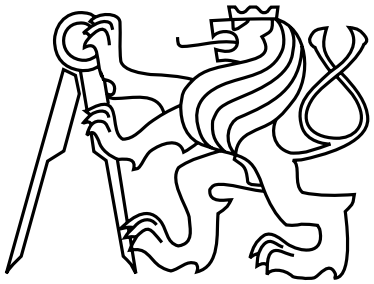
České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra kybernetiky



Bakalářská práce

Paralelní řešič soustav lineárních rovnic v aritmetice kódů zbytkových tříd

Martin Sadílek

Vedoucí práce: Ing. Tomáš Zahradnický, Ph.D.

Studijní program: Softwarové technologie a management, Bakalářský

Obor: Inteligentní systémy

26. května 2011

Poděkování

Ze všeho nejdříve bych chtěl poděkovat vedoucímu mé bakalářské práce, Ing. Tomáši Zahradnickému, Ph.D., za jeho ochotu, trpělivost, cenné rady a také za podklady k bakalářské práci.

Dále bych chtěl poděkovat MetaCentru a jeho pracovníkům za to, že mi umožnili testovat svoji aplikaci na počítačích s vícejádrovými procesory.

Na závěr bych chtěl ze všecho nejvíce poděkovat svojí rodině, která mě vždy podporovala a vytvářela mi pevné zázemí při psaní bakalářské práce.

Prohlášení

Prohlašuji, že jsem práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.
Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Kladně dne 26. 5. 2011 .....

Abstract

This bachelor thesis deals with a portion of a MOSFET parameter extraction algorithm that is responsible for solution of a set of linear equations in the multiple-modulus arithmetic of the residual number system. The arithmetic is especially appropriate for parallel computing hence it transforms solution of a set of linear equations onto solutions of multiple independent sets of linear congruencies. The algorithm of the solver is optimized for multiple processor cores at a thread level. The run and scalability of the algorithm is tested in Cesnet z. s. p. o's MetaCentrum NGI environment.

Abstrakt

Tato bakalářská práce se zabývá částí algoritmu extrakce parametrů pro tranzistor MOSFET, která je odpovědná za řešení soustav lineárních rovnic, a která používá mnohamodulovou aritmetiku kódů zbytkových tříd. Tato aritmetika je zvlášť vhodná pro paralelní výpočty, protože převádí řešení jedné soustavy lineárních rovnic na řešení mnoha nezávislých soustav lineárních kongruencí. Algoritmus řešiče je prací optimalizován pro běh na vícejádrových procesorech na úrovni vláken. Běh a škálovatelnost algoritmu je testován v prostředí MetaCentra NGI společnosti Cesnet z. s. p. o.

Obsah

Kapitola 1

Úvod

Extrakce parametrů je proces, při kterém získáváme hodnoty parametrů matematického modelu představujícího popis nějaké fyzikální reality z množiny naměřených dat. Matematický model můžeme chápat jako funkci mnoha proměnných. Některé proměnné odpovídají přímo naměřeným datům, jiné proměnné fyzikálním konstantám, některé proměnné neznáme anebo nejsme schopni změřit. Část proměnných matematického modelu označujeme jako tzv. parametry a cílem procesu extrakce parametrů je tyto parametry nalézt tak, aby rozdíly mezi naměřenými daty a daty predikovanými z matematického modelu byly pokud možno minimální. Rovnost mezi naměřenými daty a daty z matematického modelu můžeme požadovat jen zřídkakdy, protože naměřená data jsou v naprosté většině zatížena chybami.

Proces extrakce parametrů je optimalizační úloha, která hledá hodnoty parametrů matematického modelu tak, aby rozdíly mezi naměřenými data a daty predikovanými z matematického modelu byly minimální. K řešení této minimalizační úlohy lze použít například metodu nejmenších čtverců anebo metodu maximální věrohodnosti. Konkretní algoritmus extrakce parametrů, na kterém staví i tato bakalářská práce, je popsán v [9, 5]. Metoda extrakce parametrů popsaná v těchto pracech vychází z metody maximální věrohodnosti, která vyžaduje řešení soustavy nelineárních rovnic (SNR) k maximalizaci věrohodnostní funkce. Tato SNR je linearizována, čímž vzniká požadavek na řešení soustavy lineárních rovnic (SLR).

Vzhledem k tomu, že zaokrouhlovací chyby mohou mít značný vliv na řešení SLR, používají citované práce k řešení SLR mnoha modulární aritmetiku kódů zbytkových tříd. Implementace procesu extrakce paremetrů [9] provádí řešení SLR zvláštním modulem, jehož optimalizace pro běh ve vícejádrovém prostředí je předmětem této bakalářské práce.

Testování algoritmu řešiče se bude provádět v prostředí MetaCentra NGI [7] společnosti Cesnet z. s. p. o. [1]. MetaCentrum se stará o provoz a uspořádání výpočetní infrastruktury a datových úložišť výkonné výpočetní techniky rozmístěnné po některých českých akademických institucích. MetaCentrum pomáhá jakémukoli samostatnému akademickému pracovišti v České republice efektivně využít dostupné výpočetní techniky pro řešení náročných výpočtů, jejichž uskutečnění je nad možnosti pracoviště.

Práce je dělena do 5 kapitol, které mají následující strukturu.

Úvod: Popisuje problematiku extrakce parametrů matematických modelů. Klíčové je při extrakci parametrů řešení soustav rovnic.

Řešení soustav lineárních rovnic v aritmetice kódů zbytkových tříd: Kapitola popisuje řešení soustav lineárních rovnic v aritmetice kódů zbytkových tříd.

Optimalizace algoritmu řešení soustav lineárních rovnic v aritmetice kódů zbytkových tříd pro vícejádrové procesory: Zde se popisují algoritmy pro řešení soustavy lineárních rovnic v aritmetice kódů zbytkových tříd upravené pro běh ve více vláknech.

Implementace a testování: Kapitola obsahuje popis, jak byl algoritmus implementován a že byl řešič testován i na vícejádrových strojích v prostředí MetaCentra NGI společnosti Cesnet z. s. p. o.

Závěr: Obsahuje shrnutí této bakalářské práce. Jak se povedlo implementovat funkční řešič soustav lineárních rovnic v aritmetice kódů zbytkových tříd a jakých výsledků jsme dosáhli.

Kapitola 2

Řešení soustav lineárních rovnic v aritmetice kódů zbytkových tříd

2.1 Úvod

V následující části práce se budeme zabývat řešením soustavy:

$$\mathbf{A}\Delta\mathbf{p} = \mathbf{b}, \tag{2.1}$$

kde $\mathbf{A} \in \mathbb{R}^{P\times P}$ je maticí soustavy P rovnic o P neznámých pocházející z algoritmu extrakce parametrů, $\mathbf{b} \in \mathbb{R}^P$ je vektor pravých stran soustavy a $\mathbf{\Delta p} \in \mathbb{R}^P$ je hledané řešení SLR, které představuje změnu hodnot parametrů pro následující iterační krok extrakčního algoritmu. Existující implementace modulu řešení SLR je popsána v [9]. Soustava (2.1) je řešena v aritmetiku kódů zbytkových tříd. Tato aritmetika poskytuje výborné předpoklady pro paralelní počítání, protože řešení soustav lineárních rovnic rozdělí na řešení tolik na sobě nezávislých soustav lineárních kongruencí (SLC), kolik použijeme modulů. Vzhledem k tomu, že SLC jsou na sobě nezávislé, probíhá jejich řešení paralelně.

Jako $\mathbf{W}$ budeme označovat rozšířenou matici soustavy $\mathbf{A}$, kterou definujeme jako:

$$\mathbf{W} = \left(\begin{array}{ccc|c} a_{1,1} & \cdots & a_{1,P} & b_1 \\ \vdots & \ddots & \vdots & \vdots \\ a_{P,1} & \cdots & a_{P,P} & b_P \end{array}\right). \tag{2.2}$$

Operaci $b \equiv a \pmod{m}$ budeme zapisovat jako $a = |b|_m$, kde jednotlivé moduly budeme označovat jako m_1, m_2, …, m_Q, kde Q označuje počet různých prvočíselných modulů. Vícemodulární aritmetika kódů zbytkových tříd je ekvivalentní jednomodulární aritmetice kódů zbytkových tříd s modulem $M = \prod_{q=1}^{Q} m_q$.

2.2 Řešení soustav lineárních rovnic v aritmetice kódů zbytkových tříd

Algoritmus řešení SLR využívá aritmetiku kódů zbytkových tříd (RNS) s mnoha moduly, raději než aritmetiku kódů zbytkových tříd s jedním velkým modulem. Jako moduly jsou používána 15bitová čísla, jejichž násobek se bez problémů vejde do 32bitového registru. Řešení SLR v RNS probíhá ve 4 hlavních početních částech, kterými jsou škálovací transformace, dopředná transformace, řešení soustav lineárních kongruencí a zpětná transformace.

2.2.1 Škálování matice

Škálování matice je nezbytná etapa při řešení SLR z toho důvodu, že upravuje matici soustavy tak, aby byly hodnoty v matici připraveny pro aplikace operací modulo. Algoritmus škálování matice je popsán například v [3] a není předmětem bakalářské práce.

2.2.2 Dopředná transformace

Účelem dopředné transformace je převedení rozšířené matice soustavy do zbytkových reprezentací. To se provede aplikací operace modulo na matici $\mathbf{W}$, kde jako modul použijeme postupně $m_1, \ldots, m_Q$, čímž vznikne Q rozšířených matic soustavy, které označíme jako $\mathbf{W}^{(k)} = |\mathbf{W}|_{m_k}$, pro $k = 1 \ldots Q$. Dopřednou transformaci popíšeme následujícím algoritmem [9]:

Algoritmus 1 Dopředná transformace

```
1: for k = 1 to Q do
2:   for i = 1 to P do
3:     for j = 1 to P + 1 do
4:       w_{i,j}^{(k)} = |w_{i,j}|_{m_k}
5:     end for
6:   end for
7: end for
```

2.2.3 Řešení soustav lineárních kongruencí

Výsledkem předchozího kroku bylo Q nezávislých soustav lineárních kongruencí (SLC), které je třeba vyřešit. Soustavy lze vyřešit pomocí Gauss-Jordanovy eliminace s modulární pivotizací [9]. Řešení probíhá v množinách $\mathbb{Z}_{m_q}$ Pivotizace je v algoritmu životně důležitá procedura, která kontroluje, zda je prvek $w_{i,i}^{(k)}$ invertibilní v daném modulu m_k a pokud není, tj. pokud $|w_{i,i}^{(k)}|_{m_k} = 0$, je nutné v matici provést výměnu řádku i s některým z řádků $i+1, i+2, \ldots, P$ tak, aby po výměně platilo $w_{i,i}^{(k)} \neq 0$ a změnit znaménko determinantu matice. Pokud bychom pivotizaci nepoužívali, je reálnou hrozbou, že řešení SLC by skončilo kvůli nemožnosti vypočíst multiplikativní inverzi v modulu m_k. V případě, že při pivotizaci nenalezne v $i$tém sloupci žádnou nenulovou hodnotu, musíme SLC prohlásit za singulární

a řešení ukončit. SLC, které se při řešení ukázaly jako singulární, jsou z dalšího řešení vyřazeny, odpovídající m_k jsou vyřazeny a Q je poníženo o počet singulárních soustav. Extrakce parametrů [9] používá následující algoritmus pro řešení SLC, který současně s řešením počítá hodnotu determinantu soustavy:

Algoritmus 2 Řešení soustav lineárních kongruencí

1: **for** $k = 1$ to Q **do**
2: $det_k = 1$
3: **for** $i = 1$ to P **do**
4: **if** $w_{i,i}^{(k)} = 0$ **then**
5: **for** $j = i + 1$ to P **do**
6: **if** $w_{j,i}^{(k)} \neq 0$ **then**
7: **for** $n = i$ to $P + 1$ **do**
8: swap$(w_{i,n}^{(k)}, w_{j,n}^{(k)})$
9: **end for**
10: $det_k = | - det_k|_{m_k}$
11: *Go to Pokracuj*
12: **end if**
13: **end for**
14: Error("Pivot nenalezen, soustava je singulární.")
15: **end if**
16: *Pokracuj*:
17: $det_k = |det_k \cdot w_{i,i}^{(k)}|_{m_k}$
18: $u = |(w_{i,i}^{(k)})^{-1}|_{m_k}$
19: **for** $n = i$ to $P + 1$ **do**
20: $w_{i,n}^{(k)} = |w_{i,n}^{(k)} \cdot u|_{m_k}$
21: **end for**
22: **for** $j = 1$ to P, **where** $j \neq i$ **do**
23: $u = w_{j,i}^{(k)}$
24: **for** $n = i$ to $P + 1$ **do**
25: $w_{j,n}^{(k)} = |w_{j,n}^{(k)} - w_{i,n}^{(k)} \cdot u|_{m_k}$
26: **end for**
27: **end for**
28: **end for**
29: **end for**

2. Inicializace hodnoty determinantu aktuálně řešené soustavy lineárních kongruencí.

4. Kontrola invertibilnosti prvku $w_{i,i}$.

5-10. V případě $w_{i,i} = 0$ prohození řádků matice tak, aby $w_{i,i} \neq 0$, a u hodnoty determinantu se změní znaménko.

18. Výpočet inverzní hodnoty prvku $w_{i,i}$ v modulu m_k.

19-21. Vynásobení $i$tého řádku matice inverzní hodnotou prvku $w_{i,i}$.

22-27. Standardní operace Gaussovy eliminace, odečtení násobku $i$tého řádku matice od $j$tého tak, aby $w_{j,i} = 0$.

2.2.4 Zpětný převod

Předchozí krok nám poskytl maximálně Q[1] řešení SLC, která je třeba nyní převést z jejich zbytkové reprezentace zpět do množiny racionálních čísel, a to provádí algoritmus zpětné transformace (zpětného převodu). Převod zbytkové reprezentace do celých čísel lze udělat za pomoci zpětného chodu Čínské věty o zbytcích, nebo za pomoci Garnerova algoritmu, někdy též označovaného jako Mixed Radix Conversion (MRC) [4]. Algoritmus extrakce parametrů upřednostňuje Garnerův algoritmus, protože během chodu algoritmu se neobjevují, narozdíl od Čínské věty o zbytcích, obrovská čísla, která by se nevešla do registru počítače.

Algoritmus 3 Zpětný převod

1: **for** $i = 1$ to Q **do**
2: $den_i = det_i$
3: **end for**
4: **for** $i = 2$ to Q **do**
5: **for** $j = i$ to Q **do**
6: $den_j = (den_j - den_{i-1}) \bmod m_j$
7: $den_j = (den_j \cdot |m_{i-1}^{-1}|_{m_j}) \bmod m_j$
8: **end for**
9: **end for**
10: $DEN = \sum_{i=1}^{Q}(den_{i-1} \prod_{j=1}^{i-1} m_j)$
11: **if** $DEN > \lfloor M/2 \rfloor$ **then**
12: $DEN = DEN - M$
13: **end if**
14: **for** $k = 1$ to P **do**
15: **for** $i = 1$ to Q **do**
16: $num_i = det_i \cdot \mathbf{x}_{m_i,k}$
17: **end for**
18: **for** $i = 2$ to Q **do**
19: **for** $j = i$ to Q **do**
20: $num_j = (num_j - num_{i-1}) \bmod m_j$
21: $num_j = (num_j \cdot |m_{i-1}^{-1}|_{m_j}) \bmod m_j$
22: **end for**
23: **end for**
24: $NUM = \sum_{n=1}^{Q}(num_{n-1} \prod_{o=1}^{n-1} m_o)$
25: **if** $NUM > \lfloor M/2 \rfloor$ **then**
26: $NUM = NUM - M$
27: **end if**
28: $x_k = \frac{NUM}{DEN}$
29: **end for**

1-3. Inicializace proměnné den_i na hodnotu determinantu soustavy lineárních kongruencí řešené v $i$tém modulu.

[1] Q mohlo být v předchozím kroku být sníženo, pokud některá ze SLC byla singulární.

4-9. Garnerův algoritmus pro výpočet determinantu soustavy lineárních rovnic.

10. Výpočet hodnoty jmenovatele výsledku DEN, která představuje hodnotu determinantu soustavy lineárních rovnic řešené v aritmetice kódů zbytkových tříd.

11-13. Korekce znaménka výsledku.

14-29. Převod výsledků soustav lineárních kongruencí ve zbytkové reprezentaci do racionálních čísel.

15-17. Inicializace proměnné num_i na hodnotu součinu determinantu a $k$té hodnoty vektoru $\mathbf{x}$ ze soustavy lineárních kongruencí řešené v modulu m_i.

18-23. Garnerův algoritmus pro výpočet x_k.

24. Hodnotu činitele výsledku lze vypočítat pomocí Hornerova schématu.

25-27. Korekce znaménka.

28. Výpočet x_k, které představuje část hledaného řešení.

2.3 Příklad na řešení soustavy lineárních rovnic v aritmetice kódů zbytkových tříd

Vzhledem k tomu, že aritmetika kódů zbytkových tříd není běžnou aritmetikou používanou pro řešení SLR, považujeme za důležité ukázat, jak řešení SLR probíhá na příkladu. Příklad ukážeme na řešení soustavy ve tvaru $\mathbf{Ax} = \mathbf{b}$ se třemi prvočíselnými moduly $m_1 = 103$, $m_2 = 109$ a $m_3 = 113$. V příkladu bude řešen příklad zadán následujícími maticemi:

$$\mathbf{A} = \begin{pmatrix} 3 & 6 & 3 \\ 1 & 2 & \frac{115}{2} \\ 4 & 10 & 10 \end{pmatrix}, \qquad \mathbf{b} = \begin{pmatrix} \frac{1}{2} \\ \frac{3}{2} \\ 1 \end{pmatrix}. \tag{2.3}$$

Nejdříve vytvoříme rozšířenou matici soustavy $\mathbf{W}$:

$$\mathbf{W} = \left(\begin{array}{ccc|c} 3 & 6 & 3 & \frac{1}{2} \\ 1 & 2 & \frac{115}{2} & \frac{3}{2} \\ 4 & 10 & 10 & 1 \end{array}\right). \tag{2.4}$$

2.3.1 Škálování matice

Matice $\mathbf{W}$ obsahuje desetinná čísla, na které by nebylo možné aplikovat operaci modulo, a proto je operace škálování matice nutná. Nejjednodušším způsobem škálování je vynásobit všechny prvky $w_{i,j}$ hodnotou 2, čímž získáme matici $\mathbf{W}'$:

$$\mathbf{W}' = \left(\begin{array}{ccc|c} 6 & 12 & 6 & 1 \\ 2 & 4 & 115 & 3 \\ 8 & 20 & 20 & 2 \end{array}\right). \tag{2.5}$$

2.3.2 Dopředná transformace

V tomto kroku vezmeme matici $\mathbf{W}'$ a z ní vytvoříme odvozené matice, které postupně vytvoříme aplikací operace modulo na všechny prvky $\mathbf{W}'$ pro všechny vstupní moduly, získáme tedy tolik matic, kolik máme modulů. V našem případě 3 matice, a to matici $\mathbf{W}^{(103)} = |\mathbf{W}'|_{103}$, matici $\mathbf{W}^{(109)} = |\mathbf{W}'|_{109}$ a matici $\mathbf{W}^{(113)} = |\mathbf{W}'|_{113}$.

$$\mathbf{W}^{(103)} = \left(\begin{array}{ccc|c} 6 & 12 & 6 & 1 \\ 2 & 4 & 12 & 3 \\ 8 & 20 & 20 & 2 \end{array}\right), \mathbf{W}^{(109)} = \left(\begin{array}{ccc|c} 6 & 12 & 6 & 1 \\ 2 & 4 & 6 & 3 \\ 8 & 20 & 20 & 2 \end{array}\right), \mathbf{W}^{(113)} = \left(\begin{array}{ccc|c} 6 & 12 & 6 & 1 \\ 2 & 4 & 2 & 3 \\ 8 & 20 & 20 & 2 \end{array}\right). \quad (2.6)$$

2.3.3 Řešení soustav lineárních kongruencí

Řešení soustavy lineárních kongruencí $\mathbf{W}^{(103)}$

$$\left(\begin{array}{ccc|c} \boxed{6} & 12 & 6 & 1 \\ 2 & 4 & 12 & 3 \\ 8 & 20 & 20 & 2 \end{array}\right)_{103} \cdot(|6^{-1}|_{103} = 86) = \left(\begin{array}{ccc|c} 1 & 2 & 1 & 86 \\ 2 & 4 & 12 & 3 \\ 8 & 20 & 20 & 2 \end{array}\right)_{103} = \quad (2.7)$$

$$= \left(\begin{array}{ccc|c} 1 & 2 & 1 & 86 \\ 0 & 0 & 10 & 37 \\ 8 & 20 & 20 & 2 \end{array}\right)_{103} = \left(\begin{array}{ccc|c} 1 & 2 & 1 & 86 \\ 0 & 0 & 10 & 37 \\ 0 & 4 & 12 & 35 \end{array}\right)_{103} = \quad (2.8)$$

V (2.8) v matici $\mathbf{W}^{(103)}$ vidíme, že na pozici (2, 2) je nulová hodnota, což by normálně znamenalo problém při eliminaci, jelikož neumíme spočítat $|0^{-1}|_{103}$. Proto se používá pivotizace, která vyhledá nejbližší další řádek i, kde na pozici $(i, 2)$ je hodnota nenulová, a řádky poté prohodí. V našem případě se prohodí druhý řádek se třetím. Musíme pak také počítat s tím, že touto operací se změní znaménko determinantu.

$$= \left(\begin{array}{ccc|c} 1 & 2 & 1 & 86 \\ 0 & \boxed{4} & 12 & 35 \\ 0 & 0 & \boxed{10} & 37 \end{array}\right)_{103} \begin{array}{l} \cdot(|4^{-1}|_{103} = 26) \\ \cdot(|10^{-1}|_{103} = 31) \end{array} = \left(\begin{array}{ccc|c} 1 & 2 & 1 & 86 \\ 0 & 1 & 3 & 86 \\ 0 & 0 & 1 & 14 \end{array}\right)_{103} = \quad (2.9)$$

$$= \left(\begin{array}{ccc|c} 1 & 2 & 1 & 86 \\ 0 & 1 & 0 & 44 \\ 0 & 0 & 1 & 14 \end{array}\right)_{103} = \left(\begin{array}{ccc|c} 1 & 2 & 0 & 72 \\ 0 & 1 & 0 & 44 \\ 0 & 0 & 1 & 14 \end{array}\right)_{103} = \left(\begin{array}{ccc|c} 1 & 0 & 0 & 87 \\ 0 & 1 & 0 & 44 \\ 0 & 0 & 1 & 14 \end{array}\right)_{103}. \quad (2.10)$$

$$\mathbf{x}^{(103)} = \begin{pmatrix} 87 \\ 44 \\ 14 \end{pmatrix}, \det \mathbf{W}^{(103)} = \left|\boxed{6} \cdot \boxed{4} \cdot \boxed{10} \cdot (-1)\right|_{103} = 69.$$

Řešení soustavy lineárních kongruencí $\mathbf{W}^{(109)}$

$$\left(\begin{array}{ccc|c} \boxed{6} & 12 & 6 & 1 \\ 2 & 4 & 6 & 3 \\ 8 & 20 & 20 & 2 \end{array}\right)_{109} \cdot(|6^{-1}|_{109} = 91) = \left(\begin{array}{ccc|c} 1 & 2 & 1 & 91 \\ 2 & 4 & 6 & 3 \\ 8 & 20 & 20 & 2 \end{array}\right)_{109} = \tag{2.11}$$

$$= \left(\begin{array}{ccc|c} 1 & 2 & 1 & 91 \\ 0 & 0 & 4 & 39 \\ 8 & 20 & 20 & 2 \end{array}\right)_{109} = \left(\begin{array}{ccc|c} 1 & 2 & 1 & 91 \\ 0 & 0 & 4 & 39 \\ 0 & 4 & 12 & 37 \end{array}\right)_{109} = \tag{2.12}$$

V (2.12) v matici $\mathbf{W}^{(109)}$ vidíme, že na pozici (2, 2) je nulová hodnota, což by normálně znamenalo problém při eliminaci, jelikož neumíme spočítat $|0^{-1}|_{109}$. Proto se používá pivotizace, která vyhledá nejbližší další řádek i, kde na pozici $(i, 2)$ je hodnota nenulová, a řádky poté prohodí. V našem případě se prohodí druhý řádek se třetím. Musíme pak také počítat s tím, že touto operací se změní znaménko determinantu.

$$= \left(\begin{array}{ccc|c} 1 & 2 & 1 & 91 \\ 0 & \boxed{4} & 12 & 37 \\ 0 & 0 & \boxed{4} & 39 \end{array}\right)_{109} \begin{array}{l} \cdot(|4^{-1}|_{109} = 82) \\ \cdot(|4^{-1}|_{109} = 82) \end{array} == \left(\begin{array}{ccc|c} 1 & 2 & 1 & 91 \\ 0 & 1 & 3 & 91 \\ 0 & 0 & 1 & 37 \end{array}\right)_{109} \tag{2.13}$$

$$= \left(\begin{array}{ccc|c} 1 & 2 & 1 & 91 \\ 0 & 1 & 0 & 89 \\ 0 & 0 & 1 & 37 \end{array}\right)_{109} = \left(\begin{array}{ccc|c} 1 & 2 & 0 & 54 \\ 0 & 1 & 0 & 89 \\ 0 & 0 & 1 & 37 \end{array}\right)_{109} = \left(\begin{array}{ccc|c} 1 & 0 & 0 & 94 \\ 0 & 1 & 0 & 89 \\ 0 & 0 & 1 & 37 \end{array}\right)_{109}. \tag{2.14}$$

$$\mathbf{x}^{(109)} = \begin{pmatrix} 94 \\ 89 \\ 37 \end{pmatrix}, \ \det \mathbf{W}^{(109)} = \left|\boxed{6} \cdot \boxed{4} \cdot \boxed{4} \cdot (-1)\right|_{109} = 13.$$

Řešení soustavy lineárních kongruencí $\mathbf{W}^{(113)}$

Už v bodě (2.6) jsme si mohli všimnout, že první řádek matice $\mathbf{W}^{(113)}$ je po transformaci z celých čísel do zbytkové reprezentace 3násobkem druhého řádku až na sloupec pravých stran. Toto by mohlo znamenat, že řádky jsou lineárně závislé, tudíž jeden z těchto dvou řádků při úpravách Gaussovou eliminací by mohl být nulový.

$$\left(\begin{array}{ccc|c} \boxed{6} & 12 & 6 & 1 \\ 2 & 4 & 2 & 3 \\ 8 & 20 & 20 & 2 \end{array}\right)_{113} (\cdot|6^{-1}|_{113} = 19) = \left(\begin{array}{ccc|c} 1 & 2 & 1 & 19 \\ 2 & 4 & 2 & 3 \\ 8 & 20 & 20 & 2 \end{array}\right)_{113} = \tag{2.15}$$

$$= \left(\begin{array}{ccc|c} 1 & 2 & 1 & 19 \\ 0 & 0 & 0 & 78 \\ 8 & 20 & 20 & 2 \end{array}\right)_{113} \rightarrow \textit{nelze pokračovat.} \tag{2.16}$$

V bodě (2.16) vidíme, že druhý řádek až na sloupec pravých stran je nulový, potvrdil se předpoklad, matice je tak singulární, její determinant je nulový. Soustava má nekonečně mnoho řešení. Proto se musí vyřadit z dalšího řešení soustavy linernách rovnic spolu s příslušným modulem 113.

2.3.4 Zpětný převod

Nyní převedeme výsledky všech soustav, které měly řešení, a která jsou ve zbytkové reprezentaci, do množiny racionálních čísel. Nejdříve se zbytková reprezentace převede do celých čísel, a potom se celá čísla jednoduchou operací převedou na racionální čísla. Ještě potřebujeme znát hodnotu M, součin všech modulů, a hodnotu $\lfloor M/2 \rfloor$.
$M = 103 \cdot 109 = 11227$, $\lfloor M/2 \rfloor$ se pak rovná 5613.

Převedení determinantu ve zbytkové reprezentaci do jedné celočíselné hodnoty
Pro převedení determinantu na celočíselnou hodnotu použijeme Garnerův algoritmus:

$$\begin{array}{|c|r|r|}
\hline
 & m_1 = 103 & m_2 = 109 \\
 & z_1 = \boxed{69} & z_2 = 13 \\
\hline
\boxed{-} & -69 & -69 \\
 & 0 & 53 \\
\hline
\boxed{\cdot} & & |103^{-1}|_{109} = 18 \\
 & & \boxed{82} \\
\hline
\end{array} \tag{2.17}$$

$\det \mathbf{A} = \boxed{69} + 103 \cdot \boxed{82} = 8515$.
Protože je $8515 > \lfloor M/2 \rfloor$, musíme od výsledku odečíst M.
Potom $\det \mathbf{A} = 8515 - 11227 = -2712$.

Výpočet x_1
Pro výpočet $x_1^{'}$ použijeme Garnerův algoritmus:

$$\begin{array}{|c|r|r|}
\hline
 & m_1 = 103 & m_2 = 109 \\
 & z_1 = |87 \cdot 69|_{103} = \boxed{29} & z_2 = |13 \cdot 94|_{109} = 23 \\
\hline
\boxed{-} & -29 & -29 \\
 & 0 & 103 \\
\hline
\boxed{\cdot} & & |103^{-1}|_{109} = 18 \\
 & & \boxed{1} \\
\hline
\end{array} \tag{2.18}$$

$x_1^{'} = \boxed{29} + 103 \cdot \boxed{1} = 132$.
$x_1 = \dfrac{x_1^{'}}{\det \mathbf{A}} = -\dfrac{132}{2712} = -\dfrac{11}{226}$.

Výpočet x_2

$$\begin{array}{|c|r|r|}
\hline
 & m_1 = 103 & m_2 = 109 \\
 & z_1 = |44 \cdot 69|_{103} = \boxed{49} & z_2 = |89 \cdot 13|_{109} = 67 \\
\hline
\boxed{-} & -49 & -49 \\
 & 0 & 18 \\
\hline
\boxed{\cdot} & & |103^{-1}|_{109} = 18 \\
 & & \boxed{106} \\
\hline
\end{array} \tag{2.19}$$

$x_2^{'} = \boxed{49} + 103 \cdot \boxed{106} = 10967.$
Protože je $10967 > \lfloor M/2 \rfloor$, musíme od výsledku odečíst M.
Potom $x_2^{'} = 10967 - 11227 = -260.$
$x_2 = \dfrac{x_2^{'}}{\det\mathbf{A}} = \dfrac{-260}{-2712} = \dfrac{65}{678}.$

Výpočet x_3

$$\begin{array}{|c|r|r|}
\hline
 & m_1 = 103 & m_2 = 109 \\
 & z_1 = |14 \cdot 69|_{103} = \boxed{39} & z_2 = |37 \cdot 13|_{109} = 45 \\
\hline
\boxed{-} & -39 & -39 \\
 & 0 & 6 \\
\hline
\boxed{\cdot} & & |103^{-1}|_{109} = 18 \\
 & & \boxed{108} \\
\hline
\end{array} \tag{2.20}$$

$x_3^{'} = \boxed{39} + 103 \cdot \boxed{108} = 11163.$
Protože je $11163 > \lfloor M/2 \rfloor$, musíme od výsledku odečíst M.
Potom $x_3^{'} = 11163 - 11227 = -64.$
$x_3 = \dfrac{x_3^{'}}{\det\mathbf{A}} = \dfrac{-64}{-2712} = \dfrac{8}{339}.$

Výsledek
Po sloučení řešení z předešlých kroků získáváme:

$$\mathbf{x} = \begin{pmatrix} -\dfrac{11}{226} \\ \dfrac{65}{678} \\ \dfrac{8}{339} \end{pmatrix}. \tag{2.21}$$

Kapitola 3

Optimalizace algoritmu řešení soustav lineárních rovnic v aritmetice kódů zbytkových tříd pro vícejádrové procesory

V předchozí kapitole jsme popsali, jak vypadá obecný algoritmus řešení soustav lineárních rovnic v aritmetice kódů zbytkových tříd. Nyní se budeme zabývat optimalizací tohoto algoritmu pro vícejádrové procesory.

3.1 Dopředná transformace

V algoritmu dopředné transformace probíhají tři vnořené for cykly a jediná operace, která probíhá uvnitř, je nezávislá na jiných datech, proto tento algoritmus pro běh ve více vláknech rozdělíme podle vnějšího z for cyklů jdoucího po jednotlivých modulech.

Algoritmus obsahuje proměnné Q_{poc} a Q_{kon}, kde první z nich udává index počátečního modulu, tj. modulu od kterého začne vlákno počítat, a druhá udává index koncového modulu, na kterém se počítání vlákna zastaví. Počet modulů pro každé vlákno je rozdělen pokud možno stejným dílem, jelikož se zde pro každý modul počítá stejný počet operací.

Algoritmus 4 Dopředná transformace navržená pro běh ve více vláknech

1: **for** $k = Q_{poc}$ to Q_{kon} **do**
2: **for** $i = 1$ to P **do**
3: **for** $j = 1$ to $P + 1$ **do**
4: $w_{i,j}^{(k)} = |w_{i,j}|_{m_k}$
5: **end for**
6: **end for**
7: **end for**

3.2 Řešení soustav lineárních kongruencí

Algoritmus vlastního řešení jednotlivých soustav lineárních kongruencí se dá rozdělit mezi vlákna podobně jako v případě algoritmu 4, dělení podle modulů [9], řešení jednotlivých soustav lineárních kongruencí tak probíhá jako celek, jedno vlákno řeší tolik soustav, kolik má přiděleno modulů.

Opět jsou zde používány jako v případě algoritmu 4 proměnné Q_{poc} a Q_{kon}, které označují index počátečního a koncového modulu a není zde problém se synchronizací. Jako v případě transformace celých čísel do zbytkové reprezentace se zde počet modulů na početní vlákno přiděluje rovnoměrně.

Algoritmus 5 Řešení soustav lineárních kongruencí navržené pro běh ve více vláknech

1: **for** $k = Q_{poc}$ to Q_{kon} **do**
2: $det_k = 1$
3: **for** $i = 1$ to P **do**
4: **if** $w_{i,i}^{(k)} = 0$ **then**
5: **for** $j = i + 1$ to P **do**
6: **if** $w_{j,i}^{(k)} \neq 0$ **then**
7: **for** $n = i$ to $P + 1$ **do**
8: swap$(w_{i,n}^{(k)}, w_{j,n}^{(k)})$
9: **end for**
10: $det_k = | - det_k|_{m_k}$
11: *Go to Pokracuj*
12: **end if**
13: **end for**
14: Error("Pivot nenalezen, soustava je singulární.")
15: **end if**
16: *Pokracuj*:
17: $det_k = |det_k \cdot w_{i,i}^{(k)}|_{m_k}$
18: $u = |(w_{i,i}^{(k)})^{-1}|_{m_k}$
19: **for** $n = i$ to $P + 1$ **do**
20: $w_{i,n}^{(k)} = |w_{i,n}^{(k)} \cdot u|_{m_k}$
21: **end for**
22: **for** $j = 1$ to P, **where** $j \neq i$ **do**
23: $u = w_{j,i}^{(k)}$
24: **for** $n = i$ to $P + 1$ **do**
25: $w_{j,n}^{(k)} = |w_{j,n}^{(k)} - w_{i,n}^{(k)} \cdot u|_{m_k}$
26: **end for**
27: **end for**
28: **end for**
29: **end for**

Každé vlákno řeší $Q_{kon} - Q_{poc} + 1$ soustav lineárních kongruencí. Pro další popis viz algoritmus 2.

3.3 Zpětný převod

V této sekci uvedeme dva různé přístupy pro optimalizaci zpětného převodu pro vícejádrové procesory. Prvním přístupem je přístup použitý v extrakčním algoritmu [9] a druhým je náš vlastní přístup, který se snaží o rovnoměrné zatížení výpočetních vláken.

3.3.1 První verze zpětného převodu

První verze zpětného převodu dělí práci mezi vlákna po řádcích matice. Rozdělování indexů řádků matice mezi vlákna probíhá rovnoměrně.

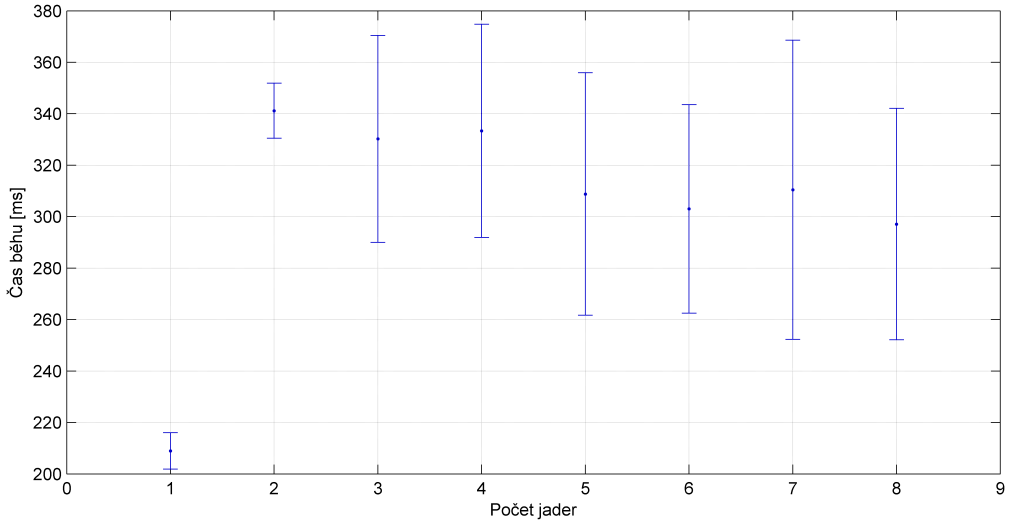
Při implementaci se neobjevil žádný problém a vlákna se nemusela nijak synchronizovat. Algoritmus obsahuje proměnné P_{poc} a P_{kon} označující indexy počátečního a koncového řádku.

Algoritmus 6 První verze zpětného převodu

1: **for** $i = 1$ to Q **do**
2: $\quad den_i = det_i$
3: **end for**
4: **for** $i = 2$ to Q **do**
5: $\quad$ **for** $j = i$ to Q **do**
6: $\quad\quad den_j = (den_j - den_{i-1}) \bmod m_j$
7: $\quad\quad den_j = (den_j \cdot |m_{i-1}^{-1}|_{m_j}) \bmod m_j$
8: $\quad$ **end for**
9: **end for**
10: $DEN = \sum_{i=1}^{Q}(den_{i-1} \prod_{j=1}^{i-1} m_j)$
11: **if** $DEN > \lfloor M/2 \rfloor$ **then**
12: $\quad DEN = DEN - M$
13: **end if**
14: **for** $k = P_{poc}$ to P_{kon} **do**
15: $\quad$ **for** $i = 1$ to Q **do**
16: $\quad\quad num_i = det_i \cdot \mathbf{x}_{m_i,\,k}$
17: $\quad$ **end for**
18: $\quad$ **for** $i = 2$ to Q **do**
19: $\quad\quad$ **for** $j = i$ to Q **do**
20: $\quad\quad\quad num_j = (num_j - num_{i-1}) \bmod m_j$
21: $\quad\quad\quad num_j = (num_j \cdot |m_{i-1}^{-1}|_{m_j}) \bmod m_j$
22: $\quad\quad$ **end for**
23: $\quad$ **end for**
24: $\quad NUM = \sum_{n=1}^{Q}(num_{n-1} \prod_{o=1}^{n-1} m_o)$
25: $\quad$ **if** $NUM > \lfloor M/2 \rfloor$ **then**
26: $\quad\quad NUM = NUM - M$
27: $\quad$ **end if**
28: $\quad x_k = \frac{NUM}{DEN}$
29: **end for**

Práce se mezi vlákna dělí až od řádku 14, každé vlákno pak počítá převod pro $P_{kon} - P_{poc} + 1$ hodnot x. Pro další popis viz popis algoritmu 3.

První verze zpětného převodu, která používá rozdělení mezi vlákna po řádcích matice, se po naprogramování a později při jednoduchém testování v MetaCentru ukázala jako neefektivní, což dokazuje následujídí obrázek s grafem závislosti rychlosti běhu na počtu jader.



Obrázek 3.1: Ukazuje rychlost běhu výpočtu první verze zpětného převodu v závislost na počtu jader procesoru.

Vyšší nepřesnosti v grafu 3.1 jsou způsobeny průměrnými hodnotami počítanými z menšího počtu opakování a velkou zátěží počítačů MetaCentra. Soustava je příliš malá na to, aby byl algoritmus efektivní a naše meření byla přesnější.

Dělení práce mezi vlákna podle počtu řádků sice bylo implementačně snažší, bohužel ve výsledku jsme nezískali s narůstajícím počtem jader nižší časy běhu. Je to způsobeno za prvé tím, že při větším počtu jader některá vlákna počítají velice malý až žádný problém, kupříkladu lze tento problém vidět při dělení výpočtu pro 13řádkovou matici za použití 16 jader, a za druhé tím, že vůbec nedělíme časově náročnější výpočet Garnerova algoritmu.

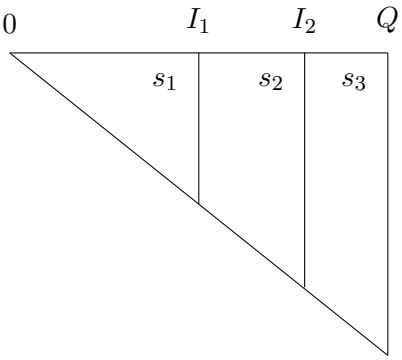
3.3.2 Druhá verze zpětného převodu

Druhá verze obsahuje proměnné Q_{poc} a Q_{kon} označující indexy počátečního a koncového modulu. Ve druhé verzi, jelikož dělíme přímo Garnerův algoritmus, dochází se vzrůstajícím indexem modulu vnitřního for cyklu ke zvyšování celkového počtu operací. Tudíž při rovnoměrném dělení indexů modulů mezi vlákna by docházelo k tomu, že vlákna počítající převod pro počáteční indexy modulů by měla práci hotovou daleko rychleji, zatímco vlákna počítající pozdější moduly by měla více práce a delší běh.

Proto musíme rozdělit Garnerův algoritmus na několik částí tak, že se v každé bude počítat stejný počet operací. Počet operací v celém Garnerově algoritmu pro Q modulů je [9]: $f(Q) = \frac{5}{2}(Q^2 - Q)$. Nyní je třeba počet operací celého algoritmu rozdělit na N částí tak, abychom v každé části provedli přibližně stejně operací. Interval modulů, který můžeme reprezentovat indexy modulů $1 \ldots Q$ rozdělíme takto:

$$\langle 1,\, Q\rangle = \langle 1,\, I_1\,) \cup \langle I_1,\, I_2\,) \cup \cdots \cup \langle I_{N-1},\, Q\rangle\,, \tag{3.1}$$

kde $I_1, \ldots, I_{N-1}$ jsou neznámé indexy modulů. Z příkladů na straně 10 a 11 je vidno, že operace Garnerova algoritmu tvoří pomyslný trojúhelník, který je zobrazen na obrázku:



Obrázek 3.2: Dělení operací Garnerova algoritmu pro $N = 3$ procesory.

Aby bylo množství práce dělené mezi procesory rovnoměrné, musí být množství operací v oblastech s_1, s_2 a s_3 přibližně stejné. Nyní zbývá určit hodnoty indexů $I_1 \dots I_{N-1}$. Počet operací v první oblasti se vypočítá:

$$s_1 = f(I_1) - f(0) = f(I_1), \tag{3.2}$$

Počet operací v druhé oblasti se vypočte:

$$s_2 = f(I_2) - f(I_1). \tag{3.3}$$

Počet operací v $N-1$. oblasti s_{N-1} se vypočte:

$$s_{N-1} = f(Q) - f(I_{N-1}). \tag{3.4}$$

Nyní máme požadavek na to, aby $s_1 = s_2 = \cdots = s_{N-1} = k$. Je-li počet operací pro celý Garnerův algoritmus $q = f(Q)$, má každá část vyřešit $k = q/N$ operací. Z rovnice (3.2) vypočteme hodnotu I_1:

$$I_1 = \frac{1 + \sqrt{1 + \frac{4(Q^2 - Q)}{N}}}{2}. \tag{3.5}$$

Druhý kořen rovnice neuvažujeme vzhledem k tomu, že $I_1 > 0$. Do rovnice (3.3) dosadíme I_1 a vypočteme I_2:

$$I_2 = \frac{1 + \sqrt{1 + \frac{4 \cdot 2(Q^2 - Q)}{N}}}{2}. \tag{3.6}$$

Druhý kořen jsme opět neuvažovali. Budeme pokračovat v postupném dosazování a pro I_{N-1} dostáváme:

$$I_{N-1} = \frac{1 + \sqrt{1 + \frac{4 \cdot (N-1)(Q^2 - Q)}{N}}}{2}. \tag{3.7}$$

Z výše uvedených rovnic vidíme, že hranici intervalu pro $i$tý procesor určíme takto:

$$I_i = \frac{1 + \sqrt{1 + \frac{4i(Q^2 - Q)}{N}}}{2}. \tag{3.8}$$

Zjednodušení vzorce

Vzorec (3.8) může být pro použití složitý, a proto přistupme k jeho zjednodušení. Můžeme předpokládat následující:

1. Množství modulů $Q > 100$, a proto $Q^2 \gg Q$, a tedy $Q^2 - Q \approx Q^2$.
2. U výrazu pod odmocninou platí že $\frac{4i(Q^2 - Q)}{N} \gg 1$, a proto jedničku zanedbáme.
3. Posun o $1/2$ modulu před odmocninou je také zanedbatelný.

Po aplikaci výše uvedených zjednodušení dostáváme:

$$I_i = \frac{1 + \sqrt{1 + \frac{4i}{N}(Q^2 - Q)}}{2} \approx \sqrt{\frac{i}{N}}Q. \tag{3.9}$$

Oba předpisy pro výpočet intervalů jako výsledek dávají iracionální čísla, a proto je třeba jako opravdový interval uvažovat zaokrouhlenou hodnotu:

$$I_i = \left\lfloor \sqrt{\frac{i}{N}}Q \right\rfloor. \tag{3.10}$$

Algoritmus 7 Druhá verze zpětného převodu

1: **for** $i = 1$ to Q **do**
2: $den_i = det_i$
3: **end for**
4: **for** $i = 2$ to Q **do**
5: **for** $j = max(i,\, Q_{poc})$ to Q_{kon} **do**
6: $den_j = (den_j - den_{i-1}) \bmod m_j$
7: $den_j = (den_j \cdot |m_{i-1}^{-1}|_{m_j}) \bmod m_j$
8: **end for**
9: **end for**
10: $DEN = \sum_{i=1}^{Q}(den_{i-1} \prod_{j=1}^{i-1} m_j)$
11: **if** $DEN > \lfloor M/2 \rfloor$ **then**
12: $DEN = DEN - M$
13: **end if**
14: **for** $k = 1$ to P **do**
15: **for** $i = 1$ to Q **do**
16: $num_i = det_i \cdot \mathbf{x}_{m_i,\,k}$
17: **end for**
18: **for** $i = 2$ to Q **do**
19: **for** $j = max(i,\, Q_{poc})$ to Q_{kon} **do**
20: $num_j = (num_j - num_{i-1}) \bmod m_j$
21: $num_j = (num_j \cdot |m_{i-1}^{-1}|_{m_j}) \bmod m_j$
22: **end for**
23: **end for**
24: $NUM = \sum_{n=1}^{Q}(num_{n-1} \prod_{o=1}^{n-1} m_o)$
25: **if** $NUM > \lfloor M/2 \rfloor$ **then**
26: $NUM = NUM - M$
27: **end if**
28: $x_k = \dfrac{NUM}{DEN}$
29: **end for**

4-9. Paralelně řešená nejnáročnější část Garnerova algoritmu pro převedení determinantu ve zbytkové reprezentaci na celé číslo.

18-23. Paralelně řešená část Garnerova algoritmu pro získání $x_k^{'} \in \mathbb{Z}$, což je x_k před převedením na racionální číslo.

Pro další popis viz algoritmus 3.

Kapitola 4

Implementace a testování

4.1 Implementace

Implementaci algoritmu řešení soustav lineárních rovnic v aritmetice kódů zbytkových tříd jsem prováděl v programovacím jazyce C++. Tento jazyk jsem použil proto, že aplikace pro extrakci parametrů [9], ze které jsem vycházel, již byla v tomto jazyce implementována.

Pro výpočet některých hodnot, jako je třeba hodnota modulu M, která se spočítá jako součin všech modulů, které nebyly vyřazeny po řešení soustav lineárních kongruencí, jsem používal knihovnu GMP [2], která je vhodná pro počítání s velkými čísly. Pomocí knihovny GMP se dá uložit prakticky neomezeně dlouhé číslo. Tato knihovna rozlišuje tři základní typy: celá čísla, racionální čísla a čísla s plovoucí řádovou čárkou.

Algoritmus řešení soustav lineárních rovnic v aritmetice kódů zbytkových tříd jsem nejdříve implementoval jen pro jedno početní vlákno, abych mohl algoritmus jednoduše otestovat a ověřit správnost počítání. Po odladění jsem provedl návrhy, jak algoritmus mezi vlákna rozdělit, aby se řešení co nejvíce zrychlilo.

Počet vytvořených vláken se odvíjí od počtu jader procesoru počítače, na kterém je aplikace spouštěna. Tudíž, když má počítač 4jádrový procesor, výše popsané algoritmy budou po každé řešit 4 vlákna. Jelikož v každém algoritmu celého řešení soustav lineárních rovnic probíhá 2 nebo více for cyklů, nabízelo se rozdělit jednotlivé části algoritmu právě na úrovni jednoho z for cyklů.

S implementací algoritmů dopředné transformace a řešení soustav lineárních kongruencí nebyl žádný větší problém. Problém však nastal u zpětného převodu. Zde se jako nejjednodušší jevilo rozdělit algoritmus v místě for cyklu jdoucího po řádcích matice, algoritmus 6. Avšak objevil se problém, že matice soustavy je malá, v případě aplikace extrakce, kterou jsem používal, je počet řádků 13. A jak potom dělit 13 řádků mezi například 16 vláken? Některá početní vlákna pak při větším počtu jader procesoru nebudou počítat nic anebo tak malý problém, že možná zabere více času vlákno vytvořit, než následující výpočet tohoto vlákna.

Proto jsem se musel pustit do jiné implementace zpětného převodu, a to implementaci řešenou na podobný způsob jako předchozí dva kroky celého řešiče, rozdělit převod v některých for cyklech jdoucí po jednotlivých modulech, algoritmus 7. A jelikož nejnáročnější

částí tohoto celého převodu je Garnerův algoritmus transformující čísla ve zbytkové reprezentaci na celá čísla, dělení práce mezi početní vlákna probíhá právě v Garnerově algoritmu na úrovni vnitřního for cyklu jdoucího po modulech.

Při výpočtu převodu ze zbytkové reprezentace do celých čísel pomocí Garnerova algoritmu jsem však při testování narazil na problém se synchronizací vláken.

Algoritmus 8 Část algoritmu 7, Garnerův algoritmus

1: **for** $i = 2$ to Q **do**
2: **for** $j = max(i, Q_{poc})$ to Q_{kon} **do**
3: $den_j = (den_j - den_{i-1}) \bmod m_j$
4: $den_j = (den_j \cdot |m_{i-1}^{-1}|_{m_j}) \bmod m_j$
5: **end for**
6: **end for**

Problém se konkrétně vyskytuje na 3. řádku algoritmu 8. Zde se od den_j odečítá den_{i-1}, avšak výpočet hodnoty den_{i-1} ne vždy muselo provést aktuální vlákno. Pokud výpočet této hodnoty neprovedlo aktuální vlákno, musí ho nějdříve spočítat některé z předchozích vláken. Jedná se o operaci $\boxed{-}$ například z příkladu (2.17) na straně 10, kde si lze všimnout, že při této operaci se vždy odečítá stejné číslo ode všech hodnot od $i$tého sloupce předchozího řádku, kde i je index z algoritmu 8.

Problém jsem vyřešil tak, že jsem si ukládal do shodně velkého pole, jako je *den*, informace o tom, na kterém indexu pole *den* je již definitivně spočítaná hodnota, aby se dala odečíst. Implementovaný algoritmus má potom v místě před 3. řádkem v algoritmu 8 aktivní čekací smyčku, která se přeruší, až bude hodnota na potřebném indexu spočítána. Aktivní čekací smyčku jsem použil proto, že se nikdy na vypočtení hodnoty nečeká moc dlouho, takže se zbytečně nezatěžuje systém.

4.2 MetaCentrum

MetaCentrum NGI [7] je projekt vzniklý v roce 1996, který každému zaměstnanci či studentovi jakékoli akademické instituce v České republice umožňuje vzdálený přístup k vysoce výkonným počítačům, na kterých může vyvíjet a testovat svoje aplikace. O MetaCentrum se stará sdružení Cesnet z. s. p. o. [1], jehož cílem je provozovat a rozvíjet národní vysokorychlostní akademickou počítačovou síť určenou pro vědu, výzkum, vývoj a vzdělávání. Jednotlivé výpočetní stroje MetaCentra jsou umístěnny v několika akademických institucích v České republice (Akademie věd ČR, Jihočeská univerzita, Masarykova univerzita, Mendelova zemědělská a lesnická univerzita v Brně, Univerzita Karlova, Vysoké učení technické v Brně, Západočeská univerzita v Plzni)

4.2.1 Obecně o MetaCentru

V prostředí MetaCentra [8] je kolem 304 fyzických strojů, které obsahují 1 až 32 procesorů (jader). Celkem je tu tedy asi 1800 procesorů, které lze používat, pokud jsou volné. O dostupnosti jednotlivých strojů a procesorech informuje na stránkách virtuální organizace MetaCentrum přehledný barevně odlišený přehled. Jednotlivými barvami se zde odlišuje plně

vytížené (modrá), částečně vytížené (světle zelená), úplně nevytížené (sytě zelená) a aktuálně nepracující (šedá) — z důvodu údržby apod. — stroje.

Abychom vůbec mohli prostředí MetaCentra používat, musíme se do MetaCentra nejdříve registrovat. Registrace je zdarma. Podává se elektronická přihláška, kde se vyplní údaje o nás, o naší akademické instituci, přihlašovací údaje a na co chceme MetaCentrum používat. Pro ověření naší totožnosti MetaCentrum umožňuje buď online ověření totožnosti u větších univerzit anebo musíme kontaktní osobě poslat vytisknutý a podepsaný formulář.

4.2.2 Přihlášení

Pokud chceme spouštět naše aplikace, do prostředí MetaCentra se nejdříve musíme vzdáleně připojit. Používá se zabezpečený komunikační protokol SSH, tudíž je docela jednoduché se připojit z unixových operačních systémů pomocí příkazu `ssh`. Z operačního systému Windows se lze připojit např. s PuTTY, který lze získat zdarma, anebo z prostředí Cygwin pomocí `ssh`.

Po přihlášení na jeden ze serverů MetaCentra přes příkazový řádek máme k dispozici normální unixové prostředí, proto je lepší mít testovanou aplikaci naprogramovanou také v unixovém prostředí.

4.2.3 Datová úložiště

Pokud chceme v prostředí MetaCentra spouštět aplikace, musíme je mít také kam uložit. K dispozici je více druhů úložišť. Nejrychlejší a nejmenší je úložiště umístěnné v `/scratch`, je na každém stroji, je však pouze lokálně dostupné. Dalším rychlým úložištěm je $NFSv3$, které je dostupné pro všechny stroje na jednom clusteru, ještě je dostupná novější alternativa úložiště $NFSv4$, která je dostupná jen na některých vybraných strojích. Nejpomalejším a dostupným na všech strojích najednou je úložiště AFS, které je dle mého právě z důvodu široké dostupnosti asi nejpraktičtější.

4.2.4 Kompilování a spouštění aplikací

V Unixu se zkompilované aplikace dají jednoduše spouštět `./aplikace` [`parametry`], v prostředí MetaCentra to není povoleno. V prostředí MetaCentra se používá plánovací systém pro spouštění aplikací PBS Torque. Úlohy se musí pouštět přes příkaz `qsub` a zadání pro spuštění aplikace se zadává do textového souboru, který se přiloží jako parametr příkazu. Jiné parametry se mohou zadat přímo za příkaz anebo se zapíší do textového souboru, který musí být ve tvaru *bash* skriptu. V textovém souboru může být nějaký kód linuxového skriptu anebo jen hlavička skriptu a následné spuštění zkompilované aplikace pomocí `./aplikace`.

Příkaz `qsub` může dostat přepínač `-q`, který udává maximální běh aplikace, kratší běhy (fronta *short*, do 2 hodin) aplikací mají větší prioritu a dlouhé běhy (fronta *long*, do měsíce) mají menší prioritu. Průměrná délka běhu se zadává pomocí fronty *normal* (běhy do 24 hodin) se střední prioritou. Dále je zde přepínač `-l walltime` = `cas`, kde za proměnnou `cas` uvedeme přesnější maximální běh aplikace, pokud ho známe. `-l walltime` =10 : 00 značí, že uživatel předpokládá, že aplikace poběží max 10 minut. Pro přepínač `-l` je ještě další nastavení, třeba `-l nodes` = `hodnota`:`ppn` = `hodnota`, kde proměnná `hodnota` u `nodes`

určuje, na kolika strojích má spuštěná aplikace běžet, anebo se zde může zadat adesa konkrétního stroje a `hodnota` u `ppn` označuje, kolik jader (procesorů) se má při výpočtu použít. U přepínače `-l` ještě můžeme za počet jader uvést, jaký typ stroje chceme použít, `-l nodes = 1:ppn = 3:x86_64` pro běh na 64bitových strojích nebo například `-l nodes = 1:ppn = 1:brno:amd64` pro běh na brněnských strojích s procesory AMD Opteron.

Dále ukáži nějaké příklady, jak v MetaCentru spustit aplikaci. Třeba příkazem:

```
qsub -q short -l nodes = 1:ppn = 1 soubor.sh.
```
(4.1)

Pokud obsah souboru `soubor.sh` bude následující:

```
#!/bin/bash
cd /afs/ruk.cuni.cz/home/username/aplikace/
./apl,
```
(4.2)

potom pomocí příkazu (4.1) zařadíme spuštění zkompilované aplikace v souboru `apl` (4.2) uloženém na *AFS* úložišti uživatele `username` do fronty aplikací s dobou běhu do 2 hodin a chceme, aby běžela na jednom stroji na jednom procesoru.

Dále můžeme aplikaci spustit takto:

```
qsub soubor.sh
```
(4.3)

Pokud obsah souboru `soubor.sh` bude následující:

```
#!/bin/bash
#PBS -q normal
#PBS -l nodes = eru2.ruk.cuni.cz:ppn = 8
cd /afs/ruk.cuni.cz/home/username/aplikace/
./apl,
```
(4.4)

potom pomocí příkazu (4.3) zařadíme spuštění zkompilované aplikace v souboru `apl` (4.4) uloženém na *AFS* úložišti uživatele `username` do fronty aplikací s dobou běhu do 24 hodin a chceme, aby běžela na jednom stroji s adresou `eru2.ruk.cuni.cz` na 8 procesorech.

Někdy se „povede“ zadat špatně nějaké údaje, chybný počet jader, strojů, chybná adresa stroje, spuštění jiné úlohy apod. Pokud má potom špatně zadaná úloha běžet třeba hodinu, je asi dobré ji zrušit. Úlohy se ruší příkazem `qdel`. Konrétní příklad:

```
qdel 66197.arien.ics.muni.cz
```
(4.5)

zastaví zpracovávání úlohy s id `66197`. Příkaz mi funguje i bez zadaného serveru, takto:

```
qdel 66197.
```
(4.6)

4.3 Vlastní testování aplikace

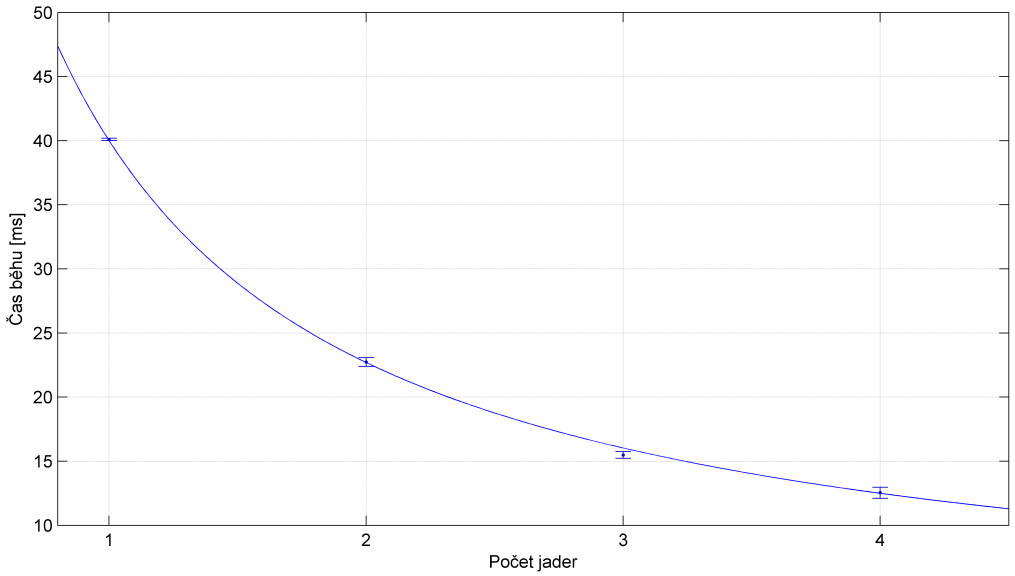
Doma na svém počítači jsem mohl svůj algoritmus testovat jen z toho hlediska, zda vrací správné hodnoty. Jelikož můj počítač obsahuje jen dvoujádrový procesor, musel jsem rychlost aplikace testovat hlavně v MetaCentru. V MetaCentru jsem si vybral stroj, který má procesor s alespoň 8 jádry, abych získal dostatek informací pro zhodnocení zrychlení aplikace v závislosti na počtu jader.

Při testování mého algoritmu řešení soustav lineárních rovnic v aritmetice kódů zbytkových tříd se používala rozšířená matice soustavy 13×14, počet modulů pro testování bylo použito 750. Časy běhů jednotlivých částí aplikace extrakce jsem měřil za pomoci třídy pro měření času, která byla v aplikaci extrakce [9] již implementována.

Celou aplikaci extrakce jsem opakoval 100× a v každé její iteraci jsem každou část řešiče spustil také 100×, z časů každé části jsem udělal aritmetický průměr, získal jsem tak celkově 100 hodnot časů pro každou část řešiče. Z těchto 100 časů jsem opět udělal průměr. A tyto průměry jsou už hodnotami vynesenými v grafech.

Spočítal jsem také směrodatné odchylky, hodnoty odchylek jsou běžně do 2 % z průměrné hodnoty času daného běhu. Maximální hodnota odchylky je 3,4 % z průměrné hodnoty, ale to je čas pro 4 jádra dopředné transformace a řešení soustav lineárních kongruencí, kde se již snižovala přesnost měření kvůli hodně rychlým běhům, jejichž čas byl rušen externími elementy.

První z grafů na obrázku 4.1 ukazuje závislost času běhu dopředné transformace a řešení soustavy lineárních kongruencí. Tyto dvě operace jsem měřil najednou, jelikož jsem je implementoval do jednoho bloku kódu, který řeší každé vlákno, a také z důvodu, že transformace celých čísel do zbytkové reprezentace je dle mého velice jednoduchá a rychlá operace, která na běh celé aplikace extrakce nemá veliký vliv.

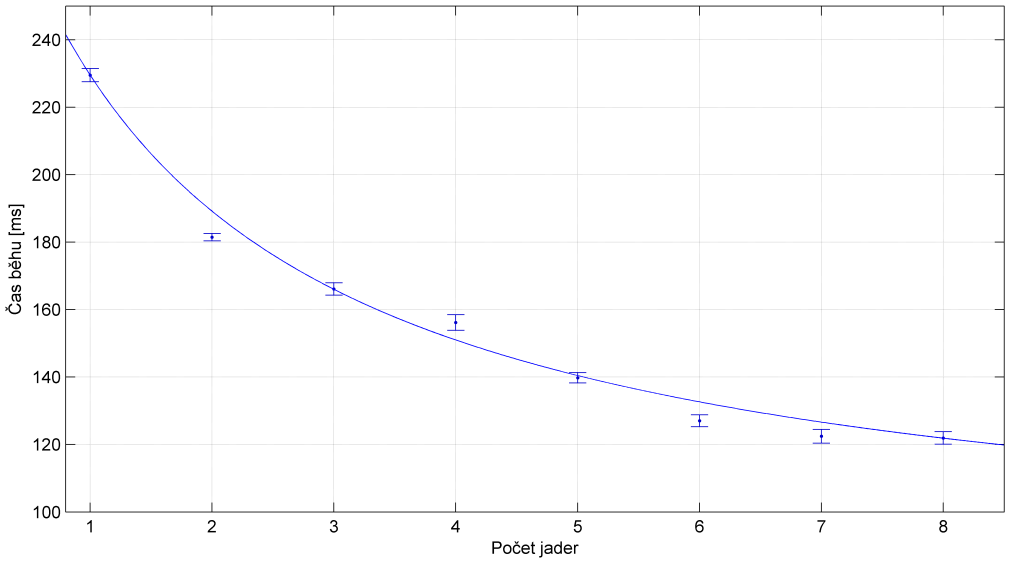


Obrázek 4.1: Ukazuje rychlost běhu výpočtu dopředné transformace a řešení lineárních kongruencí v závislost na počtu jader procesoru.

V grafu 4.1 je vidět, že zvýšení počtu jader procesoru a tím i počtu početních vláken určitě přispívá ke zrychlení běhu prvních dvou implementovaných algoritmů. Algoritmus je testován jen pro maximálně 4 jádra, jelikož časy běhů pro více jader jsem již nebyl schopen přesně změřit.

Testování dopředné transformace bylo celkově docela problematické. Za prvé, měření takto nízkých hodnot časů je docela složité, jelikož minimální změřitelné hodnoty pomocí implementované třídy, kterou jsem používal, jsou v řádech desítek ms a nižší časy třída již nezměří. A za druhé, v MetaCentru je multiuživatelské prostředí, tudíž stroje nejsou zatíženy jen mojí aplikací. Pokud jsem opakovaně měřil časy běhů na nezatíženém stroji, získával jsem přibližně stejné výsledky, avšak po připojení jiných uživatelů na stejný stroj se čas běhu aplikace znatelně zvýšil. Ke zpřesnění měření nepomohlo ani to, když jsem v každé iteraci celé aplikace měřil průměr z 10 000 hodnot.

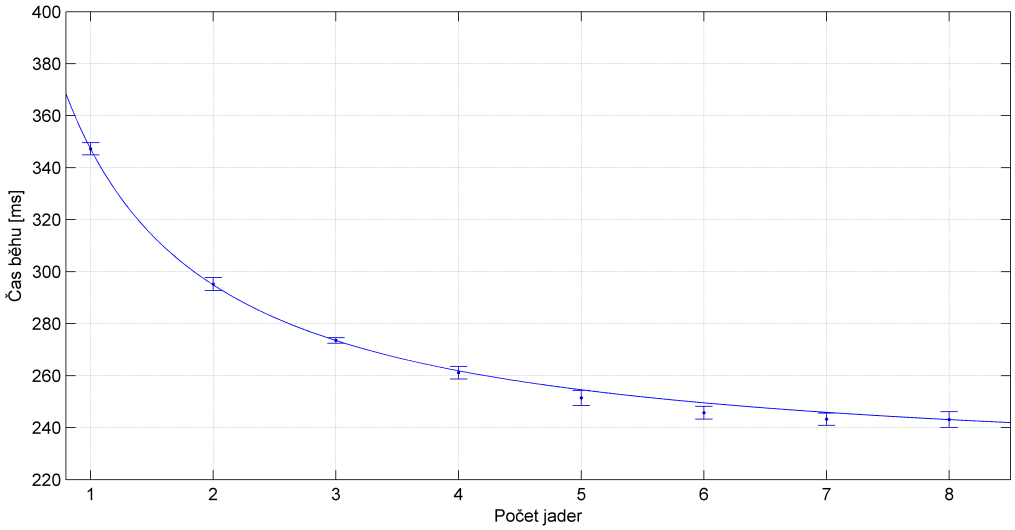
Další z grafů ukazuje závislost času běhu zpětného převodu na počtu jader procesoru.



Obrázek 4.2: Ukazuje rychlost běhu výpočtu zpětného převodu v závislost na počtu jader procesoru.

Dle grafu na obrázku 4.2 lze soudit, že i zpětný převod se s větším počtem jader zrychluje. Nezrychluje se sice tak rychle jako dopředná transformace spolu s řešení soustav lineárních kongruencí, ale přesto znatelně. Zpětný převod je citelně delší než řešení soustav lineárních kongruencí, tudíž se lépe měří. Ale i u zpětného jsem při větším počtu jader někdy potýkal s jistými nepřesnostmi v měření, které přikládám na vrub časové náročnosti vytváření více než 8 vláken spojenou s vlivem výpočtů dalších uživatelů v MetaCentru. Proto jsem zpětný převod testoval jen pro maximálně 8 jader, jelikož při větším počtu jader se vyskytovala větší chybovost a menší rozlišitelnost zlepšení.

Poslední z grafů ukazuje závislost času běhu celé aplikace extrakce na počtu jader procesoru.



Obrázek 4.3: Ukazuje rychlost běhu celé aplikace extrakce v závislost na počtu jader procesoru.

Graf na obrázku 4.3 ukazuje, že se celá aplikace také znatelně zrychlila. Celá aplikace běží asi o polovinu pomaleji než zpětný převod, tudíž zpětný převod má velký vliv na běh celé aplikace. Opět testováno jen pro maximálně 8 jader, při větším počtu jader se vyskytovaly nepřenosti v měření.

U svého algoritmu jsem testováním zjistil, že se opravdu s přibývajícími vlákny zlepšuje čas běhu. Avšak soustava rovnic a celý problém je pravděpodobně natolik jednoduchý, že při větším počtu jader jsem získával nepřesné výsledky, proto jsem měření musel často opakovat znovu, a třeba i vyčkat na chvíli, kdy na stroji budu testovat jen já, aby moje výsledky nebyly tak ovlivněny. Ani to však moc k přesnějším výsledkům nepomohlo, jelikož při větším počtu jader zde hraje roli čas vytváření vláken, který není pokaždé shodný a někdy viditelně znepřesní měření. To lze vidět u grafu 4.1, kdy pro jedno jádro je směrodatná odchylka prakticky neznatelná, zatímco pro větší počet jader již je odchylka znát, přestože není velká.

Proto by bylo asi lepší pro získání opravdu kvalitních výsledků použít nějakou přesnější metodu. Další možností je zvětšit řešený problém, a jelikož vytváření vláken zabírá určitý čas, bylo by lepší vlákna vůbec nevytvářet, ale použít thread pool, pomocí kterého jen existujícím vláknům přiřadíme práci bez nutnosti vlákna pokaždé vytvářet.

Kapitola 5

Závěr

V této práci jsme se dozvěděli něco málo o extrakci parametrů. Extrakce parametrů nám pomáhá k hledání parametrů matematicých modelů. Matematický model predikuje výstupní hodnoty, které nemusí být přesné, a extrakce parametrů nám pomůže tyto hodnoty zpřesnit. V extrakci parametrů je důležitou součástí řešení soustavy rovnic. Hlavní náplní této práce bylo řešení soustav lineárních rovnic v aritmetice kódů zbytkových tříd. Práce popisuje průběh algoritmu ve čtyřech krocích (škálování matice, dopředná transformace, řešení soustav lineárních rovnic, zpětný převod), postup byl vysvětlen i na ukázkovém příkladu.

Dalším bodem bylo vzít algoritmus řešení soustav lineárních rovnic v aritmetice kódů zbytkových tříd a optimalizovat ho pro běh na počítačích s vícejádrovými procesory. Návrh pro běh na vícejádrových procesorech zase nebyl takový problém, jelikož použitý řešič pracuje ve zbytkové reprezentaci, tudíž se nabízí dělit algoritmus v místech for cyklů jdoucích po indexech modulů. Takto jsou řešeny dopředná transformace, řešení soustav lineárních kongruencí i zpětný převod. U zpětného převodu se nabízelo ještě dělení ve for cyklu jdoucího po řádcích matice, to se však neukázalo jako efektivní. Implementace škálování matice nebyla náplní práce.

Dělení modulů mezi jednotlivá vlákna probíhá u dopředné transformace a řešení soustav lineárních rovnic rovnoměrně, pouze u zpětného převodu se musí dělit nerovnoměrně. Pro počáteční vlákna se musí přidělovat více modulů, jelikož se u prvních modulů provádí méně operací. Počet operací s rostoucím indexem modulů roste též. Pro dělení práce u zpětného převodu lze použít odvozený vzorec 3.10. U zpětného převodu byla jako u jediné části výpočtu nutná synchronizace vláken.

Pro testování algoritmu řešení SLR v aritmetice kódů zbytkových tříd jsem byl nucen použít MetaCentrum z toho důvodu, že nemám k dispozici počítač s více než 2jádrovým procesorem. Pro spouštění aplikací v MetaCentru se používá plánovací systém, který je však po osahání docela jednoduchý na ovládání.

Při testování jsem zjistil, že se jednotlivé části algoritmu řešení soustav zrychlují a zrychluje se tím i celá aplikace extrakce. Avšak vyskytly se komplikace s měřením hodně rychlých časů. Řešený problém je zkrátka jednoduchý, aplikace jako celek probíhá v rámci stovek milisekund, některé části, dopředná transformace, řešení soustav lineárních kongruencí, probíhají v rámci jednotek milisekund až desítek mikrosekund (odhad). V takto malých časech hrají čím dál tím větší roli rušivé elementy, ostatní uživatelé, vytváření vlákna apod., proto třeba

ani přesnějším měřením nemusíme získat relevantní výsledky. Pro lepší otestování aplikace bychom museli značně zvětšit řešený problém.

Literatura

[1] *CESNET z. s. p. o.*
<http://www.cesnet.cz/>

[2] *GMP: The GNU Multiple Precision Arithmetic Library.*
<http://gmplib.org/>

[3] Gregory, R. T.: *Error-free computation: Why It Is Needed and Methods For Doing It* . Robert E. Krieger Publishing Company, Inc., 1 1980.

[4] Lórencz, R.: *Aplikovaná numerická matematika a kryptografie.* Vydavatelství ČVUT, 10 2004

[5] Lórencz, R.; Reckleben C.; Hansen, K.: *A Novel Extraction Method for BJT-Parameters. J. Elec. E.*, 51:21 29, 2000.

[6] Marquardt, D. W.: *An algorithm for least-squares estimation of nonlinear parameters. Journal of Applied Mathematics.* 11(2):431-441, 1963.

[7] *MetaCentrum NGI.*
<http://www.metacentrum.cz/cs/>

[8] *MetaCentrum: Virtual Organization.*
<http://metavo.metacentrum.cz/cs/>

[9] Zahradnický, T.: *MOSFET Parameter Extraction Optimalization.* Disertační práce, Katedra počítačů, Fakulta elektrotechnická, ČVUT v Praze, Praha, 2010.

Příloha A

Obsah přiloženého CD


/


nfet


src


CExtractionApp.cpp — *třída s extrakční aplikací*


CIntervalSplitter.cpp — *umožňuje rozdělit interval na zadaný počet částí*


CJacobian.cpp — *třída představující Jacobiho matici*


CMathematicalModel.cpp — *třída s matematickým modelem*


CMatrix.cpp — *třída představující matici*


CModularSLESolver.cpp — *zde jsem implementoval svůj algoritmus řešení SLR*


CModuleBank.cpp — *třída, ve které jsou uloženy moduly pro řešení SLR*


CParameterSet.cpp — *třída představující množinu parametrů*


CTimeMeasure.cpp — *třída umožňující měřit čas běhu nějakého úseku algoritmu*


CVector.cpp — *třída představující vektor*


config.h — *soubor s různým nastavením pro aplikaci extrakce*


invbank.mod — *vyhledávací tabulka pro multiplikativní inverze 15bitových čísel*


main.cpp — *hlavní soubor aplikace*


Makefile — *projektový soubor*


sadilma2_bakalarska_prace_2011.pdf — *elektronická verze bakalářské práce*