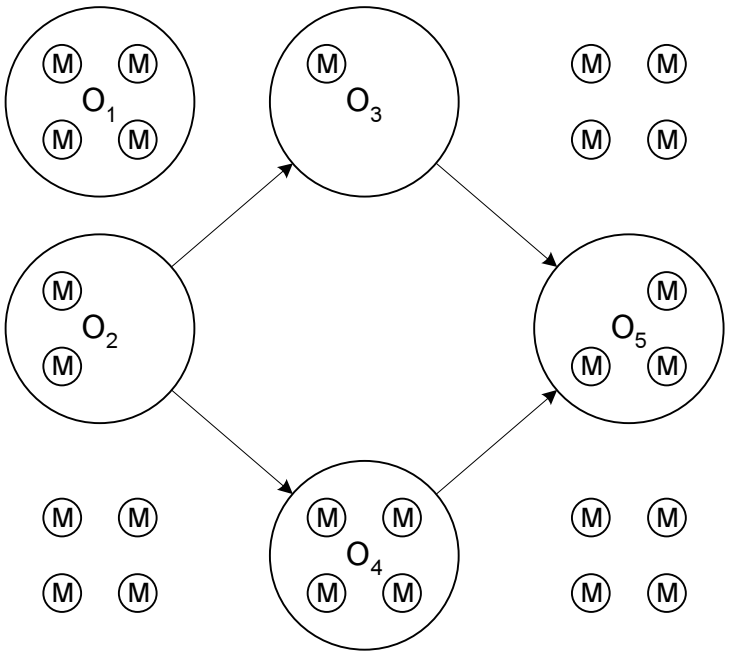
$$Impl : O \times M \rightarrow I$$

Z operátoru získáme koeficient rychlosti implementace pro každý výpočetní uzel, jak je vyžadováno nezávislými paralelními stroji (2.1).

6.2 Výpočetní uzly a procesy

Výpočetní uzel bude dále zjemněn na jednotlivé procesory. To ovšem neznamená, že v reálném systému je vždy nutné členit výpočetní uzly až na jednotlivé procesory. Pouze tím oddělujeme pojmy stroj s jedním operačním systémem (výpočetní uzel) a výpočetní jednotku, která je předmětem plánování – procesor.

Obrázek 6.1: Příklad vztahu operací, operátorů a výpočetních uzlů



Realizací naplánované operace v místě a čase je **proces**[1] (nebo procesy). Místem realizace procesu je výpočetní uzel. Jedna operace ale může být prováděna více procesy současně. Tento fakt umožní provádět paralelizaci programu proudového zpracování dat, jehož instance jsou posloupnosti instancí problémů, řešitelných proudovým zpracováním dat (5.1.1). Pokud má být plán proveditelný, musíme předpokládat, že vstupní instance problému budou dělitelné na dostatečně mnoha místech, aby byl náběh paralelizace zanedbatelný.

Ukázka vztahu mezi operacemi, operátory a výpočetními uzly je uvedena na obrázku 6.1. O značí operace s hranami vyjadřujícími precedence. M uvnitř každé operace jsou výpočetní uzly, které danou operaci počítají formou operátorů. Nevyužité výpočetní uzly jsou vně operací. V příkladu jsou uvedeny dvě úlohy, první se skládá z jediné operace O_1. Druhá z operací O_2 až O_5. V reálném případě budou mnohé uvedené výpočetní uzly reprezentovat identické stroje; více než jeden operátor poběží na jednom uzlu.

Běh procesů v rámci výpočetního uzlu řídí **plánovač procesů**. Výcho-

1. obdobně jako v kapitole 4 budeme procesem rozumět sekvenční zpracování instrukcí

diska implementace jsou diskutována v 4.3 a kapitole 5. Předpokládáme, že plánovač procesů dokáže spouštět procesy tak, aby udržoval zadaný poměr času, které každý proces stráví na procesoru.

6.3 Plánovač úloh a plánovač procesů

Vstupem **plánovače úloh** jsou operace, graf jejich precedencí, funkce implementací $Impl$ a graf výpočetních uzlů. Výstupem jsou operátory s okamžiky jejich spuštění, datové toky mezi výpočetními uzly a poměr běhu operátorů v rámci každého výpočetního uzlu. Tyto informace jsou předány jednotlivým výpočetním uzlům, kde jsou realizovány plánovači procesů. Každý výpočetní uzel má právě jeden plánovač procesů, který pracuje na základě informací od plánovače úloh. Komunikace mezi plánovači procesů není nutná (vzhledem k jejich počtu ani vhodná).

Účelem poměru běhu operátorů v rámci výpočetních uzlů je, aby byl chod proudových operátorů implicitně synchronizovaný. V ideálním případě jejich naplánovaných hodnot i jejich vynucení se data producentů nebudou hromadit před konzumenty, a vynucovat si pozastavení producentů (obdobně z pohledu konzumentů a dostatku vstupních dat). V důsledku by tedy – producenti i konzumenti – maximálně využívali distribuované výpočetní a přenosové kapacity.

Realizaci plánů plánovače úloh, provoz plánovačů procesů a proudových operátorů provádí Grid Streaming Runtime (GSR).

6.4 Paralelizace operací

Jak již bylo uvedeno, předpokládáme dělitelnost vstupních dat každé operace. Jednotlivé díly pak představují data, která lze zpracovat programem bez znalosti dílů předchozích i následujících. Neznamená to však, že pro každý dále nedělitelný díl budeme plánovat a spouštět nový proces. Využijeme pouze faktu, že program bude po zpracování dílu ve výchozím stavu.

Výsledkem plánovače úloh jsou datové toky mezi výpočetními uzly. Ty však nejsou starostí jednotlivých operátorů, ale GSR, který musí zajistit, aby výstupy byly rozděleny a přepraveny právě plánovanými cestami, a následně spojeny před vstupem do operátoru. Rozdělování výstupů zajišťují demultiplexory, spojování multiplexory. Na obrázku 6.2 nahoře je graf precedencí nějaké úlohy. V dolní části pak plán obdržený od plánovače úloh. Obdélníky jsou operace/operátory. Vlevo od každého obdélníku jsou

vstupy, vpravo výstupy. V dolním grafu je vidět, že ke vstupům C_{31} a D_2 vede více datových toků. Obdobně výstupy B_{22} a B_{32}. Do těchto míst musí před spuštěním GSR vložit multiplexory resp. demultiplexory.

Data předávaná mezi operátory jsou strukturovaná jako pole (5.3). Díky tomu demultiplexor nevyžaduje významnější spolupráci s operátorem; stačí mu dimenze při jejímž počátku začne producent zapisovat na vstup novou instanci problému proudového zpracování.

6.5 Lineární problém

6.5.1 Časové intervaly

Při konstrukci plánovacího problému jako problému lineárního programování, je nutné začít s reprezentací času. Čas budeme reprezentovat v intervalech proměnné délky, a tedy proměnnými. Maximální počet intervalů teoreticky nutných pro nalezení optimálního plánu nastiňuje následující tvrzení.

Tvrzení 6.5.1 *Nechť S jsou optimální plány úloh T. Dále nechť F*

$$F : S \times T \to \mathbb{R}$$

udává konec zpracování dané úlohy v daném plánu.

$$P : S \times T \to 2^{\mathbb{R}}$$

P udává veškerá přerušení zpracování dané úlohy v daném plánu, ať již počátek nebo konec přerušení. Pak

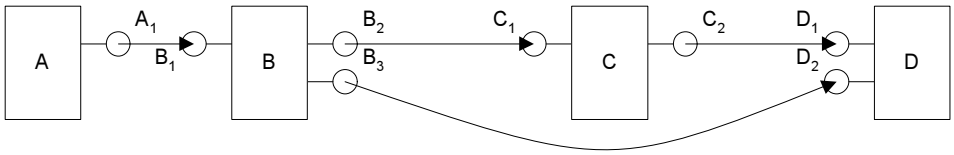
$$\forall a \in \mathbb{R} : \exists t \in T : a \in P(s,t) \Rightarrow (\exists u \in T : a = F(s,u))$$

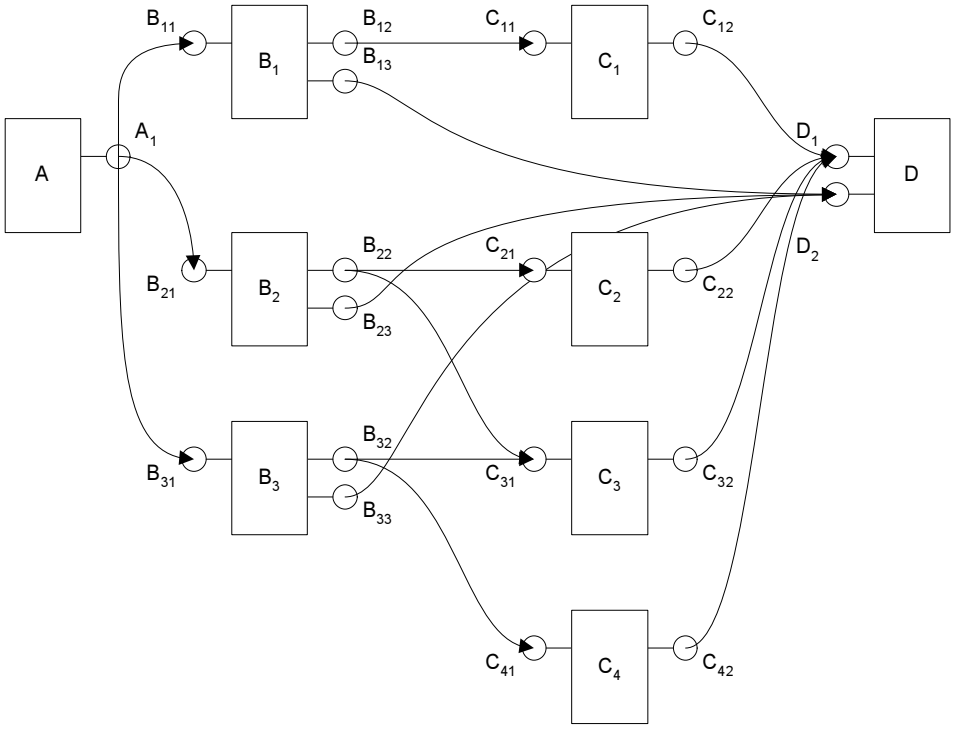
počet intervalů optimálního plánu je shora omezen počtem úloh.

Důkaz Konec každého intervalu musí být způsoben ukončením úlohy. Pokud není, tak nedošlo ke změně prostředí strojů, ukončení intervalu je neopodstatněné a existuje optimální plán bez tohoto přerušení. Počet ukončení úloh je roven počtu úloh.

V současné době lineární model neobsahuje omezení, která by vynucovala dokončení obecného množství úloh do určitého času (deadline), ani obecné množství rezervací od kdy úlohu spustit. Proto se v samotném plánu

Obrázek 6.2: Graf precedencí realizovaný grafem operátorů





omezíme na dva intervaly. První bude interval, jehož plán bude proveden. Druhý interval pojme veškerou nedokončenou práci úloh. Velikost prvního intervalu je předpokládaná doba, po kterou nedojde k přidání nové úlohy do systému, která by vyžadovala okamžité zpracování, a nenastane výpadek žádného výpočetního uzlu.

6.5.2 Konstanty – informace o operacích a procesorech

Pro naplánování běhu potřebujeme několik informací o operacích a jejich vstupech a výstupech. Nechť M je množina procesorů a O operací.

$WorkCPU:$	$O \to \mathbb{R}$	celková práce operace	(6.4)
$WorkIO:$	$O \times O \times M \to \mathbb{R}$	práce k vyprodukování výstupu	(6.5)
$Runable:$	$O \times M \to \{0, 1\}$	může běžet na procesoru	(6.6)
$Capacity:$	$M \times M \to \mathbb{R}$	přenosové kapacity	(6.7)
$Io:$	$M \times M \to \mathbb{R}$	výpočetní náročnost přenosů	(6.8)
$CPU:$	$M \to \mathbb{R}$	rychlost procesoru	(6.9)

$WorkCPU$ je množství práce v instrukcích nutných k výpočtu všech plánovaných dat dané operace. Pro zjištění vyprodukovaných dat v průběhu výpočtu slouží koeficient $WorkIO(f, g)$, který udává množství vyprodukovaných dat (v bytech) konzumentovi g (konzument g udává výstup operace f) za jednu instrukci výpočtu f. Vztažením k procesoru tato funkce vyjadřuje rychlost implementace dané operace přidělené funkcí $Impl$, pokud existuje.

$Runable$ vyjadřuje prostou definovanost funkce $Impl$ pro danou operaci a procesor. $Capacity$ je množství bitů, které mohou být přeneseny ze zdrojového procesoru k cílovému procesoru. Hodnoty nemusí být symetrické vzhledem k dvojici procesorů.

Io udává, kolik instrukcí v průměru stojí přenesení jednoho bitu z daného procesoru na daný procesor. Tím plánovač omezí alespoň přibližně zpracování čistých výpočtů a zamezí přetížení procesoru. Důležité zejména pro úlohy s minimem zpracování dat; v extrému pouze přenos dat. CPU určuje průměrný počet instrukcí procesoru za sekundu.

6.5.3 Proměnné

Nechť M je množina procesorů, O operací a $I = \{1, 2\}$ intervalů.

$\forall i \in I:$	s_i	délka časových intervalů	(6.10)

$$\forall o \in O, m \in M, i \in I: \quad u_{o,m,i} \quad \text{zatížení procesoru v \%} \tag{6.11}$$

$$\forall f, g \in O, p, q \in M, i \in I: \quad t_{f,g,p,q,i} \quad \text{množství přenosů} \tag{6.12}$$

$$tabs_{f,g,p,q,i} \tag{6.13}$$

Délka intervalu s_i je v sekundách. Zatížení daného procesoru v daném okamžiku počítáním dané operace je v procentech maximálního výkonu. Nakonec $t_{f,g,p,q,i} - tabs_{f,g,p,q,i}$ vyjadřuje přenos dat mezi procesory p a q, kde na p běží producent f a na q konzument g. Veškeré proměnné v LP jsou nezáporné, proto pro datový tok potřebujeme dvě proměnné v uvedeném vztahu pro pokrytí celého intervalu reálných čísel. Přenosová proměnná může být záporná, protože záměna f za g efektivně otočí směr přenosu.

6.5.4 Lineární omezení

Nejprve omezíme spuštění operací jen na povolené procesory (rovnice 6.14). Při implementaci přímo odstraníme veškeré proměnné, které se vztahují k dané dvojici operace a procesoru.

$$\begin{aligned} &\forall o \in O, m \in M, i \in I \\ &\quad Runable(o, m) = 0 \quad \Rightarrow \quad u_{o,m,i} = 0 \end{aligned} \tag{6.14}$$

Dále musíme zajistit synchronní zpracování operací v rámci každé úlohy (rovnice 6.15) jak bylo popsáno v sekci 6.1.

$$\begin{aligned} &\forall f \in O, (f, g) \in P, i \in I \\ &\sum_{m \in M} \frac{u_{f,m,i}}{WorkCPU(f)} \quad = \quad \sum_{m \in M} \frac{u_{g,m,i}}{WorkCPU(g)} \end{aligned} \tag{6.15}$$

Nyní přistoupíme k omezení přenosů dat tak, aby odpovídaly probíhajícím výpočtům. Přenosová omezení dávají do vztahu producenty s konzumenty ovšem s tím, že jedna operace může být počítána současně na více procesorech (6.2).

$$\begin{aligned} &\forall f \in O, (f, g) \in P_u, i \in I, p, q \in M \qquad \text{symetrie} \\ &t_{f,g,p,q,i} - tabs_{f,g,p,q,i} = -(t_{f,g,q,p,i} - tabs_{f,g,q,p,i}) \end{aligned} \tag{6.16}$$

$$\begin{aligned} &\forall f \in O, (f, g) \in P_u, i \in I, m \in M \qquad \text{kontinuita} \\ &\sum_{q \in M} (t_{f,g,m,q,i} - tabs_{f,g,m,q,i}) = \end{aligned}$$

$$= \left(\frac{u_{f,m,i}}{WorkCPU(o)} - \frac{u_{g,m,i}}{WorkCPU(g)} \right) \cdot WorkIO(f,g,m) \tag{6.17}$$

$$\forall p,q \in M, i \in I : c = 0 \vee \exists (p,q,c) \in Capacity \qquad \text{kapacita}$$

$$\sum_{f,g \in O} t_{f,g,p,q,i} \leq c \cdot s_i \tag{6.18}$$

Pomocí symetrických přenosů (rovnice 6.16) si připravíme proměnné pro hlavní omezení toků, kontinuitu (rovnice 6.17). Ta zabezpečuje, že množství přijatých dat ke každému procesoru, který počítá danou operaci, bude odpovídat vyprodukovanému množství. Kapacita už jen zamezuje překročení uživatelem dodaných přenosových kapacit (nerovnice 6.18). Je dobré si všimnout, že pracujeme s kladnou částí přenosu $t_{f,g,p,q,i}$.

Nyní je třeba se omezit na maximální výkon procesorů. Přitom nesmíme zapomenout, že i samotný přenos dat vyžaduje výpočetní kapacitu.

$$\forall m \in M, i \in I$$

$$\sum_{o \in O} u_{o,m,i} + \sum_{f,g \in O, q \in M} (t_{f,g,m,q,i} + tabs_{f,g,m,q,i}) \cdot Io(m,q) \leq Speed(m) \cdot s_i \tag{6.19}$$

Výkonnostní omezení procesorů jako součet čisté výpočetní práce a komunikační práce (nerovnice 6.19). Výraz $t_{f,g,p,q,i} + tabs_{f,g,p,q,i}$ počítá absolutní hodnotu $t_{f,g,p,q,i} - tabs_{f,g,p,q,i}$. To je třeba, protože procesor je zatěžován jak u odesílatele, tak u příjemce.

$$\forall o \in O : \sum_{m \in M, i \in I} u_{o,m,i} \geq WorkCPU(o) \tag{6.20}$$

Poslední nezbytné omezení zajistí, že práce všech operací bude dokončena (nerovnice 6.20).

6.5.5 Další možná omezení

Nyní následují do implementace nezahrnutá omezení, která lze v modelu případně vyjádřit.

$$\forall o \in O, i \in I : \exists (o,r) \in CpuRate$$

$$\sum_{m \in M} u_{o,m,i} = s_i \cdot r \tag{6.21}$$

Průměrný trvale udržitelný výpočetní postup, který má být garantován (rovnice 6.21). $CpuRate$ udává průměrný počet instrukcí zpracovaný za sekundu. O operacích s tímto omezením se předpokládá, že běží po celou dobu plánovaného období.

Alternativně lze předchozí formulovat jako garantovaný datový přenos u vybrané dvojice navazujicích operací $(f, g) \in P$.

$$\forall i \in I : \exists (f, g, r) \in DataRate$$
$$\sum_{m \in M} u_{f,m,i} \cdot WorkIO(f, g, m) = s_i \cdot r \tag{6.22}$$

6.5.6 Účelová funkce

Cílem je naplánovat úlohy tak, aby se zatížily co nejvíce procesory. Ovšem toto zatížení nesmí obsahovat komunikační složku.

$$\min \left(\sum_{i \in I} s_i + \sum_{i \in I, m \in M, o \in O} u_{o,m,i} - \sum_{o \in O} WorkCPU(o) \right)$$

Jsou-li procesory zatíženy užitečnou prací, pak nutně minimalizují celkovou dobu běhu. Intervaly tedy musí obsahovat minimum nevyužitého času. Protože $\sum_{o \in O} WorkCPU(o)$ je konstanta, lze ji vynechat. Naopak je nutné přidat část proměnných počítající absolutní hodnotu transportu dat. Konečná účelová funkce je

$$\min \left(\sum_{i \in I} s_i + \sum_{i \in I, m \in M, o \in O} u_{o,m,i} + \sum_{f,g \in O, p,q \in M, i \in I} t_{f,g,p,q,i} \right)$$

6.6 Měřítko

Hledání globálního optima lineárního problému je iterativní proces. Ve spojení s aritmetikou s pevnou přesností (např. ANSI/IEEE Standard 754-1985, Standard for Binary Floating Point Arithmetic) je nutné minimalizovat numerické chyby úpravou koeficientů matice problému. Cílem úpravy je formulace ekvivalentního problému takového, že koeficienty v matici budou řádově poblíž jedné.

Hledáme tedy lineární funkce $f(x) = a \cdot x$ pro transformaci interpretace proměnných lineárního problému, a dále lineární funkce pro interpretaci sčítance v rámci každého omezení.

Ekvivalence transformovaných omezení Lineární omezení

$$C_0 : a_0 x_0 + \cdots + a_n x_n = b$$

je ekvivalentní s omezením

$$C_1 : T_0 r_{C_0} a_0 x_0 + \cdots + T_n r_{C_n} a_n x_n = r_{C_b} b$$

pokud jsou splněny následující omezení.

$$\begin{aligned} D_0 &: \log_2 T_0 + \log_2 r_{C_0} = \log_2 r_{C_b} \\ &\vdots \\ D_n &: \log_2 T_n + \log_2 r_{C_n} = \log_2 r_{C_b} \end{aligned}$$

Definiční obor logaritmů omezuje T_i, r_{C_i} a r_{C_b} pro $0 \leq i \leq n$ na kladné hodnoty. Hodnoty T_i příslušejí proměnné x_i a jsou stejné pro každé omezení z množiny všech omezení.

Optimalizace transformovaných omezení Nechť máme množinu omezení tvaru C_1 jako v předchozím odstavci. Pak lineární problém složený z omezení $D_0, \cdots, D_n$ (také z předchozího odstavce), kde pro všechna i jsou $\log_2 T_i$ a $\log_2 r_{C_i}$ a $\log_2 r_{C_b}$ proměnné, nové proměnné $digits$, dodatečných omezení

$$\begin{aligned} E_0 : &\quad -digits \leq \log_2 r_{C_0} + \log_2 |a_0| \leq digits \text{ pro } a_0 \neq 0 \\ &\vdots \\ E_n : &\quad -digits \leq \log_2 r_{C_n} + \log_2 |a_n| \leq digits \text{ pro } a_n \neq 0 \\ E_b : &\quad -digits \leq \log_2 r_{C_b} + \log_2 |b| \leq digits \text{ pro } b \neq 0 \end{aligned}$$

pro každé původní omezení a účelové funkce

$$\min digits$$

je numericky optimální reprezentace původního lineárního problému. Tedy koeficienty matice lineárního problému budou řádově poblíž jedné. Proměnná r_{C_b} se nazývá koeficient omezení C. Proměnné T_i se říká koeficient proměnné x_i původního problému. Hodnoty r_{C_i} není nutné ukládat, protože je lze dopočítat z příslušného koeficientu omezení a koeficientu proměnné.

Koeficienty proměnných (T_i) se nevyskytují v omezeních E_i, protože se jedná o interpretace hodnot. Například u časové proměnné určují zda je v sekundách, milisekundách, hodinách apod.

6.7 Měřítko v problému plánování

Nyní lze sestavit lineární problém měřítka pro lineární problém plánovače. Stačí uvažovat pouze základní šablony omezení s tím, že za funkce, které dosazují konstanty plánovaného problému (6.5.2), dosadíme extrémní hodnoty. Vzhledem k tomu, že v žádném omezení se nevyskytují více než dvě netriviální konstanty, počet uvažovaných omezení bude maximálně počet šablon krát čtyři (veškeré kombinace minim a maxim všech konstant).

Dále dojde k omezení počtu transformovaných proměnných na tři:

$$\begin{aligned} T_s(x) &= a_s \cdot x \quad \text{pro } \{x | x \in s_i\} \\ T_u(x) &= a_u \cdot x \quad \text{pro } \{x | x \in u_{o,m,i}\} \\ T_t(x) &= a_t \cdot x \quad \text{pro } \{x | x \in t_{f,g,p,q,i}\} \end{aligned}$$

Díky této radikální redukci bude problém optimální transformace konstantní velikosti s řádově desítkou proměnných.

Vyjádření optimální transformace měřítka jako lineárního problému bylo umožněno převodem omezení ve tvaru součinů na součet. Další výhodou logaritmizace je převedení hodnot parametrů neznámých rozsahů na rozsahy, s výrazně menší hrozbou numerických chyb – nejedná se tedy o vytloukání klínu klínem ve vztahu k problému plánovače.

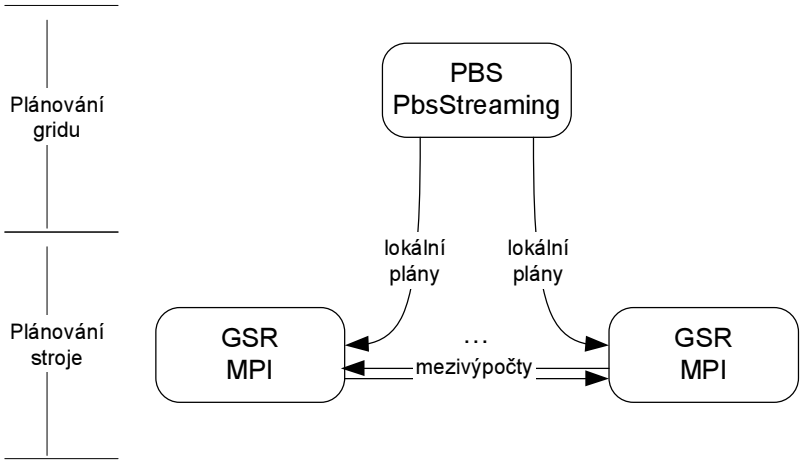
6.8 Výpočet a realizace plánu

Výpočet v této kapitole popsaného lineární problému je implementován vlastním programem PbsStreaming.exe. Ten komunikuje s PBS serverem a získává od něj proudové úlohy shromážděné od uživatelů. Jeho úkolem je vytvoření matice lineárního problému popsaného v této kapitole a její vyřešení. Výsledný plán neprovádí PbsStreaming přímo, ale předává jej dál GSR (viz 5.4), který běží na každém výpočetním uzlu vyhrazeném pro proudové úlohy.

K řešení matice lineárního problému současná implementace používá knihovnu lp_solve verzi 5.5. k výpočtu jak hlavního problému plánování tak problému měřítka. Výběr lp_solve byl ovlivněn jak její licencí (GPL), tak tím, že ji lze použít jako knihovnu bez nutnosti tvorby zadání lineárního problému formou MPS souboru. Bohužel se ukázalo, že kvalita není na vysoké úrovni. Při použití některých funkcí[2] pomocí kterých se tvoří matice problému docházelo k poškození již zadaných dat a výsledek LP problému byl následně chybný. PbsStreaming proto obsahuje zpětnou kontrolu matice

2. například add_constraintex

Obrázek 6.3: Hierarchie plánovačů s tokem dat



v různých fázích její tvorby. Při pokusu o opravu zdrojového kódu lp_solve jsem zjistil, že oblast kódu, kde se pravděpodobně chyba nacházela, se v programu vyskytuje v jednotkách (možná i v desítkách) kopií. O dobrém návrhu tento stav rozhodně nesvědčí a vzhledem k tomu, že oprava chyby byla mimo rámec diplomové práce, jsem ji nerealizoval. Problém jsem obešel přijatelnou posloupností volání funkcí při tvorbě matice LP.

Další problém lp_solve tkví v její numerické nestabilitě při některých lineárních problémech. Některé, i malé, matice LP nemusí být vyřešeny, přestože jiní řešitelé[3] při ručním zadání téhož výsledek naleznou. PbsStreaming nevyužívá žádných speciálních funkcí lp_solve a přechod na jinou knihovnu LP není do budoucna nijak omezen.

Jak bylo nastíněno v prvním odstavci PbsStreaming nerealizuje vypočtený plán přímo, ale pomocí MPI (Message Passing Interface) spustí GSR na každém ze spravovaných uzlů. Vybraný uzel, zvaný master, pak přebere globální plán a zařídí jeho distribuci pro jednotlivé uzly. Nejdůležitější informací, kterou každý uzel dostane k proudovým operátorům, jsou podíly výkonu stroje – váhy. Jejich realizaci provádí už zmíněný WeightedScheduler (viz 5.4.4).

Při plánování proudových úloh je nezbytná hierarchie plánovačů – globálního na úrovni celého gridu a lokálního pro každý výpočetní uzel. Vztah spolu s tokem dat je ilustrován na obrázku 6.3.

Z pohledu PBS je však tento stav méně příhodný. PBS jako správce úloh nemá kontrolu nad jednotlivými procesy úloh na úrovni operačního sys-

3. CPLEX

tému. To není z principu možné, protože precizní kontrola poměrů zatížení stroje, kterou zajišťuje WeightedScheduler je možná jen proto, že veškeré úlohy běží v rámci jediného procesu operačního systému. Na rozdíl od plánovače, který je rozšířením očekávaným již při návrhu PBS, s rozšířením na úrovni jednotlivých uzlů počítáno není. Jediným řešením do budoucna tak zůstává implementace pracovního uzlu PBS zvaného MOM přímo v GSR.

Kapitola 7

Závěr

V diplomové práci bylo prozkoumáno plánování proudově propojených úloh na úrovni výpočetního gridu. Výsledky této analýzy byly použity při realizaci implementací prostředí pro zpracování proudových úloh na platformě CLI pojmenovaného Grid Streaming Runtime i globálního proudového plánovače volně spolupracujícího s PBS.

Výsledky práce ukazují, že lineární programování je možné použít k plánování výpočetních úloh i bez celočíselných proměnných (ILP), a tedy s polynomiální složitostí. Nutností ale bylo jak omezení výpočetního modelu na proudové zpracování, tak návrh netriviálního běhového systému – GSR. Ojedinělé je i plánování datových toků mezi výpočetními uzly, které bývá opomíjeno jinými plánovači. Datový tok zabírá nejen kapacitu přenosového média, ale i poměrnou část výkonu procesoru. Velký rozsah lineárních problémů si vyžádal nutnost přímočaré kontroly ekvivalence programu se specifikací. Proto byla využita relační algebra jako nástroj implementace lineárních omezení. V návaznosti na to jsou i veškerá externí rozhraní plánovače i GSR realizována relační formou.

Dále byla věnována pozornost problému, jak maximálně usnadnit psaní distribuovaných programů – jmenovitě proudových operátorů – s ohledem na bezchybnost a udržovatelnost návrhu. Střídmé rozhraní správy paměti a omezení komunikace na explicitně definované komunikační kanály izolují programátora proudových operátorů před synchronizačními problémy a před hrozbou uváznutí. Veškeré starosti s výměnou dat obstarává GSR. Díky absenci funkcí, které by souvisely s distribuovaným charakterem programu, je možné z operátorů ručně vytvořit graf precedencí a ten spustit speciálním lokálním plánovačem. Běh takového programu je pak v každém okamžiku deterministický. Obrovská výhoda ve chvíli, kdy je nutné odladit defektní program.

Usnadnění programování ale nebylo na úkor výkonu celého systému. Ač se proudové operátory programují modelem zasílání zpráv mezi procesy, nevlastní každý operátor vlastní vlákno, ale díky navrženému kooperativnímu plánování nad CLI, je transformován na volání procedur řízené

konečným automatem. Samotný návrh nebyl příliš komplikovaný, ale při implementaci bylo nutné použít knihovnu bez jakékoliv dokumentace a komentářů ve zdrojových textech. V důsledku to znamenalo velmi zdlouhavý proces ladění.

Do budoucna lze očekávat rozšíření jak v tvorbě vlastních proudových operátorů, které budou počítat potřebné proudové úlohy (zejména zpracování multimédií), tak ve vylepšení implementace, která vznikla v rámci této práce. Jmenovitě se jedná o snížení složitosti plánovače výpočetních uzlů, který používá Weighted Round Robin se složitostí $O(\log n)$. Vzhledem k vysoké četnosti volání plánovače by složitost mohla klesnout až k $O(1)$. Cestou by mohlo být vyžití poznatků z algoritmu známého jako Deficit Round Robin.

Další vylepšení lze očekávat v zavedení plně dynamického plánování zajištěním migrace úloh mezi výpočetními uzly. Vzhledem k použití přenositelných binárních programů CLI je tento úkol proveditelný.

Literatura

[1] Peter Brucker. *Scheduling Algorithms*. Springer, second revised and enlarged edition, 1998.

[2] ECMA. *Common Language Infrastructure (CLI)*. ECMA, 4 edition, 2006. ECMA-335, ISO/IEC 23271:2006.

[3] Mei-Hui Su James Blythe Yolanda Gil Carl Kesselman Gaurang Mehta Karan Vahi Ewa Deelman, Gurmeet Singh. Pegasus: a framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 2005.

[4] Ian Foster. What is the grid? a three point checklist. Technical report, Argonne National Laboratory & University of Chicago, 2002. http://www-fp.mcs.anl.gov/ foster/Articles/WhatIsTheGrid.pdf.

[5] Geoffrey C. Fox Fran Berman, Anthony J. G. Hey. *Grid Computing Making the Global Infrastructure a Reality*. Wiley, 2003.

[6] R. Govindarajan. Minimizing buffer requirements under rate-optimal schedule in regular dataflow networks. Technical report, School of Computer Science, McGill university, 1994.

[7] Mark Wallace Hani El Sakkout. Probe backtrack search for minimal perturbation in dynamic scheduling. *CONSTRAINTS, 4(5):359–388*, 2000.

[8] C. Kesselman I. Foster. *The Grid: Blueprint for a New Computing Infrastructure.* Morgan-Kaufman, second edition, 2004.

[9] W. Kocjan. Dynamic scheduling: State of the art report, 2002.

[10] Wagner Meira Jr. Dorgival Guedes Luiz Thomaz do Nascimento, Renato A. Ferreira. Scheduling data flow applications using linear programming. *Proceedings of the 2005 International Conference on Parallel Processing*, 2005.

[11] Michael Beynon Tahsin Kurc Umit Catalyurek Alan Sussman Joel Saltz Matthew Spencer, Renato Ferreira. Executing multiple pipelined data analysis operations in the grid. *Proceedings of the IEEE/ACM SC2002 Conference*, 2002.

[12] Manfred Glesner Michael Münch, Norbert Wehn. An efficient ilp-based scheduling algorithm for control-dominated vhdl descriptions. *Proceedings of the 9th International Symposium on System Synthesis*, 1996.

[13] Michela Milano. *Constraint and Integer Programming*. Kluwer Academic Publishers, 2004.

[14] Y. Robert O. Beaumont, A. Legrand. Scheduling strategies for mixed data and task parallelism on heterogeneous clusters and grids. *Proceedings of the Eleventh Euromicro Conference on Parallel,Distributed and Network-Based Processing*, 2003.

[15] Michael L. Pinedo. *Scheduling: Theory, Algorithms, and Systems.* Prentice Hall, second edition, 2002.

[16] Michael L. Pinedo. *Planning and Scheduling in Manufacturing and Services*. Springer, 2005.

[17] M. Hajian R. Rodosek, M. Vallace. A new approach to integrating mixed integer programming with constraint logic programming. *Annals of Operations Research*, 1998.

[18] Les Proll Sarah Fores. Driver scheduling by integer linear programming - the tracs ii aproach. Technical Report 98.01, School of Computer Studies, University of Leeds, 1998.

[19] Farzan Fallah Seda Ogrenci Memik. Accelerated sat-based scheduling of control/data flow graphs. *Proceedings of the 2002 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 2002.

[20] Matthew Bartschi Wall. *A Genetic Algorithm for Resource-Constrained Scheduling*. PhD thesis, Massachusetts Institute of Technology, 1996.