Univerzita Komenského

Fakulta Matematiky, Fyziky a Informatiky

SIMULÁCIA PRAM VÝPOČTOV

Diplomová práca

Študijný program: Informatika
Študijný odbor: 2508 Informatika
Školiace pracovisko: Katedra Informatiky
Školiteľ: doc. RNDr. Rastislav Královič, PhD.

2012 **Bc. György Tomcsányi**

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. György Tomcsányi
Študijný program: informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: 9.2.1. informatika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: slovenský

Názov: Simulácia PRAM výpočtov

Cieľ: Cieľom práce je navrhnúť a implementovať systém umožňujúci simulované spúšťanie paralelných algoritmov. Cieľovým použitím sú praktické cvičenia k predmetu "Efektívne paralelné algoritmy". Systém má umožniť písať PRAM programy vo vyššom programovacom jazyku a priamo pre daný procesor alebo použitím WT prezentácie tak, aby väčšina algoritmov spomínaného kurzu bola priamočiaro implementovateľná. Systém má umožniť sledovanie spotrebovaného času a práce a pri WT prezentácii zabezpečiť vhodný scheduling.

Vedúci: doc. RNDr. Rastislav Královič, PhD.
Katedra: FMFI.KI - Katedra informatiky

Dátum zadania: 29.10.2010

Dátum schválenia: 03.11.2010 prof. RNDr. Branislav Rovan, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Ďakujem vedúcemu tejto diplomovej práce doc. RNDr. Rastislavovi Královičovi, PhD. za cenné rady a pripomienky.

Abstrakt

Autor:	Bc. György Tomcsányi
Názov diplomovej práce:	SIMULÁCIA PRAM VÝPOČTOV
Škola:	Univerzita Komenského v Bratislave
Fakulta:	Fakulta matematiky, fyziky a informatiky
Katedra:	Katedra informatiky
Vedúci diplomovej práce:	doc. RNDr. Rastislav Královič, PhD.
Rozsah práce:	42 strán
Bratislava, jún 2012	

Parallel Random Access Machine - PRAM je abstraktný model, ktorý sa stal základným nástrojom na analýzu paralelných algoritmov. Umožňuje študovať podstatné problémy pri paralelizácii a zostáva na dostatočne vysokej úrovni, aby bol nezávislý od konkrétnej architektúry. Preto sa tento model používa na prezentovanie paralelných algoritmov na rôznych informatických kurzoch. Vzniká preto prirodzená požiadavka umožniť písanie algoritmov pre PRAM na reálnom systéme. Cieľom tejto práce je návrh a implementácia simulátora pre PRAM, ktorý umožní simulované spúšťanie paralelných algoritmov.

Kľúčové slová: PRAM, simulácia, paralelné algoritmy

Obsah

Zoznam obrázkov

Zoznam tabuliek

Kapitola 1

Úvod

1.1 Motivácia

Počítače sa stali súčasťou každodenného života a sú zodpovedné za čoraz viac úloh. Rozšírenie miery automatizácie zabezpečuje aj neustáli rast výkonu týchto zariadení, čo umožňuje ich nasadenie na riešenie stále komplexnejších a výpočtovo náročnejších úloh. Zrýchľovanie sa dosahuje hlavne zvyšovaním frekvencie vykonávania funkcií jednotlivých komponentov. Pravidlá fyziky však neumožnia nekonečné zrýchľovanie frekvencie a preto sa už dlhší čas skúmajú ďalšie možnosti zvyšovania výkonu.

Posledné desaťročia panuje presvedčenie, že časom bude hlavná cesta zvyšovania výkonu paralelizmus. V súčasnosti prichádzajú na trh viacjadrové procesory, čo potvrdzuje túto myšlienku. Táto architektúra prináša nové problémy, ktoré treba vyriešiť pre efektívne využitie ich potenciálu. Pri návrhu algoritmov treba brať do úvahy paralelizmus a prispôsobiť ich špecifickým požiadavkám.

Paralelné algoritmy sú už dlhšiu dobu aktívne skúmané. Jeden z najpoužívanejších modelov na ich analýzu je Parallel Random Access Machine - **PRAM**. Prvá, nám známa práca o PRAMe vyšla v roku 1978 [FW78]. Tento model umožňuje sústrediť sa na jadro problému paralelizácie a pritom ostáva na dostatočne vysokej úrovni, aby bol nezávislý od konkrétnej architektúry. Je síce pochybná možnosť jej efektívnej implementácie, ale pomerne veľa autorov sa domnieva, že PRAM je aj z praktického hľadiska relevantný model, viď napr. [Vis11].

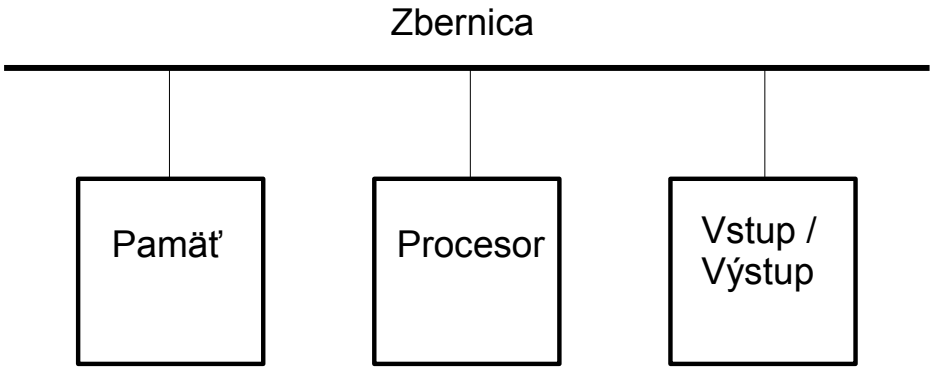
Ako všeobecný a najrozšírenejší model sa PRAM dostal aj do kurikula informatických odborov väčšiny univerzít. V súčasnosti aj na Univerzite Komenského majú študenti možnosť študovať paralelné algoritmy pomocou tohto modelu. Spomínané algoritmy sú často pomerne komplikované a štu-

denti nemajú možnosť si ich implementovať, keďže PRAM je abstraktný model. Počas kurzu vzniká možnosť cvičiť programovanie pomocou PRAMu podobných technológií (napr. openMP, XMTC), ale bolo by lepšie algoritmy písať v prostredí čo najpodobnejšom učebnicovej reprezentácii. Preto sme sa rozhodli implementovať prostredie na simulované spúšťanie programov pre PRAM, v ktorom sa nekladie dôraz na efektivitu a praktický paralelizmus, ale na použiteľnosť ako didaktická pomôcka.

Cieľom práce je teda navrhnúť a implementovať prostredie pre štúdium paralelných algoritmov pomocou ich simulovaného spúšťania. Systém má umožniť písanie programov vo vyššom programovacom jazyku, ktorý abstrahuje od implementačných detailov a tak umožní sústrediť sa na principiálne problémy pri návrhu efektívnych paralelných algoritmov. Systém zabezpečí vhodné plánovanie úloh a meranie spotrebovaného času a práce.

1.2 Sekvenčné algoritmy

Prvý krok pri tvorbe programov je definovanie modelu, pre ktorý ho píšeme. Základný kameň návrhu dnešných počítačov je von Neumannova architektúra, ktorá definuje 3 hlavné komponenty: vstupno - výstupné zariadenia, procesor a operačnú pamäť, viď obrázok 1.1. Tieto komunikujú prostredníctvom zbernice. Často používaným formálnym modelom pre štúdium sekvenčných algoritmov je **Random Access Machine** (RAM), ktorý je dobrou abstrakciu nad von Neumannovou architektúrou. V ďalšej sekcii ukážeme rozšírenie tohto modelu, ktorý budeme používať neskôr pri implementácii paralelných algoritmov.



Obr. 1.1: von Neumannova architektúra

Pri analýze algoritmu nás hlavne zaujíma jeho zložitosť a to pre dve základné miery: čas a pamäť. Túto zložitosť meriame ako funkciu, ktorá závisí

od veľkosti vstupu. Bežne nás zaujíma odhad pre najhorší prípad, teda pre danú veľkosť vstupu ohraničiť potrebné prostriedky pre všetky vstupy danej veľkosti. Tieto ohraničenia vyjadrujeme asymptoticky:

- $T(n) = O(f(n))$ ak existujú kladné konštanty c a n_0, pre ktoré platí $T(n) \leq cf(n)$ pre všetky $n \geq n_0$
- $T(n) = \Omega(f(n))$ ak existujú kladné konštanty c a n_0, pre ktoré platí $T(n) \geq cf(n)$ pre všetky $n \geq n_0$
- $T(n) = \Theta(f(n))$ ak $T(n) = O(f(n))$ a $T(n) = \Omega(f(n))$

Ako sa meria množstvo použitých prostriedkov pre RAM? Čas behu algoritmu je odhadnutý funkciou vyjadrujúcou počet potrebných základných inštrukcií v závislosti od veľkosti vstupu. Tieto inštrukcie sú: čítanie a zápis do pamäte, aritmetické a logické operácie. Funkcia pre pamäťovú zložitosť vyjadruje maximálny počet využitých pamäťových blokov v závislosti od veľkosti vstupu.

1.3 Paralelné algoritmy

1.3.1 Model so zdieľanou pamäťou

Pre paralelné algoritmy existuje niekoľko teoretických modelov, ako napr. Booleovské obvody a sieťové modely. My si bližšie ukážeme **model so zdieľanou pamäťou** (shared-memory model) na základe [JaJ92]. Tento model sa skladá z niekoľkých procesorov, ktoré vykonávajú spoločný program. Jednotlivé procesory komunikujú prostredníctvom zdieľanej (globálnej) pamäte. Každý procesor je jednoznačne identifikovateľný svojim indexom (processor ID), ku ktorému má prístup vo forme lokálnej premennej.

Existuje niekoľko variant tohto modelu, my si ukážeme pre nás dôležité charakteristiky. Prvé rozdelenie na základe počtu súčasných prúdov dát a inštrukcií sa nazýva *Flynnova klasifikácia* [Fly72]:

- **SISD** (*Single Instruction, Single Data*): Model s jedným procesorom, ktorý výkonáva jeden inštrukčný prúd nad jedným prúdom dát. Zodpovedá to Neumannovej architektúre
- **MISD** (*Multiple Instruction, Single Data*): Viac procesorov s rôznymi programami pracuje nad jedným prúdom dát. Neexistuje veľa implementácií tejto architektúry, pre paralelné modely sú obvykle vhodnejšie nasledujúce typy.

- **SIMD** (*Single Instruction, Multiple Data*): Viac procesorov s rovnakým programom pracuje nad rôznymi dátami.
- **MIMD** (*Multiple Instruction, Multiple Data*): Viac procesorov s rôznymi programami pracuje nad rôznymi dátami.

Iná charakterizácia paralelných modelov je podľa ich synchronizácie:

- **Asynchrónne** modely: Každý procesor má vlastné hodiny, môžu pracovať inou rýchlosťou. Synchronizácia procesorov je úloha programátora. Je to potrebné napríklad pri prístupe do pamäte, keďže obsah globálnej pamäte sa dynamicky mení počas behu programu na rôznych procesoroch.
- **Synchrónne** modely: Procesory majú spoločné hodiny.

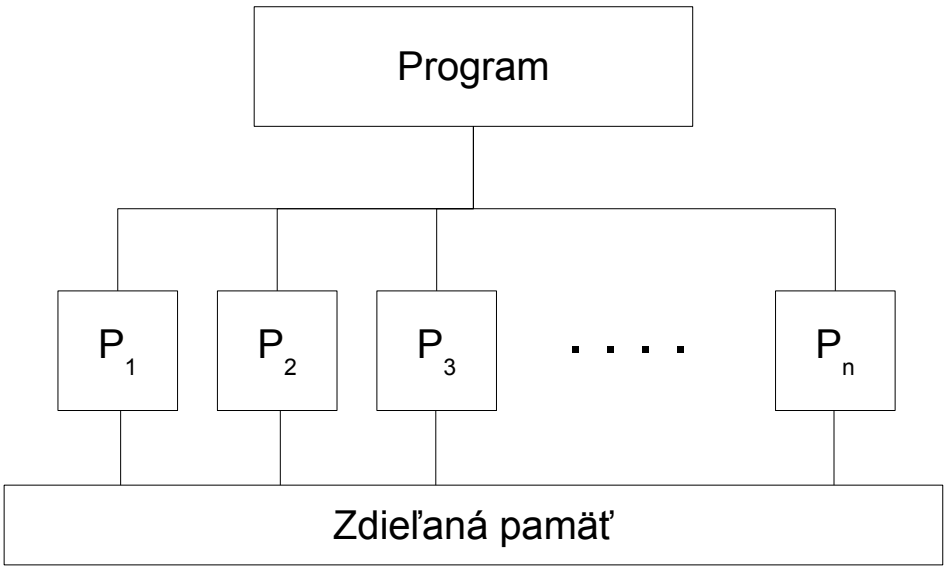
1.3.2 Parallel Random Access Machine

V tejto práci sa budeme venovať synchrónnemu modelu so zdieľanou pamäťou typu SIMD, ktorý je rozšírením modelu RAM - **Parallel Random Access Machine** (PRAM). Jej hlavné komponenty sú znázornené na obrázku 1.2. Množina procesorov má prístup k spoločnému programu a zdieľanej pamäti. Každý procesor má svoje počítadlo inštrukcií, ktoré určuje aktuálne vykonávanú inštrukciu v programe. Počítadlo sa mení dvoma spôsobmi: inkrementuje sa synchronizovane pre všetky procesory po vykonaní inštrukcie alebo sa zmení inštrukciou skoku. Keďže tieto skoky môžu byť podmienené výsledkom vyhodnotenia nejakého výrazu a procesory majú prístup k svojmu ID, môže sa stať, že podmienka sa vyhodnotí rôzne pre rôzne procesory. To spôsobí, že niektoré procesy budú vykonávať inú časť kódu ako ostatné.

Typy PRAM

Pri súčasnom prístupe do pamäte sa môže stať, že dva procesory chcú naraz pracovať s rovnakou časťou globálnej pamäte. PRAMy delíme do rôznych tried na základe techniky ako spracujú tieto **konflikty**:

- **EREW** (*Exclusive Read Exclusive Write*): Nie sú povolené žiadne súčasné prístupy do pamäte.
- **CREW** (*Concurrent Read Exclusive Write*): Sú povolené inštrukcie súčasného čítania.



Obr. 1.2: Parallel Random Access Machine

- **CRCW** (*Concurrent Read Concurrent Write*): Sú povolené inštrukcie súčasného čítania aj zapisovania. Tu rozlišujeme ešte podľa toho, ako sa spracujú súčasné zápisy:
  - **common**: Povolí zapísať len ak sa všetky procesory pokúšajú zapísať tú istú hodnotu.
  - **arbitrary**: Povolí niektorému procesoru zapísať.
  - **priority**: Povolí zapísať procesoru s najmenším processor ID.

1.3.3 Analýza paralelných algoritmov

V sekcii 1.2 sme definovali analýzu sekvenčných algoritmov. V tejto sekcii ukážeme zložitostné aspekty paralelných algoritmov (na základe [JaJ92]). Najprv porovnáme výkon sekvenčného a paralelného algoritmu pre ten istý problém. Majme problém P, ktorý vieme riešiť optimálnym sekvenčným algoritmom v čase $T^*(n)$. Majme paralelný algoritmus, ktorý vyrieši problém P v čase $T_p(n)$ na PRAMe s p procesormi. **Zrýchlenie** získané paralelným algoritmom definujeme:

$$S_p(n) = \frac{T^*(n)}{T_p(n)}.$$

My by sme chceli navrhnúť algoritmy pre ktoré platí $S_p(n) \approx p$, t.j. pri použití p procesorov sme dosiahli p-násobné zrýchlenie. $T_1(n)$ je čas behu paralelného

algoritmu ak má k dispozícií jeden procesor. Za povšimnutie stojí, že táto hodnota sa nie nutne rovná $T^*(n)$. Definujme ďalej **efektívnosť** nasledovne:

$$E_p(n) = \frac{T_1(n)}{pT_p(n)}.$$

Hodnota $E_p(n)$ blízka 1 znamená dobré využitie všetkých p procesorov. Existuje ohraničenie času behu označené $T_\infty(n)$, od ktorého neexistuje rýchlejší algoritmus bez ohľadu na počet procesorov.

Definujme teraz presnejšie výkon paralelných algoritmov. Majme opäť problém P a paralelný algoritmus, ktorý ho rieši v čase $T(n)$ s $P(n)$ procesormi. Súčin $C(n) = T(n) * P(n)$ voláme **cena** paralelného algoritmu. Tento algoritmus sa dá jednoducho simulovať na sekvenčnom modely v čase $O(C(n))$. Podobne sa dá simulovať pôvodný algoritmus na PRAMe s p procesormi v čase $O(C(n)/p)$. Novou mierou zložitosti je **komunikácia**, teda počet pamäťových operácií medzi globálnou a ľubovolnou lokálnou pamäťou.

1.3.4 Work-Time Presentation Framework

V tejto časti ukážeme, ako sa píšu programy pre PRAM. V sekcii 1.3.1 sme definovali náš model ako niekoľko procesorov s prístupom ku zdieľanej pamäti, kde každý procesor vykonáva ten istý program. To bude prvý prístup, ktorý budeme používať na paralelné programovanie, *písanie programu pre daný procesor*. Konkrétny programovací jazyk predstavíme v kapitole 2, zatiaľ nám stačí intuitívna predstava získaná skúsenosťou zo sekvenčného programovania.

Použitý model značne uľahčuje programovanie, lebo nám umožní sústrediť sa na principiálne problémy pri tvorbe algoritmov a zjednodušuje detaily. Paradigma, ktorú teraz prezentujeme nám ďalej pomôže zavedením ďalšej úrovne abstrakcie. Zadefinujeme tiež novú mieru zložitosti **práca**, čo je celkový počet vykonaných operácií.

Work-time (WT) paradigma rozdelí návrh paralelných algoritmov na dve úrovne. Vyššia úroveň je popis algoritmu bez špecifických detailov. Nízka úroveň definuje princíp plánovnia.

Vyššia úroveň (*WT prezentácia algoritmov*) definuje algoritmus ako postupnosť časových jednotiek, kde každá časová jednotka môže obsahovať ľubovolné množstvo súčasných inštrukcií. Na dosiahnutie tohto správania sa slúži nasledujúca jazyková konštrukcia:

$$\textbf{for } l \leq i \leq u \textbf{ pardo } prkaz$$

Príkaz (alebo blok príkazov) závisí od indexu i. Príkazy pre všetky indexy z intervalu $\langle l, u \rangle$ sú vykonané súčasne. Tento prístup abstrahuje od procesorov

PRAMu, keďže počet súčasne vykonaných inštrukcií nie je limitovaný počtom procesorov.

Nižšia úroveň (*princíp plánovania*): k danému algoritmu napísanom pomocou WT prezentácie, ktorý spotrebuje čas $T(n)$ a vykoná $W(n)$ práce, vieme skoro vždy skonštruovať algoritmus pre PRAM s p procesormi, ktorý vykoná maximálne $\lfloor \frac{W(n)}{p} + T(n) \rfloor$ paralelných krokov. Nech $W_i(n)$ je počet inštrukcií vykonaných počas i-tej časovej jednotky. Simulujme každú množinu operácií $W_i(n)$ v čase maximálne $\lceil \frac{W_i(n)}{p} \rceil$. Ak je simulácia úspešná, algoritmus vykoná maximálne

$$\sum_{i=1}^{T(n)} \lceil \frac{W_i(n)}{p} \rceil \leq \sum_{i=1}^{T(n)} \left(\lfloor \frac{W_i(n)}{p} \rfloor + 1 \right) \leq \lfloor \frac{W(n)}{p} \rfloor + T(n)$$

krokov. Úspešnosť tejto simulácie závisí od nasledujúcich implementačných detailov:

- výpočet $W_i(n)$ pre každé i (väčšinou triviálne)
- plánovanie procesorov, hlavne alokácia úloh pre jednotlivé procesory. Pre každý krok musí každý procesor vedieť, či je aktívny alebo nie a vykonať správnu inštrukciu.

1.4 Simulátory PRAMu

Našim cieľom je navrhnúť a implementovať simulátor PRAMu, ktorý vie interpretovať programy pre daný procesor a tiež programy napísané pomocou WT prezentácie. V tejto sekcii prezentujeme niektoré výsledky výskumu problematiky paralelných programovacích jazykov.

Ako prvú spomenieme prácu pracovníkov Saarland University, ktorí navrhli a skonštruovali PRAM v hardvéri – *SB-PRAM* [PBB+02]. Je to paralelná architektúra, ktorá používa multithreading, pipelining a hashovanie pamäťových adries z dôvodu znáhodnenia prístupov do pamäte. Prototyp má 64 procesorov. Testy ukázali, že výkon stroja je len o $1,34\%$ horší ako predpovedali simulácie, ktoré predpokladali ideálnu zdieľanú pamäť s uniformným prístupovým časom. Základom softvérovej výbavy stroja je UNIX-like operačný systém, kompilátor pre jazyk FORK a C/C++ rozšírené s konceptom zdieľaných premenných.

Jeden z prvých návrhov reálneho programovacieho jazyka pre PRAM bolo rozšírenie jazyka Modula-2 – *Modula-2** [THT+90]. Ich hlavným prínosom je popísanie synchrónnych a asynchrónnych cyklov *FORALL*, ktoré sa podobajú

na naše *pardo* cykly. Súčasťou práce sú aj implementačné techniky týchto jazykových konštruktov pre rôzne paralelné architektúry.

Iným návrhom je jazyk FORK [HSS92], ktorý paralelizmus rieši formou dynamického vytvárania procesorov inštrukciou *fork*. Túto zložitejšiu konštrukciu zaviedli namiesto *pardo* kvôli jej obmedzenosti v niektorých prípadoch. Naša interpretácia cyklov *pardo* je troška iná, zahŕňa aj črty inštrukcie *fork*.

Jazyk ParC [BAFR96] je rozšírením jazyka C. Hlavné črty sú paralelné konštrukty, ktoré sa dajú do seba vnárať a logická štruktúra premenných, ktorá nepotrebuje explicitne deklarovať zdieľané premenné.

Doteraz prezentované jazyky boli navrhnuté pre reálne paralelné architektúry. Našim cieľom je umožniť písanie paralelných programov na dostupných sekvenčných počítačoch. Teraz predstavíme dosiahnuté výsledky v simulácii algoritmov pre PRAM. Prvý nám známy emulátor PRAMu je prácou University of Joensuu. PRAM je možné programovať priamo v assembleri [H$\ddot{9}$2] alebo vo vyššoúrovňovom jazyku pm2 [Juv92]. Používajú podobné *pardo* cykly, ale implementácia má niekoľko nedostatkov. Hlavný problém je synchronizácia procesorov namiesto procesov. Túto otázku podrobnejšie preskúmame neskôr v sekcii 2.4. Ďalšie nevýhody sú dôsledkom tejto voľby a to hlavne zavedenie synchronizačných konštrukcií do jazyka a tiež potenciálne nedeterministické správanie algoritmov (v závislosti od plánovania procesov). Okrem toho chýba možnosť vnárania *pardo* cyklov do seba. Tento emulátor kladie vyššie nároky na správnu implementáciu algoritmov, pri ktorých treba použiť explicitnú synchronizáciu. Našim cieľom naopak bolo abstrahovať od týchto problémov a pracovať s algoritmami písaných na vyššej úrovni abstrakcie.

Kapitola 2

Návrh

Keďže cieľom práce je spúšťanie paralelných programov písaných pre teoretický model, nie je ich možné jednoducho interpretovať na bežných PC. Preto sme navrhli virtuálny stroj pre PRAM, ktorý umožní simulované spúšťanie spomínaných algoritmov. Následne sme vytvorili jednoduchý programovací jazyk, pomocou ktorého sa dajú priamočiaro implementovať programy pre PRAM.

V tejto časti sa budeme venovať popisu jazyka a predstavíme tiež virtuálny stroj. Cieľom tejto kapitoly je predstaviť základ fungovania simulátora PRAMu na vyššej úrovni. Podrobnosti a implementačné detaily uvádzame v kapitole 3.

2.1 Architektúra PRAMu

V tejto časti prezentujeme návrh virtuálneho stroja pre PRAM. Jej hlavnou úlohou je interpretovať programy, plánovať procesy, manažment pamäte a vstupno-výstupné operácie. Základné komponenty sme predstavili v časti 1.3.2, teraz ich popíšeme detailnejšie.

Pred popisom vnútorných častí PRAMu uvedieme jej rozhranie smerom k programátorovi. Prvotný vstup je samozrejme samotný program, ktorého podrobnosti predstavíme v ďalších častiach. Počas inicializácie sa nahrá program do pamäte PRAMu aby k nemu neskôr mal prístup. Následne programátor už v roli používateľa interaguje s programom pomocou vstupno výstupných rozhraní. Tieto umožňujú načítať dáta do pamäte za účelom ich spracovania a následného prezentovania výsledkov používateľovi. Okrem samotného výstupu má programátor tiež možnosť dostať údaje o zložitosti behu algoritmu.

Našim cieľom je spúšťanie paralelných algoritmov a našou základnou pra-

covnou jednotkou bude proces. Proces má prístup k programu a vo svojom vnútornom stave si pamätá pozíciu inštrukcie, ktorú v danom momente vykonáva - tzv. *Instruction Pointer*. Je dôležité si uvedomiť, že nezávisle od počtu procesov, program je jeden spoločný. Okrem toho môže každý proces nadobudnúť dva stavy:

1. **aktívny:** proces sa vykonáva
2. **spiaci:** proces čaká na nejakú udalosť

Srdcom PRAMu je množina procesorov, ktoré vedia vykonávať inštrukcie procesov. Ich počet budeme označovať p a po úvodnom nastavení bude fixný (viď inicializácia v 3.5). Všetky procesory sú riadené spoločnými hodinami, preto vykonávajú kód synchronizovane po inštrukciách. Typický (zatiaľ značne zjednodušený) sled udalostí počas vykonávania programu: vytvoria sa procesy a inicializujú si vnútorné stavy. Procesy sa namapujú na procesory a následne synchronizovane vykonávajú inštrukcie. Viac o synchronizácií procesov v sekcii 2.4.

Ďalšia časť PRAMu je pamäť, ktorá je jedinou možnosťou komunikácie paralelných procesov. Pamäť je zdieľaná, čo znamená, že každý proces k nej má prístup. Súčasný prístup na jedno pamäťové miesto spôsobí konflikt, ktorý sa rieši podľa typu PRAMu, viď sekciu 1.3.2. Okrem toho má každý proces možnosť vytvoriť si vlastnú, lokálnu pamäť ku ktorej bude mať výlučný prístup.

Vstupno-výstupné rozhranie umožňuje interagovať s programom pomocou načítavania a vypisovania obsahu pamäte. Pri prístupe k tomuto rozhraniu z paralelného prostredia prirodzene vnikajú problémy, ako interpretovať tieto príkazy. Používateľ uvidí serializovanú verziu výsledku, kde poradie jednotlivých operácií nie je definované. Preto je odporúčané pristupovať k tomuto rozhraniu len použitím jedného procesu.

2.2 Jazyk

Definujeme podstatné prvky programovacieho jazyka pre PRAM. Pre lepšie pochopenie odporúčame pozrieť si príklad implementovaný v časti 4 a gramatiku jazyka v prílohe A.

Jazyk na písanie programov pre PRAM obsahuje klasické prvky, ktoré sú známe z jazykov C alebo Pascal. Sem patria hlavne:

1. **priradenie:**
   premenná := výraz

Výraz na pravej strane sa vyhodnotí a výsledok sa priradí premennej na ľavej strane.

2. **blok begin/end:**
   begin
   príkaz
   príkaz
   ...
   príkaz
   end
   Definuje postupnosť príkazov.

3. **príkaz if:**
   if (podmienka) then príkaz [else príkaz]
   Vyhodnotí sa podmienka. Ak jej hodnota je pravda (true) tak sa vykoná príkaz (blok príkazov) za slovom then. Vetva else je nepovinná a vykoná sa, ak sa podmienka vyhodnotí ako nepravda (false).

4. **cykly:**
   for premenná := počiatočná hodnota to konečná hodnota do príkaz
   while (podmienka) do príkaz
   do príkaz while (podmienka)
   Interpretácia cyklu for je nasledovná: najprv sa priradí premennej počiatočná hodnota. Ak je premenná menšia alebo rovná ako konečná hodnota, tak sa vykoná príkaz (blok príkazov) a následne sa zvýši hodnota premennej. Tento proces sa opakuje kým hodnota premennej nepresiahne konečnú hodnotu, vtedy sa vykonávanie cyklu skončí. Pri cykle while sa vyhodnotí podmienka. Ak je pravdivá vykoná sa príkaz (blok príkazov) a znovu sa vyhodnotí podmienka. Cyklus skončí ak sa podmienka vyhodnotí ako nepravda. Pri cykle do-while sa najprv vykoná telo a potom sa vyhodnotí podmienka. Ak je pravdivá, cyklus sa opakuje inak sa ukončí.

5. **volanie funkcie:**
   menoFunkcie(parameter1, parameter2, ...)
   Odovzdá sa riadenie funkcii *menoFunkcie* s parametrami *parameter1*, *parameter2*, ...

6. **vrátenie hodnoty:**
   return výraz
   Vráti sa riadenie volajúcej funkcii, ktorá dostane vyhodnotenú hodnotu výrazu.

2.3 Paralelizmus

Definovali sme základy jazyka. V tejto časti predstavíme paralelné konštrukcie jazyka. Budeme používať dva prístupy na reprezentáciu paralelných algoritmov. Prvý z nich je písanie programu pre daný procesor. Virtuálny stroj emuluje PRAM s p procesormi (p sa nastaví pred vykonaním programu a počas behu sa nemení), ktorí vykonávajú náš program. Každý procesor má prístup k svojmu ID vo forme lokálnej premennej. V jazyku sa tieto algoritmy implementujú pomocou paralelných procedúr, ktoré sa definujú kľúčovým slovom **parallel** pred definíciou. Tieto funkcie nemôžu mať explicitné argumenty, preto ich nazývame *paralelné procedúry*. Jediná možnosť na komunikáciu s touto metódou je zdieľaná pamäť.

Druhú možnosť sme už popísali v časti 1.3.4, je to Work-Time prezentácia. Do jazyka pridáme nasledujúcu konštrukciu:

$$\textbf{for } i := l \textbf{ to } u \textbf{ pardo } prikaz$$

Interpretácia cyklu je nasledovná: vytvorí sa proces pre každú hodnotu z intervalu $\langle l, u \rangle$ a každý začne vykonávať príkaz (blok príkazov). Procesy budú mať prístup k svojmu indexu i. *Pardo* cykly je možné do seba vnárať, v tom prípade bude mať proces prístup ku všetkým indexom nadradených cyklov.

Je dôležité poznamenať, že pri prvom prístupe sme definovali program pre jednotlivé procesory a vedeli sme, že počet paralelne vykonávaných inštrukcií bude presne p. Pri použití cyklu *pardo* sa môže vytvoriť viac (alebo menej) ako p procesov a preto je úlohou virtuálneho stroja, aby naplánoval procesy pre jednotlivé procesory. Pre programátora je tento proces transparentný, nevie ho ovplyvniť a ani nemá možnosť zistiť koľko procesorov má k dispozícii. Preto hovoríme, že táto forma písania algoritmov je na vyššej úrovni, keďže namiesto procesorov uvažujeme procesy. V ďalšej časti ukážeme ťažkosti pri synchronizácii, ktoré vyplývajú z tohto faktu.

2.4 Synchronizácia

Jedna z najdôležitejších vlastností PRAMu je synchronizácia, preto sa jej podrobnejšie venujeme v tejto časti. Prvú úroveň synchronizácie zaisťuje fakt, že procesory majú spoločné hodiny, dôsledkom čoho sa inštrukcie vykonajú na všetkých procesoroch naraz. Tento mechanizmus nazveme **synchronizácia procesorov** a v ďalších odsekoch ukážeme, prečo nie je postačujúca pre naše účely.

Synchronizácia procesorov je vhodný mechanizmus na zaistenie synchronizácie v prípade ak uvažujeme len programy písané pre konkrétne procesory

a nevytvárame dynamicky nové procesy. Našim cieľom však je interpretovať aj programy písané pomocou WT prezentácie, kde nemôžeme nič predpokladať o počte procesov a predpokladáme tzv. implicitnú synchronizáciu.

Teraz predstavíme riešenie emulátoru z práce [Juv92], ktoré používa synchronizáciu procesorov. V inicializačnej časti programu sa nastaví počet procesorov - p, ktoré sa neskôr nemení. V tele programu sa pomocou cyklu *pardo* môže vytvoriť ľubovoľné množstvo procesov. V prípade ak počet procesov nie je väčší ako počet procesorov, sa môžu všetky procesy namapovať na procesory a vykonávať inštrukcie naraz. Tu nevznikajú problémy, preto sa ďalej budeme venovať druhej možnosti, t.j. ak je počet procesov väčší ako počet procesorov.

Priraďovanie procesov na procesory zaisťuje plánovač úloh. Ten v prvom kroku vytvorí skupinu procesov veľkosti p, ktoré namapuje na procesory a vykoná ich celé telo. V ďalšom kroku plánovač z procesov, ktoré vynechal v prvom kroku vytvorí ďalšiu skupinu veľkosti p a opakuje proces. Tento cyklus končí keď už nie sú čakajúce procesy. Pri tomto prístupe je ľahko vidieť, že sa stráca informácia o vzťahoch medzi procesmi. Pre lepšiu názornosť demonštrujeme tieto nedostatky na príkladoch.

Uvažujme nasledujúci program pre common-CRCW PRAM:

```
shared int a;
a := 0;
for i := 1 to 10 pardo
    a := a + 1;
write a;
```

Programátor by predpokladal, že sa vytvorí 10 procesov, ktoré načítajú hodnotu premennej a (0), pripočítajú k nej 1 a túto zvýšenú hodnotu naraz napíšu naspäť (keďže použitý model je common-CRCW a procesy zapisujú rovnakú hodnotu, nevznikajú konflikty). Výsledkom by teda malo byť číslo 1. V skutočnosti je výsledok výpočtu závislý od počtu procesorov PRAMu. Ak je ich aspoň 10, tak dostaneme správny výsledok, ale situácia sa zmení ak je ich menej. Uvažujme 5 procesorov. V tomto prípade plánovač úloh vytvorí skupinu veľkosti 5 a vykoná ich telo. Procesy načítajú hodnotu 0, pripočítajú k nej 1 a túto novú hodnotu zapíšu. V ďalšom kole plánovač vytvorí novú skupinu zo zvyšných 5 procesov, ktoré v tele načítajú už zmenenú hodnotu premennej a (1) a opäť ju inkrementujú. Výsledok výpočtu bude preto 2.

Nasledujúci príklad poukazuje na problém pri riešení konfliktov pri prístupe do pamäte ak sa použije synchronizácia na úrovni procesorov. Je daný nasledujúcu program pre EREW PRAM (% označuje operáciu zvyšok po delení):

```
shared array a;
for i := 0 to 9 pardo
    a[ i % 5] := 1;
```

Očakávaný výsledok behu programu je konflikt pri zapisovaní do pamäte. Procesy s hodnotou identifikátora i 0 a 5 budú chcieť zapísať do pola a na pozíciu 0. Keďže nemáme povolené paralelné zápisy, program by mal skončiť s chybou. V skutočnosti je výsledok závislý od počtu procesorov a od plánovania úloh. Uvažujme opäť 5 procesorov. Predpokladajme, že plánovač vykoná procesy v nasledujúcich skupinách: $\langle 0, 4\rangle$ a $\langle 5, 9\rangle$. Pri tomto scenári nevzniknú konflikty, keďže v oboch skupinách zapisujú procesy na rôzne pozície v poli. Ak by však plánovač vytvoril skupinu, ktorá obsahuje už zmienené procesy 0 a 5, konflikt by vznikol. Keďže o plánovaní procesov nemôžeme predpokladať nič, výsledok programu je nedeterministický.

Autori emulátora si tieto problémy uvedomujú a ponúkajú čiastočné riešenie vo forme synchronizačných konštrukcií v jazyku. Týmto rozhodnutím sa jazyk približuje k programovaniu reálnych paralelných programov a dáva do rúk programátora silné prostriedky. Našim cieľom je však abstrahovať od týchto praktických problémov a umožniť tak sústrediť sa viac na samotný algoritmus a nie jej implementáciu na rôzne konkrétne platformy so špecifickými požiadavkami.

Tieto príklady demonštrovali, že mapovanie vyššoúrovňového jazyka na nižšiu úroveň spôsobuje rôzne problémy. Závislosť od počtu procesorov a plánovania úloh zapríčiňuje, že výpočty môžu dospieť k rôznym výsledkom a ich správanie je nedeterministické. Takéto riešenie pre nás nie je vhodné a preto sme zaviedli synchronizáciu na vyššej úrovni, kde ako jednotku neuvažujeme procesory ale procesy - tento mechanizmus sme nazvali **synchronizácia procesov**.

Náš cieľ je teda zaistiť aby synchronizácia brala do úvahy vzťahy medzi procesmi a konflikty sa riešili tiež na tejto úrovni. Plánovač funguje podobne ako pri predošlom prístupe, ale jedna iterácia cyklu nie je vykonanie inštrukcie pre podmnožinu procesov ale pre všetky. Opäť uvažujme príklad ak počet procesov je viac ako p. Plánovač na procesory namapuje p procesov, ktoré vykonajú len jednu inštrukciu. Potom plánovač prepne na ďalšiu skupinu procesov, ktoré tiež vykonajú jednu inštrukciu. Toto opakuje kým danú inštrukciu nevykoná každý proces. Týmto končí jedno kolo plánovania a len teraz sa riešia konflikty pamäte.

Tento cyklus plánovača funguje dobre, ale neošetruje prípady keď chceme kvôli rozsynchronizovaným procesom zaviesť ďalšiu kontrolu. Problém nastáva ak sa procesy pri podmienených skokoch (ktoré vznikajú pri všetkých konštrukciách ktoré obsahujú podmienku) rozhodnú vykonávať iné, poten-

ciálne rôzne časovo náročné vetvy výpočtu. V tomto prípade predpokladáme, že po vykonaní týchto vetiev, keď budú procesy rozsynchronizované, sa opäť „stretnú“ - zosynchronizujú sa. Túto požiadavku voláme *implicitná synchronizácia*. Hlavná myšlienka je zaistiť, aby v prípade ak procesy spolu vojdú do bloku, tak ho aj spolu opustili. Blok v tomto kontexte znamená *cyklus, if a pardo*. Ak proces vykoná svoju vetvu výpočtu rýchlejšie, tak na konci musí počkať, kým aj ostatné procesy dobehnú do toho bodu. Proces, ktorý takto čaká nazveme *spiaci proces*. Problém sa rieši zavedením *synchronizačných objektov*, ktoré použijeme na vytvorenie stromovej štruktúry, ktorá predstavuje procesy v rôznych vetvách výpočtu. Konkrétnejšie sa tejto problematike budeme venovať v 3.6.

2.5 Správa pamäte

Ako sme už v sekcii 1.3.2 písali, PRAM je model so zdieľanou pamäťou. Teraz predstavíme, ako sa s ňou pracuje. Aj keď pracujeme s jednou spoločnou pamäťou, každý proces si môže pre svoje potreby rezervovať jej časť. Preto z hľadiska programátora rozlišujeme dve časti pamäte:

- **Zdieľaná** - k tejto časti môže pristupovať ľubovoľný proces a preto môže dôjsť k súčasnému čítaniu alebo zápisu jej časti
- **Lokálna** - každý proces má vlastnú pamäť, v ktorej môže mať uložené dáta pre vlastnú potrebu. Ostatné procesy k nej nemajú prístup a preto nevznikajú konflikty.

Programátor s pamäťou interaguje deklaráciou premenných a ich následným používaním. Rozlišujeme preto zdieľané a lokálne premenné. Definícia premennej vyzerá nasledovne:

shared/local typ meno;

Prvá položka je nepovinná, ak sa neuvedie predpokladá sa *local*. Typy v súčasnosti podporujeme dva: celé čísla so znamienkom - *int* a polia - *array*. Polia je potrebné alokovať a po použití uvoľniť. Takto sa alokuje pole obsahujúce 10 čísel. Je možné vytvárať viacrozmerné polia ako pole polí.

```
local array pole;
pole := new(10, int);
pole[5] := 1;
delete pole;
```

Pri súčasnom prístupe do zdieľanej pamäte treba dbať na dodržanie pravidiel zvoleného modelu. Ak vznikne konflikt, ktorý nie je povolený program skončí s chybovou hláškou. Technickej časti implementácie pamäte sa venujeme v sekcii 3.7.

2.6 Vstup a výstup

Vstupno-výstupné operácie umožňujú programu interagovať s používateľom. Príklad načítania a vypísania premennej znázorňuje nasledujúci príklad:

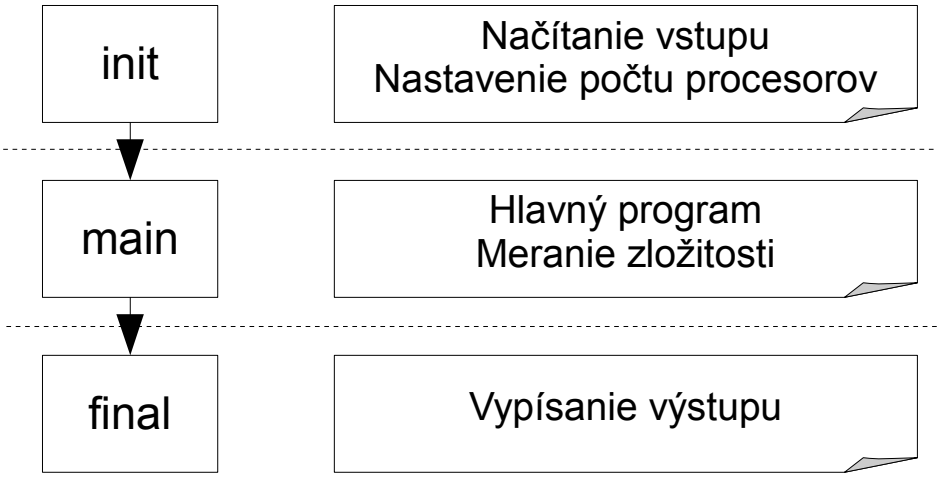
int a;
read a;
write a;

Tieto operácie je možné používať aj za prítomnosti viac procesov, vykonanie inštrukcií sa serializuje. Poradie vykonávania procesov nie je definované, preto môže dôjsť k nedeterministickému správaniu. Odporúča sa vstupno-výstupné operácie používať mimo paralelných blokov.

2.7 Beh programu

V tejto časti prezentujeme spôsob púšťania programov a rôzne fázy výpočtu. Prvé sú nastavenia modelu a počtu procesorov. Spôsob riešenia konfliktov sa nastavuje pred behom interpretra prepínačom *-t*. Po spustení programu sa vykoná inicializácia definovaná programátorom - špeciálna procedúra *init*. V programe sa táto metóda definuje ako ľubovoľná iná, ale iba ona vie nastaviť počet procesorov pomocou funkcie *setP*, ktorá akceptuje jeden parameter - počet požadovaných procesorov. V tejto metóde nie je možné vytvárať procesy, keďže slúži na inicializáciu. Toto je tiež správne miesto na načítanie vstupu, lebo čas strávený tu nie je započítaný do výslednej zložitosti. Dôvodom je, aby sa dali merať algoritmy s menšou ako lineárnou zložitosťou.

Po inicializácii sa spustí metóda *main*, v ktorej už nie je možné ďalej meniť počet procesorov. Tu sa odohrá hlavný výpočet s prípadným použitím ďalších metód a funkcií. Po jej vykonaní nasleduje nepovinná záverečná funkcia *final*, ktorá sa dá použiť na vypísanie väčšieho výstupu bez započítania do celkového času. Fázy behu programu sú znázornené na obrázku 2.1.



Obr. 2.1: Fázy behu programu

Kapitola 3

Implementácia

Obsahom tejto kapitoly je technický popis programovacieho jazyka a virtuálneho stroja pre PRAM spolu s nízkoúrovňovím jazykom. Sú tu prezentované ďalšie detaily správy pamäte a podrobnosti synchronizácie procesov. V krátkosti tiež predstavíme preklad vyššoúrovňového jazyka na assembler pre virtuálny stroj.

3.1 Úvod

Táto kapitola obsahuje detaily implementácie simulátora PRAMu. Pred konkrétnejším popisom tu načrtneme koncept architektúry. Výsledkom tejto práce je interpreter paralelných programov, ktorý sa skladá z viac častí. Vstupom preň je zdrojový kód napísaný v nami navrhnutom programovacom jazyku spolu so vstupom pre tento program a výstupom je výstup simulácie daného programu na danom vstupe. Spracovanie zdrojového kódu začína prekladom do počítačom jednoduchšie interpretovateľného nízkoúrovňového jazyka - *assembleru*. Následne sa tento kód vykoná na virtuálnom stroji a dostaneme tak výsledok simulácie.

3.2 Virtuálny stroj

Ako prvé predstavíme virtuálny stroj, na ktorom chceme vykonávať preložené programy. Z dôvodu jednoduchosti sme sa rozhodli pre zásobníkový virtuálny stroj. Meno naznačuje, že argumenty inštrukcií sa ukladajú na internom zásobníku a práca s pamäťou sa redukuje na načítanie pamäťového miesta na zásobník a uloženie obsahu zásobníka do pamäte. Zdrojový kód sa skladá z dvojíc (inštrukcia, argument), kde druhá položka je často nulová.

V nasledujúcej časti prezentujeme kompletnú inštrukčnú sadu virtuálneho stroja s ich krátkym popisom. Účel niektorých inštrukcií spresníme v ďalších sekciách.

3.2.1 Inštrukčná sada

Inštrukcie vo všeobecnosti očakávajú argumenty okrem svojho druhého komponentu na zásobníku. Počas jej vykonania sa tieto argumenty zo zásobníku odstránia a ak je potrebné sú nahradené výsledkom operácie.

Riadiace inštrukcie

Tieto inštrukcie riadia smer vykonávania kódu. Okrem poslednej inštrukcie majú všetky argument adresy v kóde. Inštrukcia *JMP* zmení hodnotu *IP* na danú adresu. Inštrukcie *JMPFALSE* a *JMPTRUE* sú analogické, ale sú podmienené výsledkom logickej operácie, ktorá je na zásobníku. Inštrukcia *CALL* zavolá funkciu na danej adrese a zaistí, aby po jej vykonaní sa riadenie vrátilo na nasledujúcu inštrukciu a stará sa tiež o odovzdávanie parametrov. Inštrukcia *RET* slúži na návrat na uloženú adresu poslednej inštrukcie *CALL* a vrátenie hodnoty, ktorá je na zásobníku ako výsledok funkcie.

Inštrukcia	**Argument**
JMP	adresa
JMPFALSE	adresa
JMPTRUE	adresa
CALL	adresa
RET	0

Tabuľka 3.1: Riadiace inštrukcie

Práca s pamäťou

Tieto inštrukcie slúžia na načítanie hodnoty z pamäte na zásobník a na zápis hodnoty zo zásobníku do pamäte. Okrem toho sú tu uvedené inštrukcie na prácu s dynamickou pamäťou. Pre podrobnosti o fungovaní pamäte viď sekciu 3.7. Inštrukcia *LOAD_CONST* načíta číselnú konštantu na zásobník. Inštrukcie *LOAD_LOC*, *LOAD_SHARE*, *STORE_LOC* a *STORE_SHARE* slúžia na priamy prístup do pamäte. Inštrukcie na čítanie načítajú hodnotu z pamäťového miesta na príslušnej adrese na zásobník. Inštrukcie na zápis naopak zapíšu hodnotu zo zásobníka na miesto do pamäte určené adresou. Inštrukcia *STORE_ARR_LOC* má 3 argumenty na stacku: pointer na pole,

index do pola a hodnotu, ktorú chceme zapísať. Inštrukcia *DEREF*, ktorá na zásobníku očakáva pointer na pole a index zapíše na zásobník hodnotu na príslušnom indexe v danom poli. Inštrukcia *NEW* vytvorí nové pole daného typu a uloží pointer naň na zásobník. Veľkosť pola očakáva na zásobníku. Inštrukcia *DELETE* uvoľní pole, na ktorý ukazuje pointer na zásobníku. Inštrukcia *SIZEOF* zistí veľkosť pola, na ktorý ukazuje pointer na zásobníku a uloží ho na zásobník.

Inštrukcia	**Argument**
LOAD_CONST	hodnota
LOAD_LOC	adresa
LOAD_SHARE	adresa
STORE_LOC	adresa
STORE_SHARE	adresa
STORE_ARR_LOC	0
DEREF	0
NEW	typ
DELETE	0
SIZEOF	0

Tabuľka 3.2: Práca s pamäťou

Aritmetické a logické operácie

Táto skupina inštrukcií vykonáva aritmetické a logické operácie. Operandy operácií sú na zásobníku (prvý je pravý operand) a výsledok sa tiež uloží na zásobník. Pre jednoduchosť vysvetlivky k týmto inštrukciám uvádzame v tabuľke 3.3.

Systémové volania

Túto skupinu reprezentuje jediná inštrukcia *SYS*, ktorá je zodpovedná za vykonávanie systémových volaní. Volanie sa konkretizuje druhým parametrom. Týmto mechanizmom sme zaistili jednoduchú možnosť pridávania ďalšej funkcionality podľa potreby. Súčasné volania prezentuje tabuľka 3.4. Vstupy a výstup volaní sa ukladajú na zásobník podobne ako pre bežné funkcie.

Správa procesov

Tieto inštrukcie slúžia na vytváranie a synchronizovanie procesov. Inštrukcia *pardo* slúži na vytvorenie nových procesov. Na zásobníku očakáva: rozsah

Inštrukcia	Argument	# parametrov	Operácia
ADD	0	2	sčítanie
SUB	0	2	odčítanie
MUL	0	2	násobenie
DIV	0	2	delenie
SHL	0	2	bitový posun doľava
SHR	0	2	bitový posun doprava
EQ	0	2	rovná sa
LESS	0	2	menšie
GRT	0	2	väčšie
LEQ	0	2	menšie alebo rovné
GEQ	0	2	väčšie alebo rovné
LAND	0	2	logické a
LOR	0	2	logické alebo
OR	0	2	bitové alebo
AND	0	2	bitové a
XOR	0	2	bitové xor

Tabuľka 3.3: Aritmetické a logické operácie

Argument	Operácia
READ	čítanie zo vstupu
WRITE	zápis na výstup

Tabuľka 3.4: Systémové volania

cyklu *pardo* (v poradí max, min) a ďalšie parametre (pre účely vnorených *pardo* cyklov). Vytvorí sa príslušný počet procesov, ktoré zdedia parametre na zásobníku a začnú vykonávať kód na danej adrese. Inštrukcia *SETP* nastavuje počet procesorov a môže sa použiť len v inicializačnej metóde. Zvyšné inštrukcie slúžia na synchronizáciu, tie predstavíme spolu s podrobnejším popisom správy procesov v 3.6.

3.3 Kompilátor

V tejto časti prezentujeme proces prekladu programovacieho jazyka pre PRAM na nízkoúrovňový jazyk virtuálneho stroja. V krátkosti popíšeme priebeh kompilácie a následne predstavíme výslednú reprezentáciu kódu. Našim cieľom nie je skúmanie techník vytvárania kompilátorov, preto sa tejto problematike venujeme len minimálne. Jednotlivé fázy kompilácie sú znázornené

Inštrukcia	**Argument**
PARDO	adresa
SETP	0
SYNC	0
JMPFALSE_SYNC	adresa
SYNC_BEGIN	0

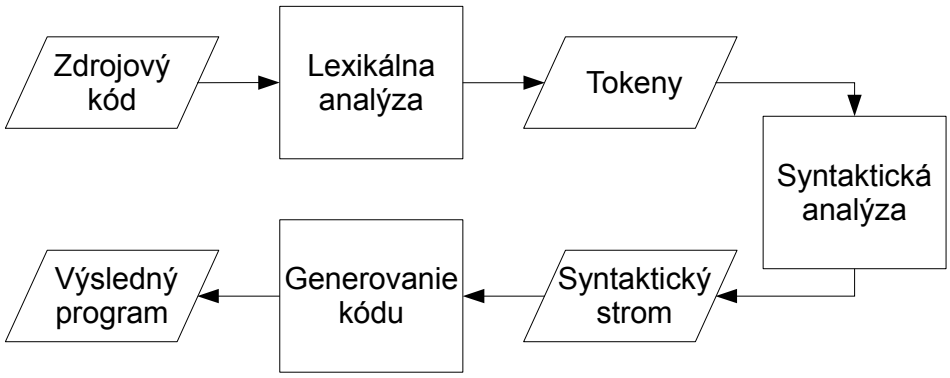
Tabuľka 3.5: Riadiace inštrukcie

na obrázku 3.1.

Prvým krokom prekladu je *lexikálna analýza*, ktorej cieľom je rozoznať postupnosti znakov tvoriacich logické časti, tzv. *lexémy*. Postupne spracúva zdrojový kód a generuje postupnosť *tokenov*, ktoré reprezentujú tieto lexémy. Vzory sa definujú pomocou regulárnych výrazov a rozoznávajú sa použitím konečných automatov. Cieľom tejto fázy je zjednodušiť ďalšie spracovanie vstupu.

Druhá fáza je *syntaktická analýza*, ktorá dostáva svoj vstup od lexikálnej analýzy vo forme postupnosti tokenov. Jej cieľom je rozoznať konštrukcie jazyka a vygenerovať *syntaktický strom*. Tieto konštrukcie sa definujú bezkontextovou gramatikou.

Nasleduje fáza generovania kódu, počas čoho sa traverzovaním syntaktického stromu a za pomoci *tabuľky symbolov* vytvorí výsledok kompilácie - program pre virtuálny stroj.



Obr. 3.1: Fázy kompilácie

Na implementáciu sme použili jazyk C++ a prvé dve fázy nám značne zjednodušili nástroje na tvorbu kompilátorov Flex a Bison. Samotný preklad nebudeme podrobnejšie predstavovať okrem vybraných častí, ktoré sa týkajú

hlavne paralelných konštrukcií. Tieto uvádzame v ďalších sekciách.

Na záver prezentujeme výslednú formu reprezentácie kódu pre virtuálny stroj. Základnou jednotkou sú dvojice (Inštrukcia, Argument), ktoré tvoria program. Rozhodli sme sa rozdelenie programu na funkcie zachovať aj na tejto úrovni. Ako jednotku okrem funkcií uvažujeme aj telá *pardo* cyklov. Tieto logické časti reprezentujeme *kódovým objektom*. Jej hlavné časti sú: meno, pole inštrukcií, tabuľka symbolov pre lokálne premenné a počet parametrov. Takéto štruktúry sa vytvárajú počas finálnej časti kompilácie a používajú sa neskôr ako zdroj inštrukcií pre interpreter.

3.4 Interpreter

Srdcom simulátora PRAMu je virtuálny stroj, ktorý interpretuje paralelné programy. Pre jednoduchosť sme tento proces rozdelili na niekoľko úrovní.

Základná jednotka interpretra je objekt reprezentujúci rámec v zásobníku volaní - *frame object*. Každej aktivácii funkcie zodpovedá vytvorenie jednej inštancie tohto objektu. Jeho hlavné komponenty sú: referencia na prislúchajúci kódový objekt, počítadlo inštrukcií (instruction pointer - IP), lokálna pamäť, zásobník pre výrazy a stav.

Vyššia úroveň je reprezentovaná objektom pre proces - *process object*. Jeho hlavné časti sú: zásobník volaní (ktorý obsahuje vyššie predstavené rámce), referencia na zdieľanú pamäť, referencia na množinu kódových objektov, pointer na synchronizačný objekt (viď 3.6) a stav.

Priebeh interpretácie riadi plánovač úloh - *scheduler*. Jeho cieľom je udržiavať zoznam aktívnych procesov, vykonávať ich inštrukcie a ukladať informácie o počte vykonaných inštrukcií. Inicializácia plánovača pre hlavnú metódu *main* prebieha nasledovne: vytvorí sa zdieľaná pamäť a následne jeden proces, ktorý sa zaradí do zoznamu aktívnych. Proces si vytvorí prvý rámec pre svoj zásobník volaní. Ten si nastaví kódový objekt pre metódu main a IP na jej začiatok. Ďalej sa v rámci vytvorí lokálna pamäť podľa tabuľky symbolov pre lokálne premenné v kódovom objekte.

Po inicializácii sa začne vykonávať aktívny proces. Zatiaľ predpokladajme, že sa ďalšie procesy nevytvárajú. Správu procesov a ich synchronizáciu predstavíme v sekcii 3.5. Vykonávanie inštrukcií sa odohráva na troch úrovniach: rámec, proces a plánovač. Teraz popíšeme prvé dve z nich. V rámci sa interpretujú nasledujúce inštrukcie: práca s pamäťou, aritmetické a logické operácie, nesynchronizované skoky a systémové volania. Ostatné nastavia špecifický stav pre rámec a riadenie sa odovzdá procesu. Inštrukcie na tejto úrovni majú priamy prístup k lokálnej pamäti a k zásobníku na vyhodnocovanie výrazov. K zdieľanej pamäti vedia pristupovať len cez proces. Proces vykonáva

volanie funkcií a návrat z nich. Okrem toho je zodpovedný za vytvorenie nových procesov, ktoré neskôr plánovač pridá do zoznamu aktívnych. Interpretácia týchto inštrukcií je v súlade s ich definíciou v sekcii 3.2.1. Ostatné inštrukcie - správu procesov a synchronizáciu - vykonáva plánovač.

3.5 Plánovač úloh

Cieľom plánovača úloh je zaistiť správne poradie vykonávania inštrukcií, vytváranie a manažovanie procesov a ich synchronizácia. V tejto časti predstavíme životný cyklus procesov, hlavný cyklus plánovača, prekladanie paralelných konštrukcií a synchronizáciu procesov.

Ako prvé, plánovač vykoná špeciálnu metódu *init*, ktorá ako jediná má možnosť nastaviť počet procesorov PRAMu pomocou inštrukcie *SETP*. Následne sa vykoná metóda *main*, ktorá obsahuje hlavnú časť algoritmu. Na vytvorenie nových procesov slúžia dve jazykové konštrukcie: *pardo* cyklus a parelelné metódy.

Pardo cyklus v programe zapisujeme nasledovne:

$$\textbf{for } i := min \textbf{ to } max \textbf{ pardo } prikaz$$

Príkaz môže znamenať aj blok príkazov. Táto konštrukcia sa počas kompilácie preloží ako špeciálne volanie funkcie, ktorá obsahuje telo cyklu. Volanie sa uskutoční inštrukciou *PARDO*, ktorej parameter je adresa vytvorenej funkcie a na zásobníku očakáva rozsah cyklu (max, min) a zdedené argumenty v prípade vnorených *pardo* cyklov. Interpretácia inštrukcie je nasledovná: vytvorí sa $max - min + 1$ procesov, ktorých zásobník volaní bude obsahovať jeden rámec s kódovým objektom reprezentujúcim telo cyklu. Každý proces dostane na svojom zásobníku výrazov zdedené parametre a hodnotu identifikátora i - číslo z intervalu $\langle min, max \rangle$. Tieto parametre jednoznačne určujú proces. Následne sa stav rodičovského procesu zmení na spiaci a aktivuje sa až všetky jeho deti skončia. Keďže táto konštrukcia sa rieši mechanizmom pre funkcie telo cyklu nemá prístup k lokálnym premenným volajúceho procesu. Tento fakt je dôležitý, keďže procesy smú komunikovať výlučne cez premenné, ktoré boli definované ako zdielané. Proces končí keď sa jeho zásobník volaní vyprázdni, t.j. skončí vykonávanie tela cyklu.

Druhá možnosť vytvárania procesov je paralelná metóda. Definuje sa kľúčovým slovom *parallel* pred definíciou metódy. Počas prekladu sa vytvorí špeciálna funkcia, ktorá sa dá aktivovať inštrukciou *CALL*. Počas volania sa zistí, že to je paralelná metóda a vytvorí sa toľko procesov koľko má PRAM procesorov. Každý proces inicializuje svoj zásobník volaní rámcom pre volanú metódu. Na zásobníku výrazov dostanú identifikátor procesu a počet

procesorov. Po vytvorení procesov sa rodičovský proces deaktivuje podobne ako v predošlom prípade a aktivuje sa až deti ukončia svoj beh.

Dôležité obmedzenie je, že tieto dva mechanizmy nie je možné kombinovať, t.j. z *pardo* cyklu nie je možné volať paralelné metódy a paralelné metódy nemôžu obsahovať *pardo* cykly.

Po predstavení vytvárania a zanikania procesov sa môžeme venovať ich plánovaniu. Základom tohto procesu je hlavný cyklus interpretra, ktorý zaisťuje mapovanie procesov na procesory a vykonávanie inštrukcií. Ako sme už písali, našim cieľom je zaistiť synchronizáciu na úrovni procesov. Za jedno kolo plánovania preto považujeme vykonanie inštrukcie pre každý aktívny proces. Hlavný cyklus pracuje nasledovne: z aktívnych procesov si vyberie skupinu veľkosti maximálne p (počet procesorov) a tieto namapuje na procesory. PRAM vykoná jednu inštrukciu. Ak ostali ešte v tomto kole nevykonané procesy opakuje cyklus. Na konci všetky procesy zvýšili svoje počítadlo inštrukcií a môže sa vykonať ďalšie kolo.

Prezentovali sme základ plánovania procesov. Doteraz predstavená funkcionalita zabezpečí aby bolo možné vytvárať dynamicky nové procesy. Takéto plánovanie však zatiaľ neumožňuje synchronizáciu na úrovni procesov, čo sme definovali ako cieľ v sekcii 2.4. Okrem toho chýba riešenie pamäťových konfliktov. Tieto problémy riešime v ďalších sekciách zavedením nových konštrukcií.

3.6 Synchronizačné objekty

Pripomeňme si pojem implicitnej synchronizácie, ktorý sme zaviedli v sekcii 2.4. Počas vykonávania programu procesy zvyšujú svoje lokálne počítadlo inštrukcií synchronizovane, čo znamená, že vykonávajú rovnaké inštrukcie. Tento stav sa môže zmeniť počas vykonania inštrukcie podmieneného skoku. Keďže procesy môžu robiť s rôznymi dátami, môže sa stať, že podmienku skoku vyhodnotia rôzne. To zapríčiní, že sa jedna synchronizovaná skupina procesov rozdelí na dve, ktoré budú vykonávať rôzne vetvy výpočtu. Tento proces nazývame rozsynchronizovanie. Podmienené skoky vznikajú pri preklade konštrukcií s podmienkou (if a cykly), ktoré prirodzene definujú blok príkazov. Našim cieľom je zaistiť aby sa procesy po vykonaní bloku opäť zosynchronizovali. Táto požiadavka implikuje, že procesy, ktoré vykonajú „kratšiu“ vetvu musia ostatné „počkať“.

Pri predošlých úvahách sa prirodzene objavil pojem skupín synchonizovaných procesov. Ak si ďalej predstavíme želaný beh programu, všimneme si, že nové skupiny sa vytvoria, ak niektorá skupina vykoná inštrukciu podmieneného skoku. Vzniknuté dve skupiny po čase ukončia svoje rôzne vetvy

výpočtu a opäť sa zosynchronizujú, t.j. skupiny sa zlúčia, čím vytvoria pôvodnú skupinu z pred skoku. Tento proces môžeme vizualizovať pomocou stromovej štruktúry, kde budú skupiny reprezentované uzlami. Začnime jednou skupinou procesov, jedným uzlom v strome. Počas rozsynchronizovania vzniknú z pôvodnej skupiny dve nové. Tento vzťah v strome reprezentujeme pridaním dvoch detí uzlu. Tento proces sa môže opakovať, čím sa vytvorí hlbší strom. Po čase však dôjde k ukončeniu niektorej vetvy, čo v strome reprezentujeme odobratím príslušného uzla. V tomto stave dostávame pôvodný uzol s jedným synom a očakávame, že sa ďalej bude vykonávať len skupina syna. Počas toho je druhá skupina inaktívna a hovoríme, že čaká v rodičovskom uzle. Po ukončení skupiny syna sa aj tento uzol zo stromu odstráni. Teraz dostávame opäť jeden uzol a chceme, aby sa spiace procesy zobudili a vykonávali ďalej kód s práve dobehnutými procesmi.

Zovšeobecnením tohto príkladu sme dospeli k návrhu stromovej štruktúry, kde každý uzol reprezentuje skupinu synchronizovaných procesov. Z hľadiska procesu príslušnosť do skupiny je ekvivalentná so zdieľaním uzla v strome, preto v ďalšom texte budeme zamieňať pojmy skupina procesov a uzol v strome. Ďalej definujeme požiadavky na vytváranie tejto štruktúry. Na základe príkladu sa ľahko nahliadne fakt, že aktívne sú práve tie uzly, ktoré nemajú deti. Pridanie detí (rozsynchronizovanie) pre rodičovský uzol znamená čakanie na skončenie oboch detí. Procesy rodičovského uzla sa priradia deťom podľa vyhodnotenia podmienky. Tento bod je dôležitý, keďže proces môže naraz patriť iba do jednej skupiny. Odobratie uzla zo stromu znamená „vrátenie“ príslušných procesov rodičovi. Ak rodič už nemá deti všetky svoje procesy zobudí, inak ich deaktivuje.

Implementácia tejto myšlienky má dve úrovne. Počas prekladu vybraných konštrukcií treba použiť špeciálne, synchronizačné inštrukcie, ktoré majú za úlohu riadiť správu stromovej štruktúry. Druhá úroveň je správna interpretácia týchto inštrukcií.

Najprv prezentujeme riešenie druhej úrovne - *synchronizačné objekty*. Tieto objekty tvoria uzly vytváraného stromu. Každý proces má priradený synchronizačný objekt, ktorý určuje jeho pozíciu v strome. Okrem toho má každý proces stav: aktívny alebo spiaci. Synchronizačné objekty spravuje plánovač úloh. Objekty sú jednoduché, pamätajú si zoznam priradených procesov, svoje deti a rodiča v strome.

Nasleduje popis inštrukcií paralelizmu vzhľadom na strom synchronizačných inštrukcií. Inštrukcia *PARDO* slúži na vytváranie procesov. V strome to znamená pridanie jedného syna aktívnemu uzlu s novovytvorenými procesmi. To zaistí, že rodičovský proces sa aktivuje až všetky jeho deti skončia. Inštrukcia *SYNC* slúži na ukončenie synchronizovaného bloku. Aktívny proces sa odstráni zo zoznamu prislúchajúceho uzla a priradí sa rodičovi. Ak

pôvodný uzol neobsahuje ďalšie procesy odstráni sa zo stromu. Ak nový uzol ešte má aktívne dieťa proces zaspí. Ak sa odstránením dieťaťa stal rodič listom zobudí všetky svoje procesy. Inštrukcia *JMPFALSE_SYNC* sa používa pri preklade štruktúry *if*. Spôsobí vytvorenie dvoch detí aktívnemu uzlu, do ktorých sa procesy presunú podľa vyhodnotenia podmienky. Inštrukcia *SYNC_BEGIN* sa používa pri preklade cyklov. V tomto prípade by mohlo dôjsť k zbytočnému prehlbovaniu stromu a preto sme zvolili tento spôsob. Používa sa na začiatku cyklu, čím sa vytvorí nový synchronizovaný blok (na koniec sa pridá inštrukcia *SYNC*). Jej interpretácia je nasledovná: aktívnemu uzlu sa pridá jeden syn, do ktorého sa priradia všetky procesy. Tento mechanizmus zaručí aby všetky procesy ukončili cyklus naraz.

Teraz predstavíme ako sa tieto inštrukcie použijú pri preklade. Ako príklad uvádzame konštrukciu if-then-else a while. Ostatné sa prekladajú analogicky.

	1	*vyhodnotenie podmienky*
if (podmienka) **then**	2	JMPFALSE_SYNC 5
príkaz1	3	*príkaz1*
else	4	JMP 6
príkaz2	5	*príkaz2*
	6	SYNC 0

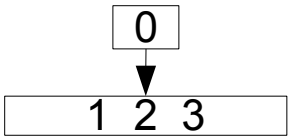
Tabuľka 3.6: Preklad konštrukcie if-then-else

	1	SYNC_BEGIN 0
	2	*vyhodnotenie podmienky*
while (podmienka)	3	JMPFALSE 6
príkaz	4	*príkaz*
	5	JMP 2
	6	SYNC 0

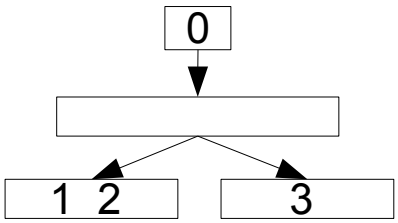
Tabuľka 3.7: Preklad konštrukcie while

Pre lepšiu názornosť ukážeme použitie synchronizačných objektov na príklade. Uvedený zdrojový kód sme zjednodušili, aby bol kratší. V prvom kroku sa vytvoria 3 procesy a do stromu sa pre ne vytvorí nový uzol. V druhom kroku príkaz if rozdelí procesy na dve skupiny, do stromu pribudnú pre ne dva nové uzly.

```
0 ⇒   for i := 1 to 3 pardo
        if (i < 3) then
          if (i < 2) then
            príkaz1
          sync
        sync
```

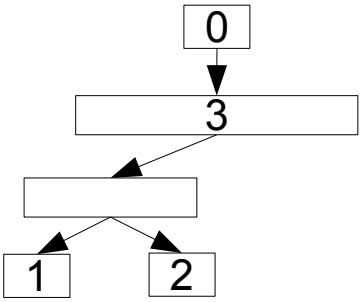



```
      for i := 1 to 3 pardo
1,2,3 ⇒   if (i < 3) then
            if (i < 2) then
              príkaz1
            sync
          sync
```



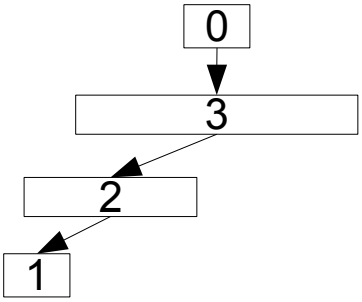
V tomto kroku sa procesy rozsynchronizovali. Procesy 1 a 2 vykonajú ďalší príkaz if, čím sa strom opäť prehĺby. Proces 3 vykoná inštrukciu SYNC, čo spôsobí odobratie príslušného uzla a priradenie procesu rodičovskému uzlu. Zároveň je tento proces deaktivovaný, keďže jeho nový uzol nie je list.

```
      for i := 1 to 3 pardo
        if (i < 3) then
1,2 ⇒     if (i < 2) then
            príkaz1
          sync
3 ⇒     sync
```



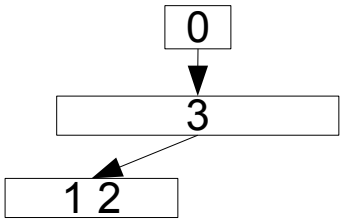
V ďalšom kroku proces 1 vykonáva telo ifu, proces 2 vykonal inštrukciu SYNC, preto sa príslušný uzol odstráni zo stromu. Proces 2 čaká v rodičovskom uzle. Proces 3 je stále v stave spiaci.

```
      for i := 1 to 3 pardo
        if (i < 3) then
          if (i < 2) then
1 ⇒         príkaz1
2 ⇒       sync
3 ⇒     sync
```



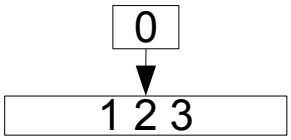
V tomto kroku proces 1 skončí telo ifu a vykoná inštrukciu SYNC. Príslušný uzol sa odstráni zo stromu. Rodičovský uzol sa stal listom, a preto sa zobudí čakajúci proces 2.

```
                for i := 1 to 3 pardo
                  if (i < 3) then
                    if (i < 2) then
                      príkaz1
1, 2 ⇒              sync
3⇒                sync
```



Procesy 1 a 2 dosiahli záverečnú inštrukciu SYNC, to znamená, že oba procesy sa odstránia z príslušného uzla. Ten sa odstráni zo stromu, keďže sa stal prázdnym. Rodičovský uzol je opäť list, preto sa zobudí proces 3.

```
              for i := 1 to 3 pardo
                if (i < 3) then
                  if (i < 2) then
                    príkaz1
                  sync
1, 2, 3 ⇒       sync
```



V ďalšom kroku procesy 1, 2 a 3 skončia, odstránia sa z príslušného uzla, ktorý sa zo stromu odstráni. Zobudí sa proces 0 a pokračuje vykonávaním programu (ten už neuvádzame).

Zavedením synchronizačných objektov a pridaním vyššie spomenutých inštrukcií sme dosiahli cieľ: písať programy vo vysokoúrovňovom jazyku a možnosť spoliehať sa na implicitnú synchronizáciu, bez nutnosti pridania zložitejších konštrukcií do jazyka.

3.7 Správa pamäte

Správa pamäte umožňuje pristupovať k premenným uložených v pamäti. Ako sme už v časti 3.2 pri popise inštrukcií ukázali, virtuálny stroj pristupuje do pamäte vždy cez lokálny zásobník výrazov. Toto rozhranie podporuje objekt *memory*, ktorý umožňuje čítať a zapisovať objekty *memObject*.

Z hľadiska návrhu virtuálneho stroja pre PRAM bolo najdôležitejšie zaistiť aby konflikty vznikajúce pri paralelnom prístupe do pamäte boli správne riešené. Tieto konflikty vznikajú ak dva alebo viac procesov počas jedného kola (ako bolo definované v sekcii 3.5) chce prečítať alebo zapísať rovnaké pamäťové miesto.

Ako prvé treba zabezpečiť, aby sa evidoval zoznam použitých pamäťových miest (kde použité znamená prečítané alebo zapísané). Pre tento účel si objekt *memory* udržiava dve množiny - prečítané a zapísané pamäťové miesta. Do týchto množín sa pridávajú prvky pri každom prístupe do pamäte a zároveň sa mažú na konci kola plánovača. Toto zaistí, aby množiny obsahovali všetky potenciálne konfliktné pamäťové miesta.

Tieto množiny stačia pre účely detekcie konfliktov pre modely EREW a CREW. Pri každom prístupe sa kontroluje, či príslušná množina obsahuje dané pamäťové miesto. Ak áno, tak došlo ku konfliktu a virtuálny stroj skončí s chybovou hláškou.

Implementovali sme dva typy modelu CRCW: common-CRCW a arbitrary-CRCW. Model priority-CRCW sme vynechali, lebo nie je dobre definovaný pre procesy vznikajúce pri vnorených *pardo* cykloch. Pre riešenie konfliktov modelu common-CRCW postačuje množina zapísaných pamäťových miest. Ak sa chystá zapísať v danom kole už zmenené pamäťové miesto, tak sa skontroluje, či sa zapisovaná hodnota zhoduje s hodnotou v pamäti. Ak nie, tak sa operácia zamietne a virtuálny stroj skončí s chybovou hláškou.

Pre účely modelu arbitrary-CRCW sa okrem množiny zapísaných pamäťových miest evidujú aj hodnoty zápisov. Na konci kola sa do pamäte zapíše jedna z týchto hodnôt. Výber sa uskutoční náhodne, čo implikuje nedeterminizmus pri tomto modely.

Tieto mechanizmy zaistia, aby pamäť fungovala podľa definície v sekcii 1.3.2. Je dôležité, že celý proces je pre programátora transparentný a k zdieľanej pamäti pristupuje rovnako ako k lokálnej. Pri porušení pravidiel modelu o tom virtuálny stroj informuje vo forme chybovej hlášky.

Kapitola 4

Príklad použitia

Táto kapitola prezentuje príklad použitia prostredia na príklade. Demonštruje tvorbu programu a ukáže ako sa dá implementovaný program analyzovať z hľadiska zložitosti.

4.1 Zadanie

Zvolili sme jednoduchý problém rátania sumy prvkov pola. Na vstupe máme n čísiel a výstup je ich suma.

4.2 Riešenie

Základom algoritmu je operácia, ktorá zredukuje veľkosť pola na polovicu tak, aby sa zachovala suma prvkov. Paralelne se sčítajú susediace prvky po dvojiciach a vytvoria tým vstup pre ďalšie kolo. Keďže pole sa v každom kroku zmenší na polovicu počet kôl je $O(\log n)$. Každé kolo vieme vykonať paralelne v konštantnom čase, preto bude výsledná časová zložitosť tiež $O(\log n)$. Počet operácií na zredukovanie pola na polovicu je linárna od dĺžky pola. Keďže každou redukciou sa pole zmenší na polovicu celková práca bude $O(n)$.

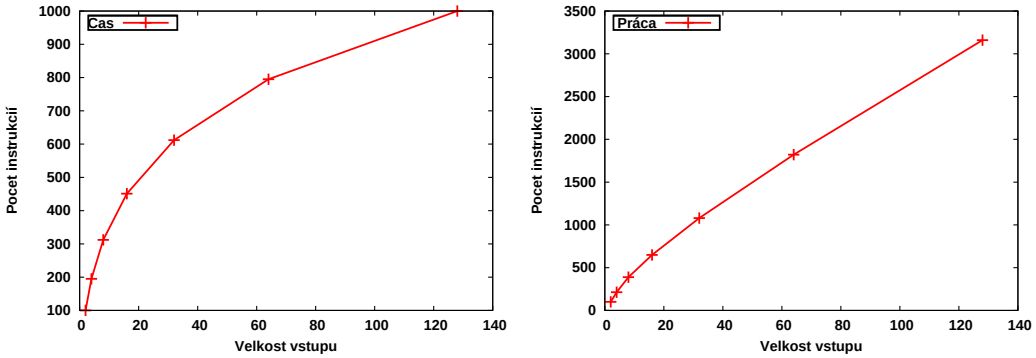
Nasleduje kompletný algoritmus:

```
// funkcia vrati prvu mocninu dvojky vacsiu ako n
int toPowOf2(int n)
begin
  if (n & (n-1) = 0) then //je to mocnina dvojky
    return n;
  else
    begin
```

```
      int res;
      res := 1;
      while (res < n)
        res := res << 1;
      return res;
    end
end

int log2(int num)
begin
  int n;
  n := 0;
  while ((1 << n) < num)
    n := n + 1;
  return n;
end

int init()
begin
  shared int n;
  read n; // nacitame pocet prvkov
  int arrsize;
  arrsize := toPowOf2(n);
  shared array A;
  A := new(arrsize+1, int);
  int i;
  for i:=1 to n do
    read A[i]; // nacitame cisla
  setP(arrsize / 2 + 1); // nastavime pocet procesorov
end

int main()
begin
  int h;
  for h:=1 to log2(n) do
    for i:=1 to (sizeof(A)-1)/(1<<h) pardo
      A[i] := A[2*i - 1] + A[2*i];

  write A[1]; // suma prvkov
  delete A;
end
```

4.3 Analýza

Na základe teoretickej analýzy vieme, že algoritmus má zložitosť $O(\log n)$ pre čas a $O(n)$ pre prácu. Namerali sme tieto hodnoty aj pomocu simulátora. Výsledky sú konzistentné s predpokladmi a sú znázornené na nasledujúcich grafoch. Veľkosť vstupu sme volili ako mocniny dvojky, keďže algoritmus pracuje nad polom takej veľkosti. Počet procesorov sme nastavili na $n/2$, lebo toto je maximálny počet súčasne vykonávaných procesov.



Obr. 4.1: Čas a práca algoritmu

Pre úplnosť uvádzame aj namerané hodnoty. Tieto môžu byť na rôznych verziach simulátora iné, kvôli inému prekladu konštrukcií.

n	2	4	8	16	32	64	128
Čas	100	195	312	451	612	795	1000
Práca	100	213	384	649	1080	1821	3160

Tabuľka 4.1: Namerané hodnoty

Kapitola 5

Záver

V tejto práci sme sa venovali problému vytvorenia prostredia na písanie paralelných algoritmov pre model PRAM. Navrhli sme programovací jazyk a implementovali sme virtuálny stroj, ktorý dokáže interpretovať paralelné programy písané v tomto jazyku. Tento systém je možné použiť ako didaktickú pomôcku na cvičenia k predmetom o paralelných algoritmoch.

Hlavné problémy pri implementácii priniesli netriviálne požiadavky na synchronizáciu procesov. Podarilo sa nám zabezpečiť synchronizáciu procesov v prostredí, v ktorom môžu rôzne procesy vykonávať rôzne vetvy programu, ktoré môžu byť rôzne časovo náročné. Zaviedli sme synchronizáciu na úrovni blokov, ktorá sa implementuje pomocou stromovej štruktúry reprezentujúcej pozíciu procesov vo vetvách programu.

Prostredie spĺňa požiadavky kladené v úvode práce, je možné priamočiaro implementovať algoritmy vo forme ako sa uvádzajú v literatúre. Systém podporuje písanie programov pomocou work-time prezentácie a tiež pre konkrétny procesor. Je možné merať zložitosť algoritmov pre dve základné miery: čas a práca.

Literatúra

[BAFR96] Yosi Ben-Asher, Dror G. Feitelson, and Larry Rudolph. Parcan extension of c for shared memory parallel processing. *Softw. Pract. Exper.*, 26:581–612, May 1996.

[Fly72] Michael J. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, C-21(9):948 –960, 1972.

[FW78] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, STOC '78, pages 114–118, New York, NY, USA, 1978. ACM.

[Ḧ92] Pasi Hämäläinen. Pram emulator - user's manual, 1992.

[HSS92] T Hagerup, A Schmitt, and H Seidl. Fork: A high-level language for prams. *Future Generation Computer Systems*, 8(4):379 – 393, 1992.

[JaJ92] Joseph JaJa. *An Introduction to Parallel Algorithms.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1992.

[Juv92] Simo Juvaste. An implementation of the programming language pm2 for pram, 1992.

[PBB+02] Wolfgang J. Paul, Peter Bach, Michael Bosch, Jörg Fischer, Cédric Lichtenau, and Jochen Röhrig. Real pram programming. In *Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, Euro-Par '02, pages 522–531, London, UK, 2002. Springer-Verlag.

[THT+90] Walter F. Tichy, Christian G. Herter, Walter F. Tichy, Christian G. Tlerter, Walter F. Tichy, and Christian G. Herter. Modula-2*: an extension of modula-2 for highly parallel, portable programs. Technical report, 1990.

[Vis11] Uzi Vishkin. Using simple abstraction to reinvent computing for parallelism. *Commun. ACM*, 54(1):75–85, January 2011.

Dodatok A

Gramatika jazyka

Tokeny:

```
T_BEGIN: begin
T_END: end
READ: read
WRITE: write
IF: if
THEN: then
ELSE: else
WHILE: while
DO: do
LOCAL: local
GLOBAL: global
SHARED: shared
FOR: for
TO: to
PARDO: pardo
RETURN: return
INT: int
ARRAY: array
NEW: new
DELETE: delete
PARALLEL: parallel
SIZEOF: sizeof
SETP: setP
ASSIGNOP: :=
SEMICOLON: ;
LPAREN: (
```

```
RPAREN: )
SHL: <<
SHR: >>
GEQ: >=
LEQ: <=
AND: &
LAND: &&
OR: |
LOR: ||
XOR: ^

INTEGER: {0-9}+
IDENTIFIER: [A-Za-z][A-Za-z0-9_]*
```

Gramatika:

```
program
: blocks

blocks
: blocks blok
| /* empty */

blok
: functiondef stmtblock

functiondef
: varType IDENTIFIER LPAREN arglist RPAREN
| varType IDENTIFIER LPAREN RPAREN
| PARALLEL varType IDENTIFIER LPAREN RPAREN

arglist
: arglist ',' varType IDENTIFIER
| varType IDENTIFIER


statements
: statements statement
| /* empty */

stmtblock
```

```
: T_BEGIN statements T_END

statement
: stmtblock
| assignment
| iteration
| READ IDENTIFIER SEMICOLON
| READ arr '[' expression ']' SEMICOLON
| WRITE expression SEMICOLON
| ifstatement
| functioncall SEMICOLON
| namespace varType IDENTIFIER  SEMICOLON
| RETURN SEMICOLON
| RETURN expression SEMICOLON
| DELETE arr SEMICOLON
| SETP expression SEMICOLON

varType
: INT
| ARRAY


functioncall
: IDENTIFIER LPAREN RPAREN
| IDENTIFIER LPAREN exprlist RPAREN

namespace
: /* empty */
| LOCAL
| GLOBAL
| SHARED

ifstatement
: IF expression THEN statement %prec REDUCE
| IF expression THEN statement ELSE statement

iteration
: WHILE expression statement
| DO statements WHILE expression SEMICOLON
| FOR IDENTIFIER ASSIGNOP expression TO expression DO statement
| FOR IDENTIFIER ASSIGNOP expression TO expression PARDO statement
```

```
assignment
: IDENTIFIER ASSIGNOP expression SEMICOLON
| arr '[' expression ']' ASSIGNOP expression SEMICOLON

arr
: IDENTIFIER
| arr '[' expression ']'

expression
: idexpression
| functioncall
| constant
| expression '+' expression
| expression '-' expression
| expression '*' expression
| expression '/' expression
| expression '>' expression
| expression '<' expression
| expression GEQ expression
| expression LEQ expression
| expression '=' expression
| expression SHL expression
| expression SHR expression
| expression AND expression
| expression OR expression
| expression XOR expression
| expression LAND expression
| expression LOR expression
| LPAREN expression RPAREN
| dereference
| NEW LPAREN expression ',' varType RPAREN
| SIZEOF LPAREN expression RPAREN

constant
: INTEGER

exprlist
: expression
| exprlist ',' expression
```

```
idexpression
: IDENTIFIER

dereference
: idexpression '[' expression ']'
| dereference '[' expression ']'
```

Dodatok B

CD

Na priloženom CD sa nachádzajú:

- zdrojové kódy programu
- príklady implementovaných algoritmov
- návod na obsluhu a inštaláciu