FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání diplomové práce

Název:	Hledání vlastních čísel a vlastních vektorů pro rozsáhlé řídké symetrické matice
Student:	Bc. Matěj Razák
Vedoucí:	Ing. Daniel Langr, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Systémové programování
Katedra:	Katedra teoretické informatiky
Platnost zadání:	do konce letního semestru 2023/2024

Pokyny pro vypracování

Seznamte se s metodami pro hledání vlastních čísel a vlastních vektorů řídkých matic. Uvažujte především Lanczosův algoritmu a metodu LOBPCG. Seznamte se s blokovými verzemi těchto metod. Seznamte se s knihovnou OPMILancz poskytující jedno-vektorovou a blokovou implementaci Lanczosova algoritmu pro hledání vlastních čísel a vektorů symetrických řídkých matic nad hybridním paralelním programovacím modelem MPI+OpenMP v jazyce C++. Navrhněte a implementujte vylepšení stávající implementace těchto metod. Navrhněte a implementujte do knihovny OMPILancz jedno-vektorovou a blokovou verzi metody LOBPCG. Proveďte rozsáhlé experimentální porovnání všech původních i navržených a implementovaných metod na rozsáhlých řídkých maticích za použití superpočítačů.

Elektronicky schválil/a doc. Ing. Jan Janoušek, Ph.D. dne 9. listopadu 2022 v Praze.

Diplomová práce

HLEDÁNÍ VLASTNÍCH ČÍSEL A VLASTNÍCH VEKTORŮ PRO ROZSÁHLÉ ŘÍDKÉ SYMETRICKÉ MATICE

Bc. Matěj Razák

Fakulta informačních technologií
Katedra teoretické informatiky
Vedoucí: Ing. Daniel Langr, Ph.D.
15. února 2024

Obsah

Seznam obrázků

Poděkování

Děkuji vedoucímu práce Ing. Danielu Langrovi, Ph.D. za trpělivost a vstřícnost při psaní této práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací. Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu) licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 15. února 2024 .....

Abstrakt

Tato práce se věnuje vybraným metodám pro hledání vlastních čísel a vlastních vektorů symetrických řídkým matic nad hybridním paralelním programovacím modelem MPI+OpenMP v jazyce C++. Práce má za cíl vylepšit existující implementaci jedno-vektorového i blokového Lanczosova algoritmu. Vyvinuté alternativní implementace jsou jedno-vektorové i blokové verze Lanczosova algoritmu a LOBPCG metody. Následuje testování a vzájemné porovnání jednotlivých implementací. Měřítkem porovnání je časová efektivita a počet iterací algoritmu. Porovnání s existující implementací prokázala lepší efektivitu mého řešení.

Klíčová slova vlastní číslo, symetrická řídká matice, Lanczosův algoritmus, LOBPCG metoda, MPI, superpočítač

Abstract

This work is devoted to selected methods for finding eigenvalues and eigenvectors of symmetric sparse matrices over the MPI+OpenMP hybrid parallel programming model in the C++ language. The aim of the thesis is to improve the existing implementation of the one-vector and block Lanczos algorithm. Alternative implementations developed are single-vector and block versions of the Lanczos algorithm and the LOBPCG method. This is followed by testing and mutual comparisons of individual implementations. The scale of comparison is time efficiency and the number of iterations of the algorithm. A comparison with an existing implementation showed better efficiency of my solution.

Keywords Eigenvalue, Symmetric sparse matrix, Lanczos algorithm, LOBPCG method, MPI, Supercomputer

Seznam zkratek

BLAS	Basic Linear Algebra Subroutines
CGS	Classic Gram-Schmidt
GCC	GNU Compiler Collection
Ev	Eigenvalue
LAPACK	Linear algebra Package
LOBPCG	Locally Optimal Block Preconditioned Conjugate Gradient
LOPCG	Locally Optimal Preconditioned Conjugate Gradient
LOCG	Locally Optimal Conjugate Gradient
MFDn	Many Fermion Dynamics for nuclear structure
MGS	Modified Gram-Schmidt
MPI	Message-Passing Interface
OMPILancz	Open-source Message-Passing Interface Lanczos
OpenMP	Open Multi-Processing
PETSc	Portable Extensible Toolkit for Scientific Computation
SLEPc	Scalable Library for Eigenvalue problem Computations

Úvod

Tato práce řeší hledání vlastních čísel a vlastních vektorů symetrických řídkých matic nad hybridním paralelním programovacím modelem MPI+OpenMP v jazyce C++. Popisuje implementace jedno-vektorových i blokových verzí Lanczosova algoritmu a LOBPCG metody.

Cílem práce bylo vylepšení stávající implementace Lanczosova algoritmu a návrh a implementace jedno-vektorové a blokové verze LOBPCG metody. Tyto implementace budou pak uloženy do knihovny OMPILancz.

Jádrem práce je implementace a následné rozsáhlé experimentální porovnání původních i navržených a implementovaných metod a jejich variant na rozsáhlých řídkých maticích za použití superpočítače Karolina na TU Ostrava.

Relevantní ukázkou praktického užití těchto algoritmů je nasazení jedno-vektorového Lanczosova algoritmu při řešení výpočtů nukleárních struktur v Ústavu jaderné fyziky AV ČR. Cíle jejich projektu *Posouvání hranic ab initio výpočtů jaderné struktury* jsou popsány takto: "Mnoho současných a budoucích experimentů využívá atomová jádra jako laboratoře k přesným testům fundamentálních symetrií přírody a hledání fyziky za hranicemi Standardního Modelu. Naším cílem je pokytnout klíčové předpovědi pro interpretaci výsledků několika významných experimentů, které se snaží o detekci temné hmoty a hledání odchylek od předpovědí Standardního modelu v elektroslabých jaderných procesech. Za tímto účelem vyvineme spolehý model jaderné interakce a nové metody pro modelování jaderné struktury, které budou použity pro výpočty elektroslabých procesů a roztylu temné hmoty na atomových jádrech."[1] Vstupním zadáním mé práce byla stávajícím (původní) implementace Lanczosova algoritmu. I data použitá na testování 6 pochází z tohoto projektu.

Přípravou pro vlastní řešení jsou kapitoly 1 a 2, které uvádí základní teoretické pojmy a teorii použitých algoritmů. Implementace těchto algoritmů jsou popsány v kapitolách 4 (původní řešení) a 5 (vlastní řešení). Základem původního řešení je jedno-vektorová a bloková verze Lanczosova algoritmu. Hledání nového řešení zahrnuje implementaci a následné testování různých variant Lanczosova algoritmu a metody LOBPCG (jedno-vektorová a bloková verze).

Vlastní řešení Lanczosova algoritmu je rozděleno na jedno-vektorovou a blokovou variantu. Jsou dále modifikovány použitím různých typů reortogonalizace (úplná, iterační a částečná), restartů (implicitní a explicitní), locking, paralelizace pomocí OpenMP a použití specifického úvodního vektoru (místo náhodného).

Kapitola 6 je věnována experimentálnímu testování. Popisuje metodiku testování, vlastní testy, porovnání výsledných implementací algoritmu, test správnosti. Návrh na budoucí vylepšení řešení je v závěru práce.

Kapitola 1

Základní pojmy a definice

Tato úvodní kapitola obsahuje základní pojmy a definice relevantní pro řešení zadaného problému. Základní pojmy a definice jsou převzaty z textu [2].

1.1 Matice

▶ **Definice 1.1** (Matice). *Nechť $m, n \in \mathbb{N}$. Uspořádaný soubor mn čísel uspořádaný v řádcích a sloupcích se nazývá matice s m řádky a n sloupci.*

Matici značíme následovně:

$$A^{m,n} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ & \vdots & & \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix},$$

kde $a_{i,j}$ jsou jednotlivé prvky matice. i je řádkový index a j sloupcový index.

Značení A_n nám udává, že matice A má n sloupců.

Matici o jednou sloupci budeme nazývat *vektorem*.

▶ **Definice 1.2** (Řídká matice). *Řídká matice je taková matice, u které vede použití speciálních formátů neukládajících nulové prvky matice a k nim příslušných algoritmů ke zvýšení efektivity výpočtů. (Demmel, 1997)*

▶ **Definice 1.3** (Symetrická matice). *Symetrická matice je taková čtvercová matice $S^{n,n}$, pro kterou platí,*

$$\forall i, j = 1, \dots, n : s_{ij} = s_{ji}.$$

▶ **Definice 1.4** (Diagonální matice). *Diagonální matice je čtvercová matice $D^{n,n}$ splňující*

$$\forall i, j = 1, \dots, n : i \neq j \implies t_{ij} = 0.$$

▶ **Definice 1.5** (Tridiagonální matice). *Tridiagonální matice je čtvercová matice $T^{n,n}$ splňující*

$$\forall i, j = 1, \dots, n : |i - j| > 1 \implies t_{ij} = 0.$$

▶ **Definice 1.6** (Podobné matice). *Čtvercové matice $A, B \in \mathbb{R}^{n,n}$ si jsou podobné, jestliže existuje regulární matice V taková, že platí vztah*

$$A = VBV^{-1}.$$

▶ **Definice 1.7** (Násobení matic). *Nechť jsou matice $A^{m,r}$ a $B^{r,n}$. Součinem těchto matic je matice $D^{m,n}$ pro jejíž prvky d_{ij} platí vztah*

$$d_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

. Značí se $D = AB$

1.2 Ortogonalita

▶ **Definice 1.8** (Ortogonální vektory). *Nechť máme dva nenulové vektory $u, v \in \mathbb{R}^n$. Tyto vektory jsou ortogonální jestliže platí vztah $u^T v = 0$. Nulový vektor v $\mathbb{R}^n$ je ortogonální ke všem vektorům v $\mathbb{R}^n$.*

▶ **Definice 1.9** (Ortonormální vektory). *Nechť máme dva ortogonální vektory $u, v \in \mathbb{R}^n$. Tyto vektory jsou navíc ortonormální jestliže platí $||u|| = ||v|| = 1$.*

▶ **Definice 1.10** (Ortogonální matice). *Ortogonální matice je čtvercová matice $Q^{n,n}$ splňující $Q^T Q = I$, kde I je jednotková matice.*

1.3 QR rozklad

V mnoha algoritmech na hledání vlastních čísel je zapotřebí najít ortonormální bázi podprostoru generovanou souborem vektorů $\langle x_1, x_2, \ldots, x_n \rangle$. Tedy najít soubor vektorů q_i, pro který platí že, $||q_i|| = 1$, $q_i^T q_j = 0$ jestliže $i \neq j$, a $\langle x_1, x_2, \ldots, x_n \rangle = \langle q_1, q_2, \ldots, q_n \rangle$.

Toho lze dosáhnout pomocí QR rozkladu.

▶ **Definice 1.11** (QR rozklad). *Nechť máme matici $A^{m,n}$ s lineárně nezávislými sloupci (vektory), pak lze matici A rozložit jako*

$$A = QR, \tag{1.1}$$

kde $Q \in \mathbb{R}^{m,n}$ s ortonormálními sloupci a $R \in \mathbb{R}^{n,n}$ je regulární matice v horním trojúhelníkovém tvaru.

QR rozklad lze získat pomocí Gram-Schmidtova procesu.

1.4 Gram-Schmidtův proces

Gram-Schmidtův proces [3] je jedním z algoritmů na výpočet QR rozkladu (viz algoritmus 3). Proces ortonormalizuje vektory postupně. V každé iteraci se zortonormalizuje další vektor. Řekněme, že máme vypočítáno prvních $j - 1$ vektorů. Algoritmus vypočítá ortogonální projekci vektoru x_j na lineární obal $\langle q_1, q_2, \ldots, q_n \rangle$. Tato projekce je odečtena od původního vektoru a výsledek je normalizován. Získáme $\langle q_1, q_2, \ldots, q_j \rangle = \langle x_1, x_2, \ldots, x_j \rangle$, kde q_j je ortogonální k vektorům $q_1, q_2, \ldots, q_{j-1}$ s normou 1. Tento proces je úspěšný pouze tehdy, když soubor vektorů x_i je lineárně nezávislý.

Následují dvě verze Gram-Schmidtova procesu pro ortogonalizaci jednoho vektoru vůči sadě vektorů.

První varianta se nazývá klasický Gram-Schmidt (CGS) (viz algoritmus 1). Druhá varianta je modifikovaný Gram-Schmidt (MGS) (viz algoritmus 2).

V exaktní aritmetice generují obě varianty stejné výsledky. Avšak v aritmetice s omezenou přesností nám mohou vycházet výrazně různé výsledky.

MGS má oproti CGS lepší numerické vlastnosti [3], [4]. Na druhou stranu je však CGS lépe paralelizovatelný než MGS.

Algoritmus 1: Klasický Gram-Schmidt

Vstup: Ortogonalizovaný vektor x_j vůči sadě vektorů Q_{j-1}
Výstup: Ortogonalizovaný vektor q_j a vektor koeficientů r_j

$r_{ij} = Q_{j-1}^T x_j$;
$q_j = x_j - Q_{j-1} r_j$;
$r_{jj} = ||q_j||_2$;

Algoritmus 2: Modifikovaný Gram-Schmidt

Vstup: Ortogonalizovaný vektor x_j vůči sadě vektorů Q_{j-1}
Výstup: Ortogonalizovaný vektor q_j a vektor koeficientů r_j

$q_j = x_j$;
for $i = 1, ..., j-1$ **do**
  $r_{ij} = q_i^T q_j$;
  $q_j = q_j - r_{ij} q_i$;
end
$r_{jj} = ||q_j||_2$;

Algoritmus 3: QR rozklad pomocí Gram-Schmidt

Vstup: Matice $X^{m,n}$
Výstup: Matice $Q^{m,n}$ a $R^{n,n}$ z rovnice 1.1

$r_{11} = ||x_1||_2$;
$q_1 = x_1 / r_{11}$;
for $j = 2, ..., n$ **do**
  $[q_j, r_j]$ = Gram-Schmidt (x_j, Q_{j-1});
  $q_j = q_j / r_{jj}$;
end

1.5 Singulární rozklad matice

▶ **Definice 1.12** (Singulární rozklad matice)**.** *Nechť máme čtvercovou reálnou matici $A^{n,n}$, potom může být rozložena na $A = U\Sigma V^T$, kde*
$U^{n,n}, V^{n,n}$ jsou ortogonální matice a $\Sigma^{n,n}$ je diagonální maticí s kladnými čísly.
Diagonální prvky matice Σ se nazývají singulární hodnoty matice A.

1.6 Vlastní čísla

▶ **Definice 1.13** (Vlastní číslo)**.** *Nechť máme čtvercovou matici $A^{n,n}$, potom nenulový vektor $x \in \mathbb{C}^n$ je nazván vlastním vektorem matice A jestliže existuje nějaké $\lambda \in \mathbb{C}$ splňující vztah*

$$Ax = \lambda x$$

λ se nazývá vlastním číslem matice A a vektor x je příslušejícím vlastním vektorem.

▶ **Věta 1.14.** *Podobné matice mají stejná vlastní čísla.*

Důkaz. $B = VAV^{-1} \iff VBV^{-1} = A$. Jestliže platí $Av = \lambda v$, potom $VBV^{-1}v = \lambda v \implies BV^{-1}v = \lambda V^{-1}v$. Jestliže je pak λ vlastní číslo s vlastním vektorem v matice A, pak má matice B stejné vlastní číslo s vlastním vektorem $V^{-1}v$. Jelikož se mohou ve výpočtu prohodit matice A a B, každé vlastní číslo B je také vlastní číslo A. Tudíž matice A a B mají stejná vlastní čísla. ◀

▶ **Věta 1.15.** *Nechť je $S \in \mathbb{R}^{n,n}$ symetrická matice. Potom jsou všechna vlastní čísla této matice reálná.*

Důkaz. Nechť $Av = \lambda v$, kde $v \neq 0$, $\lambda \in C$. Pak $\lambda \overline{v}^T v = \overline{v}^T(\lambda v) = \overline{v}^T Av = (A^T \overline{v})^T v = (A\overline{v})^T v = (\overline{A}\overline{v})^T v = (\overline{\lambda}\overline{v})^T v = \overline{\lambda}\overline{v}^T v$. Protože $v \neq 0$, pak $\overline{v}^T v \neq 0$ a $\lambda = \overline{\lambda}$. Tudíž vlastní číslo λ je reálné číslo. ◀

Kapitola 2

Použité algoritmy

2.1 Lanczosův algoritmus

Lanczosova metoda [5], [6], [7], [8] je jedna z hlavních využívaných metod na hledání aproximací největších či nejmenších vlastních čísel a jejich přidružených vektorů pro symetrické matice. Využívá jenom operaci matice-vektor násobení pro vstupní matici, což je výhodné pro matice, které jsou příliš rozsáhlé na řešení například QR algoritmem. Lanczosovu metodu můžeme chápat jako specifický případ Arnoldiho metody [9] jenom pro symetrické matice.

Funkce metody stojí na tzv. Krylovových podprostorech [2].

▶ **Definice 2.1** (Krylovův podprostor). *Máme čtvercovou matici $A^{n,n}$ s řádem n, vektor v a $m < n$, pak*

$$K_m(v, A) = \langle v, Av, ..., A^{m-1}v \rangle \tag{2.1}$$

se nazývá Krylovovým podprostorem.

Za využití vlastností tohoto podprostoru se hledá taková matice $V^{n,m}$, aby platil vztah

$$T = V^T AV, \tag{2.2}$$

kde matice $T^{m,m}$ je tridiagonální.

Metoda převádí symetrickou metodu $A^{n,n}$ na tuto tridiagonální matici T pomocí následujícího rekurentního vztahu

$$\beta_{j+1}v_{j+1} = Av_j - \alpha_j v_j - \beta_j v_{j-1}, \tag{2.3}$$

kde v_1 je počáteční vektor s normou 1, $\beta_1 = 0$, $\alpha_j = v_j^T Av_j$, $\beta_{j+1} = v_{j+1}^T Av_j$ a tzv. Lanczosovy vektory v tvoří ortonormální bázi.

Matice T_m je pak definována následovně

$$T_m = \begin{pmatrix} \alpha_1 & \beta_2 & & & \\ \beta_2 & \alpha_2 & \beta_3 & & \\ & \beta_3 & \alpha_3 & \ddots & \\ & & \ddots & \ddots & \beta_m \\ & & & \beta_m & \alpha_m \end{pmatrix} \tag{2.4}$$

V případě hledání všech vlastních čísel n, pak vektor $v_{n+1} = 0$ a platí vztah

$$AV - VT = 0, \tag{2.5}$$

Z toho vyplývá, že Lanczosova metoda vytváří tridiagonální matici ortogonálně podobnou A.

Následný výpočet vlastních čísel z tridiagonální matice pak může být efektivnější než z původní matice A. Nechť (λ_i, y_i) je pár vlastní číslo a jeho vektor, pak aproximace vlastního čísla matice A je λ_i a vektoru je $x_i = V_m y_i$.

Takto popsaná metoda je znázorněna v následujícím pseudokódu 4.

Algoritmus 4: Lanczosův algoritmus - jedno-vektorový

Vstup: Symetrická matice A, počet hledaných vlastních čísel m
Vstup: počáteční vektor v_1 s normou 1
Výstup: V_m, T_m, v_{m+1}, β_{m+1}, tak že $AV_m - V_m T_m = \beta_{m+1} v_{m+1} e_m^T$

```
1 for j = 1, ..., m do
2     u_{j+1} = A v_j;
3     Ortogonalizace u_{j+1} vzhledem k V_j; /* viz pseudokod 2 */
      /* α_{j+1} = u_j^T u_{j+1} dostaneme v předchozím kroku */
4     β_{j+1} = ||u_{j+1}||_2;
5     if β_{j+1} == 0 then
6         Ukonči algoritmus
7     end
8     ; v_{j+1} = u_{j+1} / β_{j+1};
9 end
```

2.1.1 Bloková verze

Kromě jedno-vektorové verze, existuje i bloková verze [10], [11], [12]. Na rozdíl od násobení matice jedním vektorem se násobí blokem vektorů. Tudíž

$$K_m(V, A) = \langle V, AV, ..., A^{m-1}V \rangle, \tag{2.6}$$

kde V má b sloupců.

Výsledná matice T_m je pak blokově tridiagonální.

$$T_m = \begin{pmatrix} A_1 & B_2^T & & & \\ B_2 & A_2 & B_3^T & & \\ & B_3 & A_3 & \ddots & \\ & & \ddots & \ddots & B_m^T \\ & & & B_m & A_m \end{pmatrix} \tag{2.7}$$

$A^{b,b}$ je diagonální a $B^{b,b}$ tvoří horní trojúhelníkovou matice, takže samotné matic $T^{bm,bm}$ je symetrická.

Bloková verze Lanczosova algoritmu je popsána v následujícím pseudokódu 5.

Algoritmus 5: Lanczosův algoritmus - blokový

Vstup: Symetrická matice $A^{n,n}$
Vstup: počáteční ortonormální matici $V_1^{n,b}$
Výstup: V_m, T_m, V_{m+1}, B_{m+1}, tak že $AV_m - V_m T_m = B_{m+1} V_{m+1} E_m^T$

```
1  Z = A V_1;
2  A_1 = V_1^T Z;
3  Z = Z - V_1 A_1;
4  Ortogonalizace Z vzhledem k V_1, V_2, ..., V_j; /* viz pseudokod 4.3 řádek 6-7 */
5  Z = V_2 B_1 /* QR-rozklad - viz pseudokod 3 */
6  for j = 2, ..., m do
7      Z = A V_j; ¨ A_j = V_j^T Z;
8      Z = Z - V_j A_j - V_{j-1} B_{j-1}^T;
9      Ortogonalizace Z vzhledem k V_1, V_2, ..., V_j; /* viz pseudokod 4.3 řádek 6-7 */
10     Z = V_2 B_1 /* QR-rozklad - viz pseudokod 3 */
11 end
```

2.1.2 Reortogonalizace

Jednou nevýhodou Lancozsovy metody je numerická nestabilita. Neboli vektory $v_1, v_2, ..., v_m$ s přibývajícím m můžou ztrácet navzájem ortogonalitu a algoritmus se stane nestabilním. Lanczosovy vektory ztrácí lineární nezávislost a výsledná matice T již nemusí být ortogonálně podobná A. Algoritmus pak může dávat falešné aproximace vlastních čísel matice A.

Proto musíme ortogonalitu v Lancozsových vektorech manuálně udržovat. Na to nám slouží reortogonalizace. V následující části jsou popsány různé alternativy jak reortogonalizaci provádět.

Úplná ortogonalizace: Jedná se o ortogonalizaci, kde se každý nový Lanczosův vektor ortogonalizuje vzhledem ke všem předchozím Lanczosovým vektorům. Hlavní výhoda této varianty je její robustnost. Na druhou stranu se musí udržovat v paměti všechny Lanczosovy vektory a výpočetní složitost je nejvyšší z daných variant. Na výpočet se využívá modifikovaný Gram-Schmidtův algoritmus (2).

Iterativní ortogonalizace: Na rozdíl od úplné ortogonalizace se využívá klasický Gram-Schmidtův algoritmus (1). Kvůli numerické nestabilitě [13], [14] je zapotřebí v určitých případech reortogonalizaci iterovat, dokud numerické chyba není dostatečně malá. Výhodou této varianty je lepší možnost paralelizace než u modifikovaného Gram-Schmidtova algoritmu. Nevýhodou je možné vícenásobné opakování reortogonalizace.

Lokální ortogonalizace: Nejjednodušší varianta, kde každý nový Lanczosův vektor ortogonalizuje vzhledem ke dvěma předchozím vektorům. Hlavními výhodami jsou nízká výpočetní složitost a nutnost pamatovat si pouze dva předchozí Lanczosovy vektory. Nevýhodou je pomalá konvergence aproximací hledaných vlastních čísel a tudíž algoritmus vyžaduje mnohem více iterací. Další nevýhodou je, že se mohou objevovat falešné aproximace vlastních čísel kvůli špatné ortogonalitě.

Z analýzy v Paigeově práci [15], [16], [17] vyplývá, že Lanczosovy vektory začínají ztrácet přesnost jakmile se vlastní čísla matice T stabilizují, neboli než se aproximace vlastních čísel A blíží konvergenci. Do té doby si vektory ortogonalitu udržují stejně dobře při úplné ortogonalizaci jako u lokální. Udržuje se tzv. semiortogonalita. Tohoto poznatku využívají následující dvě varianty.

Selektivní ortogonalizace [18]: V průběhu algoritmu se v intervalech počítají aproximace vlastních vektorů z matice T. Nové Lanczosovy vektory se pak ortogonalizují vzhledem k těm vlastním vektorům, které skoro zkonvergovaly. Otázkou u této metody je, v jakých intervalech počítat tyto aproximace a jak dlouho vydrží ortogonalita nových vektorů k těmto aproximacím.

Částečná ortogonalizace: Druhou variantou je tzv. částečná ortogonalizace. Jelikož je tato varianta implementována v rámci práce, je rozebrána v následující podkapitole ve větším detailu. Používá se ω-rekurence na simulování ztráty ortogonality a reortogonalizace se dělá pouze na podsetu Lanczosových vektorů, kde ztráta ortogonality překročí stanovený limit. Nevýhodou je potřeba udržovat v paměti všechny Lanczosovy vektory.

Částečná ortogonalizace

Částečná ortogonalizace [11], [12], [19], [20] ortogonalizuje vektory u_{j+1} a u_{j+2} vzhledem k pouze podmnožině předchozích Lanczosových vektorů. Pomocí rekurence se udržuje aproximace ztráty ortogonality. Jestliže definujeme úroveň ortogonality mezi Lanczosovy vektory v j-té iteraci jako

$$w_j \equiv \max_{1 \leq k \leq j-1} |w_{j,k}| \text{ ,kde } w_{j,k} \equiv |v_j^T v_k|, \tag{2.8}$$

pak při úplné ortogonalizaci se udržuje přibližně $w_j \approx \epsilon$. To však není nezbytné. Ukazuje se, že stačí udržovat semiortogonalitu $w_j \approx \sqrt{\epsilon}$, aby se v algoritmu neobjevovaly falešné aproximace vlastních čísel [20].

Vektory u_{j+1} a u_{j+2} se tedy ortogonalizují vzhledem k předchozím vektorům u kterých platí, že

$$w_{j+1,k} > \sqrt{\epsilon} \tag{2.9}$$

a k vedlejším vektorům které stále překračují nějakou stanovenou mez η. Tato mez je nastavena v článku [20] na $\epsilon^{3/4}$.

Následuje jeden ze způsobů jakým aproximovat skalární součin vektorů u_{j+1} a v_k:

$$\begin{aligned} w_{k,k} &= 1 \text{ , pro } k = 1, \ldots, j, \\ w_{k,k-1} &= \psi_k \text{ , pro } k = 2, \ldots, j \text{ s } w_{k,0} = 0, \\ w_{j+1,k} &= \frac{1}{\beta_{j+1}}[\beta_{k+1} w + j, k + 1 + (\alpha_k - \alpha_j) w_{j,k} + \beta_k w_{j,k-1} - \beta_j w_{j-1,k}] + \vartheta_{j,k} \text{ , pro } 1 \leq k < j \end{aligned} \tag{2.10}$$

Hodnoty ψ_k $\vartheta_{j,k}$ náhodně vybrané zaokrouhlovací chyby. V článku [20] jsou vybrány následovně:

$$\begin{aligned} \vartheta_{j,k} &= \epsilon(\beta_{k+1} + \beta_{j+1})\Theta, \text{ kde } \Theta \in N(0, 0.3), \\ \psi_{j,k} &= \epsilon n \frac{\beta_2}{\beta_{j+1}} \Psi, \text{ kde } \Psi \in N(0, 0.6) \end{aligned} \tag{2.11}$$

U blokové verze Lanczosova algoritmu funguje tato ortogonalizace obdobně. Jenom výpočet aproximace ztráty ortogonality se v některých bodech mění. Ten provádíme pomocí normy $||W_{j,k}||_2$, kde $W_{j,k} \equiv V_j^T V_k$. $w_{j,k}$ počítáme jako horní mez pro $||W_{j,k}||_2$ následující rekurencí:

$$\begin{aligned} w_{j+1,j} &= \epsilon b \\ w_{j+1,k} &= \tilde{\beta}_j[\beta_k w + j, k + 1 + \beta_{k-1} w_{j,k-1} + \beta_{j-1} w_{j-1,k} + (\alpha_k - \alpha_j) w_{j,k}] \text{ , pro } k = 1, \ldots, j-1, \end{aligned} \tag{2.12}$$

kde b je velikost bloku, α_k, β_k jsou největší singlulární hodnoty matice A_k, B_k a $\tilde{\beta_k} = 1/\rho_b(B_k)$, kde $\rho_b(B_k)$ je nejmenší singulární hondota matice B_k.

2.1.3 Restart

Účelem restartů [5], [11], [21], [22] je snížení paměťových nároků a výpočetní složitosti reortogonalizace, jelikož s rostoucím počtem iterací roste počet Lanczosových vektorů.

Ideou restartů je restartovat algoritmus po m-iteracích a začít novou sadu iterací s vhodnějším počátečním vektorem a vhodnější sadou vektorů v. Jednou z variant restartů je "Thick restart"(také explicitní restart).

Před restartem Lanczosovy vektory splňovaly následující rekurenci

$$AV_m = V_m T_m + \beta_m v_{m+1} e_m^T. \tag{2.13}$$

Během restartu se najdou již zkonvergované vlastní vektory matice T_m, nazveme je y_i. Vypočteme matici $\hat{V}_k = V_m Y$, kde k je počet vektorů y a Y je matice obsahující ve sloupcích vektory y. Pro přehlednost je pro hodnoty po restartu využita stříška.

Po restartu budou nové vektory ve $\hat{V}_k$ splňovat následující rekurenci

$$A\hat{V}_k = \hat{V}_k \hat{T}_k + \beta_m \hat{v}_{k+1} s^T, \tag{2.14}$$

kde $\hat{v}_{k+1} \equiv v_{m+1}$ a $s \equiv Y^T e_m$.

Matice $\hat{T}_k$ je diagonální a diagonální hodnoty jsou vlastní čísla matice T_m odpovídající vlastním vektorům y. V dalším kroku musíme však $\hat{T}_k$ rozšířit o řádek a sloupec. Výsledná $\hat{T}_{k+2}$ bude vypadat následovně

$$T_{k+2} = \begin{pmatrix} \hat{T}_k & \beta_m s & \\ \beta_m s^T & \hat{\alpha}_{k+1} & \hat{\beta}_{k+1} \\ & \hat{\beta}_{k+1} & \hat{\alpha}_{k+2} \end{pmatrix} \tag{2.15}$$

Tudíž v následujících iteracích matice T_m již nebude tridiagonální, jelikož $b_m s$ je vektor o velikosti k.

U blokové verze Lanczosova algoritmu funguje tento restart obdobně. Jediným rozdílem je, že vektor $b_m s$ již nebude vektorem ale maticí o počtu sloupců rovnému velikosti bloku.

Ostatní varianty restartů, jako je třeba implicitní restart [21], se liší ve výběru počátečního vektoru $\hat{v}_{k+1}$ a výpočtu matice $\hat{V}_k$.

2.1.4 Locking

Jednou možností vylepšení restartu je zavedení tzv. *locking* [5]. Jedná se o vyřazení dostatečně zkonvergovaných vlastních čísel a vektorů z aktivního výpočtu.

Jestliže po m iteracích je spuštěn restart a je zkonvergovaných k vlastních čísel může se V_m napsat jako

$$V_m = [V^{(l)}_{1:k} | V^{(a)}_{k+1:m}], \tag{2.16}$$

kde (l) jsou vyřazené vektory a (a) aktivní vektory. Po restartu se bude počítat $m-k$ Lanczosových vektorů. Lanczosovy vektory se stále musí ortogonalizovat vůči vyřazeným vektorům, aby se neobjevovaly duplicitní hodnoty.

2.2 LOBPCG metoda

Další metodou na hledání aproximací největších či nejmenších vlastních čísel a jejich přidružených vektorů pro symetrické matice je LOBPCG (Locally Optimal Block Preconditioned Conjugate Gradient) [23], [24], [25], [26], [27].

Metoda využívá vlastnosti Rayleighova kvocientu [27], že pro nejmenší vlastní číslo matice A platí vztah

$$\lambda_1 = \min_{x \neq 0} \rho(x), \qquad \rho(x) = \frac{x^T A x}{x^T x}. \tag{2.17}$$

Pro případ hledání největšího vlastního čísla se hledá maximum.

Metoda hledá toto minimum (či maximum) iterativně hledáním směru největšího spádu.

Ten se získá pro k-tou iteraci následujícím vztahem

$$r = Ax - \rho(x_k) x_k. \tag{2.18}$$

Tento způsob může konvergovat k řešení velmi pomalu. Proto se může na vektor největšího spádu r aplikovat předkondicionér T. Získáme tím vektor

$$w_k = Tr. \tag{2.19}$$

Podrobně je teoretický postup metody popsán v [27].

2.2.1 Jedno-vektorová verze

Jedno-vektrorová verze hledá pouze nejmenší (či největší) vlastní číslo. Jelikož se nevyužívá blokových operací můžu se tato verze nazývat LOPCG. Nebo při nevyužití žádného předkondicionéru LOCG.

Algoritmus pro hledání nejmenšího vlastního čísla je popsán v následujícím pseudokódu 6.

Algoritmus se lze upravit na hledání více nejmenších vlastních čísel opakovaným spuštěním algoritmu a udržováním ortogonality mezi vektorem spádu a vlastními vektory již nalezených vlastních čísel.

Algoritmus 6: LOBPCG algoritmus - jedno-vektorový

Vstup: Symetrická matice $A^{n,n}$
Vstup: počáteční vektor x_1 s normou 1
Výstup: aproximace μ_k a x_k nejmenšího vlastního čísla a vektoru matice A, kde k je poslední iterace algoritmu

$p_1 = 0$;
for $i = 1, ...,$ *do konvergence* **do**
  $\mu_i = x_i^T A x_i$;
  $r = Ax_i - \mu_i x_i$;
  $\omega_i = Tr$;
  Aplikuj Rayleigh-Ritz metodu na $\langle x_i, \omega_i, p_i \rangle$;
  $x_{i+1} = \underset{y \in \langle x_i, \omega_i, p_i \rangle, y^T y = 1}{\operatorname{argmin}} (y^T A y)$;
  $p_{i+1} = x_{i+1} - x_i$;
  Zkontroluj konvergenci;
end

Aplikace Rayleigh-Ritz metody je popsána např. v [27].

2.2.2 Bloková verze

Jedno-vektorová varianta může konvergovat k řešení velmi pomalu, zejména pokuď jsou hledaná vlastní čísla shloučená blízko u sebe. Proto se může vyplatit hledat více nejmenších vlastních čísel najednou. K tomu slouží bloková verze LOBPCG. V případě nevyužití předkondicionéru můžeme metodu nazývat LOBCG.

Algoritmus 7 je stejný jako u jedno-vektorové verze až na přidání blokových operací.

Algoritmus 7: LOBPCG algoritmus - blokový

Vstup: Symetrická matice $A^{n,n}$
Vstup: počáteční ortonormální matice vektor $X_1^{n,b}$ s normou 1
Výstup: aproximace M_k a X_k b nejmenších vlastních čísel a vektorů matice A, kde k je poslední iterace algoritmu

$P_1 = 0$;
for $i = 1, ...,$ *do konvergence* **do**
  $M_i = X_i^T A X_i$;
  $R = AX_i - M_i X$;
  $\Omega_i = TR$;
  Aplikuj Rayleigh-Ritz metodu na $\langle X_i, \Omega_i, P_i \rangle$;
  $X_{i+1} = \underset{Y \in \langle X_i, \Omega_i, P_i \rangle, Y^T Y = I_b}{\operatorname{argmin}} (Y^T A Y)$;
  $P_{i+1} = X_{i+1} - X_i$;
  Zkontroluj konvergenci;
end

Aplikace Rayleigh-Ritz metody je popsána např. v [27].

Kapitola 3
Externí využitý software

Tato kapitola se zabývá ostatním softwarem, který implementaci využívá.

3.1 OpenMP

Na paralelní provedení v rámci procesu pomocí vláken se využívá knihovna OpenMP [28].OpenMP je vysoko-úrovňové API na programování vícevláknových implementací nad virtuálně sdílenou pamětí.

OpenMP využívá tzv. fork-join model paralelizace. V určitých částech implementace jsou vytvářena, prováděna a následně ukončována vlákna. V ostatních částech je pouze hlavní vlákno.

OpenMP se skládá z direktiv, proměnných a operací.

Při použití OpenMP knihovny GCC kompilátor musí mít argument *-fopenmp*.

V implementaci se využívá iterační datový paralelismus pomocí direktiv *parallel* a *for*, jelikož se jedná zejména o práce s maticemi a vektory.

Direktiva *parallel* označuje region implementace, který se má provádět paralelně. Regionu se přiřadí jedno či více vláken.

Direktiva *for* sděluje, že iterace příslušného cyklu se provádějí paralelně pomocí vláken daného regionu.

3.2 MPI

MPI (Message-Passing Interface) [29] je knihovna implementující protokol pro předávání zpráv z jednoho procesu na druhý. Jedná se o zprávy mezi dvěma procesy i hromadné.

Jednou z důležitých oblastí jsou komunikátory, které tvoří skupiny procesů během běhu programu.

Následuje list hlavních použitých MPI funkcí:

- *MPI_Bcast* pošle zprávu z daného procesu všem ostatním procesům v daném komunikátoru
- *MPI_Reduce* redukuje hodnoty ze všech procesů daného komunikátoru na jednu hodnotu
- *MPI_Send* pošle zprávu z daného procesu jinému procesu
- *MPI_Recv* proces přijme zprávu od určitého procesu
- *MPI_Allreduce* kombinuje hodnoty od všechny procesů daného komunikátoru a výsledné hodnoty rozešle zpět

- *zápis a čtení ze souboru* různé MPI funkce na práci se souborem

Implementace používá pouze blokující MPI funkce, neboli funkce jsou ukončeny, až když je vstupní buffer možné modifikovat.

3.3 LAPACK

LAPACK (Linear Algebra PACKage) [30] je knihovna napsaná ve Fortranu s rutinami na výpočet nejběžnějších problémů z lineární algebry. LAPACK využíva knihovnu BLAS (Basic Linear Algebra Subroutines). BLAS obsahuje nízkoúrovňové rutiny na výpočet běžných výpočtů z lineární algebry.

Knihovna LAPACK je využita v implementaci na výpočet vlastních čísel a vektorů a na výpočet singulárních hodnot menších matic.

Při použití LAPACK knihovny GCC kompilátor musí mít argument *-llapack*.

Následuje list použitých LAPACK operací:

- *?sbevx* počítá vlastní čísla a vektory reálné symetrické band (subdiagonály jsou blokové) matice
- *?stevx* počítá vlastní čísla a vektory reálné symetrické tridiagonální matice
- *?syevx* počítá vlastní čísla a vektory reálné symetrické matice
- *?gesvd* provede singulární rozklad reálné matice

Otazník v operacích značí možné přesnosti výpočtu, *s* značí jednoduchou (single) a *d* dvojitou (double) přesnost.

Kapitola 4
Původní řešení

Základním podkladem textu této kapitoly je nepublikovaná prezentace D. Langra a T. Dytrycha [31].

Kapitola se zabývá již existující implementací na hledání vlastních čísel a vektorů určenou na výpočetní cluster. Jedná se o open-source knihovnu OMPILancz (`https://gitlab.com/daniel.langr/ompilancz`). Knihovna obsahuje implementaci Lanczosova algoritmu ve dvou verzích, jedno-vektorové a blokové.

V následující sekci je rozebrán princip mapování procesů a komunikace mezi nimi (viz 4.1). Následuje popis jedno-vektorové (viz 4.2) a blokové (viz 4.3) implementace. V další sekci (viz 4.4) jsou prodiskutovány různé návrhy na zlepšení stávající implementace. V poslední sekci (viz 4.5) jsou zmíněny ostatní relevantní implementace a jejich omezení.

4.1 Mapování procesů

Mnoho programů na výpočetní cluster pracující s rozsáhlými maticemi je musí mapovat mezi jednotlivé procesy. OMPILancz knihovna využívá tzv. šachovnicové rozdělení matice. Každému procesu je přidělen vlastní blok. Vzhledem k tomu, že jediná operace s rozdělovanou maticí je násobení matice-matice či matice-vektor, toto rozdělení je efektivnější vzhledem k míře komunikace mezi procesy a k paměťovým nárokům než mnoho dalších rozdělení.

Následuje přesný popis implementace tohoto rozdělení.

Nechť je $A^{n,n}$ reálná symetrická matice. Jelikož je A symetrická, mohou si procesy ukládat pouze jednu trojúhelníkovou část. Budeme předpokládat, že $\hat{A}$ je horní část. Nechť pro čísla N a $n_0, ..., n_{N-1}$ platí $n_0, ..., n_{N-1} = n$. Pak $\hat{A}$ můžeme rozdělit na bloky $\hat{A}_{I,J}$ o velikosti $n_I * n_J$, kde $0 \leq I, J < N$.

Počet procesorů je $p = N(N+1)/2$, tak každému patří právě jeden blok $\hat{A}_{I,J}$. Nazvěme proces, kterému patří blok $\hat{A}_{I,J}$, procesem $P_{I,J}$. Procesy, u kterých platí $I = J$, nazveme *diagonální procesy*.

Všechny procesy patří do MPI *world* komunikátoru. Implementace zavádí další tři:

- *Řádkový(r_comm)*: Komunikace mezi procesy na stejném řádku. Procesy se stejným I.
- *Sloupcový(c_comm)*: Komunikace mezi procesy na stejném sloupci. Procesy se stejným J.
- *Diagonální(d_comm)*: Komunikace mezi diagonálními procesy.

Tyto komunikátory jsou v naimplementovány ve třídě *mapping*.
Každá skupina obsahuje nejvíce N procesů.

Kromě namapování matice A, implementace mapuje vektory. Konkrétně Lanczosovy vektory. Každý vektor v je rozdělen na části $v_0, ..., v_{N-1}$, kde v_I má n_I prvků. v_I jsou namapovány na diagonální procesy $P_{I,I}$. Toto rozdělení je zavedeno, jelikož diagonální procesy udržují pouze polovičku bloku matice A. Dále procesy držící části vektoru v jsou tak na stejném komunikátoru.

4.2 Jedno-vektorová verze

V následujícím pseudokódu 8 je implementace jedno-vektorové verze Lanczosova algoritmu bez restartu. V každé iteraci je jenom jedna operace se vstupní maticí. A to násobení matice vektorem. Implementace využívá na zachování ortogonality mezi Lanczosovy vektory úplnou ortogonalizaci.

Algoritmus 8: OMPILANCZ - Lanczosuv algoritmus - jedno-vektorový

Vstup: reálná symetrická matice A
Vstup: počet hledaných vlastních čísel n_λ
Vstup: maximální počet iterací q ($q \geq n_\lambda$)
Vstup: tolerance konvergence ϵ
Výstup: aproximace n_λ nejmenších vlastních čísel $\lambda_1, ..., \lambda_{n_\lambda}$ matice A
Výstup: aproximace n_λ přidružených vlastních vektorů $x_1, ..., x_{n_\lambda}$ matice A

1 v_1 = náhodný nenulový znormovaný vektor;
2 **for** $j = 1, ..., q$ **do**
3 $\quad u_{j+1} = Av_j$;
4 $\quad \alpha_j = u_{j+1}^T v_j$
5 $\quad$ **for** $i = 1, ..., j$ **do**
6 $\quad\quad u_{j+1} = u_{j+1} - (u_{j+1}^T v_i)v_i$
7 $\quad$ **end**
8 $\quad \beta_{j+1} = ||u_{j+1}||_2$;
9 $\quad$ **if** $j \geq n_\lambda$ **then**
10 $\quad\quad T^j \leftarrow$ tridiagonální matice, kde $T^j_{k,k} = \alpha_k$ a $T^j_{k+1,k} = T^j_{k,k+1} = \beta_{k+1}$;
11 $\quad\quad$ vyřeš $T^j y_i^j = \theta_i^j y_i^j$ pro $i = 1, ..., n_\lambda$;
12 $\quad\quad$ **if** $\beta_{j+1}|e_j^T y_i| < \epsilon$ *pro* $i = 1, ..., n_\lambda$ **then**
13 $\quad\quad\quad$ skonči smyčku
14 $\quad\quad$ **end**
15 $\quad$ **end**
16 **end**
17 **for** $i = 1, ..., n_\lambda$ **do**
18 $\quad \lambda_i = \theta_i$; $x_i = Vy_i$, kde $V = (v_1, ..., v_{n_\lambda})$;
19 **end**

4.2.1 MPI komunikace

Tato sekce popisuje operace v algoritmu 8, v kterých se využívá komunikace mezi procesy.

- *Generování iniciálního vektoru* v_1 Operace se účastní pouze diagonální procesy. Každý diagonální proces si vytvoří svoji část vektoru s náhodnými hodnotami od 0 do 1. Následně se celý vektor znormuje za využití skalárního součinu dvou vektorů (viz níže).

- *Násobení vstupní matice A vektorem v zprava (3. řádek)* Všechny procesy jsou využité. Operace lze rozdělit do tří částí. Nejprve se rozešlou části x z diagonálních procesů na všechny ostatní. K tomu se využije MPI funkce *MPI_Bcast* za využití *r_comm* a *c_comm* komunikátorů. V dalším kroku každý proces vypočítá svojí část násobení. Toto násobení je dvojnásobné, jelikož každý proces musí provést operaci i pro korespondující dolní část A. Operace pro jednotlivé procesy je dodána uživatelem. To umožní podporu různého uložení matice A. V poslední části procesy pošlou svoje dílčí výsledky diagonálním procesům pomocí MPI funkce *MPI_Reduce* zase za využití *r_comm* a *c_comm* komunikátorů.

- *Skalární součin dvou vektorů (4.,6.,8. řádek)* Operace se účastní pouze diagonální procesy. Každý diagonální proces spočítá skalární součin svých částí vektoru a výsledky jsou redukovány všem diagonálním procesům pomocí MPI funkce *MPI_Allreduce* za využití *d_comm* komunikátoru.

- *Tridiagonální řešič vlastních čísel a vektorů (11. řádek)* Operaci počítá pouze *root* proces $P_{0,0}$. Počítají se vlastní čísla a vektory tridiagonální matice T^j za využití funkce knihovny LAPACK *?stevx*.

- *Kontrola konvergence (12. řádek)* Operaci provádí pouze *root* proces $P_{0,0}$. Avšak všechny procesy o výsledku musejí být informováni. Toho je dosáhnuto pomocí MPI funkce *MPI_Bcast* za využití *world* komunikátoru.

- *Výpočet vlastních vektorů matice A (18. řádek)* Konkrétně se jedná o násobení matice V vektorem y zprava. Operace se účastní pouze diagonální procesy. Vlastní vektory matice T^j jsou rozeslány *root* procesem $P_{0,0}$ ostatním diagonálním procesům pomocí MPI funkce *MPI_Bcast* za využití *d_comm* komunikátoru. Každý diagonální proces pak vypočítá svojí část výsledných vlastních vektorů.

4.3 Bloková verze

V této sekci je popsána implementace 9 blokové verze Lanczosova algoritmu.

Stejně jako u předchozí verze se jedná o implementaci bez restartu s využitím úplné ortogonalizace na udržení ortogonality mezi Lanczosovy vektory. V každé iterace je zase jenom jedna operace se vstupní maticí. Tentokrát je to však násobení matice maticí, kde druhá matice má počet řádků podle velikosti bloku.

Algoritmus 9: OMPILANCZ - Lanczosův algoritmus - blokový

Vstup: reálná symetrická matice R
Vstup: počet hledaných vlastních čísel n_λ
Vstup: maximální počet iterací q (q)
Vstup: tolerance konvergence ϵ
Vstup: velikost bloku s
Výstup: aproximace n_λ nejmenších vlastních čísel $\lambda_1, ..., \lambda_{n_\lambda}$ matice A
Výstup: aproximace n_λ přidružených vlastních vektorů $x_1, ..., x_{n_\lambda}$ matice A

1 $V_1 = s$ náhodných ortonormálních vektorů;
2 **for** $j = 1, ..., q$ **do**
3 $\quad U_{j+1} = RV_j$;
4 $\quad A_j = U_{j+1}^T V_j$;
5 $\quad U_{j+1} = U_{j+1} - A_j V_j$;
6 $\quad$ **for** $i = 1, ..., j$ **do**
7 $\quad\quad U_{j+1} = U_{j+1} - (U_{j+1}^T V_i) V_i$
8 $\quad$ **end**
9 $\quad V_2 B_1 = U_{j+1}$;
10 $\quad T^j \leftarrow$ bloková tridiagonální matice, kde $T^j_{k,k} = A_k$ a $T^j_{k+1,k} = T^j_{k,k+1} = B_{k+1}$;
11 $\quad$ vyřeš $T^j y_i^j = \theta_i^j y_i^j$ pro $i = 1, ..., n_\lambda$;
12 $\quad$ **if** $j + s > n_\lambda$ **then**
13 $\quad\quad$ **if** $B_{j+1} | E_j^T y_i | < \epsilon$ *pro* $i = 1, ..., n_\lambda$ **then**
14 $\quad\quad\quad$ skonči smyčku
15 $\quad\quad$ **end**
16 $\quad$ **end**
17 **end**
18 **for** $i = 1, ..., n_\lambda$ **do**
19 $\quad \lambda_i = \theta_i$; $x_i = V y_i$, kde $V = (v_1, ..., v_{n_\lambda})$;
20 **end**

4.3.1 MPI komunikace

Následovat bude popis operací blokové verze (viz 9), v kterých se využívá komunikace mezi procesy.

- *Generování iniciální sady vektorů V_1 (1. řádek)* Operace se účastní pouze diagonální procesy. Každý diagonální proces si udržuje podmnožinu řádků V_1, kde jednotlivé vektory jsou

po sloupcích. Postupně se přidávají vektory s náhodnými hodnotami od 0 do 1 do V_1 a ortonormalizují se vzhledem k předchozím. Komunikace mezi procesy je potřeba pouze u skalárního součinu dvou vektorů (viz jedno-vektorová verze).

- *Násobení vstupní matice R maticí V zprava (3. řádek)* Všechny procesy jsou využité. Nejdříve si každý diagonální proces převede matici V na lineární pole po řádkách matice V. Dále operace probíhá obdobně jako u jedno-vektorové verze. Na konci výpočtu, jelikož je výsledná matice zapsaná opět jako lineární pole, musí diagonální procesy převést výsledek zpět na matici. Matice se zapisuje jako lineární pole pro efektivnější přenos dat mezi procesy.

- *Násobení transponované matice maticí (4.,7. řádek)* Výpočtu operace se účastní pouze diagonální procesy. Výsledek je zapisován jako lineární pole. K jediné komunikaci mezi procesy dochází při skalárním součinu dvou vektorů (viz jedno-vektorová verze). Následkem této operace na řádku 4 je, že každý diagonální proces má v paměti matici A, která se procesům hodí v následující operaci.

- *Škálované odčítání matice maticí (5.,7. řádek)* Operace se účastní diagonální procesy. Není potřeba žádné komunikace mezi procesy.

- *QR rozklad matice (9. řádek)* Operace se provádí na diagonálních procesech. Provede se QR rozklad pomocí Gram-Schmidtova procesu (viz 3). Uskutečňují se dvě různé komunikace mezi diagonálními procesy. Jedna je při škálovaném odčítání matice maticí (viz výše) a druhá je při skalárním součinu dvou vektorů (viz jedno-vektorová verze).

- *Blokový tridiagonální řešič vlastních čísel a vektorů (11. řádek)* Operaci provádí pouze *root* proces $P_{0,0}$. Počítají se vlastní čísla a vektory blokové tridiagonální matice T^j za využití funkce knihovny LAPACK *?sbevx*. Matice A a B zapsáné jako lineární pole jsou nejdříve převedeny do LAPACK pásmové formy potřebnou pro funkci *?sbevx*.

- *Kontrola konvergence (12. řádek)* Operaci provádí pouze *root* proces $P_{0,0}$. Není nutná žádná komunikace mezi procesy až na koncové rozeslání výsledku všem procesům stejně jako v jedno-vektorové verzi.

- *Výpočet vlastních vektorů matice A (19. řádek)* Operace se účastní pouze diagonální procesy a probíhá obdobně jako u jedno-vektorové verze.

4.4 Možná vylepšení

V této podkapitole jsou stručně popsány idee na možná vylepšení výše zmíněných metod knihovny OMPILancz.

Více podrobně jsou rozvedené v kapitole Implementace.

- *Přidání restartu*: S přibývajícím počtem iterací Lanczosova algoritmu rostou jak paměťové nároky nutným držením všech Lanczosových vektorů, tak i výpočetní složitost způsobená udržováním ortogonality mezi Lanczosovy vektory. Možným řešením obou nedostatků je po určitém počtu iterací algoritmus restartovat a periodicky tak snižovat počet Lanczosových vektorů.

- *Přidání paralelizaci pomocí OpenMP*: Metody knihovny OMPILancz obsahují paralelizaci pouze mezi procesy pomocí MPI. Nabízí se přidělat paralelizaci nad sdílenou pamětí pomocí OpenMP.

- *Různé reortogonalizace*: Jiným způsobem jak zlepšit výpočetní složitost při větším počtu iterací Lanczosova algoritmu je zvolit jiný způsob reortogonalizace Lanczosových vektorů. Jak bylo zmíněno v teoretické části, stačí udržovat semiortogonalitu mezi vektory.

Jako další varianta se nabízí ortogonalizace pomocí klasického Gram-Schmidtova procesu, který je lépe paralelizovatelný.

Na druhou stranu pro velké výpočty může být zapotřebí opakovat reortogonalizaci v jedné iteraci v případě velkých změn během operace.

- *Vyřazení již zkonvergovaných vlastních čísel z aktivního výpočtu*: Při hledání více vlastních čísel již nemusí být nutné dále zlepšovat již dostatečně zkonvergovaná čísla. Proto se mohou vyřadit z aktivního výpočtu (tzv. locking).
- *Zadání specifického úvodního Lanczosova vektoru*: Zapracování možnosti zadání konkrétního úvodního Lanczosova vektoru namísto náhodného. Při vhodném zadání by algoritmus mohl vyžadovat méně iterací než s náhodným vektorem.

4.5 Jiné implementace

Jednou z nejrozšířenější knihovnou implementující výpočet vlastních čísel je SLEPc [32]. Jedná se o nadstavbu knihovny PETSc, která se zaobírá operacemi z lineární algebry za využití MPI. Nevýhodou této knihovny je však využití pouze řádkového rozložení procesů na matici. Neboli každý proces vlastní podmnožinu řádků matice.

Další implementací je knihovna Anasazi. Je součástí software projektu Trilinos [33]. Nevýhodou je nemožnost pracovat specificky se symetrickou maticí. Nelze tedy využít možnosti ukládat pouze polovinu matice.

Jednou z možností, která podporuje šachovnicové rozložení procesů i práci se symetrickou maticí, je součástí "MFDn nuclear structure solver". Tato implementace však není veřejně přístupná.

Kapitola 5
Implementace

Tato kapitola popisuje implementační část práce. Jako programovací jazyk byl použit výhradně C++. Implementaci lze rozdělit do čtyř hlavních tříd. První dvě jsou třídy *Lanczos_eigensolver* a *Lanczos_block_eigensolver*. Tyto třídy implementují jedno-vektorový a blokový Lanczosův algoritmus. Základní verze těchto tříd pocházejí z knihovny *OMPILancz*. V sekci 5.4 se podrobněji popisují naimplementované změny, které jsou zhruba zmíněny v sekci 4.4.

Další dvě třídy jsou *LOPCG_eigensolver* a *LOBPCG_eigensolver*. Implementace těchto tříd je popsána v sekci 5.5. Stejně jako u Lanczosova algoritmu třídy implementují jedno-vektorovou i blokovou verzi LOBPCG metody.

5.1 Vstup

Na použití implementace musí uživatel dodat symetrickou matici A a operátor, který násobí tuto matici vektorem x zprava (Ax). U blokových verzí se násobení provádí blokem vektorů. Tímto násobením se zabývá třída *matrix_times_vector* (u blokových verzí třída *matrix_times_block_vector*).

Každý proces se zabývá dvěmi násobeními (horní a dolní trojúhelník matice), jelikož se jedná o symetrickou matici a stačí uchovávat pouze jednu polovinu matice.

Operátor má čtyři parametry (u blokového operátoru pět parametrů):

- x část vektoru odpovídající části v horním trojúhelníku matice
- y vektor výsledků z horní trojúhelníkové matice
- xt část vektoru odpovídající části v dolním trojúhelníku matice
- yt vektor výsledků z dolní trojúhelníkové matice
- s velikost bloku

Blok vektorů se do operátoru na výpočet linearizuje.

5.2 Spuštění

Pro spuštění je zapotřebí mít nainstalovanou knihovnu LAPACK. Kompilátor se spouští s argumentem *-llapack*.

Následující třídy jsou naimplementovány do *namespace ompilancz*. Všechny hlavní třídy (*Lanczos_eigensolver*, *Lanczos_block_eigensolver*, *LOPCG_eigensolver*, *LOBPCG_eigensolver*) mají konstruktor se stejnými parametry.

Těchto pět parametrů (+ jeden nepovinný) jsou následující:

- *I,J* koordináty procesu (viz 4.1)
- *N* počet diagonálních procesů
- *n,m* dimense dílčí matice odpovídající danému procesu
- *os* nepovinný *ostream* pro možnost výpisu iterací algoritmu jinam než do terminálu, tento *ostream* lze nastavit také voláním metody *setStream* s *os* jako argument

Výpočty u výše zmíněných tříd se spouští metodou *solve*. Tato metoda má čtyři (u blokových verzí pět) následujících parametrů:

- *matvec_operator* operátor na násobení vstupní matice dodaný uživatelem
- *nev* počet hledaných vlastních čísel
- *maxit* maximální počet iterací
- *s* u blokových verzí velikost bloku vektorů
- *eps* požadovaná přesnost reziduální chyby aproximace hledaných vlastních čísel

5.3 Výstup

Metoda *solve* vypisuje každou iteraci momentální aproximaci prvních dvou a posledního aktivně hledaného vlastního čísla. Dále se vytváří logový soubor *eigenvalues.dat* s aproximacemi všech hledaných vlastních čísel po každé iteraci.

Po výpočtu metody *solve* lze volat několik metod na zobrazení výsledků.

Metoda *residuals* s možným *ostream* argumentem vypisuje výsledné aproximace hledaných vlastních čísel společně s jejich výslednou reziduální chybou.

Metoda *print_times* s možným *ostream* argumentem vypisuje celkový čas výpočtu a další dílčí relevantní časy pro daný výpočet.

Metoda *store_eigenvectors* s argumentem *prefix* názvu souborů. Metoda uloží každý výsledný vlastní vektor do souboru *prefixXXX.dat*. Výpočet je naimplementován v metodě *store_to_files* třídy *block_diagonal_vector* a v metodě *store_to_file* třídy *diagonal_vector*.

5.4 Lanczosův algoritmus

Kapitola je rozdělena do několika sekcí. Každá sekce se zabírá jednotlivou změnou vzhledem k *OMPILancz*, které jsou zmíněné v sekci 4.4. Jednotlivé sekce zahrnují jedno-vektorovou i blokovou verzi Lanczosova algoritmu.

V sekcích 5.4.1 a 5.4.2 jsou popsány různé restarty. Poté následuje sekce 5.4.3 popisující možnost vyřazení již zkonvergovaných vlastních čísel z aktivního výpočtu.

Dále jsou dvě sekce (5.4.4 a 5.4.5) zabývající se dalšími naimplementovanými způsoby ortogonalizace Lanczosových vektorů vedle původní úplné ortogonalizace z *OMPILancz*.

Sekce 5.4.6 se zabývá zavedením paralelizace do výpočtu pomocí OpenMP.

Poslední změna, která umožňuje zadat konkrétní úvodní Lanczosův vektor, je popsána v sekci 5.4.7.

Komunikace mezi MPI procesy ve třídách *Lanczos_eigensolver* a *Lanczos_block_eigensolver* byla zachována z původního řešení (viz 4).

5.4.1 Implicitní restart

Implicitní restart je naimplementován ve třídě *implicitRestart*. Je naimplementován pouze pro jedno-vektorovou variantu Lanczosova algoritmu. Viz pseudokód 10.

Uživatel vytvoří *shared_ptr* na třídu *implicitRestart* a dále nastaví restart pomocí metody *setupRestart*.

Tato metoda má tří parametry:

- *restart shared_ptr* ukazující na třídu *implicitRestart*
- *restart_iteration_* zadání, jak často se restart spouští, neboli počet Lanczosových vektorů nepřekročí mez v *restart_iteration_*
- *locking* nastavuje, je-li zapnutá funkce locking (viz 5.4.3)

Restart je spouštěn z metody *solve* třídy *Lanczos_eigensolver* pomocí volání metody *execute*.

Výpočet implementace implicitního restartu prokazoval nestabilitu zejména pro delší výpočty. Byl proveden pokus o napravení této nestability změnou všech relevantních proměnných z typu *double* na *long double*. Nestabilita se však touto změnou nezlepšila.

Algoritmus 10: Lanczosův algoritmus - Implicitní restart

Vstup: reálná symetrická matice A
Vstup: počet hledaných vlastních čísel n_λ
Vstup: maximální počet iterací q ($q \geq n_\lambda$)
Vstup: tolerance konvergence ϵ
Výstup: aproximace n_λ nejmenších vlastních čísel $\lambda_1, ..., \lambda_{n_\lambda}$ matice A
Výstup: aproximace n_λ přidružených vlastních vektorů $x_1, ..., x_{n_\lambda}$ matice A

```
1  Vypočítej Lanczosův algoritmus pro (k + p) iterací;
2  while konvergence nebo maximální počet iterací není dosáhnut do
3      Vypočítej vlastní čísla tridiagonální matice T^{k+p};
4      shifts = p nepotřebných vlastních čísel;
5      QQ = I^{k+p};
6      for i = 1, ..., p do
7          μ = i-tý shift;
8          [Q R] = QR rozklad (T^{k+p} − μI);
9          QQ = QQ * Q;
10         T^{k+p} = RQ + μI;
11     end
12     V^k = V^{k+p} QQ;
13     V^{k+1} = v^{k+p+1};
14     β^k = β^{k+p} * QQ[k + p : k + p; k : k];
15     přidej β^k k T^k;
16     Pokračuj Lanczosův algoritmus pro p steps, počínaje s V^{k+1} a T^k;
17 end
```

- *MPI (řádky 12-13)* Výpočtu se účastní všechny diagonální procesy. Avšak kromě *root* procesu a jednoho dalšího, ostatní diagonální provádějí výpočet pouze na řádkách 12-13.
- *MPI (cyklus 6-11)* Proces $P_{1,1}$ se podílí na cyklu výpočtem 9. řádku paralelně s výpočtem zbytku cyklu *root* procesem. *Root* proces mu vždy předá matici Q pomocí *MPI_Bcast*. Na konci cyklu pak proces $P_{1,1}$ rozešle matici QQ všem ostatním diagonálním procesům. Zbytek výpočtu s T maticí je pak prováděn již *root* procesem.
- *Řešič vlastních čísel (řádek 3)* Jelikož tento restart zachovává tridiagonalitu matice T, je na tento výpočet použita standardně funkce knihovny LAPACK *?stevx*.

5.4.2 Explicitní restart

Explicitní restart je naimplementován ve třídě *thickRestart*. Oproti implicitnímu restartu je přístupný pro obě varianty Lanczosova algoritmu. Až na práci s rozdílným počtem vektorů je algoritmicky totožný pro jedno-vektorovou a blokovou variantu.

Uživatel nastavuje tento restart totožně jako u implicitního restartu v předcházející kapitole (5.4.1).

Také je spouštěn v metodě *solve* voláním metodou *execute*, jejíž implementaci lze vidět v pseudokódu 11. Pseudokód je ukázán pro jedno-vektorovou variantu, avšak pro blokovou variantu je analogický.

Algoritmus 11: Lanczosův algoritmus - Explicitní restart

Vstup: reálná symetrická matice A
Vstup: počet hledaných vlastních čísel n_λ
Vstup: maximální počet iterací q ($q \geq n_\lambda$)
Vstup: tolerance konvergence ϵ
Výstup: aproximace n_λ nejmenších vlastních čísel $\lambda_1, ..., \lambda_{n_\lambda}$ matice A
Výstup: aproximace n_λ přidružených vlastních vektorů $x_1, ..., x_{n_\lambda}$ matice A

1 Vypočítej Lanczosův algoritmus pro $(k + p)$ iterací;
2 **while** *konvergence nebo maximální počet iterací není dosáhnut* **do**
3 Y = prvních k vlastních vektorů matice T^{k+p};
4 Matice T^{k+1} je sestavena podle popisu v kapitole o explicitním restartu (2.1.3);
5 $V^k = V^{k+p}Y$;
6 $V^{k+1} = [V^k,\ v_{k+p+1}]$;
7 Pokračuj Lanczosův algoritmus pro dalších p iterací s maticemi V^{k+1} a T^{k+1};
8 **end**

- *MPI* Výpočtu se podílejí všechny diagonální procesy. *Root* proces má již vypočtené potřebné vlastní vektory Y. Tyto vektory přepošle ostatním diagonálním procesům ve 3. řádku pseudokódu pomocí *MPI_Bcast*.

- *Řešič vlastních čísel* Jelikož tento restart nezachovává tridiagonalitu matice T u jedno-vektorové verze Lanczosova algoritmu, po prvním restartu již nemůže algoritmus využívat jako řešič funkci knihovny LAPACK *?stevx*. Začne se využívat funkce *?sbevx* na výpočet blokově tridiagonálních matic pro obě verze Lanczosova algoritmu.

5.4.3 Locking

Dalším možným vylepšením stávající verze Lanczosova algoritmu je přidání možnosti vyřazení tzv. locking již dostatečně zkonvergovaných vlastních čísel z aktivního výpočtu.

Locking je naimplementován v rámci restartu, jelikož jeho výpočet je založen na stejném principu jako restart. Tudíž je možný pouze s aktivním restartem. Implementace pak spouští restart i v případě, když nejmenšího právě hledané vlastní čísla dostatečně zkonverguje.

U blokové verze Lanczosova algoritmu se locking provádí, pouze když počet zkonvergovaných vlastních čísel je roven násobku velikosti bloku. Důvodem je, že v implementaci počet hledaných vlastních čísel musí být dělitelný velikostí bloku.

Locking je realizován v obou restartech a je nastavován metodou *setupRestart* parametrem *locking*.

5.4.4 Částečná reortogonalizace

Částečná reortogonalizace je naimplementována ve třídě *PartialOrtho* nadtřídy *reorthogonalization*. Pseudokód lze vidět níže 12.

Implementace je obdobná pro jedno-vektorovou a blokovou verzi Lanczosova algoritmu. Hlavním rozdílem je výpočet proměnné ω (viz 2.1.2).

Uživatel nastaví tuto reortogonalizaci voláním metody *setupReorthogonalization*. Jako argument se použije *shared_ptr* na třídu *PartialOrtho*.

Tato reortogonalizace se nemůže používat sp´olečně s restartem.

Algoritmus 12: Lanczosův algoritmus - částečná reortogonalizace

Vstup: sada Lanczosových vektorů Q^{j+1}
Výstup: semi-zorthogonalizovane vektory Q^{j+1}
Výstup: α a β pro tridiagonalni matice
Inicializace: $first\_step = true$;
Start:
if $j > 2$ **then**
  $Q[j+1] = Q[j+1] - beta[j-1]Q[j-1]$;
end
$\alpha = Q[j+1]^T Q[j]$
$Q[j+1] = Q[j+1] - \alpha Q[j]$;
$\beta = ||Q[j+1]||$;
$alpha[j] = \alpha$;
$beta[j] = \beta$;
for $k = 1, \ldots, j$ **do**
  Vypočítej rekurenci pro ω_{j+1k};
end
if $first\_step$ **then**
  for $k = 1, \ldots, j$ **do**
    if $|\omega_{j+1k}| > \sqrt{\epsilon}$ **then**
      nalez r_i a s_i, takové že $|\omega_{j+1l}| > \eta$, kde $l = r_i, r_i + 1, \ldots, k, dots, s_i - 1, s_i$;
    end
  end
end
for *každé i, kde* r_i *a* s_i *nejsou* 0 **do**
  for $l = r_i, r_i + 1, \ldots, s_i - 1, s_i$ **do**
    $Q[j+1] = Q[j+1] - (Q[j+1]^T Q[l])Q[l]$;
  end
end
if $first\_step$ **then**
  $first\_step = false$;
end
else
  $first\_step = true$;
  $r_i = 0$;
  $s_i = 0$;
end

- *MPI* Výpočtu se podílejí všechny diagonální procesy. Avšak pouze *Root* proces počítá rekurenci ω. Pomocí *MPI_Bcast* pak následně přepošle ostatním diagonálním procesům vypočítané intervaly vektorů, vůči kterým se bude nový vektor ortogonalizovat.

- *Výpočet ω u blokové verze* Na výpočet ω se potřebují singulární hodnoty matic *alpha* a *beta*, které se vypočítají pomocí funkce knihovny LAPACK *?gesvd*.

5.4.5 Iterační reortogonalizace

Iterační reortogonalizace je naimplementována ve třídě *fullMultiBlockOrtho* nadtřídy *reorthogonalization*. Pseudokód lze vidět pro jedno-vektorovou verzi 13 a pro blokovou 14.

Rozdílem od původní verze úplné reortogonalizace je využití klasického Gram-Schmidtova procesu.

Nastavení této reortogonalizace je obdobné jako u částečné. Rozdílem je možnost u konstruktoru třídy *fullMultiBlockOrtho* nastavit proměnnou *maximální počet opakování*, která zadá maximální počet iterací ortogonalizace.

Algoritmus 13: Lanczosův algoritmus - Úplná reortogonalizace podle KGS (1)

Vstup: sada Lanczosových vektorů Q^{j-1}
Vstup: orthogonalizovany vektor x_j
Výstup: zorthogonalizovany vektor v_j
Výstup: α a β pro tridiagonalni matice

$crit = 1/\sqrt{2}$;
$dot_x = ||x_j||$;
$h_j = 0$;
do
  $[v_j, r_j]$ = Klasický Gram-Schmidt(x_j, Q^{j-1});
  $h_j = h_j + r_j$;
  $\beta = ||v_j||$;
  iterace++;
while $dot_x * crit > beta$ *a iterace* $<$ *maximální počet opakování*;
α = poslední prvek vektoru h_j;

Algoritmus 14: Blokový Lanczosův algoritmus - Úplná reortogonalizace podle KGS (1)

Vstup: sada Lanczosových vektorů Q^{j-1}
Vstup: orthogonalizovany blok X_j
Výstup: zorthogonalizovany blok V_j
Výstup: A a B pro tridiagonalni matice

$crit = 1/\sqrt{2}$;
$V_j = X_j$;
$H_j = 0$;
do
  $normy$ = vektor norem vektorů V_j;
  $R_j = Q^{j-1T} V_j$;
  $V_j = V_j - Q^{j-1} R_j$;
  $H_j = H_j + R_j$;
  for $i = 1, ...,$ *velikost bloku* **do**
    $div[i] = ||V_j[i]||/normy[i]$;
  end
  $beta\_div$ = min$(div[i])$;
  iterace++;
while $crit > beta\_div$ *a iterace* $<$ *maximální počet opakování*;
$[V_j, B]$ = QR rozklad(V_j);
A = poslední blok matice H_j;

- *MPI* Na výpočtu se podílejí všechny diagonální procesy, stejně jako u původní verze úplné reortogonalizace.

5.4.6 Paralelizace pomocí OpenMP

Většina výpočetně náročných operací v implementaci jsou maticové či vektorové výpočty umístěné ve třídách *block_diagonal_vector* a *diagonal_vector*.

V těchto případech jsou využité direktivy *parallel* a *for*. Je využito statické přidělení iterací, jelikož v těchto maticových operacích není zapotřebí dynamické režie.

Jednou často využívanou změnou oproti původní implementaci je zavedení metody *local_dot_product* ve třídě *diagonal_vector*. Původní metoda *dot_product* obsahuje MPI funkci *MPI_Allreduce*, která znemožňuje umístění paralelní region nad cyklus, ze kterého se metoda *dot_product* volá. Metoda *local_dot_product* umožňuje paralelní region použít a MPI funkci *MPI_Allreduce* dát až vně paralelního regionu.

5.4.7 Specifický úvodní vektor

Na začátku výpočtu metody *solve* se vygeneruje náhodný vektor, ze kterého výpočet začíná. Metoda *setInitialVector* umožní vložit specifický vektor na místo náhodného.

Metoda má dva parametry. První je vektor s názvy souborů obsahující vektory ve stejném

formátu jako je ukládá metoda *store_eigenvectors*. Druhý parametr dává velikost vektoru v souboru.

Při zadání více vektorů u jedno-vektorové verze, se jednotlivé hodnoty na stejných pozicích sčítají.

U blokové verze se metoda jmenuje *setInitialBlock*.

Důvod pro tuto funkcionalitu a proč se tato funkcionalita netestuje je vysvětlen v sekci 5.5.5 o LOBPCG metodě.

5.5 LOBPCG metoda

Kapitola je rozdělena do několika sekcí. V sekcích 5.5.1 a 5.5.2 jsou popsány implementace jedno-vektorové a blokové verze LOBPCG.

Následuje sekce 5.5.3 popisující vyřazení již zkonvergovaných vlastních čísel z aktivního výpočtu (locking).

V poslední sekci 5.4.7 je popsána možnost zadání konkrétních úvodních vektorů.

5.5.1 Jedno-vektorová verze

Jedno-vektorová verze LOBPCG metody je naimplementovaná ve třídě *LOPCG_eigensolver*. Viz algoritmus 15.

Tato implementace má jedinou nastavitelnou vlastnost. Jedná se o zadání specifického úvodního vektoru (viz 5.5.5).

Hlavní implementace algoritmu je v metodě *Solve*. Avšak vnitřek vnějšího *for* cyklu pro nediagonální procesy je naimplementován v metodě *solve_not_diag*.

Algoritmus 15: LOBPCG algoritmus - jedno-vektorový

Vstup: Symetrická matice $A^{n,n}$
Vstup: počáteční vektor x_1 s normou 1
Vstup: počet hledaných vlastních čísel n_λ
Vstup: maximální počet iterací q ($q \geq n_\lambda$)
Vstup: tolerance konvergence ϵ
Výstup: aproximace (μ a x) n_λ nejmenších vlastních čísel a vektorů matice A

```
1  for i = 1, ..., n_λ do
2      x = náhodný nenulový znormovaný vektor;
3      p = 0;
4      w = A * x;
5      xtheta = x * (x^T w);
6      r = w − xtheta;
7      for j = 1, ..., q do
8          Zortogonalizuj vektory x, r, p vzhledem k sobě a k již nalezeným vlastním vektorům;
9          Teig = [w Ar Ap]^T [w Ar Ap];
10         Vypočítej nejmenší vlastní číslo λ a jeho vlastní vektor y;
11         r = r * y[1];
12         r = r + y[2]p;
13         p = r;
14         x = x * y[0];
15         x = x + p;
16         w = A * x;
17         xtheta = x * λ;
18         r = w − xtheta;
19         if ||r|| < ε then
20             break;
21         end
22     end
23 end
```

- *Násobení vstupní matice A vektorem zprava (řádek 4, 9, 16)* Tři násobení vstupní matice vektorem, které se účastnějí všechny procesy (MPI komunikace mezi procesy je stejná jako u původní jedno-vektorové verze Lanczosova algoritmu).

- *Řešič vlastních čísel (řádek 10)* Na výpočet (pouze proces *Root*) se používá funkce knihovny LAPACK *?syevx* na symetrické matice, jelikož matice *Teig* je pouze symetrická.
- *MPI* Po výpočtu vlastního čísla a vektoru je proces *Root* rozešle ostatním diagonálním procesů pomocí MPI funkce *MPI_Bcast*.

5.5.2 Bloková verze

Bloková verze LOBPCG metody je naimplementovaná ve třídě *LOBPCG_eigensolver*. Viz algoritmus 16.

Hlavní implementace algoritmu je v metodě *Solve*. Avšak vnitřek vnějšího *for* cyklu pro nediagonální procesy je naimplementován v metodě *solve_not_diag*.

Locking (vyřazení) vlastních čísel z aktivního výpočtu je popsáno v sekci 5.5.3.

Stejně jako u jedno-vektorové verze jdou zadat specifické úvodní vektory (viz 5.5.5).

Algoritmus 16: LOBPCG algoritmus - blokový

Vstup: Symetrická matice $A^{n,n}$
Vstup: počet hledaných vlastních čísel n_λ
Vstup: maximální počet iterací q (q)
Vstup: tolerance konvergence ϵ
Vstup: velikost bloku s
Výstup: aproximace (μ a x) n_λ nejmenších vlastních čísel a vektorů matice A

```
1  X^{n,s} = náhodný nenulový ortonormální matice;
2  P^{n,s} = 0;
3  W = A * X;
4  XTheta = X * (X^T W);
5  R = W − XTheta;
6  for j = 1, ..., q do
7      Zortogonalizuj matice X, R, P vzhledem k sobě a k již nalezeným vlastním vektorům;
8      Teig = A * [R P];
9      Teig = [W Teig];
10     Teig = [X R P][Teig];
11     Vypočítej počet hledaných nejmenších vlastních čísel λ a jejich vlastních vektorů Y matice Teig;
12     P = [R P]* poslední řádky matice vlastních vektorů Y;
13     X = X* první řádky matice vlastních vektorů Y;
14     X = X + P;
15     for k = 1, ..., n_λ− počet již nalezených vlastních čísel do
16         XTheta[k] = X[i] * λ_k;
17     end
18     W = A * X;
19     R = W − XTheta;
20     Najdi již zkonvergované (R[k = 1, ..., n_λ] < ε) vlastní čísla (λ[k = 1, ..., n_λ]) a ulož si je společně s
         příslušnými vlastními vektory (sloupce matice X);
21     if jsou nalezená všechna hledaná vlastní čísla then
22         break;
23     end
24     Odstraň příslušné sloupce z matic X, R a P;
25 end
```

- *Násobení vstupní matice A vektorem zprava (řádek 3, 8, 18)* Dvě násobení vstupní matice blokem vektorů, které se účastnějí všechny procesy (MPI komunikace mezi procesy je stejná jako u původní blokové verze Lanczosova algoritmu).
- *Řešič vlastních čísel (řádek 11)* Na výpočet (pouze proces *Root*) se používá funkce knihovny LAPACK *?syevx* na symetrické matice, jelikož matice *Teig* je pouze symetrická.
- *MPI* Po výpočtu vlastních čísel a vektorů je proces *Root* rozešle ostatním diagonálním procesů pomocí MPI funkce *MPI_Bcast*.

5.5.3 Locking

Jelikož jedno-vektorový LOBPCG počítá vlastní čísla postupně, nemá význam locking u této varianty uvažovat.

U blokové varianty je locking uskutečněn automaticky, jelikož na rozdíl od Lanczosova algoritmu nevyžaduje v LOBPCG metodě žádné složité výpočty. Je naimplementován v metodě *locking*.

5.5.4 Paralelizace pomocí OpenMP

Paralelizace v LOBPCG metodě pomocí OpenMP u obou verzí je prováděna ve třídách *block_diagonal_vector* a *diagonal_vector*. Popis paralelizace v těchto třídách je ve stejnojmenné kapitole u Lanczosova algoritmu (5.4.6).

5.5.5 Specifický úvodní vektor

Funkcionalita funguje stejně jako u Lanczosova algoritmu. Je naimplementována v metodě *setInitialVectors* u jedno-vektorového LOBPCG a *setInitialBlock* u blokové verze.

Idea pro tuto funkcionalitu přišla z článku [24] o LOBPCG, kde testují metodu na fyzikálním problému, který je použit v této práci u testování. Jako úvodní vektory (doplněné nulami) byly použity výstupní vlastní vektory z určitého menšího problému. Výsledný naměřený čas byl výrazně menší než u výpočtu s náhodným úvodním vektorem.

Avšak tato funkcionalita nebude v této práci testována, jelikož nepřímočarý způsob mapování vstupní matice na procesy v testování by vyžadoval komplikovanější implementaci tohoto problému.

Kapitola 6
Testování

Kapitola se věnuje testování implementace popsané v kapitole 4 a 5. Uvádí stručně použitý hardware a software, popisuje metody testování a použité metriky. Dále popisuje výsledky testů jednotlivých variant algoritmů, jejich porovnání, celkové porovnání a závěrečné vyhodnocení testování.

6.1 Hardware a software

Testování výkonnosti implementací jednotlivých algoritmů a jejich porovnání bylo prováděno na superpočítači Karolina v Národním superpočítačovém centru VŠB Technické univerzity Ostrava [34]. Má konfiguraci: 720 univerzálních uzlů, každý uzel má 2 procesory AMD Zen 2 EPYC™ 7H12, 2.6 GHz a 256 GB RAM.

Úlohy se dávají do fronty úloh. Je tím zajištěna nezávislost úlohy na jiných procesech. Opakovaná měření dávají výsledky s s minimálními odchylkami.

Pro kompilaci byl použit wrapper *mpic++ g++* kompilátoru verze 11.3.0. Jako přepínače byly použity *-O3 -std=c++14 -DNDEBUG -DHAVE_INLINE -ffast-math -funroll-loops -fopenmp -llapack*. Pro spuštění byl použit příkaz *mpirun* s přepínačem *–map-by numa*.

6.2 Způsob testování

Vstupní data (rozsáhlé symetrické řídké matice) se generují z modelových prostorů 10B s Nmax 6, 12C s Nmax 6 a 10B s Nmax 8. Na toto generování je použit software *lsu3shell* přístupný na gitlabu (`https://gitlab.com/tdytrych/lsu3shell`).

Použité vstupní matice jsou:

- *10B s Nmax 6* - testování s 15 procesy, matice velikosti 502751 řádků
- *12C s Nmax 6* - testování s 66 procesy, matice velikosti 1260008 řádků
- *10B s Nmax 8* - testování s 435 procesy, matice velikosti 5660372 řádků

Pokus o použití testování s 630 procesy na matici vygenerovanou modelovým prostorem 12C s Nmax 8 skončil chybou hlásící nedostatek paměti.

Výsledná měření jsou vypočítána jako průměr deseti dílčích měření u výpočtů s 15 procesy. U výpočtů s více procesy bylo provedeno vždy pouze jedno dílčí měření. Tyto výpočty byly příliš časově náročné (nad možnosti přiděleného výpočetního času). Navíc dílčí měření vykazovaly minimální odchylku.

Efektivita řešení byla porovnávána z hlediska času a počtu iterací algoritmu.
Testování lze rozdělit do třech částí:

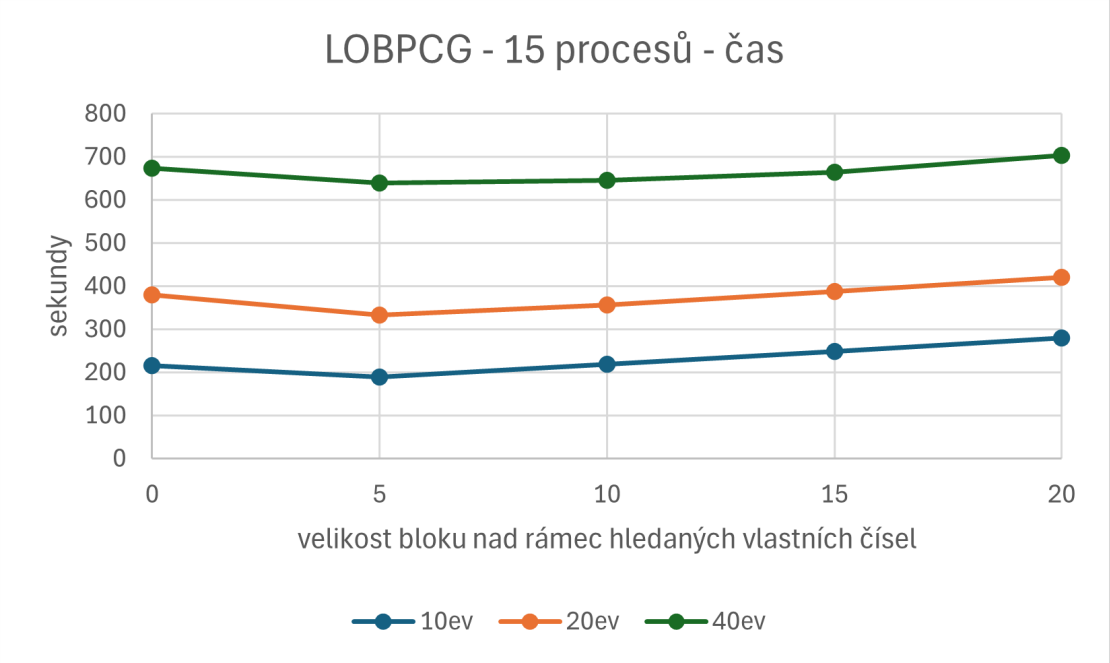
- testování jednotlivých algoritmů a jejich variant
- porovnání variant daného algoritmu
- porovnání různých algoritmů

6.3 Testování algoritmu LOBPCG

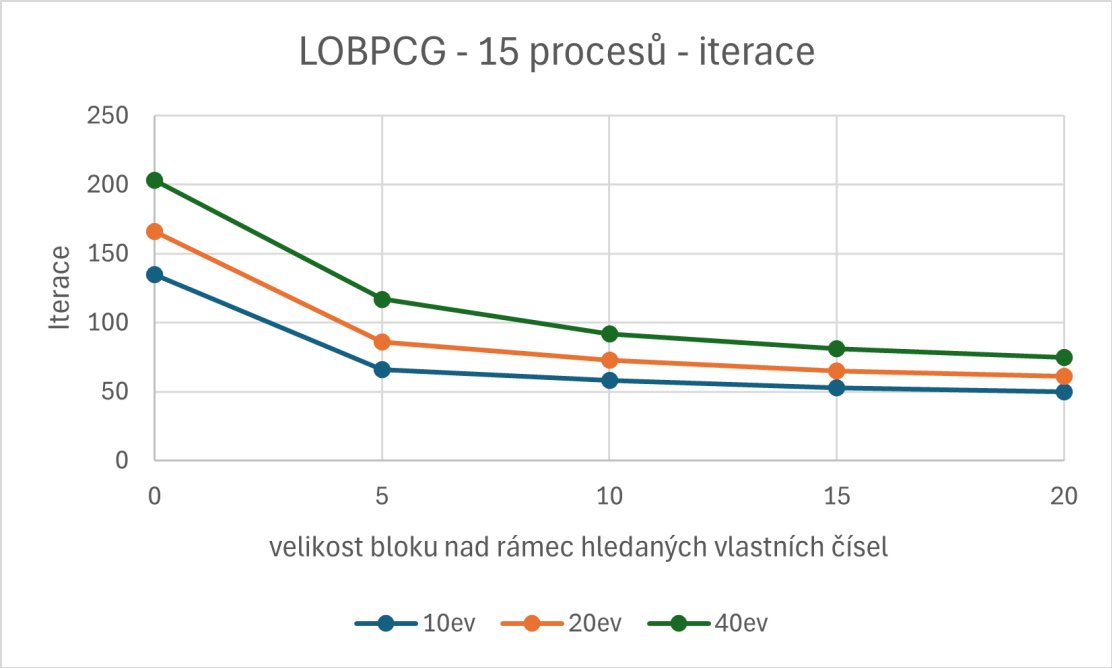
Tento test demonstruje, jak je algoritmus LOBPCG (viz 2.2.2) efektivní v závislosti na velikosti bloku a na počtu hledaných vlastních čísel.

Výsledek testu je vidět v následujících šesti grafech:

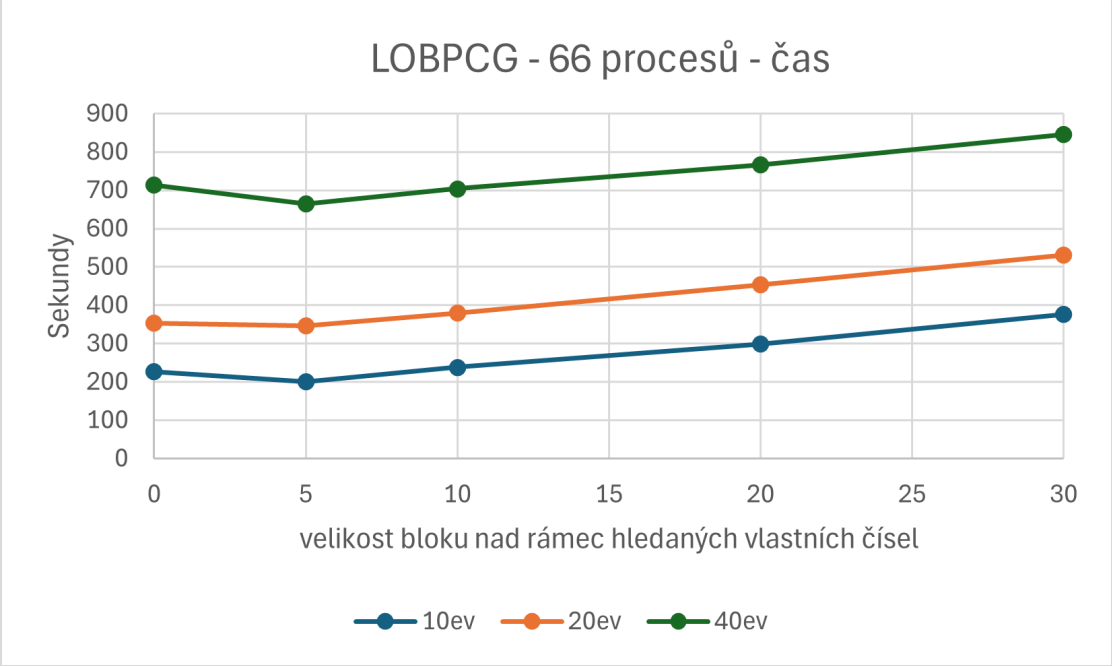
- 15 procesů, testován čas
- 15 procesů, testován počet iterací
- 66 procesů, testován čas
- 66 procesů, testován počet iterací
- 435 procesů, testován čas
- 435 procesů, testován počet iterací



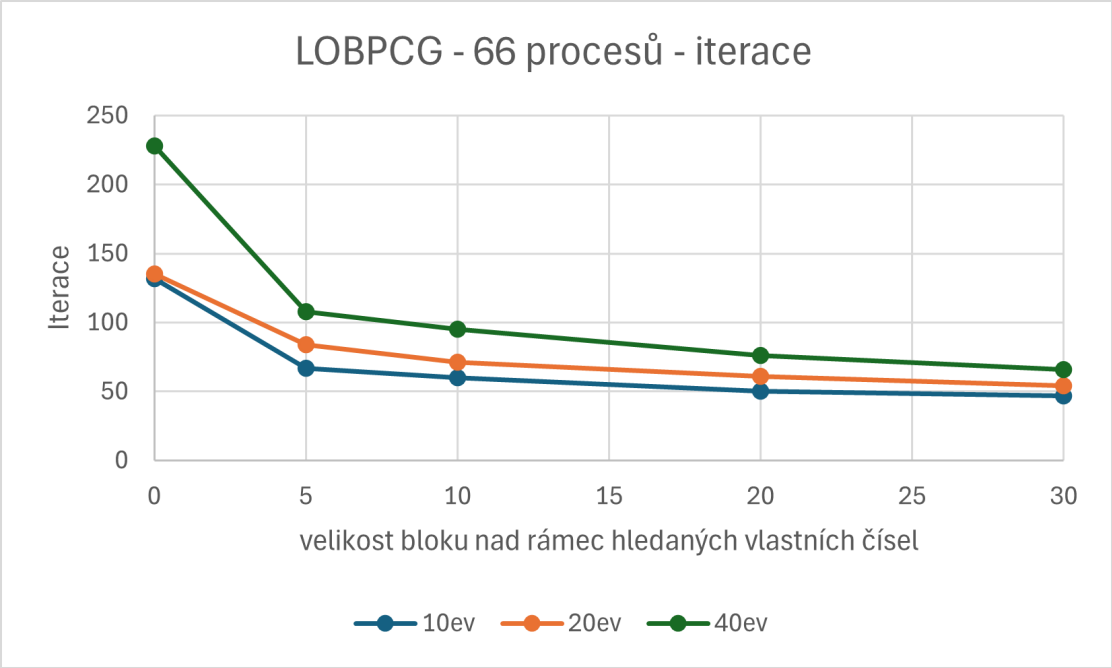
Obrázek 6.1 LOBPCG - 15 procesů - čas



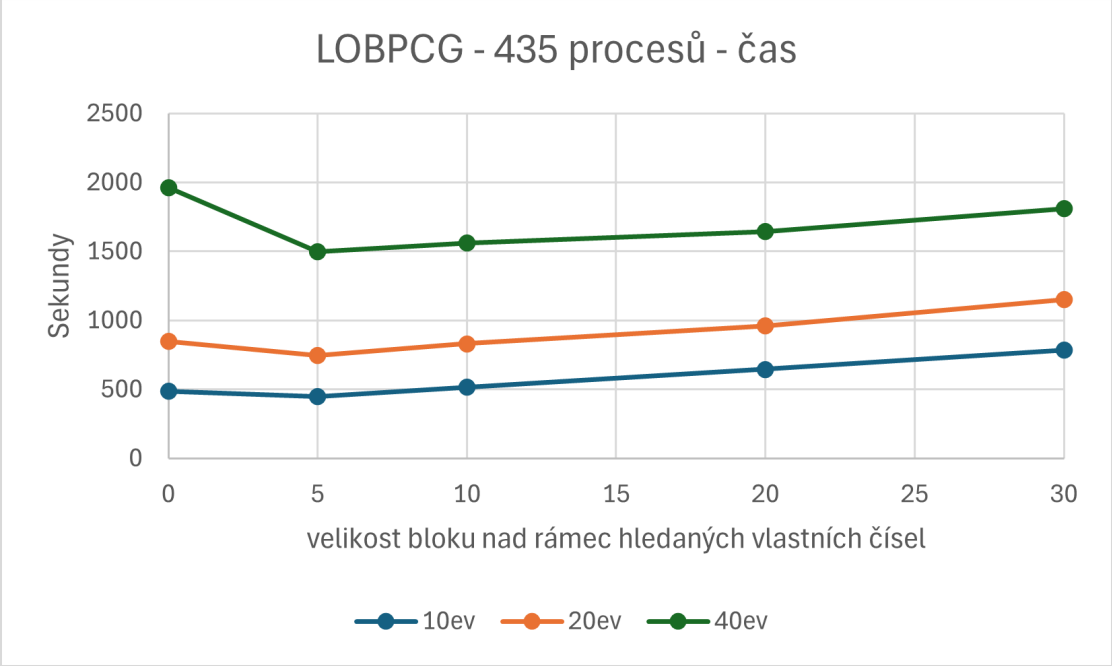
Obrázek 6.2 LOBPCG - 15 procesů - iterace



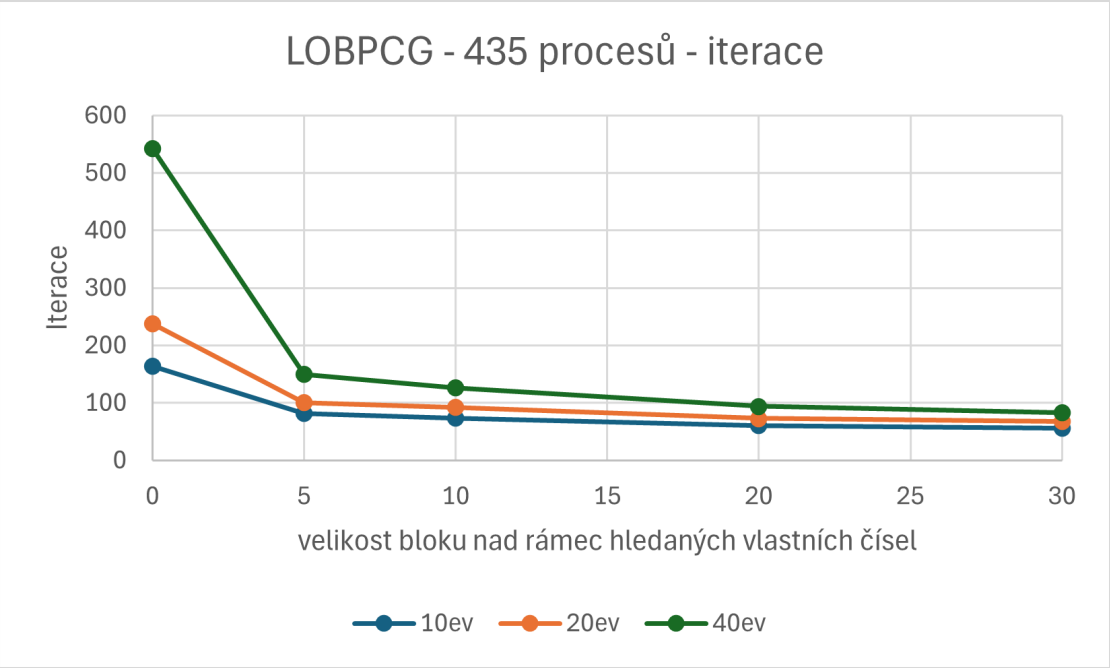
Obrázek 6.3 LOBPCG - 66 procesů - čas



Obrázek 6.4 LOBPCG - 66 procesů - iterace



Obrázek 6.5 LOBPCG - 435 procesů - čas



Obrázek 6.6 LOBPCG - 435 procesů - iterace

Grafické znázornění výsledku testu ukazuje:

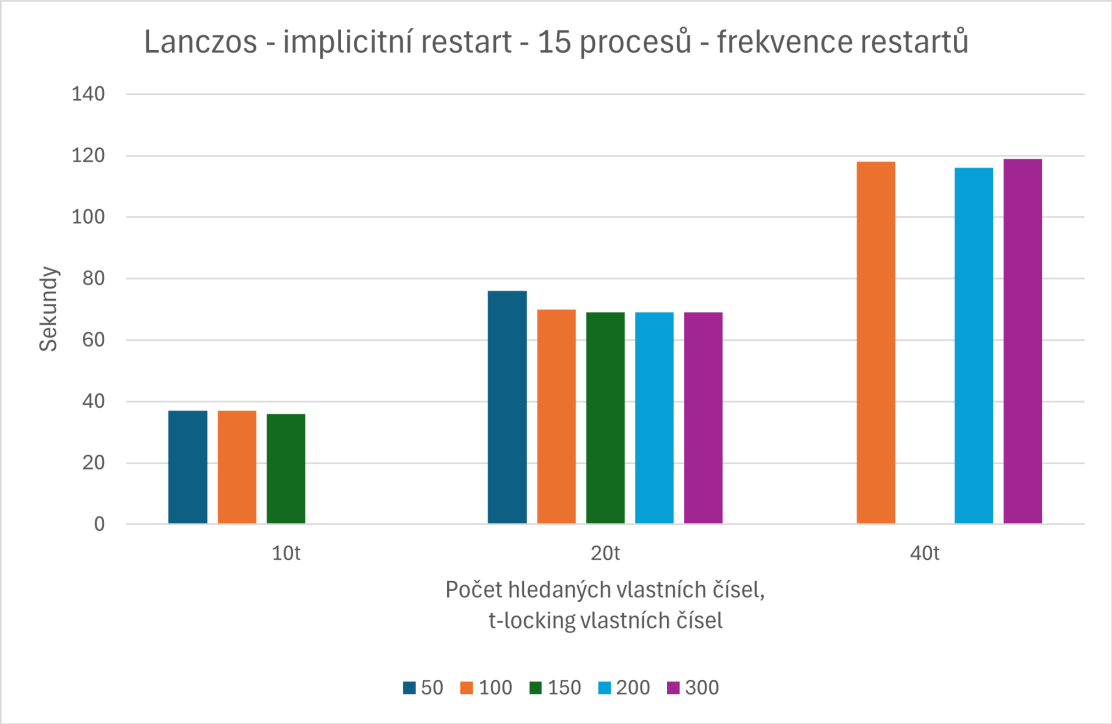
- výsledný tvar grafu nezávisí na počtu procesů
- časově je algoritmus nejrychlejší v případě, kdy velikost bloku je o 5 vyšší než počet hledaných vlastních čísel
- při dalším zvyšování velikosti bloku (o 10, 15, 20, 30) se algoritmus postupně zpomaluje
- počet iterací algoritmu klesá s velikostí bloku
- s přibývající velikostí bloku se klesání iterací algoritmu zpomaluje

6.4 Testování implicitního restartu Lanczosova algoritmu

Tento test demonstruje, jak je Lanczosův algoritmus s implicitním restartem(viz 2.1.3) efektivní v závislosti na počtu hledaných vlastních čísel a na frekvenci restartu. Testování bylo prováděno s verzí používající locking (5.4.3) vlastních čísel i bez něho.

Výsledek testu je vidět v následujícím grafu:

- 15 procesů, testován čas



Obrázek 6.7 Lanczos - implicitní restart - 15 procesů - čas

Tento graf zobrazuje výsledky testu pro variantu s 15 procesy a verzí s použitím locking vlastních čísel. Frekvence restartu výrazně neovlivňuje výsledný čas.

Další varianty tohoto testu (hledání 100 vlastních čísel, nepoužití locking, 66 a 435 procesů) nedávaly přijatelné výsledky. Výsledná vlastní čísla nekonvergují ke správným výsledkům.

Důvody tohoto chování jsou:

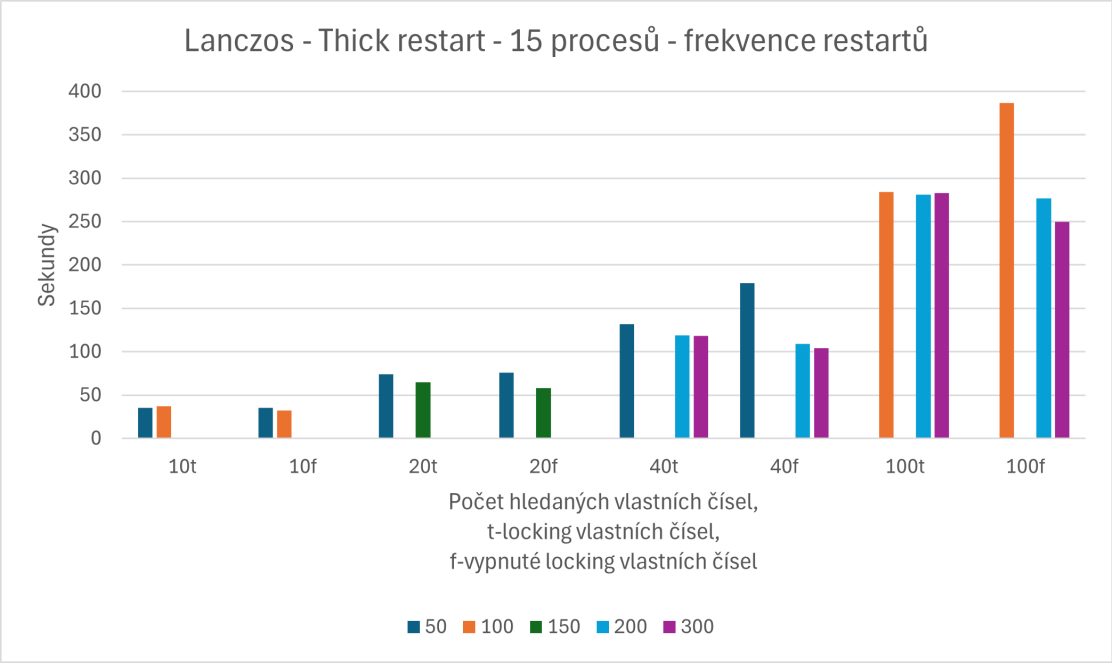
- nestabilita této varianty algoritmu roste s počtem restartů
- pokuď při restartu jsou některá vlastní čísla překonvergovaná, pak restart negativně ovlivňuje konvergenci dalších vlastních čísel.

6.5 Testování thick restartu Lanczosova algoritmu

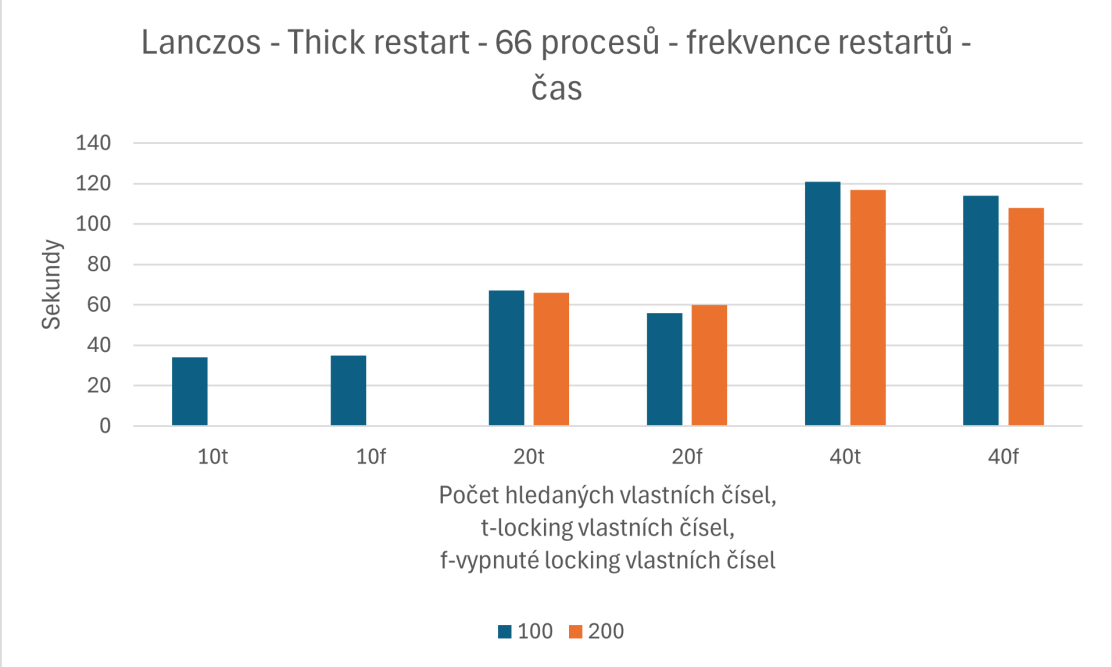
Tento test demonstruje, jak je Lanczosův algoritmus s thick restartem (viz 2.1.3) efektivní v závislosti na počtu hledaných vlastních čísel a na frekvenci restartu. Testování bylo prováděno s verzí používající locking (5.4.3) vlastních čísel i bez něho.

Výsledek testu je vidět v následujících pěti grafech:

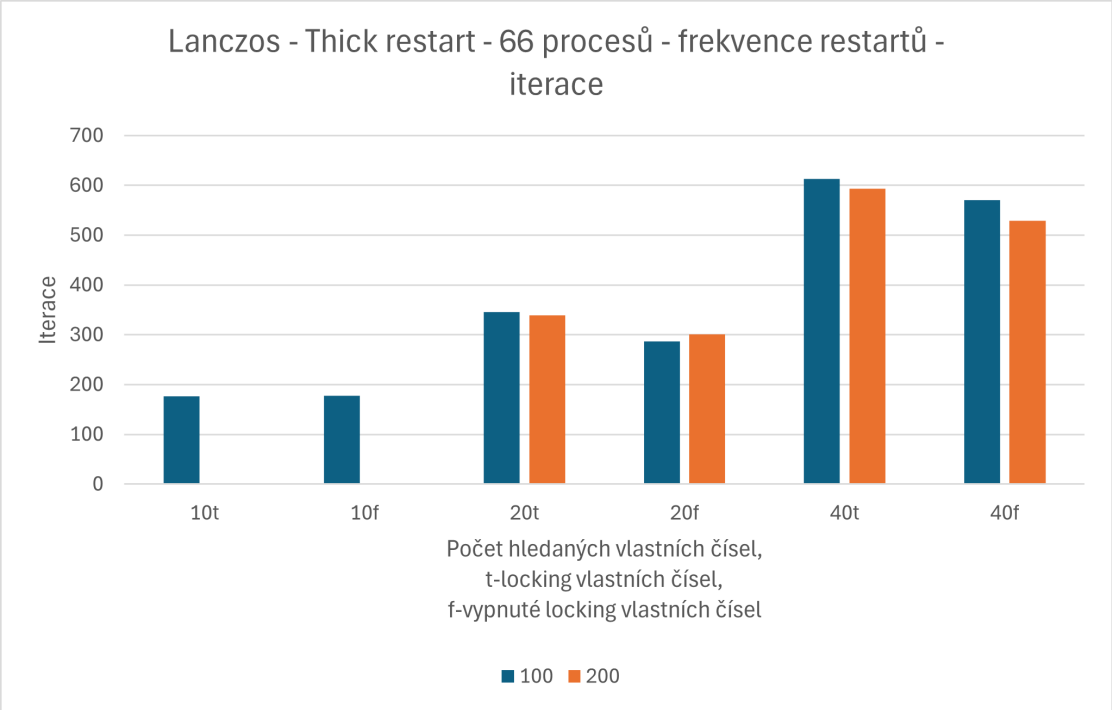
- 15 procesů, testován čas
- 66 procesů, testován čas
- 66 procesů, testován počet iterací
- 435 procesů, testován čas
- 435 procesů, testován počet iterací



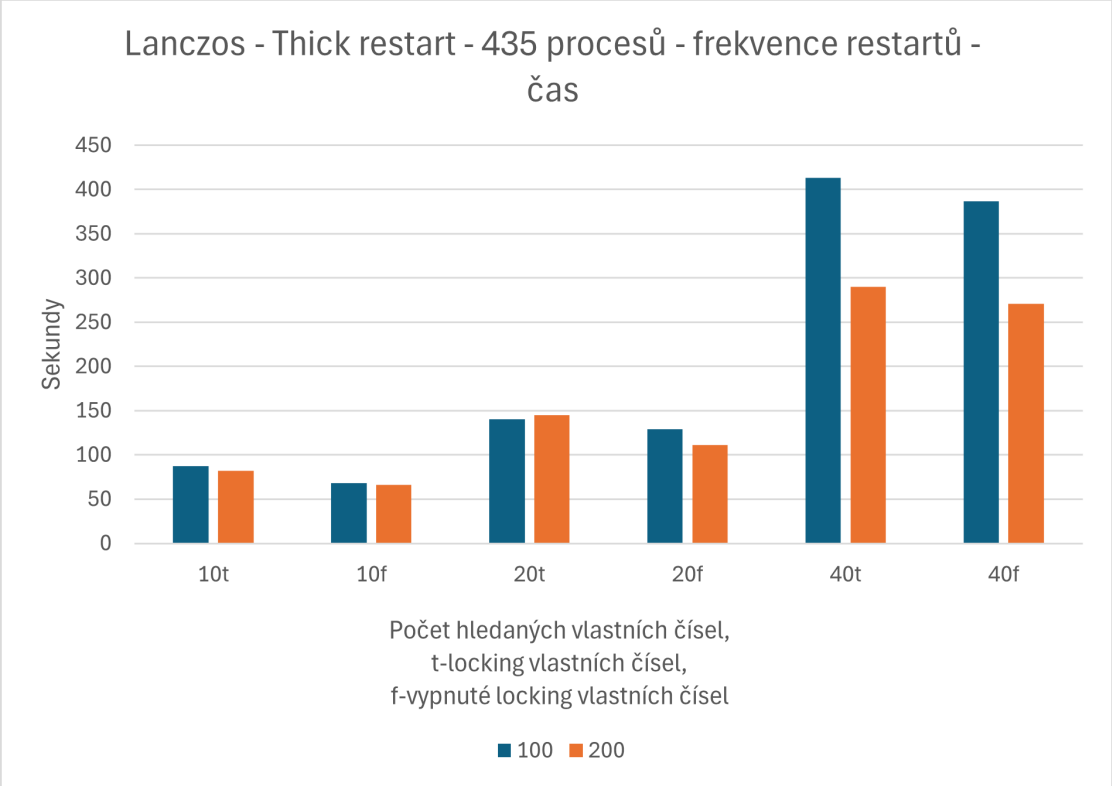
Obrázek 6.8 Lanczos - Thick restart - 15 procesů - čas



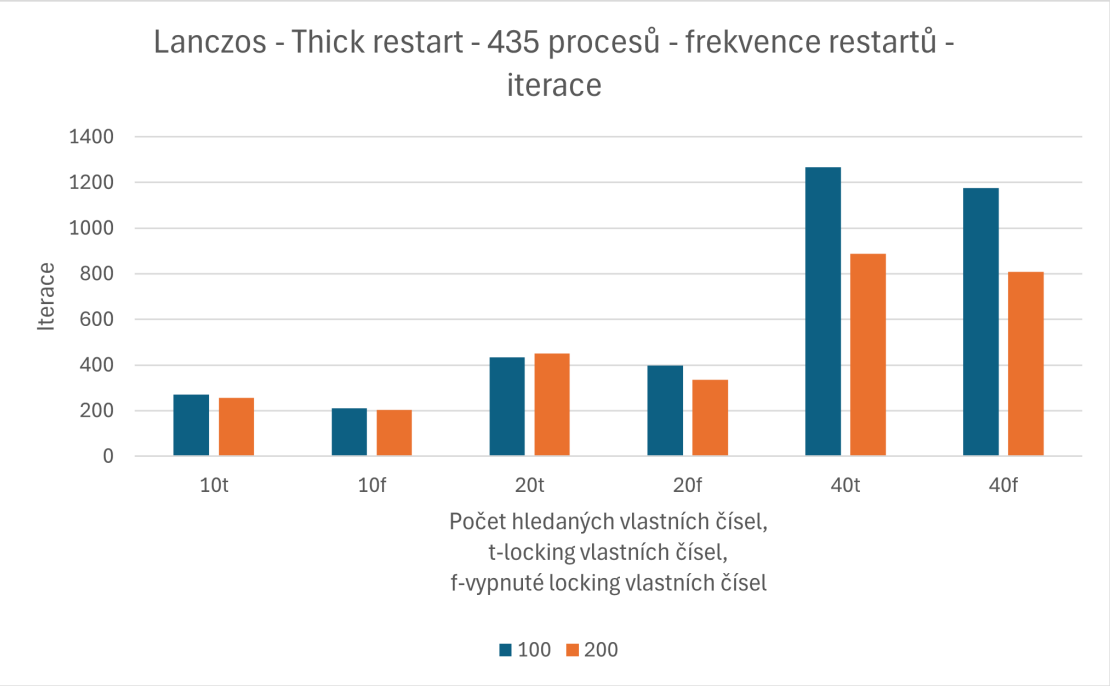
Obrázek 6.9 Lanczos - Thick restart - 66 procesů - čas



Obrázek 6.10 Lanczos - Thick restart - 66 procesů - iterace



Obrázek 6.11 Lanczos - Thick restart - 435 procesů - čas



Obrázek 6.12 Lanczos - Thick restart - 435 procesů - iterace

Grafické znázornění výsledku testu ukazuje:

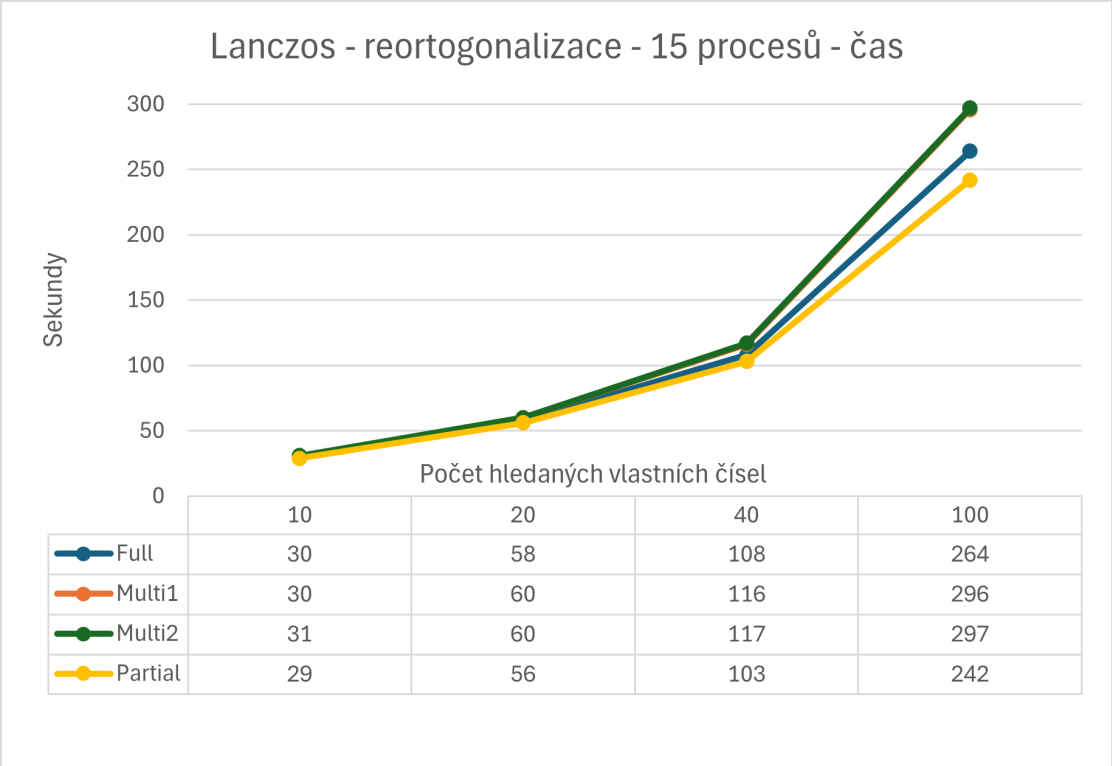
- pro všechny testované varianty dosahuje menší frekvence restartu lepší či stejné efektivity (čas i počet iterací)
- pro všechny testované varianty (až na několik výjimek u 15 procesů) je varianta bez použití locking vlastních čísel efektivnější (časově i počtem iterací)

6.6 Testování různé reortogonalizace Lanczosova algoritmu

Tento test demonstruje, jak je Lanczosův algoritmus s různými typy reortogonalizace efektivní v závislosti na počtu hledaných vlastních čísel. Testování bylo prováděno na čtyřech různých typech reortogonalizace: úplná (Full), částečná (Partial)(5.4.4), iterační s maximálním jedním opakováním (Multi1)(5.4.5), iterační s maximálními dvěma opakováními (Multi2).

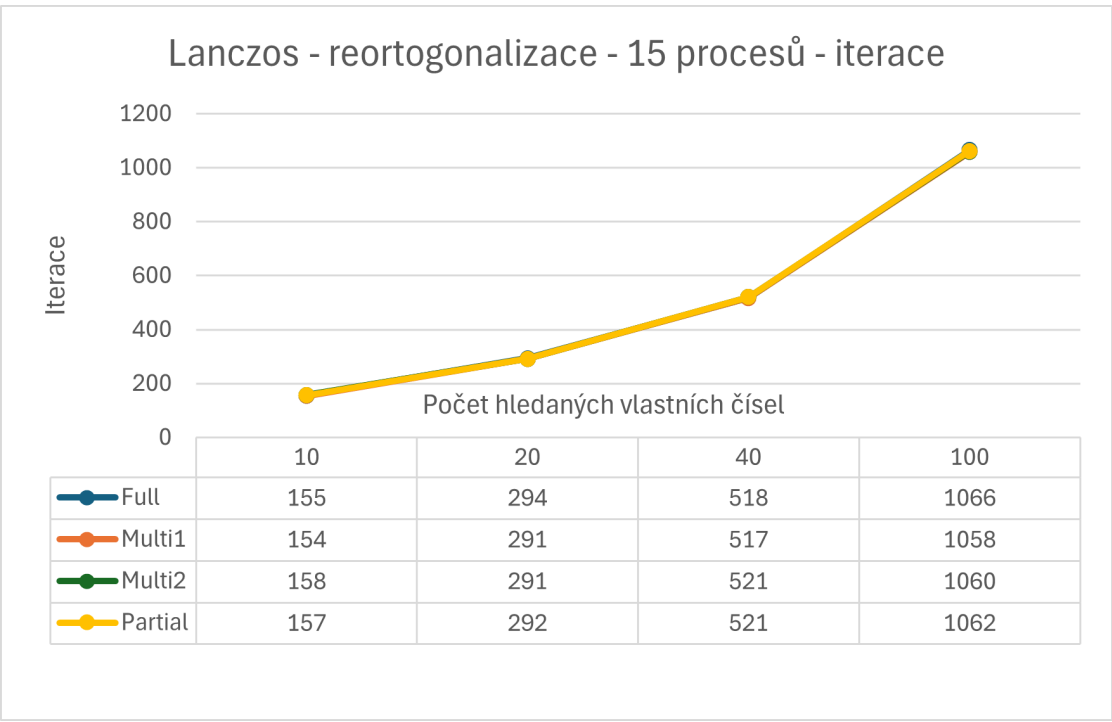
Výsledek testu je vidět v následujících šesti grafech:

- 15 procesů, testován čas
- 15 procesů, testován počet iterací
- 66 procesů, testován čas
- 66 procesů, testován počet iterací
- 435 procesů, testován čas
- 435 procesů, testován počet iterací



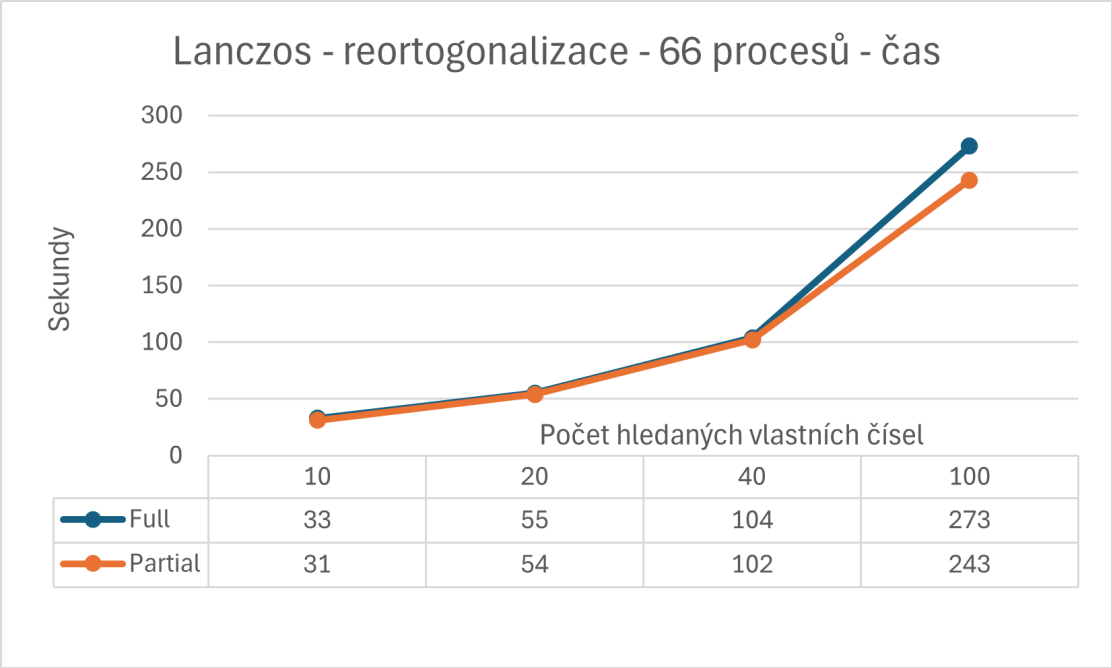
	10	20	40	100
Full	30	58	108	264
Multi1	30	60	116	296
Multi2	31	60	117	297
Partial	29	56	103	242

Obrázek 6.13 Lanczos - Reortogonalizace - 15 procesů - čas

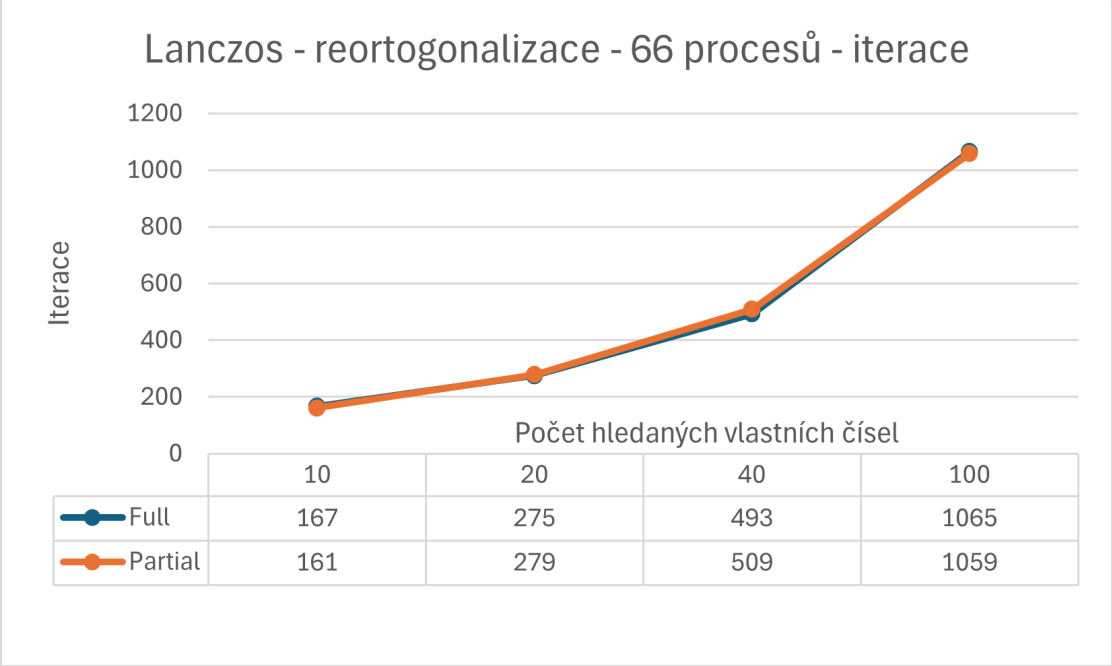


	10	20	40	100
Full	155	294	518	1066
Multi1	154	291	517	1058
Multi2	158	291	521	1060
Partial	157	292	521	1062

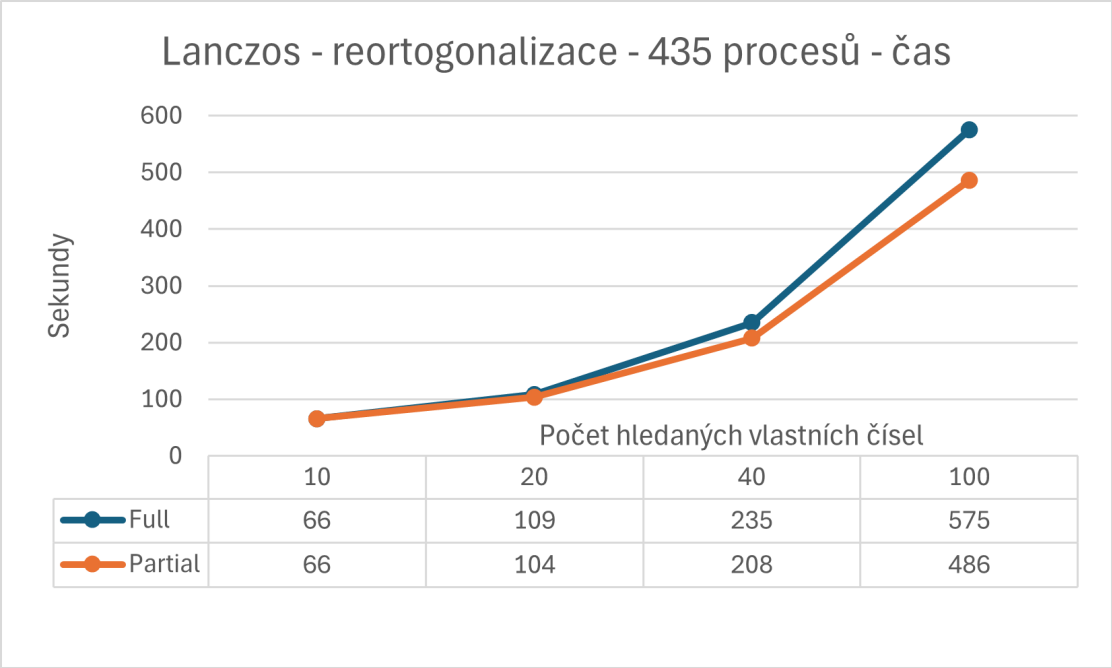
Obrázek 6.14 Lanczos - Reortogonalizace - 15 procesů - iterace



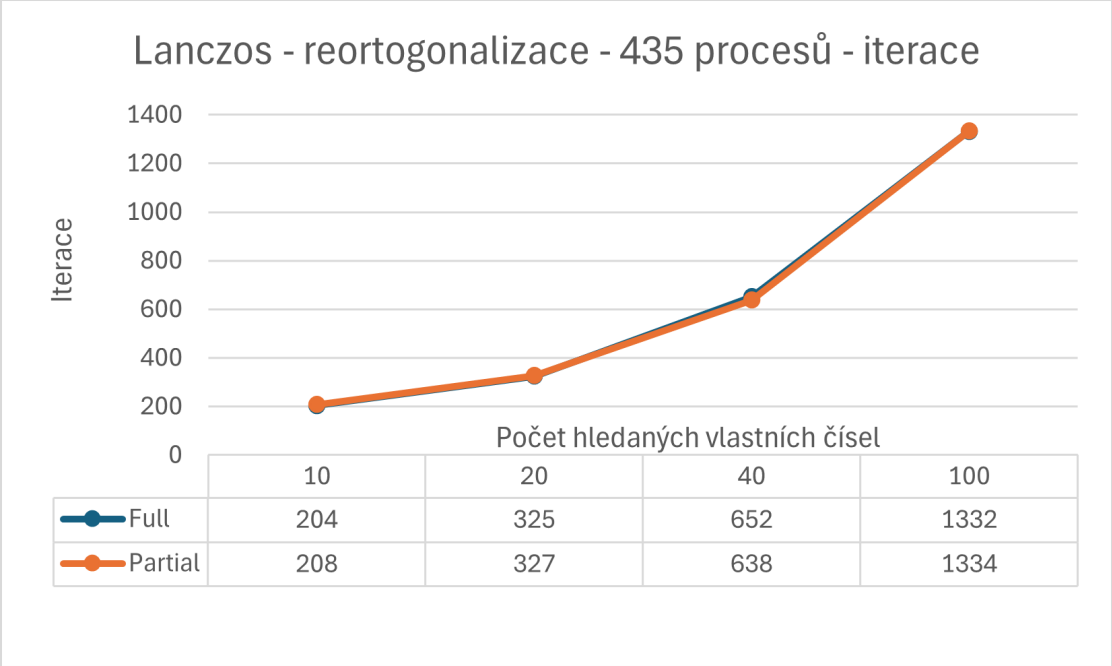
Obrázek 6.15 Lanczos - Reortogonalizace - 66 procesů - čas



Obrázek 6.16 Lanczos - Reortogonalizace - 66 procesů - iterace



Obrázek 6.17 Lanczos - Reortogonalizace - 435 procesů - čas



Obrázek 6.18 Lanczos - Reortogonalizace - 435 procesů - iterace

Grafické znázornění výsledku testu ukazuje:

- výsledný tvar grafu nezávisí na počtu procesů
- typ reortogonalizace neovlivňuje počet iterací algoritmu

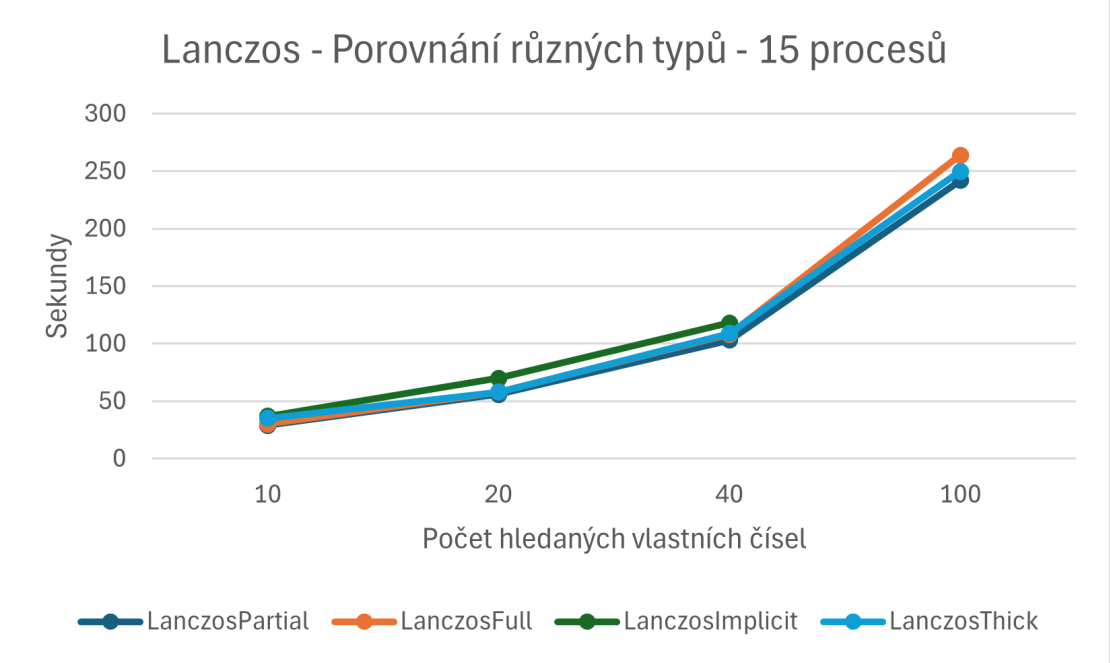
- varianty *Multi1* a *Multi2* ve verzi s 15 procesy jsou pomalejší než ostatní varianty
- jelikož varianty *Multi1* a *Multi2* jsou stejně časově efektivní, nezáleží na nastavení maximálního počtu opakování reortogonalizace
- s přibývajícím počtem hledaných vlastních čísel je částečná reortogonalizace efektivnější než úplná, a to srůstajícím tempem

6.7 Porovnání Lanczosova algoritmu

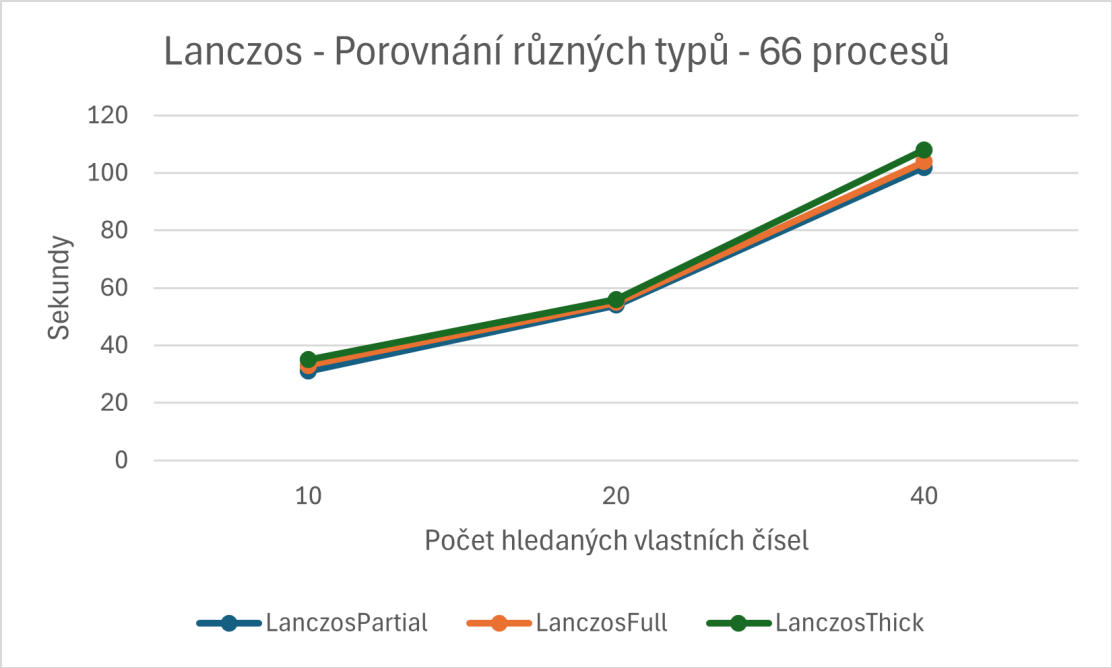
Tento test porovnává efektivitu různých variant Lanczosova algoritmu, které byly použity v předchozích testech.

Tyto varianty jsou:

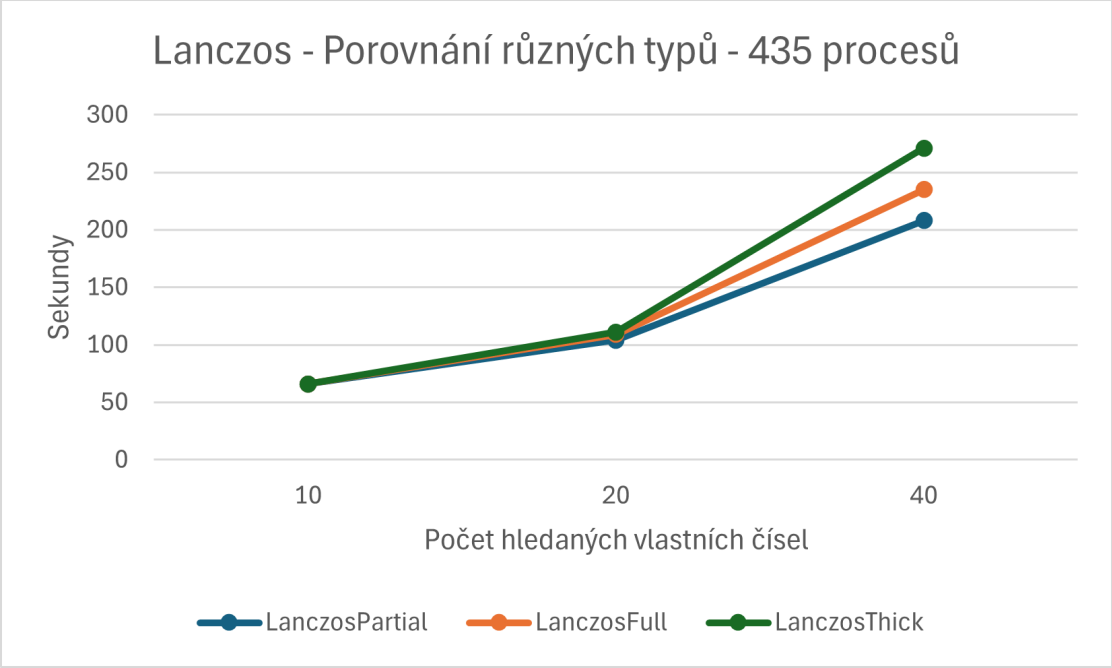
- *LanczosPartial* parciální reortogonalizace
- *LanczosFull* úplná reortogonalizace
- *LanczosImplicit* implicitní restart (časově nejefektivnější varianta z testu 6.4)
- *LanczosThick* thick restart (časově nejefektivnější varianta z testu 6.5)



Obrázek 6.19 Lanczos - porovnání různých typů algoritmu - 15 procesů - čas



Obrázek 6.20 Lanczos - porovnání různých typů algoritmu - 66 procesů - čas



Obrázek 6.21 Lanczos - porovnání různých typů algoritmu - 435 procesů - čas

Grafické znázornění výsledku testu ukazuje:

- časově nejefektivnější testovanou variantou je parciální reortogonalizace
- u varianty s 15 procesy a 100 hledaných vlastních čísel je úplná reortogonalizace pomalejší

než thick restart (důvodem je, že při hledání dostatečně velkého počtu vlastních čísel, počet Lanczosových vektorů naroste natolik, že restart začne být časově výhodnější)

- u varianty s 15 procesy je implicitní restart nejpomalejší

6.8 Testování různé velikosti bloků blokové verze Lanczosova algoritmu

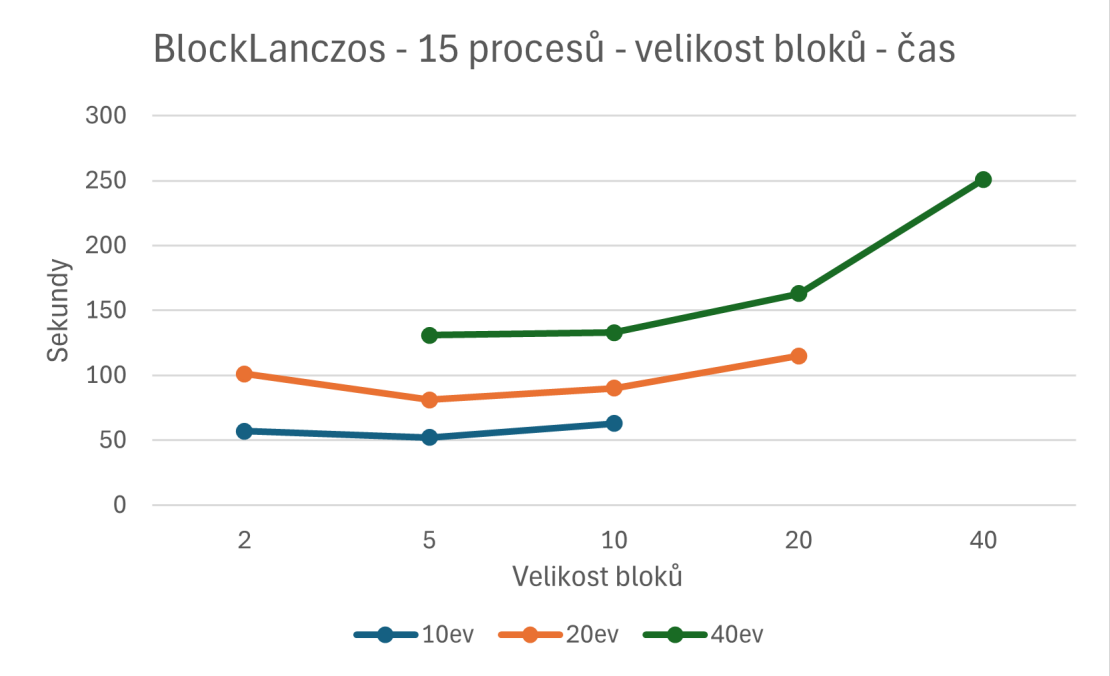
Tento test demonstruje vliv velikosti bloků na efektivitu blokové verze Lanczosova algoritmu.

Testované velikosti bloků jsou 2, 5, 10, 20, 40 vektorů.

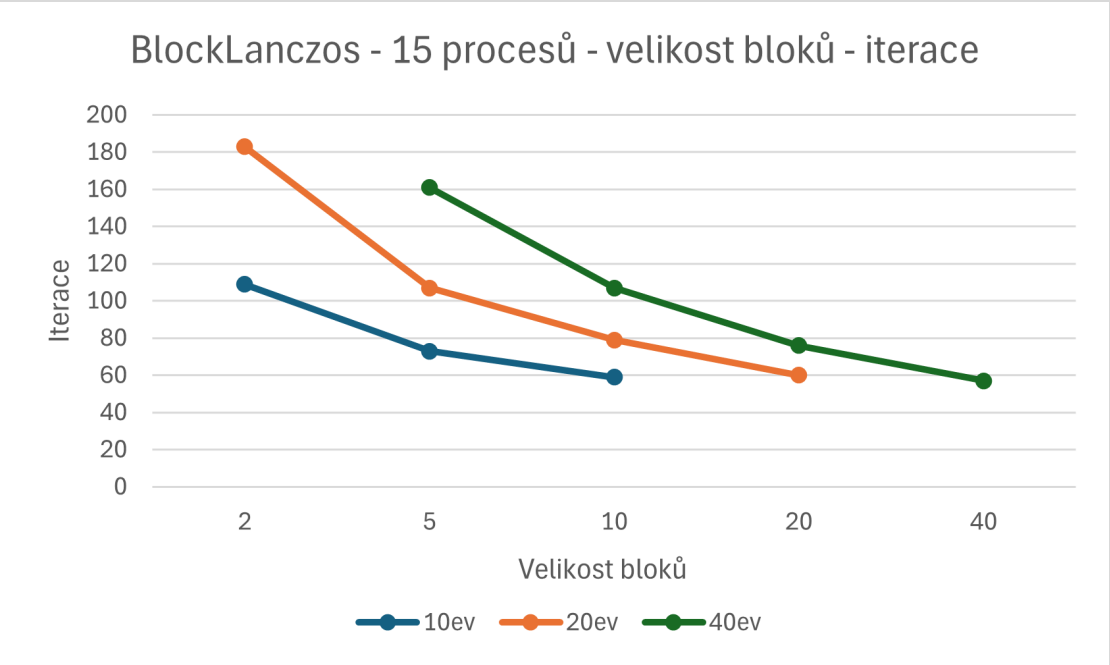
Výsledek testu je vidět v následujících dvou grafech:

- 15 procesů, testován čas

- 15 procesů, testován počet iterací



Obrázek 6.22 Blokový Lanczos - velikost bloků - 15 procesů - čas



Obrázek 6.23 Blokový Lanczos - velikost bloků - 15 procesů - iterace

Grafické znázornění výsledku testu ukazuje:

- s rostoucí velikostí bloků (5, 10, 20, 40 vektorů) se výpočet zpomaluje
- případ s velikostí bloků 2 se zdá být anomálií, jelikož je pomalejší než s velikostí 5
- s rostoucí velikostí bloků se počet iterací snižuje

V dalších testech s blokovou verzí Lanczosova algoritmu se používá blok o 10 vektorech, protože je společně s velikostí 5 vektorů časově nejefektivnější, ale z hlediska počtu iterací je lepší.

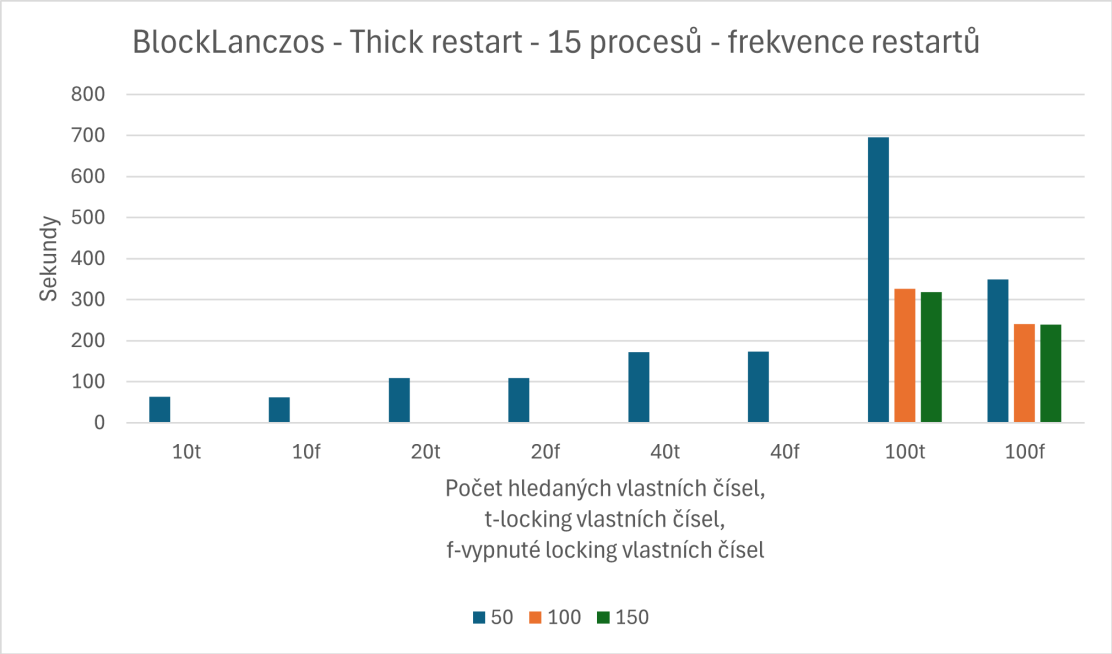
6.9 Testování thick restartu blokové verze Lanczosova algoritmu

Tento test demonstruje, jak je blokový Lanczosův algoritmus s thick restartem (2.1.3) efektivní v závislosti na počtu hledaných vlastních čísel a na frekvenci restartu.

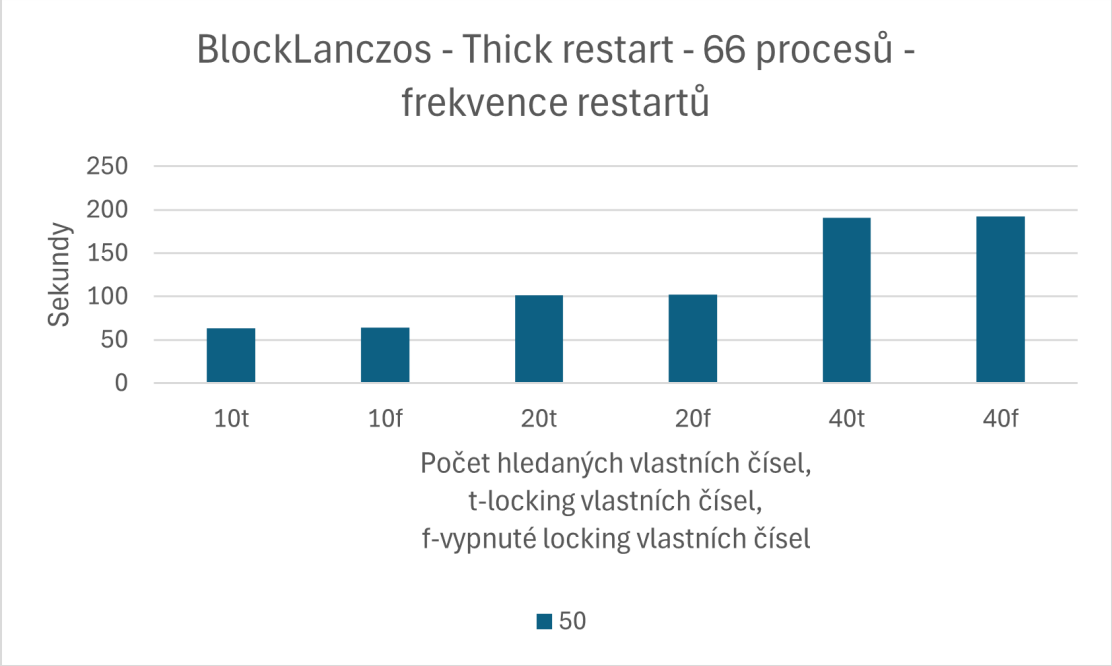
Testování bylo prováděno s blokem velikosti 10 vektorů a s verzí používající locking (5.4.3) vlastních čísel i bez něho.

Výsledek testu je vidět v následujících třech grafech:

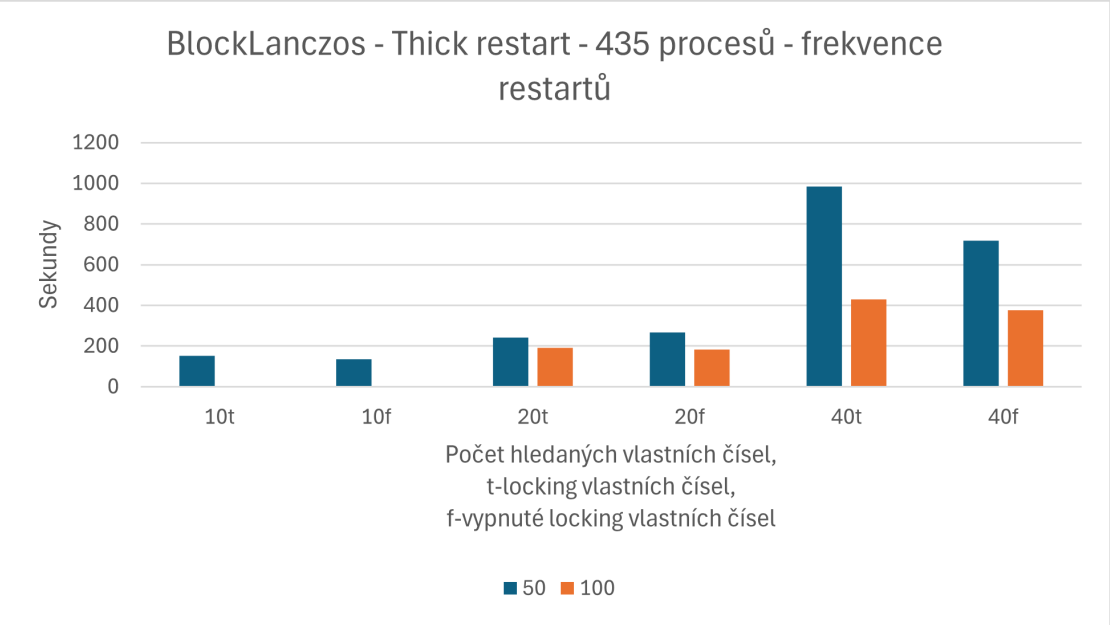
- 15 procesů, testován čas
- 66 procesů, testován čas
- 435 procesů, testován čas



Obrázek 6.24 Blokový Lanczos - Thick restart - 15 procesů - čas



Obrázek 6.25 Blokový Lanczos - Thick restart - 66 procesů - čas



Obrázek 6.26 Blokový Lanczos - Thick restart - 435 procesů - čas

Grafické znázornění výsledku testu ukazuje:

- pro všechny testované varianty dosahuje menší frekvence restartu lepší časovou efektivity
- pro všechny testované varianty je varianta bez použití locking vlastních čísel lepší či stejně efektivní

6.10 Testování různé reortogonalizace blokové verze Lanczosova algoritmu

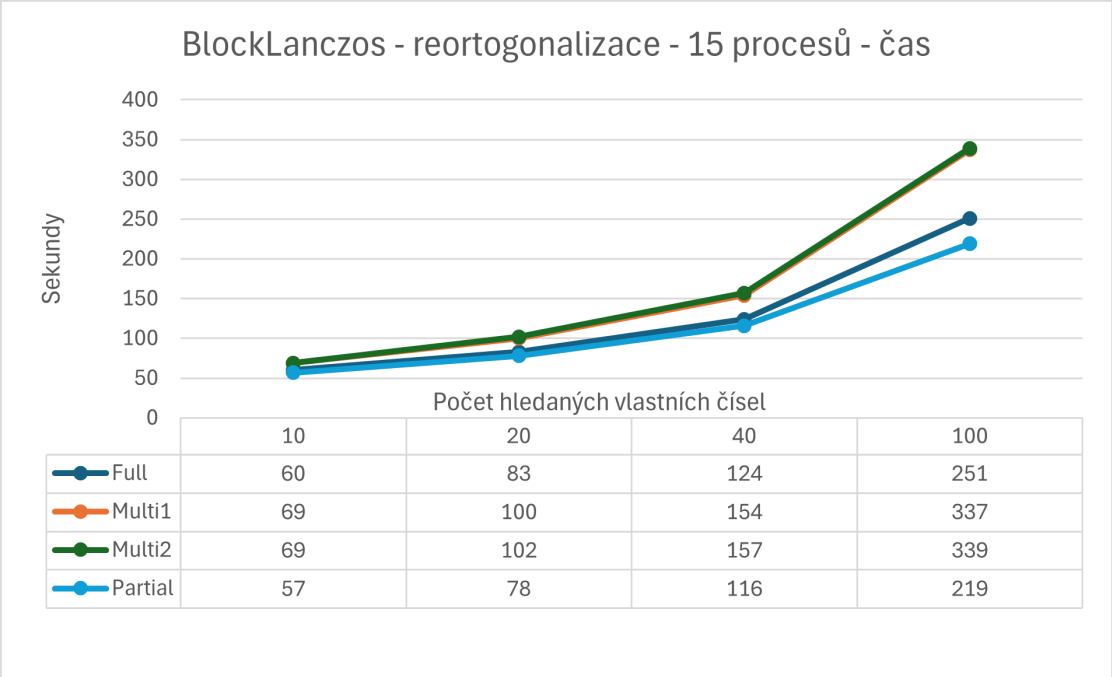
Tento test demonstruje, jak je blokový Lanczosův algoritmus s různými typy reortogonalizace efektivní v závislosti na počtu hledaných vlastních čísel.

Testování bylo prováděna na čtyřech různých typech reortogonalizace: úplná (Full), částečná (Partial)(5.4.4), iterační s maximálním jedním opakováním (Multi1)(5.4.5), iterační s maximálními dvěma opakováními (Multi2).

Varianta s 15 procesy byla testována s blokem velikosti 10 vektorů.

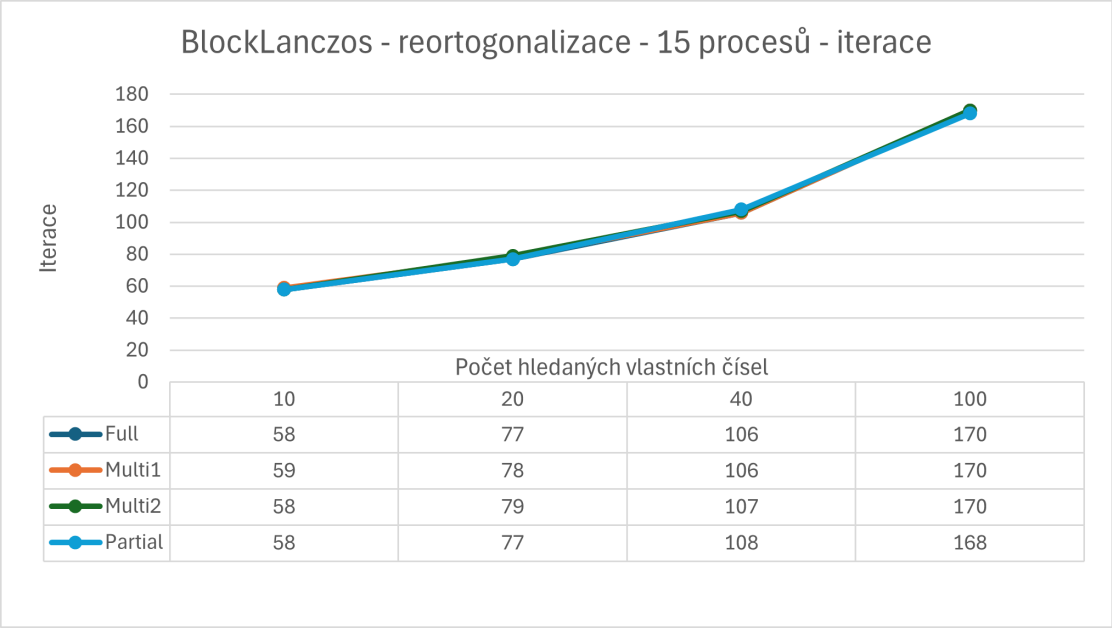
Výsledek testu je vidět v následujících šesti grafech:

- 15 procesů, testován čas
- 15 procesů, testován počet iterací
- 66 procesů, testován čas
- 66 procesů, testován počet iterací
- 435 procesů, testován čas
- 435 procesů, testován počet iterací



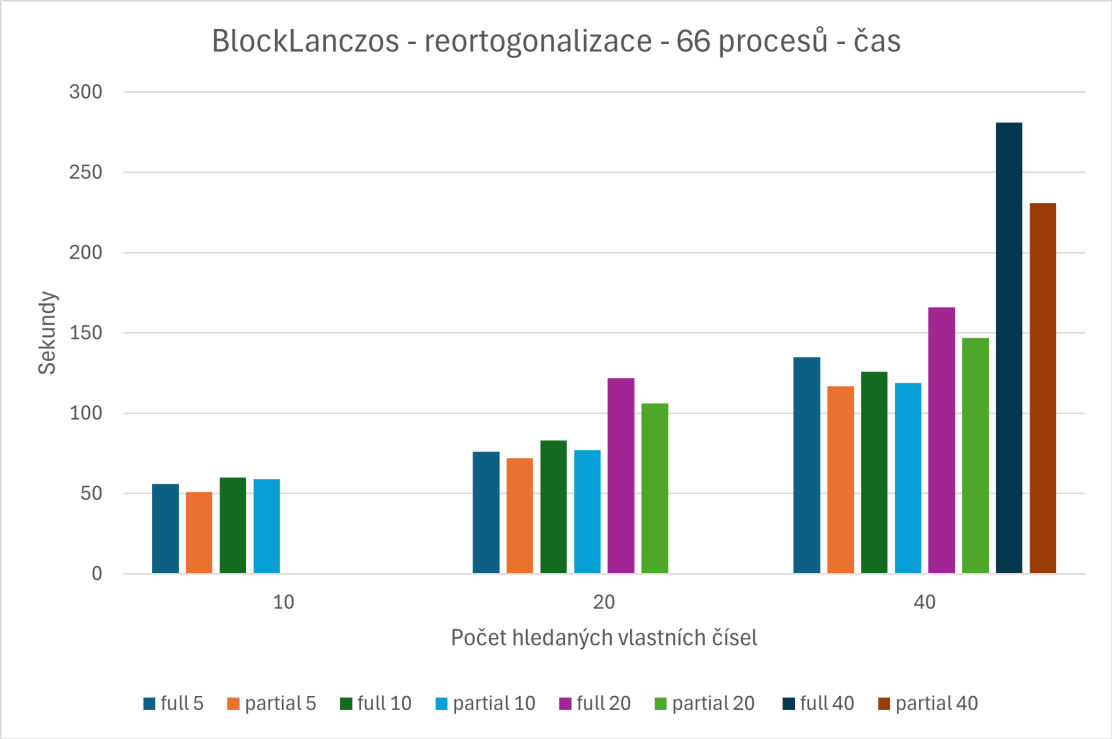
	10	20	40	100
Full	60	83	124	251
Multi1	69	100	154	337
Multi2	69	102	157	339
Partial	57	78	116	219

Obrázek 6.27 Blokový Lanczos - reortogonalizace - 15 procesů - čas

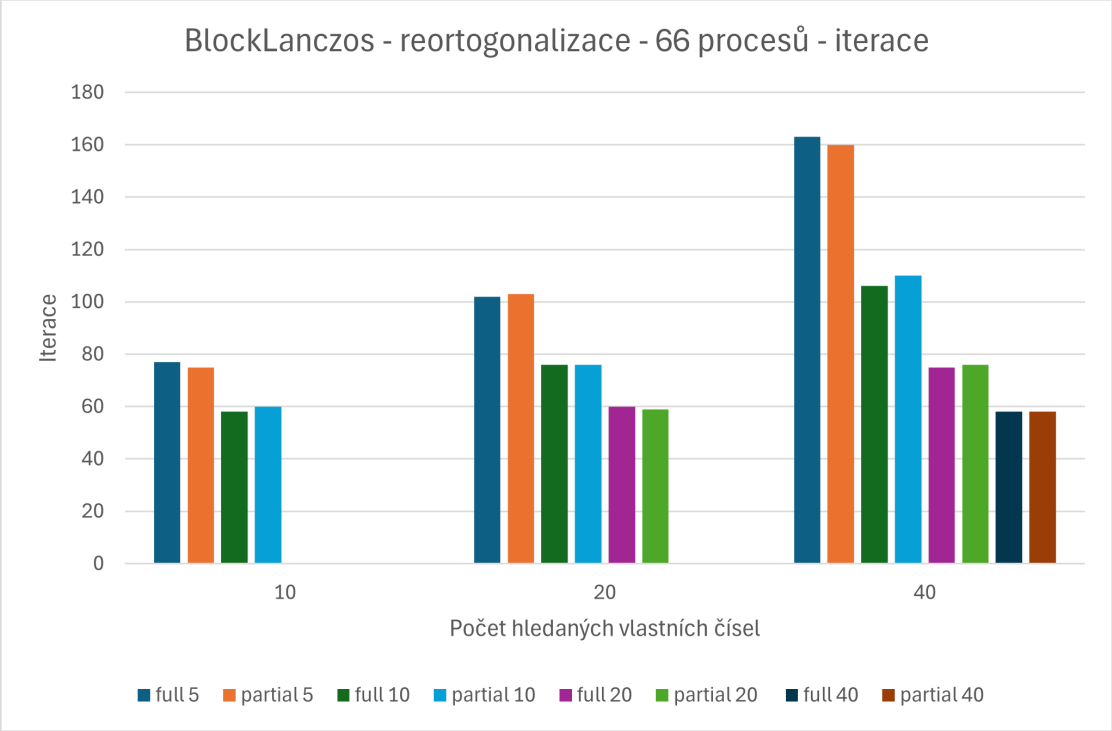


	10	20	40	100
Full	58	77	106	170
Multi1	59	78	106	170
Multi2	58	79	107	170
Partial	58	77	108	168

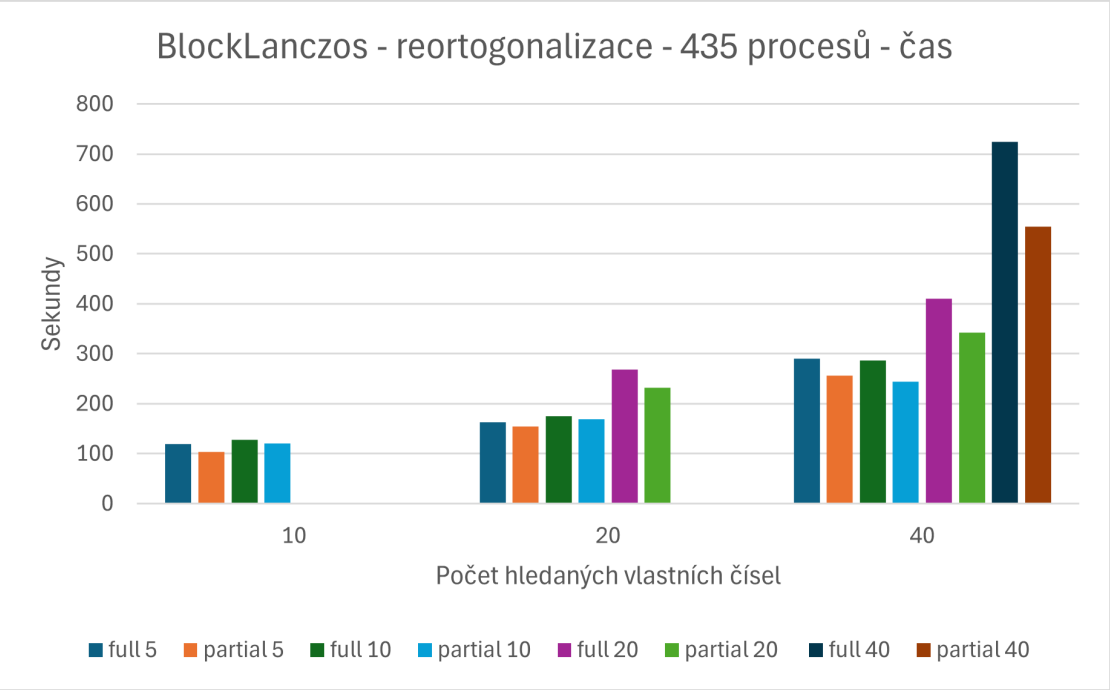
Obrázek 6.28 Blokový Lanczos - reortogonalizace - 15 procesů - iterace



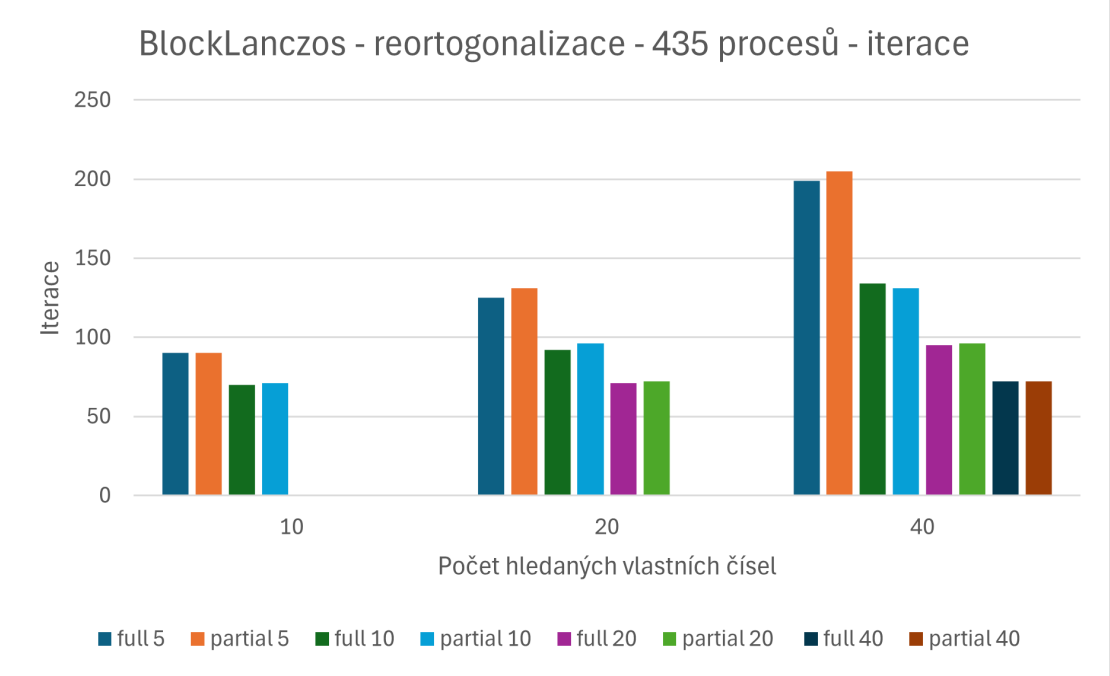
Obrázek 6.29 Blokový Lanczos - reortogonalizace - 66 procesů - čas



Obrázek 6.30 Blokový Lanczos - reortogonalizace - 66 procesů - iterace



Obrázek 6.31 Blokový Lanczos - reortogonalizace - 435 procesů - čas



Obrázek 6.32 Blokový Lanczos - reortogonalizace - 435 procesů - iterace

Grafické znázornění výsledku testu ukazuje:

- typ reortogonalizace neovlivňuje počet iterací algoritmu

- varianty *Multi1* a *Multi2* ve verzi s 15 procesy jsou pomalejší než ostatní varianty
- jelikož varianty *Multi1* a *Multi2* jsou stejně časově efektivní, nezáleží na nastavení maximálního počtu opakování reortogonalizace
- s přibývajícím počtem hledaných vlastních čísel je částečná reortogonalizace efektivnější než úplná, a to vzrůstajícím tempem
- závislost velikosti bloků u varianty s částečnou reortogonalizací se shoduje s testem v 6.8

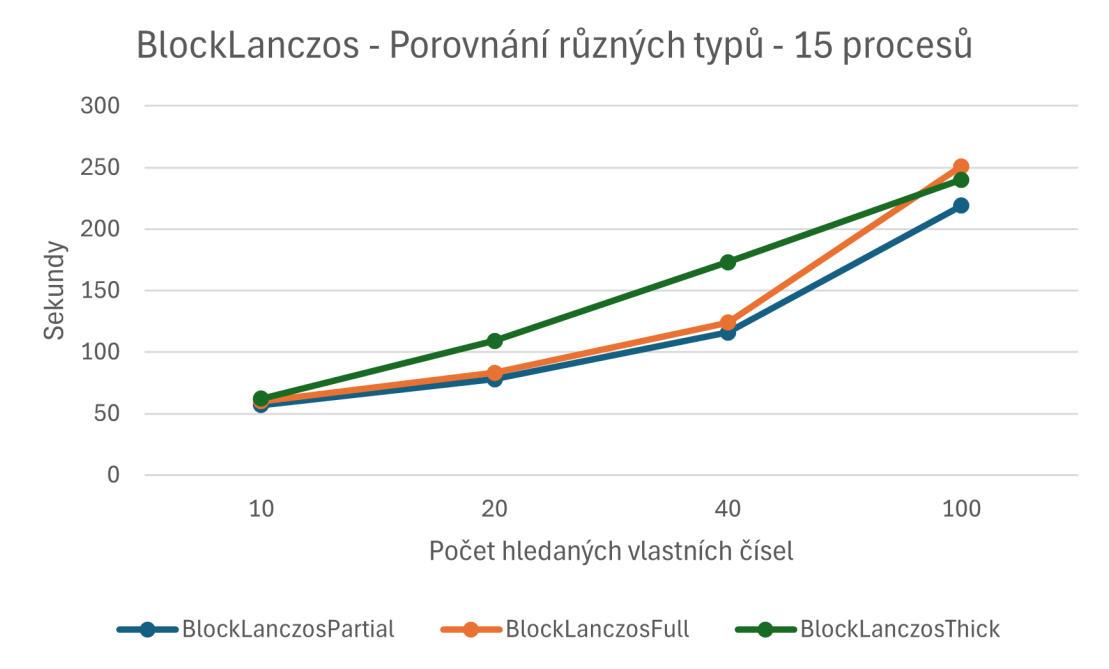
6.11 Porovnání blokové verze Lanczosova algoritmu

Tento test porovnává efektivitu různých variant blokové verze Lanczosova algoritmu, které byly použity v předchozích testech.

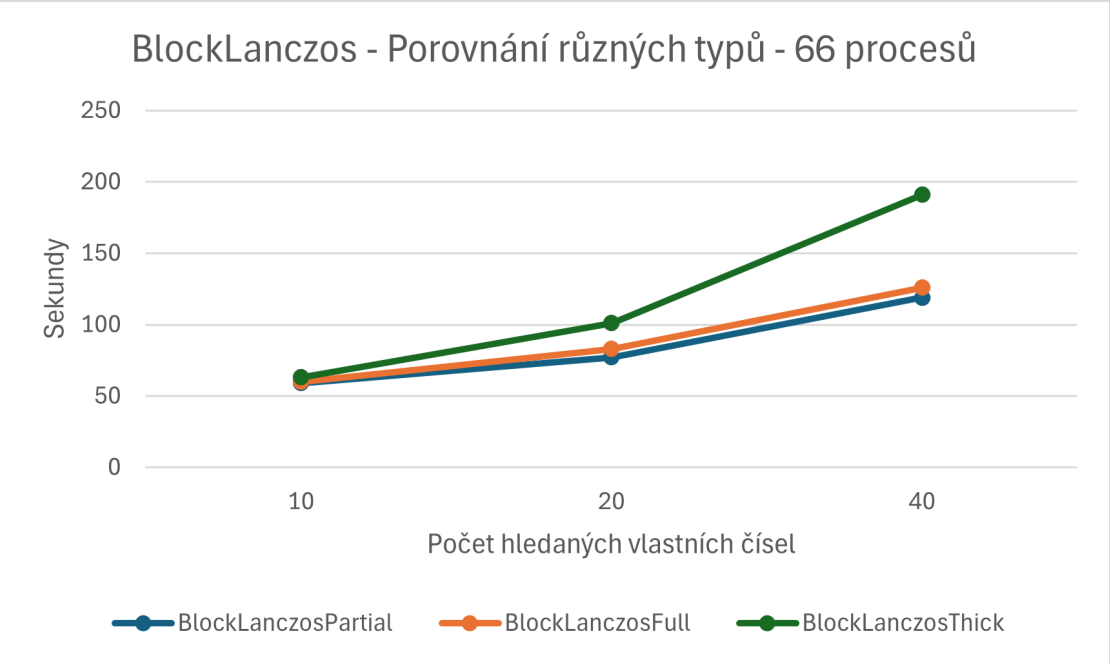
Testování bylo prováděno s blokem velikosti 10 vektorů.

Tyto varianty jsou:

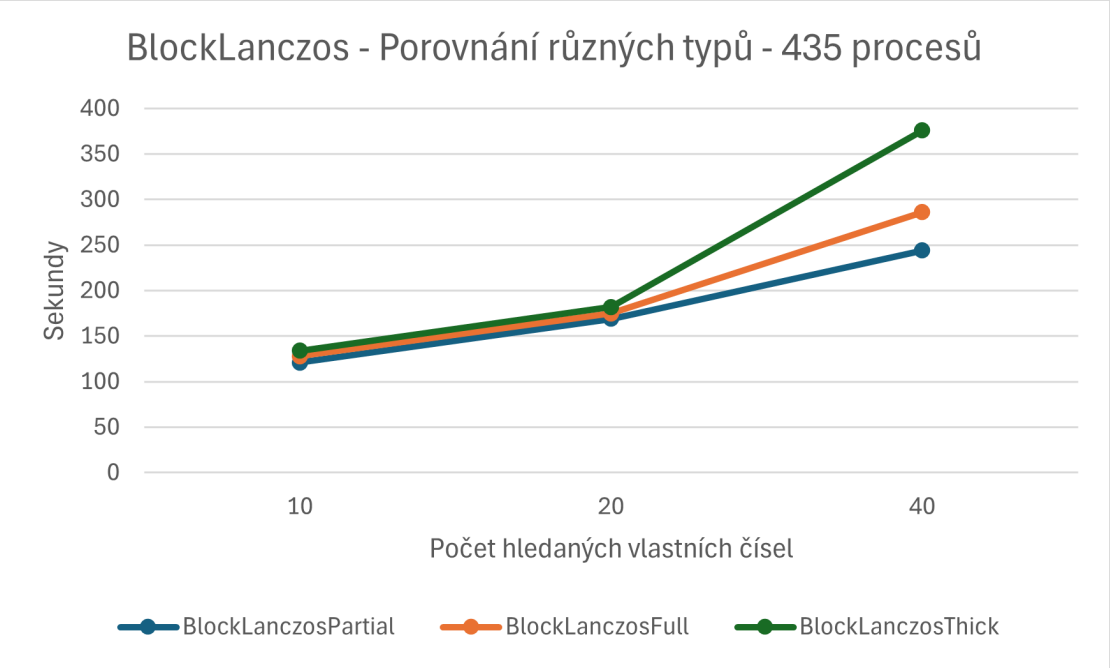
- *BlockLanczosPartial* parciální reortogonalizace
- *BlockLanczosFull* úplná reortogonalizace
- *BlockLanczosThick* thick restart (časově nejefektivnější varianta z testu 6.9)



Obrázek 6.33 Blokový Lanczos - porovnání různých typů algoritmu - 15 procesů - čas



Obrázek 6.34 Blokový Lanczos - porovnání různých typů algoritmu - 66 procesů - čas



Obrázek 6.35 Blokový Lanczos - porovnání různých typů algoritmu - 435 procesů - čas

Grafické znázornění výsledku testu ukazuje:

- časově nejefektivnější testovanou variantou je částečná reortogonalizace
- u varianty s 15 procesy a 100 hledaných vlastních čísel je úplná reortogonalizace pomalejší

než thick restart (důvodem je, že při hledání dostatečně velkého počtu vlastních čísel, počet Lanczosových vektorů naroste natolik, že restart začne být časově výhodnější)

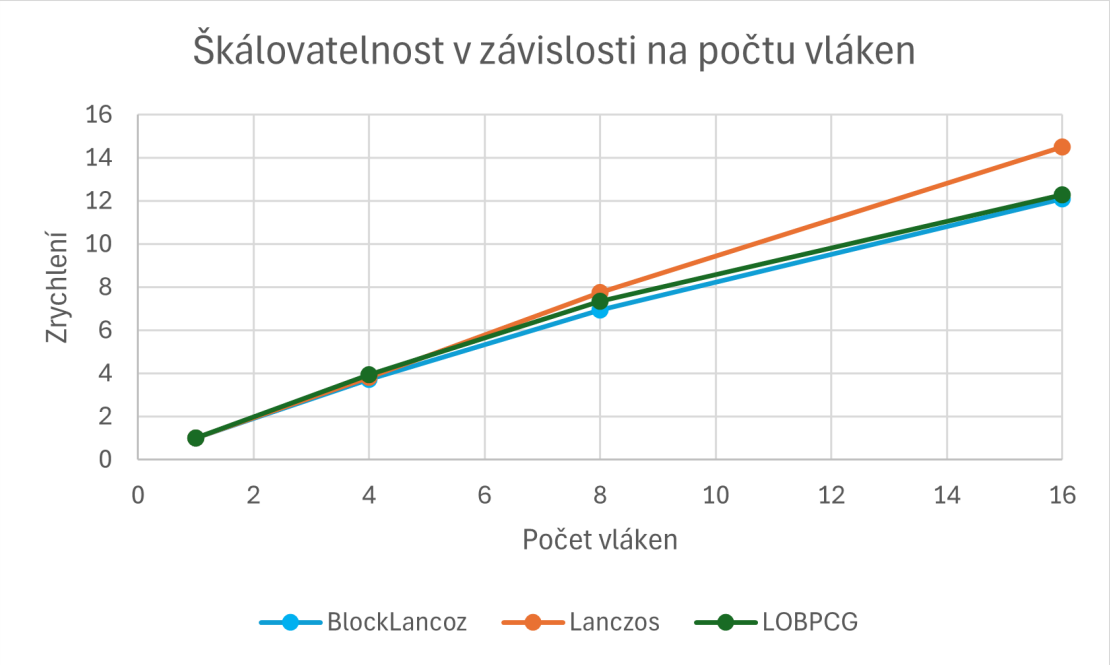
6.12 Škálovatelnost algoritmů vzhledem k počtu vláken

Tento test porovnává škálovatelnost třech hlavních použitých algoritmů.

Tyto algoritmy jsou:

- *BlockLanczos* bloková verze Lanczosova algoritmu s částečnou reortogonalizací
- *Lanczos* jedno-vektorový Lanczosův algoritmus s částečnou reortogonalizací
- *LOBPCG* metoda LOBPCG s velikostí bloku 25 vektorů

Test hledal 20 vlastních čísel.
Testování bylo spuštěno s 1, 4, 8, 16 vlákny.



Obrázek 6.36 Škálovatelnost v závislosti na počtu vláken

Všechny tři algoritmy jsou dobře škálovatelné (vzhledem k počtu použitých vláken), ale Lanczosův algoritmus je škálovatelný nejlépe.

6.13 Testování konvergence chyby hledaných vlastních čísel

Tento test demonstruje konvergence chyby hledaných vlastních čísel v závislosti na času a na počtu iterací algoritmu.

Tyto algoritmy jsou:

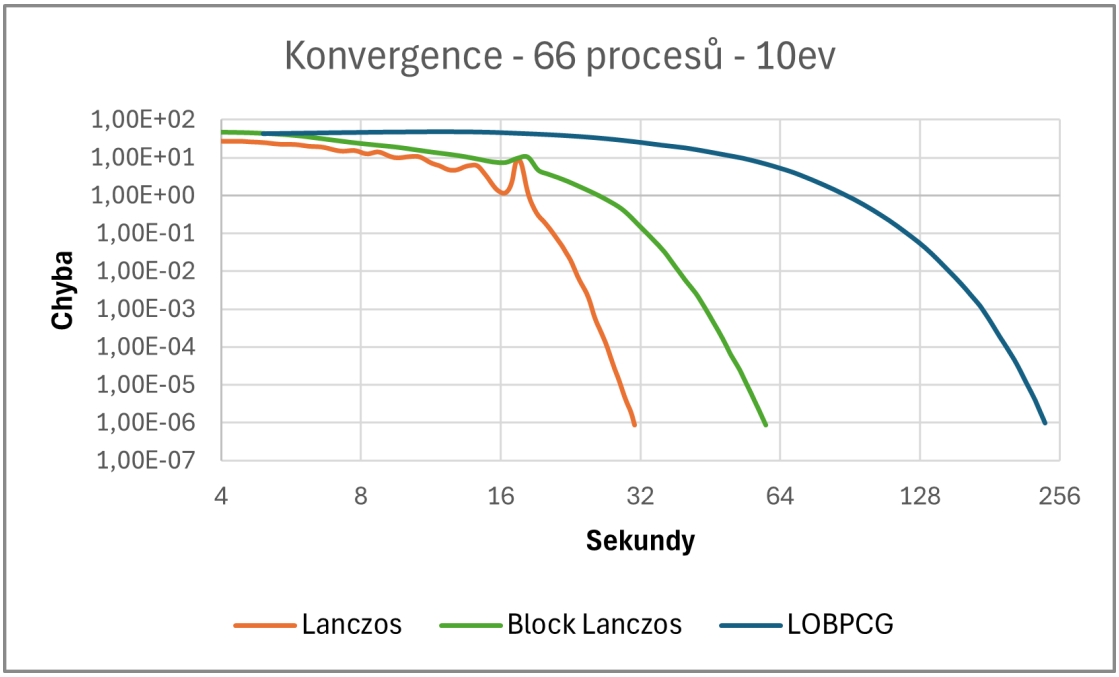
- *BlockLanczos* bloková verze Lanczosova algoritmu s částečnou reortogonalizací, u testování času s bloky velikosti 10 vektorů, u testování počtu iterací s bloky velikosti počtu hledaných vlastních čísel

- *Lanczos* Lanczosův algoritmus s částečnou reortogonalizací
- *LOBPCG* metoda LOBPCG s velikostí bloku = počet hledaných vlastních čísel + 5

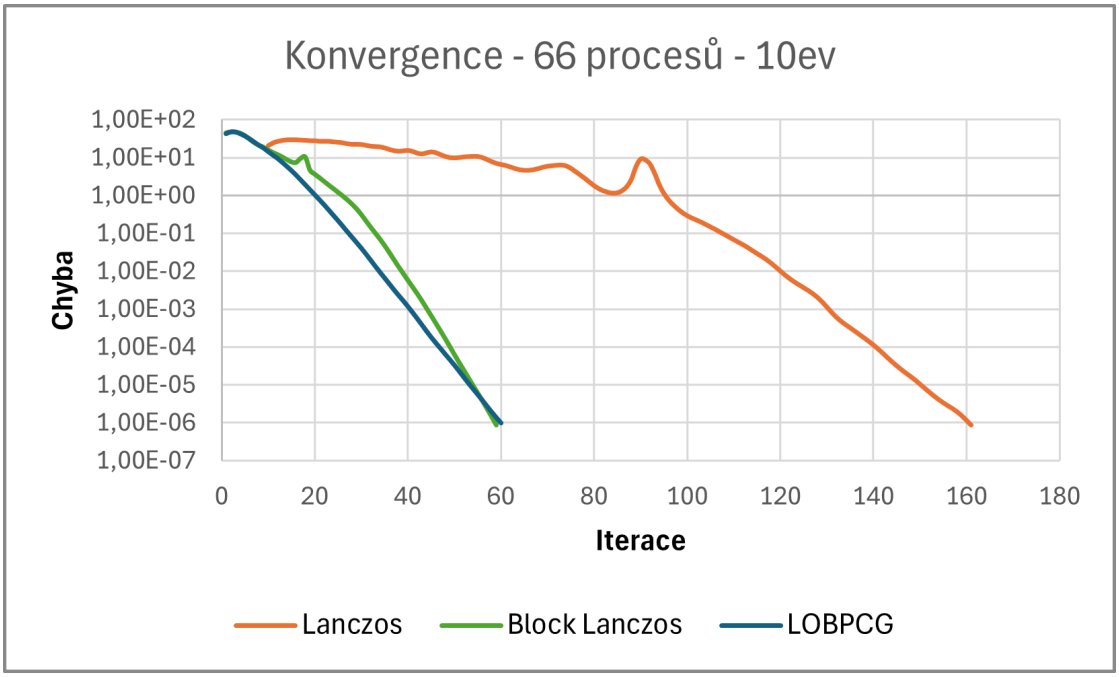
Test hledal 10 a 40 vlastních čísel a spouštěl se s 66 a 435 procesy.

Výsledek testu je vidět v následujících osmi grafech:

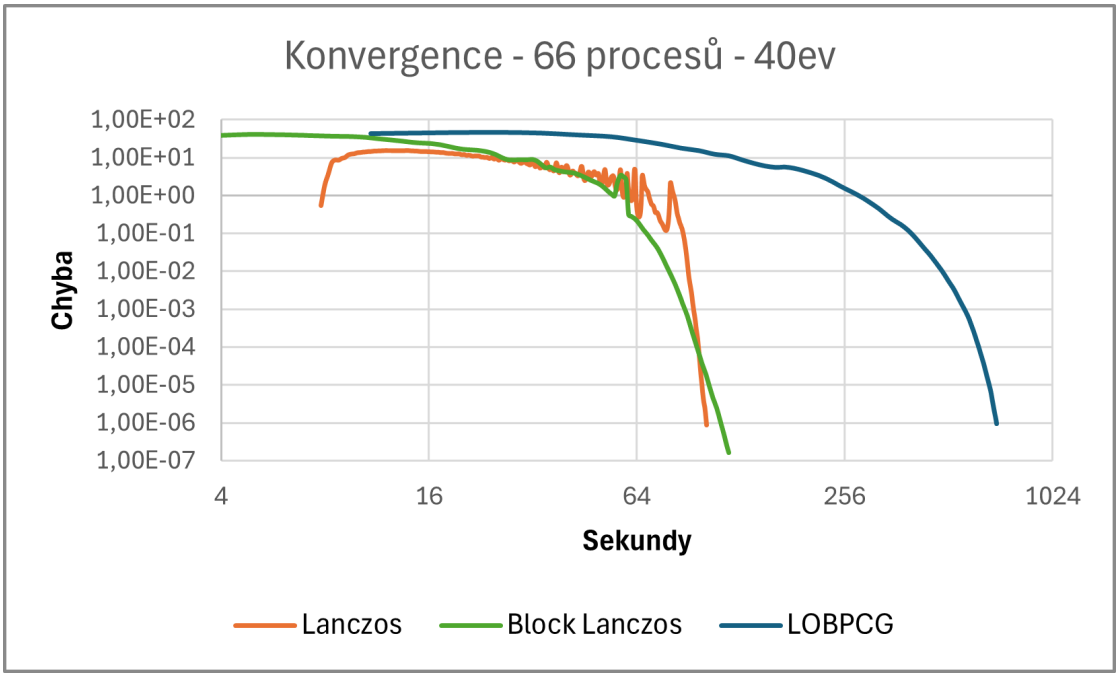
- 66 procesů, 10 hledaných vlastních čísel, testován čas
- 66 procesů, 10 hledaných vlastních čísel, testován počet iterací
- 66 procesů, 40 hledaných vlastních čísel, testován čas
- 66 procesů, 40 hledaných vlastních čísel, testován počet iterací
- 435 procesů, 10 hledaných vlastních čísel, testován čas
- 435 procesů, 10 hledaných vlastních čísel, testován počet iterací
- 435 procesů, 40 hledaných vlastních čísel, testován čas
- 435 procesů, 40 hledaných vlastních čísel, testován počet iterací



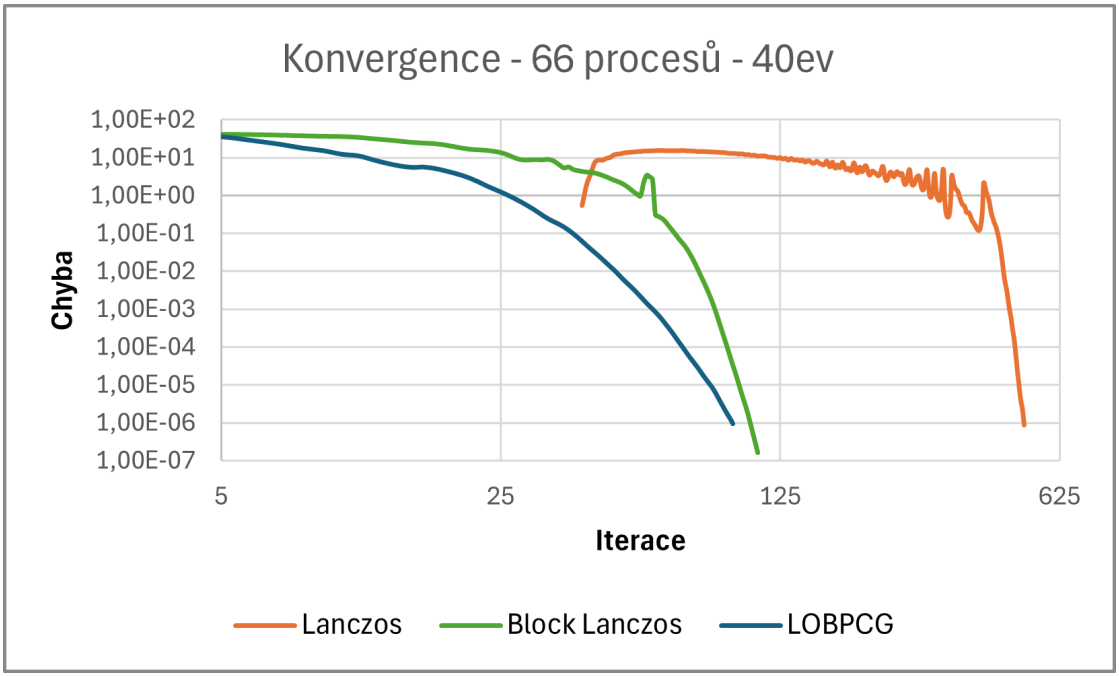
Obrázek 6.37 Konvergence chyby - 66 procesů - 10 vlastních čísel - čas



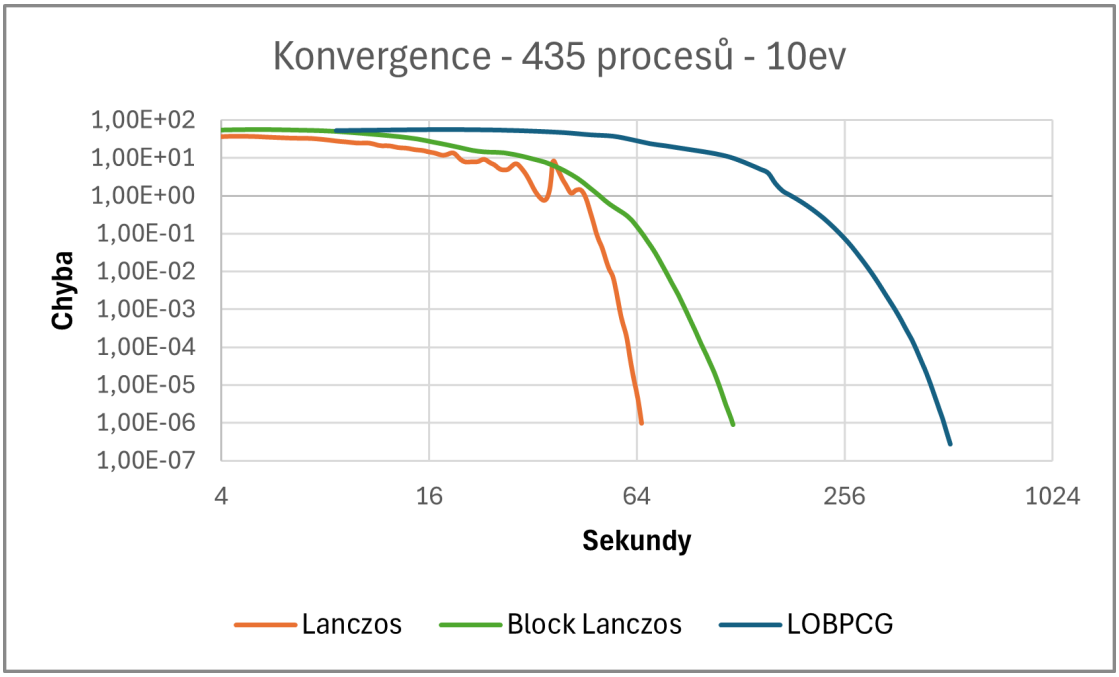
Obrázek 6.38 Konvergence chyby - 66 procesů - 10 vlastních čísel - iterace



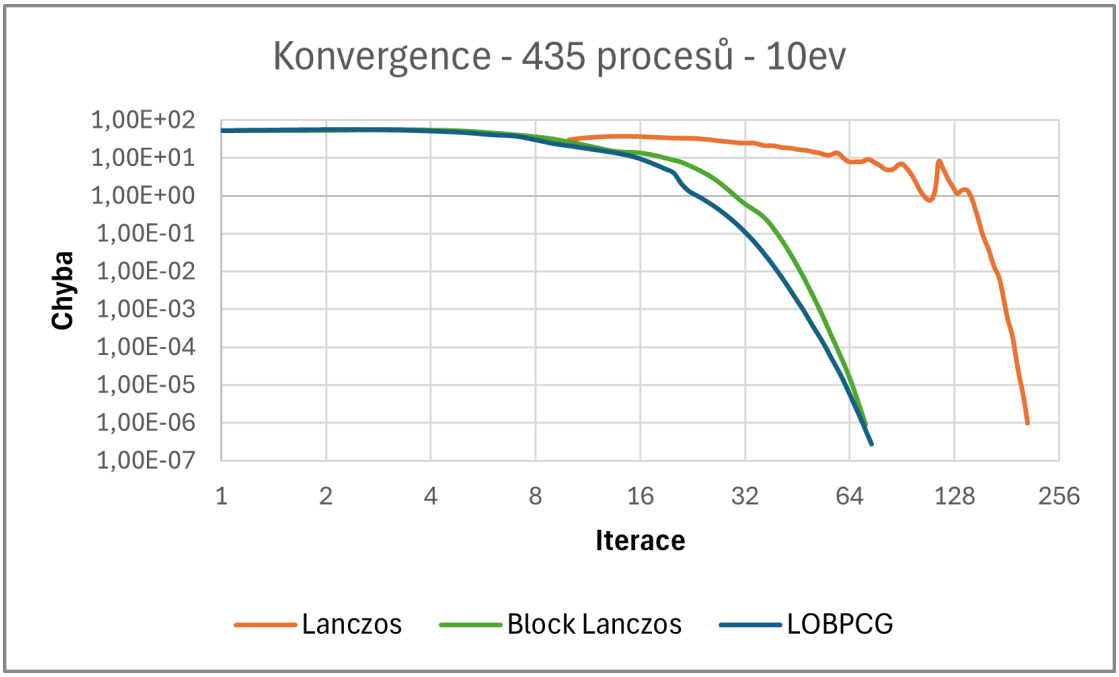
Obrázek 6.39 Konvergence chyby - 66 procesů - 40 vlastních čísel - čas



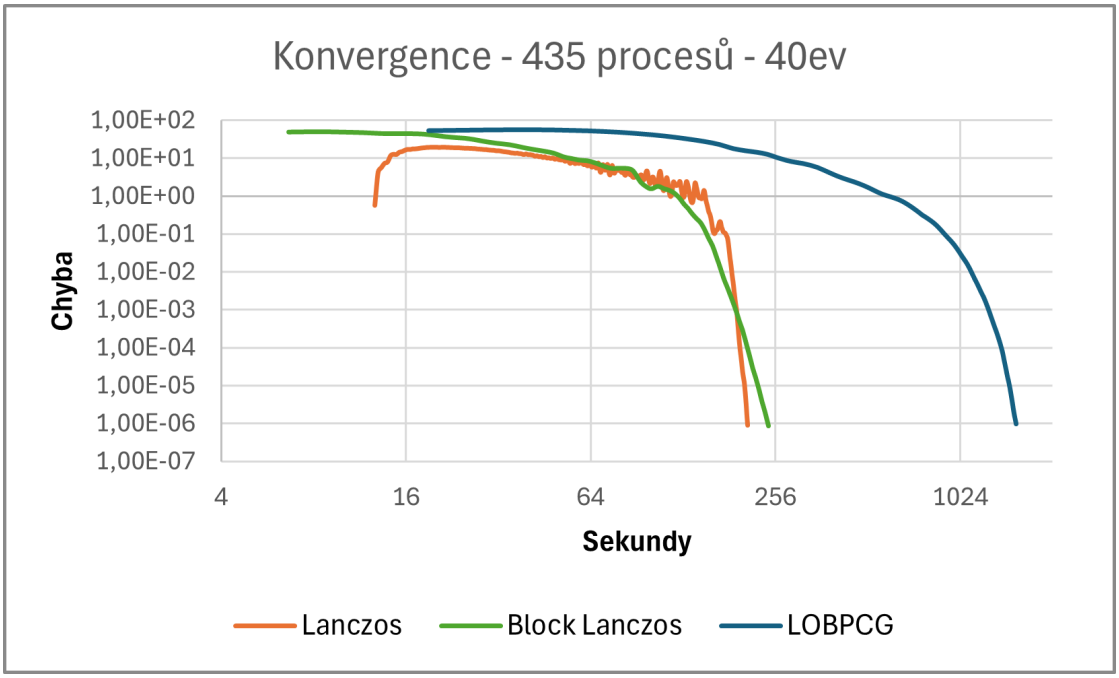
Obrázek 6.40 Konvergence chyby - 66 procesů - 40 vlastních čísel - iterace



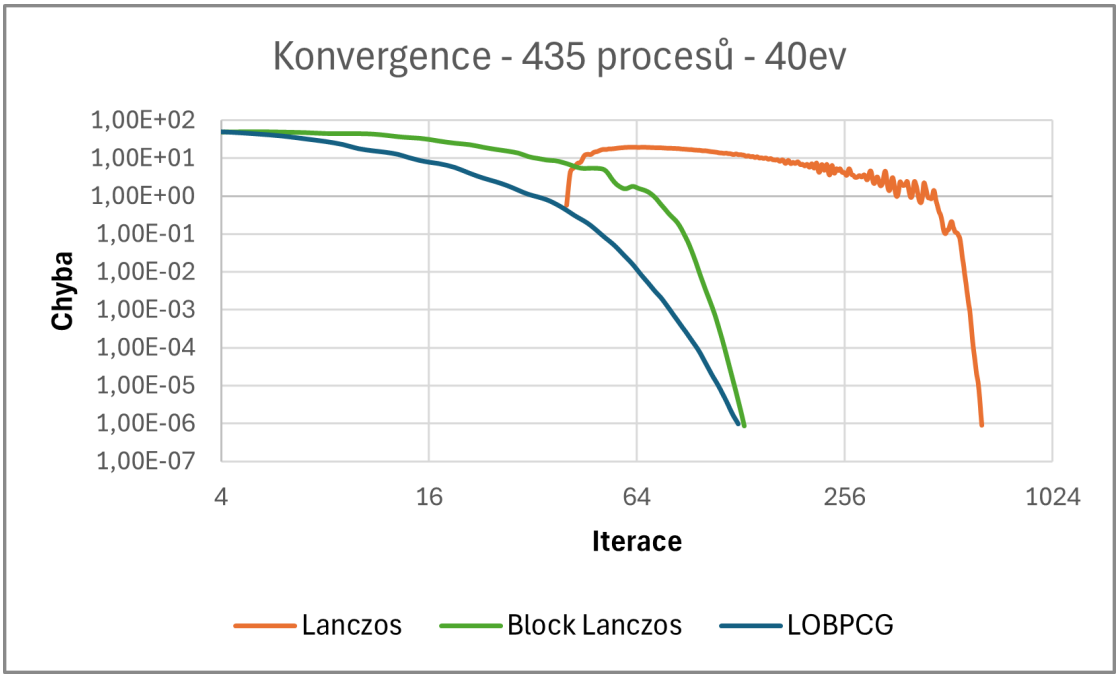
Obrázek 6.41 Konvergence chyby - 435 procesů - 10 vlastních čísel - čas



Obrázek 6.42 Konvergence chyby - 435 procesů - 10 vlastních čísel - iterace



Obrázek 6.43 Konvergence chyby - 435 procesů - 40 vlastních čísel - čas



Obrázek 6.44 Konvergence chyby - 435 procesů - 40 vlastních čísel - iterace

Grafické znázornění výsledku testu ukazuje:

- u testování počtu iterací má konvergence podobný průběh u algoritmů blokový Lanczos a LOBPCG
- u testování počtu iterací má LOBPCG ze začátku rychlejší konvergence chyby, avšak ke konci blokový Lanczos má konvergence chyby rychlejší
- Lanczosův algoritmus má iteračně konvergenci chyby značně pomalejší než testované blokové algoritmy, tento algoritmus provádí iterace po jednou vektoru
- časově je konvergence chyby algoritmu LOBPCG výrazně pomalejší než u jiných testovaných algoritmů
- časově je u hledání menšího počtu vlastních čísel konvergence chyby nejrychlejší u Lanczosova algoritmu (jedno-vektorový)
- v případě s hledáním většího počtu vlastních čísel se Lanczosův algoritmus časově vyrovnává blokové verzi Lanczosova algoritmu

6.14 Porovnání testovaných algoritmů

Tento test porovnává efektivitu jednotlivých algoritmů v závislosti na času a na počtu iterací algoritmu.

Tyto algoritmy jsou:

- *LOBPCG5* metoda LOBPCG s velikostí bloku = počet hledaných vlastních čísel + 5
- *BlockLanczos10Partial* bloková verze Lanczosova algoritmu s částečnou reortogonalizací s bloky velikosti 10 vektorů

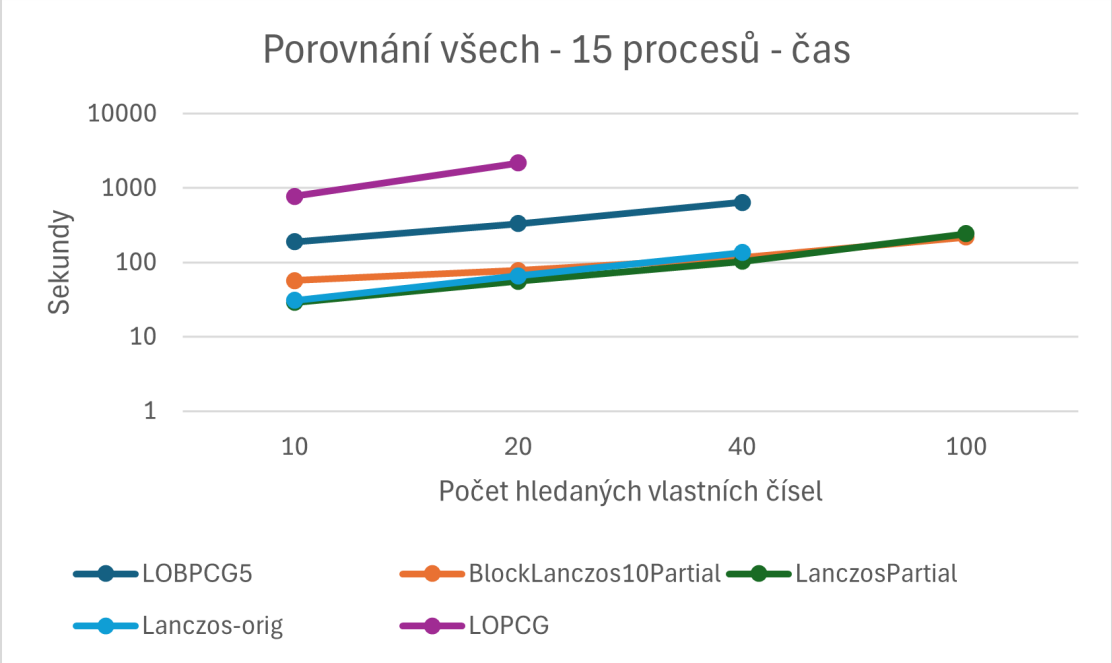
- *LanczosPartial* Lanczosův algoritmus s částečnou reortogonalizací
- *Lanczos-orig* původní Lanczosův algoritmus (4.2)
- *LOPCG* Jedno-vektorová metoda LOPCG
- *LOBPCG30* metoda LOBPCG s velikostí bloku = počet hledaných vlastních čísel + 30
- *BlockLanczosMaxPartial* bloková verze Lanczosova algoritmu s částečnou reortogonalizací s bloky velikosti počtu hledaných vlastních čísel

Původní blokový Lanczosův algoritmus není testován, protože jeho implementace v OMPI-Lancz byla nefunkční.

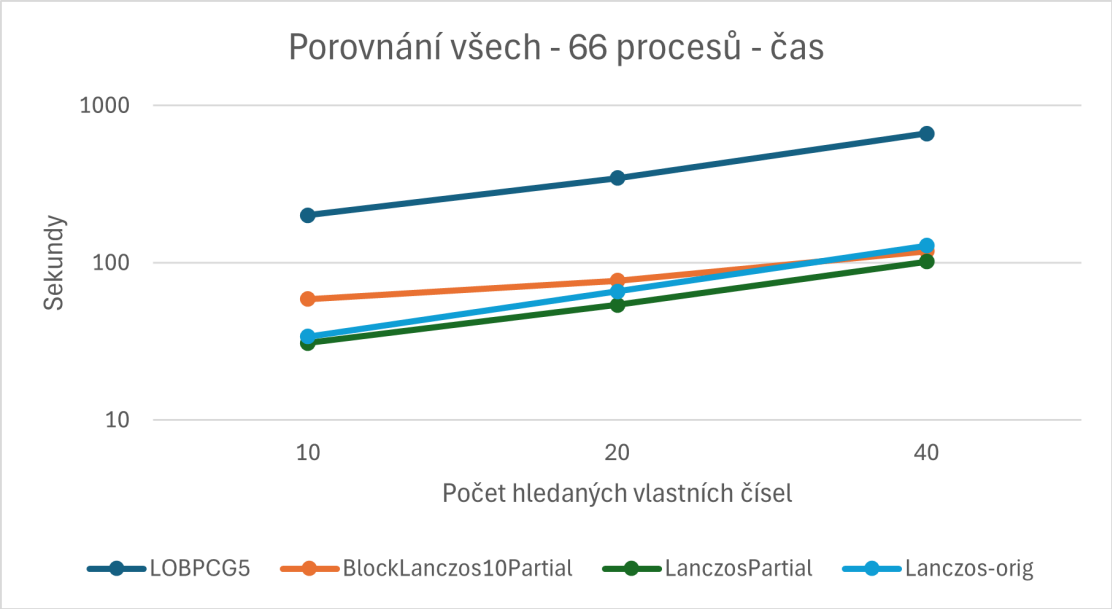
Test hledal 10, 20 a 40 vlastních čísel a spouštěl se s 15, 66 a 435 procesy.

Výsledek testu je vidět v následujících osmi grafech:

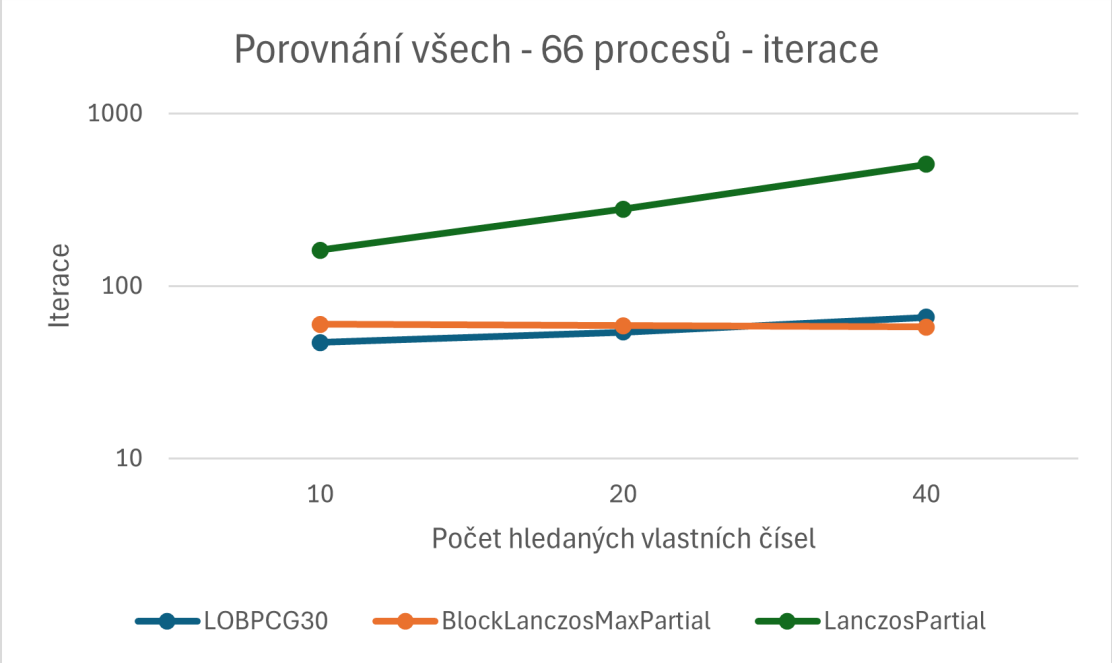
- 15 procesů, testován čas
- 66 procesů, testován čas
- 66 procesů, testován počet iterací
- 435 procesů, testován čas
- 435 procesů, testován počet iterací



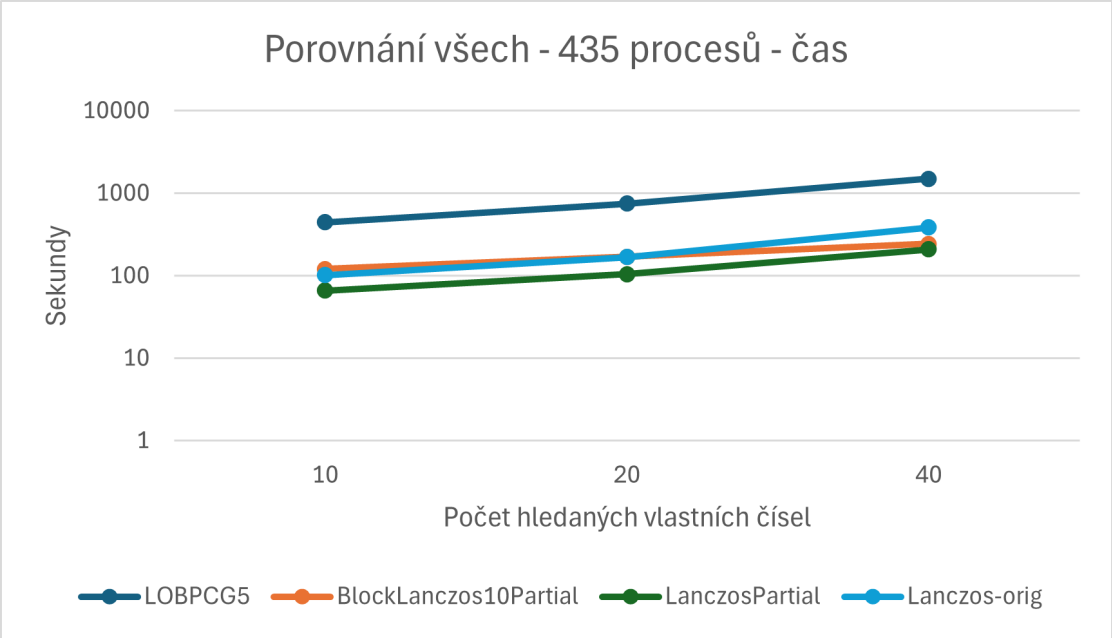
Obrázek 6.45 Celkové porovnání - 15 procesů - čas



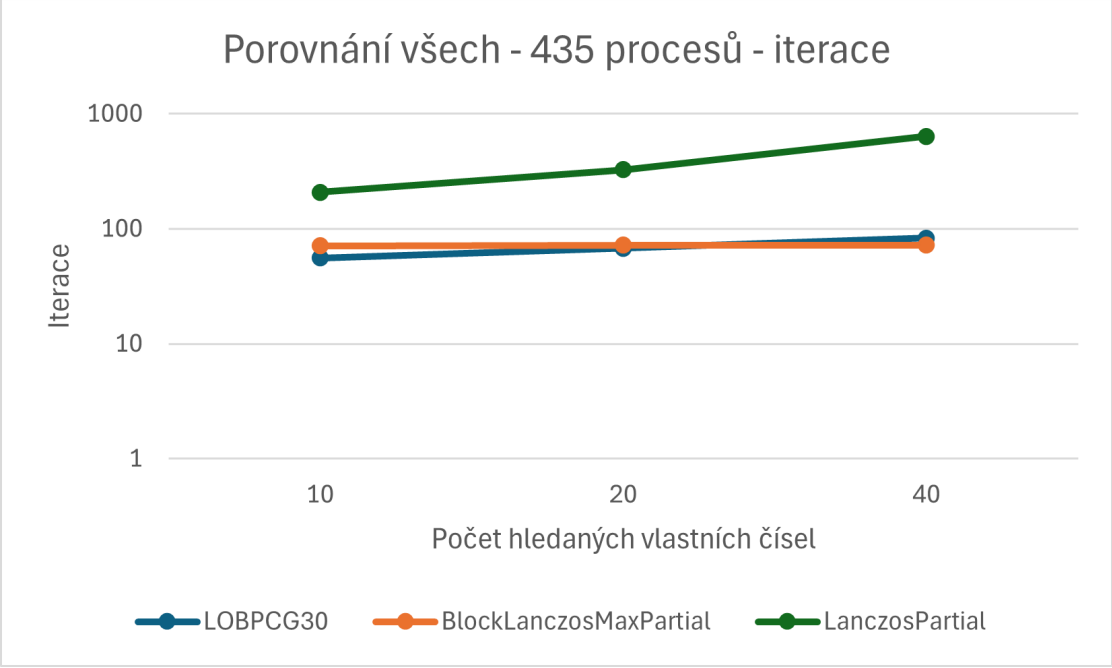
Obrázek 6.46 Celkové porovnání - 66 procesů - čas



Obrázek 6.47 Celkové porovnání - 66 procesů - iterace



Obrázek 6.48 Celkové porovnání - 435 procesů - čas



Obrázek 6.49 Celkové porovnání - 435 procesů - iterace

Grafické znázornění výsledku testu ukazuje:

- LOPCG je výrazně nejpomalejší z testovaných algoritmů
- časově je u hledání menšího počtu vlastních čísel nejrychlejší Lanczosův algoritmus (jednovektorový)

- s přibývajícím počtem hledaných vlastních čísel se mu časově přibližuje bloková verze Lanczosova algoritmu, dokonce v případě o 15 procesech s hledáním 100 vlastních čísel je již bloková verze nejrychlejší
- metoda LOBPCG je časově výrazně pomalejší, jelikož násobení vstupní matice se provádí trojitým blokem vektorů (viz 16)
- referenční řešení Lanczosova algoritmu je méně efektivní než Lanczosův algoritmus s částečnou reortogonalizací
- z hlediska počtu iterací je Lanczosův algoritmus značně neefektivní
- iteračně je u hledání menšího počtu vlastních čísel nejrychlejší testovaná metoda LOBPCG
- u většího počtu hledaných vlastních čísel je iteračně nejefektivnější testovaný algoritmus blokové verze Lanczose

6.15 Shrnutí testování

Testování lze rozdělit na dvě skupiny podle kritéria, časová efektivita a efektivita podle počtu iterací.

Z hlediska časové efektivity

- při hledání menšího počtu vlastních čísel je nejefektivnější Lanczosův algoritmus s částečnou reortogonalizací
- restarty nezlepšují efektivitu v testovaných případech oproti částečné reortogonalizaci, naznačují ale možné zlepšení při vyšším počtu hledaných vlastních čísel (> 100)
- pro blokový Lanczos i pro jedno-vektorový Lanczos je v testovaných případech vždy nejefektivnější varianta s částečnou reortogonalizací
- nejefektivněčjší velikost bloků pro blokový Lanczos je 5 a 10 vektorů
- a pro metodu LOBPCG to je velikost = 5 + počet hledaných vlastních čísel

Z hlediska počtu iterací

- čím větší blok u blokových testovaných algoritmů, tím je počet iterací menší
- testované blokové algoritmy jsou efektivnější než jedno-vektorový Lanczos
- s přibývajícím počtem hledaných vlastních čísel se blokový Lanczos stává efektivnější než metoda LOBPCG
- různé varianty reortogonalizace výrazně neovlivňují počet iterací

6.16 Testy správnosti

Všechny výše popsané testy jsou prováděny na třech různých vstupních maticích. Tyto matice jsou vygenerované dodanými modelovými prostory 10B s Nmax 6, 12C s Nmax 6 a 10B s Nmax 8.

Výstupem jsou pak aproximace vlastních čísel, jejich reziduální chyby a vlastní vektory těchto vstupních matic.

Test správnosti se provádí porovnáním výsledných vlastních čísel referenční implementace algoritmu se všemi testovanými verzemi algoritmů popsanými v této práci.

Tato porovnání vedla k potvrzení správnosti aproximace výsledných vlastních čísel.

Další potvrzení správnosti výsledných vlastních čísel a vektorů se provádí výpočtem reziduálních chyb těchto vlastních čísel.

Tyto chyby u žádných testů nepřekročily požadovanou mez.

Výjimkou je test implicitního restartu Lanczosova algoritmu (viz 6.4), kde aproximace vlastních čísel ne vždy konvergovaly ke správným výsledkům. Ani pokus o řešení rozšířením přesnosti všech proměnných z *double* na *long double* nevedl ke zlepšení.

Kapitola 7

Závěr

Cílem práce bylo navržení, implementace a následné experimentální porovnání všech původních i navržených a implementovaných metod na rozsáhlých řídkých maticích za použití superpočítače.

Tyto implementace jsou jedno-vektorový Lanczosův algoritmus (původní i vlastní), blokový Lanczosův algoritmus, jedno-vektorová LOBPCG metoda a bloková LOBPCG metoda. Původní implementace blokové verze Lanczosova algoritmu byla nahrazena vlastní implementací, protože původní implementace nebyla kompletní.

Práce dokumentuje teoretickou přípravu od stanovení cílů, přes popis základních pojmů (viz kap. 1) a uvedení algoritmů řešící problém hledání vlastních čísel a vektorů řídkých symetrických matic (viz kap. 2), až k implementaci původního (viz kap. 4) i vlastního řešení práce (viz kap. 5).

Popsané algoritmy byly implementovány v jazyce C++.

Testování a experimentální porovnání původních a implementovaných metod je popsáno v kapitole 6. Probíhalo na superpočítači Karolina. Zaměřeno bylo na srovnání efektivity (časové náročnosti a počtu iterací algoritmu) jednotlivých algoritmů a jejich verzí. Porovnání s původním řešením prokázalo vylepšenou efektivitu mého řešení.

Další možností rozšíření práce by mohlo být u blokové verze Lanczosova algoritmu varianta zvýšení velikosti bloků nad rámce počtu hledaných vlastních čísel. Testování ukázalo, že s rostoucí velikostí bloků se snižuje počet iterací algoritmu. A dále možnost testování na ještě rozsáhlejších maticích, které superpočítač Karolina neumožnil kvůli vysokému nároku na paměť.

Na vylepšení efektivity LOBPCG metody je možnost dodatečné implementace předkondicionéru (viz 2.2), která by měla zrychlit konvergenci hledaných vlastních čísel.

Bibliografie

1. *Posouvání hranic ab initio výpočtů jaderné struktury* [online]. 2022 [cit. 2024-02-14]. Dostupné z: https://starfos.tacr.cz/projekty/GA22-14497S.
2. ANTON, Howard. *Elementary Linear Algebra*. J. Wiley, Canada, 2000. ISBN 0-471-17055-0.
3. HERNANDEZ, V.; ROMAN, J.E.; TOMAS, A.; VIDAL, V. Orthogonalization Routines in SLEPc. 2007. Dostupné také z: https://slepc.upv.es/documentation/reports/str1.pdf.
4. GIRAUD, Lucky; LANG, Julien; ROZLOŽNÍK, Miro; ESHOF, Jasper van den. Rounding Error Analysis of the Classical Gram–Schmidt Orthogonalization Process. [B.r.]. Dostupné také z: http://www2.cs.cas.cz/mweb/download/abstr/RoHOUS2005.pdf.
5. HERNANDEZ, V.; ROMAN, J.E.; TOMAS, A.; VIDAL, V. Lanczos Methods in SLEPc. 2006. Dostupné také z: https://slepc.upv.es/documentation/reports/str5.pdf.
6. SIMON, Horst D. The Lanczos Algorithm for Solving Symmetric Linear Systems. In: 1982. Dostupné také z: https://www.researchgate.net/publication/235084574_The_Lanczos_Algorithm_for_Solving_Symmetric_Linear_Systems.
7. CULLUM, Jane; WILLOUGHBY, Ralph A. A Survey of Lanczos Procedures for Very Large Real 'Symmetric' Eigenvalue Problems. *Journal of Computational and Applied Mathematics*. 1985, s. 37–60. Dostupné také z: https://core.ac.uk/download/pdf/82275235.pdf.
8. MEURANT, Gerard; STRAKOŠ, Zdeněk. The Lanczos and conjugate gradient algorithms in finite precision arithmetic. *Acta Numerica*. 2006, roč. 15, s. 471–542. Dostupné také z: https://www.karlin.mff.cuni.cz/~strakos/download/2006_MeSt.pdf.
9. HERNANDEZ, V.; ROMAN, J.E.; TOMAS, A.; VIDAL, V. Arnoldi Methods in SLEPc. 2006. Dostupné také z: https://slepc.upv.es/documentation/reports/str4.pdf.
10. KIM, Yong Joo. Block Lanczos Algorithm. In: Naval Postgraduate School, 1989. Dostupné také z: https://apps.dtic.mil/sti/tr/pdf/ADA224011.pdf.
11. MEERBERGEN, Karl; SCOTT, Jennifer A. The Design of a Block Rational Lanczos Code with Partial Reorthogonalization and Implicit Restarting. 1970. Dostupné také z: https://www.researchgate.net/publication/2642472_The_design_of_a_block_rational_Lanczos_code_with_partial_reorthogonalization_and_implicit_restarting.
12. GRIMES, Roger G.; LEWIS, John Gregg; SIMON, Horst D. A Shifted Block Lanczos Algorithm for Solving Sparse Symmetric Generalized Eigenproblems. *SIAM J. Matrix Anal. Appl.* 1994, roč. 15, s. 228–272. Dostupné také z: https://www.researchgate.net/publication/24290905_A_Shifted_Block_Lanczos_Algorithm_for_Solving_Sparse_Symmetric_Generalized_Eigenproblems.

13. ABDELMALEK, N.N. Roundoff Error Analysis for Gram–Schmidt Method and Solution of Linear Least Squares Problems. 1971, roč. 11, s. 345–368.

14. RUTISHAUSER, Heinz. *Description of Algol 60. Handbook for Automatic Computation, Vol. 1a.* Springer Berlin, Heidelberg, Germany, 1967. ISBN 978-3-662-38103-8.

15. PAIGE, C. C. Computational Variants of the Lanczos Method for the Eigenproblem. *IMA Journal of Applied Mathematics*. 1972, roč. 10, č. 3, s. 373–381. ISSN 0272-4960. Dostupné z DOI: 10.1093/imamat/10.3.373.

16. PAIGE, C. C. Error Analysis of the Lanczos Algorithm for Tridiagonalizing a Symmetric Matrix. *IMA Journal of Applied Mathematics*. 1976, roč. 18, č. 3, s. 341–349. ISSN 0272-4960. Dostupné z DOI: 10.1093/imamat/18.3.341.

17. PAIGE, C. C. Accuracy and Effectiveness of the Lanczos Algorithm for the Symmetric Eigenproblem. *Linear Algebra and its Applications*. 1979, roč. 34, s. 341–349. ISSN 0024-3795. Dostupné z DOI: https://doi.org/10.1016/0024-3795(80)90167-6.

18. PARLETT, Beresford N.; SCOTT, David S. The Lanczos Algorithm with Selective Orthogonalization. *Mathematics of Computation*. 1979, roč. 33, s. 217–238. Dostupné také z: https://www.ams.org/journals/mcom/1979-33-145/S0025-5718-1979-0514820-3/S0025-5718-1979-0514820-3.pdf.

19. QIAO, Sanzheng; LIU, Guohong; XU, Wei. Block Lanczos Tridiagonalization of Complex Symmetric Matrices. In: Department of Computing a Software, McMaster University, Hamilton, Ontarios, 2005. Dostupné také z: http://www.cas.mcmaster.ca/~qiao/publications/spie05.pdf.

20. SIMON, Horst D. The Lanczos Algorithm with Partial Reorthogonalization. *Mathematics of Computation*. 1984, roč. 42, s. 115–142. Dostupné také z: https://www.ams.org/journals/mcom/1984-42-165/S0025-5718-1984-0725988-X/S0025-5718-1984-0725988-X.pdf.

21. CALVETTI, Daniela; REICHEL, Lothar; SORENSEN, Danny C. An Implicitly Restarted Lanczos Method for Large Symmetric Eigenvalue Problems. *Electronic Transaction on Numerical Analysis*. 1994, roč. 2, s. 1–21. Dostupné také z: https://www.emis.de/journals/ETNA/vol.2.1994/pp1-21.dir/abstr1-21.pdf.

22. SHIMIZU, Noritaka; MIZUSAKI, Takahiro; UTSUNO, Yutaka; TSUNODA, Yusuke. Thick-Restart Block Lanczos Method for Large-Scale Shell-Model Calculations. *Comput. Phys. Commun.* 2019, roč. 244, s. 372–384. Dostupné také z: https://www.semanticscholar.org/reader/9215be96616777237a75e1681ca3f646aab62844.

23. KNYAZEV, Andrew V. Toward the Optimal Preconditioned Eigensolver: Locally Optimal Block Preconditioned Conjugate Gradient Method. *SIAM Journal on Scientific Computing*. 2000, roč. 23. Dostupné také z: https://www.researchgate.net/publication/2434555_Toward_The_Optimal_Preconditioned_Eigensolver_Locally_Optimal_Block_Preconditioned_Conjugate_Gradient_Method.

24. SHAO, Meiyue; AKTULGA, Hasan Metin; YANG, Chao; NG, Esmond G.; MARIS, Pieter; VARY, James P. Accelerating Nuclear Configuration Interaction Calculations through a Preconditioned Block Iterative Eigensolver. *Computer Phys. Commun.* 2017, s. 1–25. Dostupné také z: https://shorturl.at/blDF0.

25. AKTULGA, Hasan Metin; AFIBUZZAMAN, Md.; WILLIAMS, Samuel; BULUC, Aydin; SHAO, Meiyue; YANG, Chao; NG, Esmond G.; MARIS, Pieter; VARY, James P. A High Performance Block Eigensolver for Nuclear Configuration Interaction Calculations. In: 2017. Dostupné také z: https://ieeexplore.ieee.org/ielaam/71/7927510/7748453-aam.pdf.

26. DUERSCH, Jed A.; SHAO, Meiyue; YANG, Chao; GU, Ming. A Robust and Efficient Implementation of LOBPCG. *SIAM J. Sci. Comput.* 2017, roč. 40, s. C655–C676. Dostupné také z: https://shorturl.at/tLW69.

27. ARBENZ, Peter. Lecture Notes on Solving Large Scale Eigenvalue Problems. Chapter 13 - Rayleigh quotient and trace minimization. In: 2012, s. 241–258. Dostupné také z: https://api.semanticscholar.org/CorpusID:12720088.

28. BOARD, OpenMP Architecture Review. *OpenMP Application Program Interface Version 5.2*. 2021. Dostupné také z: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf.

29. COMMUNITY, The Open MPI. *Open MPI v5.0.x* [online]. 2021 [cit. 2024-02-14]. Dostupné z: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf.

30. *LAPACK — Linear Algebra PACKage* [online]. 2023 [cit. 2024-02-14]. Dostupné z: https://www.netlib.org/lapack/.

31. LANGR, Daniel; DYTRYCH, Tomáš. *The HPC-enabled Lanczos Eigensolver for the Checkerboard Matrix Partitioning*. 2023. Unpublished presentation.

32. VALÈNCIA, Universitat Politècnica de. *SLEPc: Scalable Library for Eigenvalue Problem Computations* [online]. 2021 [cit. 2024-02-14]. Dostupné z: https://slepc.upv.es/documentation/.

33. TEAM, The Trilinos Project. *The Trilinos Project Website* [online]. 2020 [cit. 2024-02-14]. Dostupné z: https://trilinos.github.io.

34. IT4INNOVATIONS. *Technical Information of the Karolina Supercomputer* [online]. 2020 [cit. 2024-02-14]. Dostupné z: https://www.it4i.cz/en/infrastructure/karolina.

Obsah přiloženého média

- `readme.txt` ..... stručný popis obsahu média
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF
  - `zadani.pdf` ..... zadání práce ve formátu PDF