TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

Zlepšení cache lokality MKP pomocí křivek vyplňujících prostor

Diplomová práce

Studijní program: N2612 – Elektrotechnika a informatika
Studijní obor: 1802T007 – Informační technologie

Autor práce: **Bc. Petr Král**
Vedoucí práce: doc. Mgr. Jan Březina, Ph.D.

Liberec 2020

TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

Zadání diplomové práce

Zlepšení cache lokality MKP pomocí křivek vyplňujících prostor

Jméno a příjmení: **Bc. Petr Král**
Osobní číslo: M18000145
Studijní program: N2612 Elektrotechnika a informatika
Studijní obor: Informační technologie
Zadávající katedra: Ústav nových technologií a aplikované informatiky
Akademický rok: **2019/2020**

Zásady pro vypracování:

1. Proveďte vlastní implementaci sestavení matice tuhosti pro Poissonovu rovnici. 2d oblast, nestrukturovaná trojúhelníková síť.
2. Nastudujte algoritmy pro klastrování množin bodů pomocí křivek vyplňujících prostor.
3. Implementujte vybrané algoritmy pro 2d případ.
4. Proveďte klastrování elementů výpočetní sítě pomocí vybraných algoritmů a porovnejte vliv na cache lokalitu a výkon asemblačního cyklu.
5. Otestujte vybrané algoritmy pro 3D případ.
6. Pokuste se aplikovat přístup v programu Flow123d.

Rozsah grafických prací:	dle potřeby
Rozsah pracovní zprávy:	40-50 stran
Forma zpracování práce:	tištěná/elektronická
Jazyk práce:	Čeština

Seznam odborné literatury:

[1] Kowarschik, Markus, and Christian Weiß. „An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms." In Algorithms for Memory Hierarchies: Advanced Lectures, edited by Ulrich Meyer, Peter Sanders, and Jop Sibeyn, 213-32. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003. ISBN: 978-3-540-36574-7
[2] Bader, Michael. Space-Filling Curves: An Introduction With Applications in Scientific Computing. Springer Science & Business Media, 2012.

Vedoucí práce:	doc. Mgr. Jan Březina, Ph.D. Ústav nových technologií a aplikované informatiky
Datum zadání práce:	9. října 2019
Předpokládaný termín odevzdání:	18. května 2020

L.S.

prof. Ing. Zdeněk Plíva, Ph.D.
děkan

Ing. Josef Novák, Ph.D.
vedoucí ústavu

V Liberci dne 17. října 2019

Prohlášení

Prohlašuji, že svou diplomovou práci jsem vypracoval samostatně jako původní dílo s použitím uvedené literatury a na základě konzultací s vedoucím mé diplomové práce a konzultantem.

Jsem si vědom toho, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci nezasahuje do mých autorských práv užitím mé diplomové práce pro vnitřní potřebu Technické univerzity v Liberci.

Užiji-li diplomovou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti Technickou univerzitu v Liberci; v tomto případě má Technická univerzita v Liberci právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Současně čestně prohlašuji, že text elektronické podoby práce vložený do IS/STAG se shoduje s textem tištěné podoby práce.

Beru na vědomí, že má diplomová práce bude zveřejněna Technickou univerzitou v Liberci v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů.

Jsem si vědom následků, které podle zákona o vysokých školách mohou vyplývat z porušení tohoto prohlášení.

15. května 2020 Bc. Petr Král

Poděkování

Mé poděkování patří Mgr. Janu Březinovi, Ph.D. za vedení mé diplomové práce, za vstřícnost a cenné rady. Mnohému mě naučil.

Zároveň také děkuji doc. RNDr. Pavlovi Satrapovi, Ph.D. za vytvoření šablony diplomové práce pro LaTeX, která jistě ušetřila práci mnoha studentům.

Zlepšení cache lokality MKP pomocí křivek vyplňujících prostor

Abstrakt

Simulátor proudění podzemní vody Flow123d provádí velké množství časově náročných výpočtů s daty představujícími síť bodů v prostoru. Při vysoké časové náročnosti těchto výpočtů je žádoucí efektivně využívat cache paměť procesoru. Pokud jsou body v datových strukturách uloženy v pořadí nezávislém na jejich vzájemné poloze v prostoru, vede to k častým výpadkům z cache paměti procesoru a delšímu běhu výpočtů

V této práci jsou popsány a porovnány metody, jakými lze problému přistupovat, a docílit vhodného přeuspořádání dat vedoucí k efektivnějšímu využití cache paměti procesoru a následné nižší časové náročnosti výpočtů nad těmito daty.

Klíčová slova:

C++, cache, cache lokalita, Flow123d, Hilbertova křivka, matice, MKP, křivka vyplňující prostor, vektor

Improving the cache locality of FEM with space filling curves

Abstract

Simulator of underground water flow named Flow123d performs a large number of time-consuming calculations with data representing points in space. With such time-consuming calculations, it is desirable to use the CPU cache effectively. If these points are not stored with respect to their positions in space in the datastructures, this leads to frequent cache misses and longer calculation times.

This thesis describes and compares methods, which could be used to achieve appropriate reorderings of the data leading to more effective use of CPU cache and following less time-consuming calculations.

Keywords:

Armadillo, C++, cache, cahce locality, Flow123d, Hilbert curve, matrix, FEM, space filling curve, vector

Obsah

1 Úvod 12
1.1 Cíl práce . . . 12
1.2 Struktura práce . . . 12

2 Cache paměť procesoru 13

3 Křivky vyplňující prostor 15
3.1 Z-křivka . . . 15
3.2 Hilbertova křivka . . . 16

4 Lokalita bodů v datových strukturách 17
4.1 Redukce souřadnic . . . 18
4.2 Křivky vyplňující prostor . . . 19
4.3 Využití ve Flow123d . . . 19

5 Výpočetní síť 20
5.1 Testovací sítě . . . 21
5.1.1 Square uniform . . . 22
5.1.2 Square refined . . . 23
5.1.3 Lshape uniform . . . 24
5.1.4 Lshape refined . . . 25
5.2 Třída Mesh . . . 26

6 MeshOptimizer 28
6.1 Použití . . . 28
6.2 Pomocné struktury . . . 29
6.2.1 Vec3 . . . 29
6.2.2 Permutee . . . 30
6.2.3 Normalizer . . . 31
6.3 Atributy . . . 32
6.4 Výpočet rozměrů . . . 32
6.5 Metody výpočtu hodnoty na křivce . . . 34
6.5.1 Redukce na první souřadnici . . . 34
6.5.2 Redukce na průměr souřadnic . . . 35
6.5.3 Z-křivka . . . 35
6.5.4 Hilbertova křivka . . . 36

6.6 Řazení dat . . . 37

7 Testy 39
7.1 Metodika . . . 40
7.1.1 Hardware . . . 40
7.1.2 Software . . . 40
7.1.3 Konfigurace . . . 40
7.1.4 Měření . . . 41
7.2 Test výpočtu lokální matice . . . 42
7.2.1 Testy nad 2D sítí . . . 43
7.2.2 Testy nad 3D sítí . . . 44
7.3 Test asemblace globální matice . . . 45
7.4 Násobení globální maticí . . . 48

8 Závěr 50

Seznam tabulek

1 Velikosti sítí 39

Seznam obrázků

1 Zjednodušené schéma RAM, CPU a cache . . . 13

2 Iterace Z-křivky křivky . . . 15
3 Iterace Hilbertovy křivky . . . 16

4 Náhodné body ve 2D prostoru . . . 17
5 Varianty pořadí průchodu bodů dle zvolených metod . . . 18

6 Vizualizace sítě Square uniform 2D small . . . 22
7 Vizualizace sítě Square uniform 3D small . . . 23
8 Vizualizace sítě Square refined 2D small . . . 23
9 Vizualizace sítě Square refined 3D small . . . 24
10 Vizualizace sítě Lshape uniform 2D small . . . 24
11 Vizualizace sítě Lshape uniform 3D small . . . 25
12 Vizualizace sítě Lshape refined 2D small . . . 25
13 Vizualizace sítě Lshape refined 3D small . . . 26

14 Výsledky 2D testu výpočtu lokální matice pro malou síť . . . 43
15 Výsledky 2D testu výpočtu lokální matice pro velkou síť . . . 44
16 Výsledky 3D testu výpočtu lokální matice pro malou síť . . . 44
17 Výsledky 3D testu výpočtu lokální matice pro velkou síť . . . 45
18 Výsledky 3D testu asemblace globální matice pro malou síť . . . 47
19 Výsledky 3D testu asemblace globální matice pro velkou síť . . . 47
20 Výsledky 3D testu násobení globální matice pro malou síť . . . 48
21 Výsledky 3D testu násobení globální matice pro velkou síť . . . 49

1 Úvod

1.1 Cíl práce

Flow123d (simulátor proudění podzemní vody) je výpočetní software operující s velkým množstvím dat představujícím objekty v prostoru. Implementuje metodu konečných prvků, přičemž klíčovou úlohou je sestavení tzv. matice tuhosti. Při zpracovávání těchto dat je klíčová rychlost prováděných výpočtů, která je závislá mimo jiné na efektivním využití cache paměti procesoru. Pokud data nejsou v datové struktuře uspořádána v podobném pořadí v jakém se objekty jimi reprezentované vyskytují v prostoru (není splněna jejich vhodná návaznost), vede to k častým výpadkům z cache paměti procesoru a tedy k pomalejšímu běhu samotných výpočtů.

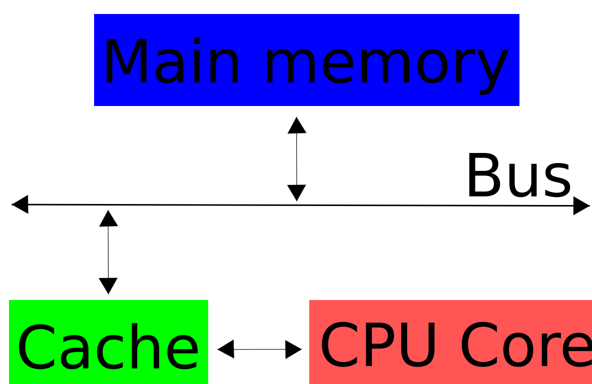
Za účelem eliminace takových situací bylo rozhodnuto, že je třeba přijít se způsobem, jak struktury optimálně přeuspořádat a jako pravděpodobný ideální nástroj k dosažení tohoto cíle bylo navrženo využít křivek vyplňujících prostor. Snaha o implementaci a ověření přístupu je předmětem této práce.

1.2 Struktura práce

V kapitole 2 je nejprve osvětlen princip funkce cache paměti procesoru. V kapitole 3 je přiblížen pojem křivek vyplňujících prostor. Následně jsou v kapitole 4 popsány způsoby, jakými lze k řazení struktur přistupovat. V kapitole 5 je představena výpočetní síť jejíž přeuspořádání je klíčovým zájmem této práce. V kapitole 6 je představena třída MeshOptimizer, která implementuje dané metody a provádí samotnou optimalizaci sítě. Metody jsou nakonec srovnány v kapitole 7.

2 Cache paměť procesoru

Operační paměť procesoru je velmi rychlá paměť malé kapacity, která slouží k dočasnému načtení dávky dat z operační paměti pro výrazné zrychlení přístupu procesoru k datům, se kterými operuje.



Obrázek 1: Zjednodušené schéma RAM, CPU a cache

Operační paměť (na obrázku modře zvýrazněná „Main memory“) slouží jako úložiště dat, se kterými procesor (na obrázku červeně zvýrazněn „CPU Core“) operuje. Procesor obsahuje registry pro uchování mezivýsledků. Načítání dat do registrů přímo z operační paměti je vzhledem k rychlosti procesoru pomalé. Aby byl přístup procesoru k datům co nejrychlejší, využívá procesor ještě malou velmi rychlou cache paměť (na obrázku zeleně zvýrazněná „Cache“), do které si z operační paměti průběžně načítá data přes sběrnici (na obrázku „Bus“).

Jedná se však každopádně o zjednodušený model. V praxi mají procesoru více než jednu úroveň cache paměti. Typicky L1, L2, případně L3.

Načítání dat do cache paměti probíhá na základě pořadí prvků daných datových struktur, ve kterých jsou uloženy. Odhadnout přesně, která data by bylo nejvýhodnější v danou chvíli načíst do cache paměti, není prakticky možné.

Jaká data se kdy načtou do cache paměti, není ani v přímé režii programátora aplikace s danými daty pracující. Je tedy žádoucí, aby data, se kterými program pracuje, byla jednak z důvodu omezené velikosti cache paměti uložena efektivně z pohledu paměťové náročnosti, a zároveň aby byla tato data uložena v daných datových strukturách co nejblíže pořadí, v jakém je bude program zpracovávat (viz kapitola 4).

Vždy, když dojde k případu, kdy požadovaná data nejsou uložena v cache paměti a je tedy nutné je načíst až z operační paměti, dochází ke zpomalení chodu programu vlivem čekání procesoru na načtení dat z operační paměti, která je výrazně pomalejší než cache paměť procesoru. Takovýto jev se nazývá výpadek cache (cache miss). Takovéto zpoždění samozřejmě nastává i v případě nutnosti zápisu hodnot do operační paměti.

Každé takovéto čekání na operační paměť zastaví chod procesoru na mnohonásobek jeho cyklů. Odhaduje se až přibližně 100 cyklů. [5]

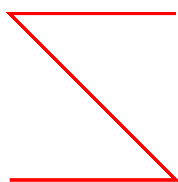
3 Křivky vyplňující prostor

Křivka vyplňující prostor je vzájemně jednoznačné zobrazení například ze dvourozměrného prostoru do prostoru jednorozměrného. Jedná se tedy další metodu, jak přiřadit vícerozměrnému vektoru jednorozměrný index, který může sloužit jako hodnota pro řazení.

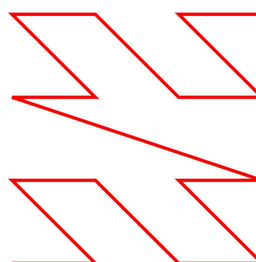
Křivek vyplňujících prostor je mnoho, z důvodů popsaných v podkapitole 4.2 byly vybrány následující dvě.

3.1 Z-křivka

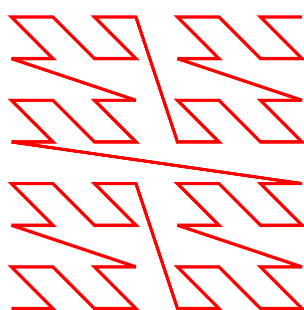
Z-křivka je první ze dvou křivek vyplňujících prostor použitých v této práci. Lze si všimnout, že během jednotlivých iterací dochází k procházení jednotlivých kvadrantů vždy v pořadí obráceného Z. [7]



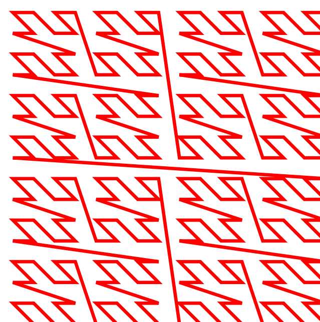
(a) první iterace



(b) druhá iterace



(c) třetí iterace

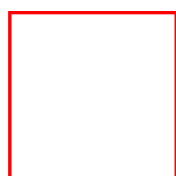


(d) čtvrtá iterace

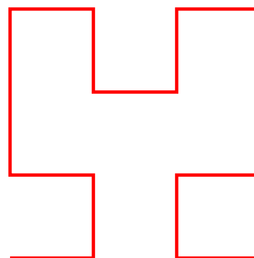
Obrázek 2: Iterace Z-křivky křivky

3.2 Hilbertova křivka

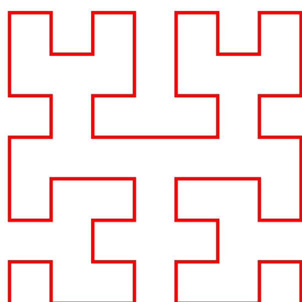
Hilbertova křivka na rozdíl od Z-křivky neprochází kvadranty jednotlivých iterací vždy ve stejném pořadí, ale pořadí je vždy takové, aby konec křivky v rámci jednoho kvadrantu navazoval na začátek křivky kvadrantu následujícího. [3]



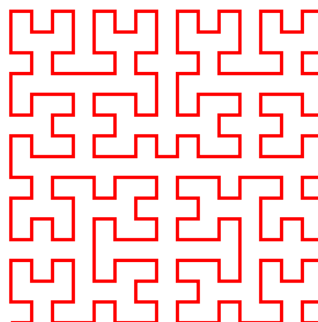
(a) první iterace



(b) druhá iterace



(c) třetí iterace



(d) čtvrtá iterace

Obrázek 3: Iterace Hilbertovy křivky

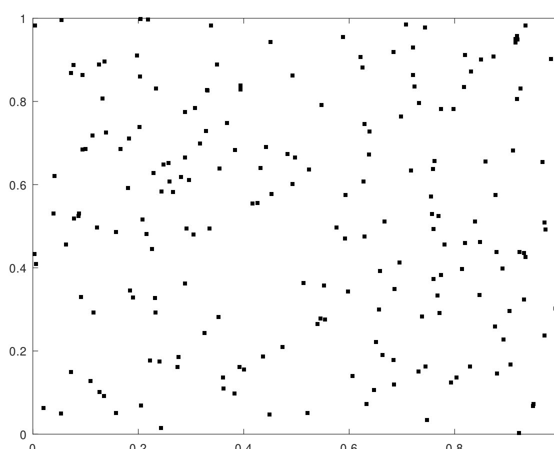
4 Lokalita bodů v datových strukturách

Snahou docílit lokality bodů v datových strukturách je zde myšlena snaha zajistit, aby se data reprezentující body v prostoru, které se v tomto prostoru nachází blízko sebe, nacházely blízko sebe zároveň v dané datové struktuře v paměti, ve chvíli, kdy jsou prostřednictvím těchto struktur zpracovávána daným programem.

Nejjednodušším případem, jaký si lze představit je případ, kdy provádíme výpočty s body z jednorozměrného prostoru. Lokality takovýchto bodů lze docílit seřazením daných jednoprvkových vektorů dle hodnoty jejich jediného prvku reprezentující hodnotu své jediné souřadnice. Tím docílíme situace, kdy body, které jsou si navzájem blíže v prostoru, si vždy budou zároveň blíže i v dané datové struktuře. Toho v případě bodů vícerozměrného prostoru obecně nelze dosáhnout.

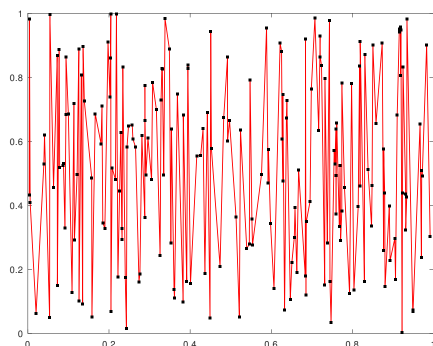
V případě bodů ve vícerozměrných prostorech (vícerozměrných bodech) je vždy nutná určitá forma zobrazení vícerozměrného vektoru na jednorozměrný prostor. Každému bodu je nutné nějakým způsobem přiřadit jednorozměrný index, dle kterého je bude možné řadit. Dále jen „hodnota na křivce“. Pořadí uložení těchto bodů v datové struktuře pak vždy odpovídá jakési křivce, po které za sebou jednotlivé body následují.

Mějme dvourozměrný souřadnicový systém s počátkem v bodě $[0; 0]$, kde $x \in [0; 1]$ a $y \in [0; 1]$. Mějme 200 bodů náhodně rozmístěných v tomto prostoru. Klíčovou úlohou této práce je nalézt způsoby, v jakém pořadí tyto body projít, aby body po sobě následující byly sobě co možná nejblíže v prostoru.

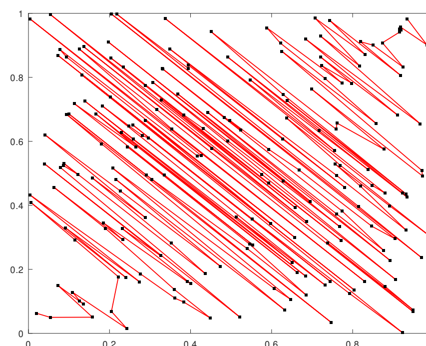


Obrázek 4: Náhodné body ve 2D prostoru

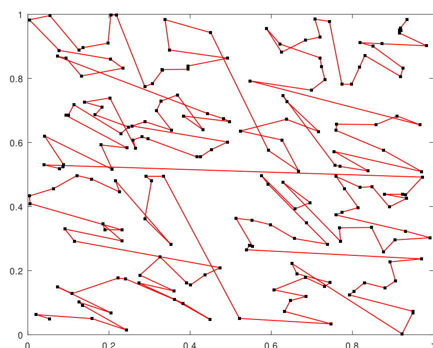
V následujících podkapitolách je popsáno několik přístupů, které bylo v této práci za tímto účelem použity.



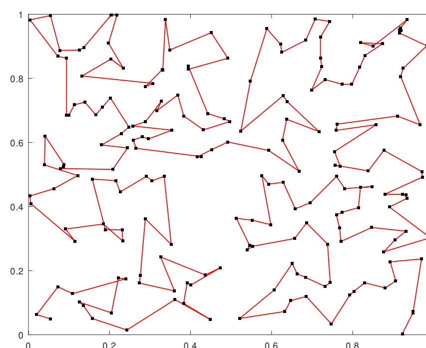
(a) dle první souřadnice



(b) dle průměru souřadnic



(c) dle Z-křivky



(d) dle Hilbertovy křivky

Obrázek 5: Varianty pořadí průchodu bodů dle zvolených metod

4.1 Redukce souřadnic

Mezi nejjednodušší způsoby, jak vektoru hodnot přiřadit jednorozměrný index, dle kterého je následně možná vektory řadit, je například metoda výběru prvku. Mějme například dvourozměrný vektor (x, y), potom funkce přiřazující řadící index tomuto vektoru může vypadat následovně. $f(x, y) = x$. Pokud použijeme tuto metodu, v podstatě to znamená, že body seřadíme podle jejich hodnoty na ose x. Tedy na následujícím obrázku by byly seřazeny zleva doprava (Obrázek 5 (a)).

Tato metoda má však jednu zřejmou nevýhodu. Vzhledem k tomu, že metoda zcela ignoruje zbývající souřadnice. Často bude nastávat situace, kdy budou seřazeny za sebou body, které jsou od sebe na zbývajících osách vzdálené a budou si často blíže než body, které si jsou jinak blíže díky malému rozdílu hodnot zbylých souřadnic.

Další metodou může být například sečtení všech souřadnic. Jedná se tedy o vztah $f(x, y) = x+y$. I v tomto případě se však jedná o zobrazení daných bodů na osu. Zde

ovšem tato osa prochází body $[0;0]$ a $[1;1]$. K jevu popsanému u předchozí metody bude docházet i v tomto případě, přestože žádná z os není přímo ignorována. V zásadě se jedná o stejný případ jako u výběru první souřadnice a jen je pootočená osa. Pravděpodobně tedy nelze od této metody obecně očekávat lepší výsledky (Obrázek 5 (b)).

4.2 Křivky vyplňující prostor

Křivek vyplňujících prostor je mnoho, nicméně lze již intuitivně předpokládat, že některé z nich není praktické pro řešení daného problému použít. Například obtížněji implementovatelné či takové, pro které zjevně neplatí požadavek, aby obrazy bodů umístěných blízko sebe v prostoru byly zároveň sobě blízké.

První ze dvou křivek vyplňujících prostor, které byly v rámci plnění cíle této práce použity, je Z-křivka (viz podkapitola 3.1). Při jejím použití pro určení pořadí průchodu náhodných bodů lze pozorovat, že již nedochází k tak velkým „skokům“ z jednoho konce na druhý, při přechodech mezi kvadranty zde ale stále ke skokům dochází (Obrázek 5 (c)).

Dalším vhodným kandidátem by mohla být například Hilbertova křivka (viz podkapitola 3.2), která je také poměrně snadno implementovatelné jak pro 2D, tak pro 3D případ a dokonce má na rozdíl od Z-křivky jednu dobrou vlastnost. Stejně jako u Z-křivky zde dochází k segmentaci (iteračním rozdělováním prostoru na čtvrtiny), ale na rozdíl od Z-křivky zde začátky segmentů plynule navazují na konce segmentů předchozích a nedochází tak zde ke zmíněným velkým „skokům“. Lze tedy intuitivně očekávat, že by mohla vykazovat lepší výsledky (Obrázek 5 (d)).

4.3 Využití ve Flow123d

Vzhledem k tomu, že klíčovou činnost Flow123d představuje metoda konečných prvků, jako data pro jeho činnost slouží výpočetní síť (viz 5).

Cache lokalita sice má vliv na rychlost chodu programu, ale to pouze na ty jeho části, kde dochází k přistupování dat. Pokud se pouze provádějí výpočty s daty, které se již v cache paměti vyskytují, nemá přeuspořádání prvků na rychlost výpočtu vliv. V rámci výpočtů s výpočetní sítí vykonává Flow123d následující operace, které jsou v principu nelokální.

- Sestavení matice tuhosti
  - čtení souřadnic vrcholů (pro výpočet lokálních matic)
  - zápis lokální matice do globální řídké matice
- Násobení (řídkou) maticí tuhosti
  - čtení pozic sloupců a hodnot matice

5 Výpočetní síť

Výpočetní síť představuje síť propojených bodů. Je definována souřadnicemi uzlů (nodů) sítě v prostoru a množinou výpočetních elementů. Elementy jsou pak tvořeny množinou zmíněných nodů. V závislosti na tom, jestli jde o rovinnou nebo prostorovou síť, představuje element trojúhelník nebo čtyřstěn. Element je tvořen odpovídajícím počtem nodů (dle počtu jeho vrcholů).

Ve Flow123d je síť implementována prostřednictvím třídy `Mesh` (viz kapitola 5.2). Její prvky (elementy) jsou implementovány prostřednictvím třídy `Element`. `Element` zde obsahuje čtveřici bodů třídy `Node`, které představují jednotlivé body v prostoru (nody). Ve 2D případě se čtvrtý node ignoruje.

Síť lze vygenerovat pomocí programu `gmsh`. Výstupem je poté soubor s příponou `.msh`, který obsahuje veškerá konkrétní data o podobě dané sítě. Jako vstup tento program využívá soubory s příponou `.geo`, které obsahují předdefinované vlastnosti sítě bez jejich konkrétních bodů. Syntaxe těchto souborů vypadá následovně.

```
fine_step = 1e-8;
mesh = 0.018;
Point(1) = {0, 0, 0, mesh};
Point(2) = {2, 0, 0, mesh};
Point(3) = {2, 1, 0, mesh};
Point(4) = {1, 1, 0, fine_step};
Point(5) = {1, 2, 0, mesh};
Point(6) = {0, 2, 0, mesh};
Line(1) = {1, 2};
Line(2) = {2, 3};
Line(3) = {3, 4};
Line(4) = {4, 5};
Line(5) = {5, 6};
Line(6) = {6, 1};
Line Loop(7) = {1, 2, 3, 4, 5, 6};
Plane Surface(7) = {7};
Physical Line(".boundary") = {1, 2, 3, 4, 5, 6};
Physical Surface("plane") = {7};
```

Ohraničení je zde definováno pomocí posloupnosti úseček (`Line Loop`). Úsečka (`Line`) představuje dvojici bodů (`Point`), které jsou tvořeny čtyřmi hodnotami. První tři představují souřadnice v prostoru. Čtvrtá určuje, jak hustá má být síť vygenerovaná v blízkosti daného bodu. V příkladu výše tedy vidíme, že síť bude obsahovat lokální zjemnění v okolí bodu 4.

Proces generování sítě z `.geo` souboru je pomocí programu Gmsh velmi snadný. Stačí použít jediný příkaz. Jelikož ale tento program není součástí FLow123d,

je nutné si ho obstarat jinou cestou. Jednoduše můžeme například použít Docker. K vytvoření docker image s programem Gmsh. lze použít následující Dockerfile.

```
FROM ubuntu:18.04
RUN bash -c "apt update &&\
             apt -y upgrade &&\
             apt -y install ssh gmsh"
WORKDIR /root/gmsh
```

Pokud se nacházíme v adresáři, kde se nachází soubor `Dockerfile` s výše zmíněným obsahem, vytvoříme odsud Docker image následujícím příkazem.

```
docker build -t gmsh_image .
```

Pokud se nacházíme v adresáři, kde se nachází daný `.geo` soubor, bude soubor k dispozici i v rámci Docker kontejneru po spuštění následujícího příkazu.

```
docker run --rm -it -v $(pwd):/root/gmsh gmsh_image bash
```

A nakonec uvnitř kontejneru spustíme příkaz `gmsh` s odpovídajícími argumenty.

```
gmsh -2 square_2D.geo
```

Argumentem `-2` specifikujeme, že chceme vygenerovat 2D síť. Pro 3D síť bychom použili argument `-3`.

5.1 Testovací sítě

Pro účely plnění cíle práce bylo vytvořeno 16 sítí. Jedná se o všechny varianty následujících vlastností:

- tvar - square, lshape (čtvercový tvar nebo tvar písmene L - Zároveň lshape má dvakrát větší maximální rozměry narozdíl od typu square, který se vyskytuje v rozmezí $[0;0]$ až $[1;1]$ respektive $[0;0;0]$ až $[1;1;0]$ (Každá síť je před výpočtem hodnoty na křivce normalizována. Viz podkapitola 6.2.3).
- struktura - uniform, refined (s pravidelně rozmístěnými elementy nebo s lokálními zjemněními - Lokální zjemnění se často vyskytují u reálných dat. V tomto ohledu je tedy tento typ sítě reálným datům podobnější. Lokální zjemnění může mít vliv na cache lokalitu kdy i při přeuspořádání podle některé z metod zmíněných v kapitole 4 se může na jednom místě v datové struktuře vyskytovat větší množství elementů.)
- dimenze - 2D, 3D (rovinná či prostorová - U 3D sítě dochází výpočtem honoty na křivce k redukci většího počtu dimenzí, což může způsobit horší efektivitu uspořádání dle zvolené křivky.)

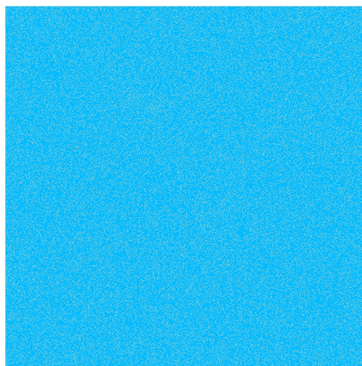
- počet elementů - small, big (Počet nodů a elementů generované sítě nelze v rámci `.geo` souboru přímo určit. Docílit požadované velikosti však lze úpravou hustotu elementů v okolí bodů (viz parametry `fine_step` a `mesh`.). Soubor `.msh` malé sítě má kolem 12 MB, velká síť kolem 150 MB. Poměr hustoty elementů ve zjemněné oblasti a nezjemněné oblasti zůstává pro obě varianty stejný.)

Počty nodů a elementů jednotlivých sítí jsou uvedeny v tabulce 1.
V následujících podkapitolách jsou představeny small verze těchto sítí.

5.1.1 Square uniform

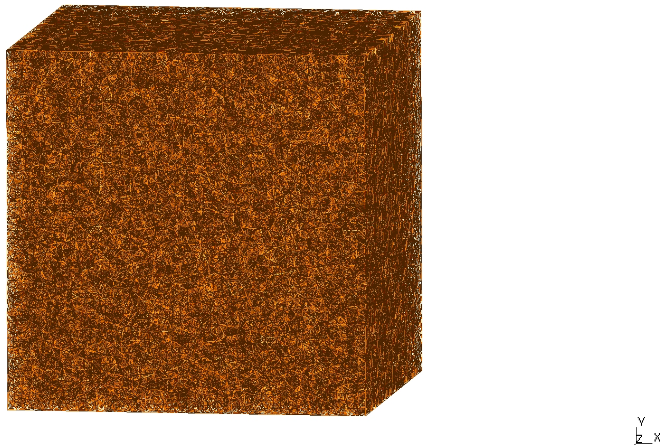
Jedná se o síť tvaru čtverce či krychle v rozmezí okrajových bodů $[0; 0]$ a $[1, 1]$, případně $[0; 0; 0]$ a $[1; 1; 1]$, která je tvořena pravidelnou strukturou. Neobsahuje tedy žádná lokální zjemnění (místa se zvýšenou koncentrací elementů).

2D



Obrázek 6: Vizualizace sítě Square uniform 2D small

3D

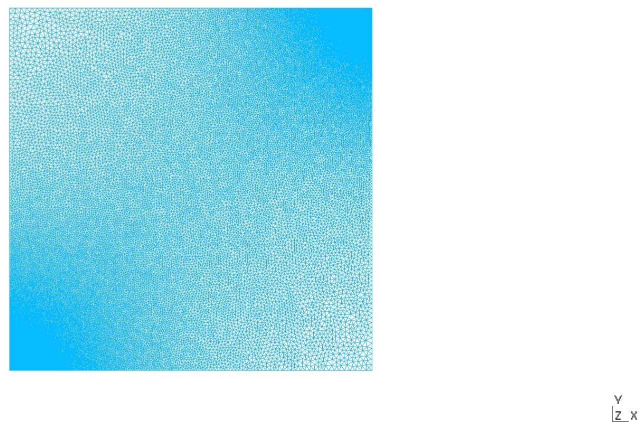


Obrázek 7: Vizualizace sítě Square uniform 3D small

5.1.2 Square refined

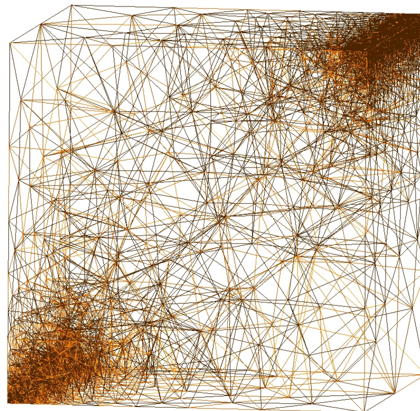
Jedná se o síť vzniklou upravením sítě Square uniform. Síť Square refined obsahuje lokální zjemnění u bodů $[0; 0]$ a $[1; 1]$ respektive $[0; 0; 0]$, $[1; 1; 0]$, $[0; 0; 1]$ a $[1; 1; 1]$. K takovýmto zjemněním v reálných datech často dochází. Cílem je tedy zjistit, zda se to nějakým způsobem projeví i při přeuspořádání.

2D



Obrázek 8: Vizualizace sítě Square refined 2D small

3D

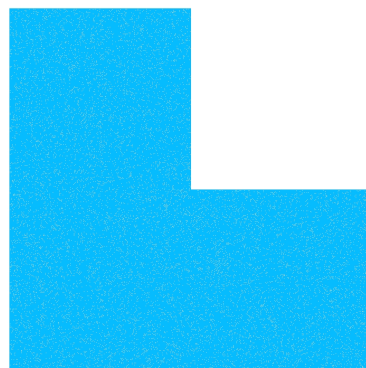


Obrázek 9: Vizualizace sítě Square refined 3D small

5.1.3 Lshape uniform

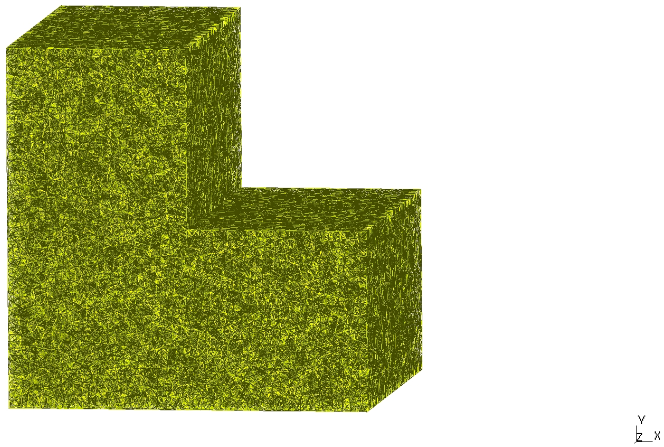
Tato síť je již výrazněji odlišná od dvou předchozích. Nejedná se již o krychli, ale o síť tvaru písmene „L“. Síť tedy obsahuje oblast (o objemu čtvrtiny sítě), ve které se nevyskytují žádné elementy. Zároveň se tato síť nenachází ve stejných mezích jako sítě předchozí, ale v mezích $[0;0]$, $[2;2]$ respektive $[0;0;0]$, $[2;2;2]$. Je tedy v těchto směrech podobnější reálným datům, neobsahuje však žádná zjemnění.

2D



Obrázek 10: Vizualizace sítě Lshape uniform 2D small

3D

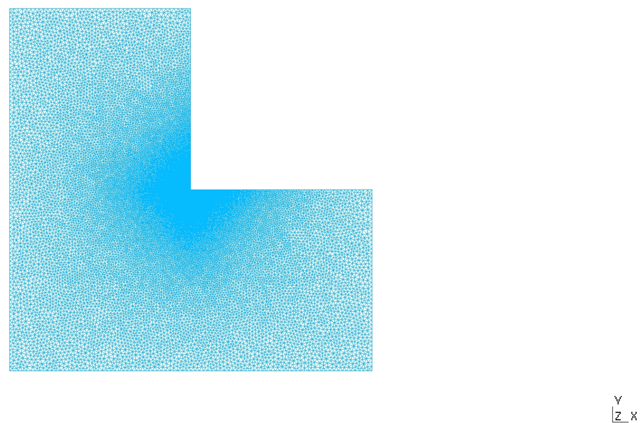


Obrázek 11: Vizualizace sítě Lshape uniform 3D small

5.1.4 Lshape refined

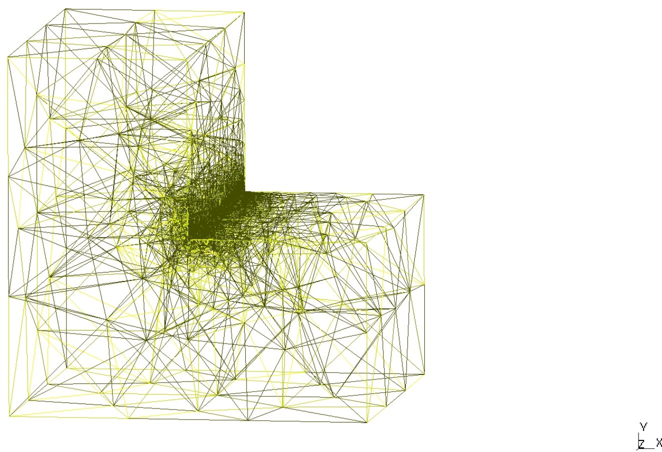
Jedná se síť stejného tvaru, jako síť Lshape uniform, obsahuje však lokální zjemnění v okolí bodu $[1; 1]$ respektive $[1; 1; 1]$. Je zde přibližně 10000krát jemnější. Tato síť je tedy ze všech testovaných reálným datům nejpodobnější.

2D



Obrázek 12: Vizualizace sítě Lshape refined 2D small

3D



Obrázek 13: Vizualizace sítě Lshape refined 3D small

5.2 Třída Mesh

Třídu `Mesh` představuje implementaci výpočetní sítě ve Flow123d. Lze ji ve použít například následujícím způsobem.

K samotné konstrukci třídy a načtení dat z `.msh` souboru slouží například následující metoda.

```
Mesh * mesh_full_constructor(const std::string &input_str,
    Input::FileFormat format = Input::FileFormat::format_JSON)
```

Použití tedy vypadá například následovně.

```
Mesh * mesh = mesh_full_constructor("{mesh_file=\"mesh/cube_3D.msh\"}");
```

Třída `Mesh` ve Flow123d mimo jiné obsahuje následující atribut, ve kterém jsou uloženy jednotlivé elementy (trojúhelníky případně čtyřstěny) sítě. Tento atribut je zde v režimu `protected`.

```
vector<Element> element_vec_;
```

Dalším klíčovým atributem, který je zde rovněž v režimu `protected`, je datová struktura uchovávající jednotlivé nody (body v prostoru).

```
Armor::Array<double> nodes_;
```

K přístupu k elementům a následně nodům slouží speciální metody. Procházet elementy sítě pak lze například následujícím způsobem.

```
for (ElementAccessor<3> elm : mesh->elements_range()) {
    NodeAccessor<3> n0 = elm.node(0);
    NodeAccessor<3> n1 = elm.node(1);
    NodeAccessor<3> n2 = elm.node(2);
    arma::mat::fixed<2, 2> M;
    arma::vec3 tmp;
    tmp = *n1 - *n0;
    // ...
}
```

6 MeshOptimizer

Bylo třeba vytvořit kompaktní nástroj řešící problém optimálního přeuspořádání prvků datových struktur třídy Mesh. Za tímto účelem byl v rámci této práce vytvořen nástroj umožňující toto přeuspořádání různými metodami s názvem MeshOptimizer. Zdrojové kódy jsou veřejně dostupné na Githubu Flow123d ve větvi určené této práci (viz https://github.com/flow123d/flow123d/tree/PK_DP_3).

Účelem MeshOptimizeru je umožnit uživateli s jeho pomocí vícero možnými metodami přeuspořádat nody a elementy dané sítě za účelem zlepšení jejich cache lokality.

Klíčová data třídy Mesh, ke kterým je nutné za účelem přeuspořádání přistupovat jsou atributy `element_vec_`, což je vektor elementů obsahující indexy jemu náležících nodů a atribut `nodes_`, který slouží jako úložiště souřadnic nodů.

```
vector<Element> element_vec_;

Armor::Array<double> nodes_;
```

6.1 Použití

Třída meshOptimizer umožňuje seřadit nody a elementy třídy mesh několika různými způsoby prostřednictvím svých metod.

K přeuspořádání nodů a elementů je nutné zavolat 5 jeho metod, u kterých je nutné zachovat pořadí 3 kroků.

1. Výpočet rozměrů elementů a nodů. Ke správnému výpočtu hodnot na křivce je nutné nejprve tyto rozměry znát. Viz podkapitola 6.4.
2. Výpočet hodnot na křivce pro nody a výpočet hodnot na křivce pro elementy. Viz podkapitola 6.5.
3. Seřazení nodů a seřazení elementů. Jakmile jsou k dispozici vypočtené hodnoty na křivce, je možné nody a elementy seřadit. Viz podkapitola 6.6.

Po skončení těchto činností je Mesh v optimalizovaném stavu a MeshOptimizer je možné zdestruovat v zájmu úspory paměti. Následuje jednoduchý příklad použití.

```
Mesh * mesh = mesh_full_constructor("{mesh_file=\"mesh/cube_3D.msh\"}");

{
```

```
    MeshOptimizer<3> mo(*mesh);
    mo.calculateSizes();
    mo.calculateNodeCurveValuesAsHilbert();
    mo.calculateElementCurveValuesAsHilbertOfCenters();
    mo.sortNodes();
    mo.sortElements();
}

// mesh is now optimized
```

Prvním krokem je konstrukce Objektu třídy MeshOptimizer. Jedná se o šablonovanou metodu, kde je nutné specifikovat, zda budeme pracovat s dvourozměrným či trojrozměrným prostorem. Konstruktoru se předá reference na objekt třídy Mesh, jehož data chceme optimalizovat.

Jelikož výpočet hodnoty křivky vyplňující prostor, na které node či element leží, probíhá aproximačně, je nutné předem určit s jakou přesností se má index pro který bod počítat. To je jedním z důvodů, proč je před samotným výpočtem vhodné použít metodu `calculateSizes()` (viz 6.4), která mimo jiné pro jednotlivé elementy a nody vypočítá vhodnou hodnotu přesnosti.

Poté zde následuje použití metody `calculateNodeCurveValuesAsHilbert()`, která pro jednotlivé nody vypočítá hodnotu „Hilbertova indexu“ (viz 6.5.4).

Metoda `calculateElementCurveValuesAsHilbertOfCenters()` poté vypočítá stejný index i pro každý element sítě.

Nakonec dojte prostřednictvím metod `sortNodes()` a `sortElements()` k samotnému seřazení prvků sítě, které by se mělo pozitivně projevit na rychlosti výpočtů prováděných nad danou sítí.

6.2 Pomocné struktury

MeshOptimizer je sice jednou samostatnou třídou, využívá ovšem ke své funkci několika pomocných struktur.

6.2.1 Vec3

Vec3 je jednou ze tří pomocných struktur, kterou MeshOptimizer používá. Tato pomocná struktura plní pro MeshOptimizer stejný účel jaký plní MeshOptimizer pro Flow123d a sice zajišťuje efektivitu využití cache paměti procesoru. Flow123d totiž k uložení bodu (hodnot 3 jeho souřadnic) používá třídu `arma::vec3`, což je třída knihovny Armadillo. Tato třída má ale bohužel pro účely uložení konstantního malého počtu čísel příliš velkou paměťovou náročnost. Je totiž původně navržen pro struktury s různě velkým počtem prvků a tak si při konstrukci již automaticky alokuje výrazně větší prostor, než může být potřeba. Velikost třídy `arma::vec3` činí 192 bytů. Velikost minimalistické struktury `Vec3`, která v MeshOptimizeru plní roli náhrady za `arma::vec3` činí pouhých 24 bytů. To je skutečně jen trojnásobek velikosti datového typu `double`, který je nosičem hodnoty souřadnice.

```
struct Vec3 {
    inline Vec3(arma::vec3 other) {
        for (uint i = 0; i < 3; ++i) {
            data[i] = other[i];
        }
    }
    inline arma::vec3 arma() const {
        return data;
    }
    inline double& operator[](uint i) {
        return data[i];
    }
    inline double operator[](uint i) const {
        return data[i];
    }
    double data[3];
};
```

Obrovskou výhodou použití takovéto náhrady je, že ji lze použít jako wrapper pro výpočty knihovna Armadillo, která v jejich implementaci využívá jinak celou řadu vhodných optimalizací.

```
Vec3 a({1, 2, 3});
Vec3 b({4, 5, 6});

Vec3 c = std::arma(a.arma() + b.arma());
```

V takovémto případě může optimalizace C++ překladače zajistit, že při daném výpočtu ve skutečnosti k žádné konstrukci `arma::vec3` nedojde, ale výpočet se přitom provede tak optimálně, jak ho implementuje knihovna Armadillo. Výsledkem je díky výrazně nižší paměťové náročnosti a tím způsobené optimální využití cache paměti procesoru výrazně vyšší rychlost výpočtu.

6.2.2 Permutee

Permutee je pomocná struktura MeshOptimizeru, která slouží jako třída prvku vektoru, který v MeshOptimizeru slouží jako permutace (viz podkapitola 6.6). Tato pomocná struktura je zde ze dvou důvodů. První důvodem je optimalizace rychlosti řazení. I při použití optimální řadícího algoritmu při řazení dochází k častému kopírování prvků. Samotná velikost prvků je tedy klíčovým atributem ovlivňujícím rychlost řazení.

```
struct Permutee {
    inline Permutee(uint _originalIndex,
                    double _curveValue) : originalIndex(_originalIndex),
                                          curveValue(_curveValue) {}
    uint originalIndex;
    double curveValue;
};
```

```
inline bool operator<(const Permutee& first, const Permutee& second) {
    return first.curveValue < second.curveValue;
}
```

Velikost elementu (třída `Element` ve Flow123d) činí 104 bytů. Velikost struktury představující bod (třída `Vec3` v MeshOptimizeru) činí 24 bytů. Třída `Permutee` je však účelně velmi minimalistická, obsahuje pouze atribut `curveValue`, což je hodnota typu `double`, podle které se řadí a atribut `originalIndex` typu `uint`, která uchovává informaci, na jaké pozici byl prvek před seřazením. Celková velikost objektu třídy `Permutee` činí pouhých 16 bytů.

Díky této pomocné struktuře tedy stačí seřadit vektor takto minimalistických hodnot následně přepsat samotná velká data jedním průchodem všech prvků na základě následujícího principu.

```
// ...
auto dataBackup = data;

for (uint i = 0; i < data.size(); ++i) {
    data[i] = dataBackup[permutation[i].originalIndex];
}
```

Druhým důvodem použití Permutee v MeshOptimizeru je nutnost mít k dispozici způsob, jak získat nové hodnoty indexů na základě indexů starých.

6.2.3 Normalizer

Normalizer je pomocná struktura, jejíž účel spočívá v přípravě dat k výpočtům hodnot, dle kterých se tyto body budou řadit. Cílem je převést vstupní body či rozměry z původního rozsahu do rozsahu jednotkového čtverce, případně krychle. Funkce přiřazující bodům index (z-křivka nebo Hilbertova křivka) totiž přiřazují hodnotu bodům mezi $[0; 0]$ a $[1; 1]$ pro 2D případ a mezi $[0; 0; 0]$ a $[1; 1; 1]$ pro 3D případ.

Normalizer má dva atributy. Prvním z nich je `shift`, což je bod, představující souřadnice dolního ohraničení. Každá souřadnice tohoto bodu je mimem z hodnot všech bodů sítě na daných souřadnicích. O hodnoty atributu `shift` se tedy celá síť posune takovým způsobem, aby se minima byla nulová.

Druhým z nich je desetinné číslo `scalar` představující poměr pro škálování. Pokud například boudou vstupní data v rozmezí $[0; 0]$ až $[3; 3]$, bude `scalar` roven 3.

```
struct Normalizer {
    inline Normalizer() : shift({0, 0, 0}), scalar(1) {}
    inline Normalizer(Vec3 _shift, double _scalar) : shift(_shift),
                                                    scalar(_scalar) {}
    inline Vec3 normalize(const Vec3 vec) {
        return arma::vec3((vec.arma() - shift.arma()) / scalar);
    }
    inline double normalize(double size) {
        return size / scalar;
```

```
    }
    Vec3 shift;
    double scalar;
};
```

Normalizer poskytuje dvě metody `normalize()`. První z nich provádí normalizaci bodu, druhá normalizaci rozměru.

Normalizer poskytuje dva konstruktory. Prvním z nich je výchozí konstruktor, při jehož použití vzniká objekt třídy Normalizer, který by při použití nezpůsobil žádnou transformaci. A druhý konstruktor, který přímo nastavit konkrétní hodnoty `shift` a `scalar`.

6.3 Atributy

MeshOptimizer obsahuje 6 atributů. Všechny jsou privátní.

```
Mesh& mesh;
std::vector<Permutee> nodeRefs;
std::vector<Permutee> elementRefs;
std::vector<double> nodeSizes;
std::vector<double> elementSizes;
Normalizer normalizer;
```

První atributem je reference na Mesh. Tato reference nemůže být konstantní, protože pochopitelně cílem MeshOptimizeru je síť upravovat. Následuje atribut `nodeRefs`, což je vektor třídy `Permutee`. Ten slouží jako permutace pro řazení nodů (viz podkapitola 6.2.2). Na stejném principu slouží i atribut `elementRefs` pro řazení elementů. Jako úložiště velikostí nodů a elementů (viz podkapitola 6.4) slouží atributy `nodeSizes` a `elementSizes`. Posledním atributem je `normalizer` (viz podkapitola 6.2.3), který je nastaven při výpočtu velikostí (viz podkapitola 6.4).

6.4 Výpočet rozměrů

Jelikož většina metod, které MeshOptimizer používá k přiřazení řadícího indexu bodům, jsou metody aproximační, je nutné vždy nejprve určit přesnost, se kterou se mají indexy pro jednotlivé body počítat. Za tímto účelem obsahuje MeshOptimizer metodu `calculateSizes()`.

Tato metoda v zásadě zajišťuje tři věci. Výpočet velikostí elementů, výpočet velikostí nodů a nastavení normalizeru.

```
inline void calculateSizes() {
    nodeSizes.resize(mesh.n_nodes(), INFINITY);
    elementSizes.reserve(mesh.n_elements());
    for (const ElementAccessor<3>& elm : mesh.elements_range()) {
        double elmSize = calculateSizeOfElement(elm);
        elementSizes.push_back(elmSize);
        const Element& el = *elm.element();
```

```
        for (uint i = 0; i < DIM + 1; ++i) {
            nodeSizes[el.nodes_[i]] = std::min({nodeSizes[el.nodes_[i]],
                                                elmSize});
        }
    }
    std::vector<arma::vec3> tmpArmas;
    tmpArmas.reserve(mesh.n_nodes());
    for (uint i = 0; i < mesh.n_nodes(); ++i) {
        tmpArmas.push_back(mesh.nodes_.vec<3>(i));
    }
    BoundingBox boundingBox(tmpArmas);
    const Vec3 dimensions = arma::vec3(boundingBox.max() - boundingBox.min());
    normalizer = Normalizer(boundingBox.min(), std::max({dimensions[0],
                                                         dimensions[1],
                                                         dimensions[2]}));
}
```

Tato metoda nejprve projde všechny elementy v síti, pro každý uloží jeho velikost do atributu `elementSizes` a pro každý node příslušný danému elementu aktualizuje jeho velikost dle hodnoty své vlastní velikosti takovým způsobem, že velikost každého nodu bude odpovídat minimu velikostí elementů, který daný node náleží. Velikosti nodů jsou pak uloženy v `nodeSizes`. Důvod, proč se tyto velikosti v MeshOptimizeru ukládají namísto, aby se počítaly vždy až když jsou potřeba během výpočtu, spočívá v tom, že velikost nodu MeshOptimizer počítá jako minimum z velikosti elementů, kterým náleží a node neobsahuje informaci o tom, jakým Elementům náleží. Ke spočtení jeho velikosti je tedy nutné projít všechny elementy. K přiřazení hodnoty na křivce pro nody a elementy se používají rozdílné metody. Bylo by tedy jinak nutné je počítat vícekrát. MeshOptimizer proto v metodě `calculateSizes()` spočítá velikosti nodů a elementů najednou a uloží si je. Po provedení všech potřebných metod k optimalizaci sítě (viz podkapitola 6.1) je možné MeshOptimizer zdestruovat, takže uložené velikosti nemusí nutně představovat paměťovou zátěž.

V druhé části se nastaví hodnoty Normalizeru pomocí BoundingBoxu, což je třída obsažená ve Flow123d, která umožňuje získat body ohraničující danou síť. Jedná se o minimum a maximum. Minimum je bod, jehož souřadnice tvoří vždy minimum z hodnot dané souřadnice všech nodů dané sítě. Maximum představuje opačnou mez. BoundingBox však vyžaduje vstupní data typu `std::vector<arma::vec3>`, nody jsou tedy nejprve překopírovány do dočasné proměnné `tmpArmas`.

Metoda využívá pomocnou privátní metodu `calculateSizeOfElement()`, která vypočítá velikost elementu jako nejmenší vzdálenost mezi jeho nody. Tato metoda má i svou 2D specializaci.

```
// 3D
inline double calculateSizeOfElement(const ElementAccessor<3>& elm) {
    return std::min({arma::norm(*elm.node(0) - *elm.node(1)),
                     arma::norm(*elm.node(1) - *elm.node(2)),
                     arma::norm(*elm.node(2) - *elm.node(0)),
                     arma::norm(*elm.node(3) - *elm.node(0)),
                     arma::norm(*elm.node(3) - *elm.node(1)),
                     arma::norm(*elm.node(3) - *elm.node(2))});
```

```cpp
// 2D
inline double calculateSizeOfElement(const ElementAccessor<3>& elm) {
    return std::min({arma::norm(*elm.node(0) - *elm.node(1)),
                     arma::norm(*elm.node(1) - *elm.node(2)),
                     arma::norm(*elm.node(2) - *elm.node(0))});
```

6.5 Metody výpočtu hodnoty na křivce

Výpočtem hodnoty na křivce rozumíme přiřazení bodu v prostoru index vypovídající v jaké části křivky se bod nachází, přičemž 0 a 1 jsou indexy bodů na koncích křivky.

Výpočet hodnoty na křivce se ve třídě MeshOptimzer provádí pro nody, ale i pro elementy. MeshOptimizer používá 4 různé metody výpočtu hodnoty na křivce.

1. redukce na hodnotu první souřadnice
2. redukce na průměr souřadnic
3. hodnota na Z-křivce
4. hodnota na Hilbertově křivce

V případě výpočtu hodnoty pro element se výpočet provádí na bodu, představujícím střed elementu. Třída ElementAccessor ve Flow123d implementuje metodu `centre()`, která v podstatě vrací průměr jemu náležících nodů.

MeshOptimizer umožňuje vypočítat hodnoty pro nody a elementy zvlášť pomocí čtyř výše zmíněných přístupů. Umožňuje však také při výpočtu hodnoty elementů využít průměr z hodnot jemu náležících nodů, případně hodnoty nodů převzít od elementů, kterým náleží. Ta je pak ale závislá na pořadí načtení elementů, neboť později zpracovávaný element přepíše nodu hodnotu získanou od elementu předchozího.

6.5.1 Redukce na první souřadnici

Metoda redukce na první souřadnici je nejjednodušší metoda a to jak na implementaci, tak co se týče časové náročnosti jejího výpočtu. Metoda spočívá v tom, že se body řadí podle pozice na jedné z os.

```cpp
inline void calculateElementCurveValuesAsFirstCoord() {
    elementRefs.reserve(mesh.n_elements());
    for (uint i = 0; i < mesh.n_elements(); ++i) {
        Vec3 tmpNorm = normalizer.normalize(ElementAccessor<3>(&mesh,
                                                                i).centre());
        elementRefs.emplace_back(i, tmpNorm[0]);
    }
}
```

6.5.2 Redukce na průměr souřadnic

Druhou nejjednodušší metodou je výpočet redukcí na průměr souřadnic. $f(x, y, z) = (x + y + z)/3$.

```
inline void calculateElementCurveValuesAsMeanOfCoords() {
    elementRefs.reserve(mesh.n_elements());
    for (uint i = 0; i < mesh.n_elements(); ++i) {
        Vec3 tmpNorm = normalizer.normalize(
            ElementAccessor<3>(&mesh, i).centre()
        );
        elementRefs.emplace_back(i, (tmpNorm[0]
                                     + tmpNorm[1]
                                     + tmpNorm[2]) / 3);
    }
}
```

6.5.3 Z-křivka

Jednou ze dvou křivek vyplňujících prostor, které MeshOptimizer využívá, je Z-křivka. Tato křivka je z nich ta snadněji implementovatelné. Časovou náročností se však od druhé neliší. Jedná se o rekurzivní funkci, která iterativně dělí oblast na kvadranty, které očíslovává v konstantním pořadí. Oproti předchozím metodám má tu výhodu, že dokáže lépe zajistit seskupenost blízkých bodů po seřazení.

```
inline void calculateElementCurveValuesAsZCurveOfCenters() {
    elementRefs.reserve(mesh.n_elements());
    for (uint i = 0; i < mesh.n_elements(); ++i) {
        elementRefs.emplace_back(
            i,
            zCurveValue(
                normalizer.normalize(ElementAccessor<3>(&mesh, i).centre()),
                normalizer.normalize(elementSizes[i])
            )
        );
    }
}
```

MeshOptimizer je šablonovaná třída, neboť některé metody se pro 2D a 3D případ liší. Metoda `zCurveValue()` je sice obecně implementována jako následující wrapper,

```
inline double zCurveValue(const Vec3 vec, double size) {
    return zCurveValue(vec[0], vec[1], vec[2], size * size * size);
}
```

implementace však obsahuje specializaci pro 2D případ.

```
template <>
inline double MeshOptimizer<2>::zCurveValue(const Vec3 vec, double size) {
    return zCurveValue(vec[0], vec[1], size * size);
}
```

Jedná se ovšem stále jen o wrappery. Cílová implementace pro 2D vypadá následovně.

```
// 2D
inline double zCurveValue(double x, double y, double eps) {
    if (eps > 1) {
        return 0;
    } else {
        if (y < 0.5) {
            if (x < 0.5) {
                return zCurveValue(2*x,2*y,4*eps) / 4;
            } else {
                return (1 + zCurveValue(2*x-1,2*y,4*eps)) / 4;
            }
        } else {
            if (x < 0.5) {
                return (2 + zCurveValue(2*x,2*y-1,4*eps)) / 4;
            } else {
                return (3 + zCurveValue(2*x-1,2*y-1,4*eps)) / 4;
            }
        }
    }
}
```

Zde si můžeme všimnout, že při každé iteraci dochází k výpočtu nad zdánlivě stejně rozměrnými daty jako v iteraci předchozí, díky tomu, že se parametry pokaždé vynásobí dvěma. Celkový výsledek iterace se pak opět vydělí, aby se zohlednila váha dané hodnoty v celkovém kontextu.

Metoda je rekurzivní, výpočet indexu je iterační, je tedy nutné určit při jaké přesnosti výpočet ukončit. K tomu slouží parametr `eps`, který představuje obsah plochy (respektive objem v prostoru), pro kterou bodům v ní ležícím přiřazujeme stejný index. Ideální hodnotou při výpočtu indexu daného bodu je tedy druhá respektive třetí mocnina nejmenší vzdálenosti od okolního bodu. Za účelem zjištění těchto rozměrů MeshOptimizer poskytuje metodu `calculateSizes()` (viz podkapitola 6.4). Pochopitelně by bylo možné výpočet provádět s nějakým konstantním velmi malým `eps`, jenže pokud bude výpočetní síť obsahovat výrazná lokální zjemnění, bude docházet k častému výpočtu se zbytečně vysokou přesností, což je pochopitelně časově náročné.

6.5.4 Hilbertova křivka

Hilbertova křivka v podstatě funguje na stejném principu, jaký je popsán u Z-křivky, ovšem s jedním rozdílem. Při každém rekurzivním volání dochází kromě škálování souřadnic také k celkové transformaci prostoru. Na následujícím výpisu kódu si lze všimnout například prohození x a y souřadnic při rekurzivním volání. Takovýmito manipulacemi dokáže tento algoritmus docílit toho, že konce segmentů této křivky plynule navazují na začátky následujících a nedochází tak zde ke „skokům“, jako je tomu u Z-křivky.

```
// 2D
inline double hilbertValue(double x, double y, double eps) {
    if (eps > 1) {
        return 0;
    } else {
        if (x < 0.5) {
            if (y < 0.5) {
                return hilbertValue(2*y,2*x,4*eps) / 4;
            } else {
                return (1 + hilbertValue(2*x,2*y-1,4*eps)) / 4;
            }
        } else {
            if (y >= 0.5) {
                return (2 + hilbertValue(2*x-1,2*y-1,4*eps)) / 4;
            } else {
                return (3 + hilbertValue(1-2*y,2-2*x,4*eps)) / 4;
            }
        }
    }
}
```

6.6 Řazení dat

Poté, co má MeshOptimizer k dispozici vypočtené veškeré hodnoty pro nody a elementy, je možné je podle nich seřadit. MeshOptimizer poskytuje zvlášť metodu pro seřazení nodů a elementů. Metoda `sortNodes()` řadící nody vypadá následovně.

```
inline void sortNodes() {
    std::sort(nodeRefs.begin(), nodeRefs.end());
    std::vector<uint> newNodeIndexes(nodeRefs.size());
    Armor::Array<double> nodesBackup = mesh.nodes_;
    for (uint i = 0; i < nodeRefs.size(); ++i) {
        newNodeIndexes[nodeRefs[i].originalIndex] = i;
    }
    for (uint i = 0; i < mesh.n_elements(); ++i) {
        for (uint j = 0; j < DIM + 1; ++j) {
            mesh.element_vec_[i].nodes_[j]
            = newNodeIndexes[mesh.element_vec_[i].nodes_[j]];
        }
    }
    for (uint i = 0; i < nodeRefs.size(); ++i) {
        mesh.nodes_.set(i) = nodesBackup.vec<3>(nodeRefs[i].originalIndex);
    }
}
```

Zde se nejprve seřadí vektor `nodeRefs` (viz podkapitola 6.2.2), čímž vznikne datová struktora představující zpětnou permutaci. K získání dopředné permutace, která bude potřeba k přepsání indexů nodů v elementech se vytvoří vektor `newNodeIndexes`. Poté se dle této permutace v elementech aktualizují indexy nodů.

Nody se na závěr nakopírují do zálohy, odkud už se v kýžené pořadí kopírují zpět do Meshe.

Na podobném principu je implementována i metoda `sortElements`, která řadí elementy sítě.

```
inline void sortElements() {
    std::sort(elementRefs.begin(), elementRefs.end());
    std::vector<Element> elementsBackup = mesh.element_vec_;
    for (uint i = 0; i < elementRefs.size(); ++i) {
        mesh.element_vec_[i] = elementsBackup[elementRefs[i].originalIndex];
    }
}
```

Zde není nutné aktualizovat žádné indexy, jako tomu bylo u metody `sortNodes`, takže je tato metoda výrazně jednodušší. Do Meshe se kopírují Elementy ze zálohy adresované zpětnou permutací, vzniklou seřazením prvků atributu `elementRefs`.

V momentě, kdy doběhnou obě řadící metody je již Mesh v požadovaném optimalizovaném stavu.

7 Testy

Pro ověření a porovnání účinnosti uspořádání výpočetní sítě pomocí zvolených metod byly provedeny testy následujících algoritmů.

- výpočet lokálních matic (viz podkapitola 7.2)
- Asemblace globální matice (viz podkapitola 7.3)
- Násobení globální maticí (viz podkapitola 7.4)

V testech byly použity matice popsané v podkapitole 5.1. Jejich `.geo` soubory se nachází v adresáři `unit_tests/mesh`. Následuje tabulka počtu nodů a elementů vygenerovaných sítí použitých v testech.

Tabulka 1: Velikosti sítí

Název sítě	**Počet nodů**	**Počet elementů**
Square uniform 3D small	46 450	247 822
Lshape uniform 3D small	46 842	247 473
Square refined 3D small	69 558	221 558
Lshape refined 3D small	62 611	233 628
Square uniform 3D big	507 861	2 950 304
Lshape uniform 3D big	475 456	2 739 781
Square refined 3D big	619 536	2 283 531
Lshape refined 3D big	568 555	2 475 491
Square uniform 2D small	100 556	200 010
Lshape uniform 2D small	110 963	220 588
Square refined 2D small	103 292	204 266
Lshape refined 2D small	104 228	206 968
Square uniform 2D big	1 201 483	2 399 152
Lshape uniform 2D big	1 325 604	2 646 584
Square refined 2D big	1 209 448	2 410 878
Lshape refined 2D big	1 197 239	2 389 340

Vytvořené testy se nachází v souboru `src/unite_tests_mesh/spacefilling_test.cpp`. Testy pro jednotlivé metody byly tedy spoštěny v rámci jednoho uni testu. Hypotézu, že by mohlo pořadí testů být příčinou zdánlivého urychlení, lze zavrhnout. Bylo ověřeno, že testy vykazují stejné výsledky i při změně pořadí.

7.1 Metodika

7.1.1 Hardware

Stroj, na kterém byly testy prováděny: HP Pavilion 15-ab124nc [4]. Jedná se o notebook s procesorem AMD A8-7410 (specifikace viz [1]).

7.1.2 Software

- Operační systém - Arch Linux
- jádro operačního systému - linux 5.6.15.arch1-1
- Docker - Docker version 19.03.10-ce, build 9424aeaee9

7.1.3 Konfigurace

Byla použita release verze docker image Flow123d (flow123d/flow-libs-dev-rel:3.0.0) spustitelná z kořenového adresáře Flow123d následujícím příkazem.

```
./bin/fterm rel
```

Obsah souboru `config.cmake`:

```
# Configuration for CI server
# debug build

# main config
set(FLOW_BUILD_TYPE release)
set(CMAKE_VERBOSE_MAKEFILE on)


# external libraries
set(PETSC_DIR              /usr/local/petsc-3.8.3/)
set(BDDCML_ROOT            /usr/local/bddcml-2.5.0/bddcml)
set(Armadillo_ROOT_HINT    /usr/local/armadillo-8.3.4)
set(YamlCpp_ROOT_HINT      /usr/local/yamlcpp-0.5.2)


# additional info
set(USE_PYTHON          "yes")
set(PLATFORM_NAME       "linux_x86_64")
```

Test byl následně zkompilován pomocí následujících příkazů.

```
make clean-all
make cmake
make all
cd unit_tests/mesh/
make spacefilling-1-test
```

7.1.4 Měření

Měření času běhu výpočtu zajišťuje v rámci toho unit testu následující třída:

```
class Stopwatch {
public:
    inline void start() {
        m_begin = std::chrono::high_resolution_clock::now();
    }
    inline void stop() {
        m_end = std::chrono::high_resolution_clock::now();
    }
    uint microseconds() {
        return std::chrono::duration_cast<std::chrono::microseconds>(m_end
                                                        - m_begin).count();
    }
    uint milliseconds() {
        return std::chrono::duration_cast<std::chrono::milliseconds>(m_end
                                                        - m_begin).count();
    }
    uint nanoseconds() {
        return std::chrono::duration_cast<std::chrono::nanoseconds>(m_end
                                                        - m_begin).count();
    }
private:
    std::chrono::time_point<std::chrono::_V2::system_clock,
        std::chrono::duration<long int, std::ratio<1, 1000000000>>> m_begin;
    std::chrono::time_point<std::chrono::_V2::system_clock,
        std::chrono::duration<long int, std::ratio<1, 1000000000>>> m_end;
};
```

Měření bylo prováděno s přesností na mikrosekundy. A to například následovně:

```
//...
stopwatch.start();
results[3] = performMultiplication(global_matrix, mesh->n_nodes());
stopwatch.stop();
std::cout << "result 4: " << results[3] << '\n';
tw.writeTime(meshName + "_zcurve", stopwatch.microseconds());
//...
```

Pomocí následující pomocné třídy byly výsledky průběžně zapisované do JSON formátu.

```
class TimeWriter {
public:
    TimeWriter(const std::string& filePath) : firstTest(true), file(filePath) {
        file.put('{');
    }
    ~TimeWriter() {
        file.put('}');
        file.put('\n');
    }
    void writeTime(const std::string& testName, double time) {
```

```
        if (!firstTest) {
            file.put(',');
        }
        file << '"' << testName << '"' << ": " << time;
        file.flush();
        firstTest = false;
    }
private:
    bool firstTest;
    std::ofstream file;
};
```

Každý test byl 5krát opakován. K minimalizaci vlivu chodu jiných procesů na stroji byly testy prováděny bez spuštěného grafického prostředí operačního systému. Kompletní výsledky viz přílohy.

7.2 Test výpočtu lokální matice

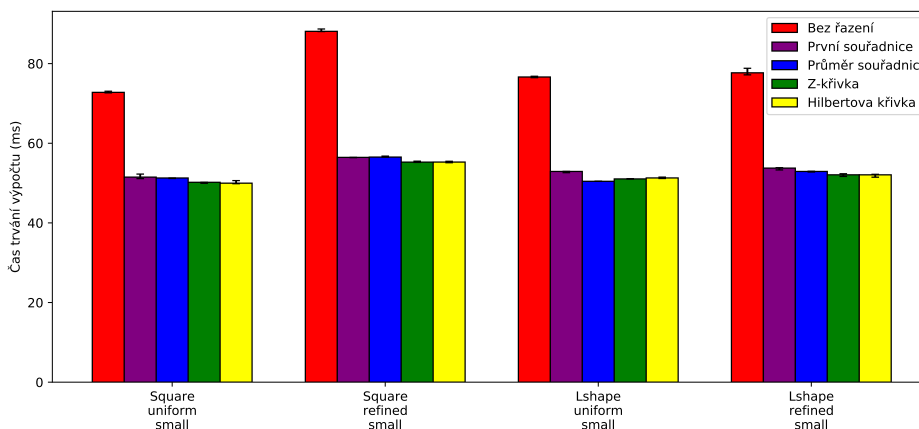
Jedná se o výpočet, který se ve Flow123d provádí při asemblaci globální matice pro každý element sítě. Provádí se tedy velmi často, takže jeho urychlení má v rámci Flow123d význam. Je ale z velké části tvořen matematickými operacemi nad již načtenými daty, takže přeuspořádání výpočetní sítě nemusí mít na rychlost tohoto algoritmu velký dopad. Následuje výpis implementace pro 2D případ.

```
double calculation2DBeforeSort(Mesh * mesh) {
    double checksum = 0;
        for (ElementAccessor<3> elm : mesh->elements_range()) {
            auto n0 = elm.node(0);
            auto n1 = elm.node(1);
            auto n2 = elm.node(2);
            arma::mat::fixed<2, 2> M;
            arma::vec3 tmp;
            tmp = *n1 - *n0;
            M.col(0) = arma::vec2{tmp[0], tmp[1]};
            tmp = *n2 - *n0;
            M.col(1) = arma::vec2{tmp[0], tmp[1]};
            double detM = arma::det(M);
            double jac = std::abs(detM) / 2;
            arma::mat::fixed<3, 3> phi = {{1, -1, -1},
                                          {0,  1,  0},
                                          {0,  0,  1}};
            arma::mat::fixed<3, 2> grad_phi = {{-1, -1},
                                               { 1,  0},
                                               { 0,  1}};
            arma::mat::fixed<3, 3> A_local = {{0, 0, 0},
                                              {0, 0, 0},
                                              {0, 0, 0}};
            for (uint i = 0; i < 3; ++i) {
                for (uint j = 0; j < 3; ++j) {
                    A_local(i, j) += arma::dot(M * grad_phi[i],
                                               M * grad_phi[j]) * jac;
```

```
                    checksum += std::abs(A_local(i, j));
                }
            }
        }
    return checksum;
}
```

7.2.1 Testy nad 2D sítí

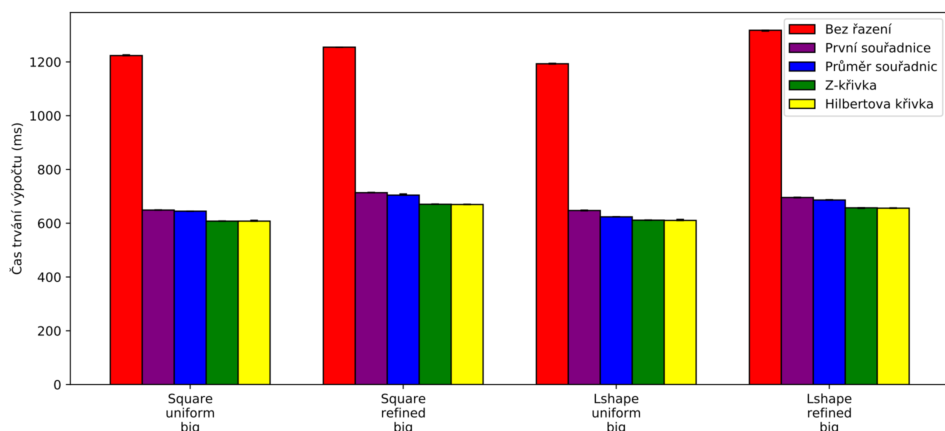
Následuje graf zobrazující výsledky testů. Skupiny sloupců odpovídají daným výpočetním sítím. Barvy sloupců náleží daným metodám přeuspořádání dle legendy. Výška sloupců představuje medián z pěti hodnot času trvání výpočtů. Medián je použit pro svou vyšší odolnost vůči vysokým hodnotám způsobených nahodilou zátěží procesoru jiným procesem, ke kterému může při měření času běhu testu dojít v porovnání s průměrem. Na vrcholech sloupců jsou zároveň znázorněné rozsahy mezi prvním a třetím kvartilem pro představu o chybě měření.



Obrázek 14: Výsledky 2D testu výpočtu lokální matice pro malou síť

Na výsledcích je patrné, že všechny metodu skutečně způsobují zrychlení výpočtu. Rozdíl mezi jednotlivými metodami však není tak velký, jak bylo očekáváno. Zdá se, že i natolik primitivní metoda jako je redukce na první prvek má značný vliv.

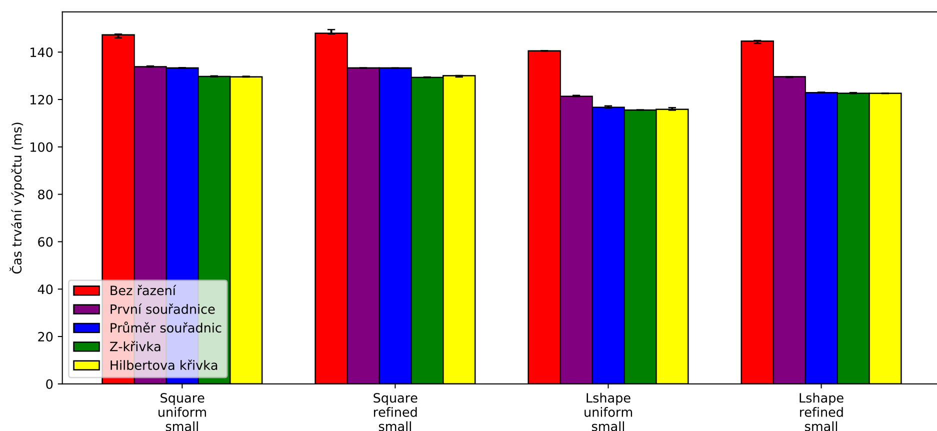
V rozporu s očekáváním zde zároveň Hilbertova křivka nevykazuje nikterak znatelně větší urychlení než Z-křivka. Pravděpodobně tedy návaznost jednotlivých segmentů nehraje tak velkou roli, jak bylo předpokládáno.



Obrázek 15: Výsledky 2D testu výpočtu lokální matice pro velkou síť

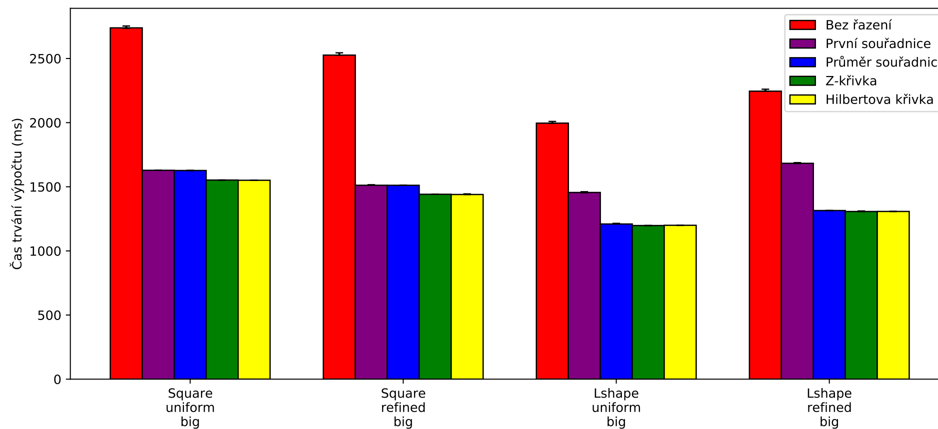
U testu velkých matic se efekt uspořádání evidentně projevuje intenzivněji, což je v souladu s očekáváním, neboť u velké sítě bude nepochybně docházet k výpadkům z cache častěji.

7.2.2 Testy nad 3D sítí



Obrázek 16: Výsledky 3D testu výpočtu lokální matice pro malou síť

Při testu nad malou 3D sítí jeví všechny metody znatelně menší efekt, což je pravděpodobně způsobeno horší teoretickou lokalitou dat vzhledem k redukci z vyššího množství dimenzí. Ani zde nejsou rozdíly mezi metodami příliš velké a Z-křivka s Hilbertovou křivkou zde vykazují téměř totožný efekt.



Obrázek 17: Výsledky 3D testu výpočtu lokální matice pro velkou síť

V ve 3D testu se efekt u větší sítě projevuje výrazněji. Zde si na rozdíl od předchozích testů můžeme všimnout mírně horšího efektu u metody redukce na první souřadnici u sítí typu Lshape. To je pravděpodobně způsobeno z pohledu metody náhlou vyšší koncentrací elementů na hraně mezi body se zjemněním. Metody průměru souřadnic totiž tuto hranu prochází zešikma, takže jednotlivé zhuštěné body jsou u ní více rozptýleny. Metodu redukce na první souřadnici tuto výhodu postrádá a tyto bodu rozprostřené po souřadnici y bude metoda vnímat jako obklopující jeden bod (viz Obrázek 13).

7.3 Test asemblace globální matice

Aby bylo možné získat reálnější představu o urychlení chodu Flow123d. Byla stejným způsobem zároveň otestována i asemblace celé globální matice. Globální matice je řídká matice velkých rozměrů (např. 100000×100000). Flow123d využívá k efektivní práci s řídkými maticemi knihovnu PETSc.

Knihovna PETSc umožňuje efektivní uložení řídkých matic a implementuje klíčové operace prováděné s maticí tuhosti. PETSc ukládá řídkou matici jako pole jejich nenulových hodnot společně s poli jednotlivých souřadnic představujících pozici daného nenulového prvku v matici.

Výpočet globální matice zahrnuje mimo jiné výpočet lokální matice pro každý element sítě.

Jelikož jsou testy kompilované v režimu `release`, který využívá optimalizační paramatr `-O3`. Musí test vracet nějakou formu kontrolního součtu, aby v rámci optimalizací nedošlo k vynechání části kódu z důvodu následného nepoužití výsledku.

```
Mat getGlobalMatrix(Mesh * mesh) {
    PetscInt dofs[4];
    Mat global_matrix;
    int n_global_dofs = mesh->n_nodes();
    MatCreateSeqAIJ(PETSC_COMM_SELF, n_global_dofs, n_global_dofs, 34,
```

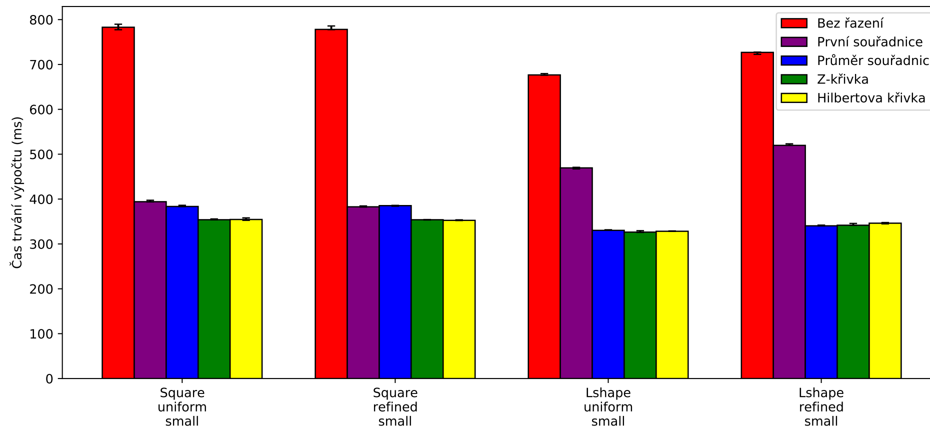
```
                    NULL, &global_matrix);
    MatAssemblyBegin(global_matrix, MAT_FINAL_ASSEMBLY);
    for (ElementAccessor<3> elm : mesh->elements_range()) {
        auto n0 = elm.node(0);
        auto n1 = elm.node(1);
        auto n2 = elm.node(2);
        auto n3 = elm.node(3);
        arma::mat::fixed<3, 3> M;
        M.col(0) = *n1 - *n0;
        M.col(1) = *n2 - *n0;
        M.col(2) = *n3 - *n0;
        double detM = arma::det(M);
        double jac = std::abs(detM) / 6;
        arma::mat::fixed<4, 4> phi_coefs = {{1, -1, -1, -1},
                                            {0,  1,  0,  0},
                                            {0,  0,  1,  0},
                                            {0,  0,  0,  1}};
        arma::mat::fixed<4, 3> grad_phi = {{-1, -1, -1},
                                           { 1,  0,  0},
                                           { 0,  1,  0},
                                           { 0,  0,  1}};
        arma::mat::fixed<4, 4> local_matrix = {{0, 0, 0, 0},
                                               {0, 0, 0, 0},
                                               {0, 0, 0, 0},
                                               {0, 0, 0, 0}};
        for (uint i = 0; i < 4; ++i) {
            for (uint j = 0; j < 4; ++j) {
                local_matrix(i, j) += arma::dot(M * grad_phi[i],
                                                M * grad_phi[j]) * jac;
            }
        }
        for(uint i = 0; i <= elm.dim(); ++i) {
            dofs[i] = elm.node(i).idx();
        }
        MatSetValues(global_matrix, 4, dofs, 4, dofs,
                     &local_matrix[0], ADD_VALUES);
    }
    MatAssemblyEnd(global_matrix, MAT_FINAL_ASSEMBLY);
    PetscScalar result;
    int idx = n_global_dofs - 1;
    MatGetValues(global_matrix, 1, &idx, 1, &idx, &result);
    MatDestroy(&global_matrix);
    return result;
}
```

Test byl implementován pouze pro 3D sítě. Tento test vlastně obsahuje zároveň test předchozí, neboť globální matici je nutné sestavit z tolika lokálních matic, kolik obsahuje síť elementů. I v rámci tohoto testu se tedy provádí výpočty lokálních matic.

Při vytvoření globální řídké matice pomocí PETSc knihovny je konstantně určen počet nenulových prvků na řádek matice. Flow123d ale počty nenulových prvků na řádek vypočítává. Lze tedy v praxi očekávat ještě větší efekt uspořádání než byl v rámci této práce naměřen díky nižší paměťové náročnosti matice sestavené s přesně

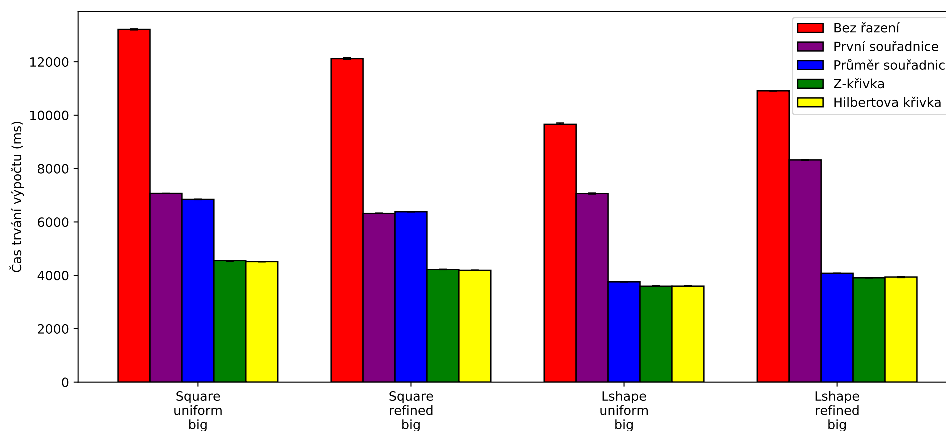
definovanými počty nenulových prvků na řádek.

U tohoto testu je očekáván větší efekt přeuspořádání neboď při asemblaci globální matice dochází k velmi častému překopírovávání dat (z lokálních matic do globální), což je operace v principu nelokální.



Obrázek 18: Výsledky 3D testu asemblace globální matice pro malou síť

Efekt přeuspořádání je zde skutečně větší než tomu bylo u pouhých výpočtů lokálních matic. I zde se ale silněji projevuje nevhodnost metody redukce na první souřadnici při použití sítě typu Lshape.



Obrázek 19: Výsledky 3D testu asemblace globální matice pro velkou síť

Zde již výsledky odpovídají i více než dvojnásobnému zrychlení. I zde lze pozorovat prakticky totožné výsledky u Z-křivky a Hilbertovy křivky. Pravděpodobně je tedy pro seřazení prvků sítě klíčová jejich postupná segmentace na základě blízkosti a nikoliv pořadí v jakém po sobě segmenty následují. Co se týče metod Redukce na první souřadnici a redukce na průměr souřadnic, ty mají u sítí typu Square téměř stejné výsledky, ale typu Lshape se opět rozcházejí.

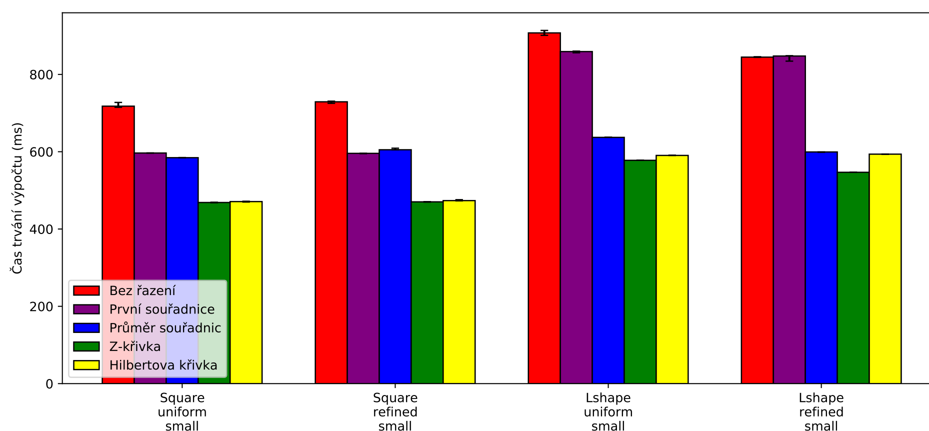
7.4 Násobení globální maticí

Kromě toho, že MeshOptimizer díky zlepšení cache lokality dat urychluje samotné sestavení matice, výsledná matice zároveň sama následně obsahuje lokalizovaná data. Urychlené by proto díky MeshOptimizeru měly být i následné operace s touto maticí.

Pro ověření byl vytvořen následující test násobení globální maticí.

```
double performMultiplication(Mat& global_matrix, PetscInt n_global_dofs) {
    Vec x, y;
    VecCreateSeq(PETSC_COMM_SELF, n_global_dofs, &x);
    VecDuplicate(x, &y);
    for (uint i = 0; i < 200; ++i) {
        MatMult(global_matrix, x, y);
    }
    PetscInt ix = 0;
    PetscScalar result;
    VecGetValues(y, 1, &ix, &result);
    return result;
}
```

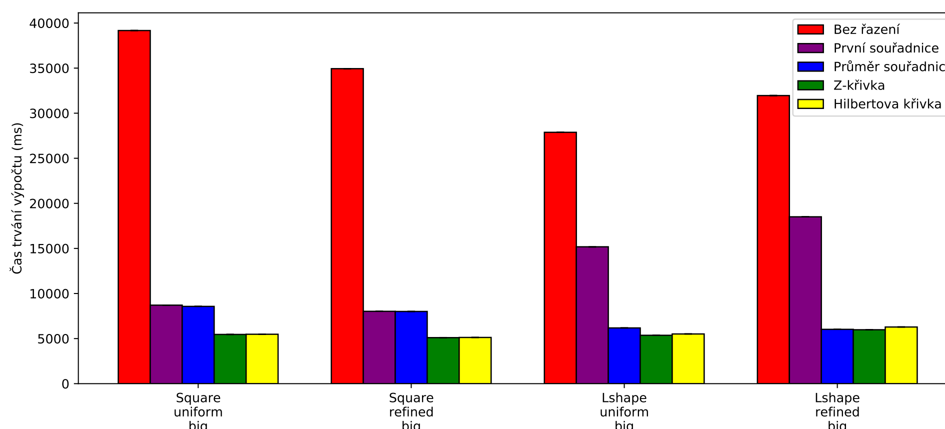
Jelikož samotné jedno násobení maticí je velmi rychlá záležitost, aby bylo možné naměřit porovnatelný čas, je nutné operaci opakovat. Násobení se tedy v tomto testu opakuje 200krát.



Obrázek 20: Výsledky 3D testu násobení globální matice pro malou síť

U malých sítí je efekt přeuspořádání výrazně menší než je tomu u předchozích testů, neboť efektivně uložená řídká matice tuhosti má mnohem nižší paměťovou náročnost než samotná síť, takže se daleko lépe vejde do cache paměti procesoru než výpočetní síť. Při malém počtu nodů pak zde nedochází tak často k výpadkům z cache paměti a přeuspořádání dat se tedy tolik neprojeví. Můžeme zde dokonce pozorovat situaci, kdy je efekt Hilbertovy křivky menší než efekt Z-křivky. vzhledem k velikosti chyby měření to nemůže být nahodilý jev vzniklý při měření, nicméně je zřejmé, že nejkritičtější oblast sítě pro křivky vyplňující prostor je její střed, kde se body

nacházejí na rozhraní čtyř segmentů se skokově odlišnými výslednými hodnotami na křivce. Síť Lshape refined obsahuje zhuštění právě v této oblasti. Je tedy možné, že zde hraje značnou roli konkrétní výskyt bodů ve vztahu ke tvaru použité křivky. Pravděpodobně nejzásadnější jev který Hilbertově křivce škodí, je u sítě typu Lshape její chybějící výskyt elementů v kvadrantu, který je z pohledu Hilbertovy křivky křivky třetí v pořadí. Rozdíl hodnot na křivce pro oblast pod tímto kvadrantem pak bude výrazněji odlišný od vedlejšího kvadrantu než u Z-křivky, která nevyplněný kvadrant prochází až jako poslední, takže díky tomu nedojde k nijak velkému skoku mezi hodnotami elementů vyskytujících se v ostatních kvadrantech.



Obrázek 21: Výsledky 3D testu násobení globální matice pro velkou síť

Efekt přeuspořádání se v souladu s očekáváním projevil daleko výrazněji v případě velkých sítí. Zde je však tento efekt Daleko výraznější než u předchozích testů. Pokud tedy ve Flow123d dochází k častému násobení maticí tuhosti, bude mít využití MeshOptimizeru významný vliv na chod Flow123d.

8 Závěr

V rámci této práce jsem se obeznámil s implementací Flow123d v oblasti MKP a datovými strukturami a funkcemi knihoven, které Flow123d k výpočtům používá. V rámci plnění zadání této práce, která spočívala v nalezení způsobu, jak pomocí křivek vyplňujících prostor přeuspořádat prvky datových struktur pro lepší využití cache paměti procesoru, jsem vytvořil speciální třídu umožňující jednoduchým způsobem pomocí několika metod přeuspořádat prvky sítě, následkem čehož dochází k urychlení klíčových výpočtů prováděných v rámci Flow123d.

Pomocí unit testu jsem testoval vliv jednotlivých metod výpočtu indexu pro přeuspořádání a prokázal, že výsledně dochází k výraznému zrychlení asemblace globální matice a dalších operací s ní následně prováděných.

Přestože bylo v rámci testování zjištěno, že podobného urychlení je možné dosáhnout i pomocí jednodušších metod, jako nejúčinnější se projevili právě křivky vyplňující prostor.

Podařilo se splnit zadání a prokázat správnost zvolené strategie k dosažení jeho stanoveného cíle.

Výsledkem praktické části mé práce je třída MeshOptimizer, které je připravena být plně využita v rámci programu Flow123d.

Literatura

[1] AMD A8-Series A8-7410 - AM7410ITJ44JB. *AMD A8-Series A8-7410 - AM7410ITJ44JB* [online]. [vid. 2020-05-31]. Dostupné z: http://www.cpu-world.com/CPUs/Puma/AMD-A8-Series%20A8-7410.html

[2] BADER, Michael. *Space-filling curves: an introduction with applications in scientific computing.* Springer Science & Business Media, 2012.

[3] Hilbert curve. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2020 [vid. 2020-05-31]. Dostupné z: https://en.wikipedia.org/wiki/Hilbert_curve

[4] HP Pavilion Notebook - 15-ab124nc (ENERGY STAR) Produktové Specifikace. *HP Pavilion Notebook - 15-ab124nc (ENERGY STAR) Produktové Specifikace | HP® Customer Support* [online]. [vid. 2020-05-31]. Dostupné z: https://support.hp.com/us-en/document/c04843208/

[5] How L1 and L2 CPU Caches Work, and Why They're an Essential Part of Modern Chips. *How L1 and L2 CPU Caches Work, and Why They're an Essential Part of Modern Chips - ExtremeTech* [online]. [vid. 2020-05-31]. Dostupné z: *https://www.extremetech.com/extreme/188776-how-l1-and-l2-cpu-caches-work-and-why-theyre-an-essential-part-of-modern-chips*

[6] KOWARSCHIK, Markus; WEIß, Christian. An overview of cache optimization techniques and cache-aware numerical algorithms. In: *Algorithms for memory hierarchies*. Springer, Berlin, Heidelberg, 2003. p. 213-232.

[7] Z-order curve. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2020 [vid. 2020-05-31]. Dostupné z: https://en.wikipedia.org/wiki/Z-order_curve

Obsah příloh

- zip archiv s kompletními výsledky testů měřených v mikrosekundách včetně python skriptu pro jejich vizualizaci.