CENTER FOR
MACHINE PERCEPTION

CZECH TECHNICAL
UNIVERSITY

DIPLOMOVÁ PRÁCE

ISSN 1213-2365

Paralelizace řešení přímého problému EEG rekonstrukce metodou BEM

Marek Jasanský

marajas@centrum.cz

CTU–CMP–2007–11

25. května 2007

Školitel: Dr. Ing. Jan Kybic

Research Reports of CMP, Czech Technical University in Prague, No. 11, 2007

Published by

Centrum strojového vnímání, Katedra kybernetiky
Fakulta elektrotechnická ČVUT
Technická 2, 166 27 Praha 6
fax: (02) 2435 7385, tel: (02) 2435 7637, www: http://cmp.felk.cvut.cz

České vysoké učení technické v Praze - Fakulta elektrotechnická

Katedra kybernetiky *Školní rok:* 2005/2006

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: Marek Jasanský

Obor: Biomedicínské inženýrství

Název tématu: Paralelizace řešení přímého problému EEG rekonstrukce metodou BEM

Zásady pro vypracování:

1. Seznamte se s problémem výpočtu elektrických polí pro prostorovou rekonstrukci EEG dat.
2. Seznamte se s metodou hraničních prvků (BEM) a jejím použitím pro EEG problém.
3. Seznamte se s metodou urychlení pomocí multipolární expanze (FMM).
4. Seznamte se s existující implementací a navrhněte způsob paralelizace, aby výsledný program byl použitelný na cluster počítačů PC.
5. Experimentálně ověřte dosažené zrychlení a zvýšení maximálního rozsahu problému.

Seznam odborné literatury:

[1] Jan Kybic et al: A Common Formalism for the Integral Formulation of the Forward EEG Method. IEEE TMI, vol. 24, 2005

[2] Jan Kybic et al: Fast Multipole Acceleration of the MEG/EEG Boundary Element Method. Physic in Medicine and Biology, 50, 2005

Vedoucí diplomové práce: Dr. Ing. Jan Kybic

Termín zadání diplomové práce: letní semestr 2005/2006

Termín odevzdání diplomové práce: květen 2007

prof. Ing. Vladimír Mařík, DrSc.
vedoucí katedry

L.S.

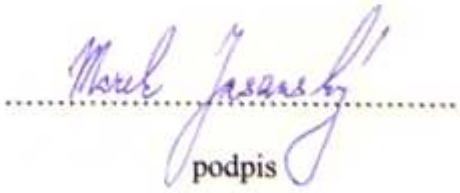
prof. Ing. Zbyněk Škvor, CSc.
děkan

V Praze dne 30.05.2006

Prohlášení

Prohlašuji, že jsem svou diplomovou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, SW atd.) uvedené v přiloženém seznamu.

V Praze dne 25. 5. 2007



.....

podpis

Abstrakt

Tato práce navrhuje a implementuje paralelní verzi již existujícího řešení EEG/MEG dopředného problému metodou hraničních prvků (BEM) v jazyce OCaml. Ke studiu činnosti mozku a k diagnostice mozkových poruch by bylo dobré znát rozložení zdrojů proudu (nosičů informace) uvnitř mozku. Tento problém řeší inverzní úloha, která konfiguraci zdrojů počítá z napětí změřeného na povrchu hlavy. Abychom mohli vyřešit inverzní úlohu, musíme najít řešení dopředné úlohy. Ta vypočítává potenciál elektrického pole na povrchu hlavy ze známého rozložení zdrojů elektrického proudu uvnitř mozku. Proudové zdroje v lidském těle jsou však velmi malé, proto pro požadovanou přesnost musíme hlavu rozdělit na řádově stovky tisíc objemových elementů a zjišťovat parametry elektromagnetického pole v každém z nich. Tento výpočet je velmi náročný a na běžných počítačích obtížně realizovatelný, cílem této práce je tedy urychlit ho řešením úlohy na více spolupracujících počítačích (výpočetním clusteru). Řešení je navrženo pro heterogenní cluster metodou zasílání zpráv v systému MPI.

Abstract

The goal of this work is to propose and develop parallel version of existing EEG/MEG forward problem solution in OCaml language. The problem is solved by Boundary Element Method (BEM). It would be best to now spatial configuration of current source inside head for studying cerebral activity. This task is called inverse problem, but to solve it we need to know a solution of forward problem. Forward task compute electric potencial on the surface of head from known configuration of current source inside the brain. For sufficient precission we have to divide the head into hundred thousands volumic elements and to compute electromagnetic field in the elements. It is hard task which could not run on single PC, thus we implement a computer program which can run on a cluster of co-operating computers. Our sofware is based on message passing method realized by MPI.

Poděkování

Děkuji vedoucímu diplomové práce panu Dr. Ing. Janu Kybicovi za jeho ochotu a vstřícný přístup. Děkuji svým rodičům za to, že mi umožnili studovat vysokou školu a získat tak mnoho cenných zkušeností. Děkuji také svým blízkým za trpělivost a radost, kterou mi přináší.

Obsah

1 Úvod

Cílem této diplomové práce je vyvinout paralelní verzi již existujícího numerického řešení MEG/EEG dopředné úlohy symetrickou metodou hraničních prvků naprogramovaného Dr. Ing. Janem Kybicem v programovém balíku [1], pro který budu v následujícím textu používat zkratku OBEM. Paralelní verze (dále PBEM) bude navržena pro heterogenní svazek počítačů (výpočetní cluster) a testována na svazku *cmpgrid* nalézajícím se v Centru strojového vnímání[1].

V tomto textu se po krátkém úvodu popisujícím stav poznání v oblasti a motivaci vytvoření paralelní verze aplikace dočtete v kapitole 2 o matematickém řešení dopředné úlohy. V kapitole 3 provedeme návrh paralelizace programu OBEM. Následující kapitola 4 pojednává o nastavení běhového prostředí (runtime environment) na *cmpgrid* a v kapitole 5 popíšeme experimenty provedené s hotovým programem. V přílohách pak naleznete seznam použité literatury, formát vstupních souborů aplikace a obsah přiloženého CD s programem a dokumentací.

1.1 Motivace

V lidském mozku se vzruchy (informace) šíří v podobě elektrických signálů nervovými vlákny [21]. Tyto signály můžeme na povrchu hlavy snímat elektroencefalografem (EEG) jako změny elektrického napětí, případně magnetoencefalografem (MEG) jako změny intenzity magnetického pole [22]. Pro pochopení činnosti mozku a pro studium různých neurologických onemocnění by bylo výhodné znát i rozmístění zdrojů proudu uvnitř mozkové tkáně.

Zjištění polohy proudových zdrojů z dat získaných EEG nebo MEG se nazývá *inverzní (zpětná) úloha*. Pro řešení inverzní úlohy potřebujeme získat model hlavy pro *dopřednou úlohu (DU)*, umožňující spočítat intenzitu pole na povrchu hlavy při známé konfiguraci zdrojů uvnitř mozku.

1 Centrum strojového vnímání, katedra kybernetiky, ČVUT FEL, <http://cmp.felk.cvut.cz>

Získání přesného modelu vyžaduje jemnou diskretizaci objemu hlavy. Při této diskretizaci je kvůli požadované přesnosti nutné hlavu rozdělit na desítky až stovky tisíc objemových elementů a počítat v nich konfiguraci elektromagnetického pole. To je výpočetně časově i paměťově velmi náročná úloha, proto tato diplomová práce popisuje její řešení na svazku počítačů. To umožnuje výpočet urychlit a díky společné paměti spojených počítačů vůbec umožnit.

1.2 Stav poznání

1.2.1 Matematické metody výpočtu dopředné úlohy

K numerickému řešení dopředné úlohy se využívá zejména *metoda konečných prvků (Finite element method – FEM)* a *metoda hraničních prvků (Boundary element method – BEM)* [23].

BEM redukuje prostorový problém na problém plošný, neboť k výpočtu využívá pouze body na rozhraní objemů s různými elektrickými parametry. Na druhou stranu, množství počítaných interakcí mezi prvky je u BEM větší než u FEM, což snižuje rychlost výpočtu. Výpočet metodou FEM je rychlejší, než při použití BEM, ale vyžaduje trojrozměrný model hlavy (3D mesh), jehož vytvoření je výpočetně náročná úloha. Ukazuje se tak, že máme-li již vytvořený trojrozměrný model hlavy, je k výpočtu dopředné úlohy výhodnější použít FEM. Pokud trojrozměrnou mřížku nemáme, je DU výhodnější řešit metodou BEM. Více o srovnání použití FEM a BEM při řešení DU viz [2].

Při řešení elektromagnetického pole metodou BEM existují tři přístupy k sestavení maticové rovnice. Jde o tzv. potenciál jednoduché vrstvy, potenciál dvojité vrstvy a symetrický potenciál [5]. OBEM i PBEM jsou založeny na metodě symetrického potenciálu – ten kombinuje vlastnosti prvních dvou přístupů a je oproti nim přesnější. Navíc metoda symetrického potenciálů vede k efektivnějšímu algoritmu [5].

1.2.2 Paralelizace výpočtu dopředné úlohy a paralelizace BEM

Co se týká paralelizace EEG/MEG dopředné úlohy, paralelizován byl výpočet metodou FEM [3, 4]. Pokud je mi známo, výpočet pomocí metody BEM zatím paralelizován nebyl.

Nicméně paralelizace metody BEM aplikované na jiné problémy provedena byla. Některé práce se zabývají urychlením běhu iterační metody, a to buď paralelním

výpočtem vstupní podmínky [24, 25], nebo vlnkovou transformací matice systému [26]. V našem případě je však nejnáročnější sestavení matice systému A (viz sekce 3.2), iterační metodu paralelizovat nepotřebujeme.

Jiné práce se zabývají pouze dvourozměrnými (plošnými) problémy [27].

Zajímavými pro náš program by mohly být práce zabývající se algoritmy vyvažování zátěže (load balancing) při rozdělování matice soustavy [28, 29].

2 Řešení dopředné úlohy

V této kapitole se seznámíme s analytickým popisem elektromagnetického pole v mozku a s numerickým řešením tohoto pole metodou hraničních prvků. Většina této kapitoly čerpá z [5].

Připomeňme, že dopředná úloha je výpočet elektrického potenciálu V na povrchu hlavy ze známé konfigurace elektrických zdrojů $\mathbf{J}^p$ uvnitř hlavy. Předpokládáme, že elektrické vlastnosti tkáně známe.

Tento výpočet může být použit i pro magnetoencefalografii (MEG), protože intenzita magnetického pole $\mathbf{B}$ může být spočítána z potenciálu V jednoduchou integrací [5].

2.1 Poissonova rovnice

Dopředná úloha je vyjádřena tzv. Poissonovou rovnicí:

$$\nabla \cdot (\sigma \nabla V) = f = \nabla \cdot \mathbf{J}^p \qquad (1),$$

kde σ [$(\Omega.\mathrm{m})^{-1}$] je vodivost a $\mathbf{J}^p$ [$\mathrm{A.m^{-2}}$] je proudová hustota – obě tyto veličiny jsou známé. Elektrický potenciál V [V] je neznámá veličina.

2.2 Proudové zdroje v mozku

Nervové vzruchy se v lidském těle šíří po vláknech (dendritech a axonech) neuronů jako změny potenciálu na buněčné membráně (povrchu) těchto vláken. Změna potenciálu je způsobena změnou propustnosti membrány pro ionty nacházející se uvnitř a vně buňky a jejich následnou výměnou mezi buňkou a okolím. Pokud se v jednom místě buněčné membrány změní např. působením neurotransmiteru [21] potenciál, vzniká napěťový gradient mezi tímto místem a místy v okolí. Tento gradient způsobuje přemisťování iontů, na které můžeme nahlížet jako na elektrický proud. [21]

Tento elementární zdroj proudu můžeme popsat jako tzv. proudový dipól [5]. Proudový dipól reprezentuje nekonečně malý orientovaný zdroj proudu v místě $\mathbf{r}_0$ s dipolárním momentem $\mathbf{q}$, a je definován jako

$$\mathbf{J}_{\text{dip}}(\mathbf{r}) = \mathbf{q}\,\delta_{\mathbf{r}_0}(\mathbf{r})$$

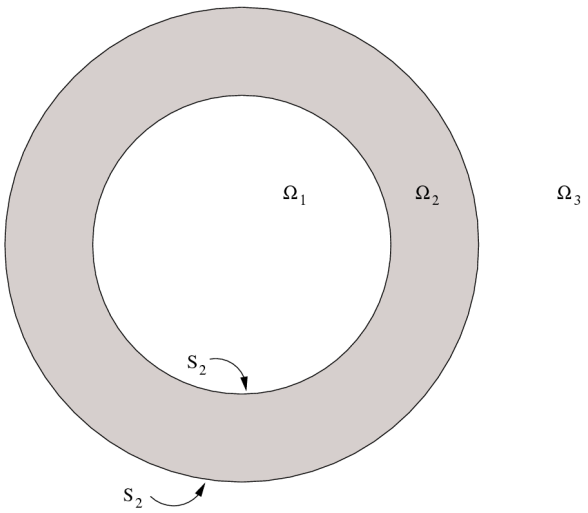
Odpovídající člen v Poissonově rovnici je

$$f_{\text{dip}} = \nabla\cdot\mathbf{J}_{\text{dip}} = \mathbf{q}\cdot\nabla\,\delta_{\mathbf{r}_0}(\mathbf{r})$$

Dipolární moment $\mathbf{q}$ definujme jako vektor $\mathbf{q}$ = Q$\mathbf{d}$, kde vektor $\mathbf{d}$ udává vzájemnou vzdálenost a orientaci dvou nábojů o velikost Q a -Q.

2.3 Metoda BEM

Mějme do sebe vnořené oblasti Ω_1, Ω_2, Ω_3 a jejich vzájemné hranice $S_1 = \partial\Omega_1 \cap \partial\Omega_2$; $S_2 = \partial\Omega_2 \cap \partial\Omega_3$, kde $\partial\Omega_1$, $\partial\Omega_2$ a $\partial\Omega_3$ jsou hranice oblastí Ω_1, Ω_2 a Ω_3 (viz obr. 1).



obr. 1.: značení oblastí a hranic oblastí

Metoda hraničních prvků je založena na teorému [5], který říká, že každá harmonická funkce u splňující jisté počáteční podmínky H je kdekoliv definována svým skokem a skokem své derivace přes hranici oblasti $\partial\Omega$, ať už je $\partial\Omega$ hranice s jedním nebo dvěma povrchy.

Harmonická funkce je taková funkce u, pro kterou platí $\triangle u = 0$, kde $\triangle$ je Laplaceův operátor. Funkce u splňuje podmínku H pokud zároveň

$$\lim_{r\to\infty} r\,|u(\boldsymbol{r})| < \infty$$
$$\lim_{r\to\infty} r\,\frac{\partial u}{\partial r}(\boldsymbol{r}) = 0$$

kde $r = \|\boldsymbol{r}\|$ a $\frac{\partial u}{\partial r}(\boldsymbol{r})$ značí parciální derivaci u v radiálním směru. Tato podmínka odpovídá fyzikální představě, že statické pole je ve velké vzdálenosti od nábojů nulové.

Existuje více typů metody BEM. V tomto textu se zaměřím na symetrickou metodu BEM která je použita v implementaci našeho programu.

Symetrický BEM systém má tento tvar:

$$\underbrace{\begin{pmatrix} \mathrm{N} & \mathrm{D}^* \\ \mathrm{D} & \mathrm{S} \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix}}_{\mathbf{u}} = \underbrace{\begin{pmatrix} \mathbf{w} \\ \mathbf{z} \end{pmatrix}}_{\mathbf{c}} \qquad (2),$$

kde neznámé $\mathbf{x}$ a $\mathbf{y}$ představují diskrétní verzi elektrického potenciálu V a proudové hustoty $p = \sigma\,\partial_{\mathbf{n}} V$; pravá strana s proměnnými $\mathbf{w}$ a $\mathbf{z}$ představuje známá pole (potenciál a proudovou hustotu) korespondující se zdroji. Matice N, D, D^* a S jsou diskrétní verze následujících spojitých operátorů (všechny tyto operátory zobrazují skalární funkci f definovanou na rozhraní $\partial\Omega$ na jinou skalární funkci). Pro $\mathbf{r}$ ležící na $\partial\Omega$ nechť:

$$\begin{aligned}
(\mathcal{N}f)(\mathbf{r}) &= \int_{\partial\Omega} \partial_{\mathbf{n},\mathbf{n}'} G(\mathbf{r}-\mathbf{r}')\, f(\mathbf{r}')\,\mathrm{d}s(\mathbf{r}') \\
(\mathcal{D}f)(\mathbf{r}) &= \int_{\partial\Omega} \partial_{\mathbf{n}'} G(\mathbf{r}-\mathbf{r}')\, f(\mathbf{r}')\,\mathrm{d}s(\mathbf{r}') \\
(\mathcal{D}^* f)(\mathbf{r}) &= \int_{\partial\Omega} \partial_{\mathbf{n}} G(\mathbf{r}-\mathbf{r}')\, f(\mathbf{r}')\,\mathrm{d}s(\mathbf{r}') \\
(\mathcal{S}f)(\mathbf{r}) &= \int_{\partial\Omega} G(\mathbf{r}-\mathbf{r}')\, f(\mathbf{r}')\,\mathrm{d}s(\mathbf{r}')
\end{aligned} \qquad (3),$$

kde $G(\mathbf{r}) = 1/(4\pi\|\mathbf{r}\|)$ a kde $\partial_{\mathbf{n}}$, $\partial_{\mathbf{n}'}$, značí parciální derivace ve směru normály povrchu v bodech $\mathbf{r}$, $\mathbf{r}'$.

Diskrétní verze operátorů získáme takto:

$$\begin{aligned}
(\mathrm{N})_{ik} &= \langle \mathcal{N}\varphi_k, \varphi_i \rangle, \qquad (\mathrm{S})_{jl} = \langle \mathcal{S}\varphi_l, \varphi_j \rangle \\
(\mathrm{D})_{jk} &= (\mathrm{D}^*)_{kj} = \langle \mathcal{D}\varphi_k, \psi_j \rangle = \langle \mathcal{D}^* \psi_j, \varphi_k \rangle,
\end{aligned} \qquad (4)$$

kde ψ resp. φ jsou bázové funkce P0 (po částech konstantní na každém trojúhelníku) resp. P1 (po částech lineární na každém trojúhelníku). Operátorem $\langle f, b\rangle$ značíme skalární součin $\langle f, g\rangle = \int_{\partial\Omega} f(\mathbf{r})\, g(\mathbf{r})\, \mathrm{d}s(\mathbf{r}')$. Vnitřní integrál ve výrazech (4) lze spočítat analyticky. Vnější integrál (skalární součin) však musíme počítat numericky, což zvyšuje časovou složitost výpočtu (viz kapitola 3). K numerické integraci využijeme Gaussovo kvadraturu adaptovánu na trojúhelníky [5].

Velkou výhodou je, že matice $(\mathrm{N})_{kl}$ může být levně vypočítána z hodnot potřebných k výpočtu matice $(\mathrm{S})_{kl}$. Platí totiž [5]:

$$\langle \mathcal{K}_{kl}\varphi'_i, \varphi'_j\rangle = -(\mathbf{q}_i \times \mathbf{n}_i)(\mathbf{q}_j \times \mathbf{n}_j)\langle \mathcal{S}_{kl}\psi_i^{(l)}, \psi_j^{(k)}\rangle \tag{5},$$

kde

$$\varphi'_i(\mathbf{x}) = (\mathbf{q}_i \cdot \mathbf{x} + \alpha_i)\,\psi_i(\mathbf{x})$$
$$\varphi'_j(\mathbf{x}) = (\mathbf{q}_j \cdot \mathbf{x} + \alpha_j)\,\psi_j(\mathbf{x})$$

jsou P1 bázové funkce φ vztažené k trojúhelníku T_i.

Jako příklad rovnice soustavy uvádím diskretizovaný systém pro vhnízděný třívrstvý model:

$$\underbrace{\begin{pmatrix} (\sigma_1+\sigma_2)\mathrm{N}_{11} & -\sigma_2\mathrm{N}_{12} & 0 & -2\mathrm{D}_{11}^{*} & \mathrm{D}_{12}^{*} \\ (-\sigma_2)\mathrm{N}_{21} & (\sigma_2+\sigma_3)\mathrm{N}_{22} & -\sigma_3\mathrm{N}_{23} & \mathrm{D}_{21}^{*} & -2\mathrm{D}_{22}^{*} \\ 0 & -\sigma_3\mathrm{N}_{32} & \sigma_3\mathrm{N}_{33} & 0 & \mathrm{D}_{32}^{*} \\ -2\mathrm{D}_{11} & \mathrm{D}_{12} & 0 & (\sigma_1^{-1}+\sigma_2^{-1})\mathrm{S}_{11} & -\sigma_2^{-1}\mathrm{S}_{12} \\ \mathrm{D}_{21} & -2\mathrm{D}_{22} & \mathrm{D}_{23} & -\sigma_2^{-1}\mathrm{S}_{21} & (\sigma_1^{-1}+\sigma_2^{-1})\mathrm{S}_{22} \end{pmatrix}}_{\mathrm{A}} \cdot \underbrace{\begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \mathbf{x}_3 \\ \mathbf{y}_1 \\ \mathbf{y}_2 \end{pmatrix}}_{\mathbf{u}} = \underbrace{\begin{pmatrix} \mathbf{w}_1 \\ \mathbf{w}_2 \\ \mathbf{w}_3 \\ \mathbf{z}_1 \\ \mathbf{z}_2 \end{pmatrix}}_{\mathbf{c}} \tag{6},$$

kde jednotlivé plochy značíme 1, 2, 3, indexy u operátorů znamenají cíle a zdroje (např. N_{23} vztahuje k potenciálu na povrchu 3 prostorový potenciál na povrchu 2) a σ_a jsou vodivosti v oblastech, číslovány od nejvnitřnější.

2.4 Řešení maticové soustavy metodou MINRES

V přecházejí sekci jsme zjistili, že metoda BEM vede k sestavení maticové rovnice (2), resp. (6). K vyřešení této rovnice je ve stávajícím programu [1] použita metoda

minimálních reziduí (MINRES). V paralelní verzi budeme rovnici řešit také touto metodou, proto je zde uvedena její stručná charakteristika.

Pro jednoduché vysvětlení předpokládejme, že řešíme obecnou maticovou rovnici $A\mathbf{x} = \mathbf{b}$, kde A je tzv. matice soustavy, $\mathbf{x}$ je vektor neznámých a $\mathbf{b}$ vektor pravé strany.

Metoda minimálních reziduí [9] je iterační metoda, proto musíme v prvním kroku stanovit odhad řešení x_0 (řešíme soustavu $Ax = b$) a dále postupně hledáme nové aproximace řešení na základě aproximací předchozích. Iterační metody lze přeřušit, pokud se během iterace dostaneme na požadovanou hodnotu přesnosti.

Metoda MINRES patří mezi tzv. Krylovovské metody, kdy aproximace řešení soustavy hledáme v afinním prostoru

$$\mathbf{x}_m \in \mathbf{x}_0 + \mathcal{K}_m(\mathrm{A}, \mathbf{r}_0)$$

kde $\mathcal{K}_m(\mathrm{A}, \mathbf{r}_0) = \mathrm{span}\{\mathbf{r}_0, \mathrm{A}\,\mathbf{r}_0, \ldots, \mathrm{A}^{m-1}\mathbf{r}_0\}$ je m-tý Krylovův podprostor a span{A} je lineární obal množiny A.

Metoda MINRES využívá Lanczosův algoritmus k vytvoření ortogonální báze $\mathrm{V}_m = [\mathbf{v}_1, \ldots, \mathbf{v}_m]$ prostoru $\mathcal{K}_k(\mathrm{A}, \mathbf{r}_0)$. Pak můžeme psát

$$\mathbf{x}_m = \mathbf{x}_0 + \mathrm{V}_m \mathbf{y}_m$$

kde $\mathbf{y}_m \in \mathbb{C}^m$ je vektor koeficientů lineární kombinace vektorů $\mathbf{v}_1, \ldots, \mathbf{v}_m$; $\mathbf{x}_m$ je m-té řešení soustavy (10) a $\mathbf{x}_0$ odhad prvního řešení této soustavy. Celý algoritmus MINRES je zde:

```
1  ϵ je tolerance, do které se má vejít norma rezidua
2  konvergence = false;
3  vyber x_0;
4  until konvergence do
5     r_0 = b − A x_0;
6     β = ||r_0||
7     /* Lanczosův proces: konstrukce báze V_m Krylovova podprostoru K */
8     β_1 v_1 = b     /* tzn. že β_1 ← ||b|| a pak v_1 ← b/β_1, ale konec když β_1 = 0 */
9     for j = 1, ..., m
10       w = A v_j;
11       α_1 = v_j^T w;
12       β_{j+1} v_{j+1} = w − α_j v_j − β_j v_{k−1};
13       h_jj = α_j;
14       h_{j+1,j} = h_{j,j+1} = 1;
15    endfor
16    /* minimalizace normy ||b − A x|| pro x ∈ x_0 + K */
17    /* řešení problému min_{y_m ∈ R^m} ||β e_1 − H̄_m y_m|| pro y_m metodou
                                                    nejmenších čtverců */
18    x_m = x_0 + V_m y_m;
19    if (||b − A x_m|| < ϵ) konvergence = true ;
20    x_0 = x_m;
21 enddo
```

Algo 1: MINRES

Všimněme si, že v algoritmu vystupuje pouze součin A$\mathbf{x}$, samotnou matici A nepotřebujeme. Toho z výhodou využijeme při paralelizaci metody, kdy A bude rozdělena na submatice v pamětech jednotlivých počítačů tvořících cluster. Tyto počítače pak budou spolupracovat na výpočtu součinu A$\mathbf{x}$ a tento součin předají jedinému počítači, na kterém proběhne iterace MINRES.

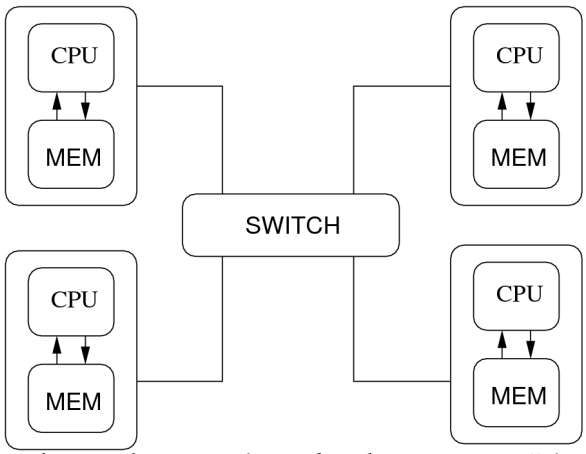
3 Návrh paralelizace

V této kapitole se nejprve zmíním o možných přístupech k paralelizaci programu na heterogenním clusteru. Potom provedu analýzu běhu aplikace OBEM a na základě této analýzy navrhnu metodu paralelizace aplikace.

3.1 Základní pojmy

3.1.1 Systémy s distribuovanou pamětí

Systémem s distribuovanou pamětí rozumíme takové uspořádání vzájemně propojených procesorů, kdy každý procesor má svou vlastní paměť, do které může přistupovat pouze on sám. Schema systému můžete vidět na obr. 2.



obr. 2.: Schema systému s distribuovanou pamětí

Chce-li nějaký procesor na clusteru číst nebo měnit data v paměti jiného procesoru, musí o to vlastníka paměti požádat a do paměti přistupovat zprostředkovaně za asistence vlastníka.

Každý počítač je v takovémto uspořádání do značné míry autonomní, proto se tomuto systému říká také *volně vázaný systém*.

3.1.2 Heterogenní cluster

Výpočetním clusterem nazýváme seskupení volně vázaných počítačů, které spolu úzce spolupracují, takže se navenek mohou projevovat jako jeden výkonný počítač. Cluster nazýváme heterogenní, pokud mají jednotlivé počítače různé výpočetní parametry (rychlost procesoru, velikost paměti). *Cmpgrid* je heterogenní cluster.

Cmpgrid však přesně nesplňuje definici systému s distribuovanou pamětí, neboť některé z jeho počítačů mají dva procesory, které sdílejí společnou paměť. Pro naše účely však budu celý cluster považovat za systém s distribuovanou pamětí a správu „minisystémů se sdílenou pamětí“ přenechám knihovně MPI (viz níže).

3.1.3 Model předávání zpráv

Jak jsme se již zmínili v sekci 4.3.1, u volně vázaných systémů nemají ostatní procesy přímý přístup do paměti patřící jednomu procesoru. Pokud tedy chtějí znát nějaká data z paměti jiného procesoru, musí o tato data vlastníka paměti požádat. Přijme-li vlastník požadavek, data z paměti sám přečte a následně je pošle žadateli. K tomu, aby jednotlivé procesory spolupracovaly, tedy stačí vytvořit mezi nimi komunikační kanál a nemusíme už např. řešit kolize přístupů do paměti více procesory zároveň. Komunikačním kanálem může být např. počítačová síť, proto lze tento model realizovat téměř na všech platformách.

Programové vybavení, které realizuje model předávání zpráv, nazýváme systém předávání zpráv. Tento systém musí řešit např. jak se program k systému předávání zpráv připojí, aby mohl využívat jeho funkce. Musí být jasný mechanismus adresace (komu zprávu doručit) a je třeba zaručit, aby zpráva dorazila v pořádku, případně aby se odesílatel dozvěděl, že zpráva nedorazila [17].

Nejrozšířenějšími volně šiřitelnými systémy pro předávání zpráv jsou PVM [18] a MPI [14]. MPI bude věnován následující sekce.

3.1.4 MPI – Message Passing Interface

MPI je standard, který říká, jakým způsobem má být realizován systém předávání zpráv. Vývoj na tomto standardu začal v roce 1992 a v současnosti dospěl do verze 2. Nejdůležitějšími cíli projektu MPI je umožnění přenositelnosti paralelních aplikací na

úrovni zdrojového kódu a vytvoření předpokladů pro efektivní implementaci modelu předávání zpráv. Mezi další cíle patří podpora heterogenních struktur a architektur, sémantika nezávislá na programovacím jazyku, jazykové rozhraní pro C/C++ a Fortran a blízkost již existujícím nástrojům [17].

MPI je zástupcem *explicitní paralelizace*, kdy je plně na programátorovi, aby detekoval paralelismus v algoritmu a implementoval ho prostřednictvím konstruktů MPI v jeho kódu. Nic v tomto směru se nedělá automaticky, nic např. nezařídí překladač.

MPI jako takový je pouze standard. Systémy implemetující MPI jsou různé, z nejznámějších jmenujme MPICH [19] a LAM [15]. Jednotlivé implemetace se můžou lišit např. v tom, jakým způsobem se spouští programy nebo jak probíhají paralelní vstupně výstupní operace.

3.1.5 Metoda master – slave

Nejvýhodnější se pro paralelizaci OBEM na *cmpgrid* jeví metoda *master - slave*, kdy pán (master) rozdělí úlohy mezi sluhy (slaves), kteří je pak řeší a dílčí výsledky vrací pánovi, který je zkompletuje. Pán řídí celý výpočet a pokud sluhové něco potřebují, obrátí se na pána. Typicky sluhové nekomunikují mezi sebou. Tento přístup je výhodný pokud je díky topologii propojovací sítě cena komunikace mezi všemi uzly je stejná a pokud je systém heterogenní. V případě jiných systémů může být výhodnější použít jinou metodu řízení paralelního výpočtu. Více se o této problematice můžete dočíst např. ve [20].

Pro oddělení operací, které provádí pán nebo sluha se používá následující jednoduchá konstrukce (algoritmus v pseudokódu):

```
if master then
{
   kód mastera...
}

else
{
   kód slave...
}
end
```

Algo 2: Rozdělení kódu na master a slave část

Jakým způsobem proces pozná, zda byl puštěn jako sluha nebo jako pán řeší systém pro zasílání zpráv, v našem případě standard MPI.

3.2 Analýza problému

Po prostudování potřebné teorie se může věnovat vlastnímu návrh paralelizace. Již víme, že nejvýhodnější bude použít metodu master – slave, nicméně stále nevíme, jaké operace zabírají nejvíce času a paměti. Provedeme proto analýzu časové a paměťové složitosti aplikace OBEM.

3.2.1 Základní algoritmus OBEM

Program OBEM je přímočarý a algoritmus jeho běhu bychom mohli popsat takto:

```
1  start OBEM
2      /* načtení dat ze vstupních souborů */
3    m = readDataFromMeshFile meshfile;
4    (xi, zeta) = readDataFromDipoleFile dipfile;
5
6      /* vytvoření párů všech trojúhelníků */
7    s = makeTrianglePairsStream m;
8      /* výpočet operátorů N, S a  D* na všech trojúhel.*/
9    allaelements = computeAllOperatorsContribs s;
10     /* vložení vypočítaných prvků do matice A */
11   a = createMatrix allelements;
12
13     /* definice funkce, která bude počítat součin A*x  */
14   fa x = computeMatrixVectorProduct a x;
15     /* vytvoření vektoru pravé strany  */
16   rhs = createRhs xi zeta m;
17
18     /* výpočet řešení soustavy metodou MINRES */
19   y = minres (fa x) rhs ;
20
21     /* uložení výsledku */
22   saveResults y filename;
23 end
```

Algo 3: OBEM - základní algoritmus

3.2.2 Časová složitost

Nejnáročnější operací je numerická integrace všech prvků při výpočtu operátorů S, D^* a N, viz (4), kapitola 2.3, protože je volána velmi často. Numerický integrál počítáme Gaussovou kvadraturou [5]. Profilací kódu (viz níže) bylo zjištěno, že jedna integrace trvá

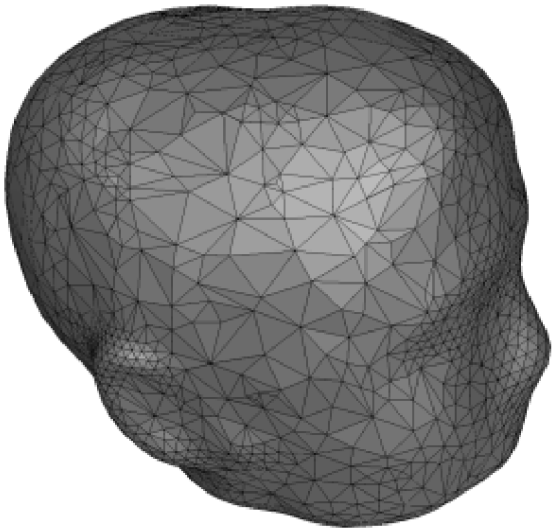
průměrně 0,0001 s. Počet volání integrace spočítáme pro třívrstvý model, tzn. přesně pro případ, který bude řešit PBEM. Existenci vrstev s nulovou vodivostí zanedbáme.

Při výpočtu operátoru S integrujeme každý trojúhelník s každým (do vnějšího i vnitřního integrálu vstupuje bázová funkce P0, viz (4) v 2.3), provedeme tedy $n_T^2/2$ integrací (n_T je počet všech trojúhelníků). Hodnotu dělíme dvěma kvůli symetrii matice. Při výpočtu operátoru D^* je integrací $3n_T^2/2$, neboť každý trojúhelník integrujeme s každými třemi vrcholy všech trojúhelníků (do vnitřního integrálu vstupuje P0, do vnějšího P1). Při výpočtu operátoru N vzájemně integrujeme všechny tři vrcholy každého trojúhelníku se třemi vrcholy všech trojúhelníků (oba integrály přes P1), takže počet operací je $3^2n_T^2/2$. Celkový počet integrací je tedy úměrný $n_T^2/2 + 3n_T^2/2 + 9n_T^2/2 = 13/2\ n_T^2 \sim n_T^2$. Je třeba poznamenat, že operátor N lze levně vypočíst z S dle (5). Při paralelizaci bohužel tuto výhodu částečně ztrácíme – více se o tomto problému zmíníme v sekcích 3.3.3 a 3.3.4.

Experimentálně jsem časovou složitost OBEM zjišťoval profilací kódu programem `ocamlprof` (viz příslušná manuálová stránka). OBEM během svého běhu shromažďuje informace pro profiler pokud je přeložen příkazem `make pc`.

Při profilaci musíme mít na zřeteli, že v aplikaci jsou používány proudy (streams). To jsou tzv. líné (lazy) datové struktury, které se vyhodnocují až v případě, kdy přistupujeme k hodnotám v nich uložených. To znamená, že pokud operace A počítá prvky matice a zároveň je ukládá do proudu, zatímco operace B prvky z proudu čte a vkládá na patřičné místo v matici, prvky se fakticky začnou počítat až při zahájení operace B. Jinými slovy, A si přímo střídá výpočetní čas s B – nejprve A spočítá jeden prvek a hned potom ho B vloží do matice. Pak A počítá další prvek a B ho zase uloží. Pokud je výpočet prvku (tedy operace A) mnohem náročnější, než jeho vložení do matice, profiler toto neodhalí a veškerý spotřebovaný čas přiřadí operaci B. Výhodou proudů je šetření paměti – každá hodnota se vypočítá až v době, kdy jí potřebujeme a pokud ji po použití potřebovat přestaneme, je okamžitě uvolněna.

Profilací kódu bylo zjištěno, že výpočet operátorů N, S a D^* trvá 82 % doby běhu programu, zatímco iterace MINRES pouze 0,01 %. Zbytek času připadá na různé pomocné funkce, poměrně časově náročný je např. garbage collector.



obr. 3.: Trojúhelníkový (2D mesh) model hlavy. Převzato z [5]

3.2.3 Paměťová složitost

Paměťově nejnáročnější je uchovat v paměti celou matici soustavy. Zatímco proměnné využívané při výpočtu prvků matice se díky garbage collectoru a díky použití proudů uvolňují, matice soustavy je vůbec největší proměnnou (dvourozměrným polem), kterou navíc potřebujeme v paměti udržet celou až do skončení výpočtu.

Velikosti matice soustavy A spočítáme podle vzorce $V = v*(n_P + n_T - n_{T\_out})^2/2$, kde V je velikost matice A, v je počet bytů alokovaných pro jedno políčko matice (8 B – typ float), n_P je počet bodů, n_T počet trojúhelníků a n_{T_out} počet trojúhelníků na hranici oblasti s nulovou vodivostí. Celý vzorec je vydělen dvěma, protože matice je symetrická a stačí tak uložit pouze její polovinu (zbytek hodnot získáme záměnou indexů). Protože $n_T \sim 2n_P$, paměťová náročnost je úměrná n_T^2.

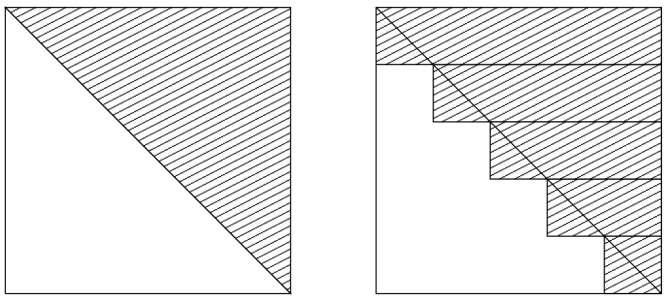
Velikost matice pro různá vstupní data můžete vidět v tab.1.

vstupní soubor	počet bodů	počet trojúhelníků	velikost A
head.1.md	126	240	327 kB
head.2.md	486	960	5 MB
head.3.md	1 926	3 840	80 MB
head.4.md	7 686	15 360	1,3 GB
head.5.md	30 726	61 440	20 GB
head.6.md	122 886	245 760	330 GB

tab.1: velikost matice soustavy v závislosti na vstupních datech

K představě, jak vypadají body a trojúhelníky v tab.1 může posloužit obr. 3. Soubory head.[1-6].md, ve kterých jsou vstupní data nalezena, naleznete na přiloženém CD (viz Příloha A) a B)).

Ve skutečnosti je kvůli implementaci velikost matice vyšší než vypočítaná hodnota, neboť trojúhelníkovou matici alokuji po blocích, viz obr. 4. Alokace po blocích je sice paměťově náročnější, než uložení čistě trojúhelníkové matice, ale je implementačně jednodušší a také umožňuje v další výpočtech jednoduché použití efektivních algoritmů pro výpočet součinu matice s vektorem (viz sekce 3.3.3).



obr. 4.: Alokace trojúhelníkové matice (šrafovaná plocha je alokovaná). a) ideální případ, b) zjednodušené rozdělení matice na obdélníkové bloky

Porovnáme-li paměťové nároky výpočtu s parametry dnešních počítačů, ve kterých je velikost operační paměti rovna řádově jednotkám gigabytů, zjistíme, že bychom na současném PC mohli spočítat úlohu o velikosti přibližně patnácti tisíc trojúhelníků.

3.3 Vlastní návrh paralelizace

3.3.1 Rozdělení výpočtu

Na základě analýzy v kapitole 3.2 jsem se rozhodl pro paralelizaci výpočtu prvků matice systému, neboť tento výpočet zabírá nejvíce času z doby běhu programu. Navíc paměťově nejnáročnější je právě matice A, proto paralelizace bude probíhat následovně:

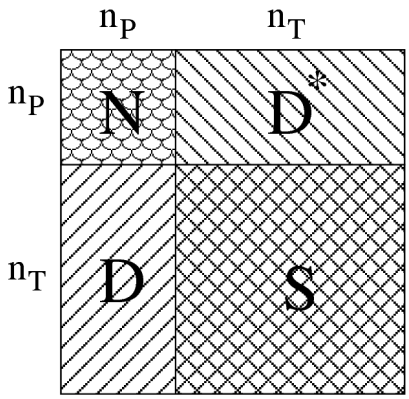
1) Každý sluha získá od pána přidělenou část vstupních dat (část trojúhelníků) a z této části vypočítá příslušné prvky matice A.

2) Sluha vypočítané prvky uloží do matice, kterou bude držet v paměti celou dobu běhu programu. Tato matice bude submaticí matice A.

3) Řešení soustavy metodou MINRES bude probíhat pouze na pánovi. Pro výpočet každé iterace MINRES je potřeba znát součin Ax_i. Na výpočtu tohoto součinu budou s pánem spolupracovat sluhové tak, aby matice A mohla být rozdělena na submatice a držena v paměti sluhů celou dobu běhu programu.

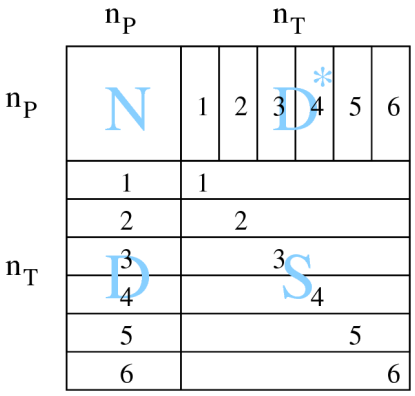
3.3.2 Výpočet submatic D* a S

Matice A se skládá ze čtyř hlavních bloků, viz rovnice (2) v kapitole 2.3. Graficky toto rozdělení vidíme na obr. 6:



obr. 5.: Struktura matice soustavy

Z obr. 5 vyplývá i velikost jednotlivých bloků a způsob indexace prvků v matici. Blok N má rozměry $n_P \times n_P$, bloky D $n_P \times n_T$ a blok S $n_T \times$ *počet_trojúhelníků*. Mezi sluhy rozděluji trojúhelníky tak, aby submatice D, D^* a S mohly být rozděleny tak jak je znázorněno na obr. 6. Rozdělení submatice N je komplikovanější a bude popsáno v další sekci.



obr. 6.: Příklad rozdělení submatic D, D^* a S mezi sluhy. Číslice identifikují jednotlivé sluhy

Sluha tak dostane n trojúhelníků a spočítá a drží v paměti příslušnou submatici $n \times N_t$ (pro operátor S), resp. $n \times N_p$ (pro operátor D). N_t je počet všech trojúhelníků a N_p počet všech bodů. Z uvedeného vyplývá, že každý sluha potřebuje znát všechny trojúhelníky – to nepředstavuje žádný problém, neboť pole s údaji o všech trojúhelnících je v porovnání s maticí zanedbatelně malé. Sluha si toto pole načte sám ze vstupního souboru.

Nyní bych se zastavil u indexace prvků v matici. K dalšímu výkladu je důležité vědět, že funkce pro výpočet prvků matice mají pro všechny operátory stejný vstupní parametr – jde vždy o dvojici trojúhelníků, jejichž vzájemná interakce se počítá. Výstupní hodnoty těchto funkcí jsou streamy (proudy) prvků typu (i, j, x), kde i a j jsou identifikační čísla elementů (trojúhelníků nebo bodů), jejichž vzájemná interakce byla spočítána. Prvek x je číselná hodnota této interakce. Hodnoty i a j jsou zároveň souřadnicemi, udávajícími polohu prvku x v matici soustavy A.

Nejjednodušší situace nastává pro operátor S. Pokud ho aplikujeme na dvojici trojúhelníků s identifikačními čísly a, b, získáme hodnotu, která je v matici uložena právě na pozici a, b. Poněkud komplikovanější situace nastává pro operátor D. Souřadnice prvku spočítaného tímto operátorem závisí nejen na vstupním trojúhelníku, ale také na vrcholu trojúhelníku (bodě). Situaci jsem vyřešil tak, že bloky mají rozměr $n \times N_p$ (viz výše), hodnoty se tak počítají přes všechny trojúhelníky a tedy i přes všechny body. Bloky proto nemusíme dělit podle souřadnice závislé na bodech.

Problémem je submatice N, neboť v ní se obě souřadnice indexují body. Problém si zaslouží být popsán v samostatné sekci.

3.3.3 Výpočet submatice N

Jak jsme se již zmínili, souřadnice hodnot v submatici N jsou identifikační čísla vrcholů trojúhelníků. Naproti tomu bloky dat B_T přidělované sluhům jsou rozděleny podle identifikačních čísel trojúhelníků. To nám při osazení matice N způsobuje problémy, neboť nevíme, z jakých vrcholů se trojúhelníky v B_T skládají a může se tak stát, že vypočítaná interakce bude spadat do jiné submatice, než kterou daný sluha osazuje. Pokusme se to ještě lépe vysvětlit na příkladu. Uvažujme případ, kdy sluha A získá přidělený interval trojúhelníků I_T, přičemž trojúhelník T_1 ležící v tomto intervalu se skládá z bodů (a_1 b_1 c_1) a trojúhelník T_2 také z I_T se skládá z bodů (a_2 b_2 c_2). Sluha A zároveň osazuje blok B_P submatice N. Pak se může stát, že některá ze souřadnic a_1 b_1 c_1 a_2 b_2 c_2 nebude patřit do bloku B_P. Existují tři hlavní metody řešení tohoto problému:

1) Každý sluha alokuje celou submatici N. Pak má jistotu, že do ní může umístit jakýkoliv příspěvek. Tento přístup optimalizuje čas výpočtu, nicméně vůbec nešetří paměť, neboť zatímco v ideálním případě budeme mít submatici N alokovanou jednou pro celou aplikaci, v tomto případě bychom jí měli alokovanou tolikrát, kolik máme sluhů.

2) Sluha spočítá všechny příspěvky operátoru N, které se dají spočítat z jemu přidělených trojúhelníku, a příspěvky, které nepatří do jím osazovaného intervalu, rozešle ostatním sluhům. Tento přístup opět optimalizuje čas výpočtu operátoru N, z mého pohledu by však vedl na příliš velké zasílání dat po síti. Každý sluha by totiž v nejhorším případě musel rozeslat příspěvky všem ostatním sluhům, tzn. počet zpráv by byl n_S (n_S - 1), kde n_S je počet sluhů, a sítí by bylo posláno (n_S -1)/ n_S hodnot vypočítaných operátory N, S a D^* – celkový počet těchto hodnot odpovídá $6{,}5n_T^2$, viz sekce 3.2.2, což by při požadované velikosti vstupních hodnot (tab.1) bylo velmi velké množství dat.

3) Kombinace předchozích dvou přístupů. Každý sluha by osazoval tu část matice, v které by se nacházela většina jím spočítaných hodnot, eventuálně by mohl osazovat částí více (jejich výběr by byl proveden opět podle rozdělení jím

spočítaných hodnot). Zbytek hodnot by poslal ostatním, případně pánovi, který by data třídil, shlukoval a přeposílal na ostatní sluhy. Tato metoda by mohla vést na optimální řešení, byla by však implementačně velmi náročná.

4) Hodnoty operátoru N vypočítané sluhou bychom mohli také reprezentovat jako řídkou matici, to je však také implementačně náročná úloha.

5) Budeme se snažit vybrat jen takové trojúhelníky, jejichž vrcholy jsou body, které spadají do submatice N osazované konkrétním slavem. Pokud tedy např. bod A je vrcholem trojúhelníků s identifikačními čísly 4, 10, 12, 16, 35 a 53, proces osazující submatici na intervalu, do kterého spadá bod A, předá všechny tyto trojúhelníky funkci počítající operátor N.

Velkou výhodou metod 1, 2, 3 a 4 je možnost levně vypočítat operátor N z operátoru S (viz sekce 2.3), neboť operátor N počítáme na stejných trojúhelnících jako operátor S. Při použití metody 5 se může stát, že některé trojúhelníky použité pro výpočet operátoru N budou stejné jako trojúhelníky, které jsme již použili při výpočtu S, a operátor N budeme moci vypočítat také levně, není to však pravidlem. O tom, jak maximalizovat šanci, že trojúhelníky vstupující do operátorů S i N budou stejné, se zmíníme ještě v sekci 3.3.4.

Z uvedených možností budeme implementovat metodu 4. Pojďme se na ní blíže podívat.

Uvažujme příklad, kdy máme osadit blok submatice N o velikosti $I \times N_p$ a kde I je nějaký interval po sobě jdoucích bodů ze vstupního souboru. Nechť bod $q \in I$ je jedním z vrcholů trojúhelníků T_1 a T_2, přičemž ostatní vrcholy těchto trojúhelníků neleží na I. Pak pokud na vstupu funkce počítající operátor N bude dvojice trojúhelníků (T_1 a T_2), na výstupu této funkce získáme kombinaci všech hodnot vrcholů T_1 a T_2 a každé této kombinaci bude přiřazena jí odpovídající hodnota integrálu. Počet všech kombinací $k_{total} = t^2 = 3^2 = 9$ a bod q bude obsažen v $k_1 = (t + (t-1)) = (3 + 2) = 5$ kombinacích; t je počet vrcholů trojúhelníku. Zbývající čtyři dvojice do intervalu I nespadají a budou muset být znovu spočítány v procesu osazujícím jinou část submatice N.

V nejhorším případě bychom tak čas výpočtu prvků submatice N prodloužili $k_{total}/k_1 = 9/5$, tj. 1,8 krát. Protože počet prvků submatice N tvoří přibližně jednu devítinu

prvků celé matice A (viz kapitola 3.2.3), celkový výpočet by se prodloužil $\frac{9}{5}\frac{1}{9}+\frac{5}{5}\frac{8}{9}=1{,}09$ krát. Ve skutečnosti se výpočet prodlouží ještě méně, neboť trojúhelníky T_1 a T_2 z našeho případu mohou být tvořeny více body z intervalu I a ztratíme tedy menší část vypočítaných hodnot. Pokud však bude jedna z ploch tvořit hranici oblastí s nulovou vodivostí, sníží se počet trojúhelníků a změní se tak poměr prvků submatice N vůči všem prvkům v matici A. V takovém případě by se čas výpočtu prodloužil, neměl by však přesáhnout 1,2 násobek času ideálního.

Výše popsanou metodu nyní shrňme a pokusme se jí zapsat v pseudokódu (Algo 4).

```
1  if master then
2  begin
3     /* vytvoření tabulky, která každému bodu přiřadí všechny
4        trojúhelníky, v nichž bod tvoří jeden z vrcholů */
5    tab = makePointTriangleTable;
6    distributePTTtoSlaves tab;
7
8     /* rozešle na slaves intervaly na kterých mají osazovat
9        matici A. Informace o těchto intervalech udržuje master
10       v tabulce "intervals" */
11   intervals = distributeIntervalsToSlaves;
12 end
13 else if slave then
14 begin
15   tab = getTabFromMaster;
16   interval = getMyInterval;
17
18     /* následující funkce vyhledá v tabulce "tab" trojúhelníky
19     obsahující body z intervalu a vrátí je ve streamu "st" */
20   st = makeStreamAccordingToInterval interval tab;
21
22     /* z trojúhelníku spočti prvky submatice N */
23   opnc = computeOpNcontribs st;
24
25     /* z vypočtených prvků sestav svou část submatice N */
26   submat_a_int = makeMatrix opnc;
27 end
```

Algo 4: Osazení části submatice N

3.3.4 Distribuce bloků

Distribuce bloků jsme se dotkli již v předchozích sekcích. Věnujme se jí však nyní trochu podrobněji. Jedna z možností je rozdělit všechna data na počet intervalů shodný s počtem sluhů. Na každého sluhu by tak pán zaslal data pouze jednou a čekal by, až mu všichni sluhové ohlásí, že mají osazené své části matice A. Tomuto způsobu se říká statické rozdělování. Problém nastává, když je cluster heterogenní a některé procesory

jsou rychlejší než jiné. Při stejně velkých blocích by tak rychlejší procesory nečinně čekaly, než své výpočty ukončí pomalejší uzly.

Proto rozdělíme data na více bloků a ty budeme distribuovat mezi sluhy postupně, dynamicky v závislosti na tom, které sluha je schopen provést další část výpočtů. Pokud některý sluha ukončí výpočet na přiděleném bloku dat, zažádá o blok další. Na konci distribuce je třeba oznámit sluhům, že všechna data byla rozeslána. Pokud tuto zprávu obdrží i poslední sluha, může začít výpočet skalárního součinu.

Nyní tento algoritmus zapíšeme v pseudokódu (Algo 5).

```
 1  if master then
 2  begin
 3     /* stanoveni délky bloku; "k" je konstanta udávající průměrný
 4      * počet bloků připadající na jednoho sluhu, "ntrngs" je
 5      * počet všech trojúhelníků a "nslaves" počet sluhů */
 6    block_length = ntrngs/(nslaves*k);
 7    while current_block <=last_block do
 8      slaveid = getDataRequest;
 9      sendCurrentBlockToSlave current_block slaveid;
10      currentBlock = returnNewBlock currentBlock block_length;
11    done;
12    n = 0;
13    while n < nslaves do
14      slaveid = getDataRequest;
15      sendNoDataMsg slaveid;
16      n = n+1;
17    done;
18  end
19
20  if slave then
21  begin
22    myblock = sendDataRequest;
23    while block <> "NoData" do
24      submatrices = makeAllComputationOnBlock block;
25      listOfSubmatrices = addSubMtoList submatrices;
26      block = sendDataRequest;
27    done;
28  end
```

Algo 5: Přidělení bloků dat sluhům

V aplikaci používáme dva druhy bloků: bloky trojúhelníků pro výpočet operátorů S a D^* a blok bodů pro výpočet operátoru N. Tyto bloky nejsou vzájemně nějak spjaty – pán rozdělí do stejného počtu bloků trojúhelníky i body a pokud sluha zažádá o data, dostane *i*-tý blok trojúhelníků a *i*-tý blok bodů, kde $i \in (1;\text{počet\_bloků})$.

Nyní se ještě budeme zabývat parametrem *k*, který se objevil na šestém řádku algoritmu 5. Jde o průměrný počet bloků připadajících na jednoho sluhu. Pro účely co

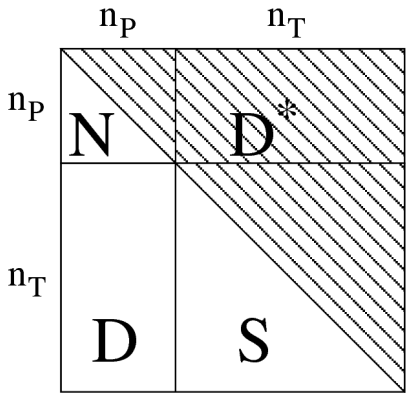
nejlepšího rozvržení dat mezi sluhy by bylo výhodné mít bloky relativně malé. Z hlediska výpočtu operátoru N z hodnot vypočítaných operátorem S je však naopak lepší, když jsou bloky co největší a maximalizuje se tak šance, že trojúhelníky použité pro výpočet operátoru N byly již použity pro výpočet operátoru S. Nastavení velikosti bloku a s ní související počet bloků závisí mj. i na parametrech výpočetního clusteru a proto hodnotu parametru *k* nastavíme experimentálně – pro malá vstupní data budeme parametr měnit a jeho nejlepší hodnotu pak použijeme pro výpočet s náročnějšími vstupními daty. Univerzální přednastavenou hodnotou je $k = 4$.

3.3.5 Uložení matice v paměti

V sekcích 3.2.3 a 3.3.2 jsem se již zmínil o formě uložení matice v paměti. Nyní si formu uložení matic popišme detailněji.

Vlastnost submatic N a S je, že jsou symetrické. Stačí nám tedy spočítat a uložit pouze jejich polovinu nad nebo pod diagonálou. Submatice D^* a D samy o sobě symetrické nejsou. Jsou však symetrické vzájemně, tedy $D^*(i,j) = D(j,i)$ pro všechna $i, j \in (1;n_P)$. Z toho plyne, že nám stačí spočítat pouze prvky submatice D^*.

Na obr. 7 tedy vidíme, jakou část matice A musíme držet v paměti (tato část je šrafovaná). Bílou část můžeme jednoduše odvodit díky symetrii matice. Připomeňme, že horní trojúhelníkovou matici budu v programu ukládat po blocích, viz obr. 4. Tyto bloky budou shodné s výpočetními bloky slaves.



obr. 7.: Uložení matice A do paměti. Stačí uložit pouze šrafovanou část, bílá část lze díky symetrii matice lehce dopočítat

3.3.6 Paralelní výpočet součinu Ax

Výpočet paralelního součinu A**x** je v zásadě jednoduchý. Master si udržuje informace o částech matice A přidělených mezi slaves a na každý slave zašle příslušnou část vektoru **x**. Slave touto částí vynásobí svou submatici a výsledek pak spolu s identifikátorem submatice zašle masterovi. Master z dílčích součinů zkompletuje součin celkový.

Jediným drobným problémem je různá orientace přidělených bloků, viz obr. 6. Proto pro výpočet skalárního součinu na submatici D^* musí být napsána jiná funkce než pro výpočty na submaticích N a S. To už je ale implementační detail.

4 Nastavení běhového prostředí

V této kapitole se seznámíme s použitým softwarem (OCaml, MPI, stávající implementace BEM, podpůrné knihovny) a jeho nastavením na svazku počítačů *cmpgrid*. Program zdrojové soubory programu *pbem* jsou k dispozici na CD-ROM přiloženém k této práci. Obsah celého CD viz příloha B) Přiložené CD.

4.1 Implementace BEM pro jeden počítač

Tato práce je paralelním rozšířením [1], proto je velké množství kódu společné a velmi podobný je i postup jak obě implementace přeložit a spustit. Pro jednoduchost budu v dalším textu označovat [1] jako OBEM a paralelní verzi jako PBEM.

Významné rozdíly jsou tyto:

- Modul `solver.ml` je v PBEM nahrazen modulem `psolver.ml`, neboť jeho kód byl významně přepsán.
- Většina nově přidaných funkcí je uložena v souboru `parbem.ml`.
- Přímo do modulu `Meegmesh.TriangularMesh` bylo vloženo několik funkcí pro manipulaci s datovým typem `Meegmesh.TriangularMesh.t` Tyto funkce jsou vloženy mezi komentáře: „`==== Parallel extension ====`“

Ostatní soubory jsou shodné pro PBEM i OBEM.

Kromě stejných knihoven využívá PBEM ještě knihovnu MPI a rozhraní OCamlMPI. Popis všech knihoven a jejich nastavení je uveden níže v této kapitole.

4.2 Jazyk OCaml

OCaml (Objective Caml) [7] je funkcionální programovací jazyk odvozený od jazyka ML. Výhodou tohoto jazyka je, že umožňuje v omezené míře využívat i prvky imperativního programování (for, while cykly, podmínky) a nabízí velké množství rozhraní do C/C++/Fortran knihoven.

Jazyk je volně šiřitelný a dostupný pro různé platformy, viz [7]. Rovněž dokumentace je přehledná, vyčerpávající a snadno dostupná, kromě oficiálního manuálu v [7] doporučuji ještě např. [8].

Na počítačích *cmpgrid* je v současnosti OCaml ve verzích 3.08.1 a 3.09.2. Rozdílnost verzí pro naše účely nepředstavuje žádný problém. Knihovny jazyka jsou v adresáři /usr/lib/ocaml

4.3 Potřebné knihovny

Pro přeložení PBEM je třeba mít nainstalované a nakonfigurované následující knihovny.

4.3.1 Cubpack++

Knihovna je volně šiřitelná pouze pro výzkumné účely. Její domovská stránka je v [9].

Při instalaci standardním postupem jsme při použití GNU překladače `g++` verze 3.4.3 i 4.1.2 měli následující problémy s překladem:

```
./vstack.c: In member function 'VectorStack<T>&      \
         VectorStack<T>::operator=(const VectorStack<T>&)':
./vstack.c:41: error: 'TheSize' was not declared in this scope
./vstack.c:44: error: 'Contents' was not declared in this scope
```

ap.

Chyba je ve změněné syntaxi jazyka C++. Nyní automaticky překladač před proměnou nedoplňuje název templatu a je tedy třeba chybné řádky upravit podle následujícího příkladu:

výpis vstack.c: lines 37 až 54 (původní verze)

```
  template <class T>
  VectorStack<T>&
  VectorStack<T>::operator=(const VectorStack<T>& v)
    {
    if (TheSize == 0)
      {
      TheSize = v.TheSize;
      Contents = new T[TheSize];
      Error(!Contents,"Vectorassign: allocation failed");
      };
    Error( (TheSize != v.TheSize),"lengths of arrays
incompatible");
    for (unsigned int i =0;i<TheSize;i++)
      {
      Contents[i] = v.Contents[i];
      };
    return *this;
    }
```

Opravená verze, výpis téhož:

```
  template <class T>
  VectorStack<T>&
  VectorStack<T>::operator=(const VectorStack<T>& v)
    {
    if (VectorStack<T>::TheSize == 0)
      {
      VectorStack<T>::TheSize = v.TheSize;
      VectorStack<T>::Contents = new T[VectorStack<T>::TheSize];
      Error(!VectorStack<T>::Contents,"Vectorassign: allocation
failed");
      };
    Error( (VectorStack<T>::TheSize != v.TheSize),"lengths of
arrays incompatible");
    for (unsigned int i =0;i<VectorStack<T>::TheSize;i++)
      {
      VectorStack<T>::Contents[i] = v.Contents[i];
      };
    return *this;
    }
```

Podobným způsobem je nutné opravit všechny nahlášené chyby. Přeložená knihovna pak již funguje bez problémů.

Protože na *cmpgrid* nemáme právo zápisu do systémových adresářů, je třeba při překladu PBEM v souboru `Makefile` uvést cestu do adresáře `Cubpack++/Code`.

4.3.2 LACAML, BLAS a LAPACK

LACAML [10] je rozhraní z jazyka OCaml do knihoven BLAS (Basic Linear Algebra Subrutines, [11]) a LAPACK (Linear Algebra PACKage, [12]).

Na *cmpgrid* je nainstalovaná verze 19980702-r1 knihovny BLAS a verze 3.0-r1 knihovny LAPACK. Kompilace balíčku LACAML probíhá v pořádku. Jedinou menší překážkou, o které jsem si již zmínil, je nemožnost zápisu do systémových adresářů. Proto musíme balíček nainstalovat do svého domovského adresáře a nastavit k němu přístup.

Přístup nastavíme následovně:

Do souboru `~/.bashrc` přidáme řádek

```
export OCAMLFIND_CONF="$HOME/etc/findlib.conf"
```

vytvoříme adresář `~/etc/` a v něm soubor `findlib.conf`. V tomto souboru nastavíme cesty do adresáře, do kterého instalujeme OCaml balíčky (v tomto případě LACAML). To uděláme například vložením následujících řádků:

```
destdir = "/home.stud/jasanm1/ocamlpack"
path = "/home.stud/jasanm1/ocamlpack:/usr/lib/ocaml/site-\
        packages"
```

Jde o příklad, kdy instaluji balíčky do adresáře `ocamlpack` ve svém domovském adresáři. Více o nastavení cest ke knihovnám jazyka OCaml najdete v manuálové stránce souboru findlib.conf.

4.3.3 ExtLib

ExtLib [13] je knihovna různých funkcí rozšiřující standardní moduly jazyka OCaml. V OBEM i PBEM se používá k manipulaci s hashovacími tabulkami.

Instalace tohoto balíčku na *cmpgrid* probíhá bez problémů, jen je třeba mít správně nastavené cesty k adresářům (viz předchozí sekce).

4.3.4 MPI a OCamlMPI

Na *cmpgrid* je nainstalována knihovna LAM MPI [15] ve verzi 7.1.1. Pokud chceme tuto knihovnu používat i z jazyka OCaml, musíme si stáhnout a nainstalovat balíček OCamlMPI [16], což je rozhraní z OCaml do MPI.

Při překladu balíčku OCamlMPI se na *cmpgridu* při standardní konfiguraci objevuje mnoho chyb podobných následující:

```
/usr/lib64/libmpi.so: undefined reference to `al_append
```

Řešením je připojit při překladu všechny knihovny potřebné pro běh. O jaké knihovny se jedná uvádím přímo ve výpisu změněných řádků souboru `ocamlmpi/Makefile`:

```
mpi.cma: $(OBJS)
        $(OCAMLC) -a -o mpi.cma -custom $(OBJS) -cclib \
-lcamlmpi -ccopt -L  $(MPILIBDIR) -cclib -llammpio -cclib \
-llamf77mpi -cclib -lmpi -cclib -llam -cclib -lutil

mpi.cmxa: $(OBJS:.cmo=.cmx)
        $(OCAMLOPT) -a -o mpi.cmxa $(OBJS:.cmo=.cmx) -cclib\
-lcamlmpi -ccopt  \
    -L$(MPILIBDIR) -cclib -llammpi  o -cclib -llamf77mpi\
-cclib -lmpi -cclib -llam  -cclib  -lutil
```

Při použití této konfigurace balíček bez problémů přeložíme, což ověříme příkazem `make test`.

4.4 Spuštění programu

4.4.1 Kompilace programu

Pokud máme všechny knihovny správně nastaveny podle předchozích sekcí, rozbalíme archiv s PBEM a v adresáři se zdrojovými soubory zadáme příkaz `make nc`. Tím spustíme překlad, jehož výstupem je spustitelný soubor `pbem`.

4.4.2 Inicializaci MPI prostředí

Ještě dříve než spustíme samotnou aplikaci PBEM, musíme inicializovat *cmpgrid*. Na serveru *fu* spustíme příkaz `power` a zobrazí se nám seznam všech dostupných počítačů v gridu včetně jejich základních parametrů, jako velikost celkové a volné paměti, počet a typ procesorů, počet přihlášených uživatelů aj.

Ze seznamu vybereme počítače, na kterých PBEM spustíme a vytvoříme textový soubor `lamhost`, který má následující strukturu:

```
<nazev_pocitace1> [cpu=<pocet_procesoru>]
    <nazev_pocitace2> [cpu=<pocet_procesoru>]
        .
        .
        .
    <nazev_pocitaceN> [cpu=<pocet_procesoru>]
```

Detailnější popis struktury souboru naleznete v manuálové stránce příkazu `bhost`. Zde je příklad funkční na *cmpgrid*:

```
$ cat lamhost
ptak cpu=2
cmpgrid-65 cpu=2
cmpgrid-66 cpu=2
cmpgrid-67 cpu=2
```

Inicializaci gridu provedeme příkazem `lamboot -v lamhost`

4.4.3 Spuštění programu

Nyní již můžeme program spustit následujícím příkazem:

```
mpirun -np pocet_spustenych_procesu ./pbem mesh_file
dipole_file dipole_number output_file <k>
```

pocet_spustenych_procesu - doporučuji ho nastavit shodný s počtem procesorům případně při malém počtu procesorů ho o jednu zvýšit (master v porovnání se slaves téměř nezatěžuje systém, a to zvlášť při malém počtu slaves, více o metodě master – slave viz 4. kapitola)

mesh_file – soubor ve formátu .md, specifikace formátu viz příloha

dipole_file – soubor s uloženými parametry dipólů, specifikace formátu viz přílohy

dipole_number – počet dipólů

output_file – jméno výstupního souboru

k - průměrný počet bloků dat přidělených jednomu sluhovi, tento parametr je nepovinný, přednastavená hodnota je 4; více viz sekce 3.3.4

Tento postup však na *cmpgrid* nefunguje, protože puštění přes `mpirun` změní nastavení proměnné `PATH` a program `pbem` pak nemůže najít všechny knihovny potřebné pro svůj běh. Tento problém vyřešíme tak, že příkazu `mpirun` nebudeme předávat přímo

program `pbem`, ale skript, který před spuštěním `pbem` správně nastaví systémové proměnné. Tento skript může vypadat takto:

```
  $ cat exppbem
  #!/bin/bash

  SPHERES="/datagrid/MEEG/Spheres/geometrie";
  CESTA="/home.stud/jasanm1/vyvoj/kompilace";
  POCETDIPOLU=5;
  HD=5;
  K=5;

  exportLD_LIBRARY_PATH=\
"/home.stud/jasanm1/lib:$LD_LIBRARY_PATH"

  $CESTA/pbem $SPHERES/head.$HD.md $SPHERES/dipoles \
$POCETDIPOLU vystup$HD $K
```

Program pak spustíme příkazem:

```
  mpirun -np pocet_spustenych_procesu exppbem
```

5 Experimenty

5.1 Rychlost

Nejprve jsme provedli porovnání běhu aplikace *obem* a *pbem* na jednom procesoru (*pbem* běžela jako dvouprocesová aplikace). Běh *obem* trval 21 min 4.732 s a *pbem* 22 min 42.909 s. Vstupními daty byl soubor head.3.md.

Dále jsme provedli sadu testů, kdy jsme měřili rychlost výpočtu pro vstupní soubory head.[1-4].md (viz tab.1) na jednom až devíti procesorech. Počet spuštěných procesů (kopií programu *pbem*) byl vždy o jeden větší než počet procesorů, protože na jednom stroji běžela zároveň *master* i *slave* verze programu. Výpočet tím byl ovlivněn zanedbatelně, neboť pán pracuje v době, kdy sluhové čekají na další pokyny a navíc doba běhu pána je oproti sluhům velmi krátká. Změřené časy vidíme v tabulce (tab. 2).

počet cpu	head.1.md	head.2.md	head.3.md	head.4.md
1	0:08	1:13	21:04	744:45
2	0:06	0:42	10:43	307:07
3	0:05,4	0:32,6	7:45	195:12
4	0:05,7	0:29	6:39	143:15
5	0:05	0:23	4:33	112:58
6	0:05	0:20	3:53	82:16
7	0:05	0:19	3:21	69:46
8	0:04,7	0:17	2:58	55:20
9	0:06	0:17,7	2:46	51:15

tab. 2: Závislost doby běhu programu na počtu procesorů. Doba běhu je uvedena ve formátu minuty:sekundy,desetiny_sekund

V ideálním případě by měla být závislost času výpočtu na počtu procesorů popsána lineární lomenou funkcí

$$f(p) = t_1 / p, \tag{7}$$

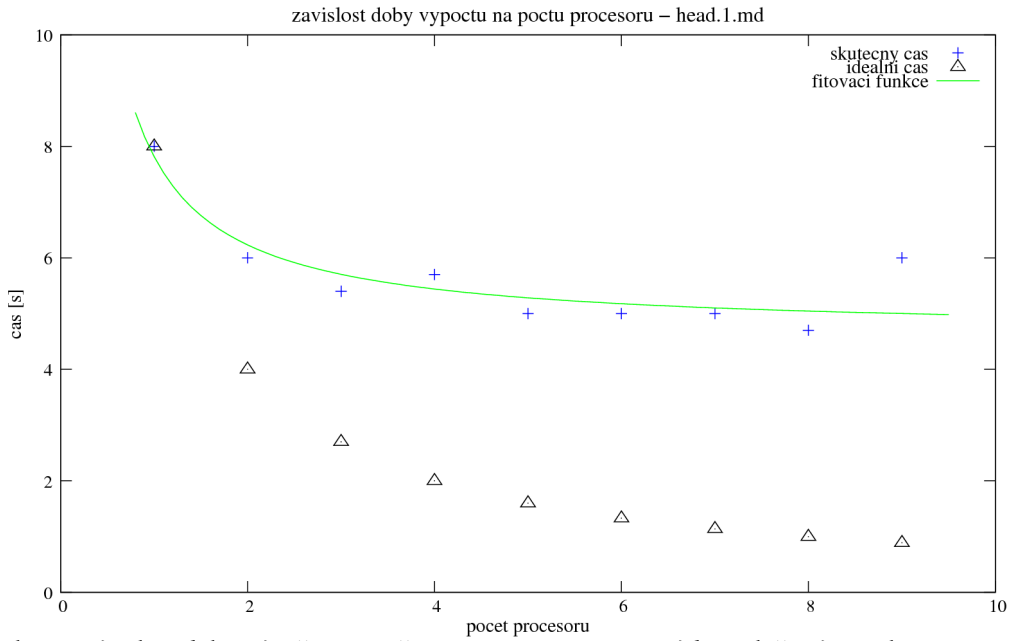
kde p je počet procesorů a t_1 doba běhu programu na jednom procesoru. V praxi je doba běhu na p procesorech $t_p > f(p)$, neboť samotná paralelní komunikace zabírá určitý

čas a navíc při paralelním návrhu někdy musíme přijímat řešení zvyšující časovou náročnost (sekce 3.3).

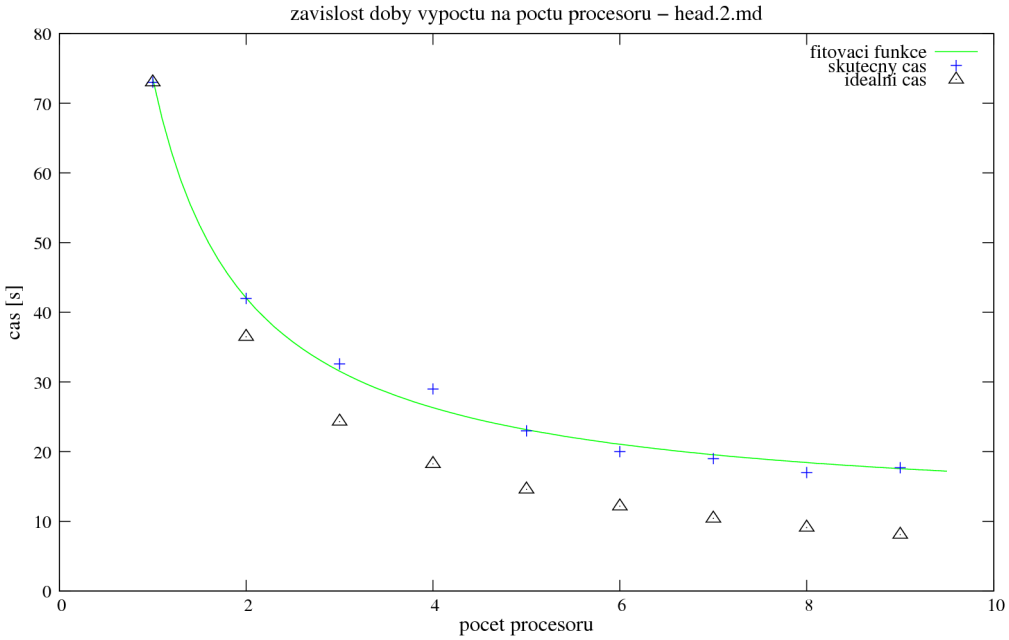
V následujících grafech (obr. 8-11) je křížky vynesena skutečná, naměřená délka běhu programu. Trojúhelníček značí dobu ideální, spočítanou ze vztahu (7), přičemž za hodnotu t_1 byla vzata změřená doba běhu programu na jednom procesoru. Křivka je určena funkcí $f_{real}(p) = a/p + b$, kde konstanty a, b byly zjištěny z naměřených hodnot funkcí *nlinfit* programu Matlab ® [30]. V tabulce (tab. 3) jsou uvedeny konstanty a, b pro různá vstupní data.

	a	b
head.1.md	3,16	4,65
head.2.md	62,95	10,58
head.3.md	1231,7	38,0
head.4.md	775,1 [min]	-47,9 [min]

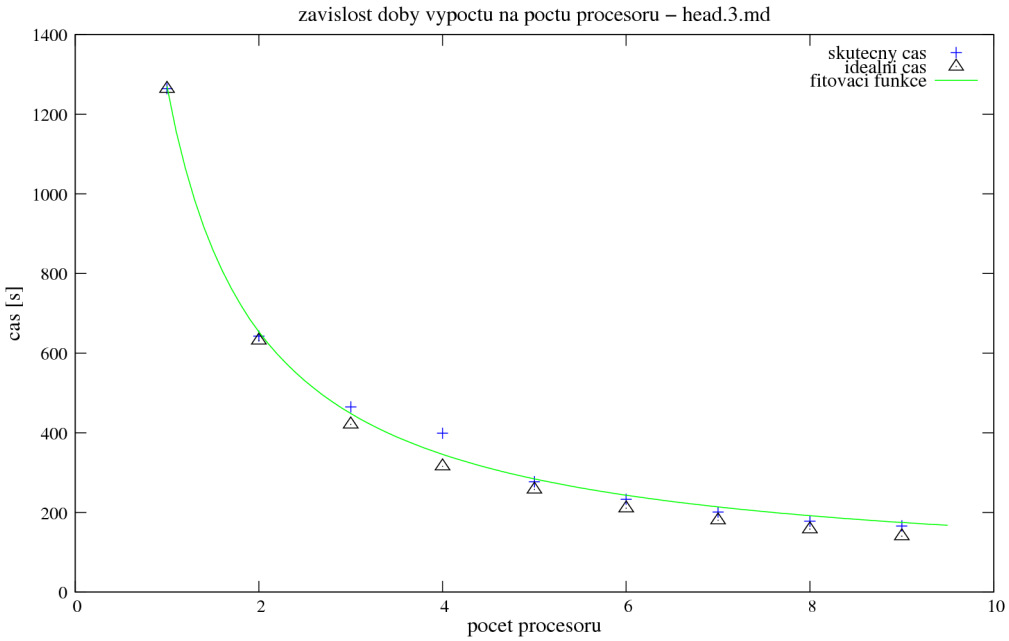
tab. 3: Konstanty lineární lomené funkce f popisující závislost doby běhu programu na počtu procesorů. Rozměr konstant je sekunda, pouze u dat ze souboru head.4.md minuta



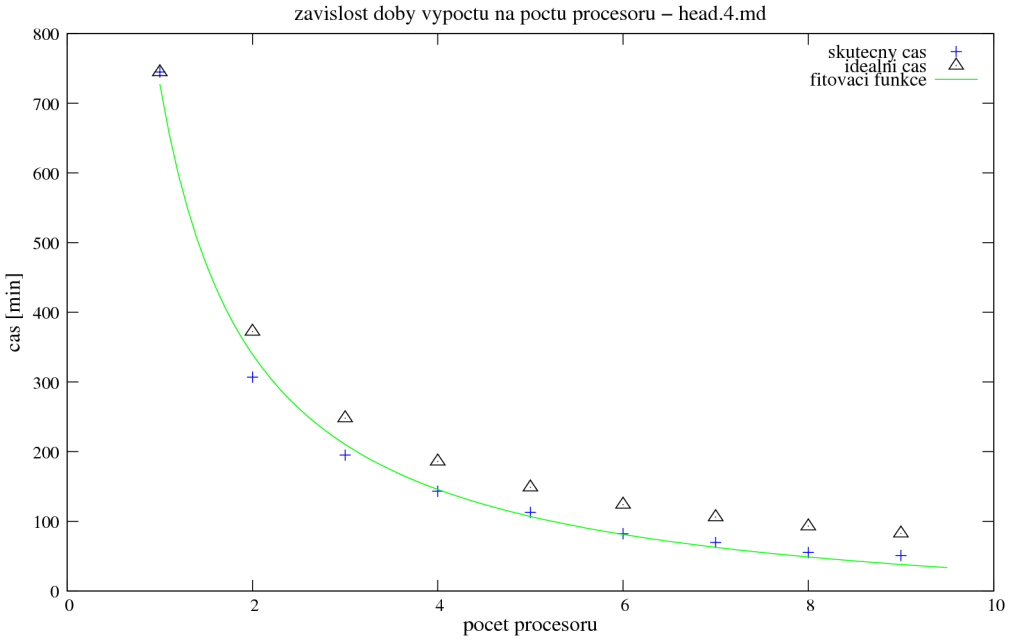
obr. 8.: Závislost doby výpočtu na počtu procesorů pro vstupní data uložená v souboru head.1.md



obr. 9.: Závislost doby výpočtu na počtu procesorů pro vstupní data uložená v souboru head.2.md



obr. 10.: Závislost doby výpočtu na počtu procesorů pro vstupní data uložená v souboru head.3.md



obr. 11.: Závislost doby výpočtu na počtu procesorů pro vstupní data uložená v souboru head.4.md. Oproti obr. 8-10 je v tomto grafu osa y pokryta údaji v minutách.

Ze čtvrtého řádku tab. 3 (záporná konstanta b) i z grafu (obr. 11) vidíme, že naše urychlení je lepší než urychlení ideální. Není tomu tak, pouze při měření doby t_1 pro vstupní data ze souboru head.4.md bylo dat tolik, že se již nevešla do operační paměti počítače, na kterém program běžel, a musela být proto zapisována do odkládacího oddílu na pevném disku. Rychlost ukládání a čtení dat z pevného disku je mnohonásobně nižší než při čtení/zápisu z operační paměti. Kvůli tomu je doba t_1 tak vychýlena, že „ideální“ hodnoty z ní stanovené jsou vyšší než hodnoty naměřené. Při větším počtu spolupracujících počítačů se již všechna data vešla do operační paměti a nemusela být ukládána na disk.

Při interpretování naměřených údajů musíme brát v úvahu také to, že *cmpgrid* je heterogenní cluster a jednotlivé procesory nejsou stejně výkonné.

5.2 Paměťová náročnost

Z časových důvodů bohužel nebyly provedeny experimenty, kterými bychom ověřili, jak velká data je program schopen zpracovat. Nicméně i během našich výpočtů se ukazovalo, že program při svém běhu zabírá přibližně třikrát více paměti, než je velikost

matice A stanovená v tab.1. V této oblasti bude třeba provést ještě více testů a optimalizací. Pomoci by mohlo častější volání garbage collectoru, neboť se stávalo, že program najednou uvolnil přibližně 2 GB dat. Nicméně jedná se pouze o domněnku, která musí být potvrzena.

5.3 Přesnost

Přesnost výsledků jsme porovnávali s výstupy programu *obem*. Výstupní soubory naleznete na CD-ROM, které je součástí této práce (Příloha B) Přiložené CD). Počítali jsme relativní odchylku obou výstupů, definovanou jako:

$$\epsilon = \|\mathbf{v}_{\text{obem}} - \mathbf{v}_{\text{pbem}}\| / \|\mathbf{v}_{\text{obem}}\|$$

V tabulce (tab. 4) vidíte v prvním sloupci chyby spočítané mezi *obem* a *pbem*, přičemž *pbem* běžela na jednom počítači jako dvouprocesová aplikace, ve druhém sloupci pak chyby mezi dvěma výpočty *pbem*, kdy jednou byla *pbem* spuštěna na jednom počítači, podruhé na pěti procesorech. V posledním sloupci je chyba mezi paralelně počítanou *pbem* a *obem*.

	obem – pbem*	pbem* – pbem	obem – pbem
head.1.md	9,6148E-11	1,4651E-11	1,7023E-11
head.2.md	6,3798E-08	1,4100E-07	8,7893E-08
head.3.md	5,3197E-07	2,6099E-07	8,2452E-07
head.4.md	1,2586E-07	1,3015E-06	1,1949E-06

tab. 4: Relativní chyba mezi aplikací obem, pbem počítané na jednom procesoru (značená *) a pbem počítané paralelně.

Vidíme, že se zvětšující se velikostí úlohy se zvětšuje i relativní chyba.

6 Závěr

Navrhli jsme a implementovali paralelní verzi řešení EEG/MEG dopředného problému metodou hraničních prvků. Cílem paralelizace bylo urychlení výpočtu a zvětšení množství vstupních dat, které je program schopen zpracovat. Výpočet se nám podařilo urychlit dostatečně – doba běhu programu se blíží ideálnímu stavu popsanému funkcí $f(p) = t_1 / p$, kde p je počet procesorů a t_1 doba běhu programu na jednom procesoru.

Z časových důvodů jsme bohužel nedokázali provést experimenty na velkém množství vstupních dat a druhý cíl této práce jsme tak neověřili.

7 Použitá literatura

[1] KYBIC, J. *BEM implementation in OCaml* [počítačový program], 2003

[2] CLERC M., DERVIEUX A., KERIVEN R., FAUGERAS O., KYBIC J. and PAPADOPOULO T. Comparison of BEM and FEM methods for the E/MEG problem. *Biomag 2002: Proceedings of the 13th International Conference on Biomagnetism*, pages 688-690, Germany, August 2002. VDE Verlag GmbH.

[3] WOLTERS C. H. A Finite Element Method in EEG/MEG Source Analysis. *Siam News*, Volume 40, Number 2, March 2007.

[4] ZHUKOV L., WEINSTEIN D. and JOHNSON CH., *Independent Component Analysis For EEG Source Localization In Realistic Head Models,* 1999, poslední revize březen 2005. Dostupné z: <http://waggle.gg.caltech.edu/~zhukov/research/eeg_meg/ieee-emb/paper16s.html>

[5] KYBIC J., CLERC M., ABBOUD T., FAUGERAS O., KERIVEN R., and PAPADOPOULO T., A common formalism for the integral formulations of the forward EEG problem, *IEEE Transactions on Medical Imaging*, vol. 24, no. 1, pp. 12–28, Jan. 2005.

[6] JIRÁNEK, P.: Iterační metody pro řešení soustav lineárních rovnic. *K^7*, Vol. 3, ISSN 1214-7370, 2004

[7] *Objective Caml,* poslední revize 1. 5. 2004. Dostupné z <http://caml.inria.fr/ocaml/>.

[8] *Objective Caml Tutorial,* poslední revize 6. 5. 2007. Dostupné z <http://www.ocaml-tutorial.org>.

[9] COOLS R., LAURIE D. et al., *Cubpack++* [počítačový program]. Ver. 1.2.1. Dostupné z <http://www.cs.kuleuven.ac.be/~nines/software/cubpack/>

[10] MOTTL M., *LACAML* [počítačový program]. Ver. 1.0-0. Dostupné z <http://www.ocaml.info/home/ocaml_sources.html>

[11] *BLAS (Basic Linear Algebra Subrutines)* [počítačový program]. Ver. 1.2-8. Dostupné z <http://www.netlib.org/blas/>

[12] *LAPACK (Linear Algebra PACKage)* [počítačový program]. Ver. 3.1.1. Dostupné z <http://www.netlib.org/lapack/>

[13] CANNASSE N., HURT B., YORIYUKI Y., *ExtLib* [počítačový program]. Ver. 1.5. Dostupné z <http://ocaml-lib.sourceforge.net/>

[14] *MPI* [popis standardu]. Dostupné z <http://www-unix.mcs.anl.gov/mpi/>

[15] *LAM MPI* [počítačový program]. Ver. 7.1.1. Dostupné z <http://www.lam-mpi.org>

[16] OCamlMPI [počítačový program]. Ver. 1.0.0. Dostupné z <http://cristal.inria.fr/~xleroy/software.html#ocamlmpi>

[17] VOLNÝ, M. *Paralelní algoritmy* [online]. 2004. Dostupné z <http://www.cs.vsb.cz/jakl/pds/volny/index.html>

[18] *PVM (Parallel Virtual Machine)* [počítačový program]. Ver. 3. Dostupné z <http://www.netlib.org/pvm3/>

[19] *MPICH2* [počítačový program]. Ver. 1.0.5p4. Dostupné z <http://www-unix.mcs.anl.gov/mpi/mpich/>

[20] TVRDÍK, P. *Paralelní systémy a algoritmy*, Praha: ČVUT, 2006

[21] SILBERNAGL S., DESPOPOULOS A. *Atlas fyziologie člověka.* Vydání 3. české. Praha: Grada Publishing, a. s., 2004. ISBN 80-247-0630-X

[22] PHILLIPS J., LEAHY R. et al. Imaging neural activity using meg and eeg. *IEEE Engieering in Medicine and Biology Magazine,* vol. 16, no. 3, pp. 34–42, 1997

[23] MACHÁC J. *Numerické metody v elektromagnetickém poli.* Vydání 1. Praha: ČVUT, 2002. ISBN 80-01-02476-8

[24] GONZÁLES P., PENA T. F., CABALEIRO J. C. Parallel sparse approximate preconditioners applied to the solution of BEM systems. *Engineering Analysis with Boundary Elements,* vol. 28, no. 9, pp. 1061-1068, 2004

[25] TANG H. K., YANG Y. J. Parallelization of BEM Electrostatic Solver Using a PC Cluster. *Nanotech 2004*, vol. 2, pp. 482-485, 2004

[26] GONZÁLES P., PENA T. F., CABALEIRO J. C. Parallel iterative scheme for solving BEM using Fast Wavelet transform. *Developments in Engineering Computational Technology*, B. H. Topping, Ed. Civil-Comp press, Edinburgh, UK, pp. 249-257, 2000

[27] CUNHA M. T., TELLES J. C., COUTINHO A. L. A portable parallel implementation of a boundary element elastostatic code for shared and distributed memory systems. *Adv. Eng. Softw.* Vol. 35, no. 7, pp. 453-460, 2004

[28] YUAN Y., BANERJEE P. A Parallel 3-D Capacitance Extraction Program. *Proceedings of the 6th international Conference on High Performance Computing*, vol. 1745. Springer-Verlag, London, pp. 202-206, 1999

[29] MADER G., UHLMANN F. H., A parallel implementation of a 3D-BEM-algorithm using distributedmemory algorithms. *IEEE Transaction on Magnetics,* vol. 33, no. 2, pp. 1796-1799, 1997

[30] *Matlab* ® [počítačový program]. The MathWorks, Inc., <http://www.matlab.com>

Přílohy

A) Formát vstupních souborů

md file: Jde o soubor, ve kterém je uložen povrch oblastí ve tvaru trojúhelníkové mřížky (2D mesh). Formát souboru je následující:

```
points počet_bodů
x1 y1 z1
x2 y2 z2
x3 y3 z3
   .
   .
   .
xN yN zN
triangles počet_trojúhelníků počet_povrchů
category sigma_int sigma_out
p1 q1 r1
p2 q2 r2
p3 q3 r3
   .
   .
   .
pM qM zM
```

Kde `x y z` jsou souřadnice jednotlivých bodů typu float, `p q r` pořadová čísla bodů/vrcholů trojúhelníku a *`sigma_int sigma_out`* jsou vnitřní, resp. vnější vodivosti oblastí oddělených plochou modelovanou plochou. N je pak počet všech bodů na všech plochách a M počet trojúhelníků na jedné ploše.

dipole file: Tento soubor popisuje na každém řádku jeden dipól. Formát řádků je následující:

```
x y z a b c
```

kde `x y z` jsou souřadnice polohového vektoru **r** dipólu a `a b c` jsou souřadnice vektoru orientujícího dipól (vektor [`a b c`]).

B) Přiložené CD

Na CD naleznete soubor *pbemocaml.tar.gz*, ve kterém jsou kompletní zdrojové kódy nutné pro přeložení programu *pbem* (včetně kódu společného s původní implementací *obem*).

Dále zde naleznete adresář *ocamlpack*, kde jsou balíčky a knihovny nutné ke kompilaci *pbem*.

V adresáři *results* jsou výsledky výpočtů programů *obem* a *pbem* pro vstupní soubory *head.*.md* a *dipoles* z adresáře *spheres*. Výsledky počítané *pbem* jsou rozdělené na dvě části – pbem_multi (výpočet probíhal na pěti procesorech) a pbem (výpočet probíhal pouze na jednom procesoru).

V kořenovém adresáři je rovněž text této diplomové práce ve formátu pdf.