Pavel Praks

Téma diplomové práce: Iterační řešení soustav lineárních rovnic s několika pravými stranami

VŠB-TU Ostrava, Fakulta elektrotechniky a informatiky

Obor: Informatika a aplikovaná matematika

Vedoucí diplomové práce: Prof. RNDr. Zdeněk Dostál, DSc.

Ostrava, 1999

Obsah

Značení

R^n	*n*-rozměrný Euklidovský prostor
Kp-operace	násobení matice soustavy vektorem
M	počet soustav v master systému
S	počet soustav ve slave systému
N_S	počet nevyřešených (aktivních) soustav

Necht' $X = [x^1, \ldots, x^i, \ldots, x^q]$ je $n \times q$ matice, pak označíme:

x^i	*i*-tý sloupec matice *X*
X(:,i)	*i*-tý sloupec matice *X* (syntaxe Matlabu)
X^T	transponování matice *X*
X'	transponování matice *X* (syntaxe Matlabu)
X_k	*k*-tá aproximace *X*

Užité zkratky

CG	metoda sdružených gradientů (Conjugate Gradients Method)
PCG	metoda předpodmíněných sdružených gradientů (Preconditioned CG Method)
BCG	metoda blokových sdružených gradientů (Block CG Method)
SCG	metoda postupných sdružených gradientů (Successive CG Method)
SBCG	metoda postupných blokových sdružených gradientů

Úvod

Cílem této práce je praktická realizace několika variant metod sdružených gradientů, jejich aplikace pro řešení soustav lineárních rovnic s několika pravými stranami. S tímto problémem se setkáváme při numerickém řešení parciálních diferenciálních rovnic, například při řešení úloh tvarové optimalizace. Vzniklá matice soustav je symetrická, řídká a pozitivně definitní. Klasický přístup k řešení soustav lineárních rovnic s několika pravými stranami je založen na přímých řešičích, které provádějí faktorizaci matice soustavy a jednotlivé soustavy řeší dopřednou a zpětnou substitucí. Je zřejmé, že výše uvedený postup přestává být efektivní pro řešení rozsáhlých soustav, neboť paměťové nároky pro uložení faktorizované matice jsou značné.

Na jiném principu pracují iterační řešiče. Tyto metody vytvářejí posloupnost aproximovaných řešení soustavy. Matice soustavy se účastní pouze operace násobení s vektorem, její řídký charakter zůstává zachován, a proto je paměťová náročnost iteračních řešičů mnohem menší než u přímých. Nevýhodou iteračních řešičů ovšem je, že řeší soustavy lineárních rovnic pouze s jednou pravou stranou. Řešení soustav s několika pravými stranami lze sice nalézt postupně, nicméně efektivita tohoto postupu je nízká, neboť řešení těchto soustav hledáme nezávisle na předešlých řešeních.

Metoda sdružených gradientů se dnes stala prakticky standardem pro řešení soustav lineárních rovnic s jednou pravou stranou. Pro problémy s více pravými stranami v komerčních balících však stále dominují přímé řešiče. Numerické experimenty nám ovšem ukazují, že iterační řešiče jsou vždy paměťově méně náročné a v některých případech navíc rychlejší než standardní přímé řešiče. Dále uváděné algoritmy rozšiřují metodu sdružených gradientů tak, aby byla efektivní i pro řešení soustav lineárních rovnic s několika pravými stranami.

Iterační řešiče pro více pravých stran můžeme rozdělit v závislosti na tom, zda jsou všechny pravé strany dostupné současně. Jiné metody použijeme, pokud známe všechny pravé strany zároveň, a jiné, pokud se pravé strany dozvídáme postupně.

Pokud známe všechny pravé strany, můžeme k řešení těchto soustav použít dále uváděné blokové a postupné sdružené gradienty. V opačném případě můžeme použít modifikovanou Lanczosovu proceduru, která spočívá ve výpočtu několika vlastních vektorů odpovídajících nejmenším vlastním číslům matice soustavy. Tyto vektory se pak použijí k redukci rezidua [Dostál, Vondrák, Rasmussen, 1997]. V této práci se však budeme zabývat algoritmy, které předpokládají, že vektory všech pravých stran jsou známy současně.

1. Zadání úlohy

Uvažujme následující systém lineárních rovnic

$$K\,X = F, \qquad (1)$$

kde K je $n \times n$ symetrická pozitivně definitní matice, $F = [f^1,\ldots,f^i,\ldots,f^q]$ je $n \times q$ matice pravých stran a $X = [x^1,\ldots,x^i,\ldots,x^q]$ je $n \times q$ matice řešení. Úkolem je efektivně nalézt řešení soustavy (1).

2. Klasický přístup k řešení soustav lineárních rovnic s několika pravými stranami

Klasický přístup k řešení (1) je založen na faktorizaci matice *K*. Pokud použijeme *LU* faktorizaci, nejdříve rozložíme matici *K* a potom řešíme jednotlivé soustavy dopřednou a zpětnou substitucí. Paměťově i časově nejnáročnější operací u tohoto postupu je faktorizace matice *K*. Tato faktorizace se však provádí pouze jednou a řešení příslušných soustav s faktorizovanou maticí lze snadno získat dopřednou a zpětnou substitucí:

rozlož $K = LU$

for $i = 1,\ldots,q$

vyřeš $Ly = f^i$

vyřeš $Ux^i = y$

end.

Přímé řešiče jsou výhodné v tom, že využívají specifických rysů soustavy, například šířky pásu matice, nebo snadného řešení soustav s několika pravými stranami. Jak již bylo uvedeno, tyto výhody přestávají platit, pokud chceme řešit velké komplexní problémy, neboť paměťový prostor pro uložení faktorizované matice může být značný. Navíc faktorizace matice je obtížně paralelizovatelná operace, neboť její škálovatelnost závisí na šířce pásu matice.

Výše zmíněné nevýhody přímých řešičů vedou k hledání nových algoritmů, zvláště pokud chceme efektivně řešit velké, např. 3D problémy s několika pravými stranami. V posledních letech byl ve vývoji iteračních řešičů zaznamenán velký pokrok. Dříve často formulovaná nevýhoda, že rychlost konvergence je závislá na typu problému již zcela neplatí, neboť účinné

předpodmiňovací techniky v mnoha případech několikanásobně zlepšily výkon těchto metod.

3. Klasická metoda sdružených gradientů

Dříve než objasníme principy metod sdružených gradientů, které jsou určeny pro řešení soustav lineárních rovnic s několika pravými stranami, odvodíme si klasickou metodu sdružených gradientů. Tato metoda byla poprvé publikovaná v článku [Hestenes, Stiefel, 1952].

Uvažujme následující soustavu lineárních rovnic o n neznámých:

$$Kx = f \tag{3.1}$$

kde K je $n \times n$ symetrická pozitivně definitní matice, f je vektor pravé strany a x je (neznámý) vektor řešení, $f, x \subset R^n$.

Na řešení (3.1) se můžeme dívat také jako na minimalizaci kvadratického funkcionálu

$$\varphi(x) = \frac{1}{2} x^T K x - x^T f \ , \tag{3.2}$$

neboť $\nabla \phi(x) = Kx - f$, což znamená, že $x = K^{-1} f$ minimalizuje $\phi(x)$. Minimalizace $\varphi(x)$ a řešení $Kx = f$ jsou tedy ekvivalentní problémy.

Nová aproximace řešení x_k může být získána z předchozí aproximace x_{k-1} pomocí směru p_k s délkou kroku α_k předpisem

$$x_k = x_{k-1} + \alpha_k p_k \tag{3.3}$$

Hodnota parametru α_k je určena tak, aby minimalizovala $\varphi(x_{k-1} + \alpha_k p_k)$, tedy

$$\frac{\partial \varphi(x_{k-1} + \alpha_k p_k)}{\partial \alpha_k} = 0 \tag{3.4}$$

nebo ekvivalentně

$$p_k^T r_k = p_k^T (r_{k-1} - \alpha_k K p_k) = 0 \ \Leftrightarrow \ \alpha_k = p_k^T r_{k-1} / p_k^T K p_k \tag{3.5), (3.6}$$

kde $r_k = f - K x_k$ je reziduum *k*-té aproximace řešení x_k.

Nový směr p_k hledáme ve tvaru

$$p_k = r_{k-1} + \beta_k p_{k-1} \tag{3.7}$$

tak, aby byl *K*-ortogonální k předchozím směrům. Tuto podmínku zabezpečí relace

$$p_i^T K p_j = 0 \qquad \text{pro } i \neq j \tag{3.8}$$

kde parametr β_k je volen tak, aby platila podmínka

$$p_k^T K p_{k-1} = 0 \;\Leftrightarrow\; \beta_k = -p_{k-1}^T K r_{k-1} / p_{k-1}^T K p_{k-1} \tag{3.9), (3.10}$$

Pozorování

Pro metodu sdružených gradientů platí:

1. $p_i^T K p_j = 0$ pro $i \neq j$
2. $r_i^T r_j = 0$ pro $i \neq j$ a
3. $p_k^T r_k = 0$

Tyto vlastnosti zaručují důležitý rys: pokud počítáme na přesné aritmetice, získáme přesné řešení maximálně po n iteracích. V praktických případech je algoritmus prováděn tak dlouho, dokud norma rezidua není dostatečně malá. Vhodné kritérium k ukončení iteračního cyklu je například

$$||r_k|| > \varepsilon ||f||,$$

kde ε je nějaké malé kladné číslo, typicky 10^{-4}.

Nyní upravme vzorec pro výpočet koeficientu α_k. Pokud dosadíme do (3.6) vzorec (3.7) a využijeme pozorování č. 3, můžeme psát

$$\alpha_k = r_{k-1}^T r_{k-1} / p_k^T K p_k . \tag{3.11}$$

Ze vzorce (3.5) plyne, že

$$r_{k-1} = r_{k-2} - \alpha_{k-1} K p_{k-1} . \tag{3.12}$$

Pokud tuto rovnici vynásobíme vektorem r_{k-1} respektive r_{k-2}, platí v souladu s pozorováním č. 2 následující vztahy:

$$r_{k-1}^T r_{k-1} = -\alpha_{k-1} r_{k-1}^T K p_{k-1}, \tag{3.13}$$

$$r_{k-2}^T r_{k-2}^T = \alpha_{k-1} r_{k-2} K p_{k-1} . \tag{3.14}$$

Jestliže si nyní ze vzorce (3.7) rekurzivně vyjádříme r_{k-2} a tento vztah dosadíme do (3.14), můžeme s využitím pozorování č. 1 psát

$$r_{k-2}^T r_{k-2}^T = \alpha_{k-1} p_{k-1} K p_{k-1} . \tag{3.15}$$

Využitím (3.13) a (3.15) se výpočet koeficientu β_k zjednoduší do tvaru:

$$\beta_k = r_{k-1}^T r_{k-1} / r_{k-2}^T r_{k-2} . \tag{3.16}$$

Těmito úpravami jsme získali efektivnější variantu algoritmu, neboť v každé iteraci potřebujeme vypočítat pouze jednu operaci násobení matice soustavy a vektoru. Algoritmus metody sdružených gradientů lze zapsat následovně:

Algoritmus CG

Inicializace: $k = 0;\ r_0 = f - Kx_0$

while $\|r_k\| > \varepsilon \|f\|$ ***do***

begin

$k = k + 1$

if $k = 1$ ***then***

$p_1 = r_0$

else

$\beta_k = r_{k-1}^T r_{k-1} / r_{k-2}^T r_{k-2}$

$p_k = r_{k-1} + \beta_k p_{k-1}$

endif.

$u_k = Kp_k$

$\alpha_k = r_{k-1}^T r_{k-1} / p_k^T u_k$

$x_k = x_{k-1} + \alpha_k p_k$

$r_k = r_{k-1} - \alpha_k u_k$

end.

3.1 Metoda předpodmíněných sdružených gradientů

Uvažujme znovu soustavu lineárních rovnic se symetrickou pozitivně definitní maticí

$$Kx = f\ .$$

Pro urychlení konvergence můžeme použít předpodmínění. Princip metody předpodmíněných sdružených gradientů spočívá v aplikaci metody sdružených gradientů na transformovaný systém lineárních rovnic

$$\hat{K}\hat{x} = \hat{f}\ , \qquad (3.1.1)$$

kde $\hat{K} = C^{-1}KC^{-1}$, $\hat{x} = Cx$, $\hat{f} = C^{-1}f$ a C je symetrická pozitivně definitní matice, kterou vybíráme tak, aby matice $\hat{K}$ byla dobře podmíněná, a z důvodu, který uvedeme později, musí mít matice C^2 „jednoduchou“ strukturu.

Jestliže aplikujeme sdružené gradienty na soustavu (3.1.1), získáme následující algoritmus:

Algoritmus PCG verze 1

Inicializace: $k = 0;\ \hat{r}_0 = \hat{f} - K\hat{x}_0$

while $\|\hat{r}_k\| > \varepsilon \|\hat{f}\|$ ***do***

begin

$k = k + 1$

if $k = 1$ ***then***

$\hat{p}_1 = \hat{r}_0$

else

$\beta_k = \hat{r}_{k-1}^T \hat{r}_{k-1} / \hat{r}_{k-2}^T \hat{r}_{k-2}$

$\hat{p}_k = \hat{r}_{k-1} + \beta_k \hat{p}_{k-1}$

endif.

$\alpha_k = \hat{r}_{k-1}^T \hat{r}_{k-1} / \hat{p}_k^T C^{-1} K C^{-1} \hat{p}_k$

$\hat{x}_k = \hat{x}_{k-1} + \alpha_k \hat{p}_k$

$\hat{r}_k = \hat{r}_{k-1} - \alpha_k C^{-1} K C^{-1} \hat{p}_k$

end. (3.1.2)

V tomto případě označuje $\hat{r}_k = \hat{f} - \hat{K}\hat{x}_k$, respektive $\hat{x}_k$, k-tou aproximaci rezidua, respektive řešení transformovaného systému. Nás však zajímá řešení původní soustavy. Jednou z možností je získat toto řešení z rovnice $x_k = C^{-1}\hat{x}_k$. Nicméně pokud definujeme $\hat{p}_k = Cp_k$, $\hat{x}_k = Cx_k$ a $\hat{r}_k = C^{-1}r_k$, vyhneme se explicitní referenci na matici C^{-1}.

Pokud dosadíme do (3.1.2) a připomeneme, že $\hat{f} = C^{-1}f$ a $\hat{x} = Cx$, pak obdržíme

Algoritmus PCG verze 2

Inicializace: $k = 0;\ C^{-1}r_0 = C^{-1}(f - Kx_0)$

while $\|C^{-1}r_k\| > \varepsilon \|f\|$ ***do***

begin

$k = k + 1$

if $k = 1$ ***then***

$Cp_1 = C^{-1}r_0$

else

$\beta_k = (C^{-1}r_{k-1})^T (C^{-1}r_{k-1}) / (C^{-1}r_{k-2})^T (C^{-1}r_{k-2})$

$Cp_k = C^{-1}r_{k-1} + \beta_k Cp_{k-1}$

endif.

$\alpha_k = (C^{-1}r_{k-1})^T (C^{-1}r_{k-1}) / (Cp_k)^T (C^{-1}KC^{-1})(Cp_k)$

$Cx_k = Cx_{k-1} + \alpha_k Cp_k$

$$C^{-1}r_k = C^{-1}r_{k-1} - \alpha_k (C^{-1}KC^{-1})Cp_k$$

end. (3.1.3)

Označme nyní předpodmiňovač $M = C^2$. Nechť M je také symetrická a pozitivně definitní matice a nechť je vektor z_k řešení soustavy $Mz_k = r_k$. Pak můžeme výše uvedený algoritmus zjednodušit do této konečné podoby:

Algoritmus PCG

Inicializace: $k = 0;\ r_0 = f - Kx_0$
while $\|r_k\| > \varepsilon \| f \|$ ***do***
begin
solve $Mz_k = r_k$
$k = k + 1$
if $k = 1$ ***then***
$p_1 = z_0$
else
$\beta_k = r_{k-1}^T z_{k-1} / r_{k-2}^T z_{k-2}$
$p_k = z_{k-1} + \beta_k p_{k-1}$
endif.
$u_k = Kp_k$
$\alpha_k = r_{k-1}^T z_{k-1} / p_k^T u_k$
$x_k = x_{k-1} + \alpha_k p_k$
$r_k = r_{k-1} - \alpha_k u_k$
end.

Pozorování:

1. Lze ukázat, že pro tento algoritmus platí:
   $r_i^T M^{-1} r_j = 0$ pro $i \neq j$
   $p_i^T (C^{-1}KC^{-1}) p_j = 0$ pro $i \neq j$
2. Jmenovatel $r_{k-2}^T z_{k-2} = z_{k-2}^T M z_{k-2}$ nebude nikdy nulový, neboť M je pozitivně definitní.
3. Matice C se používá pouze při odvozování, v samotném algoritmu používáme předpodmiňovač $M = C^2$.
4. Předpodmiňovač M je vhodné volit tak, aby soustava $Mz_k = r_k$ byla snáze řešitelná a zároveň algoritmus rychle konvergoval.

3.2 SSOR předpodmínění

V našich experimentech jsme použili SSOR předpodmínění. Matice M se v tomto případě vypočítá vztahem

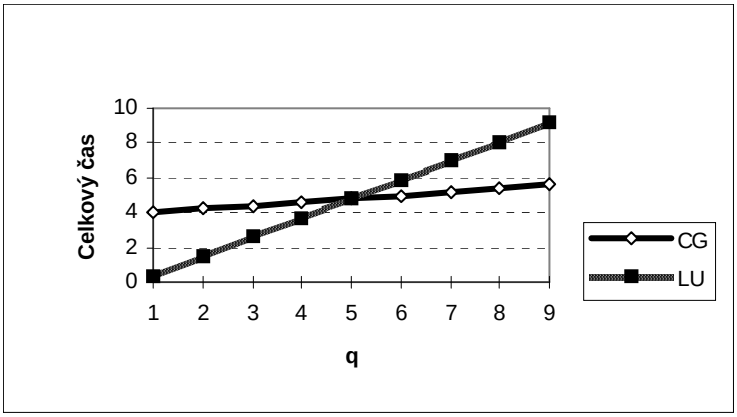
$$M = (L + D)D^{-1}(L^T + D), \qquad (3.2.1)$$

kde L, respektive D, je dolní trojúhelníková část, respektive diagonála matice K.

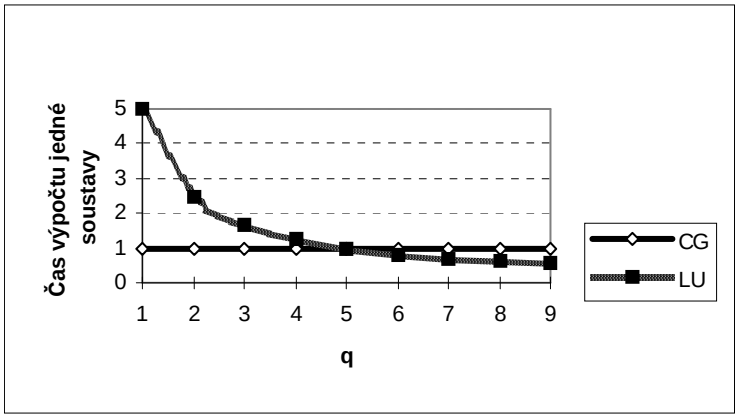
(Platí tedy $K = L + D + L^T$.)

4. Srovnání klasických metod

Jak již bylo uvedeno, pro řešení soustav lineárních rovnic s několika pravými stranami můžeme samozřejmě použít i standardní metodu sdružených gradientů tak, že budeme postupně řešit soustavy $Kx^1 = f^1$, $Kx^2 = f^2$, ..., $Kx^q = f^q$. Počet iterací potřebných k vyřešení q soustav je však obecně q-krát vyšší než pro soustavu s jednou pravou stranou. To znamená, že celková doba výpočtu je asi q-krát delší než doba výpočtu pro soustavu s jednou pravou stranou. Pokud však použijeme přímý řešič založený na faktorizaci matice K, pak pro každou další pravou stranu počítáme pouze dopřednou a zpětnou substituci.



a)



b)

Obrázek 4.1 Doba výpočtu metody sdružených gradientů (CG) a přímého řešiče (LU) v závislosti na počtu pravých stran a) celková doba výpočtu b) průměrná doba výpočtu na jednu pravou stranu.

Porovnání rychlosti přímého a iteračního řešiče je na obrázku 4.1. Parametr *q* představuje počet pravých stran.

Pro úlohu s jednou pravou stranou je rychlejší iterační řešič, neboť cena faktorizace u přímého řešiče je vysoká. Pokud zvětšujeme počet pravých stran, stává se výhodnější použití přímého řešiče, neboť cena dopředné a zpětné eliminace je nižší než cena sdružených gradientů. Ovšem na druhé straně paměťové nároky u metody sdružených gradientů jsou vždy nižší než je tomu u přímých řešičů. Při těchto úvahách musíme vzít samozřejmě v úvahu velikost problému (počet neznámých). Pro běžné 2D simulace jsou přímé řešiče dobrou alternativou (jsou paměťově náročnější, ale celkově rychlejší), nicméně pro 3D problémy už vítězí iterační řešiče jak v oblasti časové, tak i v oblasti paměťové složitosti.

Hlavní důvod, proč výkon metody sdružených gradientů u problémů s více pravými stranami zaostává, je ten, že řešení soustav probíhá navzájem nezávisle na sobě. To znamená, že informace, které byly získány při řešení předchozích soustav, nejsou použity pro řešení zbývajících soustav. Lze proto předpokládat, že souběžné řešení několika soustav může výkon sdružených gradientů zlepšit.

5. Metody sdružených gradientů pro řešení soustav lineárních rovnic s několika pravými stranami

V této kapitole popíšeme principy, které rozšíří metodu sdružených gradientů o algoritmy, které lze použít pro efektivní řešení soustav lineárních rovnic s několika pravými stranami.

5.1. Metoda BCG

Metoda BCG je zobecněním klasické metody sdružených gradientů. Uvažujme soustavu lineárních rovnic (1). Obdobně jako u metody sdružených gradientů definujme funkcionál

$$\Phi(X) = [\phi_1(x_1),...,\phi_i(x_i),...,\phi_q(x_q)] \tag{5.1.1}$$

kde

$$\varphi_i(x_i) = \frac{1}{2} x_i^T K x_i - x_i^T f_i . \tag{5.1.2}$$

I v tomto případě můžeme ukázat, že minimalizace $\Phi(X)$ je řešení soustavy (1).

Předpokládejme, že *k*-tá aproximace řešení je $X_k = [x_k{}^1,...,x_k{}^i,...,x_k{}^q]$. Potom odpovídající reziduum $R_k = [r_k{}^1,...,r_k{}^i,...,r_k{}^q]$ je definované předpisem

$$R_k = F - KX_k = [f^1 - Kx_k{}^1,..., f^q - Kx_k{}^q]. \tag{5.1.3}$$

Novou aproximaci řešení X_k získáme

$$X_k = X_{k-1} + P_k \alpha_k , \tag{5.1.4}$$

kde $P_k = [p_k{}^1,..., p_k{}^i,..., p_k{}^q]$ je $n \times q$ matice směrů, α_k je $q \times q$ matice délky kroku, která je určena splněním podmínky

$$\min_{\alpha_k \subset R^q \times R^q} \Phi(X_{k-1} + P_k \alpha_k), \tag{5.1.5}$$

tedy

$$\frac{\partial \Phi(X_{k-1} + P_k \alpha_k)}{\partial \alpha_k} = 0, \tag{5.1.6}$$

nebo jednodušeji

$$P_k^T R_k = P_k^T (R_{k-1} - K P_k \alpha_k) = 0. \tag{5.1.7}$$

Z podmínky kolmosti R_k na P_k můžeme vyjádřit α_k vztahem

$$\alpha_k = (P_k^T K P_k)^{-1} P_k^T R_{k-1}. \tag{5.1.8}$$

Nový směr P_k hledáme ve tvaru

$$P_k = R_{k-1} + P_{k-1} \beta_k \tag{5.1.9}$$

tak, aby byl *K*-ortogonální k předchozímu směru P_{k-1}, tedy aby platilo

$$P_k^T K P_{k-1} = 0. \qquad (5.1.10)$$

Splnění této podmínky zaručíme $q \times q$ maticí β_k, která je určena předpisem

$$\beta_k = -(P_{k-1}^T K P_{k-1})^{-1} P_{k-1}^T K R_{k-1}. \qquad (5.1.11)$$

Pokud definujeme $P_0 = R_0 = F - KX_0$ a upravíme vztahy (5.1.7) a (5.1.10), můžeme napsat následující vlastnosti:

1. $P_i^T K P_j = 0$ pro $i \neq j$
2. $R_i^T R_j = 0$ pro $i \neq j$ a
3. $P_k^T R_k = 0$.

V souladu s těmito vlastnostmi můžeme výpočet α_k a β_k také vyjádřit vztahy

$$\alpha_k = (P_k^T K P_k)^{-1} R_{k-1}^T R_{k-1} \qquad (5.1.12)$$

$$\beta_k = (R_{k-2}^T R_{k-2})^{-1} R_{k-1}^T R_{k-1} \qquad (5.1.13)$$

Podobně jako u metody standardních sdružených gradientů (CG), je důležitý fakt, že při použití přesné aritmetiky může BCG metoda získat přesné řešení nejvýše při [*(n + m -1) / m*] iteracích. Nicméně v praktické implementaci BCG je iterační proces ukončen za splnění podmínky

$$\max(||r_k^i|| > \varepsilon ||f^i||, i = 1,\ldots,q). \qquad (5.1.14)$$

Předpodmíněná metoda BCG

Pokud použijeme předpodmiňovač *M*, pro výpočet α_k a směru P_k použijeme tyto předpisy:

$$\alpha_k = (P_k^T K P_k)^{-1} R_{k-1}^T Z_{k-1}, \qquad (5.1.15)$$

$$P_k = Z_{k-1} + P_{k-1}\beta_k \qquad (5.1.16)$$

kde

$$Z_k = M^{-1} R_k,$$

$$\beta_k = (R_{k-2}^T Z_{k-2})^{-1} R_{k-1}^T Z_{k-1}.$$

Analogicky k CG algoritmu můžeme BCG s předpodmiňovačem *M* napsat následovně:

Algoritmus BCG [Block CG]

Inicializace: $k = 0;\ R_0 = F - KX_0$

while $\max(||r_k^i|| > \varepsilon ||f^i||, i = 1,\ldots,q)$ ***do***

begin

solve $MZ_k = R_k$

$k = k + 1$

if $k = 1$ ***then***

$P_1 = Z_0$

else

$\beta_k = (R_{k-2}^T Z_{k-2})^{-1} R_{k-1}^T Z_{k-1}$

$P_k = Z_{k-1} + P_{k-1}\beta_k$

endif.

$U_k = KP_k$

$\alpha_k = (P_k^T U_k)^{-1} R_{k-1}^T Z_{k-1}$

$X_k = X_{k-1} + P_k \alpha_k$

$R_k = R_{k-1} - U_k \alpha_k$

end.

Konvergence metody BCG

Předpoklad, že všechny soustavy rovnic konvergují ve stejné iteraci, je v praktických případech nereálný. Jak bylo uvedeno výše, iterační cyklus BCG algoritmu se ukončuje až v okamžiku, kdy jsou normy všech reziduí menší než je předepsaná tolerance ε. Často se však stává, že jsou některé soustavy lineárního systému již s uspokojivou přesností vyřešeny. V těchto případech BCG zpřesňuje tyto řešení zbytečně, a navíc tento postup může vést i k numerické nestabilitě celého algoritmu, neboť normy vektorů *p* a *r*, které přísluší těmto vyřešeným soustavám, mohou být významně menší než normy vektorů zbylých nevyřešených soustav, což způsobí, že matice $(R_k^T M^{-1} R_k)^{-1}$ a $(P_k^T K P_k)^{-1}$ budou špatně podmíněny, nebo budou dokonce singulární.

Vylepšení metody BCG

Abychom se vyhnuli výše uvedeným problémům, budeme metodu BCG provádět jen pro ty soustavy, které ještě nejsou s dostatečnou přesností vyřešeny. Pokud se při testování přesnosti řešení těchto soustav ukáže, že některé soustavy jsou již s dostatečnou přesností vyřešeny, vyjmeme je z iteračního cyklu, a další iterace budeme provádět jen s dosud nevyřešenými soustavami.

Příčiny numerické nestability metody BCG

Při implementaci metody BCG potřebujeme vypočítat inverzní matice k $(R_k^T M^{-1} R_k)$ a $(P_k^T K P_k)$. Aby byl algoritmus BCG stabilní, je dále nutné zaručit, aby sloupcové vektory matic P_k a R_k byly lineárně nezávislé. V případě, že matice P_k a R_k nemají plnou hodnost, není metoda BCG definována. Jelikož jsou hodnosti P_k a R_{k-1} shodné, stačí sledovat hodnost jen jedné z těchto matic. V literatuře se setkáváme s několika postupy jak zachovat numerickou stabilitu BCG. Jednou z možností je vyjímat závislé soustavy z problému. Nicméně normy reziduí z redundantních sloupců, které vyjímáme z iteračního procesu, nemusí být vždy nutně malé. Řešení soustav, které odpovídají těmto vyjmutým vektorům, můžeme vypočítat zvlášť. V další sekci však ukážeme daleko efektivnější postup manipulace s těmito redundantními sloupci.

5.2. Metoda SCG

Podstatou metody SCG je využití směru, který byl už použit k řešení jedné soustavy, k získání aproximace řešení pro všechny zbylé soustavy. Tato metoda může řešení soustav lineárních rovnic s několika pravými stranami výrazně urychlit, neboť získání nového směru je u metody sdružených gradientů „drahá“ operace - zahrnuje násobení matice a vektoru $u_k = Kp_k$ a vyřešení předpodmíněného systému $Mz_k = r_k$. Uvažujme lineární systém s q pravými stranami

$$K[x^1, x^2, ..., x^m, ..., x^s, ..., x^q] = [f^1, f^2, ..., f^m, ..., f^s, ..., f^q] \qquad (5.2.1)$$

Předpokládejme, že předpodmíněné sdružené gradienty jsou použity k řešení *m*-té soustavy lineárního systému (v literatuře bývá tato soustava označována jako „main“, „master“ nebo „seed“). Tento proces generuje až do *k*-té iterace posloupnost $(L^{-1}KL^{-T})$-ortogonálních neboli sdružených vektorů, které vytvářejí Krylovův podprostor $\Re_k^m \subset R^{n \times k}$. Podstatou metody je tedy využití těchto sdružených vektorů k výpočtu řešení a reziduí pro soustavy ve „slave“ systému. Řešení a rezidua „slave“ soustav se v každé iteraci vypočítají vztahy

$$x_k^s = x_{k-1}^s - \alpha_k^{sm} p_k^m \qquad (5.2.2)$$

$$r_k^s = r_{k-1}^s - \alpha_k^{sm} K p_k^m \qquad (5.2.3)$$

Parametr α_k^{sm} vychází z podmínky, že vektor r_k^s je kolmý k vektoru p_k^m, tedy

$$\left(r_k^s\right)^T p_k^m = 0 \quad \Leftrightarrow \quad (r_{k-1}^s - \alpha_k^{sm} K p_k^m)^T p_k^m = 0 \tag{5.2.4}$$

Pro výpočet parametru α_k^{sm} pak dostáváme vztah

$$\alpha_k^{sm} = \frac{(r_{k-1}^s)^T p_k^m}{(p_k^m)^T K p_k^m} \tag{5.2.5}$$

Popis SCG metody

Při inicializaci vybereme (libovolnou) soustavu ze systému lineárních rovnic a označíme ji jako „master“. Zbývající soustavy tvoří „slave“ systém. Jakmile je „master“ systém s požadovanou přesností vyřešen, vyjmeme ho z problému a ze „slave“ systému vybereme další soustavu, kterou označíme jako „master“. Na tento nový „master“ systém spustíme znovu metodu sdružených gradientů. Řešení, respektive rezidua „slave“ soustav, aktualizujeme v každé iteraci vzorcem (5.2.2), respektive vzorcem (5.2.3). Iterační proces končí v okamžiku, kdy jsou všechny soustavy s požadovanou přesností vyřešeny. Metodu SCG s předpodmiňovačem M pro řešení soustav lineárních rovnic s několika pravými stranami můžeme zapsat následovně:

Algoritmus SCG

Inicializace: $k = 0;\ r_0^i = f^i - K x_0^i,\ i = 1,\ldots,q$
for $m = 1$ ***until*** q ***do***
begin */* Pro všechny soustavy */*
$new\_main = TRUE$
while $||r_k^m|| > \varepsilon ||f^m||$ ***do***
begin
solve $Mz_k = r_k^m$
$k = k + 1$
if *new_main* ***then***
new_main = FALSE
$p_k = z_{k-1}$
else
$\beta_k = (r_{k-1}^m)^T z_{k-1} / (r_{k-2}^m)^T z_{k-2}$
$p_k = z_{k-1} + \beta_k p_{k-1}$
endif.
$u_k = K p_k$
$\delta = p_k^T u_k$
for $j =$ m ***until*** q ***do***
begin */* „slave“ systém je (m+1,...,q) */*

$$\alpha_k = (r_{k-1}^j)^T z_{k-1} / \delta$$
$$x_k^j = x_{k-1}^j + \alpha_k p_k$$
$$r_k^j = r_{k-1}^j - \alpha_k u_k$$

end.

end.

end.

Konvergence metody SCG

Pro studium konvergenčních vlastností „slave“ systému separujeme vektor pravé strany f^s do komponent $(f^s)^\perp$ a $(f^s)^\parallel$ tak, aby platil vztah

$$f^s = (f^s)^\perp + (f^s)^\parallel, \qquad (5.2.6)$$

kde

$$(f^s)^\parallel = \frac{(f^s)^T f^m}{(f^m)^T f^m} f^m = \xi f^m$$

$$(f^s)^\perp = f^s - \xi f^m \qquad (5.2.7)$$

Na řešení „slave“ soustavy se můžeme dívat jako na řešení dvou následujících soustav lineárních rovnic:

$$K(x^s)^\parallel = (f^s)^\parallel \qquad (5.2.8)$$

$$K(x^s)^\perp = (f^s)^\perp \qquad (5.2.9)$$

$$x^s = (x^s)^\parallel + (x^s)^\perp \qquad (5.2.10)$$

Pro jednoduchost předpokládejme, že počáteční aproximace řešení x_0 je nulový vektor. Proto můžeme psát:

$$r_0^m = f^m \qquad (5.2.11)$$

$$r_0^s = f^s \qquad (5.2.12)$$

$$(r_0^s)^\parallel = (f^s)^\parallel \qquad (5.2.13)$$

$$(r_0^s)^\perp = (f^s)^\perp \qquad (5.2.14)$$

Z minimalizační vlastnosti „slave“ systému plyne, že „slave“ systém $K(x^s)^{\parallel} = (f^s)^{\parallel}$ bude mít stejnou rychlost konvergence jako „master“ systém, tedy že jeho rezidua splňují podmínku:

$$\begin{aligned} \|(r_k^s)^{\parallel}\|_{K^{-1}} &\le \gamma^k \|(r_0^s)^{\parallel}\|_{K^{-1}} \\ &= \gamma^k \|(\xi r_0^m)\|_{K^{-1}} \end{aligned} \qquad (5.2.15)$$

kde $\gamma = (\sqrt{(\kappa)} - 1) / (\sqrt{(\kappa)} + 1)$, κ je číslo podmíněnosti matice *K*, a k je číslo iterace.

Rezidua „kolmé“ části „slave“ systému můžeme vyjádřit jako:

$$\begin{aligned} (r_k^s)^{\perp} &= (f^s)^{\perp} - K(f_k^s)^{\perp} \\ &= (r_0^s)^{\perp} - \sum_{j=1}^{k} (\alpha_j^{sm})^{\perp} K p_j^m \\ &= r_0^s - \xi r_0^m - \sum_{j=1}^{k} (\alpha_j^{sm})^{\perp} K^j r_0^m \\ &= (I - P_{k+1}) r_0^s , \end{aligned} \qquad (5.2.16)$$

kde $P_{k+1} r_0^s = \sum_{j=1}^{k+1} \eta_j K^{j-1} r_0^m$ je projekce z r_0^s do Krylova prostoru z „master“ systému $\Re_{k+1}^m$ a η_j je konstantní koeficient.

Kombinací výše uvedených výsledků dostáváme:

$$\begin{aligned} \|r_k^s\|_{K^{-1}} &= \|(r_k^s)^{\parallel} + (r_k^s)^{\perp}\|_{K^{-1}} \\ &\le (I - P_{k+1}) r_0^s\|_{K^{-1}} + \gamma^k \|\xi r_0^m\|_{K^{-1}} \end{aligned} \qquad (5.2.17)$$

Nyní můžeme formulovat následující konvergenční vlastnosti.

1. Jestliže je r_0^s kolmý k $\Re_{k+1}^m$, tedy $\xi = 0$ a $P_{m+1} r_0^s = 0$, pak nelze očekávat žádné zlepšení rychlosti konvergence „slave“ systému.
2. Jestliže je r_0^s rovnoběžný k r_0^m ($(r_0^s)^{\perp} = 0$ nebo r_0^s je obsažený v $\Re_{k+1}^m$), pak $P_{k+1} = I$ a norma rezidua „slave“ systému se bude zmenšovat stejně rychle jako norma rezidua „master“ systému.
3. Jestliže je r_0^s obsažen v $K\,\Re_{k+1}^m$, tedy $\xi = 0$ a $(I - P_{k+1}) r_0^s = 0$, pak je „slave“ systém vyřešen přesně a $\|r_k^s\| = 0$.

Pokud se norma rezidua „slave“ systému nezmenšila pod požadovanou toleranci, přemístíme „slave“ systém do „master“ systému, nastavíme $r_0^m \equiv r_k^s$ a na tento nový „master“ systém spustíme znovu metodu sdružených gradientů.

V následujícím textu budeme označovat vektor, který byl v předchozím „master“ systému, závorou (např. $\bar{r}^m$). Vektor, který je nyní v „master“ systému, budeme označovat bez závory (např. r^m). Nechť l je celkový počet iterací předchozího „master“ systému. Pro studium vlastností nynější „master“ soustavy připomeneme, že počáteční rezidua jsou kolmá ke Krylovu podprostoru z předchozího „master“ systému, tedy

$$(\bar{r}_0^m)^T v = 0 \qquad \text{pro všechna } v \in \bar{\Re}_l^m, \tag{5.2.18}$$

kde $\bar{\Re}_l^m = span\{\bar{r}_0^m, K\bar{r}_0^m, \ldots, K^{l-1}\bar{r}_0^m\} = span\{\bar{p}_1, \ldots, \bar{p}_l\}$. Nynější „master“ systém bude generovat směry rozšířením nového Krylova podprostoru

$$\Re_l = span\{r_0^m, Kr_0^m, \ldots, K^{l-1}r_0^m\} = span\{p_1, \ldots, p_l\} \tag{5.2.19}$$

Předchozí a nynější směry lze v Krylovu podprostoru vyjádřit:

$$\bar{p}_i = \sum_{j=1}^{i} \eta_j K^{j-1} \bar{r}_0^m \tag{5.2.20}$$

$$p_i = \sum_{j=1}^{i} \eta_j K^{j-1} r_0^m \tag{5.2.21}$$

Nyní chceme ukázat, že:

$$(\bar{p}_i)^T p_j = 0 \quad \text{pro } 1 \le j \le l - i + 1 \tag{5.2.22}$$

K tomu potřebujeme dokázat, že každá složka p_j je kolmá k $\bar{p}_i$, tedy

$$(\bar{p}_i)^T (K^{j-1} r_0^m) = 0 \qquad \text{pro } 1 \le j \le l - i + 1 \tag{5.2.23}$$

Využitím symetrie matice *K* dostáváme:

$$\begin{aligned} (\bar{p}_i)^T (K^{j-1} r_0^m) &= \left(\sum_{k=1}^{i} \eta_k K^{k-1} \bar{r}_0^m \right)^T (K^{j-1} r_0^m) \\ &= \left(\sum_{k=1}^{i} \eta_k K^{j+k-2} \bar{r}_0^m \right)^T (r_0^m) \end{aligned} \tag{5.2.24}$$

Pokud je $j + i - 2$ menší než $l - 1$, pak výraz $\sum_{k=1}^{i} \eta_k K^{j+k-2} \bar{r}_0^m$ je obsažen v předchozím Krylovu podprostoru, a ten je proto kolmý k nynějšímu reziduu r_0^m, to znamená, že

$$(\bar{p}_i)^T (K^{j-1} r_0^m) = \left(\sum_{k=1}^{i} \eta_k K^{j+k-2} \bar{r}_0^m \right)^T (r_0^m) = 0 . \qquad (5.2.25)$$

Závěr tohoto pozorování je ten, že vektory $\bar{p}_i$ a p_j vytvářejí dva Krylovy podprostory, které jsou si pro $i + j \leq l + 1$ navzájem ortogonální. Zvláště pokud j nepřekročí $l/2$, systém $\bar{p}_1, \ldots, \bar{p}_i, p_1, \ldots, p_i$, který má dimenzi $2i$, vytváří bázi podprostoru:

$$\begin{aligned} \Re^* &= span\{\bar{r}_0^m, K\bar{r}_0^m, \ldots, K^{i-1}\bar{r}_0^m; r_0^m, Kr_0^m, \ldots, K^{i-1}r_0^m\} \\ &= span\{Z, KZ, \ldots, K^{i-1}Z\}, \end{aligned} \qquad (5.2.26)$$

kde $Z = \{\bar{r}_0^m, r_0^m\}$. Tedy, pro počet iterací menší než $l/2$ je nynější „master“ systém matematicky ekvivalentní s metodou blokových sdružených gradientů a prvních $l/2$ iterací z nynějšího „master“ systému konverguje stejně rychle jako metoda BCG.

5.3 Metoda SBCG

Jak bylo uvedeno výše, pokud se v iteračním cyklu metody BCG ukáže, že matice směrů P_k a matice reziduí R_k nemají plnou hodnost, musíme příslušné závislé soustavy vyjmout z problému. Nabízí se otázka, jak lze efektivně získat řešení těchto závislých soustav. Elegantní metodu formulovali Suarjana a Law [Suarjana, Law, 1994]. Navrhují vyjímat závislé soustavy pouze z procesu *K*-ortogonalizace a příslušná řešení těchto závislých soustav počítat blokovou formou SCG algoritmu. Tuto metodu pojmenovali Successive Block Conjugate Gradient (SBCG). Algoritmus zavádí podobně jako SCG dvě skupiny soustav:

- master (účastní se procesu *K*-ortogonalizace)
- slave (neúčastní se procesu *K*-ortogonalizace, obsahuje lineárně závislé sloupce)

Řešení soustav, které jsou v „master“ systému se počítají blokovými sdruženými gradienty. Zbylé soustavy, které jsou ve „slave“ systému se dopočítají blokovou formou metody SCG.

Popis SBCG metody

Při inicializaci jsou všechny soustavy označené jako „master“. „Slave“ systém tedy neobsahuje žádnou soustavu. Jestliže se kdykoliv v iteračním cyklu zjistí, že některé sloupce matice P_k nebo R_k ztratily svou lineární nezávislost, přemístíme soustavy, které odpovídají těmto závislým sloupcům z „master“ do „slave“ systému. Tím máme zajištěnu stabilitu řešení, neboť proces *K*-ortogonalizace pak bude vždy prováděn pouze s lineárně nezávislými maticemi.

Jedna z výhod této metody je, že soustavy, které přemístíme do „slave“ systému, konvergují současně se soustavami, které zůstaly v „master“ systému. Tato vlastnost platí, neboť rezidua „slave“ systému jsou v podprostoru reziduí „master“ systému. Je tedy splněna druhá podmínka konvergenční vlastnosti SCG algoritmu, která zajišťuje, že soustavy v „master“ i „slave“ systému konvergují současně.

SBCG algoritmus s předpodmiňovačem *M* je uveden níže.

Algoritmus SBCG [Successive Block CG]

Inicializace: $k = 0;\ R_0 = F - KX_0$

$m = [1,...,q],\ s = []$ /* *master = všechny soustavy* */

while $(\|r_k^m\| > \varepsilon \| f^m \|)$ ***do***

begin

solve $MZ_k = R_k^m$

$k = k + 1$

if $k = 1$ ***then***

$P_1 = Z_0$

else

$\beta_k = ((R_{k-2}^m)^T Z_{k-2})^{-1} (R_{k-1}^m)^T Z_{k-1}$

$P_k = Z_{k-1} + P_{k-1}\beta_k$

endif.

Lin. závislé soustavy přesuň z „master“ do „slave“ systému

$U_k = KP_k$

$[\alpha_k^{mm}, \alpha_k^{sm}] = (P_k^T U_k)^{-1} [(R_{k-1}^m)^T Z_{k-1}, (R_{k-1}^s)^T Z_{k-1}]$

$[X_k^m, X_k^s] = [X_{k-1}^m, X_{k-1}^s] + P_k [\alpha_k^{mm}, \alpha_k^{sm}]$

$[R_k^m, R_k^s] = [R_{k-1}^m, R_{k-1}^s] - U_k [\alpha_k^{mm}, \alpha_k^{sm}]$

end.

Stanovení lineární závislosti vektorů

Pro zaručení numerické stability SBCG metody potřebujeme zaručit, aby soustavy, které jsou v „master“ systému, měly vždy lineárně nezávislé sloupce matic P_k a R_k. Připomeňme, že tento požadavek se řeší přesouváním lineárních soustav z „master“ do „slave“ systému.

Nechť máme vektory p_i a p_j. Úhel mezi těmito vektory je definován předpisem

$$\cos(\phi) = \frac{p_i^T p_j}{\| p_i \| \| p_j \|} \tag{5.3.1}$$

Pokud je $\phi = 0$, pak platí

$$\frac{p_i^T p_j}{\| p_i \| \| p_j \|} = 1 \tag{5.3.2}$$

a vektor p_i vyjmeme z „master“ systému, neboť je lineárně závislý na p_j.

Praktické zkušenosti s chováním SBCG algoritmu nám však ukázaly, že vyjímat soustavy z „master“ systému až v okamžiku, kdy již jsou lineárně závislé, není efektivní. K numerické nestabilitě při výpočtu inverzí k $R_{k-2}^T Z_{k-2}$ a $P_k^T U_k$ dochází v „master“ systému díky zaokrouhlovacím chybám v počítači i v případě, kdy sloupce matic P_k respektive R_k jsou již „téměř“ lineárně závislé. Proto jsme jako kritérium pro stanovení lineární závislosti vektorů p_i a p_j vzali splnění podmínky

$$1 - \frac{|p_i^T p_j|}{\| p_i \| \| p_j \|} < coef \ , \tag{5.3.3}$$

kde *coef* je koeficient lineární závislosti. Pokud je podmínka (5.3.3) splněna, odstraníme *i*-tou soustavu z „master“ systému.

SBCG jako zobecnění SCG a BCG

Na SBCG se můžeme dívat jako na zobecnění SCG metody. V SCG algoritmu obsahoval „master“ systém právě jednu soustavu. V SBCG může „master“ systém obsahovat více než jednu soustavu. Pokud bude obsahovat „master“ systém SBCG pouze jednu soustavu, přejde SBCG v SCG algoritmus.

Pokud budou všechny soustavy v „master“ systému, SBCG přejde v BCG algoritmus. Chování SBCG algoritmu ovlivňuje velkou měrou velikost koeficientu *coef*.

Pozorování

1. Pokud je *coef* libovolné číslo větší než 1, SBCG přejde na SCG algoritmus, neboť podmínka přechodu soustavy do „slave“ systému (5.3.3) je splněna pro jakékoliv dva vektory p_i a p_j. „Master“ systém bude tedy vždy obsahovat právě jednu soustavu.

2. Pokud se *coef* blíží k jedničce, převládá „successive“ charakter SBCG, neboť podmínka přechodu soustavy do „slave“ systému (5.3.3) je snadněji splnitelná a proto soustavy opouštějí „master“ systém snadno.

3. Pokud se *coef* blíží k nule, bude převládat blokový charakter SBCG, neboť podmínka (5.3.3) přechodu soustavy do „slave“ systému se stává méně snadno splnitelnou, a proto soustavy opouštějí „master“ systém pomaleji.

4. Pokud zvolíme za *coef* číslo menší než 0, podmínka (5.3.3) přechodu soustavy do „slave“ systému nebude nikdy splněna, to znamená, že počet slave soustav bude vždy nulový. SBCG tedy přejde na BCG algoritmus. Optimální velikost koeficientu *coef* je nutné vybrat na základě numerických experimentů.

5.4 Obecné porovnání BCG, SCC a BSCG

V následující tabulce je uvedeno porovnání principů výše zmiňovaných algoritmů při řešení soustavy lineárních rovnic s q pravými stranami. Údaje v tabulce se týkají pouze jedné iterace metody.

	SCG	**SBCG**	**BCG**
Proces K-ortogonalizace	prováděn jen pro jednu soustavu	prováděn jen pro lin. nezávislé soustavy	prováděn pro všechny soustavy
Počet master soustav	$M = 1$	$M \subset < 1; q >$	$M = q$
Počet násobení matice a vektoru	$Kpoper = 1$	$Kpoper \subset < 1; q >$	$Kpoper = q$
Počet slave soustav	$S = q - 1$	$S \subset < 0; q - 1 >$	$S = 0$

Tabulka 5.4.1 Porovnání principů SCG, SBCG a BCG

Porovnání výpočetní a paměťové náročnosti

Počet iterací u BCG metody je sice obvykle menší než u SCG, ale na druhé straně výpočetní náročnost jedné iterace je menší u SCG, neboť BCG metoda provede tolik operací násobení matice a vektoru, (*Kp-operací*), kolik je nevyřešených soustav, zatímco SCG metoda provádí vždy pouze jednu *Kp-operaci*. Na druhé straně - násobení matice a vektoru, které probíhá v jedné iteraci BCG metody, jsou na sobě nezávislé operace a mohou být vykonávány souběžně na více procesorech. Srovnání paměťové náročnosti je v tabulce 5.4.2. Nejméně paměti vyžaduje SCG algoritmus. Paměť se ušetří při procesu *K*-ortogonalizace, která je vždy vykonávána jen pro jednu soustavu.

Data	**SCG**	**SBCG**	**BCG**
Z, P, U	$n \times 1$	$n \times M$	$n \times q$
β	1	$M \times M$	$q \times q$
α	$1 \times q$	$M \times q$	$q \times q$
X, R	$n \times q$	$n \times q$	$n \times q$

Tabulka 5.4.2 Porovnání paměťové náročnosti SCG, SBCG a BCG

Shrnutí

V této části v krátkosti shrneme výhody a nevýhody SCG, BCG a SBCG algoritmů.

1. BCG

- výhoda: současné řešení soustav vede k větší rychlosti konvergence a efektivní paralelizaci
- nevýhoda: numerická nestabilita - metoda obecně není stabilní

2. SCG

- výhoda: snadná a paměťově nenáročná implementace, numerická stabilita - metoda vždy konverguje
- nevýhoda: velikost zrychlení závisí na typu pravých stran. V nejhorším případě nemusí dojít oproti standardním sdruženým gradientům k žádnému zrychlení

3. SBCG

- výhoda: spojuje výhody obou základních přístupů
- nevýhoda: poměrně obtížná implementace

6 Návrh paralelizace metod sdružených gradientů pro řešení soustav lin. rovnic s několika pravými stranami

V této kapitole se budeme zabývat analýzou a návrhy paralelních implementací metod BCG, SCG a SBCG pro multiprocesorové systémy s distribuovanou pamětí. Vycházet budeme z algoritmů publikovaných v předchozí kapitole. Algoritmy napíšeme v syntaxi programovacího jazyka Matlab, který je dnes v podstatě standardem v maticovém počítání.

6.1 Zadání úlohy

Uvažujme soustavu lineárních rovnic s q-pravými stranami (1). Tuto soustavu chceme řešit nějakým paralelním algoritmem na víceprocesorovém počítači. Aby výsledný program fungoval co možná nejlépe, měl by být přizpůsoben pro konkrétní výpočetní systém, na kterém bude spouštěn. Při návrhu musíme vzít v úvahu typ architektury paralelního počítače, počet a rychlost procesorů a výkonnost meziprocesorové komunikace.

6.2 Obecné schéma paralelního řešiče

Úlohy budeme řešit na počítači IBM SP. Jedná se o symetrický multiprocesorový systém s osmi RISC procesory, které jsou propojeny vysokorychlostní sběrnicí. Jedná se tedy o výkonný paralelní počítač s distribuovanou pamětí a s relativně malým počtem procesorů. Pokud porovnáme výkon jednotlivých procesorů a rychlost meziprocesorové komunikace, ukáže se, že cena komunikace mezi procesory je poměrně vysoká. V našem návrhu se proto nebudeme bránit replikaci výpočtů, pokud se tím sníží počet předávaných zpráv. Budeme se také snažit rozdělit problém (1) na skupinu relativně nezávislých úloh. Rodičovský proces vytvoří q dětských procesů. i-tý dětský proces bude řešit i-tou soustavu systému.

Obecné schéma paralelního řešiče je následující:

Rodičovský proces:

1. *Načti parametry úlohy, vytvoř q dětských procesů*
2. *for i = 1:q*
   Pošli i-tému dětskému procesu parametry úlohy
   end.
3. *for i = 1:q*
   Přijmi od i-tého dětského procesu řešení soustavy x^i
   end.
4. *Konec*

***i*-tý dětský proces:**

1. *Přijmi od rodičovského procesu parametry a data úlohy*
2. *Vyřeš soustavu* $x^i = Kf^i$
3. *Pošli rodičovskému procesu řešení* x^i
4. *Konec*

6.3 Principy meziprocesové komunikace

Procesy, které jsou spuštěny na paralelním počítači s distribuovanou pamětí, spolu mohou komunikovat pouze prostřednictvím předávaných zpráv. Potřebujeme tedy zavést jazyk, který by tuto aktivitu popisoval. Jedna z možností je použít *send / receive* notaci:

send({*matice*} , {*cíl*})
recv({*matice*} , {*originál*})

Jestliže např. *i*-tý proces provede **send**(*A*, *j*), potom tato instrukce zkopíruje matici *A*, která je uložena na *i*-tém procesu a pošle ji *j*-tému procesu. Matice může být samozřejmě řádu 1×1 nebo $n \times 1$. V těchto případech se posílá skalár nebo vektor.

Jestliže *j*-tý proces provede instrukci **recv**(*A, i*), potom je vykonávání programu v tomto procesu zastaveno do té doby, dokud tento proces nepřijme matici *A* od *i*-tého procesu.

V následujících kapitolách se budeme zabývat druhým krokem dětského procesu, tedy samotným řešením soustav. Nejdříve uvedeme sekvenční verze, z nich potom odvodíme paralelní verze algoritmů BCG, SCG a SBCG. Komunikační protokol se bude držet zásady, že proces pošle data dalším procesům jak jen to bude možné a bude je přijímat až v okamžiku, kdy je skutečně potřebuje.

6.4 Návrh paralelizace metody BCG

Sekvenční verze BCG metody

Implementace BCG metody v syntaxi jazyka Matlab je uvedena níže. Vektor *m* obsahuje množinu indexů „master“ soustav, ve které jsou pořadová čísla nevyřešených soustav. Při inicializaci obsahuje vektor *m* pořadová čísla všech soustav.

Funkce *nrmtst* provádí kontrolu relativní přesnosti řešení a aktualizuje vektor *m*. V případě, že je soustava s požadovanou přesností vyřešena, je její

index z této množiny vyjmut. Je-li množina *m* prázdná, všechny soustavy jsou vyřešeny a BCG algoritmus se ukončí.

```
function X = bcg( K, F, repsil, coef )
                              /* repsil - rel. přesnost řešení */
k = 0; Kpoper = 0; new = 1
q = size(F,2)                 /* Zjištění počtu pravých stran */
m = 1:q                       /* Master obsahuje indexy všech pravých stran*/

M = precond(K)                /* Výpočet předpodmínění */
X = zeros(size(F)); R = F; P = R

for i = 1:q,                  /* Výpočet epsil */
    epsil(i) = repsil * norm(R(:,i))
end

while 1
    Z(:,m) = M \ R(:,m)
    ZR(m,m) = Z(:,m)' * R(:,m)
    k = k + 1
    if new == 1
    new = 0
        P(:,m) = Z(:,m)
    else
        beta = ZRold(m,m)' \ ZR(m,m)
        P(:,m) = Z(:,m) + oldP(:,m) * beta
    end
    U(:,m) = K * P(:,m)
    Kpoper = Kpoper + size(m,2)
    UP(m,m) = U(:,m)' * P(:,m)
    alfa = UP(m,m)' \ ZR(m,m)
    X(:,m) = X(:,m) + P(:,m) * alfa
    R(:,m) = R(:,m) - U(:,m) * alfa
    m = nrmtst( R(:,m), m, epsil(m))
                              /* Odstraň indexy vyřešených soustav */
    if size(m,2)== 0,         /* Pokud je master prázdný, konec */
        break
    end
    ZRold(m,m) = ZR(m,m)
    Pold(:,m) = P(:,m)
end
```

Paralelní verze BCG metody

Pokud vyjdeme ze sekvenční verze BCG algoritmu, dostaneme následující paralelní verzi:

```
function X = bcg_palg( K, F, repsil)

k = 0; Kpoper = 0; new = 1
q = size(F,2); m = 1:q
M = precond(K); X = zeros(size(F)); R = F; P = R
for i = 1:q
    epsil(i) = repsil * norm(R(:,i))
end

while 1
    for i = m
        Z(:,i) = M \ R(:,i)                                  (1)
    end
    for i = m
        ZR(m,i) = Z(:,m)' * R(:,i)                           (2)
    end

    k = k + 1
    if new == 1                                              (3)
    new = 0
    for i = m
        P(:,i) = Z(:,i)
        end
    else
        for i = m
            beta = ZRold(m,m)' \ ZR(m,i)
            P(:,i) = Z(:,i) + oldP(:,m) * beta
        end
    end
      for i = m
        U(:,i) = K * P(:,i)                                  (4)
    end
    Kpoper = Kpoper + size(m,2)
    for i = m
        UP(m,i) = U(:,m)' * P(:,i)                           (5)
    end
    for i = m
        alfa = UP(m,m)' \ ZR(m,i)                            (6), (7)
        X(:,i) = X(:,i) + P(:,m) * alfa
```

```
            R(:,i) = R(:,i) - U(:,m) * alfa
        end
        m = nrmtst( R(:,i), m, epsil(i))                    (8)
        if size(m,2) == 0,
            break
        end
        for i = m
            ZRold(:,i) = ZR(:,i)                            (9)
            Pold(:,i) = P(:,i)
        end
    end
```

Z principu BCG algoritmu plyne, že všechny dětské procesy jsou si rovnocenné. Následující tabulka ukazuje data, která jsou dostupná na *i*-tém dětském procesu a data, která jsou nutná k výpočtu řešení.

Nutná data	**Dostupná data**
***i*-tý proces**	***i*-tý proces**
Z(:,m)	*Z(:,i)*
ZR([m,i],i)	*ZR(i,i)*
P(:,m)	*P(:,i)*
U(:,m)	*U(:,i)*
UP(m,m)	*UP(i,i)*
X(:,i)	*X(:,i)*
R(:,i)	*R(:,i)*

Tabulka 6.4.1. Přehled nutných a dostupných dat v i-tém procesu BCG metody

Obsahy sloupců tabulky jsou různé. Je zřejmé, že získání všech potřebných dat se neobejde bez meziprocesové komunikace. Seznam a parametry všech redukcí prováděných v BCG metodě jsou uvedeny v následující tabulce. Zkratka comp znamená výpočet.

Redukce	***i*-tý proces**	**Velikost zprávy**
Z(:,m)	***send**(Z(:,i), [m-i])* ***recv**(Z(:,m-i), [m-i])*	*(M*-1)*n* *(M*-1)*n*
P(:,m), *U(:,m)*	obdobně jako *Z(:,m)*	
ZR(m,m)	*comp ZR(i,i)* *redukce Z(:,m)* *comp ZR(m-i,i)*	

	send*(ZR(m,i),[m-i])* ***recv****(ZR(m,m-i),[m-i])*	*(M*-1*)M* *(M*-1*)M*
UP(m,m)	obdobně jako *ZR(m,m)*	

Tabulka 6.4.2. Seznam redukcí prováděných v BCG metodě

Redukce matice *Z(:,m)*

i-tý proces ($i \subset m$) vypočítá vektor *Z(:,i)* a rozešle ho všem zbývajícím procesům. Potom přijme od zbývajících procesů příspěvky *Z(:,m-i)*.

Redukce matice *ZR(m,m)*

Po provedení redukce *Z(:,m)* má již každý proces tuto matici, a proto může vypočítat součin *ZR(m,i) = Z(:,m)´* R(:,i)*. Výsledek tohoto součinu je vektor *ZR(m,i)*. *i*-tý proces jej následně rozešle všem zbývajícím master procesům. Poté tento proces přijme od zbývajících master procesů matici *ZR(m,m-i)*.

Obdobně se postupuje i při redukci matice *UP(m,m)*.

Analýza komunikačního modelu

Pokud analyzujeme komunikační model, zjistíme, že zprávy mají některé společné parametry, viz tabulka 6.4.3. Pokud je odesílatelem zprávy *i*-tý proces, potom příjemci této zprávy jsou všechny zbývající procesy *(m - i)*.

Odesílatel	**Počet odesílatelů**	**Příjemce**	**Počet příjemců**
i-tý proces, $i \subset m$	*M*	*m - i*	*M* - 1

Tabulka 6.4.3. Společné parametry zpráv

Počet zpráv

Zabývejme se nyní počtem zpráv, který musí *i*-tý proces rozeslat ($i \subset m$), aby se provedla redukce matice *Z(:,m)*. Každý proces rozešle svůj příspěvek matice *Z* všem zbývajícím procesům - těch je *M* - 1. Celkový počet odeslaných zpráv pro jeden proces je tedy *M* - 1. Obdobný výsledek platí i pro počet přijímaných zpráv. Každý proces dostane od zbývajících procesorů zprávu s jejich příspěvkem *Z*, celkem tedy přijme *M* - 1 zpráv.

Počet zpráv	***i*-tý proces**
odeslaných	*M* - 1
přijatých	*M* - 1

Tabulka 6.4.4 a) Počet zpráv odeslaných a přijatých jedním procesem

Počet zpráv	***i*-tý proces**
odeslaných	*M (M* - 1*)*
přijatých	*M (M* - 1*)*

Tabulka 6.4.4 b) Počet zpráv odeslaných a přijatých všemi procesy

Celkový počet odeslaných (respektive přijatých) zpráv v systému je roven součtu odeslaných (respektive přijatých) zpráv všemi procesory.

Pro BCG algoritmus je tento počet rovný M^2 - *M*.

Stejné výsledky pochopitelně obdržíme i při redukci *P(:,m), U(:,m), ZR(m,m)* a *UP(m,m).*

Komunikační protokol metody BCG

Komunikační protokol metody BCG je uveden níže. Příspěvky *ZR(m,i)* a *UP(m,i)* jsou počítány průběžně a nečeká se tedy na zaslání všech složek vektorů *Z(:,m), P(:,m),* respektive *U(:,m).*

1) Výpočet *Z(:,m) = M \ R(:,m)*

Master	**Poznámka**
Z(:,i) = M \ R(:,i)	Řešení předpodmínění

2) Výpočet *ZR(m, m) = Z(:,m)' * R(:,m),* redukce *ZR(m,[m,i])*

Master	**Poznámka**
send(Z(:,i), [m - i])	Rozešli vektor *Z(:,i)*
*ZR(i,i) = Z(:,i)' * R(:,i)*	Vypočítej *ZR(i,i)*
for j = m - i *recv(Z(:,j), j)* *ZR(j,i)= Z(:,j)' * R(:,i)* *end*	Přijmi od zbývajících procesů *Z(:,m-i)* Vypočítej *ZR(m-i,i)*
send(ZR(m, i), [m - i])	Rozešli vektor *ZR(m,i)*
for j = m - i *recv(ZR(m, j), j)* *end*	Přijmi od zbývajících procesů *ZR(m,j)*

3) Výpočet *P(:,m)*

Master	**Poznámka**
if new == 1 *new = 0* *P(:,i) = Z(:,i)* *else*	

beta = Zrold(m,m)' \ ZR(m,i) *P(:,i)=Z(:,i) + oldP(:,m)*beta* *end*	Vypočítej koeficient *beta* a *i*-tý sloupec matice *P*

4) Výpočet *U(:,m) = K * P(:,m)*

Master	**Poznámka**
*U(:,i) = K * P(:,i)*	Vypočítej *i*-tý sloupec matice *U*

5) Výpočet *UP(m,i) = U(:,m)' * P(:,i)*, redukce *UP(m,m)*

Master	**Poznámka**
send(U(:,i), [m - i])	Rozešli vektor *U(:,i)*
*UP(i,i) = U(:,i)' * P(:,i)*	Vypočítej *UP(i,i)*
for j = m - i *recv(U(:,j), j)* *UP(j,i)= U(:,j)' *P(:,i)* *end*	Přijmi od zbývajících procesů *U(:,m-i)* Vypočítej *UP(m-i,i)*
send(UP(m, i), [m - i])	Rozešli *UP(m,i)*
for j = m - i *recv(UP(m, j), j)* *end*	Přijmi od masteru *UP(m,j)*

6) Rozeslání *P(:,m)* (nutný pro výpočet *X(:,i),* $i \subset m$)

Master	**Poznámka**
send(P(:,i), [m - i])	Rozešli *P(:,i)*

7) Výpočet *alfa*, *R(:,m)*, příjmutí *P(:,m)*, výpočet *X(:,[m,s])*

Master	**Poznámka**
alfa = UP(m,m)' \ ZR(m,i)	
*R(:,i) = R(:,i) - U(:,m) * alfa*	
for j = m - i *recv(P(:,j), j)* *end*	Přijmi *P(:,m-i)*
*X(:,i) = X(:,i) + P(:,m) * alfa*	

8) Testování přesnosti řešení

Master	**Poznámka**
m = nrmtst(R(:,i), m, epsil(i))	Při vyřešení „své“ soustavy informuj zbývající procesy a ukonči se.

9) Aktualizace *ZRold(m,m)* a *Pold(:,m)* pro novou iteraci BCG

Master	**Poznámka**
ZRold(:,i) = ZR(:,i) *Pold(:,i) = P(:,i)*	

6.5 Návrh paralelizace metody SCG

Sekvenční verze metody SCG

Implementace SCG metody v syntaxi jazyka Matlab je uvedena níže.

Z principu SCG algoritmu plyne, že master soustava je pouze jedna. Její pořadové číslo je zapsáno ve vektoru *m*. Jelikož je počet master soustav roven jedné, vektor *m* obsahuje jen jedno číslo. Vektor *s* obsahuje pořadová čísla všech slave soustav.

Funkce *nrmtst* provádí kontrolu relativní přesnosti řešení master soustavy a všech slave soustav.

Popišme si nyní, jak se může během iteračního cyklu měnit obsah proměnné *m* a vektoru *s*. Při inicializaci SCG metody obsahuje proměnná *m* index první soustavy, indexy všech zbylých soustav jsou uloženy ve vektoru *s*. Soustava může být vyjmuta z master systému pouze tehdy, je-li vyřešena.

Soustava může být vyjmuta ze slave systému při splnění jedné z následujících podmínek:

- soustava je vyřešena
- master systém je prázdný. V tomto případě se přesune jeden index slave soustavy do master soustavy.

V případě, že jsou obě množiny indexů prázdné, jsou všechny soustavy vyřešeny a SCG algoritmus se ukončí.

```
function X = scg( K, F, repsil)
k = 0; Kpoper = 0; new = 1
q = size(F,2)
m = 1                /* Master obsahuje index první pravé strany */
s = 2:q              /* Slave obsahuje indexy zbývajících pravých stran */

M = precond(K)
X = zeros(size(F)); R = F; p = R(:,m)

for i = 1:q,         /* výpočet epsil*/
    epsil(i) = repsil * norm(R(:,i))
end

while 1
```

```
        z = M \ R(:,m)
        zR([m,s]) = z' * R(:,[m,s])
        k = k + 1
        if new == 1
        new = 0
                p = z
        else
                beta = zR(m) / zRoldm
                p = z + beta * p
        end
        u = K * p
        Kpoper = Kpoper + 1
        up= u' * p
        alfa = zR([ma,sl]) / up
        X(:,[m,s]) = X(:,[m,s]) + p * alfa
        R(:,[m,s]) = R(:,[m,s]) - u * alfa

        m = nrmtst( R(:,m), m, epsil(m))
        s = nrmtst( R(:,s), s, epsil(s))

        if size(m,2)== 0,       /* Pokud je master prázdný, přesuň 1. index ze
slave do master */
                if size(s,2)~= 0, m = s(1); s(1) = [ ]; new = 1
        else                    /* Pokud je prázdný i slave, konec */
                break
                end
        end

        zRoldm=zR(m)
    end
```

Paralelní verze metody SCG

Pokud vyjdeme ze sekvenční verze SCG algoritmu, dostaneme následující paralelní verzi:

```
    function X = scg_palg( K, F, repsil)

    k = 0; Kpoper = 0; new = 1
    q = size(F,2); m = 1; s = 2:q
    M = precond(K); X = zeros(size(F)); R = F; p = R(:,m)
    for i = 1:q
        epsil(i) = repsil * norm(R(:,i))
```

```
end

while 1
    z = M \ R(:,m)                                          (1)
    for i = [m,s]
        zR(i) = z' * R(:,i)                                 (2)
    end
    k = k + 1
    if new == 1                                             (3)
    new = 0
        p = z
    else
        beta = zR(m) / zRoldm
        p = z + beta * p
    end
    u = K * p                                               (4)
    Kpoper = Kpoper + 1
    up= u' * p                                              (5, 6)
    for i = [m,s]
        alfa = zR(i) / up                                   (7)
        X(:,i) = X(:,i) + p * alfa
        R(:,i) = R(:,i) - u * alfa
    end
    [m,s] = nrmtst( R(:,i), [m,s], epsil(i))                (8)
    if size(m,2) == 0,
        if size(s,2) ~= 0, m = s(1); s(1) = [ ]; new = 1
    else
         break
        end
    end
    zRoldm=zR(m)                                            (9)
end
```

Z principu SCG algoritmu plyne, že dětské procesy se dělí na master a slave procesy. Master proces je vždy pouze jeden. Následující tabulka ukazuje data, která jsou dostupná na *i*-tém dětském procesu a data, která jsou nutná k výpočtu řešení.

Nutná data	**Dostupná data**	
***i*: [Master, Slave]**	***i*: Master**	***i*: Slave**
z *zR(i)*	*z* *zR(i)*	

p *u* *up* *X(:,i)* *R(:,i)*	*p* *u* *up* *X(:,i)* *R(:,i)*	 *X(:,i)* *R(:,i)*

Tabulka 6.5.1. Závislost dostupnosti dat na typu dětského procesu

Obsahy prvních dvou sloupců tabulky jsou stejné. Master proces nepotřebuje od slave procesů žádná data. Jiná situace je u slave procesu. Slave proces potřebuje k výpočtu vektory *z*, *p*, a *u*. Tato data si tedy musí vyžádat od master procesu. Seznam a parametry všech redukcí prováděných v SCG metodě jsou uvedeny v následující tabulce.

Redukce	***i*: Master proces**	**Velikost zprávy**	***i*: Slave proces**	**Velikost zprávy**
z	***send****(z, s)*	*Sn*	***recv****(z, m)*	*n*
p *u*	obdobně jako vektor *z*		obdobně jako vektor *z*	
zR(i)	*comp zR(i)* *redukce z*		*redukce z* *comp zR(i)*	
up	*comp up* ***send****(up, s)*	 *S**1	 ***recv****(up, m)*	 1

Tabulka 6.5.2. Seznam redukcí prováděných v SCG metodě

Redukce vektoru *z*

Popišme si, jak bude probíhat komunikace při redukci vektoru *z*. Komunikace bude záviset na tom, zda daný proces je master nebo slave.

Komunikace master procesu

Předpokládejme, že *i*-tý proces je master. Pak tento proces vypočítá vektor *z* a rozešle ho všem slave procesům.

Komunikace slave procesu

Z principu SCG algoritmu plyne, že slave proces nepočítá vektor *z*. Proto slave proces nic neposílá, nýbrž tento vektor přijme od master procesu.

Redukce *zR(i)*

Po provedení redukce vektoru *z* mají již všechny procesy tento vektor, a proto mohou vypočítat součin $zR(i) = z' * R(:,i)$.

Obdobně můžeme postupovat i při redukci *up*. Jediný rozdíl je v tom, že slave procesy nic nepočítají a přijmou od master procesu celý koeficient *up*.

Další možností je, že každý slave proces si vypočítá koeficient *up* sám. (Replikace výpočtů.)

Analýza komunikačního modelu

Pokud analyzujeme komunikační model, zjistíme, že zprávy mají některé společné parametry. Odesílatelem zprávy je vždy master proces, příjemci této zprávy jsou všechny slave procesy.

Odesílatel	Počet odesílatelů	Příjemce	Počet příjemců
m	1	*s*	*S*

Tabulka 6.5.3. Společné parametry zpráv

Počet zpráv

Zabývejme se nyní počtem zpráv, který musí každý proces rozeslat, aby se provedla redukce dat z tabulky 6.5.2. Počet zpráv závisí na tom, zda daný proces je master nebo slave.

V případě, že se jedná o master proces, rozešle tento proces vektor *z* všem slaveům - těch je *S*. Celkový počet zpráv, které rozešle master proces, je tedy *S*.

Slave procesy nerozesílají žádné zprávy.

Počet přijatých zpráv také závisí na typu procesu. Master proces nepřijímá žádné zprávy. Slave proces přijímá data od master procesu. Protože v SCG metodě je jen jeden master proces, přijme každý slave proces právě jednu zprávu. Celkový počet přijatých a odeslaných zpráv pro jeden a dále pro všechny procesy, je uveden v následujících tabulkách.

Počet zpráv	Master	Slave
odeslaných	*S*	0
přijatých	0	1

Tabulka 6.5.4a) Počet zpráv odeslaných a přijatých jedním procesem

Počet zpráv	Master	Slave
odeslaných	$1 \times S$	0
přijatých	0	$S \times 1$

Tabulka 6.5.4b) Počet zpráv odeslaných a přijatých všemi procesy

Celkový počet odeslaných (respektive přijatých) zpráv v systému je roven součtu odeslaných (respektive přijatých) zpráv master procesem a všemi slave procesy.

Pro SCG algoritmus je tento počet rovný počtu slave soustav *S*.

Stejné výsledky pochopitelně obdržíme i při redukci *p*, *u*, *zR(i)* a *up* - viz tabulka 6.5.2.

Komunikační protokol SCG

Komunikační protokol SCG metody je uveden níže.

1) Výpočet *z = M \ R(:,m)*

Master	**Slave**	**Poznámka**
z = M \ R(:,i)		m: Řešení předpodmínění

2) Výpočet *zR(i)*

Master	**Slave**	**Poznámka**
send(z, s)		m: Rozešli vektor *z*
*zR(i) = z' * R(:,i)*		m: Vypočítej *zR(i)*
	recv(z, m) *zR(i) = z' * R(:,i)*	s: Přijmi od master procesu vektor *z* s: Vypočítej *zR(i)*

3) Výpočet vektoru *p*

Master	**Slave**	**Poznámka**
if new == 1 *new = 0* *p = z* *else* *beta = zR(m) / zRoldm* *p = z + beta * p* *end*		m: Vypočítej koeficient *beta* a vektor *p*

4) Výpočet *u = K * p*

Master	**Slave**	**Poznámka**
*u = K * p*		Slave nepotřebují matici *K*

5) Výpočet koeficientu *up = u' * p*, rozeslání *up* a vektoru *u*

Master	**Slave**	**Poznámka**
send(u, s)		m: Rozešli vektor *u*
*up = u' * p*		m: Vypočítej *up*
send(up, s)		m: Rozešli *up*
	recv(up, m)	s: Přijmi od master procesu koeficient *up*

6) Rozeslání vektoru *p* (nutný pro výpočet *X(:,i), $i \subset s$*)

Master	**Slave**	**Poznámka**
send(p, s)		m: Rozešli vektor *p*

7) Výpočet koeficientu *alfa*, *R(:,[m,s])*, přijetí vektoru *p*, výpočet *X(:,[m,s])*

Master	**Slave**	**Poznámka**
alfa = zR(i) / up	*alfa = zR(i) / up*	
	recv(u, m)	s: Přijmi vektor *u*
*R(:,i) = R(:,i) - u * alfa*	*R(:,i) = R(:,i) - u * alfa*	
	recv(p, m)	s: Přijmi vektor *p*
*X(:,i) = X(:,i) + p * alfa*	*X(:,i) = X(:,i) + p * alfa*	

8) Testování přesnosti řešení, nastavení nové master soustavy

Master	**Slave**	**Poznámka**
m = nrmtst(R(:,i), m, epsil(i))	*s = nrmtst(R(:,i), s, epsil(i))*	
Při vyřešení „své“ soustavy informuj zbývající procesy a ukonči se	*Při vyřešení „své“ soustavy informuj zbývající procesy a ukonči se*	
	if size(m,2)==0 *if size(s,2) ~= 0,* *m = s(1)* *s(1) = []* *new = 1* *else* *break* *end*	Pokud je master i slave prázdný, ukonči se; jinak přesuň slave soustavu *s(1)* do master procesu.

9) Aktualizace koeficientu *zRoldm* pro novou iteraci

Master	**Slave**	**Poznámka**
zRoldm=zR(i)		

6.6 Návrh paralelizace metody SBCG

Sekvenční verze metody SBCG

Implementace SBCG metody v syntaxi jazyka Matlab je uvedena níže. Vektor *m* obsahuje pořadová čísla master soustav. Vektor *s* obsahuje pořadová čísla slave soustav.

Funkce *findep* provádí kontrolu lineární závislosti master soustav. Závislé soustavy se přesouvají z vektoru *s* do vektoru *m*.

Funkce *nrmtst* provádí kontrolu relativní přesnosti řešení master a slave soustav. Aktualizuje tedy také vektory *m* a *s*.

Popišme si nyní, jak se může během iteračního cyklu měnit obsah vektorů *m* a *s*. Při inicializaci SBCG metody obsahuje vektor *m* pořadová čísla všech soustav, vektor *s* je prázdný. Soustava může být vyjmuta z master systému při splnění jedné z následujících podmínek

- soustava je vyřešena
- soustava je lineárně závislá na jiné soustavě master systému. Pak je tato závislá soustava přesunuta do vektoru *s*.

Soustava může být vyjmuta ze slave systému při splnění jedné z následujících podmínek:

- soustava je vyřešena
- master systém je prázdný. V tomto případě se přesune jedna soustava ze slave systému do master systému.

(Tyto podmínky se shodují s SCG algoritmem.)

V případě, že jsou obě množiny indexů prázdné, jsou všechny soustavy vyřešeny a SBCG algoritmus se ukončí.

```
function X = sbcg( K, F, repsil, coef )
            /* coef - koeficient lineární závislosti pro findep */
k = 0; Kpoper = 0; new = 1
q = size(F,2)
m = 1:q     /* Master obsahuje indexy všech pravých stran */
s = [ ]             /* Slave je prázdný */

M = precond(K)
X = zeros(size(F)); R = F; P = R

for i = 1:q,
    epsil(i) = repsil * norm(R(:,i))
end

while 1
    Z(:,m) = M \ R(:,m)
    ZR(m,[m,s]) = Z(:,m)' * R(:,[m,s])
    [m,s] = findep( coef, ZR, m, s)
       /* Test hodnosti matice ZR(m,m), aktualizace [m,s] - přesunutí lin.
    závislých indexů z master do slave */
    k = k + 1
    if new == 1
    new = 0
            P(:,m) = Z(:,m)
    else
            beta = ZRold(m,m)' \ ZR(m,m)
            P(:,m) = Z(:,m) + Pold(:,m) * beta
    end
    U(:,m) = K * P(:,m)
    Kpoper = Kpoper + size(m,2)
```

```
        UP(m,m) = U(:,m)' * P(:,m)
        alfa = UP(m,m)' \ ZR(m,[m,s])
        X(:,[m,s]) = X(:,[m,s]) + P(:,m) * alfa
        R(:,[m,s]) = R(:,[m,s]) - U(:,m) * alfa
        m = nrmtst( R(:,m), m, epsil(m))
        s = nrmtst( R(:,s), s, epsil(s))
           if size(m,2)== 0,   /* Pokud je master prázdný, přesuň 1. index ze
        slave do master*/
                if size(s,2)~= 0, m = s(1); s(1) = [ ]; new = 1
        else                        /* Pokud je prázdný i slave, konec */
                 break
                end
        end
        ZRold(m,m) = ZR(m,m)
        Pold(:,m) = P(:,m)
    end
```

Paralelní verze SBCG

Pokud vyjdeme ze sekvenční verze SBCG algoritmu, dostaneme následující paralelní verzi:

```
    function X = sbcg_palg( K, F, repsil, coef)

    k = 0; Kpoper = 0; new = 1
    q = size(F,2); m = 1:q; s = [ ]
    M = precond(K); X = zeros(size(F)); R = F; P = R
    for i = 1:q
        epsil(i) = repsil * norm(R(:,i))
    end

    while 1
        for i = m
                Z(:,i) = M \ R(:,i)                                    (1)
        end
        for i = [m,s]
                ZR(m,i) = Z(:,m)' * R(:,i)                             (2)
        end
        [ m, s ] = findep( coef, ZR, m, s)                             (3)
        k = k + 1
        if new == 1                                                    (4)
        new = 0
        for i = m
```

```
            P(:,i) = Z(:,i)
            end
        else
            for i = m
                beta = ZRold(m,m)' \ ZR(m,i)
                P(:,i) = Z(:,i) + oldP(:,m) * beta
            end
        end
          for i = m
            U(:,i) = K * P(:,i)                                    (5)
        end
        Kpoper = Kpoper + size(m,2)
        for i = m
            UP(m,i) = U(:,m)' * P(:,i)                             (6)
        end
        for i = [m,s]
            alfa = UP(m,m)' \ ZR(m,i)                              (7), (8)
            X(:,i) = X(:,i) + P(:,m) * alfa
            R(:,i) = R(:,i) - U(:,m) * alfa
        end
        [m,s] = nrmtst( R(:,i), [m,s], epsil(i))                   (9)
        if size(m,2) == 0,
            if size(s,2) ~= 0, m = s(1); s(1) = [ ]; new = 1
        else
            break
            end
        end
        for i = m
            ZRold(:,i) = ZR(:,i)                                   (10)
            Pold(:,i) = P(:,i)
        end
    end
```

Z principu SBCG algoritmu plyne, že dětské procesy se dělí na master a slave. Při inicializaci jsou všechny dětské procesy master. Následující tabulka ukazuje data, která jsou dostupná na *i*-tém dětském procesu a data, která jsou nutná k výpočtu řešení.

Nutná data	**Dostupná data**	
***i*: [Master, Slave]**	***i*: Master**	***i*: Slave**
Z(:,m) *ZR([m,i],i)*	*Z(:,i)* *ZR(i,i)*	

P(:,m)	*P(:,i)*	
U(:,m)	*U(:,i)*	
UP(m,m)	*UP(i,i)*	
X(:,i)	*X(:,i)*	*X(:,i)*
R(:,i)	*R(:,i)*	*R(:,i)*

Tabulka 6.6.1. Závislost dostupnosti dat na typu dětského procesu

Obsahy sloupců tabulky jsou různé. Je zřejmé, že získání všech potřebných dat se neobejde bez meziprocesové komunikace. Seznam a parametry všech redukcí prováděných v SBCG metodě jsou uvedeny v následující tabulce.

Redukce	***i*: Master proces**	**Velikost zprávy**	***i*: Slave proces**	**Velikost zprávy**
Z(:,m)	***send**(Z(:,i), [m-i,s])* ***recv**(Z(:,m-i), [m-i])*	*(q-1)n* *(M-1)n*	***recv**(Z(:,m), m)*	*Mn*
P(:,m), *U(:,m)*	obdobně jako *Z(:,m)*		obdobně jako *Z(:,m)*	
ZR([m,i], *m)*	*comp ZR(i,i)* *redukce Z(:,m)* *comp ZR(m-i,i)* ***send**(ZR(m,i),[m-i,s])* ***recv**(ZR(m,m-i),[m-i])*	 *(q-1)M* *(M-1)M*	*redukce Z(:,m)* *comp ZR(m,i)* ***recv**(ZR(m ,m), m)*	 M^2
UP(m,m)	*comp UP(i,i)* *redukce U(:,m)* *comp UP(m-i,i)* ***send**(UP(m,i),[m-i,s])* ***recv**(UP(m,m-i),m-i])*	 *(q-1)M* *(M-1)M*	 ***recv**(UP(m ,m), m)*	 M^2

Tabulka 6.6.2. Seznam redukcí prováděných v SBCG metodě

Redukce matice *Z(:,m)*

Komunikace bude v tomto případě záviset na tom, zda daný proces je master nebo slave.

Komunikace master procesu

i-tý master proces vypočítá vektor *Z(:,i)* a rozešle ho všem slave a zbývajícím master procesům. Potom přijme od zbývajících master procesů příspěvky *Z(:,m-i)*.

Komunikace slave procesu

Z principu SBCG algoritmu plyne, že slave proces nepočítá vektor *Z(:,i)*. Proto slave proces nic neposílá, ale jen přijme od master soustav matici *Z(:,m)*.

Redukce matice *ZR(m,[m,i])*

Po provedení redukce *Z(:,m)* má již každý proces tuto matici a proto může vypočítat součin *ZR(m,i) = Z(:,m)´* R(:,i)*. Další postup opět závisí na typu procesu.

Komunikace master procesu

i-tý master proces rozešle svůj vypočtený příspěvek *ZR(m,i)* všem slave a zbývajícím master procesům. Potom přijme od zbývajících master procesů příspěvky *ZR(m,m-i)*.

Komunikace slave procesu

Slave procesy nerozesílají žádné zprávy, pouze přijmou od master soustav matici *ZR(m,m)*.

Obdobně postupujeme i při redukci matice *UP(m,m)*. Jediný rozdíl je v tom, že slave procesy nic nepočítají a přijmou od master procesů celou matici *UP(m,m)*.

Analýza komunikačního modelu

Pokud analyzujeme komunikační model, zjistíme, že zprávy mají některé společné parametry. Odesílatelem zprávy je vždy master proces, příjemce této zprávy jsou všechny zbývající procesy.

Odesílatel	Počet odesílatelů	Příjemce	Počet příjemců
$i \subset m$	M	$m\text{-}i,\ s$	$M - 1 + S$

Tabulka 6.6.3. Společné parametry zpráv

Počet zpráv

Zabývejme se nyní počtem zpráv, který musí každý proces rozeslat, aby se provedla redukce matice *Z(:,m)*. Počet zpráv zase závisí na tom, zda daný proces je master nebo slave.

V případě, že se jedná o master proces, rozešle svůj příspěvek matice *Z* všem slaveům - těch je *S* a všem zbývajícím master procesům - těch je *M* - 1. Celkový počet odeslaných zpráv pro jeden master proces je tedy *S* + *M* - 1.

Slave procesy nerozesílají žádné zprávy.

Počet přijímaných zpráv také závisí na typu procesu. Master proces dostane od každé zbývající master soustavy jednu zprávu. Celkem přijme *M* - 1 zpráv. Slave proces přijímá zprávy od všech master soustav, celkem přijme *M* zpráv. Celkový počet přijatých a odeslaných zpráv pro jeden a pro všechny procesy je uveden v následující tabulce.

Počet zpráv	Master	Slave
odeslaných	M - 1 + S	0
přijatých	M - 1	M

Tabulka 6.6.4a) Počet zpráv odeslaných a přijatých jedním procesem

Počet zpráv	Master	Slave
odeslaných	$M (M - 1 + S)$	0
přijatých	$M (M - 1)$	$S M$

Tabulka 6.6.4b) Počet zpráv odeslaných a přijatých všemi procesy

Celkový počet odeslaných (a přijatých) zpráv v systému je roven součtu odeslaných (a přijatých) zpráv všemi master a všemi slave procesy.

Pro SBCG algoritmus je tento počet rovný $M^2 + MS - M$.

Stejné výsledky pochopitelně obdržíme i při redukcích *P(:,m), U(:,m), ZR(m,m)* a *UP(m,m)* - viz tabulka 6.6.2.

Komunikační protokol SBCG

Komunikační protokol SBCG metody je uveden níže. Příspěvky *ZR(m,i)* a *UP(m,i)* jsou počítány průběžně a nečeká se tedy na zaslání všech složek vektorů *Z(:,m), P(:,m)* respektive *U(:,m).*

1) Výpočet *Z(:,m) = M \ R(:,m)*

Master	Slave	Poznámka
Z(:,i) = M \ R(:,i)		m: Řešení předpodmínění

2) Výpočet *ZR(m,[m,s]) = Z(:,m)' * R(:,[m,s])*

Master	Slave	Poznámka
send(Z(:,i), [m - i, s])		m: Rozešli svůj *Z(:,i)*
*ZR(i,i) = Z(:,i)' * R(:,i)*		m: Vypočítej *ZR(i,i)*
for j = m - i *recv(Z(:,j), j)* *ZR(j,i)= Z(:,j)' * R(:,i)* *end*	*for j = m* *recv(Z(:,j), j)* *ZR(j,i)= Z(:,j)' * R(:,i)* *end*	m: Přijmi od zbývajících master procesů *Z(:,m-i)* m: Vypočítej *ZR(m-i,i)* s: Přijmi od master procesů *Z(:,m)* s: Vypočítej *ZR(m,i)*
send(ZR(m, i), [m,s])		m: Rozešli *ZR(m,i)*
for j = m - i *recv(ZR(m, j), j)* *end*	*for j = m* *recv(ZR(m, j), j)* *end*	Přijmi od master procesů *ZR(m,j)*

3) Test hodnosti matice *ZR(m,m)*

Master	Slave	Poznámka
[m,s] = findep(coef, ZR, m, s)	*[m,s] = findep(coef, ZR, m, s)*	Aktualizace [*m,s*] - přesunutí lineárně závislých soustav z master do slave

4) Výpočet *P(:,m)*

Master	Slave	Poznámka
if new == 1 *new = 0* *P(:,i) = Z(:,i)* *else* *beta = ZRold(m,m)' \ ZR(m,i)* *P(:,i)=Z(:,i) + oldP(:,m)*beta* *end*		m: Vypočítej koeficient *beta* a *i*-tý sloupec matice *P*

5) Výpočet *U(:,m) = K * P(:,m)*

Master	Slave	Poznámka
*U(:,i) = K * P(:,i)*		Slave nepotřebují matici *K*

6) Výpočet *UP(m,i) = U(:,m)' * P(:,i)*, redukce *UP(m,m)*

Master	Slave	Poznámka
send(U(:,i), [m - i, s])		m: Rozešli *U(:,i)*
*UP(i,i) = U(:,i)' * P(:,i)*		m: Vypočítej *UP(i,i)*
for j = m - i *recv(U(:,j), j)* *UP(j,i)= U(:,j)' *P(:,i)* *end*		m: Přijmi od zbývajících master procesů *U(:,m-i)* Vypočítej *UP(m-i,i)*
send(UP(m, i), [m - i, s])		m: Rozešli *UP(m,i)*
for j = m - i *recv(UP(m, j), j)* *end*	*for j = m* *recv(UP(m, j), j)* *end*	Přijmi od master procesů *UP(m,j)*

7) Rozeslání *P(:,m)* (nutný pro výpočet *X(:,i),* $i \subset m, s$)

Master	Slave	Poznámka
send(P(:,i), [m - i, s])		m: Rozešli *P(:,i)*

8) Výpočet *alfa*, *R(:,[m,s])*, přijmutí *P(:,m)*, výpočet *X(:,[m,s])*

Master	Slave	Poznámka
alfa = UP(m,m)' \ ZR(m,i)	*alfa = UP(m,m)' \ ZR(m,i)*	
	for j = m *recv(U(:,j), j)* *end*	s: Přijmi *U(:,m)* (Přijetí *U(:,m)* u master soustav bylo provedeno

		v šestém kroku)
*R(:,i) = R(:,i) - U(:,m) * alfa*	*R(:,i) = R(:,i) - U(:,m) * alfa*	
for j = m - i *recv(P(:,j), j)* *end*	*for j = m* *recv(P(:,j), j)* *end*	m: Přijmi *P(:,m-i)* s: Přijmi *P(:,m)*
*X(:,i) = X(:,i) + P(:,m) * alfa*	*X(:,i) = X(:,i) + P(:,m) * alfa*	

9) Testování přesnosti řešení, nastavení nové master soustavy - pokud je vektor *m* prázdný

Master	**Slave**	**Poznámka**
m = nrmtst(R(:,i), m, epsil(i))	*s = nrmtst(R(:,i), s, epsil(i))*	
Při vyřešení „své“ soustavy informuj zbývající procesy a ukonči se	*Při vyřešení „své“ soustavy informuj zbývající procesy a ukonči se*	
	if size(m,2)==0 *if size(s,2) ~= 0,* *m = s(1)* *s(1) = []* *new = 1* *else* *break* *end*	Pokud je master i slave prázdný, ukonči se; jinak přesuň slave soustavu *s(1)* do master.

10) Aktualizace *ZRold(m,m)* a *Pold(:,m)* pro novou iteraci SBCG

Master	**Slave**	**Poznámka**
ZRold(:,i) = ZR(:,i) *Pold(:,i) = P(:,i)*		

7. Implementace SBCG algoritmu v jazyce C

V této kapitole popíšeme sekvenční a paralelní implementaci SBCG metody v prostředí ANSI C bez předpodmínění. Paralelní verze této metody bude vycházet z obecného paralelního návrhu, který je popsán v kapitole 6.6.

7.1 Implementační strategie

Prostředí systému Matlab se vyznačuje velice lehkou manipulací s daty. Uživatel se nemusí starat o deklaraci proměnných. I velikost vektorů a matic lze snadno měnit. Toho jsme také využili při implementaci SBCG metody v prostředí Matlabu. Zopakujme, že vektory *m* a *s* obsahují pořadová čísla soustav, která jsou v master a slave systémech. Velikosti těchto vektorů se v iteračním cyklu SBCG algoritmu mění.

Z implemetace v Matlabu je vidět, že při maticových operacích vystupují vektory *m* a *s* ve dvou následujících konfiguracích:

- pouze vektor *m* - výpočet prováděný pouze v master soustavách - proces *K*-ortogonalizace - např. výpočet *Z(:,m), P(:,m), UP(m,m)*
- vektor *m* a *s* společně, tedy výpočty, který provádějí nevyřešené soustavy - např. aktualizace reziduí a řešení *X(:,[m,s]), R(:,[m,s]).*

Jediná výjimka je při kontrole přesnosti řešení - funkce *nrmtst*, která se volá dvakrát: nejdříve pro master a pak pro slave soustavy.

Operace sjednocení vektorů *m* a *s* se v Matlabu implementuje jednoduše: [*m,s*]. V prostředí ANSI C je však situace jiná, a proto jsme při implementaci postupovali jinak. Místo vektoru *m* jsme zavedli vektor *master*, který neobsahuje pořadová čísla master soustav, ale pouze hodnoty 0 nebo 1.

Uvažujme soustavu lineárních rovnic s *q* pravými stranami (1), $i \subset < 1; q >$. Pokud *i*-tá soustava tohoto problému náleží do master systému, platí, že *master(i)*=1. V opačném případě je *master(i)*=0.

Dále jsme zavedli vektor *n_s* (*n_s* je zkratka not_solved), který informuje o nevyřešených soustavách. Pokud je *i*-tá soustava problému nevyřešená, platí, že *n_s(i)*=1. V opačném případě je *n_s(i)*=0.

Důležitou vlastností těchto vektorů je, že nemění v průběhu iteračního cyklu svou velikost. Jejich velikost je pevně daná a rovná se počtu pravých stran problému.

Uveďme si vztah mezi vektory *m* a *s* respektive *master* a *n_s* na příkladu.

Příklad

Nechť máme soustavu lineárních rovnic s pěti pravými stranami. Tuto soustavu řešíme SBCG algoritmem. Nechť v průběhu iteračního cyklu metody nastane situace, ve které master systém obsahuje první a třetí soustavu a slave systém druhou a čtvrtou soustavu. (Pátá soustava je již vyřešena.) Vektory *m* a *s* vypadají následovně:

$m = [1,3], \quad s = [2,4].$

K těmto vektorům přísluší následující vektory *master* a *n_s*:

$master = [1, 0, 1, 0, 0], \quad n\_s = [1, 1, 1, 1, 0].$

Toto značení má tu výhodu, že cyklus

for i = [*m,s*]
. . . . .

lze do jazyka C snadno přepsat:

```
for(i=0; i<rhs; i++)   /* v jazyce C indexujeme od nultého indexu */
        if (n_s[i])
                . . . . .
```

Ke zjednodušení dochází i ve funkci *findep()*, která zajišťuje stabilitu SBCG algoritmu. V případě, že je reziduum *i*-té soustavy lineárně závislé, funkce *findep()* provede *master*[*i*]=0 a nemusí řešit přesouvání indexů z vektoru *m* do vektoru *s*. Obsah vektoru *n_s* se voláním *findep()* vůbec nemění.

7.2 Implementace základních vstupně-výstupních funkcí

Funkce, které inicializují základní datové typy a zajišťují základní vstupně-výstupní operace, se nalézají v souboru *matrixio.c*.

Pro vytvoření vektoru délky *size* slouží funkce

```
double *create_vector(int size);
```

a

```
int *create_int(int size);
```

Matici vytváří funkce

```
double **create_matrix(int columns, int rows);
```

Soubor *matrixio.c* implementuje i načítání a ukládání matice do souboru. Struktura datového souboru je v následující tabulce:

Název proměnné	**Typ**	**Délka**	**Popis**
rhs	integer	1	počet sloupců matice
n	integer	1	počet řádků matice
data	double	*rhs * n*	hodnoty uložené po sloupcích

Tabulka 7.2.1 Struktura datového souboru pro uložení matice pravých stran F a matice řešení X.

Pro načítání matice ze souboru *file* slouží funkce

```
double **read_mx(FILE *file,int *rhs, int *n);
```

Poznámka: *n* - počet řádků, *rhs* - počet pravých stran (tj. sloupců matice)

K uložení matice *X* do souboru *file* slouží funkce

```
void write_mx(FILE *file,int rhs, int n, double **X);
```

7.3 Implementace maticových operací

Přehled datových typů a příklady volání funkcí, které jsou použity v C implementaci SBCG metody, jsou uvedeny v následující tabulce.

Název proměnné	Typ	Velikost	Příklad volání v C
K	řídká matice, double	$n \times n$	
P, Pold, U, X, R	matice, double	$n \times q$	*P=create_matrix(q,n);*
ZR, ZRold, UP	matice, double	$q \times q$	
α, β, *epsil*	vektor, double	*q*	*alfa=create_vector(q);*
master, n_s	vektor, integer	*q*	*master=create_int(q);*
Kpoper, k, new, q	integer	1	

Tabulka 7.3.1 Přehled datových typů a příklady volání v SBCG metodě

Při práci se všemi funkcemi je nutné mít na paměti rozdíly v indexování vektorů a matic. V Matlabu začíná číslování od jedné, zatímco v jazyce C od nuly. Nyní si na příkladech ukážeme volání funkcí, které v jazyce C implementují požadované maticové operace SBCG algoritmu. Tyto funkce se nalézají v souborech *mxveclib.c*, *lulib.c* (řešení soustavy lineárních rovnic) a *sparse.c* (násobení řídké matice a vektoru).

1. skalární součin obdelníkových matic

   *ZR(m,[m,s]) = Z(:,m)' * R(:,[m,s])*
   ```
   mxmx(n, rhs, P, master, R, n_s, ZR);
   ```

2. bloková AXPY operace

   *X(:,i) = X(:,i) + P(:,m) * alfa*
   ```
   baxpy(n, rhs, master, PLUS, alfa, P, *(X+i));
   ```

   *R(:,i) = R(:,i) - U(:,m) * alfa*
   ```
   baxpy(n, rhs, master, MINUS, alfa, U, *(R+i));
   ```

3. operace násobení řídké matice a vektoru

   *U(:,i) = K * P(:,i)*
   ```
   spsymxv(K, *(P+i), *(U+i));
   ```
   Poznámka: *K* je proměnná typu *sparse_mx*. Tento datový typ je popsán níže.

4. řešení soustavy lineárních rovnic.

Funkce pro řešení soustavy lineárních rovnic se nacházejí v souboru *lulib.c*. Řešení probíhá ve dvou krocích:

a) LU rozklad matice
b) řešení soustavy

beta = ZRold(m,m)' \ ZR(m,i)

ad a) `ludcmp(ZRold,sizema,indx);`
ad b) `lubksb(ZRold,sizema,indx,beta);`

5. kopírování vektorů

Pold(:,i) = P(:,i)
`dcopy( n, *(P+i), *(Pold+i));`

Funkce *baxpy()* a *dcopy()* využívají služeb optimalizované knihovny pro numerické výpočty BLAS *(www.netlib.org)*, funkce `ludcmp()` a `lubksb()` byly převzaty z *dechert.econ.uh.edu/pub/compecon/numrec.html*.

Násobení matice a vektoru

V této kapitole se budeme podrobně věnovat operaci násobení matice a vektoru. Efektivní implementace této klíčové operace může značně snížit systémové nároky SBCG algoritmu.

Nechť A je $n \times n$ matice, nechť $x, y \in R^n$. Obvyklý postup k výpočtu součinu $y = Ax$ je následující:

$$y_i = \sum_{j=1}^{n} a_{i,j} x_j$$

Odpovídající funkce pro násobení matice a vektoru lze v Matlabu zapsat následovně:

```
function y=mxv(A,x)
y=zeros(size(x));
for i=1:n
    for j=1:n
        y(j) = y(j) + A(i,j)x(i);
    end;
end;
```

Pokud je matice *A* symetrická, tedy $a_{i,j} = a_{j,i}$, k součinu *y=Ax* potřebujeme znát pouze diagonálu a dolní (nebo horní) trojúhelníkovou část matice *A*. Následující algoritmus vypočítá součin *y=Ax* pouze s využitím diagonály a dolní trojúhelníkové části matice *A*.

```
function y = sym_mxv(A,x)
y = zeros(size(x));
for i = 1:n
    for j = 1:i
        y(j) = y(j) + A(i,j)x(i);
        if i ~= j
            y(i) = y(i) + A(i,j)x(j);
        end;
    end;
end;
```

Reprezentace řídkých matic

Výhodou iteračních řešičů je, že matice soustavy se účastní pouze operace násobení vektorem, řídký charakter matice tak zůstává zachován.

Uvažujme matici *A* řádu $n \times n$, která obsahuje *nnz* nenulových hodnot. Klasická reprezentace řídké matice je založena na myšlence, že pro každý nenulový prvek matice *A* budeme ukládat trojici sestávající z čísla řádku, čísla sloupce a příslušné hodnoty:

Název vektoru	Typ	Délka	Popis
columns	integer	*nnz*	čísla nenulových sloupců
raws	integer	*nnz*	čísla nenulových řádků
data	double	*nnz*	nenulové hodnoty

Tabulka 7.3.2 Klasická reprezentace řídké matice

Je zřejmé, že tento systém potřebuje pro uložení všech indexů v paměti prostor *nnz + nnz*. Existuje však mechanismus, který redukuje počet nutných indexů na *n + nnz*.

Vylepšená varianta

Čísla nenulových sloupců a samotná data jsou uložena stejně jako v předchozí metodě. Pokud však ukládáme nenulové prvky matice po řádcích, stačí jen vědět, od kterého indexu jsou ve vektoru *data* uloženy nenulové prvky příslušného řádku matice a od kterého indexu jsou ve vektoru *columns* uložena příslušná čísla těchto nenulových sloupců. Tuto informaci v sobě může nést

vektor, jehož velikost se rovná pouze počtu řádků matice. Reprezentace vylepšené metody je uvedena v následující tabulce.

Název vektoru	Typ	Délka	Popis
columns	integer	*nnz*	čísla nenulových sloupců
i1columns	integer	*n*	čísla 1. nenulových řádků
data	double	*nnz*	nenulové hodnoty

Tabulka 7.3.3 Vylepšená implementace řídké matice

Nechť i označuje i-tý řádek matice A, $i \subset <1; n>$.

Pak vektor *i1columns* obsahuje na i-té složce index, od kterého jsou ve vektoru *data* uloženy nenulové hodnoty a ve vektoru *columns* uložena čísla sloupců i-tého řádku matice A. Pokud platí *i1columns(i)* = -1, znamená to, že i-tý řádek matice A obsahuje pouze nulové hodnoty.

Příklad

Uvažujme matici

$$A = \begin{pmatrix} 10 & 0 & 0 & 0 & 20 \\ 0 & 30 & 0 & 40 & 50 \\ 60 & 70 & 80 & 0 & 0 \\ 0 & 0 & 0 & 90 & 100 \\ 0 & 110 & 120 & 0 & 130 \end{pmatrix} \qquad n = 5,\ nnz = 13.$$

Klasická reprezentace dává následující výsledky:

columns = `[1, 5, 2, 4, 5, 1, 2, 3, 4, 5, 2, 3, 5]`
raws = `[1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 5, 5, 5]`
data = `[10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130]`

Pokud použijeme pro uložení vylepšenou variantu, vektory *columns* a *data* budou shodné s předchozí variantou a vektor *i1columns* bude obsahovat:

i1columns = `[1, 3, 6, 9, 11]`.

Paměťová úspora vylepšené varianty se pochopitelně výrazněji projeví až při uložení rozsáhlejších matic soustav.

Programová implementace řídkých matic

Funkce, které pracují s řídkými maticemi, se nalézají v souboru *sparse.c*. V této knihovně se nachází funkce pro načtení řídké matice z datového souboru a funkce, která zajišťuje operaci násobení řídké symetrické matice a vektoru.

V naší implementaci je řídká matice reprezentovaná datovou strukturou *sparse_mx*, která je definovaná v hlavičkovém souboru *sparse.h:*

*typedef struct { int rows; int nnz; int *i1coll; int *coll; double *data;} sparse_mx;*

Pro načtení řídké matice z datového souboru slouží funkce

*sparse_mx read_sp(FILE *file);*

Struktura datového souboru pro uložení řídké matice je zobrazena v následující tabulce.

Název proměnné	**Typ**	**Délka**	**Popis**
rows	integer	1	počet sloupců matice
nnz	integer	1	počet nenulových hodnot matice
i1columns	integer	*rows*	čísla 1. nenulových řádků
columns	integer	*nnz*	čísla nenulových sloupců
data	double	*nnz*	nenulové hodnoty

Tabulka 7.3.4 Struktura datového souboru pro uložení řídké matice

Operaci násobení řídké symetrické matice a vektoru zajišťuje funkce

*void spsymxv(sparse_mx K, double *x, double *y);*

Tato funkce provede operaci $y=Kx$ s využitím symetrie matice K. Pro součin jsou použity pouze prvky dolní trojúhelníkové části matice K, čímž klesne paměťová náročnost celé operace.

7.4 Implementace sekvenční varianty SBCG metody

Hlavní program SBCG metody je uložen v souboru *sbcg.c*. Parametry programu, které se načítají z příkazové řádky, jsou následující:

- název souboru, ve kterém je v řídkém formátu uložena matice soustavy K
- název souboru, ve kterém je uložena matice pravých stran F
- název souboru, do kterého se má uložit řešení soustavy X
- relativní přesnost řešení ε
- koeficient lineární závislosti *coef*

Hlavní program provede kontrolu správnosti parametrů, načte do paměti matice *K* a *F*, zavolá funkci *sbcg()*, která vyřeší načtenou soustavu SBCG metodou a řešení soustavy potom uloží do souboru. Funkce *sbcg()* je uložena v souboru *sbcglib.c*. V tomto souboru se nacházejí i funkce, které úzce souvisí s SBCG algoritmem. Jedná se o funkci *findep()*, která odstraňuje lineárně závislé soustavy z „master" systému a o funkci *mvtoma()*, která v případě, kdy je master systém prázdný, přesune jednu slave soustavu do master systému.

Volání funkce *sbcg()* má následující tvar:

*int sbcg(int n, int rhs, sparse_mx K, double **F, double **X, double repsil, double coef, info *result);*

kde *n* je velikost matice soustavy *K*, *rhs* je počet pravých stran, *F* je matice pravých stran, *X* matice řešení, *repsil* je relativní přesnost řešení, *coef* je koeficient lineární závislosti a *result* je struktura, která obsahuje informace o počtu iterací, *Kp*-operací a dobu řešení SBCG metody. Funkce *sbcg()* uloží do matice *X* řešení soustavy a do struktury *result* informace o řešení. V případě, že byla zvolena příliš malá hodnota koeficientu lineární závislosti *coef*, může dojít k numerické nestabilitě algoritmu, neboť LU rozklad matic *ZR*, případně *UP* nemusí nutně existovat. V tomto případě vrací funkce *sbcg()* hodnotu 0, v opačném případě funkce vrací hodnotu 1.

7.5 Implementace paralelní varianty SBCG metody

V této kapitole popíšeme paralelní implementaci SBCG metody. Pro meziprocesorovou komunikaci využijeme služeb knihovny PVM (Paralel Virtual Machine).

Paralelní implementace SBCG metody je tvořena dvěma spustitelnými programy - programem *psbcg* (obsahuje kód rodičovského procesu) a *pslave* (obsahuje kód dětského procesu, který je spouštěn na jednotlivých procesech prostředím PVM).

Parametry soustavy se načítají z příkazové řádky programu *psbcg* a jsou následující:

- název souboru, ve kterém je v řídkém formátu uložena matice soustavy *K*
- název souboru, ve kterém je uložena matice pravých stran *F*
- název souboru, do kterého se má uložit řešení soustavy
- relativní přesnost řešení ε
- koeficient lineární závislosti *coef*
- počet dětských procesů, které má rodičovský proces vytvořit

Implementace funkcí pro meziprocesovou komunikaci

Při meziprocesové komunikaci se používají standardní funkce prostředí PVM a funkce *bsend()*, která se nachází v souboru *palglib.c*. Parametry volání této funkce jsou následující:

*void bsend(int n, int rhs, double *z, int I, int *odes, int msgtag, int *tids);*

kde *n* je velikost vektoru *z*, *rhs* představuje počet pravých stran problému a tedy i počet procesů a velikost vektoru *tids*, který obsahuje adresy (tidy) všech dětských procesů, a vektoru *odes*, který obsahuje informace o soustavách, kterým má být zpráva zaslána (zpravidla se jedná o vektor *n_s*, neboť zprávy rozesíláme všem aktivním procesům), *I* je číslo pravé strany, *msgtag* je uživatelská značka zprávy (tag), $msgtag \geq 0$. Funkce *bsend()* rozešle vektor *z* všem zbývajícím aktivním procesům z vektoru *odes*.

Komunikační protokol této funkce je v následující tabulce.

Obsah	Typ	Délka	Značka zprávy	Popis
z	double	*n*	*msgtag*	redukce vektoru *z*

Tabulka 7.5.1 Komunikační protokol funkce b_send()

Ukažme si chování této funkce na příkladu rozeslání *I*-tého sloupce matice *P*. Uvažujme *I*-tý dětský proces. Rozeslání *I*-tého sloupce matice *P* všem zbývajícím aktivním procesům (tj. procesům z vektoru *n_s*) se provede voláním

*bsend(n, rhs, *(P+I), I, n_s, msgtag, tids);*

Paralelní implementace funkce pro násobení dvou matic

Z komunikační analýzy paralelní verze SBCG algoritmu provedené v kapitole 6.6 plyne, že operace redukce matice a následný součin dvou obdélníkových matic se v algoritmu opakuje dvakrát. Zopakujme, že v prvním případě potřebujeme na *I*-tém procesu získat matici *Z(:,m)* a součin *ZR([m,I],I)* (značeno v syntaxi jazyka Matlab) a v druhém případě provádíme tutéž operaci pro matici *U(:,m)* a součin *UP(m,m)*.

Vzhledem k tomu, že se jedná o podobné operace, vytvořili jsme funkci, která toto implementuje. Tato funkce se jmenuje *palg_mxmx()* a je uložena v souboru *psbcglib.c*. Její parametry jsou následující:

*void palg_mxmx(int n, int rhs, double **Z, int *inxZ, double *r, int *inxR, double **ZR, int I, int *odes, int *tids, int msgtag);*

kde *n* je počet řádků matice *Z* a vektoru *r*, *rhs* je počet sloupců matice *Z* a velikost vektorů *inxZ*, *inxR*, *odes* a vektoru *tids*, který obsahuje adresy (tidy) všech dětských procesů, *msgtag* je uživatelská značka zprávy (tag), $msgtag \geq 0$.

Chování funkce na *I*-tém procesu:

- vstup: *n*, *rhs*, *I*, *msgtag*, vektory *inxZ*, *inxR*, *odes*, *tids* a vektor *r* (*I*-tý sloupec matice *R*). Pokud je *I*-tý proces zároveň master proces, pak na vstupu je také aktuální *I*-tý sloupec matice *Z*. Vektor *odes* obsahuje indexy všech aktivních úloh (odpovídá vektoru *n_s*)
- výstup: aktualizovaná matice *Z(:,inxZ)*, v matici *ZR* je uložen výsledek operace *Z(:,inxZ)'*R(:,inxR)*.

Komunikační protokol funkce *palg_mxmx()* je v následující tabulce. Funkce používá dva msgtagy. Jeden pro redukci matice *Z* a druhý pro redukci matice *ZR*.

Obsah	**Typ**	**Délka**	**Značka zprávy**	**Popis**
Z(:,i)	double	*n*	*msgtag*	redukce matice *Z*
ZR(m,i)	double	*M*	*msgtag* + 1	redukce matice *ZR*

Tabulka 7.5.2 Komunikační protokol funkce palg_mxmx()

Protokol zpráv SBCG metody

Program, který implementuje paralelní verzi SBCG algoritmu, se nachází ve třech fázích:

1) Inicializační fáze. V této fázi rodičovský program načte parametry z příkazové řádky, vytvoří dětské procesy a zašle jim parametry řešené úlohy.
2) Iterační fáze. V této fázi probíhá na dětských procesech iterační cyklus SBCG metody, komunikace probíhá pouze mezi dětskými procesy. Rodičovský proces je v této fázi blokován, čeká na zprávu od dětských procesů, která by obsahovala řešení příslušné soustavy.
3) Fáze redukce. V této fázi dětský proces vyřeší svou úlohu, pošle řešení rodičovskému procesu a ukončí se. Rodičovský proces přijme řešení od všech dětských procesů, zapíše ho do souboru a ukončí se.

Nyní si pro každou z těchto fází podrobněji popíšeme protokol zpráv.

Incicializační fáze

Dětské procesy přijmou od rodičovského procesu zprávu obsahující parametry soustavy (Msgtag 1), načtou do paměti datové soubory soustavy a pošlou rodičovskému procesu informaci o počtu načtených řádků soustavy (Msgtag 10). Pokud nastane v některém procesu chyba při načítání datových souborů, proces vrátí kód chyby (počet načtených řádků menší než 1) a ukončí se. Kódy možných chyb jsou uvedeny v tabulce 7.5.5.

Obsah	Typ	Délka	Popis
I	integer	1	pořadové číslo soustavy
lenK	integer	1	délka jména souboru obsahující matici soustavy *K*
argv[1]	string	*lenK*	název souboru obsahující matici *K*
lenF	integer	1	délka jména souboru obsahující matici pravých stran *F*
argv[2]	string	*lenF*	název souboru obsahující matici *F*
repsil	double	1	relativní přesnost řešení
coef	double	1	koeficient lineární závislosti
nhost	integer	1	počet dětských procesů
tids	integer	*nhost*	adresy dětských procesů

Tabulka 7.5.3 Protokol zprávy Msgtag 1

Obsah	Typ	Délka	Popis
n	integer	1	počet načtených řádků soustavy nebo kód chyby

Tabulka 7.5.4 Protokol zprávy Msgtag 10

n	Popis
-1	chyba při čtení matice *K*
-2	chyba při čtení matice *F*
-3	matice *K* a *F* mají různý počet řádků

Tabulka 7.5.5 Kód chyby (Msgtag 10)

Iterační fáze

V této fázi dochází k samotnému řešení soustav lineárních rovnic. Funkce, které implementují paralelní verzi SBCG metody, se nacházejí v souboru *psbcglib.c*. *I*-tý dětský proces, $I \subset < 1; rhs >$, zavolá funkci *psbcg()*, která vyřeší *I*-tou soustavu paralelní metodou SBCG. Volání funkce *psbcg()* má následující tvar:

*int psbcg(int I, int n, int rhs, sparse_mx K, double *r, double *x, double repsil, double coef, int *tids, info *result);*

Oproti sekvenční verzi přibyly následující parametry:

I - číslo pravé strany soustavy, který daný (*I*-tý) proces řeší; *tids* je vektor délky *rhs* obsahující adresy (tidy) všech dětských procesů. Připomeňme si, že obě tyto informace dostane každý dětský proces od rodičovského v inicializační fázi; *r* je reziduum *I*-té soustavy a *x* je řešení *I*-té soustavy.

Chování funkce na *I*-tém procesu:

- vstup: *I*, *n*, *rhs*, *coef*, řídká matice soustavy *K*, vektor f^I a *tids.*
- výstup: vektor *x*, struktra *result*

Po zavolání funkce *psbcg()* každý z dětských procesů vypočítá svou podmínku zakončení a pošle ji všem zbývajícím procesům (Msgtag 500).

Obsah	Typ	Délka	Popis
epsil[I]	double	1	podmínka ukončení SBCG metody pro *I*-tou soustavu

Tabulka 7.5.5 Protokol zprávy Msgtag 500

Voláním *palg_mxmx()* se aktualizují matice *Z* a *ZR*. Potom se zavolá funkce *findep()*. Testování lineární závislosti matice *ZR* tedy probíhá na všech dětských procesech. Jednotlivé dětské procesy si v případě ztráty lineární závislosti některé z master soustav aktualizují vektor *master*. Všechny master procesy potom vypočítají matici směrů *P* a voláním *b_send()* ji rozešlou všem zbývajícím procesům. Další volání *palg_mxmx()* aktualizuje matice *U* a *UP*. Potom každý z dětských procesů vypočítá nové hodnoty příslušných sloupců matic *X* a *R* a normu „svého“ rezidua, kterou rozešle všem zbývajícím procesům (Msgtag 400).

Obsah	Typ	Délka	Popis
$\|\|r^I\|\|$	double	1	norma rezidua *I*-té soustavy

Tabulka 7.5.6 Protokol zprávy Msgtag 400

Jednotlivé dětské procesy porovnají zaslané normy reziduí s podmínkou ukončení metody a samy si případně aktualizují vektory *m* a *n_s*.

Fáze redukce

Pokud *I*-tý proces vyřeší „svou“ soustavu, pošle příslušné řešení rodičovskému procesu (Masgtag 1) a ukončí se. Rodičovský proces přijme postupně řešení od všech dětských procesů, zapíše ho do souboru a ukončí se.

Obsah	Typ	Délka	Popis
I	integer	1	číslo soustavy

x^I	double	n	řešení i-té soustavy

Tabulka 7.5.7 Protokol zprávy Msgtag 1

8 Numerické experimenty

Tato kapitola se skládá ze dvou částí. V první části ukážeme na příkladu rozdílné chování BCG a SBCG algoritmu, druhá část je již věnovaná řešení konkrétního problému tvarové optimalizace. Ve všech případech byly metody ukončeny při dosažené relativní přesnosti řešení $\varepsilon \leq 10^{-4}$.

8.1 Porovnání numerické stability BCG a SBCG

Matice soustavy byla tvořena dvoudimenzionálním Laplace operátorem řádu 100×100, matice pravé strany byla tvořena vektory e_i, i = 1,...,11, které mají i-tou složku rovnou dvěma a ostatní složky jsou nulové. Jedná se tedy o soustavu lineárních rovnic s jedenácti lineárně nezávislými pravými stranami. Následující tabulka znázorňuje průběh řešení tohoto problému pomocí algoritmu BCG a SBCG. Pro každou zmiňovanou variantu je znázorněno:

- číslo iterace (k)
- počet soustav v master systému a počet nevyřešených soustav (M, N_S)
- číslo podmíněnosti matic $R_k^T M^{-1} Z_k$ a $P_k^T K P_k$ (condZR, condUP)
- průměrná norma rezidua (norm(R))

BCG						***SBCG,*** $coef = 1.10^{-1}$					
k	**M**	**N_S**	**condZR**	**condUP**	**norm(R)**	**k**	**M**	**N_S**	**condZR**	**condUP**	**norm(R)**
00	11	11	1.00e+000	2.87e+000	6.04e-01	00	11	11	1.00e+000	2.87e+000	6.04e-01
01	11	11	4.19e+016	1.14e+016	2.83e-01	01	09	11	7.86e+000	3.59e+000	3.29e-01
02	11	11	4.13e+007	1.15e+007	1.67e-01	02	09	11	1.16e+002	3.24e+001	2.07e-01
03	11	11	1.27e+008	4.15e+007	1.10e-01	03	05	11	5.83e+001	4.69e+001	1.51e-01
04	11	11	3.59e+008	1.89e+008	8.17e-02	04	04	11	9.36e+001	7.02e+001	1.24e-01
05	11	11	8.14e+008	5.32e+008	6.35e-02	05	03	11	5.20e+001	6.03e+001	1.03e-01
06	11	11	3.38e+009	2.55e+009	4.79e-02	06	02	11	1.39e+001	1.96e+001	8.76e-02
07	11	11	2.16e+010	1.18e+010	3.60e-02	07	02	11	2.15e+001	3.10e+001	7.71e-02
08	11	11	1.02e+011	5.93e+010	2.79e-02	08	01	11	1.00e+000	1.00e+000	7.06e-02
09	11	11	2.92e+011	1.88e+011	2.14e-02	09	01	11	1.00e+000	1.00e+000	6.73e-02
10	11	11	1.65e+012	1.12e+012	1.33e-02	10	01	11	1.00e+000	1.00e+000	6.33e-02
11	11	11	2.49e+014	3.02e+014	4.89e-03	11	01	11	1.00e+000	1.00e+000	6.21e-02
12	11	11	1.39e+015	2.04e+015	1.92e-03	12	01	11	1.00e+000	1.00e+000	5.95e-02
13	11	11	3.57e+015	6.44e+015	1.38e-03	13	01	11	1.00e+000	1.00e+000	5.71e-02
14	09	09	3.83e+012	3.92e+012	1.45e-03	14	01	11	1.00e+000	1.00e+000	5.74e-02
15	04	04	4.39e+010	6.73e+010	2.78e-03	15	01	11	1.00e+000	1.00e+000	5.89e-02
16	04	04	4.88e+010	5.36e+010	2.58e-03	16	01	11	1.00e+000	1.00e+000	5.61e-02
17	04	04	7.77e+010	8.11e+010	2.43e-03	17	01	11	1.00e+000	1.00e+000	5.44e-02
18	04	04	9.72e+010	1.32e+011	2.05e-03	18	01	11	1.00e+000	1.00e+000	5.39e-02
19	03	03	2.44e+010	2.98e+010	2.03e-03	19	01	11	1.00e+000	1.00e+000	5.31e-02
20	03	03	2.44e+010	2.57e+010	1.76e-03	20	01	11	1.00e+000	1.00e+000	5.25e-02

21	03	03	4.21e+010	4.95e+010	1.71e-03	21	01	11	1.00e+000	1.00e+000	5.23e-02
22	03	03	7.62e+010	6.73e+010	1.87e-03	22	01	11	1.00e+000	1.00e+000	5.22e-02
23	03	03	2.40e+011	2.28e+011	1.76e-03	23	01	11	1.00e+000	1.00e+000	5.22e-02
24	03	03	6.31e+011	6.50e+011	1.82e-03	24	01	11	1.00e+000	1.00e+000	5.24e-02
25	03	03	1.70e+012	1.53e+012	2.02e-03	25	01	10	1.00e+000	1.00e+000	3.33e-02
26	03	03	4.71e+012	4.88e+012	2.03e-03	26	01	10	1.00e+000	1.00e+000	2.86e-02
27	03	03	8.86e+012	9.73e+012	2.03e-03	27	01	10	1.00e+000	1.00e+000	2.65e-02
28	03	03	1.66e+013	1.81e+013	1.85e-03	28	01	10	1.00e+000	1.00e+000	2.60e-02
29	03	03	2.65e+013	2.53e+013	2.05e-03	29	01	10	1.00e+000	1.00e+000	2.60e-02
30	03	03	5.89e+013	3.59e+013	2.14e-03	30	01	10	1.00e+000	1.00e+000	2.52e-02
31	03	03	5.53e+013	1.59e+013	2.45e-03	31	01	10	1.00e+000	1.00e+000	2.50e-02
32	03	03	1.02e+014	1.66e+014	2.36e-03	32	01	10	1.00e+000	1.00e+000	2.44e-02
33	03	03	7.40e+012	6.19e+012	2.10e-03	33	01	10	1.00e+000	1.00e+000	2.43e-02
34	03	03	7.86e+012	5.49e+013	2.03e-03	34	01	10	1.00e+000	1.00e+000	2.41e-02
35	03	03	8.38e+012	8.70e+013	2.03e-03	35	01	10	1.00e+000	1.00e+000	2.40e-02
36	03	03	1.05e+013	1.57e+014	2.04e-03	36	01	10	1.00e+000	1.00e+000	2.39e-02
37	03	03	9.19e+012	1.44e+014	1.95e-03	37	01	10	1.00e+000	1.00e+000	2.36e-02
38	03	03	6.05e+012	8.62e+013	1.75e-03	38	01	10	1.00e+000	1.00e+000	2.34e-02
39	03	03	3.11e+012	3.24e+013	1.53e-03	39	01	10	1.00e+000	1.00e+000	2.33e-02
40	03	03	1.48e+012	7.47e+012	1.52e-03	40	01	09	1.00e+000	1.00e+000	2.40e-02
41	03	03	1.07e+012	1.30e+012	1.66e-03	41	01	09	1.00e+000	1.00e+000	2.34e-02
42	03	03	1.16e+012	4.29e+011	1.87e-03	42	01	09	1.00e+000	1.00e+000	2.31e-02
43	03	03	1.43e+012	5.39e+011	1.99e-03	43	01	09	1.00e+000	1.00e+000	2.31e-02
44	03	03	1.92e+012	5.53e+012	1.95e-03	44	01	09	1.00e+000	1.00e+000	2.30e-02
45	03	03	2.42e+012	2.34e+013	1.94e-03	45	01	09	1.00e+000	1.00e+000	2.28e-02
46	03	03	3.13e+012	5.07e+013	2.04e-03	46	01	09	1.00e+000	1.00e+000	2.24e-02
47	03	03	4.73e+012	8.91e+013	2.08e-03	47	01	09	1.00e+000	1.00e+000	2.23e-02
48	03	03	5.70e+012	1.32e+014	1.99e-03	48	01	09	1.00e+000	1.00e+000	2.22e-02
49	03	03	6.31e+012	1.11e+014	1.89e-03	49	01	08	1.00e+000	1.00e+000	2.28e-02
50	03	03	1.30e+013	2.58e+014	1.87e-03	50	01	08	1.00e+000	1.00e+000	2.22e-02
51	03	03	2.07e+013	4.76e+014	1.87e-03	51	01	08	1.00e+000	1.00e+000	2.21e-02
52	03	03	1.96e+013	3.25e+014	1.81e-03	52	01	08	1.00e+000	1.00e+000	2.20e-02
53	03	03	3.12e+013	6.60e+014	1.70e-03	53	01	08	1.00e+000	1.00e+000	2.11e-02
54	03	03	2.62e+013	4.74e+014	1.68e-03	54	01	08	1.00e+000	1.00e+000	2.05e-02
55	03	03	2.97e+013	6.89e+014	1.64e-03	55	01	08	1.00e+000	1.00e+000	2.04e-02
56	03	03	2.24e+013	4.63e+014	1.58e-03	56	01	07	1.00e+000	1.00e+000	1.71e-02
57	03	03	1.81e+013	4.14e+014	1.76e-03	57	01	07	1.00e+000	1.00e+000	1.62e-02
58	03	03	1.71e+013	4.41e+014	2.12e-03	58	01	07	1.00e+000	1.00e+000	1.60e-02
59	03	03	2.26e+013	5.89e+014	2.56e-03	59	01	07	1.00e+000	1.00e+000	1.60e-02
60	03	03	2.51e+013	6.08e+014	3.03e-03	60	01	07	1.00e+000	1.00e+000	1.60e-02
61	03	03	1.71e+013	9.31e+014	3.35e-03	61	01	07	1.00e+000	1.00e+000	1.57e-02
62	03	03	9.29e+012	1.16e+015	2.89e-03	62	01	07	1.00e+000	1.00e+000	1.56e-02
63	03	03	7.17e+012	1.51e+015	2.57e-03	63	01	07	1.00e+000	1.00e+000	1.55e-02
64	03	03	5.85e+012	1.10e+015	2.56e-03	64	01	06	1.00e+000	1.00e+000	1.58e-02
65	03	03	5.47e+012	6.13e+014	2.53e-03	65	01	06	1.00e+000	1.00e+000	1.51e-02
66	03	03	4.41e+012	2.28e+014	2.39e-03	66	01	06	1.00e+000	1.00e+000	1.46e-02
67	03	03	3.18e+012	3.05e+013	2.43e-03	67	01	06	1.00e+000	1.00e+000	1.45e-02
68	03	03	3.21e+012	2.06e+013	2.41e-03	68	01	06	1.00e+000	1.00e+000	1.42e-02
69	03	03	3.32e+012	3.91e+013	2.38e-03	69	01	06	1.00e+000	1.00e+000	1.41e-02
70	03	03	3.33e+012	6.59e+013	2.51e-03	70	01	05	1.00e+000	1.00e+000	1.19e-02
71	03	03	3.68e+012	7.69e+013	2.52e-03	71	01	05	1.00e+000	1.00e+000	1.08e-02
72	03	03	3.66e+012	7.99e+013	2.49e-03	72	01	05	1.00e+000	1.00e+000	1.06e-02

73	03	03	3.49e+012	7.47e+013	2.39e-03	73	01	05	1.00e+000	1.00e+000	1.04e-02
74	03	03	3.23e+012	1.29e+014	2.36e-03	74	01	05	1.00e+000	1.00e+000	1.03e-02
75	03	03	3.50e+012	3.82e+014	2.40e-03	75	01	05	1.00e+000	1.00e+000	1.03e-02
76	03	03	2.94e+012	2.31e+014	2.14e-03	76	01	05	1.00e+000	1.00e+000	1.02e-02
77	03	03	2.32e+012	1.95e+014	2.03e-03	77	01	05	1.00e+000	1.00e+000	1.02e-02
78	03	03	2.45e+012	2.86e+014	2.10e-03	78	01	05	1.00e+000	1.00e+000	1.02e-02
79	03	03	3.34e+012	4.88e+014	2.12e-03	79	01	04	1.00e+000	1.00e+000	9.52e-03
80	03	03	3.73e+012	6.01e+014	1.99e-03	80	01	04	1.00e+000	1.00e+000	8.94e-03
81	03	03	2.72e+012	3.16e+014	1.72e-03	81	01	04	1.00e+000	1.00e+000	8.57e-03
82	03	03	1.39e+012	5.61e+013	1.47e-03	82	01	04	1.00e+000	1.00e+000	8.44e-03
83	03	03	9.49e+011	2.83e+013	1.46e-03	83	01	04	1.00e+000	1.00e+000	8.34e-03
84	03	03	1.06e+012	6.66e+013	1.51e-03	84	01	04	1.00e+000	1.00e+000	8.23e-03
85	03	03	1.16e+012	1.23e+014	1.58e-03	85	01	04	1.00e+000	1.00e+000	8.13e-03
86	03	03	1.18e+012	1.28e+014	1.57e-03	86	01	04	1.00e+000	1.00e+000	8.05e-03
87	03	03	1.12e+012	1.14e+014	1.55e-03	87	01	03	1.00e+000	1.00e+000	6.93e-03
88	03	03	1.16e+012	1.03e+014	1.54e-03	88	01	03	1.00e+000	1.00e+000	6.09e-03
89	03	03	1.45e+012	2.49e+014	1.68e-03	89	01	03	1.00e+000	1.00e+000	5.86e-03
90	03	03	1.79e+012	3.03e+014	1.75e-03	90	01	03	1.00e+000	1.00e+000	5.74e-03
91	03	03	2.29e+012	4.02e+014	1.74e-03	91	01	03	1.00e+000	1.00e+000	5.69e-03
92	03	03	2.95e+012	7.41e+014	1.73e-03	92	01	03	1.00e+000	1.00e+000	5.65e-03
93	03	03	2.95e+012	7.37e+014	1.74e-03	93	01	03	1.00e+000	1.00e+000	5.62e-03
94	03	03	2.91e+012	7.45e+014	1.76e-03	94	01	02	1.00e+000	1.00e+000	2.00e-03
95	03	03	2.55e+012	5.64e+014	1.84e-03	95	01	02	1.00e+000	1.00e+000	7.00e-04
96	03	03	2.67e+012	4.15e+014	1.84e-03	96	01	02	1.00e+000	1.00e+000	4.51e-04
97	03	03	4.61e+012	1.06e+015	1.86e-03	97	01	02	1.00e+000	1.00e+000	3.43e-04
98	03	03	6.17e+012	1.68e+015	1.83e-03	98	01	02	1.00e+000	1.00e+000	2.87e-04
99	03	03	6.28e+012	2.09e+015	1.82e-03	99	01	01	1.00e+000	1.00e+000	1.40e-04

- diverguje

Tabulka 8.1.1 Protokol řešení BCG a SBCG algoritmu

Zabývejme se nejdříve algoritmem BCG. Z tabulky je vidět, že čísla podmíněnosti matic $(R_k^T M^{-1} R_k)$ a $(P_k^T K P_k)$ jsou přibližně shodná a mají rostoucí tendenci. V 14-té a 15-té iteraci se čísla podmíněnosti obou matic zmenší. Tento pokles je způsoben vyjmutím vyřešených soustav z problému. Podobná situace nastala i v 19-té iteraci, ale číslo podmíněnosti matic zůstává v tomto případě téměř beze změny. Také velikost normy rezidua od této iterace již podstatně neklesá. BCG algoritmus nekonverguje, a to i přesto, že pravé strany jsou lineárně nezávislé.

Vedlejší tabulka ukazuje chování SBCG algoritmu. Již v první iteraci dochází ke zmenšení počtu master soustav z 11 na 9. Tento pokles pokračuje i v dalších iteracích a je způsoben relativně vysokou hodnotou koeficientu lineární závislosti, převládá tedy SCG charakter SBCG algoritmu. Celkově SBCG algoritmus potřeboval na řešení přesně 100 iterací a 137 operací násobení matice soustavy vektorem (Kp-operací). SCG algoritmus vyřešil tuto úlohu za 150 operací a klasická metoda sdružených gradientů dokonce až za 249 iterací.

8.2 Modelový příklad systému ODESSY

V této části ukážeme výsledky numerických testů algoritmů SCG, BCG a SBCG, které byly aplikovány na modelovou třídimenzionální úlohu (kvádr) tvarové optimalizace systému ODESSY Optimum Desig System (ODESSY). Tento systém byl vyvinut na Univerzitě Aalborg v Dánsku. Úloha byla počítána na síti $10\times10\times35$. Matice soustavy byla řádu 10500×10500, velikost datového souboru obsahující tuto matici v řídké podobě byla 4,3 MB. Üloha obsahovala pět pravých stran, jejich normy jsou v tabulce 8.2.1.

i	1	2	3	4	5
$\|\| f^i \|\|$	6 559	17 966	5 315	29 099	2 690

Tabulka 8.2.1 Normy matice pravých stran F

Výsledky numerických experimentů jsou znázorněny v tabulce 8.2.2.

Sloupec *Method* obsahuje jména metod, pro které byla tato úloha řešena. CG zde označuje klasický algoritmus sdružených gradientů, kdy jednotlivé soustavy jsou řešeny postupně, metody označené 1,00E-01, ...1,00E-06 představují velikost koeficientu lineární závislosti SBCG algoritmu, *k* je počet iterací, *Kp-oper* je počet operací násobení matice soustavy vektorem, *Time* je doba výpočtu na systému IBM SP s procesorem Thin a 256 MB RAM.

V následujících sloupcích je výsledné zrychlení, které se vztahuje k algoritmu CG. Při volbě $coef = 1.10^{-2}$ je SBCG algoritmus pomalejší než klasická metoda sdružených gradientů. Nejvýraznějšího zrychlení bylo dosaženo při SBCG algoritmu s parametrem $coef = 1.10^{-5}$, respektive $coef = 1.10^{-6}$. Výsledky se sice v tomto případě vůbec neliší od BCG algoritmu, ale je nutné si uvědomit, že BCG algoritmus nemá obecně zaručenou konvergenci.

Method	**k**	**Kp-oper**	**Time (sec)**	**Speed up k**	**Speed up Kp-oper**	**Speed up Time**
CG	965	965	90	1	1	1
SCG	634	634	68	1,52	1,52	1,32
1,00E-01	687	780	89	1,40	1,23	1,01
1,00E-02	883	1014	117	1,09	0,95	0,76
1,00E-03	397	606	70	2,43	1,59	1,28
1,00E-04	302	651	76	3,19	1,48	1,18
1,00E-05	110	482	58	8,77	2,00	1,55
1,00E-06	110	482	58	8,77	2,00	1,55
BCG	110	482	58	8,77	2,00	1,55

Tabulka 8.2.2 Porovnání rychlosti konvergence metod CG, SCG, BCG a SBCG

Protokol řešení SBCG algoritmu s $coef = 1.10^{-6}$ je v tabulce 8.2.3. Je z něho patrno, že v tomto případě převládl blokový charakter SBCG algoritmu - všechny nevyřešené soustavy jsou zároveň v master systému. Dále se ukázalo, že rychlost konvergence není pro jednotlivé pravé strany shodná. Tento fakt je způsoben různou velikostí norem pravých stran (viz tabulka 8.2.1).

SBCG, $coef = 1.10^{-6}$						***SBCG,*** $coef = 1.10^{-6}$ ***(pokračování)***					
k	**M**	**N_S**	**condZR**	**condUP**	**norm(R)**	**k**	**M**	**N_S**	**condZR**	**condUP**	**norm(R)**
00	05	05	2.30e+02	1.08e+02	8.33e+03	56	05	05	1.96e+08	1.85e+08	3.34e+03
01	05	05	2.97e+02	1.27e+03	6.38e+03	57	05	05	2.41e+08	2.98e+08	2.68e+03
02	05	05	1.95e+02	4.07e+02	6.65e+03	58	05	05	2.68e+08	1.92e+08	2.48e+03
03	05	05	1.77e+02	1.64e+02	5.46e+03	59	05	05	3.57e+08	3.47e+08	2.26e+03
04	05	05	2.12e+02	2.87e+02	5.45e+03	60	05	05	3.46e+08	3.48e+08	1.88e+03
05	05	05	2.43e+02	2.14e+02	4.77e+03	61	05	05	4.66e+08	5.58e+08	1.54e+03
06	05	05	3.05e+02	3.13e+02	4.67e+03	62	05	05	3.10e+08	3.40e+08	1.32e+03
07	05	05	4.19e+02	3.49e+02	4.36e+03	63	05	05	3.67e+08	3.64e+08	9.40e+02
08	05	05	7.32e+02	6.48e+02	4.44e+03	64	05	05	3.71e+08	3.92e+08	8.30e+02
09	05	05	1.06e+03	1.27e+03	4.27e+03	65	05	05	3.76e+08	3.33e+08	7.39e+02
10	05	05	9.11e+02	9.54e+02	4.20e+03	66	05	05	4.66e+08	5.13e+08	5.82e+02
11	05	05	9.96e+02	1.02e+03	4.21e+03	67	05	05	4.82e+08	4.34e+08	5.24e+02
12	05	05	9.90e+02	9.92e+02	4.08e+03	68	05	05	5.72e+08	6.65e+08	4.17e+02
13	05	05	1.29e+03	1.11e+03	4.30e+03	69	05	05	5.55e+08	6.62e+08	3.22e+02
14	05	05	1.66e+03	1.83e+03	4.39e+03	70	05	05	4.33e+08	4.28e+08	2.81e+02
15	05	05	1.63e+03	1.05e+03	4.65e+03	71	05	05	4.40e+08	5.09e+08	2.18e+02
16	05	05	3.34e+03	2.29e+03	5.14e+03	72	05	05	4.24e+08	5.18e+08	2.02e+02
17	05	05	6.09e+03	6.60e+03	5.07e+03	73	05	05	4.45e+08	5.49e+08	1.49e+02
18	05	05	6.09e+03	5.42e+03	5.25e+03	74	05	05	3.71e+08	3.92e+08	1.16e+02
19	05	05	8.29e+03	6.53e+03	5.56e+03	75	05	05	3.22e+08	3.79e+08	9.68e+01
20	05	05	1.10e+04	8.72e+03	5.82e+03	76	05	05	3.28e+08	3.71e+08	7.76e+01
21	05	05	1.56e+04	1.28e+04	5.88e+03	77	05	05	3.02e+08	2.56e+08	6.82e+01
22	05	05	2.28e+04	1.92e+04	5.77e+03	78	05	05	3.87e+08	4.12e+08	5.52e+01
23	05	05	3.22e+04	3.40e+04	6.23e+03	79	05	05	3.81e+08	3.84e+08	4.87e+01
24	05	05	4.63e+04	6.40e+04	5.90e+03	80	05	05	3.60e+08	3.94e+08	4.08e+01
25	05	05	4.38e+04	4.83e+04	5.76e+03	81	05	05	4.16e+08	4.26e+08	3.14e+01
26	05	05	5.72e+04	4.61e+04	5.69e+03	82	05	05	4.20e+08	3.93e+08	2.84e+01
27	05	05	8.20e+04	6.22e+04	5.71e+03	83	05	05	4.41e+08	4.94e+08	2.24e+01
28	05	05	1.20e+05	1.08e+05	6.15e+03	84	05	05	3.47e+08	3.24e+08	2.00e+01
29	05	05	1.81e+05	1.64e+05	6.92e+03	85	04	04	4.27e+08	5.28e+08	1.86e+01
30	05	05	3.00e+05	3.24e+05	6.95e+03	86	04	04	3.66e+08	3.88e+08	1.47e+01
31	05	05	3.96e+05	3.41e+05	6.36e+03	87	04	04	3.66e+08	4.19e+08	1.21e+01
32	05	05	4.16e+05	3.11e+05	6.84e+03	88	04	04	3.06e+08	3.68e+08	1.00e+01
33	05	05	7.73e+05	7.26e+05	7.63e+03	89	04	04	2.82e+08	3.44e+08	8.11e+00
34	05	05	1.40e+06	1.49e+06	8.07e+03	90	04	04	2.12e+08	2.89e+08	6.19e+00
35	05	05	2.21e+06	2.26e+06	7.92e+03	91	04	04	1.40e+08	9.81e+07	5.26e+00
36	05	05	2.62e+06	1.59e+06	8.15e+03	92	04	04	1.42e+08	2.09e+08	4.18e+00
37	05	05	3.50e+06	2.80e+06	9.50e+03	93	04	04	9.90e+07	1.51e+08	3.71e+00

38	05	05	7.10e+06	9.97e+06	9.47e+03	94	04	04	7.73e+07	1.20e+08	2.98e+00
39	05	05	8.10e+06	7.99e+06	9.22e+03	95	03	03	2.65e+07	3.89e+07	2.77e+00
40	05	05	9.20e+06	7.07e+06	9.84e+03	96	02	02	3.31e+04	3.04e+04	3.29e+00
41	05	05	1.53e+07	1.33e+07	1.04e+04	97	02	02	3.28e+04	3.59e+04	2.53e+00
42	05	05	2.17e+07	1.87e+07	1.11e+04	98	02	02	2.62e+04	2.58e+04	2.04e+00
43	05	05	3.09e+07	3.22e+07	1.05e+04	99	01	01	1.00e+00	1.00e+00	3.09e+00
44	05	05	3.65e+07	3.08e+07	1.05e+04	100	01	01	1.00e+00	1.00e+00	2.82e+00
45	05	05	5.04e+07	5.22e+07	9.93e+03	101	01	01	1.00e+00	1.00e+00	2.22e+00
46	05	05	6.85e+07	6.55e+07	8.72e+03	102	01	01	1.00e+00	1.00e+00	1.88e+00
47	05	05	9.72e+07	8.03e+07	8.18e+03	103	01	01	1.00e+00	1.00e+00	1.65e+00
48	05	05	9.84e+07	9.99e+07	7.24e+03	104	01	01	1.00e+00	1.00e+00	1.34e+00
49	05	05	1.02e+08	1.08e+08	5.97e+03	105	01	01	1.00e+00	1.00e+00	1.15e+00
50	05	05	9.48e+07	8.37e+07	5.49e+03	106	01	01	1.00e+00	1.00e+00	1.03e+00
51	05	05	9.12e+07	9.17e+07	5.18e+03	107	01	01	1.00e+00	1.00e+00	9.25e-01
52	05	05	1.01e+08	1.16e+08	4.50e+03	108	01	01	1.00e+00	1.00e+00	7.86e-01
53	05	05	1.13e+08	1.23e+08	4.10e+03	109	01	01	1.00e+00	1.00e+00	7.01e-01
54	05	05	1.37e+08	1.33e+08	3.68e+03	110	01	01	1.00e+00	1.00e+00	6.70e-01
55	05	05	1.81e+08	1.81e+08	3.51e+03	111	01	01	1.00e+00	1.00e+00	6.08e-01

Tabulka 8.2.3 Protokol řešení SBCG algoritmu

V následujících tabulkách je porovnání účinnosti algoritmů SCG a SBCG v závislosti na počtu pravých stran. Je vidět, že pro úlohu s počtem pravých stran menších než pět jsou výkonnostní rozdíly mezi oběma algoritmy malé. Teprve při úloze s pěti pravými stranami se projeví výhoda SBCG algoritmu, který vyřeší danou úlohu za 482 Kp-operací, zatímco SCG algoritmus potřebuje na tutéž úlohu 634 Kp-operací.

q	k	Kp-oper	Time (sec)	Speed up k	Speed up Kp-oper	Speed up Time
1	193	193	18	1	1	1
2	264	264	26	1,46	1,46	1,38
3	320	320	33	1,80	1,80	1,63
4	434	434	46	1,77	1,77	1,56
5	634	634	68	1,52	1,52	1,32

a) Algoritmus SCG

q	k	Kp-oper	Time (sec)	Speed up k	Speed up Kp-oper	Speed up Time
1	193	193	18	1	1	1
2	138	274	27	2,79	1,40	1,33
3	119	345	37	4,86	1,67	1,45
4	112	425	49	6,89	1,81	1,46
5	110	482	58	8,77	2,00	1,55

b) Algoritmus SBCG, $coef = 1.10^{-6}$

Tabulka 8.2.4 Porovnání účinnosti algoritmů SCG a SBCG v závislosti na počtu pravých stran

8.3 Testy paralelní verze

V této kapitole jsou uvedeny výsledky numerických experimentů paralelních verzí algoritmů SCG, BCG a SBCG. Testy byly prováděny na systému IBM SP. Verze PVM byla 3.3.11.

Příklad 1

Matice soustavy byla opět tvořena dvoudimenzionálním Laplace operátorem řádu 100×100, matice pravé strany byla tvořena vektory e_i, i = 1,...,4, které mají i-tou složku rovnou dvěma a ostatní složky jsou nulové. Následující tabulky znázorňují výsledky řešení tohoto problému pomocí paralelních verzí algoritmů SCG, BCG a SBCG.

Sloupec *Proces* obsahuje pořadová čísla procesů (vždy platí, že i-tý proces řeší i-tou soustavu problému), k je počet iterací, která daná metoda vykonala na konkrétním procesu, *Kp-oper* je počet operací násobení matice soustavy s vektorem, který vykonal konkrétní proces, *Time* je doba výpočtu na daném procesu.

Proces	k	Kp-oper	Time (sec)	k	Kp-oper	Time (sec)
1	23	23	0.56	17	17	0.89
2	40	17	0.73	17	17	0.89
3	50	10	0.82	17	17	0.89
4	60	10	0.82	18	18	0.90

a) SCG *b) BCG*

Tabulka 8.3.1 Porovnání paralelních verzí algoritmů SCG a BCG

Podívejme se nejdříve na paralelní verzi SCG metody. Nejdříve je master proces proces č.1 a na vyřešení úlohy potřebuje 23 iterací. Pak se stává master procesem proces číslo dvě, který potřebuje na vyřešení dalších 10 iterací. Obdobně se chovají i procesy číslo tři a čtyři.

BCG algoritmus má rozdílné chování, řešení všech procesů je známo téměř současně. Tři procesy vyřeší své soustavy v 17-té iteraci, poslední proces je ukončen v 18-té iteraci.

Proces	k	Kp-oper	Time (sec)	k	Kp-oper	Time (sec)
1	23	23	0.67	15	15	0.77
2	36	16	0.80	16	16	0.81

3	43	10	0.86	16	16	0.81
4	50	11	0.86	16	16	0.81

a) $coef = 1.10^{-1}$ b) $coef = 1.10^{-4}$

Tabulka 8.3.2 Porovnání rychlosti paralelní verze algoritmu SBCG v závislosti na velikosti coef

Chování SBCG algoritmu ovlivňuje velikost koeficientu lineární závislosti. Pokud je $coef = 1.10^{-1}$, převládá SCG charakter, řešení soustav dostáváme postupně. Při volbě $coef = 1.10^{-4}$ převládá již BCG charakter SBCG algoritmu, řešení soustav dostáváme téměř současně.

Příklad 2

V této části uvedeme výsledky testu příkladu z kapitoly 8.2 s tím rozdílem, že jsme počet pravých stran zredukovali na čtyři. Úlohu jsme testovali pro počet procesorů 1, 2 a 4. V následujících tabulkách je porovnání rychlostí SCG a SBCG algoritmu v závislosti na počtu použitých procesorů. V případě, kdy byl počet procesorů roven jedné, případně dvěma, se jednalo o pseudoparalelní běh. Na daném procesoru běžely čtyři případně dva procesy. Během experimentů se ukázalo, že vytíženost jednotlivých procesorů je různá a rychle se mění. Proto jsou v tabulkách zaznamenány průměrné hodnoty dosažených časů.

Proces	**k**	**Kp-oper**	**Time (sec) 1 CPU**	**Time (sec) 2 CPU**	**Time (sec) 4 CPU**
1	186	186	160.72	225.11	153.14
2	265	79	193.94	157.79	224.48
3	326	61	215.26	279.00	258.22
4	452	126	259.36	323.07	302.79

a) Algoritmus SCG

Proces	**k**	**Kp-oper**	**Time (sec) 1 CPU**	**Time (sec) 2 CPU**	**Time (sec) 4 CPU**
1	95	95	315.21	661.45	865.62
2	105	105	334.01	680.96	890.08
3	105	105	334.37	681.11	890.10
4	120	120	337.66	687.09	896.03

b) Algoritmus SBCG, $coef = 1.10^{-6}$

Tabulka 8.3.3 Porovnání rychlostí paralelních verzí algoritmů SCG a SBCG v závislosti na počtu procesorů (CPU)

Dětské procesy jsou vybaveny mechanismy zajišťující jejich relativní nezávislost k sobě navzájem: Komunikace v iterační fázi probíhá přímo mezi dětskými procesy bez účasti rodičovského procesu. Každý z dětských procesů si dále sám detekuje případnou ztrátu lineární závislosti v master soustavách. Vyjímání závislých soustav z master systému probíhá bez meziprocesorové komunikace se zbývajícími dětskými procesy. Přesto však dosažené rychlosti paralelních algoritmů nemohou konkurovat sekvenčním verzím. Ukázalo se, že rychlost jednotlivých procesorových uzlů je v porovnáním s rychlostí jejich vzájemné komunikace velmi vysoká. Z technických důvodů nemohla být provedena optimalizace komunikace pomocí vizualizačního programu XPVM.

Závěr

Algoritmy, které jsou použity v této diplomové práci, vycházejí především z prací autorů [Feng, Owen, Peric, 1995] a [Suarjana, Law, 1994]. Tato diplomová práce obsahuje i některé nové prvky. Jedná se především o volbu mechanismu, kterým je zabezpečena numerická stabilita SBCG algoritmu a o část věnované analýze paralelních verzí algoritmů BCG, SCG a SBCG (kapitola 6). Další část práce se týkala sekvenční a paralelní implemetace SBCG metody v prostředí ANSI C. Byly vytvořeny knihovny, které zajišťují jednotlivé matematické i vstupně-výstupní operace, včetně využití techniky řídkých matic. Funkčnost celého řešiče byla úspěšně odzkoušena vyřešením poměrně velké úlohy ze systému ODESSY. Z numerických experimentů vyplynulo, že výhody SBCG algoritmu se projevily až u problému s pěti pravými stranami. BCG algoritmus pro svou obecnou numerickou nestabilitu nelze doporučit. Při experimentech s paralelní verzí SBCG algoritmu se ukázalo, že režie spojená s komunikací je vysoká.

Literatura

[Dostál, Vondrák, Rasmussen, 1997] Z. Dostál, V. Vondrák, J. Rasmussen, “Využití iteračních řešičů v úlohách tvarové optimalizace” (1997)

[Feng, Owen, Peric, 1995] Y.T. Feng, D. R .J. Owen, D. Peric, “A block conjugate gradient method applied to linear system with multiple right hand sides”, Comput. Methods in Appl. Mech. Eng. 127, 203-215 (1995)

[Suarjana, Law, 1994] M. Suarjana, K. H. Law, “Successive conjugate gradient methods for structural analysis with multiple load cases”, Int. J. Num. Methods Eng., 37, 4185-203 (1990)

[Praks, 1999] P. Praks, “Metody sdružených gradientů pro řešení soustav lineárních rovnic s několika pravými stranami”, příspěvek ve sborníku 7. konference studentů v matematice škol VŠTEZ, Bechyně (1999)

[Golub, Loan, 1989] G. H. Golub, C. F. Van Loan, “Matrix Computations”, The Johns Hopkins University Press (1989)

[Hestenes, Stiefel, 1952] M. R. Hestenes, E. Stiefel, “Methods of Conjugate Gradients for Solving Linear Systems”, J. Res. Nat. Bur. Stand. 49, 409-36 (1952)