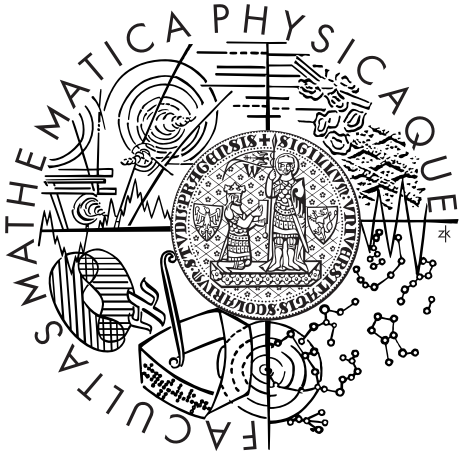
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Bc. Viktor Šíp

Numerická simulace proudění stlačitelných tekutin pomocí paralelních výpočtů

Katedra numerické matematiky MFF UK

Vedoucí diplomové práce:	doc. RNDr. Vít Dolejší, Ph.D., DSc.
Studijní program:	Matematika
Studijní obor:	Numerická a výpočtová matematika

Praha 2011

Děkuji doc. RNDr. Vítu Dolejšímu, Ph.D., DSc., mému vedoucímu práce, za zajímavé téma, odborné rady a podnětné připomínky. Dále děkuji RNDr. Martinu Mádlíkovi, Ph.D., za konzultace a technickou pomoc s paralelním programováním a používáním clusteru.

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V Praze dne 31. 7. 2011 Viktor Šíp

Název práce: Numerická simulace proudění stlačitelných tekutin pomocí paralelních výpočtů

Autor: Bc. Viktor Šíp

Katedra: Katedra numerické matematiky

Vedoucí diplomové práce: doc. RNDr. Vít Dolejší, Ph.D., DSc., Katedra numerické matematiky

Abstrakt: Předmětem práce je paralelní implementace programu na numerickou simulaci proudění stlačitelných tekutin. Program je založen na nespojité Galerkinově metodě, která je díky svým vlastnostem velmi vhodná pro paralelizaci. V práci popíšeme Navier-Stokesovy rovnice a jejich diskretizaci pomocí nespojité Galerkinovy metody. Vyložíme výhody, které použití nespojité Galerkinovy metody přináší, a formulujeme algoritmus pro běh na jediném procesoru. Dále se zaměříme na paralelní implementaci algoritmu a jednotlivé problémy s tím související. V poslední kapitole předložíme výsledky numerických experimentů ukazujících efektivitu paralelní implementace.

Klíčová slova: Nespojitá Galerkinova metoda, paralelní výpočty, Navier-Stokesovy rovnice, proudění tekutin

Title: Numerical simulation of compressible flows using the parallel computing

Author: Bc. Viktor Šíp

Department: Department of Numerical Mathematics

Supervisor: doc. RNDr. Vít Dolejší, Ph.D., DSc., Department of Numerical Mathematics

Abstract: In the present work we implemented parallel version of a computational fluid dynamics code. This code is based on Discontinuous Galerkin Method (DGM), which is due to its favourable properties suitable for parallelization. In the work we describe the Navier-Stokes equations and their discretization using DGM. We explain the advantages of usage of the DGM and formulate the serial algorithm. Next we focus on the parallel implementation of the algorithm and several particular issues connected to the parallelization. We present the numerical experiments showing the efficiency of the parallel code in the last chapter.

Keywords: Discontinuous Galerkin method, parallel computing, Navier-Stokes equations, fluid dynamics

Obsah

Úvod

V mnoha odvětvích průmyslu a vědy se setkáváme s problémem proudění tekutin. K jeho řešení existují tři cesty: praktické experimenty, analytické řešení a počítačová simulace. Praktické experimenty v aerodynamických tunelech jsou však jak časově, tak i finančně náročné a analytické řešení Navier-Stokesových rovnic, které proudění tekutin popisují, lze nalézt jen pro velmi jednoduché problémy. Proto mají metody počítačové simulace proudění tekutin (*Computational Fluid Dynamics, CFD*) již dlouho své nezastupitelné místo v mnoha oborech, jako je letectví, automobilový průmysl či strojírenství.

Se zvyšujícím se výkonem dostupného hardware se však také neustále zvětšují problémy přicházející z průmyslové praxe. Problémy s miliony elementů již nejsou ničím neobvyklým a je tedy nutné hledat postupy, pomocí kterých je možné náročné úlohy řešit v rozumném čase. V současné době jsou trendem paralelní výpočty na výpočetních clusterech, na kterých se nárůst výkonu a dostupné paměti realizuje přidáváním dalších výpočetních jednotek. Zvýšení výkonu je tak mnohonásobně levnější, než kdybychom se snažili zvyšovat výkon jednotlivých procesorů.

Tyto výpočetní clustery se mohou skládat z desítek až tisíců procesorových jader propojených komunikační sítí. Proto je k simulaci nutné vybrat dobře paralelizovatelnou numerickou metodu, u které lze potřebné výpočty efektivně rozdělit mezi velké množství procesorů. Při snaze o efektivní paralelizaci programů nám totiž v cestě stojí dvě překážky: velké množství nutné komunikace mezi procesory a nevyvážené množství práce jim přidělené. Naším cílem by tedy mělo být rozdělit problém na několik dílčích problémů, stejně výpočetně náročných a na sobě v co největší možné míře nezávislých.

Jak v práci ukážeme, nespojitá Galerkinova metoda (*Discontinuous Galerkin Method, DGM*), kterou se v práci zabýváme, je pro tento postup velmi vhodná. DGM v sobě kombinuje myšlenky metody konečných prvků a metody konečných objemů a z obou si bere to lepší - především libovolný řád aproximace a kompaktnost vzniklého numerického schématu. Za to platíme cenu v podobě vyšších výpočetních nároků.

Hlavním cílem této práce byla paralelizace programu ADGFEM [1] pro řešení parciálních diferenciálních rovnic pomocí nespojité Galerkinovy metody. Tento program je vyvíjen na Katedře Numerické Matematiky MFF UK pod vedením doc. Víta Dolejšího. Práce volně navazuje na diplomovou práci Mgr. Michala Zajace [18], obhájenou v roce 2008.

V práci se zabýváme následujícími tématy: v kapitole 1 formulujeme Navier-Stokesovy rovnice, které popisují proudění stlačitelných tekutin a popíšeme jejich základní vlastnosti. V kapitole 2 se věnujeme popisu DGM, jejímu srovnání s dalšími metodami používanými v CFD, diskretizaci Navier-Stokesových rov-

nic pomocí DGM a numerickému řešení vzniklého problému. Způsob paralelizace programu pomocí rozdělení výpočetní sítě na části je podrobně rozebrán ve třetí kapitole. Numerické výsledky ukazující, jak se paralelizace programu zdařila, jsou prezentovány v kapitole 4.

Kapitola 1

Navier-Stokesovy rovnice

1.1 Formulace

Chování vazkých stlačitelných tekutin je popsáno tzv. Navier-Stokesovými rovnicemi. Navier-Stokesovy rovnice jsou nelineární parciální diferenciální rovnice a my je zformulujeme v následujícím textu. Jejich tvar lze odvodit ze zákonů zachování, toto odvození lze nalézt např. v [10].

Mějme omezenou oblast $\Omega \in \mathbb{R}^d, d = 2, 3$ a čas $T > 0$. Označme časoprostorový válec $Q_T = \Omega \times (0, T)$ a uvažujme následující fyzikální veličiny:

- hustotu ρ;
- rychlost $\boldsymbol{v} = (v_1, \cdots, v_d)$;
- celkovou energii e;
- tlak p.

Navier-Stokesovy rovnice můžeme zapsat v bezrozměrném tvaru

$$\frac{\partial \boldsymbol{w}}{\partial t} + \nabla \cdot \vec{\boldsymbol{f}}(\boldsymbol{w}) = \nabla \cdot \vec{\boldsymbol{R}}(\boldsymbol{w}, \nabla \boldsymbol{w}) \quad \text{v} \quad Q_T. \tag{1.1}$$

Na levé straně rovnice stojí časová derivace hledaného stavového vektoru,

$$\boldsymbol{w} = \begin{pmatrix} \rho \\ \rho v_1 \\ \vdots \\ \rho v_d \\ e \end{pmatrix}$$

a dále divergence nevazkých (Eulerových) toků,

$$\vec{\boldsymbol{f}}(\boldsymbol{w}) = (\boldsymbol{f}_1(\boldsymbol{w}), \cdots, \boldsymbol{f}_d(\boldsymbol{w}))$$

s jednotlivými složkami

$$\boldsymbol{f}_s = \begin{pmatrix} \rho v_s \\ \rho v_s v_1 + \delta_{s1} p \\ \vdots \\ \rho v_s v_d + \delta_{sd} p \\ (e + p) v_s \end{pmatrix}, \qquad s = 1, \cdots, d.$$

Ve výrazu na pravé straně vidíme

$$\vec{\boldsymbol{R}}(\boldsymbol{w}, \nabla \boldsymbol{w}) = (\boldsymbol{R}_1(\boldsymbol{w}, \nabla \boldsymbol{w}), \cdots, \boldsymbol{R}_d(\boldsymbol{w}, \nabla \boldsymbol{w}))$$

což jsou vazké toky s

$$\boldsymbol{R}_s(\boldsymbol{w}, \nabla \boldsymbol{w}) = \begin{pmatrix} 0 \\ \tau_{s1} \\ \vdots \\ \tau_{sd} \\ \sum_{k=1}^{d} \tau_{sk} v_k + \frac{\gamma}{RePr} \frac{\partial \theta}{\partial x_s} \end{pmatrix}, \qquad s = 1, \cdots, d.$$

Zde jsme použili značení: γ Poissonova konstanta, Re Reynoldsovo číslo, Pr Prandtlovo číslo, θ teplota, τ vazká část tenzoru napětí a δ_{ij} Kroneckerovo delta. Symboly ∇ a $\nabla\cdot$ označujeme gradient a divergenci. Uvažujeme Newtonovskou tekutinu, pro kterou je vazká část tenzoru napětí symetrická a jednotlivé prvky mají tvar

$$\tau_{sk} = \frac{1}{Re} \left[\left(\frac{\partial v_s}{\partial x_k} + \frac{\partial v_k}{\partial v_s} \right) - \frac{2}{3} \sum_{i=1}^{d} \frac{\partial v_i}{\partial x_i} \delta_{sk} \right].$$

Navíc předpokládáme ideální plyn, pro který můžeme tuto soustavu doplnit o rovnici

$$p = (\gamma - 1)(e - \rho |\boldsymbol{v}|^2/2) \tag{1.2}$$

a definici celkové energie

$$e = c_V \rho \theta + \rho |\boldsymbol{v}|^2/2, \tag{1.3}$$

kde c_V je specifické teplo při konstantním objemu. Tuto soustavu rovnic musíme doplnit vhodnou počáteční podmínkou

$$\boldsymbol{w}(x, 0) = \boldsymbol{w}^0(x) \quad x \in \Omega \tag{1.4}$$

a okrajovými podmínkami. Rozdělme $\partial\Omega = \partial\Omega_i \cup \Omega_o \cup \Omega_w$, kde Ω_i, Ω_o a Ω_w jsou postupně vstup, výstup a nepropustná stěna. Na těch předepisujeme podmínky

$$a) \quad \rho = \rho_D, \quad \boldsymbol{v} = \boldsymbol{v}_D, \quad \sum_{k=1}^{d} \left(\sum_{l=1}^{d} \tau_{lk} n_l \right) v_k + \frac{\gamma}{RePr} \frac{\partial \theta}{\partial \boldsymbol{n}} = 0 \quad \text{na} \quad \partial\Omega_i, \tag{1.5}$$

$$b) \quad \sum_{k=1}^{d} \tau_{sk} n_k = 0, s = 1, \cdots, d, \quad \frac{\partial \theta}{\partial \boldsymbol{n}} = 0 \quad \text{na} \quad \partial\Omega_o, \tag{1.6}$$

$$c) \quad \boldsymbol{v} = 0, \quad \frac{\partial \theta}{\partial \boldsymbol{n}} = 0 \quad \text{na} \quad \partial\Omega_w, \tag{1.7}$$

kde ρ_D a $\boldsymbol{v}_D$ jsou předepsané funkce a $\boldsymbol{n}$ je jednotková normála $\partial\Omega$ orientovaná ven z oblasti.

Poznamenejme zde, že vynecháním vazkých toků v Navier-Stokesových rovnicích (1.1), tedy pro $Re \to \infty$, dostaneme nevazké Eulerovy rovnice

$$\frac{\partial \boldsymbol{w}}{\partial t} + \nabla \cdot \vec{\boldsymbol{f}}(\boldsymbol{w}) = 0 \quad \text{v} \quad Q_T.$$

Problém určený Navier-Stokesovými rovnicemi (1.1), stavovými rovnicemi (1.2) a (1.3), počáteční podmínkou (1.4) a okrajovými podmínkami (1.5) - (1.7) nazýváme *problémem proudění stlačitelných tekutin.*

Ještě zde doplníme platné vztahy pro vazké i nevazké toky, které využijeme v další části. Nevazké toky $\boldsymbol{f}_s$ můžeme vyjádřit jako

$$\boldsymbol{f}_s = \boldsymbol{A}_s(\boldsymbol{w})\boldsymbol{w}, \quad s = 1, \cdots, d, \tag{1.8}$$

kde

$$\boldsymbol{A}_s(\boldsymbol{w}) \equiv \frac{D\boldsymbol{f}_s(\boldsymbol{w})}{D\boldsymbol{w}} \tag{1.9}$$

jsou Jacobiho matice zobrazení $\boldsymbol{f}_s$.

Vazké toky můžeme vyjádřit jako

$$\boldsymbol{R}_s(\boldsymbol{w}, \nabla\boldsymbol{w}) = \sum_{k=1}^{d} \boldsymbol{K}_{s,k}(\boldsymbol{w})\frac{\partial \boldsymbol{w}}{\partial x_k}, \tag{1.10}$$

kde $\boldsymbol{K}_{s,k}$ jsou matice rozměrů $(d+2) \times (d+2)$, jejich tvar lze nalézt například v [10].

Kapitola 2

Diskretizace problému

2.1 Tradiční metody

V této kapitole se budeme zabývat diskretizací problému z první kapitoly pomocí nespojité Galerkinovy metody (Discontinuous Galerkin Method, DGM).

Ještě než se začneme věnovat aplikaci DGM na Navier-Stokesovy rovnice, věnujme pár odstavců hlavním myšlenkám DGM a srovnání této metody s metodami ostatními používanými v oblasti CFD: metodou konečných objemů (Finite Volume Method, FVM) a metodou konečných prvků (Finite Element Method, FEM). Obě tyto metody mají své výhody i nevýhody, a nevýhody obou pomáhá překonat právě nespojitá Galerkinova metoda.

Metoda konečných objemů vychází z integrálního tvaru zákonů zachování. Základní myšlenka spočívá v diskretizaci výpočetní oblasti na jednotlivé elementy a aproximaci řešení konstantní hodnotou na každém elementu. Nejednoznačnost řešení na hranici dvou elementů je řešena pomocí tzv. numerického toku. Mezi výhody FVM patří možnost použití na oblastech se složitou geometrií, jelikož není omezena jen na strukturované sítě. Dále pak relativní jednoduchost implementace a také kompaktnost této metody: v každém časovém kroku potřebujeme spočítat hodnotu hledané funkce na všech elementech. Pro tento výpočet nám na každém elementu značí znát hodnotu funkce z předcházejícího kroku jen na elementu samotném a hodnoty na hranách elementu. To je velkou výhodou pro efektivní paralelizaci této metody: je-li síť rozdělena mezi více procesorů, mezi jednotlivými procesory není třeba vyměňovat příliš mnoho informací. Podstatnou nevýhodou je obtížnost použití vyšších řádů aproximace. Toho se dá dosáhnout pomocí různých metod rekonstrukce, při kterých je řešení vyššího řádu zkonstruováno s využitím hodnot funkce na sousedních elementech. To je ovšem zvláště na nestrukturovaných sítích technicky obtížné. Navíc tím částečně přicházíme o výše zmiňovanou výhodu kompaktnosti schématu.

Metoda konečných prvků je založena na slabé formulaci daného problému, kde přibližné řešení hledáme jako lineární kombinaci spojitých bázových funkcí. Ty se konstruují pomocí sítě konečných prvků tak, že funkce jsou na každém elementu polynomiální. Stejné funkce obyčejně používáme i jako testovací funkce.

Výhodou metod konečných prvků zůstává vhodnost pro použití na oblastech se složitou geometrií. Oproti FVM je nasazení vyšších řádů aproximace jednoduché a přirozené, je to pouze otázkou volby stupně bázových funkcí. Nevýhodou pak je, že i při použití explicitní diskretizace v čase musíme v každém časovém

kroku řešit soustavu rovnic. To je způsobenou překrývajícími se nosiči bázových a testovacích funkcí a požadavkem na spojitost těchto funkcí na hranici elementů. Navíc, použití FEM se spojitými bázovými funkcemi na problémy z mechaniky tekutin, zejména pokud je řešení nejspojité (např. rázové vlny), může vést k oscilacím nefyzikálního původu v přibližném řešení. Tento problém lze však řešit přidáním stabilizačních členů.

2.2 Nespojitá Galerkinova metoda

Nevýhody obou výše zmíněných metod pomáhá překonat *nespojitá Galerkinova metoda konečných prvků*. Na následujících řádkách stručně popíšeme historii DGM a nastíníme její myšlenku.

Nespojitá Galerkinova metoda byla poprvé použita v [16] pro řešení rovnice transportu neutronů v roce 1973, v oblasti mechaniky tekutin se DGM začala prosazovat v devadesátých letech. V roce 1989 byla DGM prvně aplikována na nelinární rovnice zákonů zachování v [4] a první použití na Navier-Stokesovy rovnice se objevilo v roce 1997 v práci [3]. V následujících letech se objevila další řada prací týkající se DGM a mechaniky tekutin. My v našem dalším textu vycházíme z prací [6], [7], [8] a [9]. Tam lze také nalézt některé podrobnější informace o použitých metodách, které v tomto textu vynecháváme.

Při popisu nespojité Galerkinovy metody opět vyjdeme ze slabé formulace daného problému. Výpočetní oblast, stejně jako u FEM, rozdělíme na jednotlivé elementy. Avšak narozdíl od FEM, u DGM na hledané řešení nekladame podmínku spojitosti mezi elementy. Bázové i testovací funkce volíme jako funkce nespojité, polynomiální vždy na jednom elementu a nulové všude jinde. Podobně jako u FVM, nejednoznačnost řešení mezi elementy obcházíme pomocí použití numerického toku.

Výhod tohoto přístupu je několik: Díky volbě báze jako po částech polynomiálních funkcí můžeme použít vyšší řády aproximace bez problémů, stačí nám zvýšit stupeň bázových funkcí. Protože bázové funkce jsou nenulové vždy jen na jediném elementu, dostaneme kompaktnější schéma, u kterého nejsou rovnice tak vzájemně provázané. Stejně jako u FVM, kompaktnost schématu zajišťuje, že pro nutné výpočty nám stačí znát hodnoty funkce jen na daném elementu a na hranici se sousedními elementy. Při paralelizaci nám stačí mezi procesory posílat jen malé množství informací. Navíc volba vhodného numerického toku nám umožní zohlednit směr toku informací u daného problému. Mezi další výhody patří i slabší podmínky na použitou triangulaci, narozdíl od FEM se v síti mohou vyskytovat tzv. hanging nody a můžeme libovolně kombinovat elementy různých typů - trojúhelníky, čtyřúhelníky i jiné polygony. Stejně tak můžeme kombinovat různé stupně aproximace na různých elementech.

Za tyto příjemné vlastnosti ovšem musíme platit cenu. Tou jsou především zvýšené výpočetní nároky. Protože hledaná funkce je mezi elementy nespojitá, musíme na hraně elementu použít dvakrát více stupňů volnosti než u FEM. Počet neznámých se tak může v závislosti na stupni aproximaci i více než zdvojnásobit.

2.3 Prostorová diskretizace

Zaveďme si nyní označení a některé další pojmy nutné pro prostorovou diskretizaci rovnic.

2.3.1 Triangulace

Rozdělme oblast Ω na konečný počet vzájemně disjunktních, polygonálních elementů K_i.

Definice. *Řekneme, že systém n polygonálních elementů* $K_i, i = 1, \cdots, n$ *je* triangulace *oblasti* Ω*, pokud:*

$$\begin{aligned} a) \quad & \overline{\Omega} = \bigcup_{i=1}^{n} K_i, \\ b) \quad & \mathrm{int}(K_i) \cap \mathrm{int}(K_j) = \emptyset \ \textit{pro}\ i \neq j. \end{aligned}$$

Tuto triangulaci budeme značit $\mathcal{T}_h$*, kde* $h = \max_{K \in \mathcal{T}_h} h_K$ *a* $h_K = diam(K)$.

Zaveďme nyní označení $\mathcal{F}_h$ pro všechny hrany (ve 2D) či stěny (ve 3D) elementů K dané triangulace $\mathcal{T}_h$. Dále značíme

- $\mathcal{F}_h^I$ vnitřní hrany;
- $\mathcal{F}_h^D$ hrany s předepsanou Dirichletovskou podmínkou (pro alespoň jednu složku $\boldsymbol{w}$);
- $\mathcal{F}_h^N$ hrany s předepsanou Neumannovskou podmínkou (pro všechny složky $\boldsymbol{w}$);
- $\mathcal{F}_h^{DN} = \mathcal{F}_h^D \cup \mathcal{F}_h^N$;
- $\mathcal{F}_h^{ID} = \mathcal{F}_h^I \cup \mathcal{F}_h^D$;
- $\mathcal{F}_h^i = \{\Gamma \in \mathcal{F}_h, \quad \Gamma \subset \partial\Omega_i\}$;
- $\mathcal{F}_h^o = \{\Gamma \in \mathcal{F}_h, \quad \Gamma \subset \partial\Omega_o\}$;
- $\mathcal{F}_h^w = \{\Gamma \in \mathcal{F}_h, \quad \Gamma \subset \partial\Omega_w\}$;
- $\mathcal{F}_h^{io} = \mathcal{F}_h^i \cup \mathcal{F}_h^o$;

Připomeneme-li si okrajové podmínky (1.5) - (1.7), pak lze vidět, že platí

$$\begin{aligned} \mathcal{F}_h^D &= \mathcal{F}_h^i \cup \mathcal{F}_h^w, \\ \mathcal{F}_h^N &= \mathcal{F}_h^o. \end{aligned}$$

Navíc zřejmě $\mathcal{F}_h = \mathcal{F}_h^I \cup \mathcal{F}_h^{DN}$.

Pro každou hranu $\Gamma \in \mathcal{F}_h$ definujme jednotkový normálový vektor $\boldsymbol{n}_\Gamma$. Pro $\Gamma \in \mathcal{F}_h^I$ volíme orientaci normálového vektoru libovolnou, ale pevnou. Pro $\Gamma \in \mathcal{F}_h^{DN}$ nechť se orientace $\boldsymbol{n}_\Gamma$ shoduje s orientací vnější normály $\partial\Omega$.

2.3.2 Prostor funkcí po částech polynomiálních

Nespojitá Galerkinova metoda nám nebrání v použití různých stupňů aproximace na různých elementech. Pro každý element $K \in \mathcal{T}_h$ označme lokální polynomiální stupeň p_K a definujme vektor $p = \{p_K, K \in \mathcal{T}_h\}$. Pak můžeme zavést prostor funkcí po částech polynomiálních příslušný vektoru p jako

$$S_{hp} = \{v; v \in L^2(\Omega), v|_K \in P_{p_K} \ \forall K \in \mathcal{T}_h\}.$$

Řešení rovnic budeme hledat v prostoru

$$\boldsymbol{S}_{hp} = (S_{hp})^{d+2}.$$

Pro potřeby dalšího textu ještě definujeme skok a průměr funkce na hranici elementů. Pro každou hranu $\Gamma \in \mathcal{F}_h^I$ existují dva elementy K_p, K_n takové, že $\Gamma \subset K_p \cap K_n$. K_n budeme značit ten z elementů, který leží ve směru normály $\boldsymbol{n}_\Gamma$, K_p ten v opačném směru. Pak značíme

$$\begin{aligned}
v|_\Gamma^{(p)} &\equiv \text{stopa } v|_{K_p} \text{ na } \Gamma;\\
v|_\Gamma^{(n)} &\equiv \text{stopa } v|_{K_n} \text{ na } \Gamma;\\
\langle v \rangle_\Gamma &\equiv \frac{1}{2}(v|_\Gamma^{(p)} + v|_\Gamma^{(n)});\\
[v]_\Gamma &\equiv v|_\Gamma^{(p)} - v|_\Gamma^{(n)}.
\end{aligned}$$

Je zřejmé, že hodnota $[v]_\Gamma$ závisí na orientaci normály $\boldsymbol{n}_\Gamma$, ale hodnota $[v]_\Gamma \boldsymbol{n}_\Gamma$ je na ní nezávislá.

Pro hranu $\Gamma \in \mathcal{F}_h^{DN}$, tedy hranu na hranici oblasti Ω, existuje element K_p takový, že $\Gamma \subset K_p \cap \partial\Omega$. Pro tuto hranu definujeme $\langle v \rangle_\Gamma = [v]_\Gamma = v|_\Gamma^{(p)}$. Symbolem $v|_\Gamma^{(n)}$ označujeme stopu v z vnějšku Ω danou buďto okrajovými podmínkami nebo extrapolací z vnitřku.

Z důvodu přehlednosti budeme v dalším textu u symbolů $\langle\cdot\rangle_\Gamma$ a $[\cdot]_\Gamma$ vypouštět index Γ, bude-li zřejmé, na jaké hraně pracujeme (typicky např. $\int_\Gamma [v] dS$).

2.4 Formulace DGM pro Navier-Stokesovy rovnice

V této sekci se zaměříme na způsob diskretizace Navier-Stokesových rovnic. Některé detaily zde vynecháváme, pro podrobnosti čtenáře opět odkážeme na [7] nebo [8].

Mějme $\boldsymbol{w} \in H^2(\Omega)^{d+2}$. Vynásobíme rovnici (1.1) testovací funkcí $\boldsymbol{\varphi} \in \boldsymbol{S}_{hp}(\Omega)$, zintegrujeme přes $K \in \mathcal{T}_h$ a posčítáme přes všechny $K \in \mathcal{T}_h$. Věnujme se nyní postupně vazkým a nevazkým členům.

2.4.1 Vazké členy

Po vynásobení vazkého členu $\nabla \cdot \boldsymbol{R}(\boldsymbol{w}, \nabla \boldsymbol{w})$ z pravé strany rovnice, zintegrování, použití Greenovy věty a následném užití vztahu (1.10) obdržíme

$$\begin{aligned}
&- \sum_{K \in \mathcal{T}_h} \int_K \sum_{s=1}^{d} \boldsymbol{R}_s(\boldsymbol{w}, \nabla \boldsymbol{w}) \cdot \frac{\partial \boldsymbol{\varphi}}{\partial x_s} dx \\
&+ \sum_{\Gamma \in \mathcal{F}_h^{ID}} \int_\Gamma \sum_{s=1}^{d} \langle \boldsymbol{R}_s(\boldsymbol{w}, \nabla \boldsymbol{w}) \rangle n_s \cdot [\boldsymbol{\varphi}] dS \\
= &\sum_{K \in \mathcal{T}_h} \int_K \sum_{s=1}^{d} \left(\sum_{k=1}^{d} \boldsymbol{K}_{s,k}(\boldsymbol{w}) \frac{\partial \boldsymbol{w}}{\partial x_k} \right) \cdot \frac{\partial \boldsymbol{\varphi}}{\partial x_s} dx \\
&+ \sum_{\Gamma \in \mathcal{F}_h^{ID}} \int_\Gamma \sum_{s=1}^{d} \left\langle \sum_{k=1}^{d} \boldsymbol{K}_{s,k}(\boldsymbol{w}) \frac{\partial \boldsymbol{w}}{\partial x_k} \right\rangle n_s \cdot [\boldsymbol{\varphi}] dS.
\end{aligned} \tag{2.1}$$

Integrály přes $\Gamma \in \mathcal{F}_h^N$ se zde nevyskytují kvůli okrajové podmínce (1.6).

K těmto členům přidáme člen zajišťující stabilitu odvozovaného schématu

$$\eta \sum_{\Gamma \in \mathcal{F}_h^{ID}} \int_\Gamma \sum_{s=1}^{d} \left\langle \sum_{k=1}^{d} \boldsymbol{K}_{s,k}^T(\boldsymbol{w}) \frac{\partial \boldsymbol{\varphi}}{\partial x_k} \right\rangle n_s \cdot [\boldsymbol{w}] dS. \tag{2.2}$$

Abychom toto vyrovnali, na druhou stranu rovnice přičteme

$$\eta \sum_{\Gamma \in \mathcal{F}_h^{D}} \int_\Gamma \sum_{s,k=1}^{d} \boldsymbol{K}_{s,k}^T(\boldsymbol{w}) \frac{\partial \boldsymbol{\varphi}}{\partial x_k} n_s \cdot \boldsymbol{w}_B dS, \tag{2.3}$$

kde vektor $\boldsymbol{w}_B$ je zadán okrajovými podmínkami. Tento člen slouží jako kompenzace stabilizačního členu v tomto smyslu: Je-li funkce $\boldsymbol{w} \in H^2(\Omega)^{d+2}$ řešením (1.1) splňujícím okrajové Dirichletovy podmínky, pak $[\boldsymbol{w}]|_\Gamma = 0$ pro $\Gamma \in \mathcal{F}_h^I$ a

$$\begin{aligned}
&\eta \sum_{\Gamma \in \mathcal{F}_h^{ID}} \int_\Gamma \sum_{s=1}^{d} \left\langle \sum_{k=1}^{d} \boldsymbol{K}_{s,k}^T(\boldsymbol{w}) \frac{\partial \boldsymbol{\varphi}}{\partial x_k} \right\rangle n_s \cdot [\boldsymbol{w}] dS \\
= &\eta \sum_{\Gamma \in \mathcal{F}_h^{D}} \int_\Gamma \sum_{s=1}^{d} \left\langle \sum_{k=1}^{d} \boldsymbol{K}_{s,k}^T(\boldsymbol{w}) \frac{\partial \boldsymbol{\varphi}}{\partial x_k} \right\rangle n_s \cdot [\boldsymbol{w}] dS \\
= &\eta \sum_{\Gamma \in \mathcal{F}_h^{D}} \int_\Gamma \sum_{s,k=1}^{d} \boldsymbol{K}_{s,k}^T(\boldsymbol{w}) \frac{\partial \boldsymbol{\varphi}}{\partial x_k} n_s \cdot \boldsymbol{w}_B dS
\end{aligned}$$

a uvedené členy se rovnají.

Podle hodnoty parametru η rozlišujeme typ schématu. Obvyklé jsou hodnoty

- $\eta = -1$: *non-symetric interior penalty Galerkin*, NIPG,
- $\eta = 1$: *symetric interior penalty Galerkin*, SIPG,
- $\eta = 0$: *incomplete interior penalty Galerkin*, IIPG.

Na základě těchto úvah definujeme pro $\bar{\boldsymbol{w}}_h, \boldsymbol{w}_h, \boldsymbol{\varphi}_h \in \boldsymbol{S}_{hp}$ následující formy. Na základě členů z výrazů (2.1) a (2.2) zavádíme formu $\boldsymbol{a}_h$,

$$\begin{aligned}\boldsymbol{a}_h(\bar{\boldsymbol{w}}_h, \boldsymbol{w}_h, \boldsymbol{\varphi}_h) = &\sum_{K\in\mathcal{T}_h}\int_K \sum_{s=1}^{d}\left(\sum_{k=1}^{d}\boldsymbol{K}_{s,k}(\bar{\boldsymbol{w}}_h)\frac{\partial \boldsymbol{w}_h}{\partial x_k}\right)\cdot\frac{\partial\boldsymbol{\varphi}_h}{\partial x_s}dx\\ &-\sum_{\Gamma\in\mathcal{F}_h^{ID}}\int_\Gamma\sum_{s=1}^{d}\left\langle\sum_{k=1}^{d}\boldsymbol{K}_{s,k}(\bar{\boldsymbol{w}}_h)\frac{\partial \boldsymbol{w}_h}{\partial x_k}\right\rangle n_s\cdot[\boldsymbol{\varphi}_h]dS\\ &-\eta\sum_{\Gamma\in\mathcal{F}_h^{ID}}\int_\Gamma\sum_{s=1}^{d}\left\langle\sum_{k=1}^{d}\boldsymbol{K}^T_{s,k}(\bar{\boldsymbol{w}}_h)\frac{\partial \boldsymbol{\varphi}_h}{\partial x_k}\right\rangle n_s\cdot[\boldsymbol{w}_h]dS\end{aligned}$$

a vzhledem k (2.3) definujeme formu $\tilde{\boldsymbol{a}}_h$,

$$\tilde{\boldsymbol{a}}_h(\bar{\boldsymbol{w}}_h, \boldsymbol{\varphi}_h) = -\eta\sum_{\Gamma\in\mathcal{F}_h^{D}}\int_\Gamma\sum_{s,k=1}^{d}\boldsymbol{K}^T_{s,k}(\bar{\boldsymbol{w}}_h)\frac{\partial\boldsymbol{\varphi}_h}{\partial x_k}n_s\cdot\boldsymbol{w}_B dS.$$

Tyto formy využijeme později v sekci 2.4.4 pro semiimplicitní diskretizaci problému.

2.4.2 Nevazké členy

Postupujeme jako předtím: nevazký člen $\nabla\cdot\vec{\boldsymbol{f}}(\boldsymbol{w})$ násobíme funkcí $\boldsymbol{\varphi}\in\boldsymbol{S}_{hp}(\Omega)$, integrujeme přes $K\in\mathcal{T}_h$ a posčítáme přes všechny $K\in\mathcal{T}_h$. Po použití Greenovy věty pak obdržíme

$$\sum_{K\in\mathcal{T}_h}\int_{\partial K}\sum_{s=1}^{d}\boldsymbol{f}_s(\boldsymbol{w})n_s\cdot\boldsymbol{\varphi}dS-\sum_{K\in\mathcal{T}_h}\int_K\sum_{s=1}^{d}\boldsymbol{f}(\boldsymbol{w})\cdot\frac{\partial\boldsymbol{\varphi}}{\partial x_s}dx.$$

Pro druhý člen použijeme vztah (1.8) a dostaneme

$$\sum_{K\in\mathcal{T}_h}\int_K\sum_{s=1}^{d}\boldsymbol{f}(\boldsymbol{w})\cdot\frac{\partial\boldsymbol{\varphi}}{\partial x_s}dx=\sum_{K\in\mathcal{T}_h}\int_K\sum_{s=1}^{d}\boldsymbol{A}_s(\boldsymbol{w})\boldsymbol{w}\cdot\frac{\partial\boldsymbol{\varphi}}{\partial x_s}dx. \tag{2.4}$$

V prvním členu použijeme techniku známou z metody konečných objemů: aproximaci pomocí numerického toku,

$$\int_\Gamma\sum_{s=1}^{d}\boldsymbol{f}_s(\boldsymbol{w})n_s\cdot[\boldsymbol{\varphi}]dS\approx\int_\Gamma\boldsymbol{H}(\boldsymbol{w}|_\Gamma^{(p)},\boldsymbol{w}|_\Gamma^{(n)},\boldsymbol{n}_\Gamma)\cdot[\boldsymbol{\varphi}]dS.$$

Díky této úpravě můžeme druhý člen upravit následovně:

$$\begin{aligned}&\sum_{K\in\mathcal{T}_h}\int_{\partial K}\sum_{s=1}^{d}\boldsymbol{f}_s(\boldsymbol{w})n_s\cdot\boldsymbol{\varphi}dS=\sum_{\Gamma\in\mathcal{F}_h}\int_\Gamma\sum_{s=1}^{d}\boldsymbol{f}_s(\boldsymbol{w})n_s\cdot[\boldsymbol{\varphi}]dS\\ \approx&\sum_{\Gamma\in\mathcal{F}_h}\int_\Gamma\boldsymbol{H}(\boldsymbol{w}|_\Gamma^{(p)},\boldsymbol{w}|_\Gamma^{(n)},\boldsymbol{n}_\Gamma)\cdot[\boldsymbol{\varphi}]dS.\end{aligned} \tag{2.5}$$

Zde používáme Vijayasundaramův numerický tok,

$$\boldsymbol{H}(\boldsymbol{w}_1, \boldsymbol{w}_2, \boldsymbol{n}) = \boldsymbol{P}^+\left(\frac{\boldsymbol{w}_1 + \boldsymbol{w}_2}{2}, \boldsymbol{n}\right)\boldsymbol{w}_1 + \boldsymbol{P}^-\left(\frac{\boldsymbol{w}_1 + \boldsymbol{w}_2}{2}, \boldsymbol{n}\right)\boldsymbol{w}_2,$$

kde $\boldsymbol{P}^{\pm}$ jsou kladná a záporná část matice

$$\boldsymbol{P}(\boldsymbol{w}, \boldsymbol{n}) = \sum_{s=1}^{d} \boldsymbol{A}_s(\boldsymbol{w}) n_s.$$

Připomeňme, že matice $\boldsymbol{A}_s$ jsou definovány vztahem (1.9).

Speciální ošetření si žádají členy, kde integrujeme přes Γ ležící na hranici oblasti Ω. Pro $\Gamma \in \mathcal{F}_h^{io}$ definujeme

$$\boldsymbol{w}|_\Gamma^{(n)} = LRP(\boldsymbol{w}|_\Gamma^{(p)}, \boldsymbol{w}_D, \boldsymbol{n}_\Gamma), \tag{2.6}$$

kde $LRP(\cdot,\cdot,\cdot)$ značí řešení *lokálního Riemannova problému* s předepsanou okrajovou podmínkou $\boldsymbol{w}_D$.

Pro $\Gamma \in \mathcal{F}_h^{w}$ pak uvažujeme

$$\int_\Gamma \boldsymbol{H}(\boldsymbol{w}|_\Gamma^{(p)}, \boldsymbol{w}|_\Gamma^{(n)}, \boldsymbol{n}_\Gamma) \cdot [\boldsymbol{\varphi}] dS = \int_\Gamma \boldsymbol{F}_W(\boldsymbol{w}, \boldsymbol{n}_\Gamma) \cdot \boldsymbol{\varphi} dS, \tag{2.7}$$

kde

$$\boldsymbol{F}_W(\boldsymbol{w}, \boldsymbol{n}) \equiv (0, pn_1, \cdots, pn_d, 0)^T.$$

Podobně jako pro vazké, i pro nevazké členy zavedeme pro $\bar{\boldsymbol{w}}_h, \boldsymbol{w}_h, \boldsymbol{\varphi}_h \in \boldsymbol{S}_{hp}$ dvě formy. Na základě vztahů (2.4), (2.5) a (2.7) definujeme

$$\begin{aligned}
\boldsymbol{b}_h(\bar{\boldsymbol{w}}_h, \boldsymbol{w}_h, \boldsymbol{\varphi}_h) = &- \sum_{K\in\mathcal{T}_h} \int_K \sum_{s=1}^{d} \boldsymbol{A}_s(\bar{\boldsymbol{w}}_h)\boldsymbol{w}_h \cdot \frac{\partial \boldsymbol{\varphi}_h}{\partial x_s} dx \\
&+ \sum_{\Gamma\in\mathcal{F}_h^I} \int_\Gamma \left(\boldsymbol{P}^+(\langle \bar{\boldsymbol{w}}_h\rangle, \boldsymbol{n})\boldsymbol{w}_h|_\Gamma^{(p)} + \boldsymbol{P}^-(\langle \bar{\boldsymbol{w}}_h\rangle, \boldsymbol{n})\boldsymbol{w}_h|_\Gamma^{(n)}\right) \cdot [\boldsymbol{\varphi}_h] dS \\
&+ \sum_{\Gamma\in\mathcal{F}_h^{io}} \int_\Gamma \left(\boldsymbol{P}^+(\langle \bar{\boldsymbol{w}}_h\rangle, \boldsymbol{n})\boldsymbol{w}_h|_\Gamma^{(p)}\right) \cdot [\boldsymbol{\varphi}_h] dS \\
&+ \sum_{\Gamma\in\mathcal{F}_h^{w}} \int_\Gamma \bar{\boldsymbol{F}}_W(\bar{\boldsymbol{w}}_h, \boldsymbol{w}_h, \boldsymbol{n}) \cdot \boldsymbol{\varphi}_h dS,
\end{aligned}$$

a

$$\tilde{\boldsymbol{b}}_h(\bar{\boldsymbol{w}}_h, \boldsymbol{\varphi}_h) = - \sum_{\Gamma\in\mathcal{F}_h^{io}} \int_\Gamma \left(\boldsymbol{P}^-(\langle \bar{\boldsymbol{w}}_h\rangle, \boldsymbol{n})\bar{\boldsymbol{w}}_h|_\Gamma^{(n)}\right) \cdot [\boldsymbol{\varphi}_h] dS.$$

Zde $\bar{\boldsymbol{F}}_W(\bar{\boldsymbol{w}}_h, \boldsymbol{w}_h, \boldsymbol{n})$ volíme jako linearizaci $\boldsymbol{F}_W(\boldsymbol{w}, \boldsymbol{n})$, pro kterou platí

$$\bar{\boldsymbol{F}}_W(\bar{\boldsymbol{w}}_h, \boldsymbol{w}_h, \boldsymbol{n}) = (\gamma - 1) D\boldsymbol{F}_W(\bar{\boldsymbol{w}}_h, \boldsymbol{n})\boldsymbol{w}_h.$$

Matice $D\boldsymbol{F}_W(\boldsymbol{w}, \boldsymbol{n})$ je tvaru $(d+2)\times(d+2)$, získaná derivací $\boldsymbol{F}_W$. Její tvar lze nalézt např v [8].

2.4.3 Vnitřní a okrajová penalizace

Jako vnitřní a okrajovou penalizaci definujeme pro $\boldsymbol{w}_h, \boldsymbol{\varphi}_h \in \boldsymbol{S}_{hp}$ formy

$$\boldsymbol{J}_h^\sigma(\boldsymbol{w}_h, \boldsymbol{\varphi}_h) = \sum_{\Gamma \in \mathcal{F}_h^{ID}} \int_\Gamma \sigma[\boldsymbol{w}_h] \cdot [\boldsymbol{\varphi}_h] dS$$
$$\tilde{\boldsymbol{J}}_h^\sigma(\boldsymbol{\varphi}_h) = \sum_{\Gamma \in \mathcal{F}_h^{D}} \int_\Gamma \sigma \boldsymbol{w}_B \cdot \boldsymbol{\varphi}_h dS,$$

kde penalizační parametr σ volíme pro $\Gamma \in \mathcal{F}_h^{ID}$ jako

$$\sigma|_\Gamma = \frac{C_W}{diam(\Gamma) Re}$$

a C_W je vhodná konstanta.

Je snadno vidět, že pro $\boldsymbol{w} \in H^2(\Omega)^{d+2}$ řešící (1.1) a splňující Dirichletovy okrajové podmínky platí $[\boldsymbol{w}]|_\Gamma = 0$ na $\Gamma \in \mathcal{F}_h^I$ a platí

$$\begin{aligned}\boldsymbol{J}_h^\sigma(\boldsymbol{w}, \boldsymbol{\varphi}_h) &= \sum_{\Gamma \in \mathcal{F}_h^{ID}} \int_\Gamma \sigma[\boldsymbol{w}] \cdot [\boldsymbol{\varphi}_h] dS = \sum_{\Gamma \in \mathcal{F}_h^{D}} \int_\Gamma \sigma[\boldsymbol{w}] \cdot [\boldsymbol{\varphi}_h] dS \\ &= \sum_{\Gamma \in \mathcal{F}_h^{D}} \int_\Gamma \sigma \boldsymbol{w}_B \cdot \boldsymbol{\varphi}_h dS = \tilde{\boldsymbol{J}}_h^\sigma(\boldsymbol{\varphi}_h)\end{aligned}$$

2.4.4 Semiimplicitní diskretizace

Složme nyní dohromady vše výše odvozné. Definujme pro $\bar{\boldsymbol{w}}_h, \boldsymbol{w}_h, \boldsymbol{\varphi}_h \in \boldsymbol{S}_{hp}$ formy

$$\begin{aligned}\boldsymbol{c}_h(\bar{\boldsymbol{w}}_h, \boldsymbol{w}_h, \boldsymbol{\varphi}_h) &= \boldsymbol{a}_h(\bar{\boldsymbol{w}}_h, \boldsymbol{w}_h, \boldsymbol{\varphi}_h) + \boldsymbol{b}_h(\bar{\boldsymbol{w}}_h, \boldsymbol{w}_h, \boldsymbol{\varphi}_h) + \boldsymbol{J}_h^\sigma(\boldsymbol{w}_h, \boldsymbol{\varphi}_h), \\ \tilde{\boldsymbol{c}}_h(\bar{\boldsymbol{w}}_h, \boldsymbol{\varphi}_h) &= \tilde{\boldsymbol{a}}_h(\bar{\boldsymbol{w}}_h, \boldsymbol{\varphi}_h) + \tilde{\boldsymbol{b}}_h(\bar{\boldsymbol{w}}_h, \boldsymbol{\varphi}_h) + \tilde{\boldsymbol{J}}_h^\sigma(\boldsymbol{\varphi}_h).\end{aligned}$$

V přechozím textu jsme formy $\boldsymbol{c}_h, \bar{\boldsymbol{c}}_h$ odvodili tak, že pro dostatečně hladkou funkci $\boldsymbol{w} : Q_T \to \mathbb{R}^{d+2}$, která řeší Navier-Stokesovy rovnice (1.1), platí

$$\frac{d}{dt}(\boldsymbol{w}, \boldsymbol{\varphi}) + \boldsymbol{c}_h(\boldsymbol{w}, \boldsymbol{w}, \boldsymbol{\varphi}) = \bar{\boldsymbol{c}}_h(\boldsymbol{w}, \boldsymbol{\varphi}) \qquad \forall \boldsymbol{\varphi} \in \boldsymbol{S}_{hp}.$$

Prostorově diskretizovaný problém můžeme tedy zapsat následujícím způsobem:

Definice. *Řekneme, že funkce $\boldsymbol{w}_h \in C^1([0,T]; \boldsymbol{S}_{hp})$ je* semidiskrétním řešením problému proudění stlačitelných tekution*, pokud*

a) $\left(\dfrac{\partial \boldsymbol{w}_h(t)}{\partial t}, \boldsymbol{\varphi}_h\right) + \boldsymbol{c}_h(\boldsymbol{w}_h(t), \boldsymbol{w}_h(t), \boldsymbol{\varphi}_h) = \bar{\boldsymbol{c}}_h(\boldsymbol{w}_h(t), \boldsymbol{\varphi}_h) \qquad \forall \boldsymbol{\varphi}_h \in \boldsymbol{S}_{hp}, \quad \forall t \in (0,T),$

b) $\boldsymbol{w}_h(0) = \boldsymbol{w}_h^0,$

kde $\boldsymbol{w}_h^0 \in \boldsymbol{S}_{hp}$ je $\boldsymbol{S}_{hp}$ aproximace počáteční podmínky.

Otázkou zůstává časová diskretizace této soustavy obyčejných diferenciálních rovnic. Tento problém je typu stiff a tudíž bychom při použití explicitní časové diskretizace museli brát velmi krátké časové kroky a výpočet by se tak stal velmi neefektivním. Nasazení implicitního schématu je ovšem také problematické: v každém časovém kroku bychom museli řešit soustavu nelineárních rovnic, a to je velmi výpočetně drahé. Proto budeme v dalším používat takzvané *semiimplicitní* schéma představené v [6] a [8].

Vraťme se zpět k formě $\boldsymbol{c}_h(\cdot,\cdot,\cdot)$. Protože formy $\boldsymbol{a}_h(\cdot,\cdot,\cdot), \boldsymbol{b}_h(\cdot,\cdot,\cdot)$ a $\boldsymbol{J}_h^\sigma(\cdot,\cdot)$ jsme konstruovali tak, že jsou lineární ve svých druhých a třetích argumentech (popř. prvním a druhém pro $\boldsymbol{J}_h^\sigma$), je i forma $\boldsymbol{c}_h$ lineární ve svém druhém a třetím argumentu. Můžeme tedy postupovat následujícím způsobem: časovou derivaci budeme aproximovat zpětnou Eulerovou metodou a s jednotlivými argumenty $\boldsymbol{c}_h$ budeme zacházet odlišně. První budeme aproximovat explicitně a druhý implicitně.

Mějmě $0 = t_0 < t_1 < \cdots < t_r = T$ rozdělení časového intervalu $(0, T)$ a označme časové kroky $\tau_k := t_k - t_{k-1}$. Pro zjednodušení značíme $\boldsymbol{w}_h^k = \boldsymbol{w}_h(t_k) \in \boldsymbol{S}_{hp}$. Pak můžeme zapsat tvar diskrétního problému.

Definice. *Řekneme, že funkce* $\{\boldsymbol{w}_h^k\}_{k=1}^r$ *jsou* přibližným řešením problému proudění stlačitelných tekutin*, pokud*

$$\begin{aligned}
&a) \quad \boldsymbol{w}_h^k \in \boldsymbol{S}_{hp} \\
&b) \quad \frac{1}{\tau_k}(\boldsymbol{w}_h^k - \boldsymbol{w}_h^{k-1}, \boldsymbol{\varphi}_h) + \boldsymbol{c}_h(\boldsymbol{w}_h^{k-1}, \boldsymbol{w}_h^k, \boldsymbol{\varphi}_h) = \bar{\boldsymbol{c}}_h(\boldsymbol{w}_h^{k-1}, \boldsymbol{\varphi}_h) \\
&\qquad \forall \boldsymbol{\varphi}_h \in \boldsymbol{S}_{hp}, \quad k = 1, \cdots, r, \\
&c) \quad \boldsymbol{w}_h^0 \in \boldsymbol{S}_{hp} \textit{ je aproximace } \boldsymbol{w}^0.
\end{aligned} \tag{2.8}$$

Oproti zcela implicitnímu schématu nyní musíme v každém kroku řešit jen výpočetně levnější soustavu lineárních rovnic, její tvar a způsob řešení si popíšeme v další sekci. Navíc, semiimplicitní schéma je numericky stabilní, vyhneme se tedy omezení na délku časového kroku, jaké bychom měli u plně explicitního schématu.

2.5 Numerické řešení

V této sekci se budeme věnovat numerickému řešení diskrétního problému (2.8).

Označme jako dof dimenzi prostoru $\boldsymbol{S}_{hp}$ a dof_K lokální stupeň volnosti pro element $K \in \mathcal{T}_h$,

$$\mathrm{dof}_K = \frac{d+2}{d!} \prod_{j=1}^{d} (p_K + j).$$

Platí $\mathrm{dof} = \sum_{K \in \mathcal{T}_h} \mathrm{dof}_K$.

Mějme bázi B prostoru $\boldsymbol{S}_{hp}$. Prostor $\boldsymbol{S}_{hp}$ je prostorem funkcí nespojitých, po částech polynomiálních, a bázi B můžeme složit z funkcí nenulových vždy na jediném elementu, tedy

$$B = \bigcup_{K \in \mathcal{T}_h} \{\boldsymbol{\psi}_j^K\}_{j=1}^{\mathrm{dof}_K},$$

kde $\boldsymbol{\psi}_j^K$ jsou lineárně nezávislé funkce takové, že $\mathrm{supp}(\boldsymbol{\psi}_j^K) \subseteq K$. Pro jednoduchost zápisu v dalším textu si seřaďme všechny funkce $\boldsymbol{\psi}_j^K$ tak, abychom mohli

psát

$$B = \bigcup_{K \in \mathcal{T}_h} \{\boldsymbol{\psi}_j^K\}_{j=1}^{\mathrm{dof}_K} = \{\boldsymbol{\psi}_i\}_{i=1}^{\mathrm{dof}}.$$

Pak každou funkci $\boldsymbol{w}_h^k \in S_{hp}$ můžeme vyjádřit jako lineární kombinaci bázových funkcí $\boldsymbol{\psi}_i(x)$,

$$\boldsymbol{w}_h^k(x) = \sum_{i=1}^{\mathrm{dof}} \xi_i^k \boldsymbol{\psi}_i(x), \qquad x \in \Omega, k = 1, \cdots, r.$$

Označme $\boldsymbol{\xi}^k \equiv (\xi_1^k, \xi_2^k, \cdots, \xi_{\mathrm{dof}}^k) \in \mathbb{R}^{\mathrm{dof}}$. Pak problém (2.8) můžeme zapsat ve tvaru

$$\left(\frac{1}{\tau_k}\boldsymbol{M} + \boldsymbol{C}(\boldsymbol{\xi}^{k-1})\right)\boldsymbol{\xi}^k = \frac{1}{\tau_k}\boldsymbol{m}(\boldsymbol{\xi}^{k-1}) + \boldsymbol{q}(\boldsymbol{\xi}^{k-1}), \quad k = 1, \cdots, r. \tag{2.9}$$

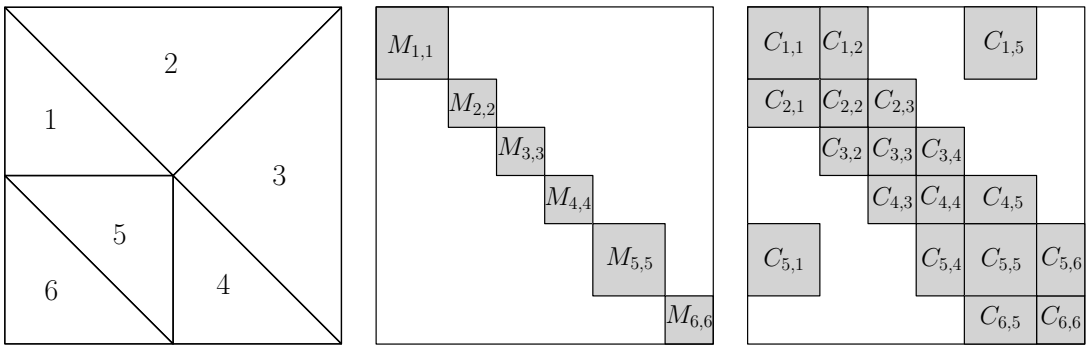
Zde značíme $\boldsymbol{M}$ blokově diagonální matice hmotnosti,

$$\begin{aligned}
&\boldsymbol{M} = \mathrm{diag}\{\boldsymbol{M}_{K,K}, K \in \mathcal{T}_h\},\\
&\boldsymbol{M}_{K,K} = \{M_K^{i,j}\}_{i,j=1}^{\mathrm{dof}_K},\\
&M_K^{i,j} = \int_K \boldsymbol{\psi}_i^K \cdot \boldsymbol{\psi}_j^K dx,
\end{aligned}$$

jejíž diagonální bloky $\boldsymbol{M}_{K,K}$ mají mají velikost $\mathrm{dof}_K \times \mathrm{dof}_K$. Matice $\boldsymbol{C}$ odpovídá formě $\boldsymbol{c}_h$,

$$\begin{aligned}
&\boldsymbol{C}(\boldsymbol{\xi}^{k-1}) = \{\boldsymbol{C}_{K_m,K_n}(\boldsymbol{\xi}^{k-1})\}_{K_m,K_n \in \mathcal{T}_h},\\
&\boldsymbol{C}_{K_m,K_n}(\boldsymbol{\xi}^{k-1}) = \{C_{K_m,K_n}^{i,j}(\boldsymbol{\xi}^{k-1})\}_{i=1,\cdots,\mathrm{dof}_{K_m},j=1,\cdots,\mathrm{dof}_{K_n}},\\
&C_{K_m,K_n}^{i,j}(\boldsymbol{\xi}^{k-1}) = \boldsymbol{c}_h(\boldsymbol{w}_h^{k-1}, \boldsymbol{\psi}_j^{K_n}, \boldsymbol{\psi}_i^{K_m}).
\end{aligned}$$

Matice $\boldsymbol{C}$ se skládá z bloků $\boldsymbol{C}_{K_m,K_n}$ velikosti $\mathrm{dof}_{K_m} \times \mathrm{dof}_{K_n}$. Protože bázové funkce $\boldsymbol{\psi}_j^K$ jsou nenulové jen na elementu K, je zřejmé, že blok $\boldsymbol{C}_{K_m,K_n}$ je nenulový pouze tehdy, pokud elementy K_m a K_n mají společnou hranu nebo $K_m = K_n$. Tvar matic pro jednoduchou triangulaci $\boldsymbol{M}$ a $\boldsymbol{C}$ je znázorněn na obrázku 2.1.



(a) Triangulace (b) Matice $\boldsymbol{M}$ (c) Matice $\boldsymbol{C}$

Obrázek 2.1: Tvar matic $\boldsymbol{M}$ a $\boldsymbol{C}$. Na elementech 1 a 5 je použit vyšší stupeň aproximace.

Vektor $\boldsymbol{m}(\boldsymbol{\xi}^{k-1}) \in \mathbb{R}^{\mathrm{dof}}$ odpovídá zbylé části po aproximaci časové derivace,

$$\begin{aligned}\boldsymbol{m}(\boldsymbol{\xi}^{k-1}) &= \boldsymbol{M}\boldsymbol{\xi}^{k-1} = \{m_K^i(\boldsymbol{\xi}^{k-1})\}_{K\in\mathcal{T}_h}^{i=1,\cdots,\mathrm{dof}_K},\\ m_K^i(\boldsymbol{\xi}^{k-1}) &= \left(\boldsymbol{w}_h^{k-1}, \boldsymbol{\psi}_i^K\right),\end{aligned}$$

a vektor $\boldsymbol{q}(\boldsymbol{\xi}^{k-1}) \in \mathbb{R}^{\mathrm{dof}}$ odpovídá formě $\tilde{\boldsymbol{c}}_h$,

$$\begin{aligned}\boldsymbol{q}(\boldsymbol{\xi}^{k-1}) &= \{q_K^i(\boldsymbol{\xi}^{k-1})\}_{K\in\mathcal{T}_h}^{i=1,\cdots,\mathrm{dof}_K},\\ q_K^i(\boldsymbol{\xi}^{k-1}) &= \tilde{\boldsymbol{c}}_h\left(\boldsymbol{w}_h^{k-1}, \boldsymbol{\psi}_i^K\right).\end{aligned}$$

Označme nyní

$$\begin{aligned}\boldsymbol{A}_k &= \frac{1}{\tau_k}\boldsymbol{M} + \boldsymbol{C}(\boldsymbol{\xi}^{k-1}),\\ \boldsymbol{d}_k &= \frac{1}{\tau_k}\boldsymbol{m}(\boldsymbol{\xi}^{k-1}) + \boldsymbol{q}(\boldsymbol{\xi}^{k-1}),\end{aligned}$$

a napišme algoritmus pro řešení problému (2.8):

Algoritmus 1.

1. *Mějme* $\boldsymbol{\xi}^0$ *tak, že* $\boldsymbol{w}_h^0 = \sum_{i=1}^{\mathrm{dof}} \xi_i^0 \boldsymbol{\psi}_i(x)$.
2. *Pro* $k = 1, \cdots, r$
   (a) *Určeme* τ_k.
   (b) *Spočítejme* $\boldsymbol{A}_k$ *a* $\boldsymbol{d}_k$.
   (c) *Vyřešme soustavu lineárních rovnic* $\boldsymbol{A}_k\boldsymbol{\xi}_k = \boldsymbol{d}_k$.
3. $\boldsymbol{\xi} := \boldsymbol{\xi}^r$.

Vzhledem k blokovému tvaru matic $\boldsymbol{M}$ i $\boldsymbol{C}$ lze vidět, že i matice $\boldsymbol{A}_k$ mají blokovou strukturu. Navíc při vhodném seřazení jednotlivých elementů je většina bloků matice umístěna blízko diagonály, což pro vhodné algoritmy usnadňuje řešení soustavy lineárních rovnic.

Zde zůstává ještě několik otevřených otázek týkajících se algoritmu: kolik kroků r musíme provést (tedy za jakých okolností ukončit výpočet), jakým způsobem volit délku časového kroku τ_k a jak řešit soustavy lineárních rovnic. Tyto problémy byly detailně zkoumány v [9] a my se jimi nebudeme zde zabývat. K některým výše zmíněným bodům se však ještě vrátíme v dalších kapitole, až budeme diskutovat paralelní algoritmus.

2.6 Řešení stacionárního problému

V předešlém textu jsme popisovali algoritmus pro řešení nestacionárních problémů. Ten lze úspěšně použít i pro řešení problémů stacionárních. Uvažme diskretizovaný stacionární problém: najít $\boldsymbol{w}_h \in \boldsymbol{S}_{hp}$ takové, že

$$\boldsymbol{c}_h(\boldsymbol{w}_h, \boldsymbol{w}_h, \boldsymbol{\varphi}_h) = \bar{\boldsymbol{c}}_h(\boldsymbol{w}_h, \boldsymbol{\varphi}_h) \quad \forall \boldsymbol{\varphi}_h \in \boldsymbol{S}_{hp}.$$

To můžeme zapsat jako systém nelineárních rovnic pro $\boldsymbol{\xi} \in \mathbb{R}^{\mathrm{dof}}$,

$$\boldsymbol{C}(\boldsymbol{\xi})\boldsymbol{\xi} = \boldsymbol{q}(\boldsymbol{\xi}). \tag{2.10}$$

Řešení tohoto problému budeme hledat pomocí nestacionární formulace. Mějme počáteční odhad řešení $\boldsymbol{\xi}_0$ a pro malé τ_k řešme problém

$$\left(\frac{1}{\tau_k}\boldsymbol{M} + \boldsymbol{C}(\boldsymbol{\xi}^{k-1})\right)\boldsymbol{\xi}^k = \frac{1}{\tau_k}\boldsymbol{m}(\boldsymbol{\xi}^{k-1}) + \boldsymbol{q}(\boldsymbol{\xi}^{k-1}), \quad k = 1, 2, 3 \cdots . \tag{2.11}$$

Jak se postupně zpřesňuje řešení $\boldsymbol{\xi}_k$, zvětšujeme krok τ_k, abychom tak pro $\tau_k \to \infty$ dostali iterační schéma pro nalezení řešení stacionárního problému

$$\boldsymbol{C}(\boldsymbol{\xi}^{k-1})\boldsymbol{\xi}^k = \boldsymbol{q}(\boldsymbol{\xi}^{k-1}). \tag{2.12}$$

Tento postup používáme proto, že kdybychom problém (2.10) řešili iterativní metodou (2.12) s nepřesným počátečním odhadem, snadno bychom dostali nefyzikální řešení. Takto lze napřed přibližné řešení dostatečně zpřesnit při malých časových krocích a ty pak postupně zvětšovat.

Jako zastavovací kritérium používáme podmínku SSres $\leq TOL$, kde SSres je tzv. *steady state residuum*, definované jako

$$\mathrm{SSres} := \frac{||\boldsymbol{C}(\boldsymbol{\xi}_k)\boldsymbol{\xi}_k - \boldsymbol{q}(\boldsymbol{\xi}_k)||_{l^2}}{||\boldsymbol{C}(\boldsymbol{\xi}_0)\boldsymbol{\xi}_0 - \boldsymbol{q}(\boldsymbol{\xi}_0)||_{l^2}} \tag{2.13}$$

a TOL je daná tolerance.

Kapitola 3

Paralelizace

V této kapitole se budeme zabývat tím, jakým způsobem jsme paralelizovali program. Samotný zdrojový kód paralelizovaného programu je k práci přiložen, komentované ukázky z něj jsou k nalezení v příloze A, návod k instalaci je v příloze B.

Před nadcházejícím výkladem si napřed ujasníme terminologii týkají se paralelního počítání. Systémy s distribuovanou pamětí, kterých se práce dotýká, jsou složeny z mnoha procesorů spojených vysokorychlostní komunikační sítí. Dnes se obvykle setkáváme s vícejádrovými procesory, které obsahují několik výpočetních jader sdílejících stejnou lokální paměť s přímým přístupem (RAM). My však budeme pro zjednodušení výkladu slovem *procesor* označovat samostatnou výpočetní jednotku s vlastní lokální pamětí, a termínu *jádro* se budeme vyhýbat. Říkáme-li, že data jsou uložena na procesoru, myslíme tím, že jsou uložena v paměti příslušné zmíněnému procesoru. *Cluster* je skupina procesorů s pamětí (*uzlů* či *nodů*), schopných vzájemné komunikace každého s každým.

Paralelizace programu má dvě zásadní výhody pro běh programu. Zaprvé je to zrychlení doby běhu, neboť potřebná práce je rozdělena mezi více procesorů, každý tak dostane menší množství práce, se kterým je dříve hotov. Zadruhé, a to je neméně podstatné, paralelizace umožňuje počítání i mnohem větších úloh, než je vůbec na jediné výpočetní jednotce možné. Velké úlohy totiž vyžadují velké množství RAM, které nemusí být na počítači dostupné. Místo obtížného a drahého navyšování výkonu a paměti na jednom počítači je tak snazší přidat další výpočetní jednotky s vlastní pamětí. Tím objem celkové paměti vzroste a my si můžeme dovolit výpočet paměťově náročnějších úloh.

Z tohoto popisu vyplývá, co si od paralelizace programu slibujeme: dobrou škálovatelnost ve smyslu časové i paměťové náročnosti. Na to, jak se podařilo tyto cíle naplnit, se podíváme v další kapitole.

Hlavní myšlenka paralelizovaného programu je následující: rozdělíme výpočetní síť na p částí, kde p je počet procesorů, které máme k dispozici. Každou část sítě přiřadíme jednomu procesoru. Všechny procesory v každém časovém kroku spočítají jednotlivé prvky matice $\boldsymbol{A}_k$ a vektoru $\boldsymbol{d}_k$ příslušné elementům z dané části sítě, a poté vhodným paralelním řešičem vyřeší vzniklou soustavu lineárních rovnic.

Z tohoto stručného popisu plynou hlavní tři témata, kterými se budeme zabývat v následujících sekcích:

- Rozdělení sítě;

- Výpočet prvků matice $\boldsymbol{A}_k$ a vektoru $\boldsymbol{d}_k$;
- Paralelní výpočet soustavy lineárních rovnic.

Pro komunikaci mezi procesory jsme použili technologii *MPI - Message Passing Interface*, viz [15]. MPI je rozhraní pro zasílání zpráv na systémech s distribuovanou pamětí a v současné době je de facto standardem pro meziprocesorovou komunikaci na počítačových clusterech.

3.1 Značení

Stejně jako dříve značíme $\mathcal{T}_h$ triangulaci oblasti Ω, $\mathcal{T}_h = \{K_i, i = 1, \cdots, n\}$, n počet elementů této triangulace a K_i jednotlivé elementy. Mějme k dispozici p procesorů. Budeme se držet v oboru paralelního programování zavedeného značení a jednotlivé procesory číslovat $0, 1, \cdots, p-1$.

Uvažujme rozdělení sítě na p částí příslušných jednotlivým procesorům. Pak značíme

- $\mathcal{T}_h^i, \quad i = 0, \cdots, p-1, \quad$ i-tou část sítě,
- $n_i, \quad i = 0, \cdots, p-1, \quad$ počet elementů v $\mathcal{T}_h^i$.

Přitom platí $\mathcal{T}_h = \bigcup_{i=0}^{p-1} \mathcal{T}_h^i$ a $n = \sum_{i=0}^{p-1} n_i$.

3.2 Rozdělení sítě

Vyložme si napřed, co od rozdělení sítě očekáváme. Máme dva základní požadavky:

1. Potřebujeme, aby všechny procesory měly přibližně stejně velký objem práce během výpočtu prvků matice a následném řešení soustavy lineárních rovnic. V opačném případě by procesory s menším množstvím práce musely v každém časovém kroku zbytečně čekat na procesory s větším množstvím práce. Pokud je na všech elementech použita aproximace stejného stupně, pak tohoto vyvážení můžeme docílit požadavkem na přibližně stejný počet elementů v každé části sítě, ideálně $n_i = n/p$.

   Pokud bychom používali různé stupně aproximace na různých elementech, formulujeme požadavek na vyváženost jinak. Počet nenulových prvků matic $\boldsymbol{A}_k$ roste přibližně se součtem druhých mocnin stupňů volnosti na elementech. Stejně roste i časová náročnost výpočtu, jak potvrzují numerické experimenty v sekci 4. Proto požadujeme, aby součet druhých mocnin stupňů volnosti byl na všech částech sítě přibližně stejný, tedy ideálně

   $$\sum_{K \in \mathcal{T}_h^i} (\mathrm{dof}_K)^2 = \frac{\sum_{K \in \mathcal{T}_h} (\mathrm{dof}_K)^2}{p}.$$

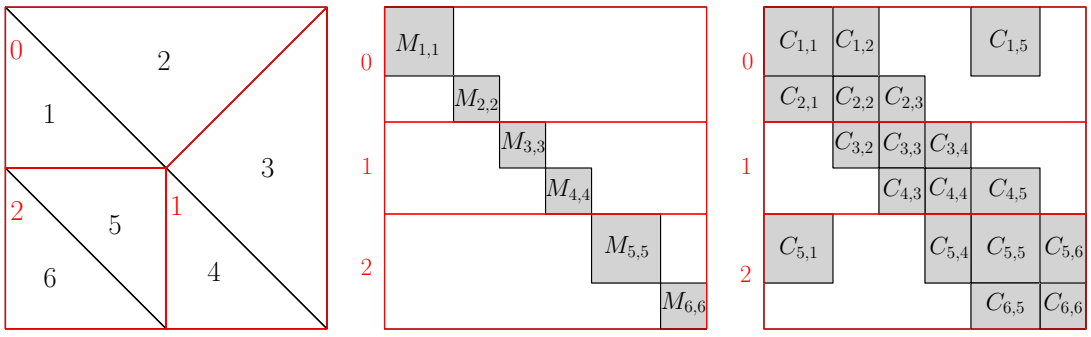
2. Protože meziprocesorová komunikace je časově drahá, chceme minimalizovat objem dat, které si musí procesory vyměňovat v každém časovém kroku. Jak si popíšeme v další sekci, pro každý element $K \in \mathcal{T}_h$ budeme počítat

maticové bloky $\boldsymbol{M}_{K,K}$ a $\boldsymbol{C}_{K,K'}$ pro všechny sousedy K' elementu K. Pro výpočet těchto bloků příslušných elementu K tedy potřebujeme znát hodnoty funkce $\boldsymbol{w}_h^k$ na elementu K a sousedech K'. Sousední elementy K' ale již mohou ležet v jiné části sítě a proto musíme hodnoty funkce $\boldsymbol{w}_h^k$ na všech elementech, které leží na hranici dvou různých částí sítě, v každém kroku mezi procesory vyměňovat. Naším cílem tedy bude najít takové rozdělení sítě, aby počet elementů ležící na hranici dvou částí sítě byl co nejmenší a my tak museli posílat co nejméně dat.

Samozřejmě očekáváme také rozumnou rychlost výpočtu dělení. Vzhledem ke komplexnosti problému jsme pro rozdělení sítě použili volně dostupný software METIS [13]. Tento balík určený k dělení grafů a sítí používá algoritmy založené na víceúrovňových schématech pro dělení grafů, podrobnější informace lze nalézt v [12]. Vytvořená dělení sítě pomocí tohoto balíku se ukázala jako kvalitní a splňující oba požadavky vyslovené výše. Konkrétní ukázky rozdělení sítí jsou prezentovány v kapitole 4.3.

3.3 Výpočet členů

V každém časovém kroku potřebujeme spočítat členy matice $\boldsymbol{A}_k$ a vektoru $\boldsymbol{d}_k$, tvar této matice a vektoru jsme popsali v sekci 2.5. Každému procesoru i přiřadíme řádky odpovídající elementům z části sítě $\mathcal{T}_h^i$. Tedy pro každý element K takový, že $K \in \mathcal{T}_h^i$, bude procesor i počítat diagonální blok $\boldsymbol{C}_{K,K}$ a bloky $\boldsymbol{C}_{K,K'}$ pro všechny elementy K', které sousedí s K. K těm pak přičte (vhodně přinásobný) diagonální blok matice hmotnosti $\boldsymbol{M}_{K,K}$. Tento blok je konstantní v průběhu celého výpočtu a proto ho lze spočíst jen jednou na začátku. Ukázka rozdělení jednoduché sítě a zobrazení částí matic příslušných jednotlivým procesorům je na obr. 3.1.



(a) Triangulace (b) Matice $\boldsymbol{M}$ (c) Matice $\boldsymbol{C}$

Obrázek 3.1: Rozdělení sítě z obr. 2.1 na tři části

Dále musí procesor i spočítat všechny příslušné části vektoru $\boldsymbol{d}_k$, tedy prvky $\{m_K^j(\boldsymbol{\xi}^{k-1})\}_{j=1}^{\mathrm{dof}_K}$ a $\{q_K^j(\boldsymbol{\xi}^{k-1})\}_{j=1}^{\mathrm{dof}_K}$ pro všechny $K \in \mathcal{T}_h^i$.

Pro výpočet mimodiagonálních bloků matice $\boldsymbol{C}$ potřebujeme znát hodnoty funkce $\boldsymbol{w}_h^k$ na všech sousedních elementech, některé z nich ovšem mohou být v části sítě příslušné jinému procesoru. Protože na začátku každého časové kroku máme na procesoru k dispozici jen hodnoty funkce $\boldsymbol{w}_h^k$ na elementech z naší

části sítě, musíme všechny hodnoty ze sousedních částí napřed získat meziprocesorovou komunikací. Aby nás časově náročná komunikace zbytečně nezdržovala, využíváme v programu možnosti překrytí komunikace a výpočtů, a to následujícím způsobem:

1. Každý procesor začne odesílat potřebná data sousedům a zároveň začne přijímat data od sousedů.
2. Zatímco probíhá komunikace, procesor spočte veškeré objemové integrály na elementech v jeho části síte, které jsou potřebné pro výpočet. Pro tyto totiž nejsou potřeba data ze sousedních elementů.
3. Procesor dokončí přijímání potřebných dat od sousedů.
4. Procesor spočte zbylé integrály přes hrany, sestaví příslušnou část matice $\boldsymbol{A}_k$ a spočte příslušnou část vektoru $\boldsymbol{d}_k$.

Pomocí tohoto postupu zabráníme situacím, kdy procesor čeká na příjem dat od jiného zaneprázdněného procesoru, a sám nic nepočítá, přestože by mohl. Mezi začátkem a koncem přijímání dat od sousedních procesorů bychom mohli počítat i ty integrály přes hrany elementů, které neleží na rozhraní dvou částí sítě, protože pro ně již máme veškerá potřebná data. To se ale ukázalo jako nepotřebné, neboť čas nutný pro spočítání objemových integrálů bohatě postačuje pro výměnu dat mezi sousedními procesory.

3.4 Řešení soustavy

Posledním úkolem v k-tém časovém kroku je paralelně vyřešit soustavu lineárních rovnic $\boldsymbol{A}_k\boldsymbol{\xi}_k = \boldsymbol{d}_k$. Prvky matice $\boldsymbol{A}_k$ i vektoru $\boldsymbol{d}_k$ již máme na jednotlivých procesorech spočteny z předchozího kroku. Pro samotný výpočet soustavy jsme použili již hotové řešení, knihovnu PETSc (Portable, Extensible Toolkit for Scientific Computation, [2]).

PETSc je knihovna zaměřená na programování aplikací pro řešení parciálních diferenciálních rovnic v paralelním prostředí. Umožňuje práci s vektory a maticemi, řešení soustav lineárních i nelineárních rovnic. Pro komunikaci mezi procesory používá dříve zmíněný standard MPI.

PETSc nabízí velké množství nepřímých řešičů soustav lineárních rovnic založených na Krylovových podprostorech. My jsme v našem programu vyzkoušeli metody GMRES (*Generalized minimal residual method*) a BiCGStab (*Biconjugate gradient stabilized method*) spolu s Jacobiho a ASM (*Additive Schwarz Method*) předpodmiňovačem. Srovnání jejich časové a paměťové náročnosti jsme zařadili do další kapitoly.

3.4.1 Paralelní předpodmiňovače

Řešení soustavy lineárních rovnic

$$\boldsymbol{A}\boldsymbol{u} = \boldsymbol{b}$$

se obvykle skládá ze dvou kroků. Prvním je aplikace předpodmiňovače, tedy matice $\boldsymbol{T}$ takové, že $\boldsymbol{T} \approx \boldsymbol{A}^{-1}$ a přitom jí lze (na rozdíl od $\boldsymbol{A}^{-1}$) rychle a relativně

jednoduše spočítat. Druhým krokem je potom výpočet předpodmíněné soustavy s nízkým číslem podmíněnosti

$$\boldsymbol{TAu} = \boldsymbol{Tb}.$$

Pro samotný výpočet soustavy je vhodné použít nějakou z iteračních metod založených na Krylovových podprostorech.

Pracujeme-li v paralelním prostředí, je mezi těmito dvěma kroky podstatný rozdíl. Implementace Krylovovských metod obvykle nevyžadují znalost matice $\boldsymbol{A}$, v jejich algoritmech se tato matice používá jen v operacích typu součinu matice s nějakým vektorem. Tyto metody jsou tedy zcela nezávislé na způsobu uložení matice, je-li dostupná funkce pro součin matice s vektorem. Proto i při paralelním výpočtu, kdy jsou části matice uložené na různých výpočetních jednotkách, můžeme tyto metody pro výpočet soustavy použít beze změn. Jediné, co potřebujeme, je efektivní paralelní implementace součinu matice s vektorem.

Některé obvykle užívané předpodmiňovače ovšem pro aproximaci inverzní matice $\boldsymbol{A}^{-1}$ potřebují znát všechny prvky matice a jejich implementace pro běh v paralelním prostředí by byla velmi neefektivní. Proto používáme předpodmiňovače vyvinuté speciálně pro paralelní běh.

Jednodušší ze dvou, které popíšeme, je *blokový Jacobiho předpodmiňovač*. Pro jeho popis zaveďme následující značení. Předpokládejme, že máme již náš problém rozdělený na p procesorů. Označme $D_i,\ i = 0, \cdots, p-1$, indexové množiny takové, že D_i obsahují čísla řádků matice uložené na procesoru i. Na stejném procesoru jsou uloženy též i části vektoru $\boldsymbol{u}$ příslušné indexům z D_i, tyto označme $\boldsymbol{u}_{D_i}$.

Fyzikální interpretace je následující: množiny D_i popisují stupně volnosti na elementech z části sítě $\mathcal{T}_h^i$. Veškerá data patřící k této části sítě jsou uložena na procesoru i.

Označme ještě m velikost matice $\boldsymbol{A}$ a m_i velikost množiny D_i.

Pro následující popis zaveďme matice $\boldsymbol{R}_i^0$. To jsou obdélníkové matice tvaru $m_i \times m$ obsahující pouze nulové a jednotkové prvky a takové, že z vektoru neznámých separují koeficienty náležící procesoru i, tedy

$$\boldsymbol{u}_{D_i} = \boldsymbol{R}_i^0 \boldsymbol{u}.$$

Pokud jsou koeficienty náležící jednomu procesoru seřazeny za sebou, pak mají $\boldsymbol{R}_i^0$ tvar

$$\boldsymbol{R}_i^0 = [0 \cdots 0\ I\ 0 \cdots 0].$$

Nyní můžeme definovat

$$\boldsymbol{A}_i = \boldsymbol{R}_i^0 \boldsymbol{A} (\boldsymbol{R}_i^0)^T,$$

což je vyseparovaný čtvercový blok matice $\boldsymbol{A}$ příslušný části D_i. Protože členy matice $\boldsymbol{A}_i$ jsou uloženy na jednom procesoru i, můžeme již lokálně spočítat inverzi $\boldsymbol{A}_i^{-1}$, či spíše její aproximaci $\tilde{\boldsymbol{A}}_i^{-1}$. Tu již počítáme pomocí předpodmiňovače určeného pro běh na jednom procesoru, např. ILU(k). Blokový Jacobiho předpodmiňovač pak dostaneme jako součet

$$\boldsymbol{T} = \sum_{i=0}^{p-1} (\boldsymbol{R}_i^0)^T \tilde{\boldsymbol{A}}_i^{-1} \boldsymbol{R}_i^0.$$

Matice $(\boldsymbol{R}_i^0)^T \tilde{\boldsymbol{A}}_i^{-1} \boldsymbol{R}_i^0$ jsou opět tvaru $m \times m$ a s jediným nenulovým blokem $\tilde{\boldsymbol{A}}_i^{-1}$.

Blokový Jacobiho předpodmiňovač tedy spočte inverze jednotlivých bloků bez jakékoliv komunikace mezi procesory. V tom je jeho výhoda i nevýhoda - ušetří se na náročné komunikaci, jednotlivé inverzní bloky jsou však spočteny zcela odtrženě od ostatních, čímž se zanedbává velké množství informace.

Podotkněme, že matice $\boldsymbol{T}$ nám zde slouží pouze pro výklad, v programu samotném nemusí být vůbec formována. Namísto toho stačí funkce pro součin této matice s vektorem.

Zobecněním blokového Jacobiho předpodmiňovače je *aditivní Schwarzův předpodmiňovač* (zkracován jako *ASM - additive Schwarz method*). Pro jeho popis induktivně zavedeme indexové množiny D_i^δ pro celočíselné $\delta > 0$. V množině D_i^1 jsou indexy odpovídající prvkům uložených na procesoru i a nebo jejich sousedům, tedy

$$D_i^1 = \{j,\ j \in D_i \vee a_{jk} \neq 0 \text{ pro nějaké } k \in D_i\}.$$

Množiny D_i^δ pro δ vyšší jak 1 přidávají i prvky sousedící vzdáleněji, popsat je můžeme induktivně

$$D_i^{j+1} = (D_i^j)^1.$$

Poznamenáme, že ač je zde sousednost definována přes nenulové prvky matice $\boldsymbol{A}$, v našem případě to přesně odpovídá sousednosti dvou elementů ve fyzikálním smyslu - viz popis tvaru matice v sekci 2.5.

Podobně jako u popisu blokového Jacobiho předpodmiňovače zavádíme obdélníkové matice $\boldsymbol{R}_i^\delta$ tak, že z vektoru neznámých tyto matice separují prvky odpovídající indexovým množinám D_i^δ, tedy

$$\boldsymbol{u}_{D_i^\delta} = \boldsymbol{R}_i^\delta \boldsymbol{u}.$$

I dále je postup obdobný: konstruujeme matice

$$\boldsymbol{A}_i^\delta = \boldsymbol{R}_i^\delta \boldsymbol{A} (\boldsymbol{R}_i^\delta)^T,$$

a k těmto podmaticím již počítáme aproximace inverzí $(\tilde{\boldsymbol{A}}_i^\delta)^{-1}$ stejně jako dříve, stejně také sestrojíme samotný předpodmiňovač

$$\boldsymbol{T}^\delta = \sum_{i=0}^{p-1} (\boldsymbol{R}_i^\delta)^T (\tilde{\boldsymbol{A}}_i^\delta)^{-1} \boldsymbol{R}_i^\delta.$$

Rozdíl je ovšem ten, že nyní je (vzhledem k tomu, že matice $\boldsymbol{A}_i^\delta$ obsahují i prvky uložené na jiných procesorech) potřeba i jistá meziprocesorová komunikace. Výhodou však je, že dochází k jistému toku informací mezi jednotlivými bloky. Jak velkému, to záleží na velikosti překryvu δ.

3.5 Paralelní algoritmus

Na závěr této kapitoly si představme paralelní verzi algoritmu 1 ze sekce 2.5. V tomto algoritmu jsou kroky, které vyžadují komunikaci mezi procesory, označeny symbolem *.

Algoritmus 2.

1. *Rozdělme síť na p částí.*

2. *Mějme* $\boldsymbol{\xi}^0$ *tak, že* $\boldsymbol{w}_h^0 = \sum_{i=1}^{\mathrm{dof}} \xi_i^0 \boldsymbol{\psi}_i(x)$*, a nechť jsou části* $\boldsymbol{\xi}^0$ *přítomné na příslušných procesorech.*

3. *Pro každý procesor a pro* $k = 1, \cdots, r$

   (a) **Určeme* τ_k*.*

   (b) **Začněme rozesílat data sousedům.*

   (c) *Spočtěme objemové integrály příslušné elementům z části sítě.*

   (d) **Ukončeme přijímání dat od sousedů.*

   (e) *Spočítejme zbylé integrály přes hrany a zkonstruujme části* $\boldsymbol{A}_k$ *a* $\boldsymbol{d}_k$*.*

   (f) **Vyřešme soustavu lineárních rovnic* $\boldsymbol{A}_k \boldsymbol{\xi}_k = \boldsymbol{d}_k$*.*

4. $\boldsymbol{\xi} := \boldsymbol{\xi}^r$*.*

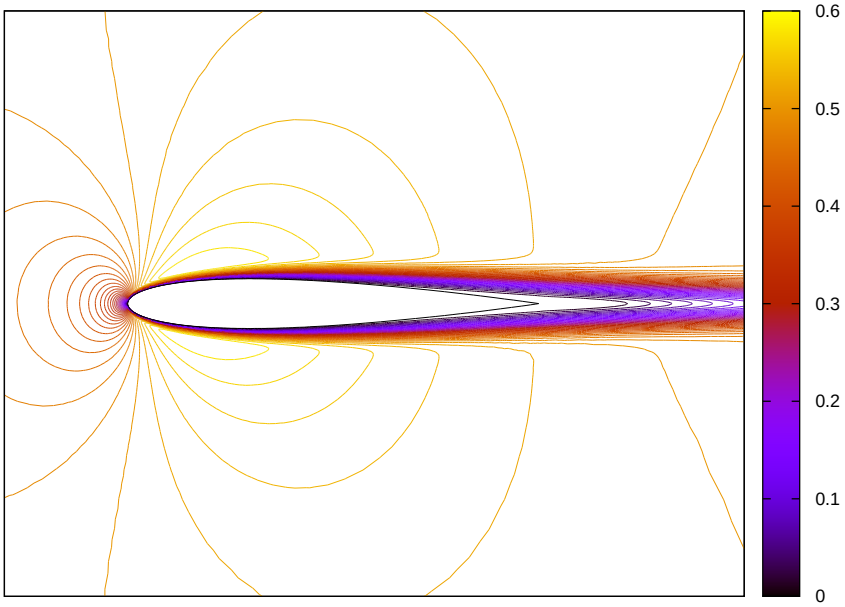
Kapitola 4

Numerické výsledky

4.1 Popis testovací úlohy

Věnujme se nyní otázce škálování paralelně běžícího programu. Veškeré testy jsme spouštěli na výpočetním clusteru Sněhurka matematické sekce MFF UK[1]. Ten se skládá ze strojů s 24 procesory Intel Core i7 běžícími na frekvenci 2266.745 MHz. Každý z nich má 12 GB RAM a obsahuje 4 jádra.

Zde prezentované výsledky jsme obdrželi při řešení problému obtékání leteckého profilu NACA 0012 ($M = 0.5$, $\alpha = 0°$, $Re = 5000$). Tuto úlohu jsme řešili



Obrázek 4.1: Obtékání profilu - isokřivky Machova čísla

na dvou různých sítích: jednou na síti o 6876 elementech (tu budeme dále značit jako *síť 1*), podruhé na jemnější síti o 29040 elementech (tu označíme *síť 2*). Na těchto sítích jsme počítali postupně s P_1, P_2 a P_3 aproximací. Velikost matic $\boldsymbol{A}_k$, vznikajících v průběhu výpočtu, je pro všechny řešené případy uvedena v tabulce 4.1.

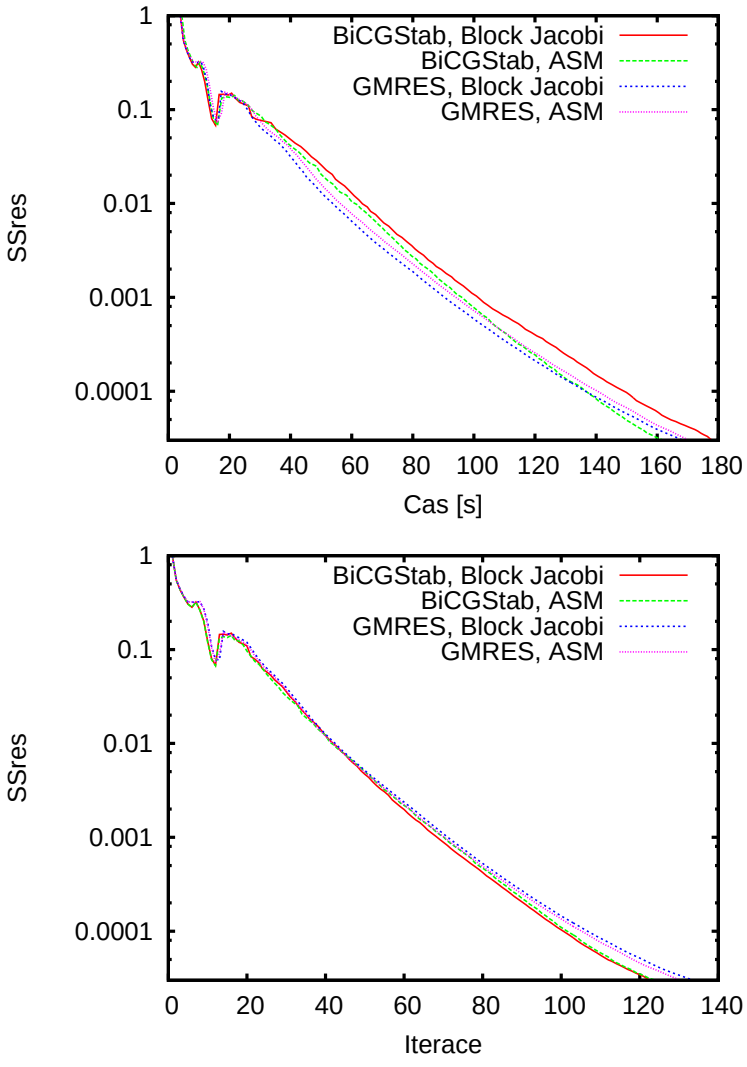
[1] http://cluster.karlin.mff.cuni.cz/

	Síť 1	Síť 2
P_1	82 512	348 480
P_2	165 024	696 960
P_3	275 040	1 161 600

Tabulka 4.1: Velikost matic $\boldsymbol{A}_k$

4.2 Volba řešiče a předpodmiňovače

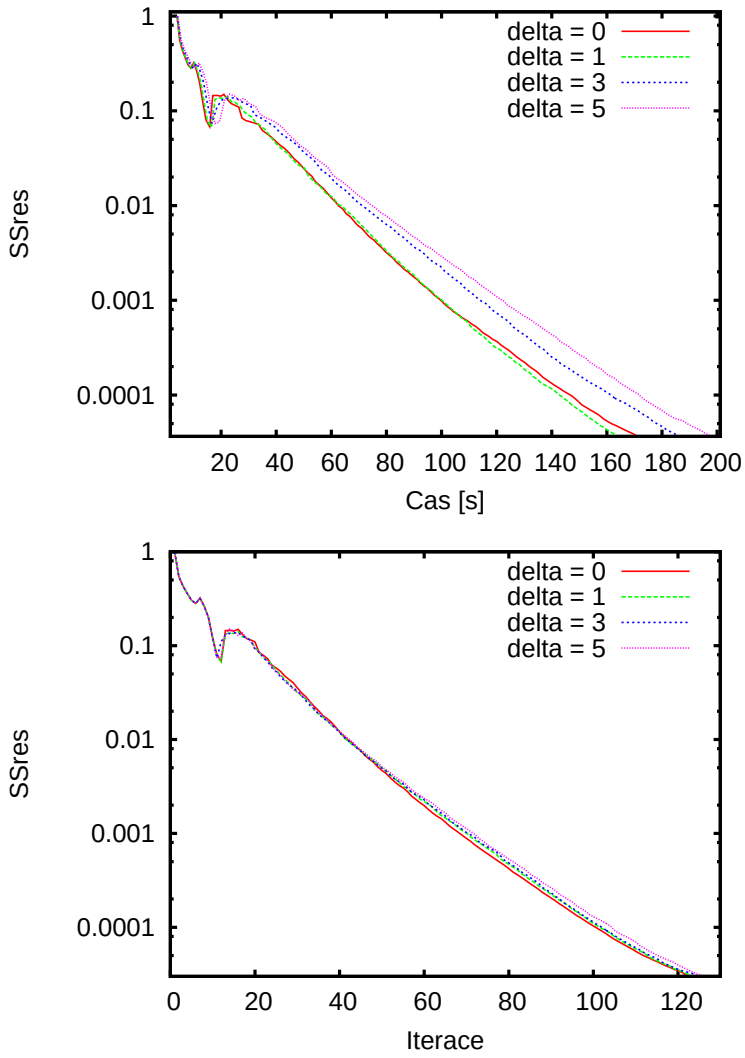
Knihovna PETSc nabízí mnoho řešičů soustav lineárních rovnic a předpodmiňovačů. My jsme v našem programu ozkoušeli řešiče GMRES a BiCGStab spolu s blokovým Jacobiho předpodmiňovačem a předpodmiňovačem ASM (které jsme popsali v předchozí kapitole). Jejich kombinace jsme otestovali na stacionárním problému obtékání profilu NACA 0012. Testy jsme provedli na obou popsaných sítích a s různými stupni aproximace, zde ukazujeme výsledky při počítání na jemnější síti 2 a s P_2 aproximací. U předpodmiňovače ASM jsme použili překryv $\delta = 1$. Počítali jsme na 16 procesorech.



Obrázek 4.2: Porovnání rychlosti konvergence

Jak je vidět na obr. 4.2, kde ukazujeme závislost residua SSres (viz (2.13)) na výpočetním čase a počtu iterací, oba zkoušené řešiče i oba předpodmiňovače fungují stejně dobře. Řešení konvergují podobnou rychlostí, metoda BiCGStab sice konverguje trochu rychleji co se týče počtu iterací, výpočetní čas však zůstává přibližně stejný. Testy s rozdílnými stupni aproximace, na odlišné síti i na různém počtu procesorů ukázaly podobné chovaní.

Testovali jsme ještě vliv překrytí δ u předpodmiňovače ASM na rychlost konvergence. Na obr. 4.3 lze vidět závislost SSres na výpočetním čase a počtu iterací pro různé hodnoty parametru δ, včetně $\delta = 0$, tedy blokového Jacobiho předpodmiňovače. Residuum klesá takřka stejně rychle (dle počtu iterací), ale výpočetní



Obrázek 4.3: Vliv překrytí δ na rychlost konvergence

čas se pro zvětšující se δ zvyšuje. Vyšší překryv nám tedy v tomto případě přináší hlavně zvýšené nároky (časové i paměťové) kvůli komunikaci mezi procesory, ale ne už očekávaný pozitivní efekt.

Výraznější rozdíl již byl v paměťových nárocích. V tabulce 4.2 ukazujeme maximální spotřebu paměti na jednom procesoru, opět pro výpočet na síti 2 s P_2 aproximací. I na ostatních příkladech se ukázalo, že blokový Jacobiho předpodmiňovač je značně úspornější než předpodmiňovač ASM, stejně jako menší nároky

	GMRES	BiCGStab
Blokový Jacobi	160 MB	150 MB
ASM	201 MB	195 MB

Tabulka 4.2: Srovnání spotřeby paměti

řešiče BiCGStab. Protože rychlost konvergence byla u všech zkoumaných variant podobná, vybrali jsme pro další výpočty v našem programu paměťově nejúspornější kombinaci řešiče BiCGStab spolu s blokovým Jacobiho předpodmiňovačem.

4.3 Rozdělení sítí

Zde detailně popíšeme rozdělení sítí, které dostaneme při výpočtech na 1 až 32 procesorech. Rozdělení sítí je nezávislé na použitém stupni aproximace, pokud je ten na všech elementech stejný.

V tabulce 4.3 je uvedeno vždy minimum a maximum z počtu elementů v jedné části sítě n_i. Dále je vypsán počet hran triangulace, které leží na hranici dvou částí sítě ($\#\mathcal{F}_h^{cut}$) a podíl tzv. *ghost cells*, tedy fiktivních elementů, které jsou uměle vytvořeny na hranicích částí sítě, ku celkovému počtu elementů (GC). Protože ke každé hraně ležící na hranici částí sítě náleží právě dva fiktivní elementy, platí

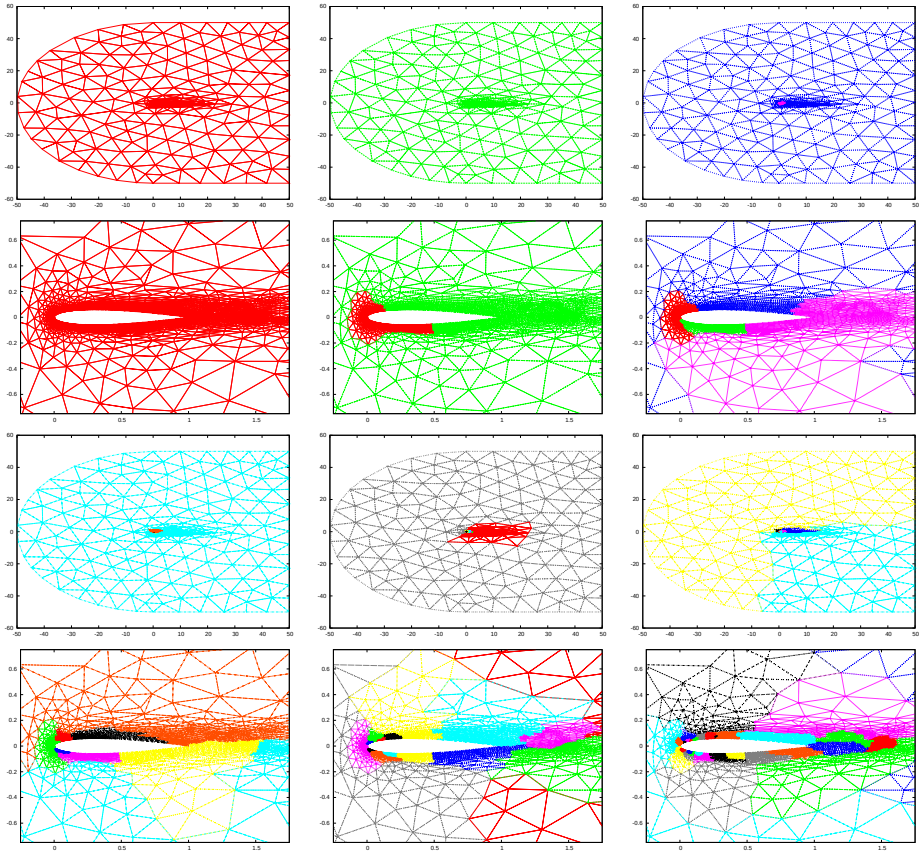
$$GC = 2\frac{\#\mathcal{F}_h^{cut}}{n},$$

kde n značí celkový počet elementů v síti. Pro ilustraci jsou na obrázcích 4.4 a 4.5 zobrazena rozdělení sítí mezi jednotlivé procesory.

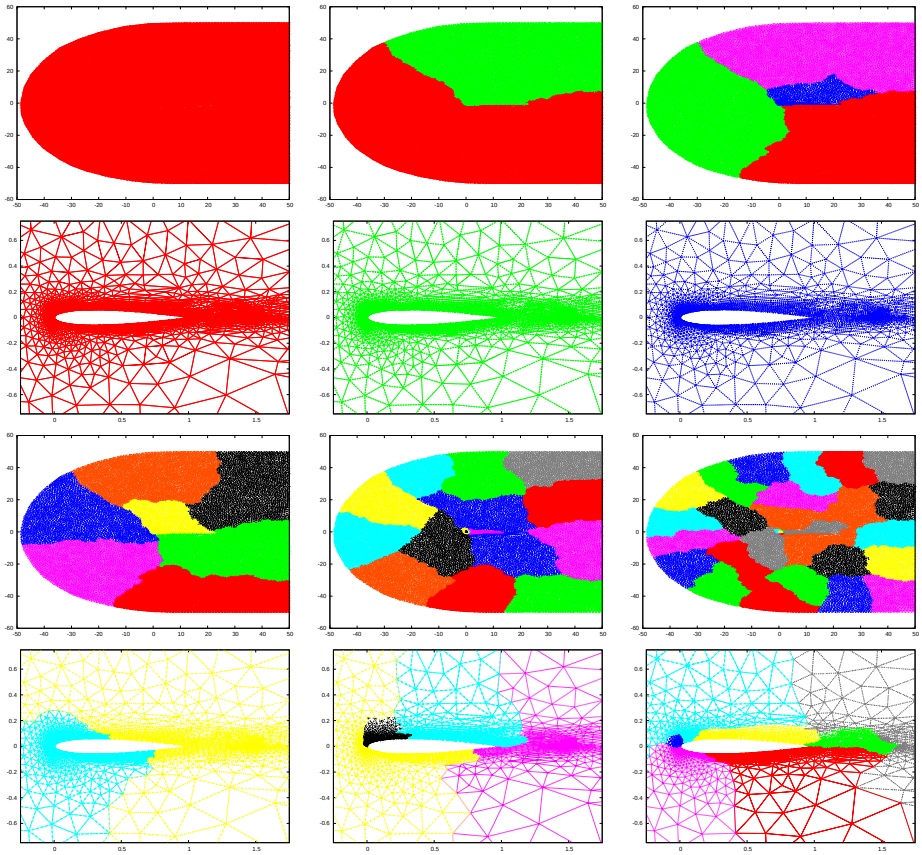
	Síť 1				
p	min n_i	max n_i	$\#\mathcal{F}_h^{cut}$	GC	GC/p
1	6876	6876	0	0.00 %	0.00 %
2	3438	3438	45	1.31 %	0.76 %
4	1719	1719	107	3.11 %	0.78 %
8	859	860	195	5.67 %	0.71 %
16	428	431	311	9.05 %	0.57 %
32	214	216	518	15.07%	0.47 %
	Síť 2				
p	min n_i	max n_i	$\#\mathcal{F}_h^{cut}$	GC	GC/p
1	29040	29040	0	0.00 %	0.00 %
2	14520	14520	103	0.71 %	0.36 %
4	7260	7260	214	1.47 %	0.42 %
8	3630	3630	415	2.86 %	0.36 %
16	1814	1816	669	4.61 %	0.29 %
32	907	908	1101	7.58 %	0.24 %

Tabulka 4.3: Rozdělení sítí

Jak je vidět z tabulky, rozdělení sítě pomocí balíku METIS funguje velmi dobře a výsledné části sítě mají takřka stejné počty elementů, což je základem pro *load balancing*, tedy pro vyvážený objem práce na jednotlivých procesorech během



Obrázek 4.4: Rozdělení sítě 1 na 1 až 32 částí - celek (nahoře) a výřez (dole)



Obrázek 4.5: Rozdělení sítě 2 na 1 až 32 částí - celek (nahoře) a výřez (dole)

následného výpočtu. O samotném rozdělení sice nelze jednoduše prohlásit, zda je optimální a zda je hodnota $\#\mathcal{F}_h^{cut}$ minimální, přesto lze z obrázků a z hodnot $\#\mathcal{F}_h^{cut}$ usuzovat, že vytvořená dělení kvalitní jsou a že splňují požadavky, které jsme na ně kladli v sekci 3.2.

To, že podíl fiktivních elementů pro zvyšující se počet procesorů roste, je problémem jen zdánlivým. Zde je nutné si uvědomit, že přidané fiktivní elementy jsou rozděleny mezi procesory, takže počet fiktivních elementů (a s tím i množství přidané potřebné paměti a množství komunikace) náležící jednomu procesoru spíše klesá, což ukazují hodnoty GC/p uvedené v tabulce.

4.4 Škálování programu

4.4.1 Časové nároky

Při zkoumání časových nároků nás zajímala především doba běhu programu na p procesorech T_p, zrychlení S_p,

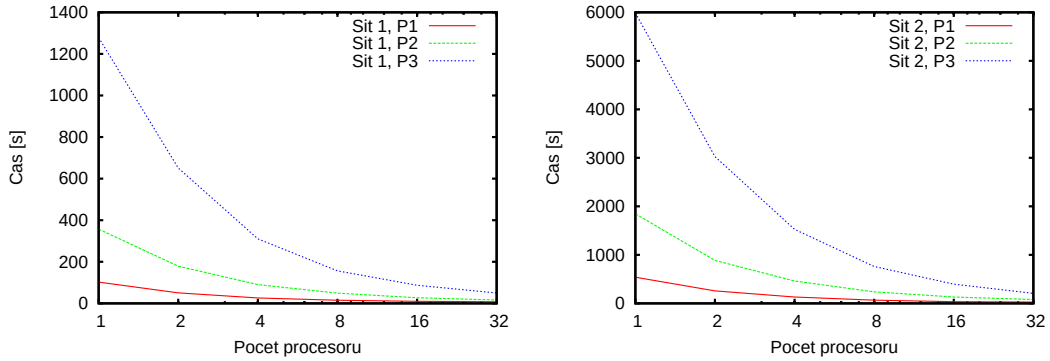
$$S_p = \frac{T_1}{T_p},$$

a efektivita E_p,

$$E_p = \frac{S_p}{p}.$$

Zrychlení nazýváme ideální, pokud $S_p = p$, a tedy $E_p = 1$.

Programem jsme řešili testovací úlohu a ukončili jsme ho po stech iteracích, kdy již bylo dosaženo požadované tolerance. Naměřené hodnoty lze nalézt v tabulce 4.4, rychlost výpočtu je znázorněna na obr. 4.6. Zrychlení lze vidět na obr. 4.7, efektivitu na obr. 4.8.



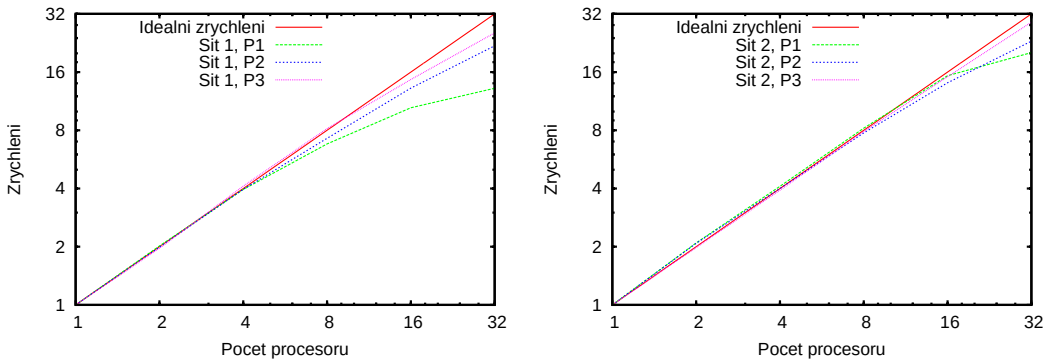
Obrázek 4.6: Doba běhu programu

Podívejme se na výsledky detailně. Je vidět, že malý počet procesorů $(1-4)$ program škáluje velmi dobře i pro úlohy s menším objemem dat. Se zvyšujícím se počtem procesorů se efektivita snižuje, a to v závislosti na velikosti úlohy. Tento pokles je způsoben tím, že stále větší část celkové doby běhu je čas potřebný pro inicializaci a ukončení programu. Ten je pro jednu úlohu konstatní a nezávislý na počtu procesorů, a tudíž přidávání dalších procesorů tento čas nemůže zkrátit.

Je vidět, že máme-li výpočetně náročnější úlohu (tedy počítáme-li na hustší síti či s vyšším stupněm polynomiální aproximace), tak efektivita klesá pro zvyšující se počet procesorů pomaleji. To odpovídá zdůvodění výše: čím větší úloha

	Síť 1, P_1			Síť 2, P_1		
p	T_p	S_p	E_p	T_p	S_p	E_p
1	102.43 s	1.00	1.00	539.73 s	1.00	1.00
2	50.75 s	2.02	1.01	258.16 s	2.09	1.05
4	25.90 s	3.95	0.99	131.49 s	4.10	1.03
8	15.07 s	6.80	0.85	65.80 s	8.20	1.03
16	9.79 s	10.46	0.65	35.21 s	15.33	0.96
32	7.77 s	13.19	0.41	26.75 s	20.18	0.63
	Síť 1, P_2			Síť 2, P_2		
p	T_p	S_p	E_p	T_p	S_p	E_p
1	357.20 s	1.00	1.00	1846.29 s	1.00	1.00
2	179.08 s	1.99	1.00	886.30 s	2.08	1.04
4	90.19 s	3.96	0.99	461.60 s	4.00	1.00
8	49.13 s	7.27	0.91	238.03 s	7.76	0.97
16	27.01 s	13.22	0.83	131.04 s	14.09	0.88
32	16.33 s	21.87	0.68	79.88 s	23.11	0.72
	Síť 1, P_3			Síť 2, P_3		
p	T_p	S_p	E_p	T_p	S_p	E_p
1	1278.27 s	1.00	1.00	5988.53 s	1.00	1.00
2	651.42 s	1.96	0.98	3026.76 s	1.98	0.99
4	310.50 s	4.12	1.03	1526.50 s	3.92	0.98
8	156.58 s	8.16	1.02	760.79 s	7.87	0.98
16	87.28 s	14.65	0.92	395.66 s	15.13	0.94
32	50.07 s	25.53	0.80	206.64 s	28.98	0.91

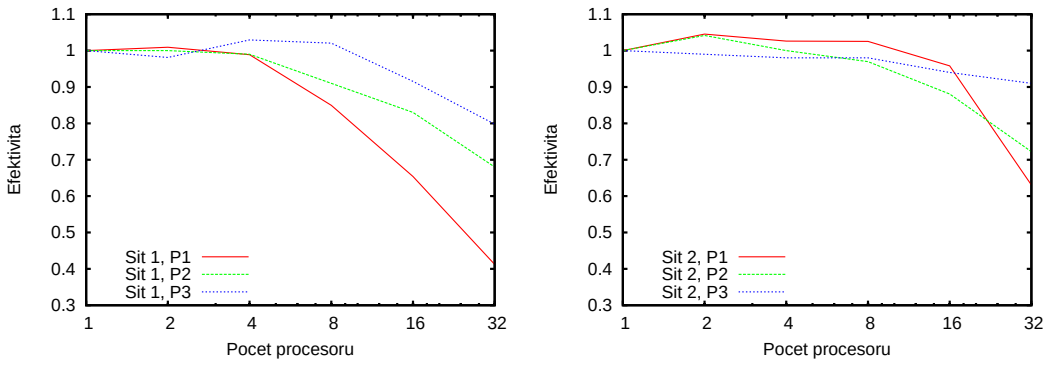
Tabulka 4.4: Časové nároky výpočtů



Obrázek 4.7: Zrychlení

(čímž zde rozumíme úlohu s vyšším počtem nenulových prvků matic $\boldsymbol{A}_k$), tím více času program stráví dobře paralelizovatelným výpočtem a času na inicializaci připadne tak menší díl. Proto nejlepší škálování vidíme při počítání na hustší síti 2 s P_3 aproximací, kde efektivita ani pro 32 procesorů neklesne pod 0.9.

Musíme zde ovšem doplnit ještě dvě poznámky: Zaprvé, v tabulce můžeme nalézt i hodnoty $E_p > 1$. To lze vysvětlit tím, že pro různý počet procesorů probíhá výpočet vždy trochu odlišně. Pro různá rozdělení sítě jsou totiž elementy seřazeny v jiném pořadí a matice A_k se liší uspořádáním bloků, vždy se tedy řeší odlišná



Obrázek 4.8: Efektivita

soustava lineárních rovnic. Kromě toho, algoritmus použitého předpodmiňovače je závislý na počtu procesorů (viz sekce 3.4.1). Odlišný výpočet, přestože vede k jen nepatrně rozdílným výsledkům, tedy může trvat i o něco kratší dobu. Při počítání na více procesorech můžeme občas navíc těžit z tzv. *cache efektu*, kdy se do rychlých cache pamětí na procesorech načte více používaných dat, což běh programu urychlí.

Zadruhé, z tabulky 4.4 je vidět, že při počítání na síti 2 s P_2 aproximací klesá efektivita rychleji než na síti 1 s P_3 aproximací, přestože první úloha je větší než druhá a výpočet první zmíněné úlohy trvá na jednom procesoru déle. To je zřejmě způsobeno tím, že během inicializace a ukončení programu probíhají i neparalelizované části programu, závislé na velikosti sítě (ale ne už na stupni aproximaci). Je to především výpis výsledků na konci běhu programu a též rozdělení sítě na začátku. Na větší síti tedy běží program v neparalelizované části programu déle, což efektivitu snižuje. Tento efekt zde sice není tak významný, při počítání podstatně větších úloh na vyšším počtu procesorů už by však mohl výkon nepříznivě ovlivňovat. Tyto části programu by bylo tedy třeba přepracovat.

4.4.2 Paměťové nároky

U stejných výpočtů jsme ještě změřili paměťové nároky programu. Naprostá většina alokované paměti slouží strukturám knihovny PETSc pro ukládání matic $\boldsymbol{A}_k$ a strukturám pro výpočty soustav $\boldsymbol{A}_k\boldsymbol{\xi}_k = \boldsymbol{d}_k$. Množství potřebné paměti a rozložení dat mezi procesory tedy nemůžeme příliš ovlivnit, nicméně pro úplnost zde naměřené hodnoty uvádíme. Měřili jsme množství paměti alokované na jednotlivých nodech (při použití celkem p nodů), v tabulce 4.5 je z těchto hodnot uvedeno maximum $\overline{m}_p$, minimum $\underline{m}_p$, jejich podíl $\overline{m}_p/\underline{m}_p$ a celkové množství potřebné paměti m_p^{tot}. V tabulce je dále uveden faktor škálování paměťových nároků M_p,

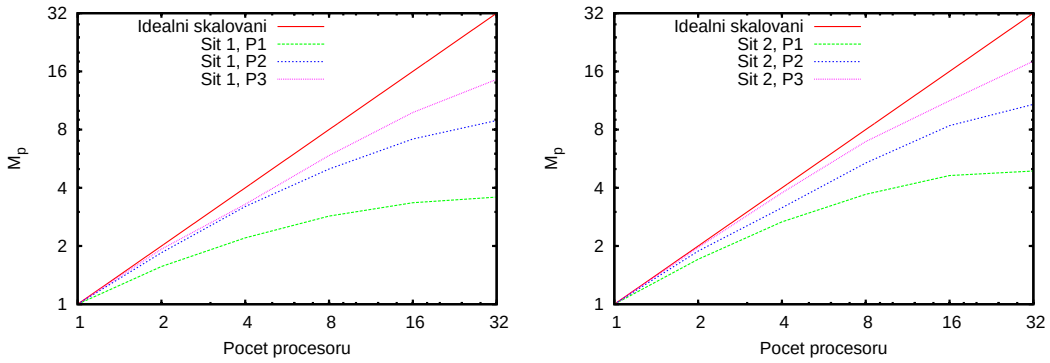
$$M_p = \frac{\overline{m}_1}{\overline{m}_p}.$$

Tento koeficient říká, kolikrát méně paměti je potřeba na jednom nodu, počítáme-li na p procesorech místo jednoho. Faktor škálování paměťových nároků pro všech šest počítaných úloh je zobrazen na obr. 4.9.

Jak vidno z obrázku i z tabulky, škálování programu v oblasti paměťových nároků není ideální a celkové množství alokované paměti při zvyšování počtu výpočetních jednotek narůstá, přesto však při každém zvýšení počtu procesorů

	Síť 1, P_1					Síť 2, P_1				
p	$\overline{m}_p$	$\underline{m}_p$	$\frac{\overline{m}_p}{\underline{m}_p}$	m_p^{tot}	M_p	$\overline{m}_p$	$\underline{m}_p$	$\frac{\overline{m}_p}{\underline{m}_p}$	m_p^{tot}	M_p
1	111	111	1.00	111	1.00	425	425	1.00	425	1.00
2	71	71	1.00	146	1.57	249	249	1.00	497	1.71
4	50	49	1.04	197	2.20	159	156	1.02	629	2.67
8	39	37	1.05	299	2.85	115	111	1.04	892	3.70
16	33	32	1.04	521	3.35	92	89	1.03	1441	4.63
32	31	30	1.05	973	3.58	87	77	1.13	2568	4.88
	Síť 1, P_2					Síť 2, P_2				
p	$\overline{m}_p$	$\underline{m}_p$	$\frac{\overline{m}_p}{\underline{m}_p}$	m_p^{tot}	M_p	$\overline{m}_p$	$\underline{m}_p$	$\frac{\overline{m}_p}{\underline{m}_p}$	m_p^{tot}	M_p
1	336	336	1.00	336	1.00	1260	1260	1.00	1260	1.00
2	183	183	1.00	365	1.85	669	668	1.00	1337	1.88
4	105	104	1.01	418	3.20	399	396	1.01	1589	3.15
8	67	65	1.03	525	5.00	234	231	1.02	1851	5.37
16	47	46	1.03	736	7.15	150	148	1.02	2374	8.38
32	38	37	1.02	1189	8.91	117	109	1.07	3512	10.80
	Síť 1, P_3					Síť 2, P_3				
p	$\overline{m}_p$	$\underline{m}_p$	$\frac{\overline{m}_p}{\underline{m}_p}$	m_p^{tot}	M_p	$\overline{m}_p$	$\underline{m}_p$	$\frac{\overline{m}_p}{\underline{m}_p}$	m_p^{tot}	M_p
1	779	779	1.00	779	1.00	3259	3259	1.00	3259	1.00
2	404	403	1.00	808	1.93	1659	1657	1.00	3316	1.96
4	237	235	1.01	944	3.29	865	861	1.00	3452	3.77
8	133	131	1.01	1050	5.87	470	465	1.01	3736	6.94
16	79	78	1.02	1258	9.81	289	286	1.01	4591	11.28
32	54	52	1.02	1690	14.55	180	178	1.01	5730	18.07

Tabulka 4.5: Paměťové nároky výpočtů (v MB)



Obrázek 4.9: Faktor škálování paměti

dochází k podstatnému snížení paměťových nároků na jediném procesoru. Navíc, koeficient $\overline{m}_p/\underline{m}_p$ blízký 1.00 pro všechny hodnoty p ukazuje, že distribuce paměti je rovnoměrná a není třeba očekávat odlišné paměťové nároky na různých nodech. Stejně jako u časových nároků, i zde platí, že program škáluje lépe pro větší množství dat.

Výše uvedené hodnoty nám tedy potvrzují, že rozložení paměti paralelně běžícího programu mezi jednotlivé výpočetní jednotky nám umožní řešit úlohy, pro které RAM na jediném procesoru nedostačuje.

Výsledky testů tedy jasně ukazují, kde leží výhody paralelizovaného programu: jsou to úlohy s velkým objemem dat a nutných výpočtů. Teprve u těch se jasně ukáže dobrá škálovatelnost programu i pro vyšší počty procesorů. Navíc, pokud množství potřebné paměti převyšuje pamět dostupnou na jediné výpočetní jednotce, je paralelní program díky rozdělení paměti jedinou možností, jak úlohu spočítat.

Závěr

Hlavním cílem práce byla paralelizace softwarového balíku pro řešení parciálních diferenciálních rovnic, který používá nespojitou Galerkinovu metodu. Tento cíl se podařilo naplnit.

Jak se ukázalo, postup založený na rozdělení sítě na části příslušné jednotlivým procesorům je pro paralelizaci velmi vhodný. Za pomoci nespojité Galerkinovy metody jsme dostali kompaktní schéma, u kterého je objem komunikace mezi jednotlivými částmi sítě (a tedy procesory) rozumně malý. K úspěšné paralelizaci programu nám pomohl i výběr efektivního předpodmiňovače a řešiče vznikajících soustav lineárních rovnic z použité knihovny PETSc.

Experimenty v kapitole 4 ukazují, že program dobře využívá výkonu dostupného díky použití více procesorů. Z testů na několika sítích s různými stupni aproximace je vidět, že větší efektivity program dosahuje na větších problémech, kdy se naprostá většina výpočetního času stráví v samotném dobře paralelizovaném výpočtu a podstatně méně během inicializace a ukončování programu. Tyto náročné úlohy jsou tedy ideálně vhodné pro paralelní počítání. Všechny provedené testy jasně ukazují, že paralelizace nám umožní mnohonásobné urychlení běhu programu.

Z experimentů lze také vidět, že množství potřebné paměti na jediném procesoru při běhu na mnoha procesorech podstatně klesá. Proto můžeme díky paralelizaci počítat i problémy podstatně větší, než jaké je možno počítat na jediném procesoru, kde jsme omezeni velikostí paměti.

I přes dosažené cíle zůstává v programu několik záležitostí, kterými by se bylo vhodné při další rozvoji zabývat. Pro vylepšení efektivity programu při počítání podstatně větších úloh, než jaké jsme řešili v této práci, a na podstatně větších clusterech (o stovkách nodů), by bylo nutné zaměřit se na dosud neparalelizované části programu během inicializace a ukončování programu, zvláště těch, které jsou závislé na velikosti úlohy. Mezi ty patří vstupně-výstupní operace (tedy čtení a zápis velkého množství dat) a také rozdělování sítě. Ty doposud běží jen na jediném procesoru a zatímco běží, blokují ostatní nečinné procesory. Na úlohách, které jsme počítali, toto zdržení není nijak zásadní, u úloh výrazně větších už by však mohlo být.

Dalším zajímavým tématem je i paralelní implementace *hp*-adaptivity v programu. V současném stavu program může počítat jen na předem dané síti a její adaptivní zjemňování není možné. Při implementaci *hp*-adaptivity by bylo nutné vyřešit otázku komunikace mezi procesory během zjemňování sítě. Nezbytné je také zabývat se migrací elementů mezi jednotlivými částmi sítě, abychom udrželi stejné množství práce na všech procesorech.

Seznam použité literatury

[1] *ADGFEM - Adaptive Discontinuous Galerkin Finite Element Method*. Dostupné z WWW: <http://atrey.karlin.mff.cuni.cz/~dolejsi/adgfem/>, 2011.

[2] Balay, S., Brown, J., Buschelman, K., Gropp, W. D., Kaushik, D., Knepley, M. G., McInnes, L.C., Smith, B. F., Zhang, H. *PETSc - Portable, Extensible Toolkit for Scientific Computation*. Dostupné z WWW: <http://www.mcs.anl.gov/ petsc/>, 2011.

[3] Bassi, F., Rebay, S. *A high-order accurate discontinuous finite element method for the numerical solution of the compressible Navier-Stokes equations*. J. Comput. Phys., **131**: 267-279. 1997.

[4] Cockburn, B., Shu, C.-W. *TVB Runge-Kutta local projection discontinuous Galerkin finite element method for scalar conservation laws II: General framework*. Math. Comput., **52**: 411-435, 1989.

[5] Devine, K. D., Flaherty, J. E., Wheat, S. R., Maccabe, A. B. *A massively parallel adaptive finite element method with dynamic load balancing*. Proceedings of Supercomputing '93: 2-11. 1993.

[6] Dolejší, V., Feistauer, M.. *Semi-implicit discontinuous Galerkin finite element method for the numerical solution of inviscid compressible flow*. J. Comput. Phys., **198** (2): 727-746, 2004.

[7] Dolejší, V. *On the discontinuous Galerkin method for the numerical solution of the Navier-Stokes equations*. Int. J. Numer. Meth. Fluids, **45**: 1083-1106, 2004.

[8] Dolejší, V. *Semi-implicit interior penalty discontinuous Galerkin methods for viscous compressible flows*. Commun. Comput. Phys., **4** (2): 231-274, 2008.

[9] Dolejší, V., Holík, M., Hozman, J. *Efficient solution strategy for the semi-implicit discontinuous Galerkin discretization of the Navier-Stokes equations*. J. Comput. Phys., **230**: 4176-4200, 2011.

[10] Feistauer, M., Felcman, J., Straškraba, I. *Mathematical and Computational Methods for Compressible Flow*. Clarendon Press, Oxford, 2003.

[11] Hesthaven, J. S., Warburton, T. *Nodal Discontinuous Galerkin Methods*. Springer, New York, 2008.

[12] Karypis, G., Kumar, V. *A fast and high quality multilevel scheme for partitioning irregular graphs*. SIAM Journal of Scientific Computing, **20**: 359-392, 1998.

[13] Karypis, G., Kumar, V. *METIS - A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices, version 4.0*. Dostupné z WWW: <http://glaros.dtc.umn.edu/gkhome/metis/metis/overview>, 2011.

[14] Landmann, B. *A parallel discontinuous Galerkin code for the Navier-Stokes and Reynolds-averaged Navier-Stokes equations*. Dissertation. Stuttgart: Universität Stuttgart, 2008.

[15] MPI Forum. *Message Passing Interface Forum*. Dostupné z WWW: <http://www.mpi-forum.org/>, 2011.

[16] Reed, W. H., Hill, T. R. *Triangular mesh methods for the neutron transport equation*. Tech. report, LA-UR-73-479, Los Alamos National Laboratory, 1973.

[17] Smith, B., Bjørstad, P., Gropp, W.. *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*. Cambridge University Press, New York, 1996.

[18] Zajac, M. *Numerické řešení problému simulace proudění stlačitelných tekutin pomocí paralelních výpočtů*. Diplomová práce. Praha: UK, 2008.

Příloha A

Ukázky paralelního kódu

Uvedeme zde několik ukázek z paralelizovaného kódu ilustrujících použití nástrojů MPI a PETSc pro paralelní výpočty. Z kódu jsou vybrány jen některé části, kompletní kód programu je k práci přiložen. Program je psán v jazyce Fortran 95.

A.1 Použití knihovny PETSc

Na několika příkladech si ukážeme příklady použití knihovny PETSc v programu. Nejprve je třeba inicializovat matici, vektor pravé strany, vektor řešení a řešič. Proměnné `bs`, `locM`, `locN`, `globM`, `globN`, `d_nnz` a `o_nnz` popisují tvar a zaplnění matice, jejich nastavení zde přeskakujeme. Pro vytvoření řídké matice s blokovým uspořádáním slouží funkce `MatCreateMPIBAIJ`, k vytvoření vektoru funkce `VecCreateMPI`.

Vše týkající se řešení soustavy obstarává objekt typu `KSP`, jeho inicializace je provedena funkcí `KSPCreate`. Volání funkce `KSPSetFromOptions` nám umožňuje specifikovat parametry týkající se řešení soustavy lineárních rovnic (jako např. volba řešiče, předpodmiňovače či tolerance) z příkazové řádky při spouštění programu. Chceme-li třeba použít řešič BiCGStab místo v programu nastaveného GMRES, stačí program spustit příkazem

```
mpirun -np 4 ./Adgfem input.ini -ksp_type bcgs
```

Makro `CHKERRQ` slouží ke kontrole chybového kódu `ierror`. Nyní samotná ukázka kódu.

```
! matrix creation
call MatCreateMPIBAIJ(MPI_COMM_WORLD,bs,locM,locN,globM,globN,  &
                      0,d_nnz,0,o_nnz,A,ierror)
CHKERRQ(ierror)
call MatSetOption(A,MAT_ROW_ORIENTED,PETSC_FALSE,ierror)
CHKERRQ(ierror)

! vectors
call VecCreateMPI(MPI_COMM_WORLD,locM,globM,b,ierror)
CHKERRQ(ierror)
call VecCreateMPI(MPI_COMM_WORLD,locM,globM,x,ierror)
CHKERRQ(ierror)

! KSP
call KSPCreate(MPI_COMM_WORLD,ksp,ierror)
```

```
CHKERRQ(ierror)

! setting the tolerances
! args: ksp, relative tolerance, absolut tolerance,          &
!       divergence tolerance, maximum number of iterations, ierror

call KSPSetTolerances(ksp,state%Mtol,                    &
                      PETSC_DEFAULT_DOUBLE_PRECISION,    &
                      PETSC_DEFAULT_DOUBLE_PRECISION,    &
                      PETSC_DEFAULT_INTEGER,ierror)
CHKERRQ(ierror)

! setting the method
call KSPSetType(ksp,KSPGMRES,ierror)
CHKERRQ(ierror)

call KSPSetFromOptions(ksp,ierror)
CHKERRQ(ierror)
```

V každém časovém kroku musíme vektor pravé strany $\boldsymbol{b}$ a matici $\boldsymbol{A}$ naplnit daty. Zde ukazujeme vkládání bloků příslušných elementům z části sítě do matice, samotný výpočet členů přeskakujeme. V poli `submesh%elem(:)` jsou uloženy indexy elementů z části sítě příslušné procesoru, každý procesor tedy v cyklu prochází přes všechny své elementy. Pro každý element spočte diagonální (`elBlock(0)%Mb`) a mimodiagonální (`elBlock(1:elem%flen)%Mb`) bloky. Ty pak voláním funkce `MatSetValuesBlocked` vkládá do matice. Když jsou všechna data vložena, připraví se matice k dalšímu použití voláním funkcí `MatAssemblyBegin` a `MatAssemblyEnd`.

```
do i = 1, submesh%nelem
  elem => grid%elem(submesh%elem(i))

  ! Terms evaluation
  ! do j=0,elem%flen
  !   elBlock(j)%Mb = ...
  ! end do

  ! insertion into the matrix
  ! diagonal blocks - adding terms
  idxm = elem%i - 1        ! row index, zero numbering
  idxn = elem%i - 1        ! column index, zero numbering
  call MatSetValuesBlocked(A,1,idxm,1,idxn,elBlock(0)%Mb,     &
                           ADD_VALUES,ierror)
  CHKERRQ(ierror)

  ! non-diagonal blocks - insertion
  do j=1,elem%flen
    id = elem%face(neigh,j)
    if(id > 0) then
      ! idxm stays the same for all neighbours
      idxn = id - 1        ! zero numbering
      call MatSetValuesBlocked(A,1,idxm,1,idxn,elBlock(j)%Mb, &
                               ADD_VALUES,ierror)
      CHKERRQ(ierror)
    end if
  end do
enddo
```

```
! Final assembly
call MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY,ierror)
CHKERRQ(ierror)
call MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY,ierror)
CHKERRQ(ierror)
```

Samotné řešení soustavy rovnic je už pak možno zapsat na pár řádků. Napřed funkcí `KSPSetOperators` k řešiči přiřadíme matici a voláním funkce `KSPSolve` spustíme řešení. Následuje jen zjištění počtu iterací a normy residua.

```
! set the matrix (A) and the matrix for constructiong ...
! ... the preconditioner (also A)
call KSPSetOperators(ksp,A,A,SAME_NONZERO_PATTERN,ierror)
CHKERRQ(ierror)

! solving the system
call KSPSolve(ksp,b,x,ierror)
CHKERRQ(ierror)

! get the number of iterations
call KSPGetIterationNumber(ksp,iter,ierror)
CHKERRQ(ierror)

! get the residual norm
call KSPGetResidualNorm(ksp,resid,ierror)
CHKERRQ(ierror)
```

A.2 Výměna dat pomocí MPI

V každém časovém kroku musí každý procesor rozeslat spočtená data z elementů ležících na hranici sousedícím procesorům a od nich naopak data přijmout. Tato ukázka demonstruje použití některých funkcí MPI. V první části napřed zabalíme odesílaná data pomocí `MPI_Pack` a poté zahájíme neblokující odesílání voláním `MPI_ISend` a přijímání voláním `MPI_IRecv`.

```
! sending myElems
bufPos = 1
do proc = 1,size(submesh%neighSubmesh)
  pos = 0
  ! pack the elements
  do i =1,size(submesh%neighSubmesh(proc)%myElem)
    elem => grid%elem(submesh%neighSubmesh(proc)%myElem(i))

    call MPI_Pack(elem%w(0,1),1,a%MPIvector,a%outBuffer(bufPos),  &
         submesh%neighSubmesh(proc)%numOfMyBytes,pos,a%Comm,ierror)
  end do

  ! send the buffer with nonblocking send
  call MPI_ISend(a%outBuffer(bufPos),                              &
       submesh%neighSubmesh(proc)%numOfMyBytes,MPI_PACKED,         &
       submesh%neighSubmesh(proc)%num,tag,a%Comm,a%sendReq(proc), &
       ierror)

  ! increase bufPos
  bufPos = bufPos + submesh%neighSubmesh(proc)%numOfMyBytes
end do
```

```
! recieving neighElems
bufPos = 1
do proc=1,size(submesh%neighSubmesh)
  ! start recieving
  call MPI_IRecv(a%inBuffer(bufPos),                                    &
       submesh%neighSubmesh(proc)%numOfNeighBytes, MPI_PACKED,          &
       submesh%neighSubmesh(proc)%num,tag, a%Comm,                      &
       a%recvReq(proc),ierror)

  ! increase bufPos
  bufPos = bufPos + submesh%neighSubmesh(proc)%numOfNeighBytes
end do
```

Po zahájením odesílání se můžeme věnovat jiné práci, v našem případě výpočtu objemových integrálů. Poté se můžeme vrátit zpět ke komunikaci. Na ukončení přenosů počkáme voláním `MPI_Waitall` a přenášená data rozbalíme pomocí `MPI_Unpack`.

```
! wait for all the recv requests
call MPI_Waitall(size(a%recvReq), a%recvReq,statuses,ierror)

! wait for the send requests
call MPI_Waitall(size(a%sendReq), a%sendReq,statuses,ierror)

! unpack the data
bufPos = 1
do proc=1,size(submesh%neighSubmesh)
   pos = 0
   do i=1,size(submesh%neighSubmesh(proc)%neighElem)
     elem => grid%elem(submesh%neighSubmesh(proc)%neighElem(i))
     call MPI_Unpack(a%inBuffer(bufPos),                                &
          submesh%neighSubmesh(proc)%numOfNeighBytes, pos,              &
          elem%w(0,1),1,a%MPIvector,a%Comm,ierror)
   end do

  bufPos = bufPos + submesh%neighSubmesh(proc)%numOfNeighBytes
end do
```

Příloha B

Instalace programu Adgfem

B.1 Instalace

Program je určen pro systémy unixového typu. K instalaci je zapotřebí mít v systému nainstalovanou implementaci MPI (například OpenMPI), knihovnu PETSc a knihovnu ParMETIS.

B.1.1 Instalace PETSc a ParMETIS

Pro instalaci PETSc stáhněte balík z `http://www.mcs.anl.gov/petsc/petsc-as/` a postupujte podle pokynů k instalaci. Pokud nemáte v systému nainstalovány balíky BLAS, LAPACK a OpenMPI (potřebné pro běh knihovny PETSc), instalátor PETSc je může sám stáhnout a nainstalovat. Tímto způsobem také doporučujeme nainstalovat knihovnu ParMETIS. Rozbalte stažený archiv např. do `/home/username/soft/` a pokračujte příkazy (při použití bash shellu)

```
export PETSC_DIR=/home/username/soft/petsc-3.1-p0
cd $PETSC_DIR
./config/configure.py --download-f-blas-lapack=1 \
      --download-openmpi=1 --download-parmetis=1
make
make test
```

B.1.2 Instalace programu Adgfem

Ujistěte se, že cesta k MPI kompilátorům je v proměnné `PATH`. Pokud jste instalovali OpenMPI spolu s PETSc, použijte pro to příkaz

```
export PATH=$PATH:/home/username/soft/petsc-3.1-p0/PETSC_ARCH/bin
```

kde `PETSC_ARCH` je název sestavení (např. `linux-gnu-c-debug`). Přesuňte se do složky s programem Adgfem a nainstalujte program příkazy

```
cd /PATH_TO_ADGFEM/Adgfem
make Adgfem
```

To vytvoří ve složce `Adgfem/bin/` spustitelný soubor `Adgfem`.

B.1.3 Spuštění programu

Přesuňte se do složky se spouštěčem a spusťte výpočet

```
cd /PATH_TO_ADGFEM/Adgfem/run
./run.sh
```

B.2 Změna parametrů výpočtu

Počet použitých procesorů (či počet vláken programu) můžete nastavit v souboru `Adgfem/run/run.sh`. Parametry výpočtu je možné nastavit ve vstupním souboru `Adgfem/data/param.ini`, nebo je možné v souboru `Adgfem/run/run.sh` nastavit jiný vstupní soubor.