

Semestrální projekt MI-PDP 2018/2019:

Paralelní algoritmus pro řešení problému pokrývání plochy dlaždicemi L

Ladislav Martínek

magisterske studijum, FIT CVUT, Thakurova 9, 160 00 Praha 6

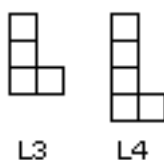
April 20, 2019

1 Definice problému

Problém, který jsme měli v rámci semestrálního projektu řešit se týkal pokrývání plochy dlaždicemi L. Tedy bylo nutné pokrýt obdelníkovou mřížku dlaždicemi tak, že se optimalizovala cena takového řešení daná funkcí.

1.1 Vstupní data

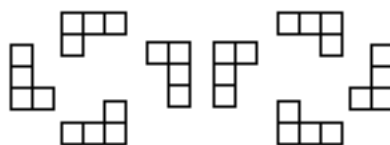
m, n = rozměry obdelníkové mřížky $R[1..m, 1..n]$, skládající se z $m \times n$ políček. Dlaždice dvou tvarů L3 a L4, viz obrázek níže 1.1.



$k < m * n$ = počet zakázaných políček v mřížce R .

$D[1..k]$ = pole souřadnic k zakázaných políček náhodně rozmístěných v mřížce R .

Pro mřížku R se zakázanými políčky danými polem D najděte **pokrytí** dlaždicemi L3 a L4 (viz obrázek 1.1) s **maximální cenou**. Dlaždice lze **pravoúhle otáčet** a **obracet** (viz obrázek 6.1).



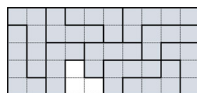
Všechna možná otočení a obrácení dlaždice L3.

Cena pokrytí (funkce, kterou v algoritmu optimalizují), ve kterém zbylo q nepokrytých nezakázaných políček, je

$$2 * \text{počet dlaždic tvaru L3} + 3 * \text{počet dlaždic tvaru L4} - 6 * q.$$

1.2 Výstup algoritmu

Číslo udávající maximální cenu pokrytí a počty dlaždic L3 a L4 a počet nepokrytých políček a jejich souřadnice. Popis pokrytí mřížky R , např. $(m \times n)$ -maticí identifikátorů, kde každá dlaždice je jednoznačně určena políčky s unikátním ID ≥ 1 , nepokrytá políčka jsou prázdná a zakázaná jsou reprezentovaná znakem 'Z'.



Mřížka R o rozměrech 10×5 políček s 3 zakázanými políčky pokrytá dlaždicemi.

Řešení vždy existuje. **Těsná dolní mez** na cenu pokrytí je $-6 * (n * m - k)$, když není vložitelná žádná dlaždice L4/L3. **Těsnou horní mez** na cenu pokrytí lze pro dané m, n, k spočítat je možné vypočítat pomocí řešení diofantské rovnice $4 * a + 5 * b = n * m - k$, které vždy existuje (neboť 4 a 5 jsou nesoudělná čísla) a kde bereme kladné řešení s maximálním b . Tuto mez někdy není možné dosáhnout (závisí i na pozici zakázaných políček).

1.3 Vstupní datové soubory

Formát vstupního textového datového souboru je následující:

1. řádka: celá čísla m a n = rozměry obdélníkové mřížky (počet řádků a sloupců)

Další řádka: číslo k = počet zakázaných políček v mřížce následujících k řádek obsahuje x,y souřadnice zakázaných políček.

2 Sekvenční řešení

Sekvenční algoritmus je typu prohledávání do hloubky s metodou větví a hranic (BB-DFS) s hloubkou prohledávání prostoru omezenou na $(m * n - k)$. Algoritmus končí, když je cena rovna horní mezi nebo když prohledá celý stavový prostor do hloubky $(m * n - k)/4$.

Po načtení dat ze souboru nebo případně z příkazové řádky (obojí lze specifikovat argumenty) jsou vytvořeny dvě instance třídy *sollution* jedna pro aktuální zpracovávanou konfiguraci a druhá pro nejlepší dozaženu. Mřížku reprezentuju jako 2D vektor a zároveň si držím parametry jako počet prázdných polí za a před zpracovávaným políčkem, počet umístěných kostiček L3 a L4 a také velikost dané mřížky. Celý algoritmus pak využívá pouze dvě metody při řešení.

První metodou je *nextFree*, která pro zadané souřadnice nalezne nejbližší volné políčko (první volání je mimo hrací plochu na začátku). Druhá metoda je pro přidání políčka na mapu *addOnMap*. Tato metoda bere souřadnici kam začínáme pokládat (pouze pokud lze jinak neproběhne úspěšně) dále relativní souřadnice ve vektoru dané kostičky a předchozí hodnotu na místě a novou hodnotu z counteru. Touto metodou lze také kostičky odebírat při návratu v DFS.

Z držných parametrů daného rozložení je vždy spočítána cena a pokud se rovná maximální možné tak algoritmus končí. Dále pro optimalizaci používám prořezávání, tedy pokud aktuální instance s teoretickým maximem pro zbývajících pole nemůže překonat aktuální maximum tak větev také končí.

Níže jsou uvedeny časy sekvenčního řešení pro 3 vybrané instance.

Instance	Čas (s)
pol13	1091.261
pol14	640.393
pol15	876.792

První instance je prázdná pole o rozměrech 22x18. Druhá instance je pole 15x15 s zakázanými políčky v rozích přibližně prostřed stěn a uprostřed. Poslední je 17x17 se vzorem 4 čverců rozmístěných přibližně symetricky, tak aby kolem sebe měli co nejvíc místa. Tedy celkem 16 zakázaných políček.

3 Popis paralelního algoritmu a jeho implementace v OpenMP - taskový paralelismus

Při implementaci v OpenMP a taskovém paralelismu jsem upravil sekvenční řešení. Implementaci pomocí taskového paralelismu ani nebudu dále využívat. Aby tento paralelismus šlo využít bylo nutné zařídit, aby v rekurzi byla předávána kopie řešení a ne pouze ukazatel na jednu kopii, to celé řešení zpomalovalo a pomocí tohoto řešení se mi nepodařilo dosáhnout obstojných výsledků.

První použitá direktiva je `omp parallel` blok a v něm první volání rekurze s direktivou `single`, je nutné aby první volání provedlo jen jedno vlákno. Samotná rekurze je stejná jako u sekvenčního řešení s těmi rozdíly, že kontrola a update nejlepšího řešení probíhá v kritické sekci a při volání rekurze a zanoření je použita direktiva `omp task`, která tvoří tasky. Hloubku zanoření kontroluju podle id kostiček položených na mapě. Tuto hloubku lze nastavit parametrem `(-nN)` na příkazové řádce stejně jako počet vláken `(-nT)`. Přepínač `-tp` slouží pro výběr task paralelismu z možností datového `(-dp)` a nebo pouze rekurzivního na jednom vlákne `(-r)`

Pří zkoušení se neosvědčilo mít hloubku zanoření více ja 4 pro 16 jader pro tento problém, který má větvičí faktor 17.

```
./[BUILD BINARY] -f [PATH TO INSTANCE] -tp -nT [NUMBER OF THREADS] -nN [PLUNGE DEPTH]
```

4 Popis paralelního algoritmu a jeho implementace v OpenMP - datový paralelismus

Při implementaci datového paralelismu v OpenMP bylo nutné vytvořit instance problému, které budou distribuované mezi vlákna. Pro toto jsem využil algoritmus BFS, kterým v režimu jednoho vlákna generuje instance "předvyplněné mapy". Počet těchto instancí lze opět určit z příkazové řádky -nN ovšem v tomto případě je to minimální počet instancí (dáno větvicím faktorem maximálně o 16 víc), které se vygenerují.

Pro samotnou implementaci pak využívám direktivu `parrallel for`, který vytvoří daný počet vláken (-nT) a přiděluje jim přegenerované instance. Přidělování používám dynamické. V rekurzi pak pro kontrolu nejlepšího řešení používám direktivu pro kritickou sekci.

Tato implementace paralelního algoritmu se mi zamlouvala líp a byla také mnohem efektivnější. To především z důvodu, že jednotlivé vlákna mohou prováděnou rekurzi jednoduše provádět na sdílené paměti, což výpočet velice zrychluje.

```
./[BUILD BINARY] -f [PATH TO INSTANCE] -tp -nT [NUMBER OF THREADS] -nN [NUMBER OF GENERATED]
```

5 Popis paralelního algoritmu a jeho implementace v MPI

Pro MPI bylo nutné přepracovat více částí. V MPI je nutné kompilovat kód pomocí speciálního kompilátoru a spouštět pomocí `mpirun`, kde jsou vytvořeny jednotlivé procesy. Implementace algoritmu je Master-slave. Tedy jeden proces je jako hlavní a ostatní jako dělníci.

Slave proces má jednoduchou implementaci, hlavní částí je blokující příjem zprávy od mastera. V této zprávě buď přijme informaci o konci nebo práci, kterou má vykonávat. Pokud je konec, tak se ukončí. Pokud se jedná o práci, tak vykonává s instancí přesně to co bylo popsáno v datovém paralelismu v kapitole 4. V kódu je jen jedno vychýlení a to, že po určitém počtu kroků provádí neblokující broadcastování nejlepšího řešení, tento broadcast začíná vždy master pokud dostane nějaké nové lepší řešení. Toto bylo nezbytně nutné pro ořezávání problému, neboť se stávalo, že zůstával běžet jeden proces (s těžkou úlohou, která mohla být oříznuta), i když řešení bylo dávno nalezeno.

Implementace mastera ze začátku byla pouze na posílání a příjem zpráv. Master má frontu na slave procesy, který má poslat zprávu a neblokujícím `send` posílá vygenerované instance, která na začátku vygeneroval pomocí BFS stejně jako v datovém paralelismu a kontroluje zda zpráva opravdu odešla. Poté čeká na příchozí zprávu od slave procesu s řešením. Pokud je řešení lepší uloží ho a neblokujícím broadcastem broadcastuje novou nejlepší hodnotu. A Slave procesu pošle novou práci, pokud nějaká je.

Zprávy v MPI posílám jako pole čísel, kde jsem do objektu řešení vytvořil serializaci právě do pole čísel.

Tuto implementaci mastera jsem později ještě rozšířil pomocí OpenMP, kde master umí využít i zbylá vlákna na procesoru k počítání. Používám k tomu zapnuté `nested parallel`, abych mohl vnořit vytváření vláken pro OpenMP. V masterovi jsou nejprve vytvořena dvě vlákna, kde jedno je klasický master a druhé je jakoby slave. K tomuto řešení jsem pro začátek ještě nastavil zámek, který je zamčený z důvodu, že by se vlákno spustilo dříve, než bude naplněná fronta. Zámek je uvolněn po naplnění fronty.

Toto slave vlákno pak odebírá problémy z fronty a řeší je datovým paralelismem. Při práci s frontou a nejlepším řešením jsou proměnné zamčené v kritické sekci, aby nedocházelo ke konfliktům s master vláknem (procesem). Při této implementaci se vyskytli nějaké problémy, které bylo nutné odladit. Výkonost se ukáže při testování tohoto řešení.

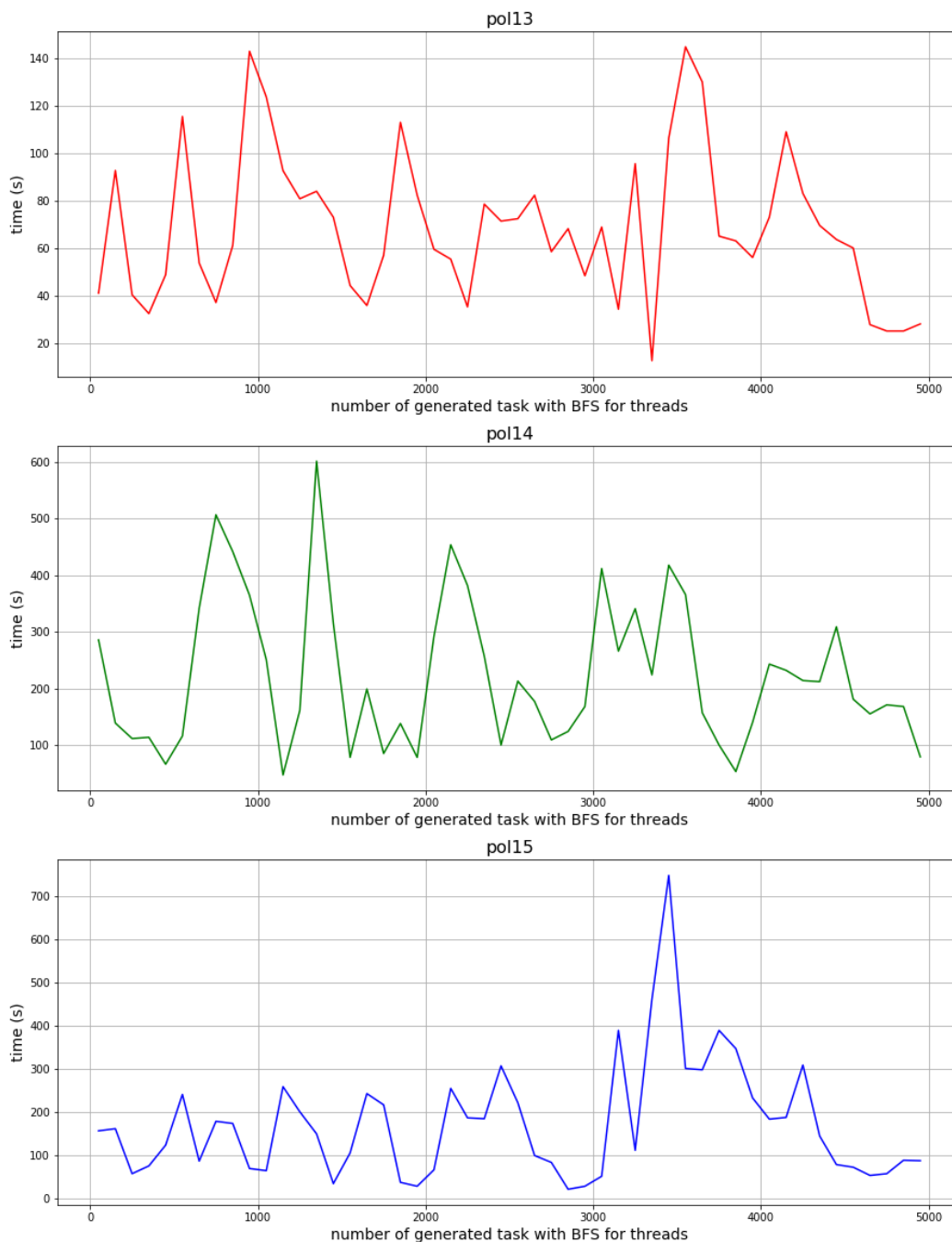
```
mpirun -n [NUMBER OF PROCESSES] [BUILD BINARY] -f [PATH TO INSTANCE] -mpi  
-nT [NUMBER OF THREADS] -nN [NUMBER OF GENERATED IN PROCESS]  
-nNP [NUMBER OF GENERATED IN MASTER]
```

6 Naměřené výsledky a vyhodnocení

Zde uvedu naměřené výsledky ze spuštěných experimentů a výpočetním klastru star. Pokusil jsem se neprve skoumat závislost na velikosti generovaných problémů a to jak na úrovni paralelismu na jednom procesoru, tak na úrovni komunikace mezi výpočetními jednotkami. Dále pak zrychlení s počtem použitých výpočetních jader.

6.1 Závislost výpočetního času na počtu generovaných tasků jeden výpočetní uzel

Jak je vizet na grafech níže je tento problém velice datově závislý a čas velice ovlivní a počet vygenerovaných instancí a to ne v malé míře. Tento test byl prováděn na jednom vypočetním uzlu na 20 jádrech. V tomto případě jsem zde očekával nějakou závislost, ale časy by nejspíš rostly až kdyby generování tasků bralo velkou část času.



6.2 Závislost výpočetního času na počtu generovaných úloh pro MPI

6.3 Závislost výpočetního času na počtu výpočetních jader na jednom výpočetním uzlu

6.4 Závislost výpočetního času na počtu výpočetních jader v klusteru

6.5 Analýza a hodnocení

Analýza a hodnocení vlastností paralelního programu, zvláště jeho efektivnosti a škálovatelnosti, případně popis zjištěného superlineárního zrychlení.

Škálovatelnost algoritmu je možná v mezích, kdy ještě BFS pro generování úloh bude časově přijatelné k času řešení celého problému. Podle naměřených výsledků bych odhadoval, že lze problém dobře škálovat, protože lze velice rychle vygenerovat velký počet úloh na začátku, ale implementace algoritmu v MPI by musela být změněna na nějakou, která bude efektivněji využívat síť. V mé implementaci posílám pokaždé celé zadání.

1. Zvolte tři instance problému s takovou velikostí vstupních dat, pro které má sekvencní algoritmus časovou složitost kolem 5, 10 a 15 minut. Pro měření času potřebný na čtení dat z disku a uložení na disk neuvazujte a zakomentujte ladici tisky, logy, zprávy a výstupy.
2. Měřte paralelní čas při použití $i = 2, \dots, 60$ výpočetních jader na síti Ethernet.
3. Tabulková a případně graficky zpracovávejte naměřené hodnoty časové složitosti měřených instancí běhu programu s popisem instancí dat. Z naměřených dat sestavte grafy zrychlení $S(n, p)$.
- 4.

7 Závěr

Semestrální práce byla velice zajímavá a práce na ní přínosná především z hlediska ošáhání si knihoven pro psaní paralelních programů jinak než přes samotné vytváření vláken pomocí Thread. Tato implementace může být velice efektivní a to především v přehlednosti kódu a časové náročnosti psaní kódu. Dále také vyzkoušení si programování pro výpočetní klastr, kde je nutné aby spolu jednotlivé procesy komunikovali jinak než přes sdílenou paměť. Semestrální práce nám dala možnost si vyzkoušet různé přístupy a možnost s knihovnami experimentovat.