Convex hull problem

Semestralni práce PJC B211

Přemek Bělka

Commit hash: d8d0545aea964fc3fe80b604e2129317ddb2df29

Zadání

Konvexní obal množiny bodů v rovině (convex hull)
Konvexní obal (convex hull) množiny bodů si lze jednoduše představit jako nejmenší konvexní mnohoúhelník, uvnitř kterého leží všechny body množiny. Jako konkrétní algoritmus je možné zvolit třeba Quickhull. Vstupem pro aplikaci bude textový soubor se souřadnicemi bodů, výstupem pak bude seznam vrcholů konvexního obalu. Doporučujeme si napsat jednoduchý generátor vstupních dat a program či skript pro převod vstupních a výstupních dat do obrázku SVG pro snadnou vizualizaci a ověření správnosti řešení.

Popis implementace

Pro jednovláknovou verzi algoritmu jsem implementoval algoritmus Quickhull podle pseudokódu dostupného na odkaze na wikipedii v zadání. Při implementaci vícevláknové verze jsem paralelizoval část, ve které algoritmus kontroluje, které body jsou uvnitř nebo vně trojúhelníku tvořeného bodem nejvíce vlevo, nejvíce vpravo a bodem, jehož vzdálnesost je nejdelší od úsečky mezi těmito dvěma body a poté rozdělí zbylé body na dvě části a rekurzivně se zanoří.

V mé implementaci jsem se rozhodl uvažovat pouze kladná celá čísla. Při generování náhodných bodů je momentálně v programu „hardcoded“ maximální délka os na int o velikosti 10.000.000.

Program podporuje export výsledku do .svg souboru, výpis na příkazovou řádku a do textového souboru.

Pro tvorbu testů jsem vygeneroval pár sad náhodných bodů v mém programu a za očekávaný výstup považuji výsledek z tohoto webu: https://planetcalc.com/8576/.

Výpis výsledku je ve formátu:
Convex hull body
##################
Vstupní množina bodů

Poznámka k paralelní implementaci

Předpokládám, že velkou část zdroju zabírá zamykání mutexů ve funkci add_to_result. Pokoušel jsem se proto i o implementaci pomocí future. Kdy funkce findhull nepřistupuje ke společným zdrojům, ale místo toho si průběžné výsledky ukládá při rekurzivním zanořování na stack a poté je spojuje do konečného výsledku. Bohužel se mi nepodařilo tímto způsobem dosáhnout žádného zlepšení, ale naopak zhoršení oproti verzi s mutexy. Domnívám se, že možným problémem v mé future implementaci by mohlo být, že čas, který naženu na odstranění mutexu, ztratím na kopírování mezivýsledků v rekurzivní funkci findhull. Přesto jsem se ale rozhodl tuto verzi zahrnout do odevzdávané práce a měření.

Verzi s mutexy se nachází ve složce quickhull_parallel, verze s futures v quickhull_parallel_future. Verze s futures je použita pouze v porovnání při měření pomocí přepínače –comparison. Při specifikování přepínače –multithreaded je použita automaticky pouze verze s mutexy.

Měřící prostředí

Macbook Air 2020
OS: MacOS Big Sur 11.1
RAM: 8 Gb
CPU: I3 1.1 GHz, 2 jádra
IDE: Clion 2021.1.3
Compiler info:

- Apple clang version 12.0.5 (clang-1205.0.22.11)
- Target: x86_64-apple-darwin20.2.0
- Thread model: posix

Měření

Měření jsem prováděl v release modu. Jako vstupní data využívám náhodně vygenerovaných 10000000 bodů, kde bod p = (x, y), $x,y \in \mathbb{N}$, $x,y \in \{0, 1, 2 \dots 10000000\}$.

Multi-thread – mutex, je měření výchozi implementace kde průběžné výsledky algoritmu ukládám do vektoru, který sdílí všechna vlákna.

Multi-thread – future, je měření implementace, kterou jsem přibalil do .zip archivu. Viz. „Poznámka k paralelní implementaci“.

Single-thread	Multi-thread - mutex	Multi-thread - future
6938 ms	5826 ms – 16 % času	5764 ms – 17 % času
7222 ms	5730 ms – 21 % času	5721 ms – 21 % času
5610 ms	3701 ms – 36 % času	4678 ms – 17 % času
4702 ms	3089 ms – 35 % času	3816 ms – 19 % času
4454 ms	3656 ms – 18 % času	5240 ms + 15 % času
4745 ms	3894 ms – 19 % času	4506 ms – 6 % času
7238 ms	5839 ms – 20 % času	6413 ms – 12 % času
4327 ms	3394 ms – 22 % času	3942 ms – 9 % času
6944 ms	5865 ms – 16 % času	6248 ms – 11 % času
6957 ms	4757 ms – 32 % času	5288 ms – 24 % času

Závěr měření

Multi-thread mutex verze byla na testovaných datech v průměru rychlejší o: (0.16 + 0.21 + 0.36 + 0.35 + 0.18 + 0.19 + 0.20 + 0.22 + 0.16 + 0.32) / 10 = **23,5 %**

Mutli-thread future verze byla na testovaných datech v průměru rychlejší o: (0.17 + 0.21 + 0.17 + 0.19 – 0.15 + 0.06 + 0.12 + 0.09 + 0.11 + 0.24) / 10 = **12.1%**. V jednom z případů trvala ale o 6 % déle než single threaded verze.

Vzhledem k tomu, že jsem měření neprováděl na příliš výkonném počítači, tak se domnívám, že výsledky mohly být negativně ovlivněny velkým počtem spuštěných aplikací na pozadí. Po měření jsem ještě všechno povypínal a poté se single-threaded verze pohybovala asi kolem 1500 ms a obě dvě multi threaded verze kolem 1000 ms. Takže zrychlení bylo přibližne kolem **30 %**, s tím, že future verze byla o pár procent pomalejší.

Příklad použití cmdline argumentů

- --help
- --random:1000 --svg:export_do_svg --output:result (spočítá convex hull nad 1000 náhodnými body, které mají souřadnice, které náleží {0,1,2 … 10000}, výsledek uloží do souboru result.txt a vytvoří .svg visualisaci do souboru export_do_svg.svg)
- --input:small_input_set.txt --svg:export_do_svg (spočítá convex hull nad náhodnými body, výsledek na příkazovou řádku a vytvoří visualisaci)