Vliv počítačové architektury na numerické výpočty ve statistice

Miroslav Tůma [1]

1 Úvod

Následující příspěvek se zabývá řešením některých numerických úloh s přihlédnutím k vlastnostem architektur použitých počítačů. Pozornost soustředíme především na několik základních podproblémů vznikajících jako součást řešení některých statistických úloh. V oblasti počítačových architektur se zaměříme především na vektorový paralelismus a na problémy datové komunikace mezi pamětí a procesorem v sekvenčních i vektorových počítačích.

2 Numerické výpočty ve statistice

Mějme $X \in R^{m,q}$ matici pozorování hodnosti $q, m \geq q$. Nechť f je předepsaný vektor pozorování a V je symetrická a positivně definitní kovarianční matice.

Základní modelový problém lineární regrese je dán následovně (viz [21]). Najdi vektor $b \in R^q$ minimalizující

$$e^T V^{-1} e, \tag{2.1}$$

pro který platí

$$e = f - Xb. \tag{2.2}$$

Řešení tohoto problému je dáno systémem normálních rovnic

$$X^T V^{-1} X b = X^T V^{-1} f. \tag{2.3}$$

Je-li V faktorizováno pomocí Choleského faktorizace tak, že $V = LL^T$, pak můžeme (2.3) přepsat na tvar

$$X^T L^{-T} L^{-1} X b = X^T L^{-T} L^{-1} f, \tag{2.4}$$

nebo

$$A^T A b = A^T g, \tag{2.5}$$

kde

$$A = L^{-1} X, g = L^{-1} f. \tag{2.6}$$

Vzhledem k předpokladu o hodnosti matice X má matice A také hodnost q a $A^T A$ je positivně definitní. Řešení našeho problému tedy získáme jako řešení lineárního systému rovnic se symetrickou a positivně definitní maticí soustavy.

[1] Ústav informatiky a výpočetní techniky - skupina numerické lineární algebry, Československá akademie věd, Pod vodárenskou věží 2, 182 07 Prague 8 - Libeň, e-mail: tuma@cspgcs11.bitnet.

Druhá možnost řešení našeho systému spočívá v použití ortogonálních transformací. Nechť poté

$$X = LQ\begin{pmatrix} R \\ 0 \end{pmatrix}, \tag{2.7}$$

kde $Q \in R^{m,m}$ je ortogonální matice a $R \in R^{q,q}$ je horní trojúhelníková matice. Nechť

$$g = Q\begin{pmatrix} \theta \\ \theta' \end{pmatrix}, \tag{2.8}$$

kde $\theta \in R^q, \theta' \in R^{m-q}$. Pak dostaneme řešení b problému daného v (2.1) a (2.2) jako řešení systému (2.9) s trojúhelníkovou maticí soustavy.

$$Rb = \theta. \tag{2.9}$$

Můžeme se o tom přesvědčit dosazením z (2.7) a (2.8) do systému normálních rovnic (2.5). Přitom postupně dostáváme

$$(R^T 0)Q^T Q\begin{pmatrix} R \\ 0 \end{pmatrix} b = (R^T 0)Q^T Q\begin{pmatrix} \theta \\ \theta' \end{pmatrix},$$

$$R^T Rb = R^T \theta. \tag{2.10}$$

Z regularity R^T získáme (2.9).

Uvedený příklad ukázal dva důležité numerické postupy se kterými se střetneme při řešení základní úlohy lineární regrese - tzv. lineárním problému nejmenších čtverců: Choleského faktorizaci symetrické a positivně definitní matice (dekompozice V a A) a ortogonální faktorizaci obecně obdélníkové matice z $R^{m,q}$ pro $m \geq q$.

Základní problém nelineární regrese se obvykle převádí na problém obecné nepodmíněné minimalizace nebo na jeho speciální podproblém - nelineární problém nejmenších čtverců. Jeho řešení obvykle spočívá v řešení posloupnosti lineárních problémů nejmenších čtverců. V případě, že k problému danému v (2.1) a (2.2) přidáme ještě nějaká další omezení na vektor b vyplývající ze zkoumané reality, pak místo systému v (2.5) dostáváme systém promítnutý na nadrovinu určenou přidanými omezeními. To vede také na problémy řešení lineárních systémů, ale s nesymetrickými maticemi. Gaussova eliminace pak nahrazuje někde Choleského faktorizaci. Těmito třemi problémy: Choleského faktorizací, Gaussovou eliminací a ortogonální faktorizací ve vztahu k architektuře počítače se tedy budeme především zabývat.

Vraťme se nyní k našemu modelovému problému. Pro řešení lineárního problému nejmenších čtverců je přehršle dalších různých postupů. Jmenujme například převedení na řešení rozšířeného systému. Tento způsob spočívá v nalezení řešení b systému se symetrickou indefinitní maticí

$$\begin{pmatrix} I & A \\ A^T & 0 \end{pmatrix}\begin{pmatrix} x \\ b \end{pmatrix}\begin{pmatrix} g \\ 0 \end{pmatrix}.$$

Souhrn různých postupů je uveden v [4]. Ponecháme-li stranou iterační metody, tak většina z nich vyžaduje jako podúlohu Choleského faktorizaci či využívá ortogonálních transformací. Soustředíme-li se na uvedené dva přístupy, pak přímé řešení systémem normálních rovnic použijeme pouze v případě, že číslo podmíněnosti $\kappa(A)$ matice A není "příliš" velké, tj. v případě, že soustava je dobře podmíněná ($\kappa(A) = ||A||||A^{-1}||$ pro nějakou normu matice A). V opačném případě tento postup nedává dobré výsledky a jsme nuceni použít ortogonální transformace.

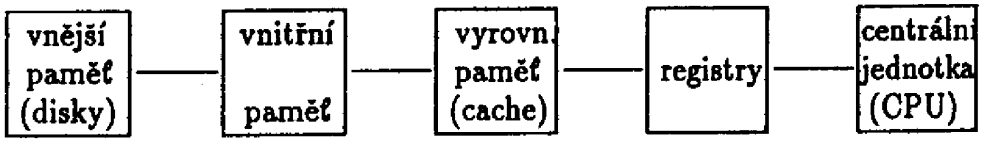
Dokončení řešení systému normálních rovnic se stane problematické pro $\kappa(A) \simeq 1/\sqrt{\epsilon}$, kde ϵ je strojová přesnost počítače (pro PC je ϵ řádově 10^{-16}). Pro ortogonální transformaci je řešení problematické obvykle až pro $\kappa(A) \simeq 1/\epsilon$.

Z hlediska asymptotického je při prvním postupu řešení lineárního problému nejmenších čtverců přes normální systém i při druhém postupu prováděném prostřednictvím Householderových zrcadlení potřeba pro $m = q$ stejný počet operací. Při $m > q$ vyžaduje první způsob méně operací.

3 Počítačové architektury a algoritmy

Vývoj v oblasti výpočetní techniky vnesl do architektur nové prvky, které jsou k dispozici uživatelům. Ačkoliv je tendence vývoje nezatěžovat uživatele novými problémy s příchodem výkonnějších počítačů, rychlost vývoje technických prvků daleko předbíhá možnosti návrhářů programového vybavení. Chceme-li efektivně používat výpočetní techniku, musíme při návrhu algoritmů a programů přihlížet aspoň k jejím hrubým charakteristikám.

První z námi sledovaných rysů je *hierarchie paměti počítače*. Typická struktura paměti je na obrázku 3.1.



Obr. 3.1: Typická struktura paměti počítače.

Paměť je obvykle tím rychlejší, čím blíže je v této struktuře centrální jednotce. Směrem k CPU také obvykle stoupá cena použitých součástek a klesá její kapacita. Proto se při běhu úlohy nevyhneme určité komunikaci mezi jednotlivými vrstvami paměťové hierarchie. Přirozeným záměrem je, aby tato komunikace byla co nejmenší. Komunikace mezi určitými vrstvami – např. mezi vyrovnávací pamětí a vnitřní pamětí – je zabezpečována technickým vybavením počítače. Mezi vnější pamětí a vnitřní pamětí je někdy zabezpečována uživatelem (používání pomocných souborů), někdy je zabezpečována programovým vybavením – např. v případě virtuální paměti, jejíž provoz je zabezpečen rozdělením paměťového prostoru na *paměťové stránky*. Nepoužívané stránky jsou odkládány na disk podle určitých, často důmyslných, pravidel.

To má vliv na konkrétní výběr algoritmu. V našich podmínkách to znamená nejenom respektování omezení velikostí reálné paměti, ale i přihlédnutí k stránkování paměti na disk a přihlédnutí k velikosti vyrovnávací paměti (cache). Návrh algoritmu by měl dosahovat co největší lokality dat v časově náročných částech algoritmu. Nepřihlédnutí k těmto faktům například při testování počítačů může znehodnotit výsledky jejich porovnávání.

Druhý ze sledovaných rysů je *vektorový paralelismus*. Mnohé dnešní počítače mají možnost efektivnějšího zpracování bloků dat nad kterými se vykonávají shodné aritmetické či logické operace. Hovoříme o zpracování vektorů nebo o *vektorových* operacích. Počítače bývají přizpůsobeny tak, že registry mohou obsahovat celé bloky dat určité velikosti (vektorové registry) a strojový kód obsahuje příkazy pro manipulaci s těmito bloky, jak pro jejich načítání a ukládání, tak pro aritmetické i logické operace.

Vlastní ukládání dat v paměti nemusí být nutně souvislé. To závisí na logice technického zařízení, které řídí jejich ukládání a čtení. Ba naopak, často jsou vektory uloženy v centrální paměti odděleně, protože u dnešních dynamických paměťových obvodů je možné přistoupit do téže fyzické paměťové oblasti po načtení nějakého paměťového prvku až po uplynutí jisté vybavovací doby. Data příslušející po sobě jdoucím indexům jednoho vektoru jsou tedy umísťována často v různých paměťových oblastech např. s pravidelnou periodou (interleave).

Fyzická realizace výhodnějšího zpracování vektorů je založena na časovém překryvu částí operací s po sobě jdoucími indexy v segmentované funkční jednotce. Každý z těchto segmentů je odpovědný za částečné dekódování, interpretaci a vykonávání instrukce. Příkladem buď násobení dvou čísel ve *funkční jednotce pro násobení v plovoucí desetinné čárce* rozčleněné na 5 segmentů. Předpokládejme, že čísla jsou uložena ve standardním formátu zahrnujícím exponent a mantisu.

Výpočet ve funkční jednotce probíhá v pěti fázích:

1. Porovnávání exponentů,

2. zarovnávání operandů,

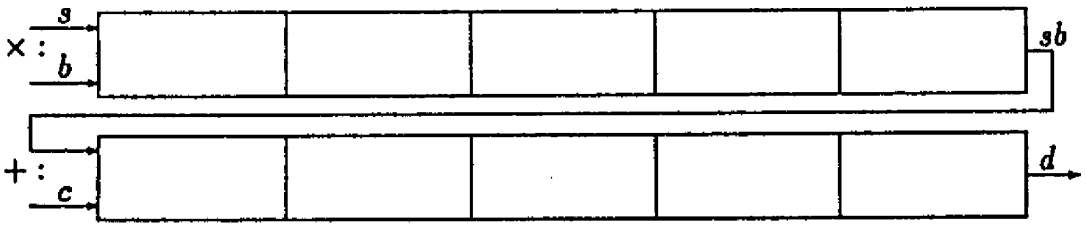
3. sčítání exponentů a násobení mantis,

4. určení normalizačního faktoru a

5. normalizace výsledku.

Mějme vektor dat $b \in R^q$, který máme násobit skalárem s abychom dostali vektor $c = bs$. Pak zatímco provádíme fázi 5 při výpočtu c_1, pak provádíme fázi 4 při výpočtu c_2, fázi 3 při výpočtu c_3, ... Tím dosahujeme na počítači rychlost nazývanou *vektorová rychlost počítače.*

Na počítači, který to umožňuje, můžeme dále zřetězit celé funkční jednotky. Mějme například spočítat výraz $d = c + sb$, kde $s \in R$ a $b, c, d \in R^q$. Mějme k dispozici funkční jednotky pro sčítání a násobení z nichž každá je segmentována na 5 částí. Výsledek z násobičky můžeme rovnou vkládat na vstup sčítačky a zřetězit tak práci obou jednotek. Pro delší vektory se časově překrývá nejen práce v jednotlivých segmentech funkčních jednotek, ale i výpočty na obou jednotkách. Zpoždění vstupu do sčítačky je dáno jen dobou k vytvoření sb_1. Situace je znázorněna na obrázku 3.2.



Obr.3.2: Zřetězení násobící a sčítací funkční jednotky.

Obecně je efektivita vektorových operací tím vyšší, čím je délka zpracovávaného bloku (délka vektoru) větší, a to až do asymptotické rychlosti provádění určité vektorové operace. Pro malé délky vektorů nemusí být zisk až tak velký, vzhledem k určitým časovým prodlevám potřebným k zahájení vektorové operace, které jsou větší než analogické prodlevy pro skalární operace.

Teprve při ideálním vektorové zpracování instrukcí, případném zřetězení operací a souběžné práci maximálního možného počtu funkčních jednotek je možné dosáhnout v našich programech rychlostí srovnatelných s údaji dodávanými výrobcem jako špičková teoretická rychlost počítače. Této hodnotě se v praxi obvykle nepřiblížíme bez znalosti základních problémů ve vztahu počítač - algoritmus. Mnoho práce za nás udělá operační systém a překladač z vyššího jazyka, nicméně spoustu problémů musíme vyřešit sami. Ať už znalostí problémů, či znalostí vhodných optimalizovaných programů.

U vektorového počítače obvykle toužíme po tom, aby se data v úlohách dala přirozeně načítat, zpracovávat a ukládat prostřednictvím operací s vektory. V námi vybraných typových úlohách je tento požadavek do značné míry splněn, pracujeme-li s hustými maticemi.

Při řešení problémů větších dimenzí mají používané matice obvykle větší množství nulových prvků. Takové matice se nazývají řídké. Z hlediska operační paměti, ale hlavně z hlediska strojového času je nutné ukládat matice v těchto problémech v "zahuštěné" formě. řídké. Tak můžeme ukládat např. jen nenulové prvky a informace o nich. Tím zmenšíme velikost potřebného prostoru, zmenšíme počet aritmetických operací s prvky matice, ale obvykle vzroste počet operací logických a operací celočíselných. Přístup k datům změní svou charakteristiku. Namísto přímé adresace prvků vektorů jsou jednotlivé nenulové prvky adresovány nepřímo.

Problém vektorů nepřímo adresovaných byl vyřešen vývojem počítačů v osmdesátých letech. Většina dnešních architektur vektorových počítačů podporuje i vektorové zpracování nepřímo adresovaných vektorů. Přesto je však toto zpracování obvykle pomalejší než zpracování přímo adresovaných vektorů. Vzhledem k tomu, že algoritmy používající nepřímo adresované nenulové prvky matic jsou i jinak vnitřně komplikovanější, je stále mnoho výzkumných projektů věnováno vývoji algoritmů,

které mají časově nejnáročnější části realizovány pomocí vektorových operací s přímo adresovanými vektory.

Následující oddíly ukazují některé příklady úprav algoritmů pro úlohy z našeho výběru vyvolané vývojem architektur počítačů.

Ani v podmínkách bez superpočítačů nejsou některé z těchto úprav příliš odtažité. Dobrým výběrem algoritmu můžeme i pro PC počítače či pracovní stanice docílit drastických časových úspor. Nové procesory pro používané počítače mají možnosti vektorové práce či řetězení funkčních jednotek (i860, R4000). I když ne všechny tyto úpravy jsou přímočaré, měli bychom si jich být vědomi při praktických výpočtech, či umět používat při sestavování algoritmů nástroje těmito principy ovlivněné.

4 Algoritmus násobení hustých matic

Tento příklad ukazuje jaké obrovské úspory v času potřebném pro běh úlohy můžeme dosáhnout jednoduchým přeformulováním algoritmu pro násobení matic. Pro tento oddíl jsme zvolili časové a prostorové údaje z reálného řešeného problému (viz [25], [10]).

Algoritmus násobení matic pro husté i řídké matice nepotřebuje rekapitulaci. Zde předpokládáme, že matice $A, B, C \in R^{q,q}$ jsou husté, kde $q = 1024$. Nechť operační systém používá techniku virtuální paměti a umožňuje uložit 65536 prvků matice na jednu stránku. Nechť operační paměť počítače má velikost 16 stránek. Při použití dvojité přesnosti pro prvky matic tomu odpovídá velikost stránky 512 kbyte a velikost operační paměti 8 Mbyte.

Není-li v průběhu výpočtu v operační paměti stránka s potřebnými daty, pak oznámí technické zařízení takzvaný *výpadek stránky* a příslušná stránka je načtena. Nechť výpadek stránky zabere přibližně 0.5 s času na zařízeních pro vstup a výstup.

Uvažujme nyní 3 způsoby počítání výrazu $C = C + A.B$ Postup bude znázorněn programovými smyčkami v obvyklé notaci. $A(:, I)$ resp. $B(J, :)$ označuje např. $I - tý$ sloupec matice A resp. $J - tý$ řádek matice B. V časových údajích se budeme zabývat časem pro vstup a výstup, nikoliv strojovým časem pro vlastní numerické výpočty.

- První způsob je tzv. algoritmus skalárního součinu. Předpokládejme, že matice A, B, C jsou uloženy tak, že na každé stránce je uloženo 64 sloupců nějaké matice. Potřebujeme tedy pro matice 48 stránek paměti celkem.

```
for i=1 to q
   for j=1 to q
      for k=1 to q
         c(i,j)=c(i,j) + a(i,k)*b(k,j)
      end k
   end j
end i.
```

Obr. 4.1: Násobení matic algoritmem skalárního součinu

Uspořádáme-li algoritmus tak, jak je znázorněno na obr. 4.1, pak pro každým řádkem matice A, který potřebujeme, zapříčiníme 16 výpadků stránky. Protože takto počítáme 1024×1024 prvků matice C, pak dostáváme přibližně 16 miliónů výpadků stránky, z nichž každý trvá 0.5 s. Celkový čas spotřebovaný pro vstup a výstup je tedy minimálně 93 dní. Vnitřní cyklus algoritmu na obr. 4.1 odpovídá skalárnímu součinu vektorů.

- Druhá možnost je algoritmus sloupcové aktualizace. Předpokládáme stejné uložení dat jako u výše uvedeného příkladu. Provedení algoritmu je znázorněno na obr. 4.2.

```
for j=1 to q
   for k=1 to q
     for i=1 to q
       c(i,j)=c(i,j) + a(i,k)*b(k,j)
     end k
   end j
end i.
```

Obr. 4.2: Násobení matic algoritmem sloupcové aktualizace

Vnitřní cyklus nyní odpovídá přičtení násobku sloupce k jinému sloupci. Pro každou hodnotu indexu j nyní potřebujeme přístup ke všem sloupcům matice A, což vede k 1024×16 výpadkům stránky. Matice B a C způsobí 16 výpadků stránky každá. Celkový čas spotřebovaný pro vstup a výstup zaviněný jen těmito výpadky stránek je tedy aspoň 2.25 hodin. Přestože toto je výrazné zlepšení proti prvnímu případu, stále je možné ještě zmenšit čas potřebný pro vstup a výstup.

- Předpokládejme, že matice jsou rozděleny na bloky o velikosti 256×256. Nechť každý blok je umístěn na samostatné stránce. Algoritmus, který využívá toto blokování, je znázorněn na obr. 4.3. Konstanta *nb* je rovna 256.

```
for j=1 to q step nb
   for k=1 to q step nb
     for jj=j to j+nb-1
       for kk=k to k+nb-1
         c(:,jj)=c(:,jj) + a(:,kk)*b(kk,jj)
       end kk
     end jj
   end k
end j.
```

Obr. 4.2: Násobení matic algoritmem sloupcové aktualizace

Počet výpadků stránky je velmi výrazně redukován. Celkové schéma způsobí pouze 96 výpadků stránky odpovídající přibližně 50 s času pro vstup a výstup.

Uvedené tři příklady jsou skrovným výběrem z více možností, jak uspořádat algoritmus násobení matic efektivně vzhledem k času spotřebovaném pro vstupní a výstupní operace. Rozdíly v tomto čase mohou být velmi drastické. Další podněty je možné najít např. v [8] a [10]. Z uvedeného vyplývají minimálně následující závěry, chceme-li vytvořit program, který se bude rutinně používat a u jehož používání bude záležet na jeho rychlosti.

- Je zapotřebí přihlížet k způsobu uložení matice (řádkové - sloupcové).
- U programů, které mají data uložená ve vnější paměti je záhodno znát velikost stránky.
- Na základě uvedeného se snažit o co největší lokalitu dat při prováděných operacích. Tomu odpovídá blokování úlohy.
- Všimněme si, že to co zde bylo řečeno pro vztah operační paměť - vnější paměť platí i pro vztah vyrovnávací paměť (cache) - operační paměť. Se znalostí základních principů můžeme dosáhnout poměrně velkých úspor času.

Mnoho moderních počítačů může provádět aritmetické operace tak rychle, že čas k jejich vykonání je srovnatelný s časem potřebným k načtení dat z paměťové struktury a k uložení výsledků. Je tedy jasné, že spíše než načtení každého údaje z paměti, jeho zpracování a uložení zpět je výhodnější snažit se vykonat na načtené části dat co nejvíce aritmetických operací, aby komunikace byla co nejmenší. Nejefektivnějším ze tří uvedených příkladů se jevil případ, kdy celé násobení bylo prováděno po blocích.

Uveďme ještě formální výsledky týkající se porovnání všech tří způsobů násobení. Označíme-li g poměr počtu vykonaných operací v plovoucí řádové čárce a počtu odkazů k paměti, M velikost vyrovnávací paměti a předpokládáme-li, že se celá úloha (tj. všechny tři používané matice) vejdou do operační paměti, pak máme v prvním případě $g = 2$, v druhém algoritmu $g \simeq M/q$ a v třetím algoritmu $g \simeq \sqrt{M/3}$. Čím vyšší tedy bude poměr g, tím bude algoritmus výhodnější při problémech s časovou náročností přenosu dat.

5 Přenos dat v algoritmech Gaussovy eliminace a Choleského dekompozice hustých matic

V Gaussově eliminaci či Choleského faktorizaci je při vektorovém pohledu na operace nejběžnější aritmetickou operací přičítání skalárního násobku jednoho vektoru k druhému vektoru. Tyto vektory mohou být buď řádky či sloupce matice, která je upravována v průběhu algoritmu. Máme-li k dispozici funkční jednotky, které mohou být zřetězeny, pak tyto vektorové operace mohou být velmi rychlé. K prvnímu porovnání se soustřeďme na algoritmus Choleského faktorizace, na kterém vyniknou všechny podstatné rysy výkladu.

Na obr. 5.1 je znázorněno schéma Choleského faktorizace $A = LL^T$ s aktualizací ve formě vnějších součinů. Nechť $A, L \in R^{q,q}$.

```
for k = 1 to q
    a_kk = √a_kk
    for i = k+1 to q
        a_ik = a_ik/a_kk
    end i
    for j = k+1 to q
        for i = j to n
            a_ij = a_ij − a_ik a_jk
        end i
    end j
end k
```

Obr. 5.1: Schéma Choleského faktorizace s aktualizací ve formě vnějších součinů.

V každém kroku k tohoto schématu přičítáme ve dvou vnitřních cyklech ke každému z dosud nezpracovaných sloupců násobek $k-tého$ sloupce. Nechť $a^{(j)}$ označuje vektor sestávající se ze poddiagonální části $j-tého$ sloupce. Nechť pro $k < j$ označuje $\hat{a}^{(k)}$ podvektor $a^{(k)}$ který získáme z $a^{(k)}$ odstraněním tolika komponent s nejmenšími řádkovými indexy, aby měl stejný počet komponent jako $a^{(j)}$. Pak mohou být operace v nejvnitřnějším cyklu schématu z obrázku 5.1 popsány vektorovou operací $a^{(j)} = a^{(j)} - a_{jk}\hat{a}^{(k)}$. Schéma pak je znázorněno na obrázku 5.2.

```
for k = 1 to q
    a_kk = √a_kk
    a^(k) = a_kk^-1 a^(k)
```

```
    for j = k+1 to q
      a^(j) = a^(j) − a_jk â^(k)
    end j
  end k
```

Obr. 5.2: Schéma Choleského faktorizace s aktualizací ve formě vnějších součinů ve vektorové formě.

V algoritmu jsou použity vektorové operace, které však mohou být, umožňuje-li to architektura počítače, efektivně zřetězené. Nároky na komunikaci jsou však přílišné. V každém kroku totiž načítáme všechny zbývající sloupce s indexy $j = k+1, \ldots, n$, jednou modifikujeme a opět ukládáme. Na počítači s vektorovými registry, ve kterých můžeme uchovávat sloupce nebo jejich značné části může být datová komunikace značně omezena. Přeformulovaný algoritmus je znázorněn na obrázku 5.3.

```
for j = 1 to q
  for k = 1 to j-1
    a^(j) = a^(j) − a_jk â^(k)
  end k
  a_jj = √a_jj
  a^(j) = a_jj^-1 a^(j)
end j
```

Obr. 5.3: Schéma jk Choleského faktorizace ve vektorové formě

V algoritmu podle obrázku 5.3 nám odpadá v každém kroku komunikace spojena s ukládáním sloupců a reálný výkon počítače na uvedené úloze se může značně zvýšit. V případě, že sloupce matice jsou dlouhé a nevejdou se celé do vektorových registrů, musíme sloupce ještě segmentovat a celý problém bude mít ještě vnitřní blokovou strukturu.

Představme si obecnější algoritmu, který provádí Gaussovu eliminaci. Obecně můžeme popsat základní schéma řešení problému schématem znázorněným na obrázku 5.4.

```
for .....
  .....
  for .....
    .....
    for .....
      .....
      a_ij = a_ij − a_ik a_kj / a_kk
    end
  end
end.
```

Obr. 5.4: Generické schéma algoritmu Gaussovy eliminace

K jednotlivým naznačeným cyklům můžeme připojit indexy i, j, k, příslušné meze cyklů a dostáváme celkem 6 možností algoritmu. To, co jsme ukázali výše pro speciální případ, byly vektorové formulace dvou možností u kterých jsme si všímali velikost komunikace dat. V závislosti na uložení dat si můžeme zvolit výhodný tvar Gaussovy eliminace. Operace, které byly ve vnitřních cyklech se daly výhodně vektorizovat a data jsme mohli načítat po vektorech.

Totéž můžeme provést pro úlohu Choleského faktorizace při rozkladu matic symetrických a positivně definitních. Další možností přeformulování rozkladů znázorníme opět pro názornost na tomto případě.

Rozepíšeme v k – *tém* kroku rozklad následujícím způsobem.

$$\begin{pmatrix} A_{11} & a_k^T & A_{13} \\ a_k & a_{kk} & a_k^T \\ A_{31} & a_k & A_{33} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 & 0 \\ l_k & l_{kk} & 0 \\ L_{31} & \lambda_k & L_{33} \end{pmatrix} \begin{pmatrix} L_{11}^T & l_k^T & L_{13}^T \\ 0 & l_{kk} & \lambda_k^T \\ 0 & 0 & L_{33}^T \end{pmatrix}$$

Na základě znalosti L_{11} a matice A získáme v k – *tém* kroku vektor l_k a prvek l_{kk} tvořící k – *tý* řádek faktoru L. Tyto hodnoty spočítáme ze vztahů

$$L_{11} l_k^T = a_k^T$$

a

$$l_{kk}^2 = a_{kk} - l_k l_k^T.$$

Tento způsob Choleského faktorizace bychom mohli opět rozepsat v několika variantách podle pořadí cyklů rozepsaného algoritmu. V uvedeném příkladě počítáme matici L po sloupcích. Obdobně můžeme formulovat tedy algoritmus, který počítá matici L po řádcích. Výrazný rozdíl proti předcházejícím šesti variantám je v tom, že jádro k – *tého* kroku je zde vyjádřeno pomocí operace matice - vektor. Tyto operace se dají na vektorových architekturách mnohem lépe optimalizovat. V závěru této kapitoly ukážeme přehled vztahu mezi počtem numerických operací a velikostí komunikace mezi registry a pamětí počítače. Tyto varianty algoritmu využívající operace typu matice - vektor jsou mnohem efektivnější na vektorových počítačích a je možné s nimi dosáhnout často téměř špičkového výkonu počítače.

Následující možnost Choleského faktorizace je ještě obecnější, neboť využívá úplného rozdělení matic na bloky. Abychom se vyhnuli nadbytečnému formalismu, situaci znázorníme na jednodušší případě matic o devíti blocích. Uvažujeme-li následující schéma

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & L_{33} \end{pmatrix} \begin{pmatrix} L_{11}^T & L_{21}^T & L_{31}^T \\ 0 & L_{22}^T & L_{32}^T \\ 0 & 0 & L_{33}^T \end{pmatrix}$$

Máme-li spočítáno L_{11} a známe-li matici A, pak můžeme spočítat L_{21} ze vztahu

$$L_{11} L_{21}^T = A_{12}$$

a poté ze vztahu

$$L_{21} L_{21}^T = A_{22} - L_{21} L_{21}^T$$

spočítáme L_{22}.

Operace použité v tomto způsobu faktorizace jsou operace typu matice - matice a jejich použití dále obecně zefektivňuje Choleského faktorizaci při používání vektorových počítačů.

Základní operace s vektory a maticemi jsou součástí tří knihoven BLAS (Basic Linear Algebra Subroutines) : BLAS1, BLAS2 a BLAS3, které jsou efektivně implementovány na všech důležitých počítačových architekturách používaných pro vědecko-technické výpočty.

Knihovna BLAS1 obsahuje základní typy vektorových operací jako jsou přičítání skalárního násobku vektoru k jinému vektoru, násobení vektoru skalárem, skalární součin vektorů, výpočet norem vektoru, generování a aplikace Givensových rotací, ...

Knihovna BLAS2 zahrnuje operace typu matice - vektor jako přičítání lineární kombinace součinu matice a vektoru a skalárního násobku vektoru k jinému vektoru, násobení vektoru trojúhelníkovou maticí, řešení systému s trojúhelníkovou maticí, aktualizace matice maticemi hodnosti 1 či 2, ...

Knihovna BLAS3 je věnována operacím typu matice - matice a obsahuje přičítaní lineární kombinace součinu matic a násobku matice a vektoru k nějaké další matici, násobení matice trojúhelníkovou maticí, řešení systému s trojúhelníkovou maticí, obecné aktualizace matice maticí hodnosti k, ...

Nechť $A, B, C \in R^{q,q}, y \in R^q, \alpha, \beta \in R$. Porovnejme počet operací v pohyblivé řádové čárce a počet načtených a uložených prvků v průběhu výpočtů tří základních vektorových operací.

Pro operaci $y = y + \alpha x$ z BLASu1 je počet operací v pohyblivé řádové čárce $2q$ a počet načtených a uložených prvků $3q$. Pro operaci BLASu2 $y = \beta y + \alpha Ax$ je počet operací v pohyblivé řádové čárce $2q^2$ a počet prvků načtených a uložených při této operaci je $q^2 + 3q$. Pro operaci BLASu3 $C = \beta C + \alpha AB$ je počet operací v pohyblivé řádové čárce $2q^3$ a počet prvků načtených a uložených při této operaci je $4q^2$.

Zatímco u operace z BLASu1 je poměr počtu paměťových referencí ku počtu operací v pohyblivé řádové čárce roven 3:2 je tento poměr u operace z BLASu2 roven 1:2 a u operace z BLASu3 dokonce 2:q. To je důvod, proč operace odpovídající BLASu3 jsou maximálně výhodné, a to nejen na superpočítačových architekturách, ale, vzhledem k hierarchizaci pamětí, i na méně výkonné výpočetní technice. Na velké většině moderních architektur jsou tyto knihovny dostupné a optimalizované přímo vzhledem k architektuře - počtu registrů, vyrovnávací paměti, ...

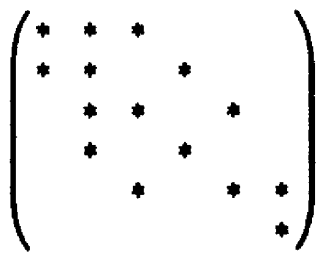
Algoritmus Gaussovy eliminace může být zpracován blokovými algoritmy úplně analogicky.

Z uvedeného je patrné, že pokud chceme počítat Gaussovu eliminaci či Choleského faktorizaci efektivně, je nutné zamyslet se nad přesuny dat a lokalitou operací algoritmu. Pokud nemáme speciální požadavky je velmi výhodné využít velkého množství úsilí specialistů, kteří vyvinuli řadu užitečných procedur pomáhající počítat tyto úlohy lineární algebry: BLAS 1 - 3. Tyto procedury jsou použity v řadě špičkových souborů pro řešení jednodušších i komplikovanějších úloh. Z souborů, které řeší systémy lineárních rovnic s hustou maticí soustavy a přihlížejí k architektuře počítačů jmenujme na prvním místě systém LAPACK (viz [2]).

6 Frontální a multifrontální metody řešení soustav s řídkou maticí soustavy

V případě, že matice soustavy je řídká, to jest, má mnoho nenulových prvků, jejichž struktury můžeme využít, používané algoritmy pro Gaussovu eliminaci či Choleského faktorizaci nejsou tak přímočaré (viz [1], [13]).

Jednou z možností je převést matici pomocí permutací na tvar, ve kterém bude tento algoritmus jednoduchý a podobný zpracování hustých matic, přičemž operace s většinou nulových prvků nebudeme vůbec provádět. Typickým příkladem jsou metody, které upraví matici tak, aby měla nenulové prvky pouze v oblasti "okolo" diagonály - metody profilové a pásové. Matice na obrázku 6.1 je typickým příkladem pásové matice. Nenulové prvky jsou znázorněny hvězdičkami.



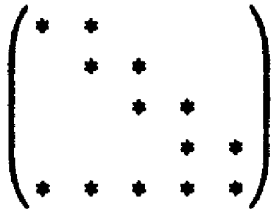
Obr. 6.1: Matice s pásem nenulových prvků v oblasti diagonály.

V obou zmíněných algoritmech používáme pouze části řádků resp. sloupců v oblasti diagonály. Tyto části jsou přímo adresované, a tak jejich vektorizace je zajištěna.

První problém je délka použitých řádků či sloupců v algoritmu. Ta může být dost malá a víme, že efektivnost závisí velmi na délce vektorů v aritmetických operacích. Vektory mohou být tak krátké, že není příliš velký rozdíl mezi výkonností počítače ve skalárním a vektorovém módu pro vykonávané operace. Vzhledem k tomu, že pás v matici je idealizací, a že provádíme operace nad některými

nulovými prvky, pak se může ukázat, že jiné algoritmy mohou být efektivnější i přes možnou nepřímou adresaci hodnot matice (viz [17]).

Schéma pásové matice můžeme trochu zjemnit tak, že pro každý řádek uchováváme informaci o tom, od kterého sloupcového indexu začíná uložení jeho nenulových prvků. Uvažujme matici z obrázku 6.2.



Obr. 6.2: Příklad matice na ilustraci profilového schématu.

Je zřejmé, že pro tuto matici je nevýhodné uchovávat šířku pásu matice, což je číslo charakterizující šířku diagonální oblasti s nenulovými prvky. Poslední řádek matice totiž narušuje uložení nenulových prvků v okolí diagonály. Uchováváme-li pro každý řádek "délku" pásu jeho nenulových prvků a poté pouze ty prvky, které se nacházejí v tomto pásu, pak můžeme matici efektivně uchovávat i v tomto případě. Toto schéma uchovávání matic se nazývá *profilové schéma*.

Důležitý fakt, pro který jsou pásová a profilová schémata uložení oblíbena, je ten, že při Gaussově eliminaci či Choleského faktorizaci jsou nenulové prvky faktorů umístěny opět v oblasti diagonály. Precizní znění tohoto tvrzení, závislé na precizních definicích pásu a profilu, nalezne čtenář v [13].

Možnost uchovávat profil matice vede k dalším úsporám paměti i času na řešení našeho problému, ale z hlediska úvahy nad vztahem architektury a algoritmu nepřináší nic nového.

V případě, že nenulové prvky matice jsou soustředěny v několika hlavních a vedlejších diagonálách matice, je možné přeformulovat uvedené algoritmy tak, aby používaly pouze prvky v těchto diagonálách a prvky nově vzniklé a algoritmus je vektorizovatelný s vektory délky $O(n)$.

Princip lokality výpočtů a minimalizace přesunů vede, pokud je matice rozsáhlejší, k blokovému provádění operací. Mnohé významné matice vznikající v statistických problémech mají blokově pásový či blokově diagonální tvar (viz [6] a [7]).

Gaussova eliminace a Choleského faktorizace vedou v případě matice obecně řídké k následujícímu problému, zmíněnému výše. Z používaných matic zachycujeme pouze strukturu a hodnoty nenulových prvků. Pro matici z obrázku 6.1 máme např. v třetím řádku nenulové pouze elementy se sloupcovými indexy 2, 3 a 5. Abychom pracovali pouze s nenulovými prvky třetího řádku, musíme použít pole s těmito sloupcovými indexy a přes jejich hodnoty provést příslušné operace. Elementy tohoto řádku máme tedy nepřímo adresované.

Na vektorovém počítači je tento postup efektivní pouze tehdy, je-li příslušná architektura vybavena technickými možnostmi pro efektivní práci s uvedenými poli. To je naštěstí případ většiny soudobých moderních počítačů pro vědecko-technické výpočty. O vlastních algoritmech zde nebudeme hovořit pro spousty neopominutelných detailů nutných pro efektivní implementaci.

Zaměříme nyní svoji pozornost na další perspektivní směr, jehož vývoj byl ovlivněn vývojem počítačových architektur - na metody frontální a multifrontální.

Frontální metody vznikly původně pro řešení úloh strukturální analýzy metodou konečných prvků. Jeden z prvních programů byl napsán pouze pro případ matic symetrických a positivně definitních (viz [16]). Pozdější rozšíření zahrnuje i jiné aplikace i algoritmy pro nesymetrické matice.

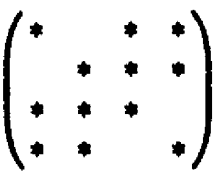
Způsob řešení spočívá v následujícím. Výpočet probíhá v husté matici dimenze $b \times c$, kde veličiny b, c specifikujeme později. Uvažujme matici z obrázku 6.1, kde $b = c = 3$. Předpokládejme zpracování řádků a sloupců ve znázorněném pořadí. V prvním kroku přesuneme do husté pracovní matice b prvních řádků a c prvních sloupců. K provedení prvního kroku našich algoritmů nám stačí tato podmatice. Máme v ní všechny elementy, které se mění. Po prvním kroku přehrajeme první řádek a sloupec faktorů do výsledné struktury a do uvolněného místa v pracovní matici nahrajeme další

řádek a sloupec. Celý postup si můžeme tak představit jako pohyb "okna" v rozkládané matici po diagonále směrem k pravému dolnímu rohu matice. Velikost pracovní matice musíme zvolit tak, abychom v každém kroku měli v ní všechny prvky, které jsou v tomto kroku upravovány. Pro aplikaci této metody není nutné, aby matice byla převedena permutacemi na pásový tvar. Permutaci můžeme určovat přímo v průběhu algoritmu, což je důležité především v případě nesymetrických matic, kde je nutná *pivotace* (tj. výběr hlavního prvku, který budeme také nazývat *pivot*, aby rozklad v Gaussově eliminaci byl stabilní.

V případě, že provádíme výběr pivota v pracovní matici, t.j. provádíme dodatečnou permutaci řádků a sloupců, je obvykle nutné mít frontální matici dostatečně velkou, abychom měli dostatečný výběr kandidátů pivotace.

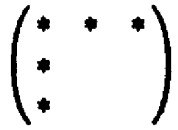
Důmyslnější varianta tohoto postupu se stala jedním z nejúspěšnějších postupů řešení především symetrických indefinitních systémů. Metoda se snaží využít výhody frontálního postupu a zmenšit množství aritmetických operací. Opět jako v případě frontální metody se budeme snažit osvětlit její výhody na příkladě. Pro detailnější výklad odkazujeme k [11].

Předpokládejme nejprve, že matice, kterou chceme rozkládat je symetrická (viz obr. 6.3).



Obr 6.3: Matice k ilustraci multifrontálního schématu.

Řádky a sloupce pivota (hlavního prvku) jsou vybírány z diagonálních prvků v přirozeném pořadí. V prvním kroku můžeme provést eliminaci odpovídající pivotu v pozici (1,1) nejprve "soustředěním" (assembly) nenulových prvků prvního řádku a sloupce tak, že dostaneme matici znázorněnou na obrázku 6.4. Tímto soustředěním rozumíme umístění nenulových prvků prvního řádku a sloupce do podmatice řádu daném počtem nenulových prvků prvního řádku.



Obr. 6.4: Struktura podmatice vzniklé soustředěním prvního řádku a sloupce.

Nulové prvky $a_{12} = a_{21}$ tak způsobily vynechání řádku a sloupce číslo 2 v tomto soustředění. Poté provedeme první krok eliminace, zpracovaný první řádek a sloupec jsou přemístěny do výsledné struktury pro ukládání faktorů L a U (resp. Choleského faktoru L). Výsledná redukovaná matice odpovídající řádkovým (a sloupcovým) indexům 3 a 4 má být upravena pomocí elementární Gaussovy transformace tvaru

$$a_{ij} = a_{ij} - a_{i1}a_{1j}/a_{11}$$

pro všechna (i,j) taková, že $a_{i1}a_{1j} \neq 0$. V konvenční Gaussově eliminaci jsou tyto operace ihned vykonány. V multifrontální formulaci jsou uchovávány pouze produkty

$$a_{i1}a_{1j}/a_{11}$$

a odpovídající úpravy prvků nejsou vykonávány okamžitě. Úprava určitého prvku musí být však vykonána do okamžiku, kdy je prvek potřebný v nějakém sloupci či řádku pivota.

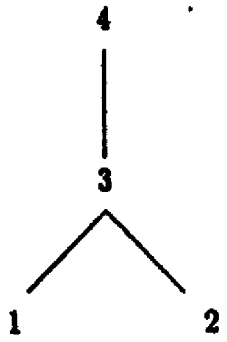
Poté jsou soustředěny prvky druhého řádku a sloupce, prvek (2,2) je v tomto druhém kroku pivot. Získáme druhý řádek a sloupec faktorů a uchováme výrazy

$$a_{i2}a_{2j}/a_{22}.$$

Opět není nutné provádět okamžitě úpravy příslušných elementů v matici složené z třetího a čtvrtého řádku a sloupce. Podmatic s uvedenými produkty tedy může být v jednom okamžiku více a uchováváme je do té doby, dokud jsou potřeba pro úpravu nějakých dalších prvků.

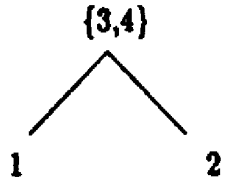
Předtím, než můžeme použít k dalším operacím prvek (3,3) jako pivot, musíme provést na třetím řádku a sloupci úpravy vyplývající z prvních dvou kroků. Poté ve čtvrtém kroku provedeme všechny úpravy na prvku (4,4) původní matice, abychom upravený prvek zařadili do struktury faktorů.

Posloupnost kroků faktorizace je znázorněna stromem na obrázku 6.5. Kroky musíme vykonávat tak, abychom pro libovolný jeho vrchol měli nejprve provedeny kroky odpovídající jeho synům (tj. vrcholům umístěným v podstromu pod tímto vrcholem). Viděli jsme, že struktura závislosti v naší matici odpovídá tomuto stromu. Větším problémem je nalezení tohoto stromu a zájemce odkazujeme k [19].



Obr. 6.5: Závislost kroků faktorizace znázorněná eliminačním stromem.

Tatáž paměť a totéž množství aritmetických operací je použito, jestliže řádky a sloupce pro třetí a čtvrtý krok jsou soustředěny najednou a poté jsou provedeny dva kroky na téže matici. Tato procedura odpovídá sloučením vrcholů 3 a 4 v stromu z obrázku 6.5. Dostaneme tak situaci znázorněnou na obrázku 6.6.



Obr. 6.6: Eliminační strom faktorizace po sloučení vrcholů 3 a 4.

V typické úloze se snažíme zvýšit efektivitu algoritmu tím, že hledáme skupiny vrcholů, které můžeme eliminovat společně. Tím zvyšujeme velikost používaných pracovních matic a nalézáme jakousi rovnováhu mezi dvěma protichůdnými faktory:

- Snahou o to, aby byly co nejhustší, což se dosáhne jejich zmenšením a
- snahou o zvýšení délky používaných vektorů, což se dosahuje vytvářením větších matic soustředěním řádků a sloupců i od více následujících pivotů.

Uvedený postup je efektivní především pro řešení symetrických indefinitních lineárních systémů. Pro systémy se symetrickou positivně definitní maticí jsou k dispozici ještě efektivnější algoritmy.

Pro nesymetrické matice se dá uvedený algoritmus použít, ale vzhledem k tomu, že používaná struktura je symetrizací struktury matice řešeného systému, není často tak efektivní jako některé jiné algoritmy. V tomto případě je obvykle nutná pivotace.

Frontální postup možno chápat jako jakési zobecnění pásových řešičů, kde operace jsou prováděny poněkud odlišným způsobem. Z toho plynou i stejné poznámky k jeho používání na moderních počítačových architekturách. Vývoj multifrontálního algoritmu byl ovlivněn z velké části podněty

z počítačových architektur. Více pracovních matic umožňuje souběžnou práci na více částech matice zároveň a to se dá efektivně využít v paralelismu, který zde nesledujeme - v multiprocesorovém paralelismu. Zpracování těchto částí v hustých pracovních maticích umožňuje přímou vektorizaci kroků algoritmu. Je výhodné, dokážeme-li veškeré pracovní matice umístit ve vyrovnávací paměti.

7 Metody ortogonálního rozkladu

Jak jsme již uvedli, efektivní provádění ortogonálních rozkladů je klíčem k řešení lineárního problému nejmenších čtverců vznikajícího například v úloze lineární regrese.

Stručně shrňme výhody a nevýhody základních tří metod ve vztahu k úloze a počítačové architektuře.

QR rozklad matice A metodou Givensových rotací (viz [15]) spočívá v jejím převedení na trojúhelníkovou matici R postupným vynulováním poddiagonálních prvků matice A. Prvky jsou přitom nulovány jeden po druhém. Jednotlivé kroky mohou být interpretovány jako násobení matice A zleva maticí rotace v určité rovině prostoru R^m. Při této operaci nulování jednotlivého prvku dochází ke změnám v rozkládané matici, ale pořadí jednotlivých ortogonálních transformací se dá určit tak, že každá pozice se nuluje jen jednou. Nevýhodou je velké množství nenumerických manipulací a také větší počet operací ve srovnání s jinými metodami. Úlohu lze vektorizovat, metody vektorizace však nejsou příliš přímočaré a efektivitou zaostávají u velké části problému za efektivní vektorovou implementací jiných metod. Výhoda tohoto postupu je větší selektivita nulovaných pozic. Situaci si přibližme na obrázku 6.3. Chceme-li eliminovat element (3,1), pak pro eliminaci použijeme Givensovu rotaci určenou řádkovými indexy 1 a 3. Struktura matice této Givensovy rotace je následující:

$$\begin{pmatrix} * & & * & \\ & 1 & & \\ * & & * & \\ & & & 1 \end{pmatrix}$$

Obr 7.1: Struktura matice Givensovy rotace určené k nulování prvku (3,1) v matici z obrázku 6.3.

Po jejím provedení je tedy výsledná struktura řádků 1 a 3 s výjimkou eliminovaného prvku rovna sjednocení struktur těchto řádků. Nevýhoda velkého množství manipulací je zřejmá, uvědomíme-li si, že pro každé vynulování potřebujeme načítat dva řádkové vektory.

Druhou možností, která nás bude hlavně zajímat, je použít metodu Householderova zrcadlení (viz [15]). Základním rysem této metody je souběžné nulování více poddiagonálních prvků částí sloupců tak, abychom získali trojúhelníkový faktor R (Přesněji, hovoříme pouze o základním způsobu použití Householderových zrcadlení, kdy pivot je vybírán z naddiagonálních prvků rozkládané matice). Jednotlivé ortogonální transformace mají tvar

$$I - \alpha u\, u^T,$$

kde $\alpha u^T u = 2$. Vektor u je přitom konstruován z části sloupce v níž jsou všechny elementy kromě prvního nulovány.

Formálně můžeme pro Householderovo zrcadlení napsat:

$$A = Q_k Q_{k-1} \dots Q_1 \begin{pmatrix} R \\ 0 \end{pmatrix}$$

pro nějaké přirozené k, kde matice $Q_k Q_{k-1} \dots Q_1$ jsou ortogonální transformace výše uvedeného tvaru. V algoritmu Givensových rotací jsme mohli rozklad vyjádřit obdobně. Použitých ortogonálních transformací by však bylo více.

V této metodě je datových manipulací výrazně méně, což indukuje lepší možnosti vektorizace. Ukažme nyní jeden způsob, nedávno zavedený, který umožňuje ještě zlepšit vyhlídky na vektorizaci (viz [24]). Tento způsob se nazývá též kompaktní schéma QR rozkladu.

Produkt k Householderových zrcadlení může být zapsán ve tvaru jako součin matic následujícím způsobem:

$$\Pi(I - \alpha_j u_j u_j^T) = I - YTT^T,$$

kde $Y = (u_1, \ldots, u_k)$ a T je horní trojúhelníková matice odpovídajících dimenzí. Tento fakt může být snadno ověřen matematickou indukcí: Máme-li $V = I - YTT^T$, pak

$$V(I - \alpha u\, u^T) = I - (Y \quad u) \begin{pmatrix} T & h \\ 0 & \alpha \end{pmatrix} (Y \quad u)^T,$$

kde $h = \alpha TW^T u$.

První výhoda uchovávání součinu matic ortogonálních matic prostřednictvím Y a T je menší množství použité paměti. Vytváření těchto matic je navíc bohaté na operace typu matice - vektor. To jsou však operace, které jsou zahrnuté v knihovně BLAS2, a tedy velmi dobře vektorizovatelné.

Celý algoritmus QR rozkladu můžeme dále aplikovat blokovým způsobem a dosáhneme toho, že velká část operací bude náležet dokonce BLASu3. Proces si znázorníme pro přehlednost na matici rozdělenou na dva sloupcové bloky.

Nejprve rozložíme matici odpovídající první blokové části pomocí Householderových zrcadlení. Z použitých ortogonálních transformací spočítáme odpovídající matice Y a T. Aplikujeme-li tento výsledek na matici A, pak dostáváme:

$$A = (I - YTY^T) \begin{pmatrix} R_k & S_k^T \\ 0 & A_k \end{pmatrix},$$

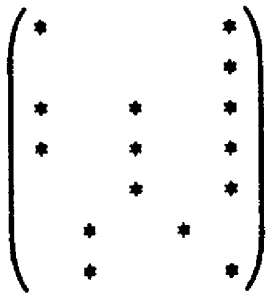
kde R_k je již vypočtená část horní trojúhelníkové matice. Sloupcový blok $\begin{pmatrix} S_k^T \\ A_k \end{pmatrix}$ je vypočten pomocí operací z knihovny BLAS3. Algoritmus je tedy vektorizován s vysokou účinností a blokové uspořádání též umožňuje dobrou lokality jednotlivých blokových kroků. Počet sloupcových bloků, na který rozdělíme matici, tedy zvolíme tak, aby co nejlépe vystihoval hierarchii a přesuny dat v našem paměťovém systému.

Rozepíšeme-li algoritmus Householderova algoritmu prostřednictvím cyklů se zaměnitelným pořadím, pak opět přicházíme k problému komunikace probíraném výše. Vzhledem ke komplikovanějším operacím jsou možná pouze tři uspořádání, lišící se efektivitou. Zájemce odkazujeme k [23].

Třetí základní způsob rozkladu - modifikovaný Gramm-Schmidtův algoritmus (viz [3], [5]) není tak často používán *pro tuto úlohu* především proto, že ortogonální matice spočtená tímto způsobem trpí numerickou ztrátou ortogonality výrazně více než v předchozím případě a tím je ovlivněna kvalita či řešení algoritmem v blokové formulaci. Jeho výhodou je ještě větší průzračnost algoritmu. Vektorové operace prováděné v jeho průběhu pracují nad vektory stejné délky m.

Obraťme nyní pozornost k QR rozkladu v případě, že rozkládaná matice je řídká. Jak jsme se zmínili, zde budeme mnohem více na vážkách, zda použít metodu Givensových rotací, neboť pak můžeme nulovat poddiagonální elementy mnohem specifičtěji s menším zaplněním matice v průběhu výpočtu. Vzhledem ke komplikovanosti datových struktur v případě řídké matice je metoda Givensových rotací dnes často používána (viz [12]) a je implementována ve známé knihovně SPARSPAK-B. Metody, které používají Householderovo zrcadlení pro řídké matice, jsou v dnešní době intenzívně rozvíjeny (viz [14]). Obraťme pozornost na variantu algoritmu podobnou multifrontálnímu postupu z kapitoly 5.

Mějme dánu matici A se strukturou nenulových prvků jako na obrázku 7.2. Zpracovávejme ji pomocí sekvence Householderových zrcadlení v pěti krocích, z nichž každý odpovídá jednomu sloupci matice, na jehož základě se spočítá příslušná ortogonální transformace.



Obr. 7.2: Struktura matice rozkládané ortogonálními transformacemi.

V prvním kroku spočítáme transformaci na základě prvků prvního sloupce. Ovlivněny jsou pouze prvky v řádcích, které mají nenulový element v prvním sloupci, tj. prvky v řádcích 1, 3, 4. V druhém kroku upravujeme výslednou matici po uložení prvního řádku faktoru R, který byl spočítán v prvním kroku. Matice se kterou pracujeme v druhém kroku má tedy sloupcovou i řádkovou dimenzi o jedničku menší. To je onen výše inzerovaný rozdíl proti modifikované Gramm-Schmidtově metodě. Všimněme si následujících detailů: Vzhledem k tomu, že v každém sloupci rozkládané matice může být jen několik nenulových prvků, je možná následující varianta. Elementy zpracovávané v určitém kroku jsou načteny do husté pracovní matice a zde zpracovány. V prvním kroku potřebujeme jen matici dimenze 3×3. Druhý postřeh je následující. Pro provedení druhého kroku rozkladu nepotřebujeme žádný z prvků upravených v prvním kroku. Data získaná v prvním kroku tedy mohou být uložena někde v pomocné datové struktuře a použita v nějakém dalším kroku. Toto nás staví před následující problémy intenzívně řešené v současné době:

1. Nalezení struktury závislosti dat používaných v jednotlivých krocích rozkladu a
2. nalezení efektivního uložení částečně zpracovaných dat pro algoritmus rozkladu.

Struktura závislosti dat, jak je dnes používána, odpovídá eliminačnímu stromu zmíněnému v multifrontálních metodách vytvořeném pro matici A^TA (viz [19]). Pro uložení částečně zpracovaných dat byla nalezena efektivní struktura (viz [18], [20]), která je také dále zdokonalována. Přehled tohoto, dnes asi nejperspektivnějšího, schématu pro ortogonální faktorizaci řídkých matic je uveden v [22].

Vývoj tohoto algoritmu byl ovlivněn následujícími koncepty:

- Přechod k Householderově metodě zrcadlení. Tato metoda má pro husté matice teoreticky malý počet operací. Přestože z důvodu velkého zaplnění (velký počet nově vzniklých nenulových prvků v pracovních strukturách) může celkový počet operací převýšit množství operací nutné při použití Givensových rotací, důležitý je též malý počet datových přesunů ve srovnání s metodou rotací.
- Možnost lepší vektorizace algoritmu založeného na Householderově metodě zrcadlení z důvodů přímého adresování elementů ve vnitřních cyklech. Uchovávání meziproduktů v přímo adresovaných strukturách. Vzhledem k složitosti uschovávání meziproduktů v průběhu algoritmu je však přímá adresace používána často i v metodě Givensových rotací.
- Možnost použít operace BLAS3.

Spojení uvedených konceptů s rozdělením matice na bloky je dnes základ solidního systému pro QR rozklad, a tím i pro řešení lineárního problému nejmenších čtverců v případě rozsáhlé řídké matice soustavy.

Literatura

[1] A.V.Aho, J.E.Hopcroft, J.D.Ullman: Data Structures and Algorithms, Addison-Wesley, Reading, MA, 1983.

[2] E.Anderson, Z.Bai, C.Bischof, J.Demmel, J.J.Dongarra, J.Du Croz, A.Greenbaum, S.Hammarling, A.McKenney, S.Ostrouchov, D.Sorensen: LAPACK User's guide, SIAM, Philadelphia, 1992.

[3] A.Björck: Solving linear least squares by Gramm-Schmidt orthogonalization, BIT, 7 (1967), 1-21.

[4] A.Björck:Algorithms for Linear Least Squares Problems, presented at the conference "Computer algorithms for solving linear algebraic equations: the state of the art", Il Ciocco, Barga, Italy, September 1990.

[5] A. Björck, C.C.Paige: Loss and recapture of orthogonality in the modified Gramm-Schmidt algorithm, SIAM J. Matrix Anal. Appl. 13 (1992), 176-190.

[6] M.G.Cox: The least squares solution of overdetermined linear equations having band or augmented band structure, IMA J. Numer. Anal. 1 (1981), 3-22.

[7] M.G.Cox, P.E.Manneback: Least-squares spline regression with block-diagonal variance matrices, IMA J. Num. Anal. 5(1985), 275-286.

[8] J.Dongarra, F.Gustavson, A.Karp: Implementing linear algebra algorithms for dense matrices on a vector pipeline machine, SIAM Review, 26 (1984), 91-112.

[9] J.J.Dongarra, J.Du Croz, I.S.Duff, S.Hammarling: A set of level 3 basic linear algebra subprograms, ACM Trans. Math. Software, 16 (1990), 1-17.

[10] J.J.Dongarra, I.S.Duff, D.C.Sorensen: Solving Linear Systems on Vector and Shared Memory Computers, SIAM, Philadelphia, 1991.

[11] I.S.Duff, J.K.Reid: The multifrontal solution of indefinite sparse symmetric sets of linear equations, ACM Trans. Math. Softw., 9 (1983), 302-325.

[12] A. George, M.T.Heath: Solution of sparse linear least squares problems using Givens rotations, Linear Algebra and its Appl., 34 (1980), 69-83.

[13] A. George, J.W.H.Liu: Computer Solution of Large Sparse Positive Definite Systems, Prentice-Hall, Englewood Cliffs, N.J., 1981.

[14] A. George, J.W.H.Liu: Householder reflections versus Givens rotations in sparse orthogonal decomposition, Linear Algebra and its Appl., 88/89 (1987), 223-238.

[15] G.H. Golub, C. van Loan: Matrix Computations, Johns Hopkins University Press, Baltimore, MD, 2nd edition, 1989.

[16] B.M.Irons: A frontal solution program for finite-element analysis, Int. J.Num. Meth. Eng., 2 (1970), 5-32.

[17] J.G. Lewis, H.D.Simon: The impact of hardware gather/scatter on sparse Gaussian elimination, Supercomputing Forum, Boeing Computer Services, 1 (1986), 9-11.

[18] J.W.H. Liu: On general row merging schemes for sparse Givens transformations, SIAM J. Sci. Stat. Comput., 7 (1986),

[19] J.W.H.Liu: The role of elimination trees in sparse factorization, SIAM J. Matrix Anal. Appl. 11 (1990), 134-172.

[20] J.W.H.Liu: The multifrontal method for sparse matrix solution. Theory and practise, SIAM Review 34 (1992), 82-109.

[21] K.V.Mardia, J.T.Kent, J.M.Bibby: Multivariate Analysis, Academic Press, London, 1979.

[22] P.Matstoms: The multifrontal solution of sparse linear least squares problems, Licentiat thesis No. 293, University of Linköping, 1991.

[23] R.B. Mattingly, C.D.Meyer, J.M.Ortega: Orthogonal reduction on vector computers, SIAM J. Sci. Stat. Comput., 10 (1989), 372-381.

[24] R. Schreiber, C. van Loan: A storage efficient WY representation for products of Householder transformations, SIAM J. Sci. Stat. Comput., 10 (1989), 53-57.

[25] D.T.Winter: Efficient use of memory and input/output. In J.J. te Riele, T.J.Dekker, H.A. van der Vorst, eds., Algorithms and Applications on Vector and Parallel Computers, North-Holland, Amsterdam, 1987.