FAKULTA INFORMAČNÍCH TECHNOLOGIÍ, VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Architektury výpočetních systémů (AVS 2020)
Projekt č. 2: Paralelizace s OpenMP

Mrázek Vojtěch <mrazek@fit.vutbr.cz>
Filip Vaverka <ivaverka@fit.vutbr.cz>

Termín odevzdání: 18. prosince 2020

1 Úvod

Primárním cílem tohoto projektu je vyzkoušet si přechod od sekvenčního k paralelnímu algoritmu a jeho praktickou implementaci a optimalizaci. Druhotným cílem je pak osvojit si metriky používané při hodnocení efektivity paralelních algoritmů. Vaším prvním úkolem bude paralelizace naivního algoritmu pro rekonstrukci polygonálního povrchu ze skalárního pole ve 3D prostoru (tzv. "Marching Cubes") a vyhodnocení vlastností tohoto řešení. Druhým úkolem bude optimalizace tohoto řešení na úrovni algoritmu pomocí hierarchického dělení prostoru (tzv. "Octree") a rychlé eliminace prázdného prostoru. Posledním úkolem je implementace řešení za použití paralelních smyček a předem vypočtených hodnot, které eliminuje nadbytečné výpočty za cenu většího využití paměti.

2 Superpočítač Salomon

Superpočítač Salomon provozovaný organizací IT4Innovations v Ostravě je složen z 1009 uzlů, každý uzel disponuje dvěma procesory Intel Xeon E5-2680v3. Každý z těchto 12ti jádrových procesorů založených na mikroarchitektuře Haswell[1] je vybaven 64 GB RAM (128 GB na uzel) ve čtyř kanálové konfiguraci (až 68 GB/s).
Pro připojení k superpočítači Salomon je třeba mít vytvořený účet, se kterým je možné se připojit na tzv. "login" uzel – `salomon.it4it.cz`. Tento uzel **neslouží** ke spouštění náročných úloh, veškeré experimenty je nutné provádět na výpočetních uzlech. Tento projekt není

[1] https://en.wikichip.org/wiki/intel/microarchitectures/haswell_(client)

výpočetně náročný, přesto by aktivita jiných uživatelů na login uzlu mohla zkreslit měření výkonosti.
Pro účely tohoto projektu je nejjednodušším řešením vytvořit *interaktivní úlohu*. Příkaz `qsub` zadá požadavek na spuštění úlohy do fronty, jakmile bude v systému dostatek volných uzlů, dojde ke spuštění úlohy.
Parametr `-A` určuje projekt, v rámci kterého máme alokované výpočetní hodiny (neměnit), `-q` určuje frontu, do které bude úloha zařazena (pokud nebude úloha dlouhou dobu spuštěna, můžete použít frontu `qprod`, ale preferujte `qexp`), parametr `-l` určuje zdroje, které budou úloze přiděleny (počet uzlů, počet procesorů, čas). Abyste předešli zkreslení výkonových statistik, vždy alokujte celý uzel, tj. 24 jader. Interaktivní úlohu pak získáte parametrem `-I`. Více o spouštění úloh na superpočítačích IT4I naleznete na stránce https://docs.it4i.cz/general/job-submission-and-execution/ . Výsledný příkaz pro získání interaktivní úlohy pak vypadá takto:

```
[dd-20-28-263@login1.salomon ~]$ qsub -A DD-20-28 -q qexp -l select=1:ncpus=24,walltime=1:00:00 -I
```

Software na superpočítači Salomon je dostupný pomocí tzv. *modulů*. Tyto moduly je potřeba před použitím načíst, jak pro kompilaci na login uzlu, tak po každém spuštění interaktivní úlohy. V tomto projektu budou potřeba moduly `intel`, `CMake` a `Python`:

```
ml intel/2020a CMake/3.16.4-GCCcore-9.3.0 Anaconda3/2019.10
```

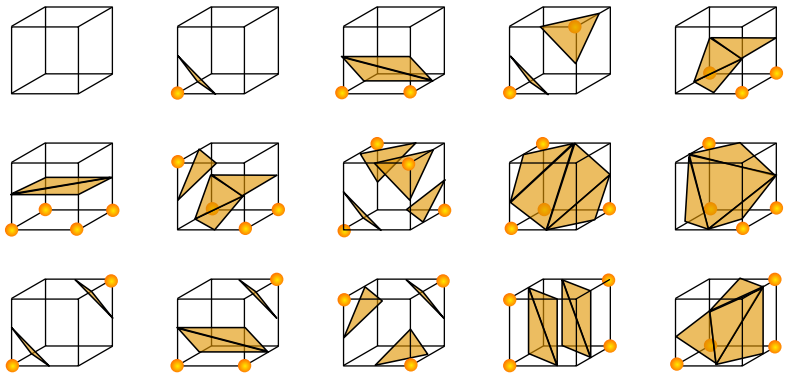
Příkaz `ml` zajišťuje práci s balíčky software, moduly. Modul `intel` zahrnuje `C/C++` kompilátor firmy Intel, který je možné vyvolat příkazy `icc` resp. `icpc`. Kompilátor firmy Intel je vybrán z důvodu čitelnějších optimalizačních výstupů. Dobrovolně můžete porovnat výkon výsledného řešení kompilovaného jiným kompilátorem (například GNU).
Modul `CMake` je použit jako systém pro překlad projektu a `Python` pro pomocné skripty pro práci se vstupními daty.

3 Rekonstrukce povrchu pomocí Marching Cubes

Ačkoli implementace samotného jádra algoritmu "Marching Cubes"[2] není vaším úkolem, je důležité získat učitou představu o jeho fungování. Marching cubes je algoritmus využívaný v počítačové grafice pro rekonstrukci polygonové sítě (povrchu/isosurface[3]) ze skalárního pole [4]. Typicky je skalární pole získáváno jako 3D pole například jako data z CT nebo MRI. Pro účely projektu je však skalární pole vyhodnocováno za běhu a pouze v bodech kde je nutné znát jeho hodnotu.
Samotný algoritmus je založen na rozdělení skalárního pole do krychlí o určité velikosti (udává rozlišení výsledné polygonové sítě). Při následném průchodu takto rozděleného pole je na

[2] https://en.wikipedia.org/wiki/Marching_cubes
[3] https://en.wikipedia.org/wiki/Isosurface
[4] https://en.wikipedia.org/wiki/Scalar_field

základě hodnot ve vrcholech každé krychle určena konfigurace polygonů potřebná pro reprezentaci části povrchu procházejícího danou krychlí. Vhodná konfigurace polygonů je vybrána z $2^8 = 256$ možných konfigurací (obrázek 3.1) tak, že hodnoty ve vrcholech krychle jsou převedeny do binární podoby porovnáním s požadovanou hodnotou na povrchu (isosurface level) – těchto 8 binárních hodnot pak tvoří 8-bit ukazatel do tabulky možných konfigurací. Pro příklad uvažujme krychli, kde všechny její vrcholy mají hodnotu menší než požadovaná hodnota – dostáváme 8-bit index 0 (`00000000b`), který ukazuje na konfiguraci bez polygonů (celá krychle leží pod povrchem). Stejný výsledek získáme, pokud celá krychle leží nad povrchem – index 255 (`11111111b`).



Obrázek 3.1: 16 z 256 možných konfigurací polygonů

Konečně posledním krokem je lineární interpolace vygenerovaných vrcholů podél hrany krychle, tak aby výsledný vrchol ležel přesně na průsečíku mezi povrchem a příslušnou hranou krychle. Obrázek 3.1 ukazuje stav před interpolací a vrcholy tedy vždy leží ve středu hrany.
Algoritmus tedy může být rozdělen do tří pomyslných částí: rozdělení a průchod skalárním polem, výpočet hodnoty skalárního pole v daném bodě a vytvoření výstupních polygonů. Tento projekt je zaměřen na první dvě z těchto částí, které mohou být snadno paralelizovány pomocí OpenMP.

4 Struktura projektu, překlad a spuštění

Archiv zadání obsahuje:

```
PMC-xlogin00.txt
data/
scripts/
src/
```

Soubor `PMC-xlogin00.txt` obsahuje otázky, které je třeba zodpovědět v průběhu řešení projektu. Složka `data` osahuje data pro testování a měření výkonnosti vašeho řešení. Složka `scripts` obsahuje skripty pro generování vstupních dat a grafů škálování. Konečně složka `src` obahuje zdrojový kód projektu.

Zdrojový kód projektu je strukturován do několika tříd, které kopírují základní části algoritmu marching cubes. Hlavní část algoritmu je rozdělena mezi virtuální třídu `BaseMeshBuilder` a z ní dědící třídu `RefMeshBuilder`. Třída `BaseMeshBuilder` obsahuje kód samotného generování trojúhelníků, ale způsob průchodu skalárním polem a výpočet hodnoty v bodě je delegován virtuálními metodami (pure virtual) do dceřiné třídy `RefMeshBuilder`. Pro načítání a předzpracování vstupních dat slouží třída `ParametricScalarField`, která vstupní data zpřístupňuje v podobě kolekce bodů v 3D prostoru. A konečně prázdné třídy `LoopMeshBuilder`, `TreeMeshBuilder` a `CachedMeshBuilder` dědící z `BaseMeshBuilder` jsou připraveny pro vaše paralelní implementace (`RefMeshBuilder` je tedy vzorem pro vaše řešení). Zdrojový kód je rozdělen do několika souborů:

- `main.cpp`: Obsahuje vstupní bod programu, zpracování parametrů (knihovna cxxopts[5]), nastavení počtu OpenMP vláken a výběr požadované implementace (referenční, paralelní smyčka, paralelní strom).
- `common/vector_helpers.h`: Obsahuje pomocnou třídu pro reprezentaci bodu v 3D prostoru (`Vec3_t`) a nástroje pro přesné měření realného času pomocí C++11.
- `common/base_mesh_builder.*`: Bázová třída pro vaše paralelní implementace Marching cubes.
- `common/parametric_scalar_field.*`: Třída reprezentující skalární pole jako množinu bodů v 3D prostoru.
- `common/ref_mesh_builder.*`: Referenční implementace třídy `BaseMeshBuilder`. Její metody `marchCubes`, `evaluateFieldAt` a `emitTriangle` implementují průchod polem, vyhodnocení hodnoty pole v bodě a uložení jednoho vygenerovaného trojúhelníku. Metoda `getTrianglesArray` naopak vrací ukazatel na pole obsahující všechny vygenerované trojúhelníky.
- `parallel_builder/loop_mesh_builder.*`: Bude obsahovat vaši implementaci (obdobu `RefMeshBuilder`) pomocí paralelních smyček OpenMP.
- `parallel_builder/tree_mesh_builder.*`: Bude obsahovat vaši implementaci (obdobu `RefMeshBuilder`) optimalizovanou pomocí paralelního stromového průchodu a OpenMP tasků.
- `parallel_builder/cached_mesh_builder.*`: Bude obsahovat vaši implementaci (obdobu `RefMeshBuilder`) optimalizovanou pomocí paralelních smyček OpenMP a předpočítání hodnot.

Pro přeložení projektu je možné použít `CMake` následujícím způsobem:

```
~$ cd Assignment
~/Assignment$ mkdir build && cd build
~/Assignment/build$ CC=icc CXX=icpc cmake ../src                    \
    -DCMAKE_BUILD_TYPE=Release                                      \
    -DCMAKE_C_FLAGS='-march=native' -DCMAKE_CXX_FLAGS='-march=native'
~/Assignment$ make -j
```

[5] https://github.com/jarro2783/cxxopts

Výstupem překladu by měl být jediný spustitelný soubor `PMC` (Parallel Marching Cubes), jehož výstupem je:

```
~/Assignment/build$ ./PMC

AVS Assignment 2 - Parallel Marching Cubes
Usage:
  AVS: Marching cubes [OPTION...] INPUT <OUTPUT>

  -l, --level arg    Iso surface value (default: 0.15)
  -g, --grid arg     Grid size (default: 64)
  -b, --builder arg  Builder name [ref, loop, tree, cached] (default: ref)
  -t, --threads arg  Number of OpenMP threads (default: 0)
      --batch        Run in silent/batch mode
  -h, --help         Print help

Missing required arguments: "INPUT"
```

Argumenty programu mají následující významy:

- `INPUT` soubor
  Vstupní soubor obsahující body, které definují skalární pole.
- `OUTPUT` soubor
  Výstupní soubor pro vytvořený model ve formátu "Wavefront *.obj". Tento parametr lze vynechat – výstup je pak zahozen (vhodné pro měření rychlosti).
- `--level l`
  Udává hodnotu "isosurface", tedy vzdálenost k nejbližšímu bodu ve vstupním souboru, na které leží hledaný povrch. Ve zdrojovém kódu lze tuto honotu přečíst z proměnné `mIsoLevel` ve třídě `BaseMeshBuilder`, nebo voláním metody `getIsoLevel` na objektu třídy `ParametricScalarField`.
- `--grid g`
  Udává délku hrany diskretizační mřížky a s ní rozlišení diskretizace a celkový počet "Marching cubes" ($N = g^3$). Hodnotu tohoto parametru lze získat v proměnné `mGridSize` ve třídě `BaseMeshBuilder`
- `--builder`
  Umožňuje zvolit konkrétní implementaci algoritmu "Marching cubes". Možnosti jsou:
  - `ref` Referenční implementace v souborech `common/ref_mesh_builder.*`
  - `loop` Vaše implementace pomocí paralelní smyčky (viz sekce 6.1)
  - `tree` Vaše implementace pomocí paralelního průchodu stromem (viz sekce 6.2)
  - `cached` Vaše implementace pomocí předpočítání hodnot pole (viz sekce 6.3)
- `--threads N`
  Maximální počet OpenMP vláken (pokud není hodnota specifikována použije se nastavení prostředí). Nastavení je realizováno voláním funkce `omp_set_num_threads` z knihovny OpenMP.

- `--batch`
  Přepnutí do režimu minimálních výpisů ve formátu CSV, vhodné pro vytváření tabulek a grafů.

Spuštění programu s některým ze vstupních souborů ze složky `data` vygeneruje příslušný polygonální povrch:

```
~/Assignment/build$ ./PMC ../data/bun_zipper_res4.pts bun_zipper_res4.obj

======================== Marching Cubes Mesh Builder ========================
Mesh Builder:        Reference
Input Field File:    ../data/bun_zipper_res4.pts
Output Mesh File:    bun_zipper_res4.obj
Grid Size:           64
Iso Level:           0.15
Field Elements:      453
============================== Building the Mesh ==============================
Number of Threads:   16
Elapsed Time:        8679 ms
Mesh Triangle Count: 42476
============================ Writting Output File =============================
Output File Size:    5441 KB
================================== DONE =====================================
```

Výstup programu ukazuje, že byla použita referenční implementace na vstupním souboru `../data/bun_zipper_res4.pts` a výstup byl uložen do souboru `bun_zipper_res4.obj`. Hrana mřížky byla 64, "iso-level" měl hodnotu 0.15 a vstupní soubor obsahoval 453 bodů. Výstupní model byl vygenerován za 8679 ms, obsahuje 42476 trojúhelníků a soubor má velikost 5441 KB.

Formát vstupních a výstupních souborů

Vstupní soubory obsahují sekvenci bodů v 3D prostoru:

```
../data/bun_zipper_res4.pts

p -0.312216 1.263040 0.051492
p -0.446774 1.312040 0.057048
p -0.683011 1.448280 0.413688
...
```

Výstupem je polygonální síť ve formátu "Wavefront .obj", soubor tedy obsahuje seznam vrcholů a seznam trojic indexů spojující vrcholy do trojúhelníků:

```
bun_zipper_res4.obj

v 9.88281631 8.1788826 0.339680344
v 9.86799812 8.1788826 0.340786725
v 9.88281631 8.17489433 0.340786755
...
f 1 2 3
f 4 5 6
f 7 8 9
...
```

Soubor tohoto formátu lze vizualizovat většinou nástrojů pro práci s 3D grafikou, například volně dostupným Blender[6].

5 Výstup a hodnocení projektu

Výstupem projektu bude soubor `xlogin00.zip` obsahující vámi modifikované soubory, textový soubor s otázkami a grafy škálování (viz sekce 6.4). Do textového souboru uveďte výsledky měření a zodpovězte požadované otázky. V každém souboru (který jste změnili) nezapomeňte uvést svůj login! Hodnotit se bude jak funkčnost a správnost implementace, tak textový komentář – ten by měl dostatečně popisovat výsledky měření a odpovídat na otázky uvedené v zadání. Hodnocení je uvedené u jednotlivých částí zadání a dohromady tvoří 25 bodů. Projekt odevzdejte v uvedeném termínu do informačního systému.

Shrnutí obsahu odevzdávaného archivu:

- `parallel_builder/loop_mesh_builder.{h/cpp}`
  Vaše implementace pomocí paralelních smyček – sekce 6.1
- `parallel_builder/tree_mesh_builder.{h/cpp}`
  Vaše implementace pomocí paralelního stromu – sekce 6.2
- `parallel_builder/cached_mesh_builder.{h/cpp}`
  Vaše implementace pomocí předpočítání hodnot pole – sekce 6.3
- `PMC-xlogin00.txt` (přejmenujte dle vašeho loginu)
  Odpovědi na uvedené otázky.
- `input_scaling_strong.png`, `input_scaling_weak.png` a `grid_scaling.png`
  Grafy silného a slabého škálování vzhledem ke vstupu a graf škálování vzhledem k velikosti mřížky.

[6] https://www.blender.org/

6 Paralelní Marching Cubes (25 bodů)

Cílem projektu je vytvořit trojici tříd implementující hlavní části algoritmu Marching cubes, tak aby bylo možné využít více-jádrové CPU. Výpis 1 ukazuje signaturu těchto tříd a referenční (sekvenční) implementaci je možné nalézt v souboru `common/ref_mesh_builder.cpp`.

```
1  class MyMeshBuilder : public BaseMeshBuilder
2  {
3  public:
4      MyMeshBuilder(unsigned gridEdgeSize);
5
6  protected:
7      unsigned marchCubes(const ParametricScalarField &field);
8      float evaluateFieldAt(const Vec3_t<float> &pos, const ParametricScalarField &field);
9      void emitTriangle(const Triangle_t &triangle);
10     const Triangle_t *getTrianglesArray() const;
11 };
```

Listing 1: Signatura tříd, které je třeba implementovat.

Každá z těchto tříd musí implementovat následující virtuální metody:

- `unsigned marchCubes(const ParametricScalarField &field)`
  Implementace průchodu mřížkou, tedy každou pozicí v 3D prostoru, kde je třeba vygenerovat polygony. Pro samotné vyhodnocení/vygenerování polygonů na dané pozici lze zavolat metodu `unsigned buildCube(/* position */, /* field */)`, která je definovaná ve třídě `BaseMeshBuilder` a očekává pozici krychle v mřížce a příslušné pole "field" a vrací počet vygenerovaných trojúhelníků.
  Metoda `marchCubes` je volána metodou `BaseMeshBuilder::buildMesh`, která očekává, že návratová hodnota `marchCubes` bude celkový počet vygenerovaných polygonů.

- `float evaluateFieldAt(const Vec3_t<float> &pos, const /* ... */ &field)`
  Implementuje výpočet hodnoty skalárního pole `field` v bodě `pos` a vrací vypočtenou hodnotu. Tato hodnota je dána výrazem 6.1, který je vlastně pouhým nalezením vzdálenosti od bodu `pos` k nejbližšímu bodu s v poli `field`.

  $$\min_{s \in F} \sqrt{\sum_{\xi \in \{x,y,z\}} (s_\xi - p_\xi)^2} \tag{6.1}$$

  Kde F je pole `field`, s je jeden z bodů definující pole F, $i \in \{x, y, z\}$ označuje dimenze a p je bod `pos`.

  POZOR: Tato metoda je volána z `BaseMeshBuilder::buildCube`!

- `void emitTriangle(const Triangle_t &triangle)`
  Metoda je volána po vygenerování každého trojúhelníku a jejím úkolem je uložit daný trojúhelník do vhodné datové struktury (v referenční implementaci postačuje C++ vektor). Metoda může být volána několikrát v rámci jednoho volání `BaseMeshBuilder::buildCube` což vyžaduje zvýšenou pozornost.

  POZOR: Tato metoda je volána z `BaseMeshBuilder::buildCube`!

- `const Triangle_t *getTrianglesArray() const`
  Metoda vrací ukazatel na pole trojúhelníků (tj. oblast paměti, kde jsou trojúhelníky těsně za sebou) a je volána pouze 1× z `BaseMeshBuilder::buildMesh` těsně před uložením polygonů do výstupního souboru. Pořadí trojúhelníků v tomto poli je nedefinované.

Uspořádání prostoru

Pro práci s již implementovanými metodami jako `buildCube` je důležité korektní předávání souřadnic jako pozice krychle. Kód pracuje pouze v kladném oktantu, všechny souřadnice tedy leží v intervalu $\langle 0, S_{i\in\{x,y,z\}}\rangle$, kde S_i je délka hrany osově zarovnaného kvádru kolem vstupních dat. Vstupní data jsou po načtení do tohoto prostoru mapovány jednoduchým posunutím.
Rozlišení mřížky (tedy velikost jedné "Marching cube") je vypočtena jako

$$\texttt{mGridResolution} = \frac{\max_{i\in\{x,y,z\}} S_i}{\texttt{mGridSize}}. \tag{6.2}$$

Tím vzniká prostor celočíselných indexů $0, 1, \ldots, \frac{\texttt{mGridSize}}{\texttt{mGridResolution}}$ v každé dimenzi. Metoda `buildCube` očekává souřadnice krychle právě v těchto celočíselných indexech. Naproti tomu metoda `evaluateFieldAt` očekává pozici v původním spojitém prostoru! Výstupní polygony jsou taktéž v původním spojitém prostoru. **Při řešení předpokládejte pouze velikosti mřížek v mocninách 2 (**$\texttt{mGridSize} \in 2^i$**)** – což znatelně zjednodušuje stromovou implementaci (viz dále).

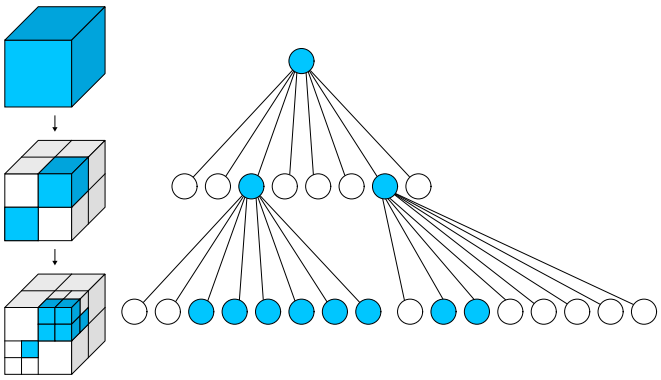
6.1 Paralelizace původního řešení (5 bodů)

Prvním úkolem je tedy vytvořit paralelní implementaci třídy `LoopMeshBuilder` v souborech `parallel_builder/loop_mesh_builder.{h/cpp}`. Jako výchozí kód můžete použít referenční implementaci v `common/ref_mesh_builder.{h/cpp}`, která obsahuje dvě vhodné smyčky: průchod mřížkou v `marchCubes` a výpočet minimální vzdálenosti v `evaluateFieldAt`.
Pro paralelizaci použijte vhodné prostředky OpenMP.
V rámci úlohy vyzkoušejte paralelizaci obou smyček (nezávisle) a jako výsledné řešení odevzdejte to, které bude vykazovat lepší výsledky. Pokud se pokusíte paralelizovat obě smyčky zároveň, nezapomeňte, že půjde o vnořený paralelismus (`evaluateFieldAt` je efektivně volána ze smyčky procházející mřížkou).
Při paralelizaci smyček také ověřte vliv různých možností rozdělení práce mezi vlákna (OpenMP static/dynamic/guided sheduling) a výsledná zjištění stručně popište v textovém souboru s odpověďmi.

6.2 Paralelní průchod stromem (10 bodů)

Druhým úkolem je vytvořit optimalizovanou paralelní implementaci třídy `TreeMeshBuilder` v souborech `parallel_builder/tree_mesh_builder.{h/cpp}`. V tomto případě je však cílem nejen paralelizace, ale také optimalizace samotného algoritmu průchodu mřížkou. Vzhledem k tomu, že hledaný povrch se nachází ve specifikované vzdálenosti od nejbližšího bodu s

(výraz 6.1 v kombinaci s parametrem programu) lze relativně snadno určit oblast mřížky, kde není možné aby jí procházel hledaný povrch. To umožňuje optimalizovat průchod mřížkou tak, že se tyto oblasti vyloučí a nebude se pro ně volat metoda `buildCube`.
Typickým přístupem k řešení tohoto problému je hierarchická dekompozice prostoru. V našem případě použijeme stromovou dekompozici (Octree decomposition[7]), kde se prostor dělí na každé úrovni na 8 potomků (viz obrázek 6.1).



Obrázek 6.1: Octree dekompozice, kde modře vyznačenými bloky prochází hledaný povrch.

Průchod takovýmto stromem je pak následující:

- Aktuální blok (na začátku celá mřížka) je rozdělena do 8 potomků (bloky o poloviční délce hrany).
- Pro každého potomka ověříme, zda je možné aby jeho podprostorem procházel hledaný povrch (isosurface). Pro účely tohoto projektu lze tento test snadno aproximovat pomocí opsané koule kolem aktuálního bloku (povrch je totiž také definovaný množinou bodů a vzdáleností – tedy množinou koulí na různých pozicích). Podmínka prázdnosti bloku je tedy
$$F(p) > l + \frac{\sqrt{3}}{2}a \tag{6.3}$$
Kde $F(p)$ je hodnota pole (tedy 6.1) ve středu bloku, l je aktuální "iso-level" a a je délka hrany bloku.
- Konečně, každý neprázdný potomek je rozdělen na dalších 8 potomků až do předem specifikované hloubky (tzv. "cut-off"), nebo velikosti hrany a (která je zmenšena na polovinu s každým zanořením).
- Na nejnižší úrovni jsou pak voláním `BaseMeshBuilder::buildCube` (jako v předchozím případě) vygenerovány samotné polygony pro všechny krychle náležející do daného podprostoru.

[7] https://en.wikipedia.org/wiki/Octree

Je zřejmé, že pro tento rekurzivní proces jsou vhodným mechanismem pro paralelizaci OpenMP tasky a jejich využití je v tomto případě tedy povinné! Nezapomeňte také, že metoda `marchCubes` vrací počet vygenerovaných trojúhelníků. Jelikož tento počet bude známý až po zanoření na poslední úroveň je vhodné použít synchronizaci tasků (rodič čeká na potomky) a případně atomické operace pro sesbírání počtu od každého potomka.
Konečně, pro výpočet "fyzické" velikosti bloků stromu a pozice jejich středů je možno použít třídní proměnnou `mGridResolution` (deklarováno v `BaseMeshBuilder`). Proměnná `mIsoLevel` obsahuje aktuální "iso-level" potřebný pro vyhodnocení prázdnosti bloku.

6.3 Před-výpočet dat mřížky (5 bodů)

V obou předchozích případech typicky dochází k několikanásobnému vyhodnocení skalárního pole ve stejném bodě (sdílené vrcholy sousedních krychlí). Třetí úlohou je tedy implementovat třídu `CachedMeshBuilder` (`parallel_builder/cached_mesh_builder.{h/cpp}`) a vyzkoušet přístup, kdy je tento problém eliminován.
Nejjednodušším přístupem je v tomto případě alokovat pole, do kterého budou uloženy hodnoty výrazu 6.1 ve všech bodech p, které budou později třeba při sestrojování polygonů jako v předchozích případech. Souřadnice těchto bodů p je možné odvodit od celočíselných souřadnic kostky c jako

$$p_\xi = c_\xi \cdot \texttt{mGridResolution}, \tag{6.4}$$

jelikož její první vrchol leží právě v tomto bodě. Hodnoty v ostatních vrcholech kostky na pozici c, pak lze považovat za náležející jejím sousedům. Je však nutné pamatovat že toto nebude platit pro poslední pozice (bez souseda) v každé dimenzi a celkem je třeba hodnoty předpočítat v $(\texttt{mGridSize} + 1)$ bodech v každé dimenzi.
Metoda `CachedMeshBuilder::evaluateFieldAt` pak bude pouze číst hodnotu z příslušné pozice v předpočítaném poli. Tato metoda je však volána s reálnými pozicemi p, které je tedy nutné přepočítat zpět na souřadnice kostky a index v poli. Přepočet pozice vrcholu p na souřadnici kostky c lze snadno provést jako

$$c_\xi = \left\lfloor \frac{p_\xi}{\texttt{mGridResolution}} + \frac{1}{2} \right\rfloor, \tag{6.5}$$

kde koeficient 1/2 zabraňuje problémům při zaokrouhlení dolů.
Jak předvýpočet, tak samotné generování polygonů by mělo proběhnout v rámci metody `CachedMeshBuilder::marchCubes` (obě části tak budou započteny do celkového času).

6.4 Grafy škálování jednotlivých řešení (5 bodů)

Pro všechny varianty algoritmu vytvořte grafy silného a slabého škálování vzhledem k počtu bodů ve vstupním souboru a počtu vláken. Vytvořte také společný graf škálování vzhledem k velikosti mřížky pro hranu délky 8 – 512 bodů při použití maximálního počtu vláken.
Pro snadné vytvoření těchto měření a grafů zadání obsahuje následující skripty:

- `generate_data_and_plots.sh`
  V typickém případě by mělo být dostačující spustit tento skript z vašeho "build" ad-

resáře, který obsahuje přeložený projekt. Tento skript provede potřebná měření a vygeneruje požadované grafy (**pokud již existují soubory s výsledky měření NEBUDE opakováno**).

- `generate_plots.py`
  Skript pro samotné vytvoření grafu na základě naměřených dat.
- `measure_input_scaling.sh`
  Skript měřící data potřebná pro slabé i silné škálování vzhledem k počtu bodů ve vstupním souboru.
- `measure_grid_scaling.sh`
  Skript měřící data potřebná pro škálování vzhledem k velikosti mřížky za použití plného počtu vláken.

Dosažené výsledky stručně zhodnoťte (a pokud možno vysvětlete) v textovém souboru s odpověďmi na otázky.