UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY
KATEDRA INFORMATIKY

Prostredie pre experimentovanie s PRAM-ami

BAKALÁRSKA PRÁCA

Dana Smažáková

Vedúca bakalárskej práce:
RNDr. Dana Pardubská, PhD.

9.2.1.Informatika BRATISLAVA, 2009

Čestné prehlásenie

Čestne prehlasujem, že som túto prácu napísala sama, s použitím len uvedenej literatúry.

V Bratislave dňa 15.6.09
Dana Smažáková

Poďakovanie

Chcela by som v prvom rade poďakovať svojim rodičom a celej svojej rodine za ich podporu. Ďalej by som chcela poďakovať vedúcej mojej bakalárskej práce a neposledne aj ostatným profesorom, ktorí prednášajú študentom informatiky na Fakulte matematiky, fyziky a informatiky Univerzity Komenského v Bratislave.

Abstrakt

Cieľom tejto práce bolo navrhnúť a naprogramovať prostredie v Jave pre experimentovanie s výpočtovým modelom Paralell Random Access Maschine. Vytvorila som model PRAMu a prehľadné prostredie, v ktorom môžeme manipulovať s týmto modelom pomocou grafického užívateľského rozhrania. Systém rešpektuje štruktúru PRAMu a je jednoduché doplniť ho o ďalšie inštrukcie. V tomto prostredí možno manipulovať s rôznymi typmi PRAMov. Konkrétne sú to: EREW, CREW a CRCW. Navyše typ CRCW môže pracovať v dvoch verziách: PRIORITY CRCW a COMMON CRCW.

Kľúčové slová:
PRAM, RAM

Obsah

1 Úvod

V súčasnosti napreduje vývoj vedy a najmä informatiky míľovými krokmi a doba, keď bol Turingov stroj jediným výpočtovým modelom, je už dávno preč. Existuje mnoho viac či menej známych výpočtových modelov. Najbližšie k reálnej počítačovej architektúre má výpočtový model Random Access Maschine, ktorého inštrukcie pripomínajú základné programovacie inštrukcie Assembleru.

Pri súčasnom vývoji sa však hľadajú cesty ako urýchliť výpočty na počítačoch. Jednou z ciest je vývoj rýchlejších procesorov, zväčšenie operačnej pamäti,... Iným pohľadom na vec je využitie paralelizmu. Preto vznikajú paralelné počítačové systémy. Používajú sa najmä na numericky extenzívne počítanie a v oblasti umelej inteligencie. K počítačovým systémom vznikajú rôzne výpočtové modely, ktoré sa snažia využívať paralelizmus. Parallel Random Access Maschine je prirodzeným rozšírením Random Access Maschine. Tento model je veľmi jednoduchý, vykonáva jednoduché inštrukcie, podobné inštrukciám Random Access Maschine, a zároveň výrazne urýchľuje výpočty.

Preto som sa rozhodla naprogramovať model Parallel Random Access Maschine a prostredie, ktoré umožňuje manipulovať s týmto modelom, písať v ňom jednoduché programy. Toto prostredie môžeme zároveň využiť na učenie sa a porozumenie tomuto výpočtovému modelu.

2 Definície a označenia

Skôr ako sa ďubeme zaoberať programovaním tohto prostredia, mali by sme si povedať, čo je presne Parallel Random Access Maschine (ďalej len PRAM) a ako je definovaný. Výpočtový model PRAM je model paralelného počítača, ktorý abstrahuje od fyzickej implementácie. Je to teda výpočtový model podobný reálnym paralelným architektúram.

PRAM je systém potenciálne nekonečne veľa samostatných výpočtových jednotiek, ktoré nazývame procesory. Každý z týchto procesorov má k dispozícii svoju vlastnú sadu registrov, nazývanú pamäť, a svoju vlastnú sadu inštrukcií, nazývanú tiež program. Navyše majú prístup k spoločnej sade registrov, pomocou ktorej spolu môžu komunikovať.

Jeden samostatný procesor so svojou privátnou pamäťou a programom tvorí výpočtový model Random access maschine (ďalej RAM). Aj tento jednoduchší model už má generatívnu silu Turingovho stroja.

2.1 RAM

Skôr ako uvediem definíciu PRAMu, popíšem jeho základnú jednotku RAM. Tento model môžeme popísať viacerými spôsobmi, z ktorých mnohé sú ekvivalentné výpočtovou silou aj zložitosťou.

Definícia 2.1. *RAM (Random Access Maschine) je výpočtový model pozostávajúci z výpočtovej jednotky s pevne daným programom, jednej vstupnej a jednej výstupnej pásky a neobmedzeného počtu registrov R_0, R_1, R_2, ..., pričom v jednom registri môžeme uchovávať ľubovoľne veľké celé číslo. Program výpočtovej jednotky je postupnosť jednoduchých inštrukcií, ktoré sú uvedené v tabuľke. Výpočet začína prvou inštrukciou a končí inštrukciou HALT.*

Inštrukcia	**Popis**
READ	Prečítaj symbol zo vstupu a zapíš ho do R_0
WRITE	Obsah registra R_0 zapíš na výstup

Inštrukcia	**Popis**
STORE R_i	Obsah R_0 zapíš do R_i
COPY R_i	Obsah R_i zapíš do R_0
CONST *c*	Do R_0 zapíš hodnotu *c*
ADD R_i	$R_0 \leftarrow [R_i] + [R_0]$
SUB R_i	$R_0 \leftarrow [R_i] - [R_0]$
MULT *c*	$R_0 \leftarrow [R_i] * c$
DIV *c*	$R_0 \leftarrow [R_i] / c$
IF ZERO *i*	Ak je obsah registra R_0 nulový, pokračuj inštrukciou *i*
GOTO *i*	Pokračuj inštrukciou *i*
HALT	Ukonči výpočet

Tabuľka 2.1: Základné inštrukcie RAMu

Všimnime si, že na zostavovanie PRAMu nám uvedená definícia nevyhovuje, lebo nebudeme mať k dispozícii vstupnú a výstupnú pásku. Preto túto definíciu pozmeníme rešpektujúc súčasnú výpočtovú silu a zložitosť modelu.

Vstup teda nebudeme zadávať na páske ale v špeciálnych registroch, pričom v jednom bude uložená veľkosť vstupu *n* a v ďalších *n* registroch budú postupne uložené jednotlivé bity vstupného slova. Podobne budeme ukladať výstup do špeciálnych registrov a nie na pásku.

2.2 PRAM

Definícia 2.2. *PRAM (Parallel Random Access Maschine) je výpočtový model pozostávajúci z neobmedzeného počtu RAM procesorov označených P_0, P_1, P_2,... a neobmedzeného počtu spoločných (zdielaných) registrov C_0, C_1, C_2,... Každý procesor P_i má svoje identifikačné číslo (index), má vlasatú pamäť t.j. neobmedzenú sadu registrov $R_{i,0}$, $R_{i,1}$, $R_{i,2}$,... a inštrukcie na priamy alebo nepriamy prístup do spoločnej pamäte. Základné inštrukcie sú popísané v tabuľke. Procesory sú zosynchronizované podľa globálnych hodín, teda inštrukcie vykonávajú v taktoch. Vstup je po bitoch zadaný v registroch C_1, C_2,..., C_n, kde n je počet bitov vstupu. Samotné n je uložené v registri C_0. Výstup je uložený v rovnakej podobe.*

Inštrukcia	**Popis**
$R_i \leftarrow [R_j]$	Obsah registra R_j skopíruj do registra R_i
IDENT	Do registra R_0 zapíš index procesora
CONST *c*	$R_0 \leftarrow c$
ADD R_i	$R_0 \leftarrow [R_i] + [R_0]$
SUB R_i	$R_0 \leftarrow [R_i] - [R_0]$
MULT *c*	$R_0 \leftarrow [R_i] * c$
DIV *c*	$R_0 \leftarrow [R_i] / c$
IF ZERO *i*	Ak R_0 obsahuje 0, pokračuj inštrukciou *i*
GOTO *i*	Pokračuj inštrukciou *i*

Inštrukcia	**Popis**
HALT	Ukonči výpočet

Tabuľka 2.2: Základné inštrukcie PRAMu

Tento výpočtový model má dva výrazné nedostatky. Prvým je predpoklad neobmedzeného počtu procesorov, podieľajúcich sa na jednom výpočte. Chceme sa pokúsiť obmedziť ich. Každý výpočet potrebuje isté množstvo procesorov závislé od dĺžky vstupu. Preto spomedzi procesorov vyberieme akéhosi vodcu P_0, ktorý zo vstupu vypočíta počet potrebných procesov. Toto číslo následne uloží do špeciálneho registra C_{-1}. Na začiatku bude aktívny len procesor P_0, ostatné budú čakať, kým P_0 nezapíše hodnotu maximálneho indexu procesora, a následne sa aktivujú všetky procesy s indexom menším ako je obsah registra C_{-1}.

Druhou možnosťou, ako riešiť tento problém, je na začiatku aktivovať P_0 a povoliť inštrukciu FORK, pomocou ktorej môže procesor aktivovať ľubovolný iný procesor.

Druhým nedostatkom zadefinovaného modelu sú konflikty pri viacnásobnom prístupe do spoločnej pamäte pri čítaní alebo zapisovaní. Najprv si každú inštrukciu rozdelíme na tri fázy:

1) prístup do spoločnej pamäte za účelom čítania z nej (ak je to potrebné)
2) vykonanie výpočtu potrebného pre danú inštrukciu
3) prístup do spoločnej pamäte za účelom zápisu (ak je to potrebné).

Teraz sme odstránili konflikty medzi čítaním a zápisom. Avšak stále sa môže viacero procesov naraz pokúsiť o zápis (alebo len o čítanie). Podľa toho ako vyriešime takéto situácie, delíme PRAMi na viacero typov:

(1) CRCW – PRAM (Concurrent Read Concurrent Write): Dovolíme procesorom súčasné čítanie aj súčasný zápis do rovnakého registra. Spoločné čítanie nevytvára žiadny vedľajší efekt ale pri spoločnom zápise sa môžu vyskytnúť problémy. Tieto implikujú tri módy tohoto typu:
   (a) PRIORITY: Ak chce vykonať zápis do jedného registra viac procesorov, zápis sa podarí len procesoru s najmenším indexom.
   (b) COMMON: Ak chce do jedného registra zapisovať viacero procesorov, zápis sa podarí len vtedy, ak zapisujú rovnakú hodnotu. V opačnom prípade sa výpočet zasekne.
   (c) ARBITRARY: Náhodne vybraný procesor žiadajúci o zápis do rovnakého registra zapíše svoju hodnotu. Tento model ako jediný využíva nedeterministický prístup.
(2) CREW – PRAM (Concurrent Read Exlusive Write): Povolíme súčasné čítanie procesov, ale zapisovať do jedného registra môže súčasne len jeden procesor. Toto musí byť ošetrené už v programe PRAMu.
(3) EREW – PRAM (Exclusive Read Exclusive Write): Čítať a zapisovať do spoločného registra môže vždy len jeden procesor.

Pram sa prevažne využíva na analýzu paralelných počítačov, presnejšie na analýzu ich časovej a priestorovej zložitosti. Preto spomeniem, že jednotlivé typy PRAMov sa dajú vzájomne simulovať. Zrejmé sú inklúzie:

EREW-PRAM ≤ CREW-PRAM ≤ PRIORITY CRCW-PRAM
PRIORITY CRCW-PRAM ≤ COMMON CRCW-PRAM
COMMON CRCW-PRAM ≤ ARBITRARY CRCW-PRAM

Dajú sa však dokázať aj opačné inklúzie, i keď nie vždy s rovnakou zložitosťou. Napríklad pri simulácii CREW-PRAMu pomocou EREW-PRAMU sa nám časová zložitosť zhorší O(logP(n)) krát, kde P(n) je počet procesorov, na ktorých PRAM pracuje.

3 Implementácia Prostredia pre experimentovanie s PRAMami

Teraz už poznáme presnú definíciu PRAMu a môžeme sa pokúsiť naprogramovať naše prostredie. Sformulujme teda možné požiadavky užívateľa na toto prostredie.

Prostredie by malo umožňovať vytváranie, teda písanie a editovanie, programu, nakonfigurovanie konkrétneho PRAMu, na ktorom má program bežať a spustenie výpočtu na nejakom vstupe. Prirodzene sa nám žiada rozdeliť celý program prostredia na dve časti.

1. Grafické užívateľské rozhranie
2. Parallel Random Access Maschine

V nasledujúcich kapitolách si postupne priblížime obe tieto časti.

Prostredie bolo programované v jazyku JAVA JRE1.5.0 _06, v Eclipse SDK verzia 3.1.1. Ďalej na programovanie grafického rozhrania som využila balíčky java.awt.*, java.io.* a javax.swing.*.

3.1 PRAM

Vzhľadom na to, že užívateľské rozhranie bude vytvárať a manipulovať s PRAMami, bude výhodnejšie, ak si ako prvú časť popíšeme túto.

Pozrime na model PRAMu z definície 2.2., a všimneme si, že tento model nie je ideálny na priamu implementáciu. Pozmeníme si ho tak, aby vyhovoval našim požiadavkám.

V prvom rade výpočet procesoru P_0, pred spustením ostatných procesorov môže byť zdĺhavý a zložitý na naprogramovanie. Preto ešte pred spustením výpočtu si užívateľ nastaví, na koľkých procesoroch sa má výpočet spustiť. Preto nie je nutné programovať prvú časť výpočtu P_0. Ak si užívateľ nebude istý presným číslom, môže túto časť vypočítať na všetkých spustených procesoroch a následne ukončiť výpočet všetkých procesorov, ktoré nepotrebujeme.

Vstup použitý na jeden program, by sme mohli chcieť použiť viac krát, preto musí existovať možnosť uložiť ho. Vstup teda bude uložený v súbore. Pri spustení PRAMu sa načíta z daného súboru. Formát vstupu chceme zachovať, preto v prvom riadku bude číslo *n,* označujúce veľkosť vstupu a na ďalších *n* riadkoch budú zadané jednotlivé vstupné čísla. Na urýchlenie výpočtu tieto čísla nemusia byť bity, ale naďalej budeme vyžadovať aby to boli celé čísla.

Kvôli pôvodnému tvaru výstupu, by bolo nutné dávať pozor pri programovaní PRAMu na formu dát uložených v spoločnej pamäti a ich následná konverzia na požadovaný tvar je tiež obtiažna. Preto veľkosť výstupu *n* bude zadaná užívateľom. Výstup sa bude tiež zapisovať do súboru. Ten teda bude obsahovať *n* riadkov a každý z nich môže obsahovať celé číslo.

Zadávať program pre každý procesor zvlášť by bolo zdĺhavé, preto budeme vyžadovať jeden program, ktorý bude spoločný pre všetky procesory. Toto môžeme vyžadovať bez výraznej zmeny časovej zložitosti, či sily modelu vďaka tomu, že na základe inštrukcie podmienky a inštrukcie IDENT môže každý procesor vykonávať rôzny program.

Teraz sa bližšie pozrime na štruktúru PRAMu. Všimnime si, že jeho najmenšia jednotka je sada registrov. Každý procesor má jednu vlastnú sadu a navyše majú jednu spoločnú sadu. Tieto sady som mohla reprezentovať pomocou poľa čísiel. Je však výhodné aby manipulácia so spoločnými registrami a súkromnými registrami bola rovnaká, alebo sa len málo líšila. Pritom pre jednotlivé typy PRAMu treba kontrolovať prístup k spoločnej sade registrov.

3.1.1 Registre

Preto bolo výhodné zakomponovať túto kontrolu už do registrov. Navrhla som triedu Registre, ktorá vie plniť základné požiadavky na sadu registrov. Navyše si môže pamätať ku každému pamäťovému miestu popisku. Tá sa môže meniť počas výpočtu PRAMu.

Do registrov budeme ukladať len čísla triedy int, kvôli jednoduchšej implementácii. Tieto čísla sa mi zdajú postačujúce pre základnú manipuláciu s PRAMami a učenie sa pracovať s nimi. Trieda Registre teda obsahuje chránené pole integerov R, kde je uložený obsah registrov, chránené pole Stringov Pozn, s popiskami pre jednotlivé registre a jeden integer veľkosť, ktorý označuje momentálnu veľkosť spomínaných polí.

Použité metódy v triede Registre:

- Registre(int max): konštruktor triedy
- public int[] getReg(): vracia obsah sady registrov
- public String[] getPozn(): vracia popisky, komentáre k registrom
- public String getPozn(int i): vracia popisku k registru i
- public int getVelkost(): vracia počet použitých registrov
- protected boolean setVelkost(int i): zmení počet používaných registrov, pričom patrične upraví používané polia, a ak sa to podarí vráti true. Je volaná vždy, keď je to potrebné ostatnými metódami
- public int getR(int i): vráti obsah registra i
- public int getR0(): vráti obsah registra 0

- public boolean setR(int i,int value): nastaví obsah registra i na hodnotu value
- public void setR(int value): nastaví obsah registra 0 na hodnotu value
- public void setPozn(int i, String value): uloží popisku k registru i

Táto trieda registrov vyhovuje požiadavkám na obyčajnú (nezdieľanú) sadu používanú jedným procesorom. Vybudujeme novú triedu ZdielanéRegistre, ktorá zdedí vlastnosti predchádzajúcej triedy a navyše bude kontrolovať prístup k jednotlivým registrom počas každého kroku. V každom kroku musí prijať rôzne požiadavky na zmenu od rôznych procesorov a následne ich vyhodnotiť. Bude potrebovať nové chránené polia: pole integerov novéR, kam budeme ukladať nové obsahy registrov, pole booleanov write na kontrolu zápisu do registrov a premennú padol, ktorá sa nastaví na hodnotu false, ak procesory porušia pravidlá pre prístup k spoločným registrom.

Trieda ZdielanéRegistre má novú dôležitú metódu public void novyKrok(), ktorá z nových hodnôt v Registroch urobí staré, vynuluje kontrolu a pripraví nové pole pre nové hodnoty v registroch (tieto sa na začiatku rovnajú hodnotám v starých registroch. Poslednou zmenou je public boolean padol(), ktorá vracia hodnotu premennej padol.

Vzhľadom na to, že požiadavky na prístup sa líšia, pre každý typ PRAMu, nevieme prepísať staré inštrukcie na zápis do registrov a čítanie z nich. Preto vytvoríme novú triedu pre každý typ PRAMu EREWRegistre, CREWRegistre, PriorityCRCWRegistre a CommonCRCWRegistre. Tieto triedy implementujú konkrétne algoritmy pre kontrolu prístupu k registrom.

Vzájomne sa líšia, napríklad trieda EREWRegistre potrebuje ďalšie pole booleanov read, ktoré kontroluje prístup k registrom pri čítaní.

3.1.2 RAM

Pozrime sa teraz na jednotlivé procesory. Majú sadu registrov, vedia vykonávať jednoduché inštrukcie, majú vlastné identifikačné číslo, môžu vykonávať program, alebo stáť, vedia vrátiť počet použitých registrov a ich obsah registrov.

Nazveme si túto triedu RAM. Sada registrov bude samozrejme inštancia popísanej triedy Registre. Ďalšie premenné sú ident, kde je uložené identifikačné číslo procesora (nastavené konštruktorom), cKroku s nasledujúcou inštrukciou, ktorú má procesor vykonať, premenné stav a padol, ktoré popisujú činnosť procesora. Okrem vracania stavových hodnôt, je verejná len metóda volania Krok(int prikaz[], ZdielaneRegistre C, LinkedList pozn), ktorá vykoná inštrukciu popísanú v príkaze, ako spoločnú pamäť použije C a popisky k registrom bude čerpať z listu pozn. Vďaka tomu, že C kontroluje prístup k registrom, sa kód pre jednotlivé typy PRAMu nelíši. Preto môžeme použiť spoločnú triedu pre všetky typy.

Ak by sme prístup k registrom nechali kontrolovať RAMom museli by sme mať pre každý typ PRAMu jeden typ RAMu. Navyše kontrola podmienok je neprehľadná a opakuje sa pre viacero inštrukcií. Použitie triedy ZdielanéRegistre je pre nás výhodnejšie.

3.1.3 PRAM

Nakoniec popíšeme abstraktnú triedu PRAM. Trieda PRAM má v prvom rade program, čo je sada inštrukcií v číselnom kóde. Jednotlivé inštrukcie sú uložené v LinkedListe, kvôli jednoduchšej manipulácii. Aby užívateľ nemusel písať program v tomto číselnom kóde, trieda PRAM zabezpečuje preklad pomocou metódy compile(File, kod, File ciselnyKod) zo súboru s bežným kódom do súboru s číselným kódom.

Okrem toho obsahuje trieda PRAM zoznam procesorov (jednotlivých RAMov) a ich počet, zoznam poznámok (popisiek) k jednotlivým inštrukciám, čo umožňuje aby inštrukcie pozostávali len z čísiel. Nesmieme zabudnúť ani na sadu zdieľaných registrov, súbor so vstupom a výstupom, veľkosť výstupu a stavové premenné. Navyše ak PRAM padne, ukladáme číslo jedného z procesorov, ktoré padli, a číslo inštrukcie , ktorú práve vykonával.

Metódy tejto triedy môžeme rozdeliť do niekoľkých kategórií:

- konštruktor: okrem nastavenia stavových hodnôt pre PRAM (napr. stav PRAMu, vytvorenie RAMov, nastavenie ich počtu, ..) preloží program do nového súboru a následne ho prečíta ako číselný kód a tým vybuduje aj zoznam popisov k jednotlivým inštrukciám. Na druhej strane nevieme vytvoriť sadu registrov (tá je závislá od typu PRAMu).
- metódy vracajúce stav PRAMu: vracajú veľkosť výstupu, počet procesorov, veľkosť spoločnej sady registrov, veľkosť najväčšej použitej sady registrov, obsah registrov, popisky k registrom , či PRAM beží, či padol a v neposlednom rade aj typ PRAMu (avšak táto metóda je ako jediná abstraktná). Tieto metódy používame počas krokovania programu.
- metódy pre kompiláciu: Tu si môžeme vybrať z dvoch možností. Metóda compile(File f1, File f2) preloží kód zo súboru f1 do súboru f2, Na druhej strane metóda compile(File f1) postupne kontroluje súbor f1 a vráti všetky nájdené chyby. Postup pri kompilácií je popísaný v kapitole 3.1.4.
- metódy vykonávané v rámci behu programu: V prvom rade treba spomenúť metódy volané užívateľom ako je Run(), Step() a Dokonci(), ale tieto využívajú aj metódy init() na načítanie vstupu a nastavenie počiatočného stavu, koniec() na vypísanie výstupu a krok() na vykonanie jedného taktu PRAMu.

Teraz môžeme vytvoriť triedu pre každý typ PRAMu: EREWPram, CREWPram, PriorityCRCWPram a CommonCRCWPram. Tieto triedy sa líšia len v konštruktore, každý si vytvorí sadu registrov ako inštanciu svojho typu registrov, a v metóde, ktorá vracia typ PRAMu.

Pozrime sa podrobnejšie na priebeh jedného kroku PRAMu. Postupne zoberiem každý procesor, ktorý ešte neskončil, vykonám na ňom jeden krok a skontrolujem či nepadol. Následne ukončím krok pre spoločnú sadu registrov. Ak niektorý procesor padne, padne aj celý PRAM. PRAM ukončí výpočet, ak skončili všetky procesory.

3.1.4 Kompilácia a inštrukcie pre PRAM

Celý preklad programu spočíva v nasledovných krokoch:

1. vytvor zoznam návestí
2. zober nový riadok
3. ak kedykoľvek nájdeš / choď na bod 6
4. zober z neho inštrukciu a prelož ju, ak tam je
5. ak to bolo priradenie a má popisku vyber ju a zapamätaj si ju
6. ak niečo nebolo správne, spracuj chybu
7. pokračuj bodom 3, kým nenájdeš koniec súboru

Sadu inštrukcií mierne pozmeníme:

Inštrukcia	**Popis**
R[i]=R[j]	Obsah registra R_j skopíruj do registra R_i
C[i]=R[j]	Zápis do spoločnej pamäte
C[R[i]]=R[j]	Nepriamy zápis do pamäte
R[i]=C[j]	Čítanie zo spoločnej pamäte
R[i]=C[R[j]]	Čítanie zo spoločnej pamäte s nepriamym prístupom
R[i]=R[R[j]]	Ďalšie inštrukcie pre prácu s registrami
C[i]=R[R[j]]	
R[R[j]]=R[j]	
R[R[j]]=C[j]	
R[R[j]]=R[R[j]]	
R[R[j]]=C[R[j]]	
C[R[i]]= R[R[j]]	
IDENT	Do registra R_0 zapíš index procesora
CONST *c*	$R_0 \leftarrow c$
ADD *i*	$R_0 \leftarrow [R_i] + [R_0]$
SUB *i*	$R_0 \leftarrow [R_i] - [R_0]$
MULT *c*	$R_0 \leftarrow [R_i] * c$
DIV *c*	$R_0 \leftarrow [R_i] / c$
IFZERO *i*	Ak R_0 obsahuje 0, pokračuj inštrukciou *i*
ISLESS(*i*,*j*) *k*	Ak platí $R_i < R_j$, pokračuj inštrukciou *k*
GOTO *i*	Pokračuj inštrukciou *i*
HALT	Ukonči výpočet

Tabuľka 3.1: Inštrukcie PRAMu

Pridali sme novú inštrukciu „ISLESS(*i*,*j*) *k*“, ktorá uľahčuje niektoré programátorské konštrukcie. Všetky inštrukcie majú tvar, ktorý užívateľ nesmie zmeniť. Napríklad príkaz „ISLESS(*i*,*j*) *k*“ vyžaduje aby neobsahoval žiadne medzery s výnimkou medzier pred číslom *k*. Vypísali sme tiež kompletnú sadu inštrukcií priradenia.

Tieto inštrukcie prekladáme do číselných inštrukcií podľa tabuľky 3.2.

Inštrukcia	**Preklad**
NOP	0
HALT	1
IDENT	2
CONST *c*	3 *c*
ADD *i*	4 *i*
SUB *i*	5 *i*
MULT *c*	6 *c*
DIV *c*	7 *c*
GOTO *i*	8 *i*
IFZERO *i*	9 *i*
ISLESS(*i*,*j*) *k*	10 *i j k*
Priradenie *parametre*	11 *parametre*

Tabuľka 3.2: Preklad inštrukcií PRAMu

Preklad priradenia je trochu zložitejší. Preto bude potrebovať väčší počet parametrov. Pravá aj ľavá strana priradenia majú v podstate rovnaký tvar. Máme štyri možnosti: R[i], C[i] , R[R[i]], C[R[i]]. Označme ich postupne číslami 0, 1, 2, 3. Tým získavame prvý a druhý parameter. Niektoré dvojice parametrov nie sú povolené, napríklad dvojica 1 1. Tretí (resp. štvrtý) parameter získame ako číslo vnútri pravej (resp. ľavej) strany. Teda priradenie „R[i]=C[R[j]]“ preložíme ako „11 0 3 i j“.

Za každým priradením môže byť popiska. Tá je uzavretá do zložených zátvoriek. Celý text vnútri bude uložený ako popiska k inštrukcii a neskôr bude pridaná do pomocných štruktúr k registrom, odkiaľ ju môžeme získať späť.

Poslednou pomôckou sú návestia. Na začiatku každého riadku môžeme pomenovať návestie. Tento názov musí byť v tvare „<názov_návestia>“. Pri prekladaní súboru si najprv vytvoríme zoznam návestí (aj s číslami riadkov v ktorých sú). Pri druhom prechode programom prekladáme inštrukcie a súčasne, ak nájdeme skokovú inštrukciu, ktorá má miesto čísla názov návestia tvaru „názov_návestia“, nahradíme ho číslom riadku zo zoznamu.

3.2 Grafické užívateľské rozhranie

Grafické užívateľské rozhranie by malo umožňovať užívateľovi ľahkú manipuláciu s PRAMami bez nutnosti dlho študovať formu požiadaviek, príkazy,... Zhrňme si naše očakávania a požiadavky:

- písanie, editovanie a tlačenie jednoduchých programov PRAMu
- zadať vstup, prípadne ho pozmeniť
- zadať veľkosť výstupu a súbor, kam bude uložený
- určiť počet procesorov
- zvoliť si typ PRAMu, s ktorým chcem pracovať
- zobraziť výstup skončeného výpočtu

- medzi dvomi krokmi zobrazovať stav spoločnej pamäte, pamäte ktoréhokoľvek procesoru a číslo inštrukcie, ktorú daný procesor vykonáva

Vytvoríme triedu GUI, ktorá bude zobrazovať a obsluhovať základné okno. Toto okno zdedí vlastnosti triedy JFrame z balíčku javax.swing. To nám zjednoduší implementáciu. Stačí ho pomenovať, vytvoriť menu a obsah okna. Menu bude obsahovať položky pre otváranie, ukladanie a tlačenie programu, Položky na nastavenie typu PRAMu a počtu jeho procesorov a príkazy Na kompiláciu a spustenie programu.

Obsah okna bude pozostávať z nasledovných častí: program, vlastnosti PRAMu, povolené príkazy a chyby po poslednej kompilácii. Z dôvodu jednoduchšej implementácie a nezávislosti som navrhla triedy PProgram a POptions. Obe triedy sú vlastne panely obsahujúce ďalšie grafické komponenty. Na príkazy a chyby nám stačia dve textové polia (nazveme ich príkazy a chyby), využijeme triedu JTextArea z balíčka javax.swing, a oba komponenty zobrazíme pomocou záložiek (tabbedPane) na to isté miesto. Samotné záložky nemusíme programovať, využijeme existujúcu triedu JTabbedPane (jawax.swing).

Okrem týchto grafických komponentov si musíme pamätať aj vlastnosti PRAMu (fileProg, kde je súbor s programom, mod s typom PRAMu a pocProc, ktorá označuje počet procesorov, ktoré majú byť spustené) a PRAM samotný (ako pram).

Navyše musíme vedieť obslúžiť rôzne udalosti pre toto okno. Na toto využijeme interface ActionListener z balíčka java.awt.event a jedinej metódy public void actionPerformed(ActionEvent e), ktorá vie obslúžiť jednotlivé udalosti.

Všetky udalosti sa spúšťajú pomocou menu. Sú to Open, Save, Save as, Print, Exit, Step, Run, Compile, zmena počtu procesorov a typu PRAMu. Udalosť Exit jednoducho ukončí beh aplikácie. Open, Save, Save as a Print obsluhujú program. Po ich vyvolaní sa zobrazí prislúchajúce dialógové okno a následne zavolajú patričnú metódu z fProg. Zmena počtu procesorov sa tiež odohrá cez štandardné dialógové okná, pričom pozmeníme nastavenie fOpt a vlastnosti pre PRAM uložené v jednotlivých premenných.

Zostávajú nám udalosti Step, Run a Compile, ktoré súvisia s behom programu. Metóda obsluhujúca udalosť Compile len zavolá Pram.compile(fileProg) a nájdené chyby vypíše do JTextArea chyby. Run a Step pripravia PRAM na vykonávanie. Pomocou dialógových okien (JDialog a JOptionPane) získame od užívateľa súbor so vstupom a výstupom. Súbor so vstupom môže užívateľ priamo otvoriť, alebo vytvoriť nový a uložiť ho. Potom si vypýtame veľkosť výstupu, ak je rovná nule súbor pre výstup už nepotrebujeme. Inak užívateľ zadá aj súbor, kam PRAM uloží výstup.

Run ďalej spustí výpočet PRAMu a po jeho skončení zobrazí výstup. Príkaz Step naopak vykoná len jeden takt PRAMu a medzistav prehľadne zobrazí v novom okne, ktoré je inštanciou triedy PStep.

Trieda PProgram obsahuje len popisku s názvom programu a textový editor. Pretože nepotrebujeme zobrazovať štylizovaný text, využijeme triedu JTextArea. S touto triedou sa najľahšie manipuluje. Môžeme si všimnúť, že jeho riadkovú štruktúru využijeme pri označovaní riadkov. Trieda PProgram vie načítať program zo súboru, uložiť ho a vytlačiť ho. Navyše vďaka triede JTextArea vie užívateľ

štandardne manipulovať s textom programu. Pre neskoršie využitie v triede PStep vieme označiť ľubovolný riadok. JTextArea neumožňuje zmenu štýlu písma pre jeden úsek, preto ho označíme pridaním sekvencie znakov „> “ na začiatok riadku. Označením iného riadku sa starý automaticky odznačí.

Trieda POptions zobrazuje počet procesorov a typ PRAMu. Obsahuje len dve textové polia s popiskami a metódy pre ich nastavenie.

3.2.1 Trieda PStep

Posledná spomenutá trieda je PStep. Je to tiež JFrame, ktorý slúži na zobrazenie stavu PRAMu medzi jednotlivými krokmi. Využíva ďalšie triedy: už spomenutú PProgram a nové triedy POptions2, PInstrukcie, PPamat a PColumns. Všetky triedy sú vlastne panely, ktoré obsahujú ďalšie komponenty. Trieda PStep tieto triedy spája a zobrazuje do jedného okna, pričom obsluhuje všetky udalosti v nich vyvolané, na čo potrebuje aj jeden PRAM.

POptions2 zobrazuje vlastnosti PRAMu ako je jeho typ, veľkosť výstupu, veľkosť doteraz použitej pamäte a počet procesorov. Navyše tu môžeme zvoliť jeden procesor (nazveme ho zvolený Procesor), pričom vidíme číslo jeho nasledujúcej inštrukcie. V neposlednom rade sú tu tlačítka Step, Run, Stop, Step X a Print, ktoré vyvolávajú patričné udalosti v triede PStep. Poslednú udalosť vyvoláme nastavením nového procesora. Trieda POptions2 má len metódy na nastavenie jednotlivých údajov a ich opätovné získanie.

PProgram slúži na zobrazenie programu a počas krokov označuje inštrukciu, ktorú má práve vykovať procesor zvolený Procesor.

Trieda PColumns slúži na editovanie tabuľky v triede PPamat. Môžeme si tu zvoliť počet zobrazených sád registrov, zvoliť si, ktoré sady to budú, a ku každej sade určiť, či sa zobrazia aj jej popisky alebo nie. Na panele sa nachádza textové pole, ktoré umožňuje určiť počet zobrazovaných sád registrov, tabuľka a tlačítko Apply. Stlačenie Apply a zmena textového poľa vyvolávajú udalosti. Tabuľka je triedy JTable z balíčku jawax.swing. Jej editovaním užívateľ popíše čísla sád registrov a popisov, ktoré sa majú zobraziť. Táto trieda má len niekoľko metód, ktoré vracajú obsah textového poľa, postupnosť čísiel zvolených sád registrov (zvolenéSady) a pole booleanov zobraziťText, popisujúce, ku ktorým sadám sa zobrazia aj popisky, ktoré získame z tabuľky. Ak sa medzi zvolenými sadami nachádza sada procesora i, i je uložené do poľa,ak je to spoločná pamäť C, predávame ju ako číslo rovné počtu procesorov. Posledná metóda nastaví počet riadkov tabuľky.

PPamat obsahuje len tabuľku zobrazujúcu momentálny stav zvolených sád registrov, prípadne aj ich popisiek. PPamat má implementované metódy pre aktualizáciu tabuľky, na zmenu zobrazovanej množiny sád registrov a ich popisiek a na vytlačenie aktuálnej tabuľky. Posledná trieda PInstrukcie obsahuje tiež len jednu tabuľku, ktorá zobrazuje ku každému procesoru číslo nasledujúcej inštrukcie. Využíva len metódu pre jej aktualizáciu. Obe tieto tabuľky nie sú editovateľné.

Panely spomenutých tried PPamat, PInstrukcie a PColumns zobrazíme v okne PStepu pomocou záložiek.

Udalosť Step, vyvolá metódu step(). Tá vykonaná jeden krok PRAMu, následne skontroluje, či PRAM ešte neskončil výpočet, alebo či nepadol a následne zavolá metódu update(). Udalosť Run nechá program dobehnúť až do konca, zatvorí

okno PStepu a zobrazí výstup. Stop výpočet definitívne zastaví a zatvorí okno PStepu. Step X si vypýta číslo x, vykoná x krokov PRAMu a zavolá update(). Metóda update() aktualizuje tabuľky v triedach PPamat a Pinstrukcia, aktualizuje číslo kroku zvoleného procesoru v POptions2 a označí patričný riadok v programe zavolaním aktualizačných metód v jednotlivých triedach.

Zostáva nám obslúžiť posledné tri udalosti. Udalosť Print zavolá metódu triedy PPamat, ktorá vytlačí aktuálnu tabuľku. Udalosť vyvolaná stlačením tlačidla Apply na panele PColumns, bude tiež obslúžená triedou PStep. PStep si vypýta od PColumns počet sád registrov, zvolenéSady a zobraziťText, a s týmito údajmi zavolá metódu triedy PPamat, ktorá patrične pozmení svoju štruktúru. Posledný typ udalosti vyvolá zmena textového poľa na panele PColumns. PStep si zoberie nové číslo z tohto poľa a zavolá metódu triedy PColumns, ktorá zmení veľkosť tabuľky na tom istom panele. Táto udalosť by mohla byť obsluhovaná v triede PColumns, pretože pracuje len na panele PColumns.

3.3 Použité triedy z Java API

Počas programovania Prostredia pre experimentovanie s PRAMami som využívala rôzne balíky z Java API.

Na programovanie grafického užívateľského rozhrania (GUI) som najviac využila balík javax.swing. Je výhodné použiť niektoré predprogramované štruktúry tohoto balíka, pretože podporuje tvorbu GUI. Taktiež som využila štruktúry pre obsluhu udalostí v GUI z balíku java.awt. Podobne môžeme využiť triedy balíku java.io na čítanie a zapisovanie do súboru.

4 Používanie prostredia pre experimentovanie s PRAMami

Už v predchádzajúcej kapitole sme si popísali aké požiadavky môže mať užívateľ na tento systém. Môže chcieť editovať program, pamäť, nastaviť parametre PRAMu, alebo spustiť PRAM. V tejto kapitole sú priložené niektoré obrázky okien, ktoré užívateľ môže vidieť počas práce s prostredím. Ďalšie sú priložené v prílohe II.

Predveďme si prácu prostredia na konkrétnom príklade.

Príklad 4. *Chceme vypočítať maximum z postupnosti (n-1) zadaných čísiel* $x_1, x_2, ..., x_{n-1}$ *na modeli COMMON CRCW PRAM.*

Vstup bude zadaný nasledovne C[0]←n, C[1]← x_1,…, C[n]← x_n. Chceme výstup C[0]=max{ $x_1, x_2, ..., x_n$ }.

4.1 Algoritmus a program

Na riešenie príkladu nám stačí $(n-1)^2$ procesorov a $2*(n-1)$ pamäte (pri malom *n*, je potrebné aspoň rádovo 10 registrov. Algoritmus je nasledovný:

1. Najprv z indexu procesora získame čísla i, j také, že bude platiť vzťah id = j*(n=1).+ i, a i<(n-1).
2. Procesory s id<(n-1) vykoná: C[n+i] ←0.
3. Každý procesor vykoná inštrukciu: if C[0] < C[0] then C[n+i] ←1.
4. Procesory spĺňajúce rovnosť j=(n-1) vykonajú: if C[n+i]=0 then C[0] ← C[i+1].

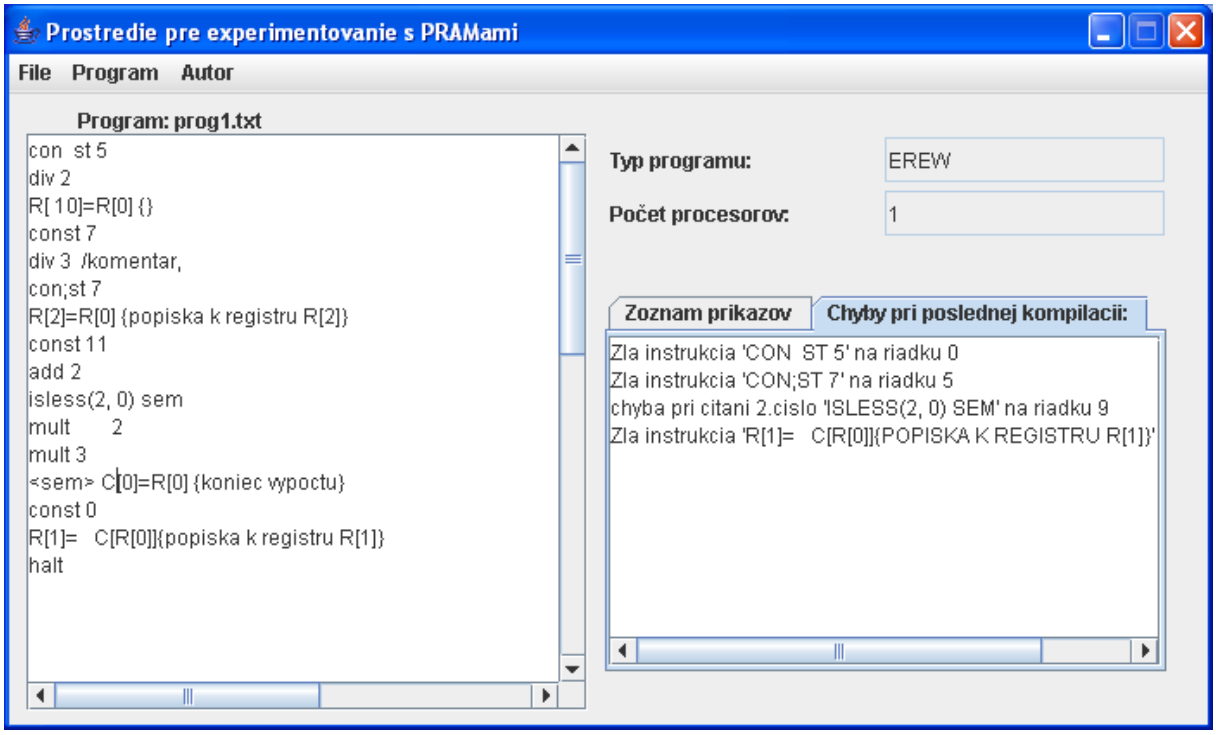
Ľahko vidieť, že takýto algoritmus dáva požadovaný výsledok, avšak musíme zabezpečiť aby sa kroky vykonávali postupne v uvedenom poradí. Môžeme si všimnúť inštrukcie tohoto algoritmu. Nie sú to inštrukcie programu pre model PRAM ale ľahko ich môžeme simulovať v konečnom počte krokov. Treba si uvedomiť, že pri vykonávaní tretieho kroku nenastane konflikt, lebo ak aj zapisuje viacero procesorov súčasne, zapisujú rovnakú hodnotu.

Tento algoritmus teraz prepíšeme na program do nášho PRAMu. Program je pripojený v prílohe 1. Pri jeho písaní som nedávala pozor, na to aby sa všetky

inštrukcie taktu dva (resp. tri) vykonali pred inštrukciami taktu tri (resp. štyri) a je zrejmé, že prvý takt budú procesory vykonávať rôzne dlho.

V tomto kóde sa budú len ťažko hľadať chyby. Preto si pomôžeme práve naprogramovaným prostredím.

4.2 Editácia programu



Obrázok 4.1: Zobrazenie programu

Po spustení Prostredia pre experimentovanie s PRAMami sa zobrazí okno podobné oknu na obrázku 4.1. V jeho ľavej časti je jednoduchý editor, do ktorého budeme písať program. Pomocou položiek menu File, Open vieme vytvoriť nový program. Obdobne môžeme tento súbor uložiť, alebo ho vytlačiť. Meno súboru, ktorý práve editujeme sa zobrazuje nad textovým editorom.

Program PRAMu sa skladá z inštrukcií popísaných v predchádzajúcej kapitole. Každý riadok programu môže obsahovať ľubovolný počet medzier, návestie, inštrukciu a ľubovolný komentár. Inštrukcia by nemala obsahovať žiadne medzery navyše. Môže byť písaná veľkými aj malými písmenami, prípadne ich ľubovoľnou kombináciou. Komentár píšeme za znak „/“.

V pravej časti okna môžeme vidieť zoznam povolených inštrukcií.

Program môžeme aj skompilovať. Pri kompilácii si overíme platnosť a formát inštrukcii. Všetky chyby sa zobrazia v pravej časti okna.

4.3 Nastavenia PRAMu

Pred spustením PRAMu musíme vedieť viacero informácii. Najprv by sme mali určiť typ PRAMu t.j. spôsob ktorým budeme pristupovať k zdieľanej pamäti. Ďalej by sme mali určiť, koľko procesorov potrebujeme na výpočet. V neposlednej rade je potrebný aj súbor s programom. Počet procesorov a typ PRAMu sa zobrazujú v pravej časti okna a meníme ich pomocou menu cez položku Program.

Pri určovaní typu PRAMu by sme si mali dôkladne premyslieť vlastnosti jednotlivých typov. Síce sú vzájomne simulovateľné, ale môžu nám nielen logaritmicky zhoršiť časovú zložitosť ale aj výrazne skomplikovať kód programu.

Počet procesorov tiež ovplyvňuje rýchlosť výpočtu.

4.4 Spustenie PRAMu

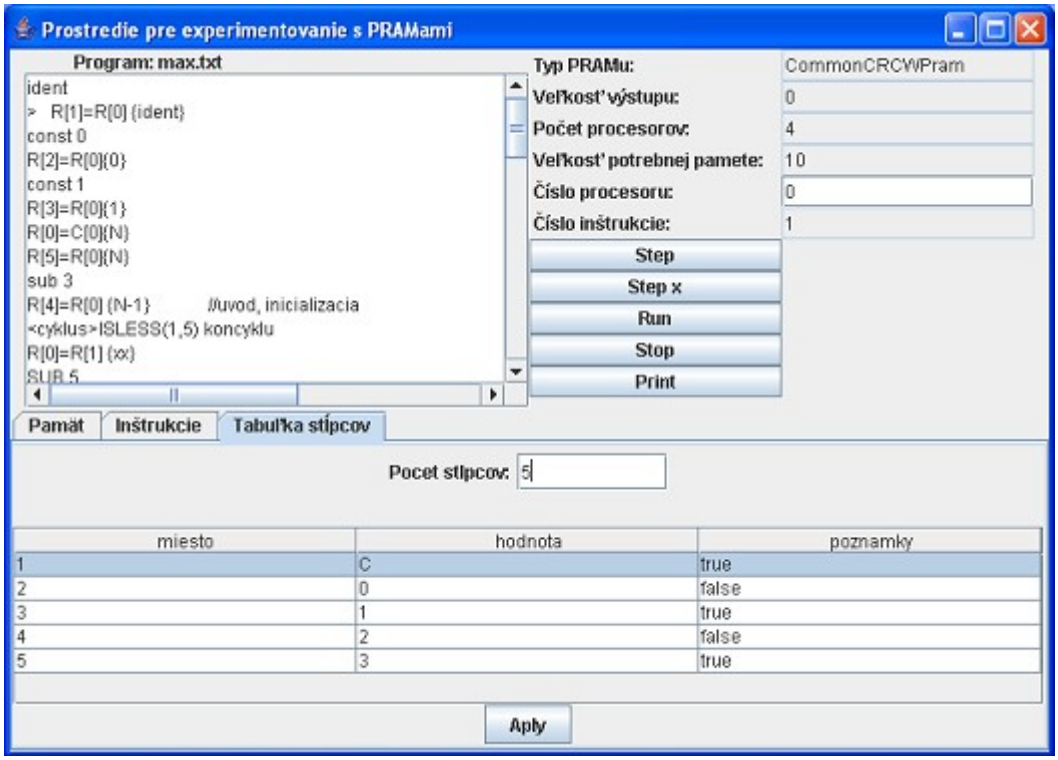
Pred spustením PRAMu by mala byť uložená posledná verzia, inak PRAM bude pracovať s uloženou verziou.

Keď už máme správne nastavené všetky vlastnosti PRAMu, môžeme ho spustiť. Ovládanie je opäť súčasťou menu, avšak sú dve možnosti ako bude PRAM spolupracovať s užívateľom. Zavolaním Run, sa spustí výpočet, ktorý užívateľ nemá možnosť sledovať. Na jeho konci sa v okne zobrazí výstup, ak bol nenulový.

Volaním Step sa vytvorí pomocné okno pre PRAM, ktoré je zobrazené na obrázku 4.2. V tomto okne máme možnosť vykonávať výpočet PRAMu po jednotlivých inštrukciách. Môžeme tu zobraziť program PRAMu, zdieľanú pamäť, pamäť jednotlivých procesorov, číslo nasledujúcej inštrukcie ľubovolného procesora a dôležité vlastnosti PRAMu.

Program sa zobrazuje neustále na ľavej strane okna. Tu ho však nemožno editovať. Na opačnej strane okna sú vypísané niektoré vlastnosti PRAMu: typ, počet procesorov, veľkosť pamäte a veľkosť výstupu. Môžeme si tu vybrať jeden procesor. K nemu sa tu zobrazí číslo nasledovnej inštrukcie. Zároveň sa táto inštrukcia zobrazí v programe. Nakoniec si môžeme vybrať, čo sa zobrazí v dolnej časti okna. Môže to byť pamäť, čísla procesov a ich nasledovná inštrukcia alebo tabuľka na výber stĺpcov.

Na obrázku 4.2 je zobrazená posledná možnosť. Najprv si musíme zvoliť koľko sád registrov chceme zobraziť. Toto číslo zadáme ako počet stĺpcov. Následne vyplníme tabuľku. V každom riadku i zvolíme, ktorú sadu registrov chceme zobrazovať v i-tom stĺpci. Ak je to sada pre j-ty procesor zadáme číslo procesoru ako hodnotu i-tého riadku. Ak chceme zobraziť spoločnú sadu registrov zadáme do tabuľky písmeno C. Ak k tejto sade chceme zobrazovať aj popisky, zadáme do stĺpca poznámky hodnotu true.

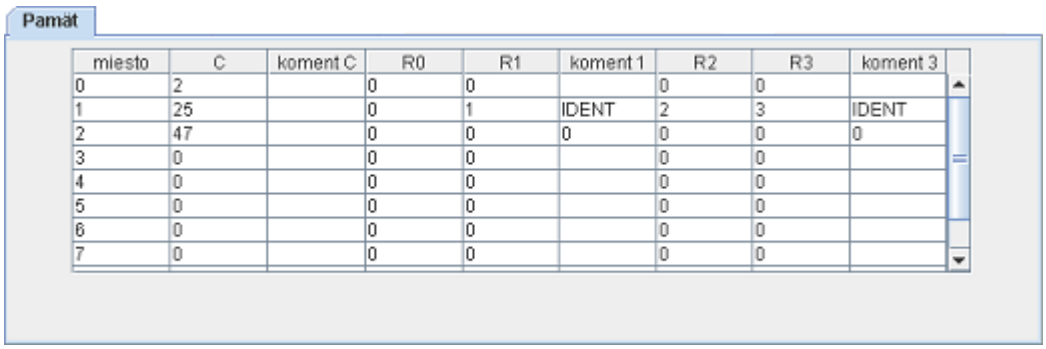


Obrázok 4.2: Zobrazenie medzikroku.

Na obrázku 4.3 je ukážka, ako sa zobrazí obsah registrov v tabuľke. Všimnime si, že jednotlivé stĺpce majú hlavičku podľa toho, čo je v nich uložené. Napríklad v štvrtom stĺpci je obsah sady registrov procesoru s číslom 0. Obsah tejto tabuľky môžeme vytlačiť príkazom Print.

Ďalší krok PRAMu sa vykoná po stlačení tlačítka Step. Keď sa výpočet skončí okno sa automaticky zruší a vypíše sa výstup. Výpočet môžeme urýchliť stlačením tlačidla Run alebo Step x a predčasne ho ukončiť stlačením Stop. Po príkaze Run PRAM dokončí výpočet a po Step x vykoná zadaných x krokov.

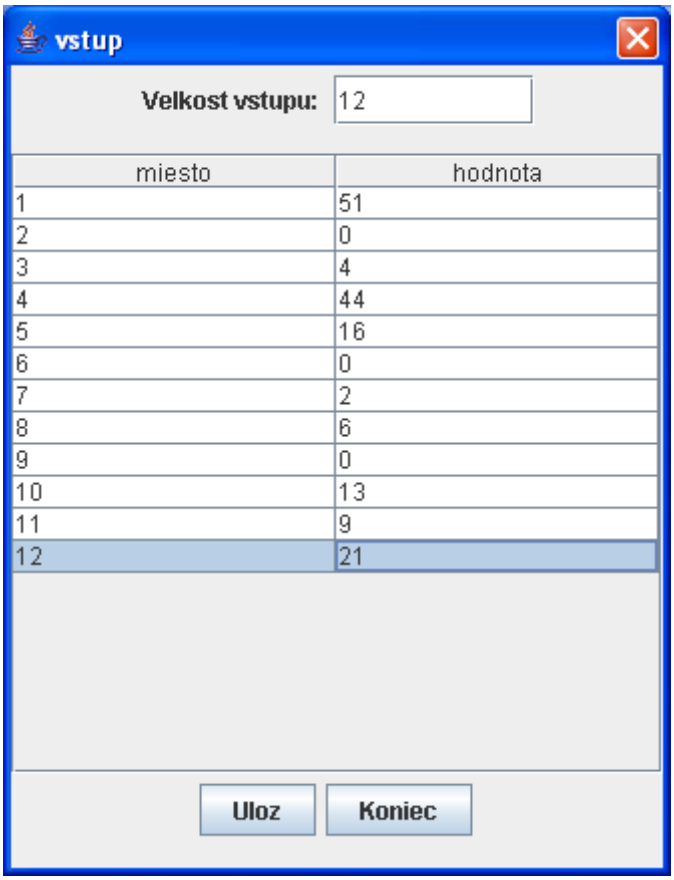
Spustenie PRAMu po krokoch je ideálne na ladenie programu a na učenie sa manipulovať s PRAMami.



Obrázok 4.3: Zobrazenie pamäte.

4.5 Zadávanie vstupného a výstupného súboru

Po spustení PRAMu si aplikácia vypýta vstupný a výstupný súbor. Vstupný súbor môžeme vybrať z už existujúcich súborov, alebo vytvoriť nový a následne ho uložiť. Nový vstup zadávame do tabuľky , ktorá je na obrázku 4.4. Najprv si určíme veľkosť vstupu a následne jednotlivé hodnoty registrov. V prvom stĺpci tabuľky sú čísla registrov a v druhom sú zadané čísla. V každej bunke druhého stĺpca musí byť celé číslo.



Obrázok 4.4: Zobrazenie zadávania vstupu.

Po zadaní vstupu si aplikácia vypýta veľkosť výstupu. Ak je nenulová musíme zadať aj súbor, do ktorého sa výstup zapíše.

Po úspešnom skončení výpočtu PRAMu, sa zobrazí existujúci výstup do podobného okna ako je 4.4. To znamená, že sa nezobrazí, ak PRAM padne alebo veľkosť výstupu je nula.

4.6 Príklad použitia prostredia

Predveďme si prácu prostredia na konkrétnom príklade.

Príklad 4. *Chceme vypočítať maximum z postupnosti (n-1) zadaných čísiel $x_1, x_2, ..., x_{n-1}$ na modeli COMMON CRCW PRAM.*

Vstup bude zadaný nasledovne C[0]←n, C[1]← x_1,…, C[n]← x_n. Chceme výstup C[0]=max{ $x_1, x_2, ..., x_n$ }.

Na riešenie príkladu nám stačí *(n-1)²* procesorov a *2*(n-1)* pamäte (pri malom *n*, je potrebné aspoň rádovo 10 registrov. Algoritmus je nasledovný:

1. Najprv z indexu procesora získame čísla i, j také, že bude platiť vzťah id = j*(n=1).+ i, a i<(n-1).
2. Procesory s id<(n-1) vykoná: C[n+i] ←0.
3. Každý procesor vykoná inštrukciu: if C[0] < C[0] then C[n+i] ←1.
4. Procesory spĺňajúce rovnosť j=(n-1) vykonajú: if C[n+i]=0 then C[0] ← C[i+1].

Ľahko vidieť, že takýto algoritmus dáva požadovaný výsledok, avšak musíme zabezpečiť aby sa kroky vykonávali postupne v uvedenom poradí. Môžeme si všimnúť inštrukcie tohoto algoritmu. Nie sú to inštrukcie programu pre model PRAM ale ľahko ich môžeme simulovať v konečnom počte krokov. Treba si uvedomiť, že pri vykonávaní tretieho kroku nenastane konflikt, lebo ak aj zapisuje viacero procesorov súčasne, zapisujú rovnakú hodnotu.

4.6.1 Písanie programu pre PRAM

Tento algoritmus teraz prepíšeme na program do našeho PRAMu. Program je pripojený v prílohe 1. Pri jeho písaní som nedávala pozor, na to aby sa všetky inštrukcie taktu dva (resp. tri) vykonali pred inštrukciami taktu tri (resp. štyri) a je zrejmé, že prvý takt budú procesory vykonávať rôzne dlho.

Začnime teda programovať. Najprv si musíme uvedomiť, že všetky operácie sčítania, odčítania, ... sa vykonávajú v registry R_0. Pred prvým krokom algoritmu si pripravíme užitočné hodnoty v niektorých registroch. Napríklad R[1]=ident, R[2]=0, R[3]=1, R[4]=n-1 a R[5]=n.

Kód:

1. ident
2. R[1]=R[0] {ID procesoru}
3. const 0
4. R[2]=R[0] {0}
5. const 1
6. R[3]=R[0]
7. R[5]=C[0] {N}
8. R[0]=C[0]
9. sub 3
10. R[4]=R[0] {N-1}

Ďalej potrebujeme cyklus, ktorý bude od R[1] odčítavať N a k R[2] pripočítavať 1 kým R[1] nebude menšie ako N.

Kód:

11. / začínam vyrábať i a j
12. <zaciatok_cyklu> isless(1,5) koniec_cyklu
13. R[0]=R[1]

14. sub 5
15. R[1]=R[0] {buduce j}
16. R[0]=R[2]
17. add 3
18. R[2]=R[0]
19. goto zaciatok_cyklu
20. <koniec_cyklu> nop
21. / i, j sú hotové

Podobne by sme mohli pokračovať až do konca programu. Celý program je pripojený ako príloha. Na záver musíme overiť, koľko ktorá časť programu trvá jednotlivým registrom aby sa nestalo, že jeden procesor vypíše výstup a iný iba porovnáva dve čísla zo vstupu.

5 Záver

Prostredie pre experimentovanie s PRAMami je prijateľný produkt, ktorý nevyžaduje špeciálny hardvér. Vďaka tomu, že bol naprogramovaný v JAVE môže bežať na bežnom počítači pod operačným systémom Windows alebo Linux.

Užívateľovi poskytuje možnosť napísať program pre PRAM, spustiť ho a sledovať jeho prácu. Navyše umožňuje užívateľovi porovnávať jednotlivé typy PRAMov a všímať si rozdiely medzi ich prácou. Môže si takto inou cestou vyskúšať ako funguje PRAM a touto cestou ho pochopiť.

Triedy PRAMu by sa mohli doplniť o chýbajúce typy Arbitrary CRCW PRAM a CROW-PRAM.Na druhej strane k ich implementácii stačí naprogramovať triedu pre ich sadu spoločných registrov. Ďalšia vhodná zmena by bola, zmeniť pamäťové registre PRAMu z integerov na stringy, čo by presnejšie simulovalo nekonečnú pamäť PRAMu. Táto zmena by nemala byť náročná, avšak mne sa integer zdal dostačujúci pre potreby vyskúšania si a experimentovania na PRAMe.

Použitá literatúra

[1] Stephen J. Chapman. Začíname programovať v jazyce JAVA. Computer Press, Praha, 2001.

[2] http://java.sun.com/j2se/1.4.2/docs/api/index.html

[3]R. Greenlaw, H. James Hoover, W. L. Ruzzo. Limits to Parrallel computation: P-completeness theory. Oxforf university press, 1995.

Príloha I.

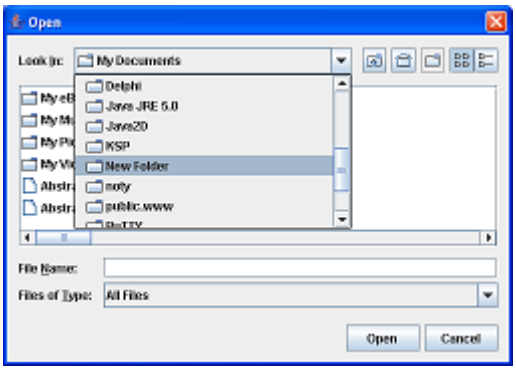
Program hľadania maxima na modele COMMON CRCW PRAM:

```
0.  ident
1.  R[1]=R[0] {ident}
2.  const 0
3.  R[2]=R[0]{0}
4.  const 1
5.  R[3]=R[0]{1}
6.  R[0]=C[0]{N}
7.  R[5]=R[0]{N}
8.  sub 3
9.  R[4]=R[0] {N-1}            //uvod, inicializacia
10. <cyklus>ISLESS(1,5) koncyklu
11. R[0]=R[1] {xx}
12. SUB 5
13. R[1]=R[0]  {buduce j}
14. R[0]=R[2]  {xx}
15. ADD 3
16. R[2]=R[0] {buduce i}
17. GOTO  cyklus
18. <koncyklu> R[10]=R[1] {i}
19. R[11]=R[2] {j}
20. R[0]=R[1] {i+1}
21. add 3
22. R[12]=R[0] {i+1}
23. R[0]=R[2] {j+1}
24. add 3
25. R[13]=R[0] {j+1}           //R[10]=i,R[11]=j
26. ident
27. ISLESS(0,5) vynuluj
28. GOTO vseci
29. <vynuluj>ADD 5 /
30. add 3
31. C[R[0]]=R[2]
32. <vseci>R[1]=C[R[12]]
33. R[2]=C[R[13]]
34. ISLESS(1,2) niejemin
35. GOTO dalej
36. <niejemin>R[0]=R[10]
```

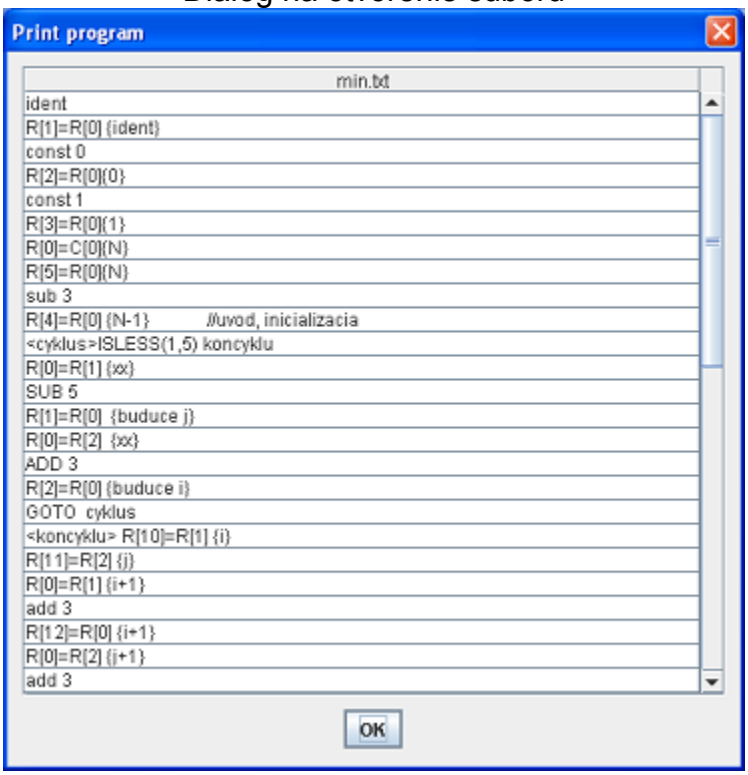
```
37. add 5    /
38. add 3
39. C[R[0]]=R[3]
40. <dalej>R[0]=R[10]
41. add 5  /
42. add 3
43. R[6]=R[0]
44. R[0]=R[4]
45. sub 11
46. IFZERO najdimin
47. HALT
48. <najdimin> R[0]=C[R[6]]
49. IFZERO ulozmin
50. HALT
51. <ulozmin> R[0]=C[R[12]]
52. C[0]=R[0]
53. HALT
```

Príloha II.

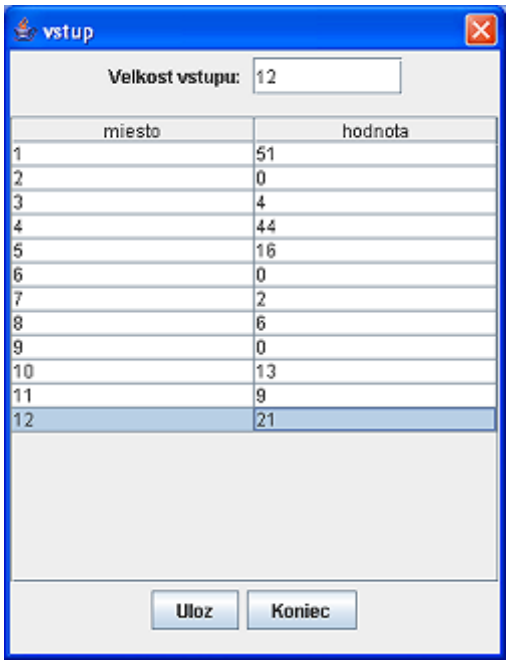
Obrázky niektorých okien, s ktorými sa užívateľ môže stretnúť v Prostredí pre experimentovanie s PRAMami:



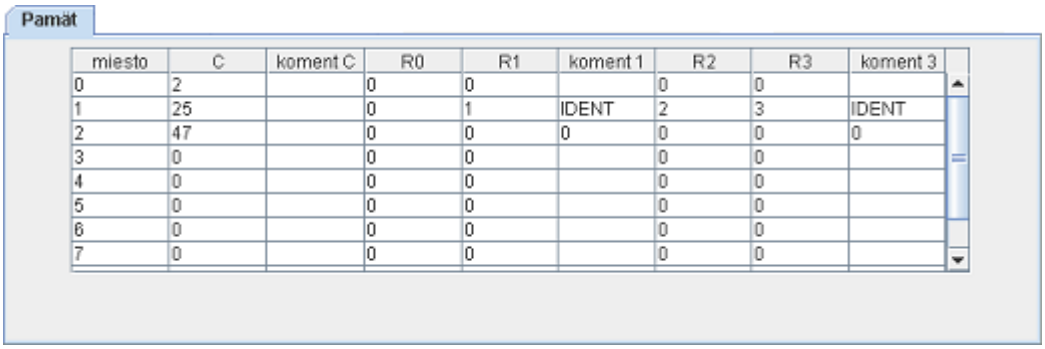
Dialóg na otvorenie súboru



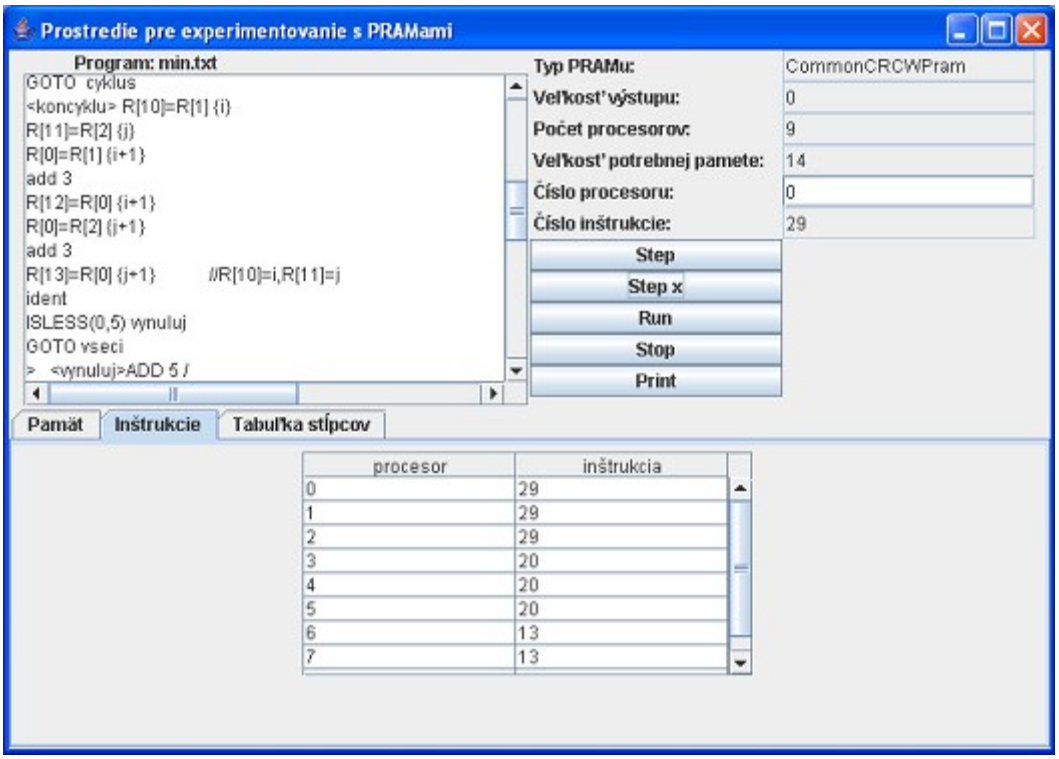
Formát pre tlač programu



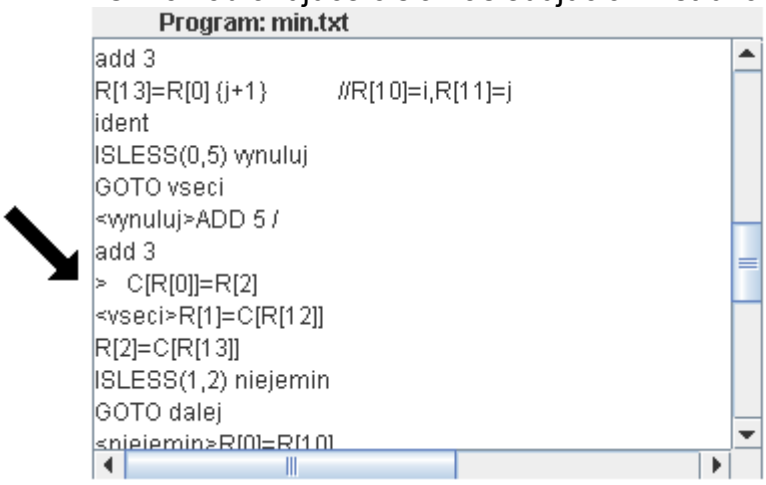
Tu môžeme editovať vstup



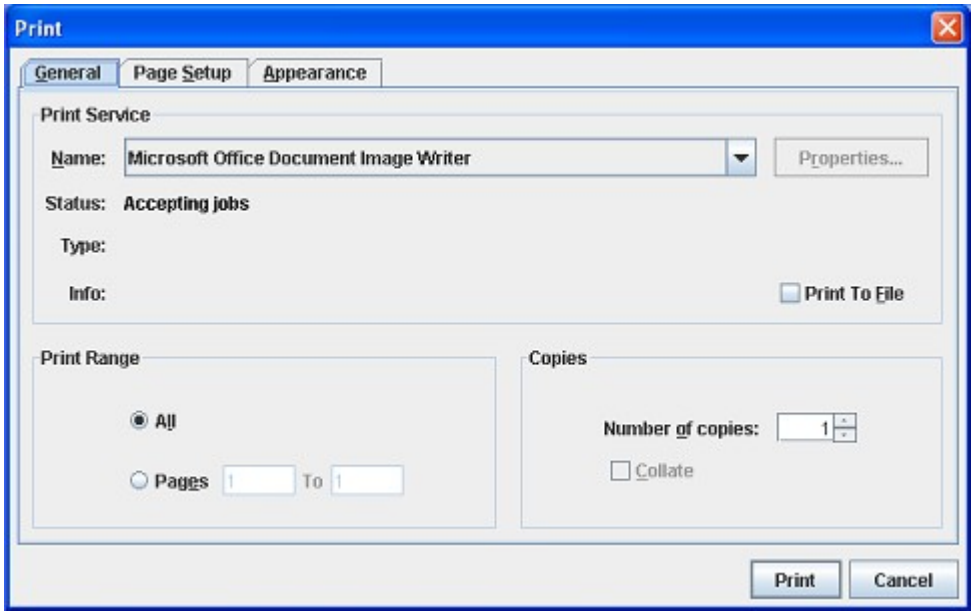
Časť okna zobrazujúca pamäť.



Okno zobrazujúce čísla nasledujúcich inštrukcií



Zvýraznenie riadku počas behu programu



Dialóg pre tlačenie.

Príloha III

CD s prostredím pre experimentovanie s PRAMami.