Semestralní projekt MI-PDP 2018/2019:

Paralelni algoritmus pro řešení problému pokrývání plochy dlaždicemi L

Ladislav Martínek

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

29. dubna 2019

1 Definice problému

Problém, který jsme měli v ráncí 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, která je uveden níže.

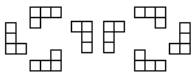
1.1 Vstupní data

m, n = rozměry obdélníkové mřížky $R[1..m, 1..n]$, skládající se z m x 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 1.1).



Všechna možná otočení a obrácení dlaždice L3.

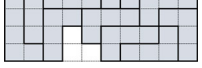
Cena pokrytí (funkce, kterou v algoritmu optimalizuji), ve kterém zbylo q nepokrytých nezakázaných políček, je zadána následovně:

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

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. $(m \cdot n)$ - 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 x 5 políček s 3 zakázanými políčky pokrytá dlaždicemi.

Řešení vždy existuje. **Těsná dolní mez** na cenu pokrytí je $-6 \cdot (n \cdot m - k)$, 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 prohledávaní prostoru omezenou na $(m \cdot n - k)$. Algoritmus končí, když je cena rovna horní mezi nebo když prohledá celý stavový prostor do hloubky $\frac{m \cdot n - k}{4}$.

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 dozaženou. Mřížku reprezentuju jako 2D vektor a zároveň si držím parametry jako počet prazdných polí za a před zpracovávaným poličkem, 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 souřednice nalezne nejbližší volné políčko (první volání je mimo hrací plochu na začátku). Druhá metoda je pro přídaní políčka na mapu $addOnMap$. Tato metoda bere souřadnici kam začínáme pokládat (pouze pokud lze jinak neproběhne úspěsně), dále relativní souřadnice ve vektoru dané kostičky, předchozí hodnotu na místě a novou hodnotu z counteru. Touto metodou lze také kostičky odebírat pří návratu v DFS při zadání původní hodnoty jako id kostičky, která na místě ležela.

Z držených parametrů daneho rozložení je vždy spočítána cena a pokud se rovná maximální možné tak algoritmus končí. Dále pro optimalizi používám prořezávání, tedy pokud aktuální instance s teoretickým maximem pro zbývající volná 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 prázná pole o rozměrech 22x18. Druhá instance je pole 15x15 s zakazanými políčky v rozích přibližně prostřed stěn a uprostřed. Poslední je 17x17 se vzorem 4 čverců rozmístěných přibližně symentricky, 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 musel upravil sekvenční řešení. Implementaci pomocí taskového paralelismu ani nebudu dále využívat pro implementaci v MPI. 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 nepodařílo 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 volaní rekurze a zanoření je použita direktiva omp task, která tvoří tasky. Hloubku zanoření kontroluju podle id kostiček položených na mapě. Tuto hlobku 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 vlákně (-r). Při zkoušení se neosvědčilo mít hloubku zanoření více ja 4 pro 16 jader.

```
./[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 vlakna generuje instance (předvyplněné mapy). Počet těchto instancí lze opět určit z příkazové řádky (-nN), ovšem v tomto případě je to minimální počet instancí (dáno větvícím faktorem maximálně o 16 víc), které se vygenerují.

Pro samotnou implementaci pak využívám direktivu parralel 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 algorimu se mi zamlouvala líp a byla také mnohem efektivnější. To především z důvodu, že jednotlivé vlákna mohou prováděnou rekurzi jednoduše provádět na jedné instanci, což výpočet velice zrychluje.

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

5 Popis paralelniho algoritmu a jeho implementace v MPI

Pro MPI bylo nutné přepracovat více části. V MPI je nutné kompilovat kód pomocí speciálního kompilátoru (mpic++) a spouštět pomocí mpirun, kde jsou vytvořeny jednotlivé procesy a je zajištěna komunikace mezi nimi. Implemetace algoritmu je Master-slave. Tedy jeden proces je jako hlavní a ostatní jako dělníci. (v mé pozdější implementaci jsou volná vlákna na masteru také pracující)

Slave proces má jednoduchou implementaci, hlavní částí je blokující příjem zprávy od mastera. V této zprávě bud 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 určitěm počtu kroků provádí neblokující broadcastování nejlepšího řešení, tento broadcast začíná vždy master pokud dostane nějaké nové lepší řešení. Toto bylo nezbytně nutné pro ořezavání 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 implemetaci 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 proměné zamčené v kritické sekci, aby nedocházelo ke konfliktům s master vláknem (procesem). Pří této implementaci se vyskytli nějaké problémy především s kritickými sekcemi, 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]
```

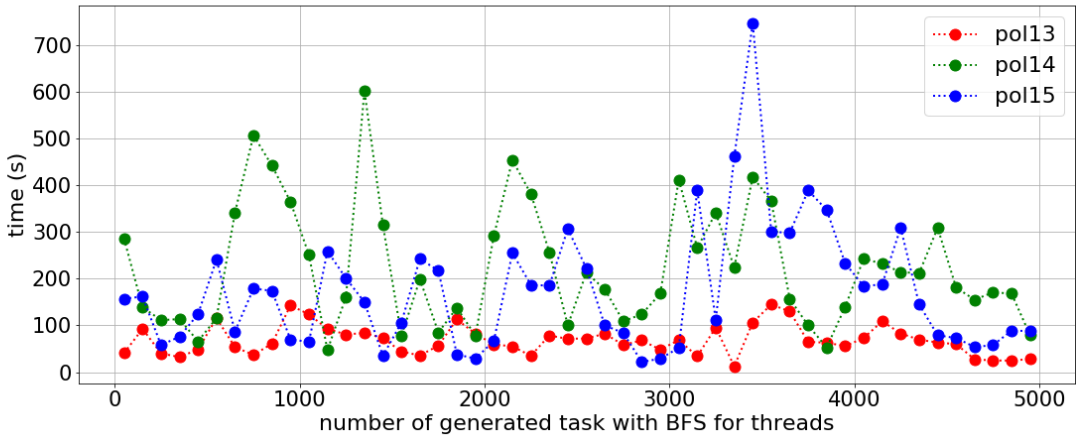
6 Naměřené výsledky a vyhodnocení

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

6.1 Závislost výpočetního času na počtu generovaných tasků jeden výpočetní uzel

Jak je vidět na grafech níže je tento problém velice datově závislý. Na grafu 1 je závislost počtu generovaných instancí na jednom CPU na výpočetní čas. V grafu není vidět žádná závislost a vypočetní čas se pohybyje ve velkém rozsahu. 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 generovaní tasků bralo velkou část času, což se děje spíše až uz sta tisíců. V tomto případě tak počáteční vygenerování rozložení určovalo jak rychle dojde k prořezání.

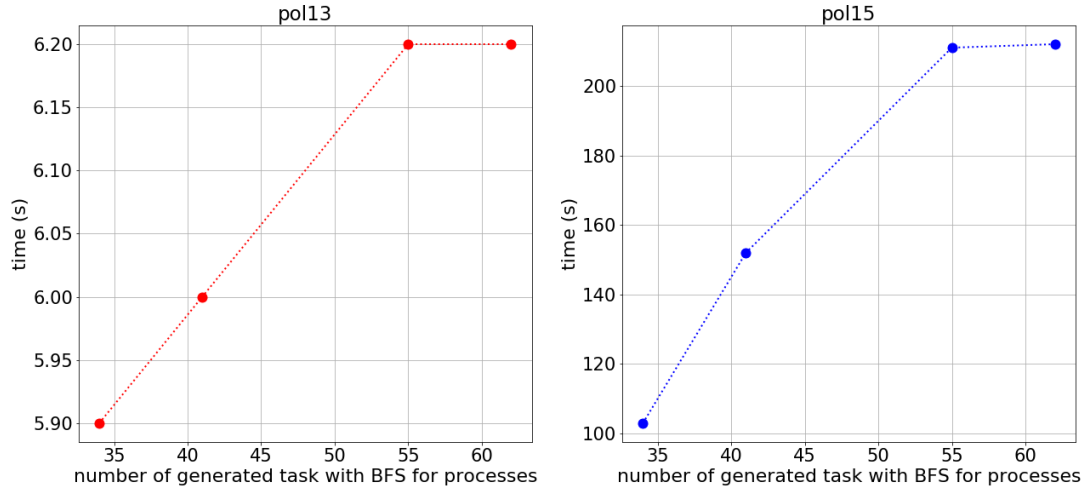
Pro další testy se u jednotlivých instancí budu snažit "netrefit"peaky, ale tak aby algoritmus běžel co nejrychleji.



Obrázek 1: Závislost počtu vygenerovaných počátečních konfigurací pomocí BFS pro datový paralelismus na jednom procesoru

6.2 Závislost výpočetního času na počtu generovaných tásků pro MPI

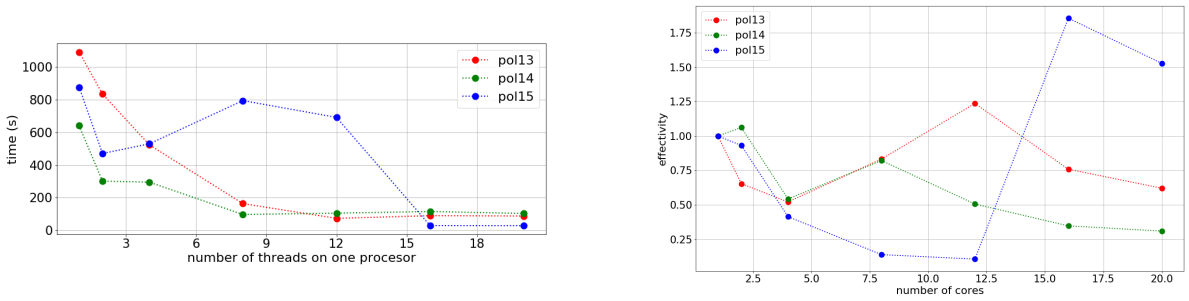
Na grafu 2 je již závislost patrnější a generování rozložení pro jednotlivé procesy je dobré co nejvíc minimalizovat. Tedy snížit počet komunikací po síti a nechat počítat na jednotlivých procesorech. Test byl prváděn na procesorech o 20 jádrech. Výsledky by možná byly odlišné při velkém počtu uzlů s méně jádry. Pro další testy držím tedy počet generovaných tasků na nízkém čísle.



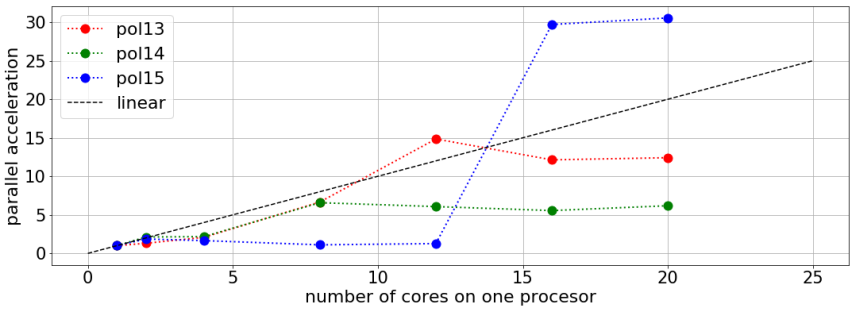
Obrázek 2: Závislost počtu vygenerovaných počátečních konfigurací pomocí BFS pro datový paralelismus na jednom procesoru

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

Dálé jsem měřil algoritmus na jednom procesoru a zkoušel zvyšovat počet výpočetních jader. Na grafu 3 je vidět klesající výpočetní čas s počtem jader, ale opět je to velice datově závislé. Také je na grafu paralelní zrychlení vidět superlineární zrychlení, které nastává právě díky prořezávání, které proběhne optimálněji a výpočet se ukončí rychle.



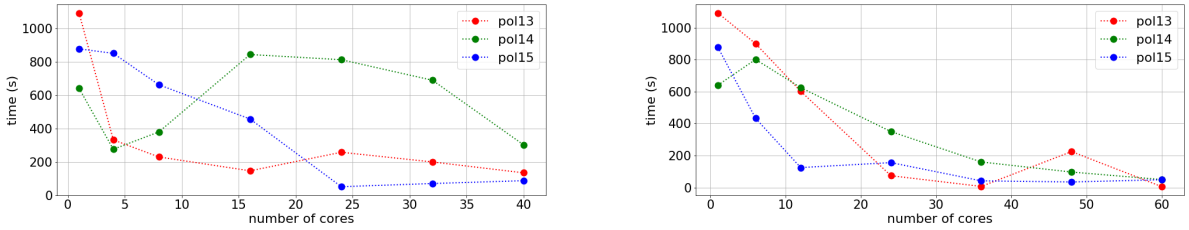
Obrázek 3: Na levém je obrázek výpočetního času na počtu jader. Na právém je efektivita.



Obrázek 4: Na obrázku je graf paralelního zrychlení

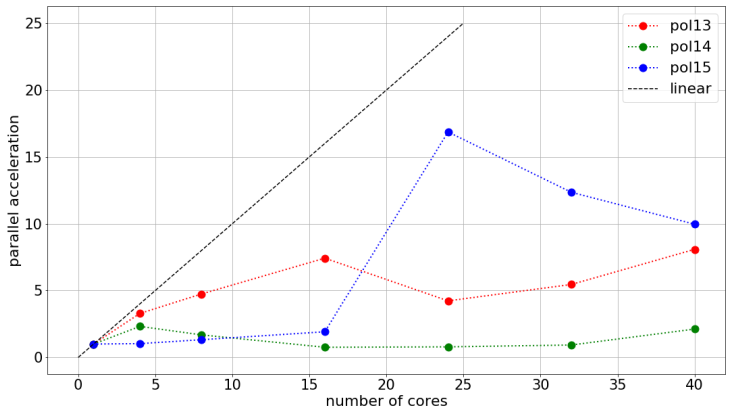
6.4 Závislost výpočetního času na počtu vypočetních jader v klusteru

V klusteru jsem díky pracujícímu masterovi testoval konfiguraci i pro 2 počítače v klastru a i pro 3. Je jasně patrné, že při 3 vypočetních uzlech se podařilo dosáhnout lepších časů i díky tomu, že v klusteru bylo k dispozici o 20 více jader.



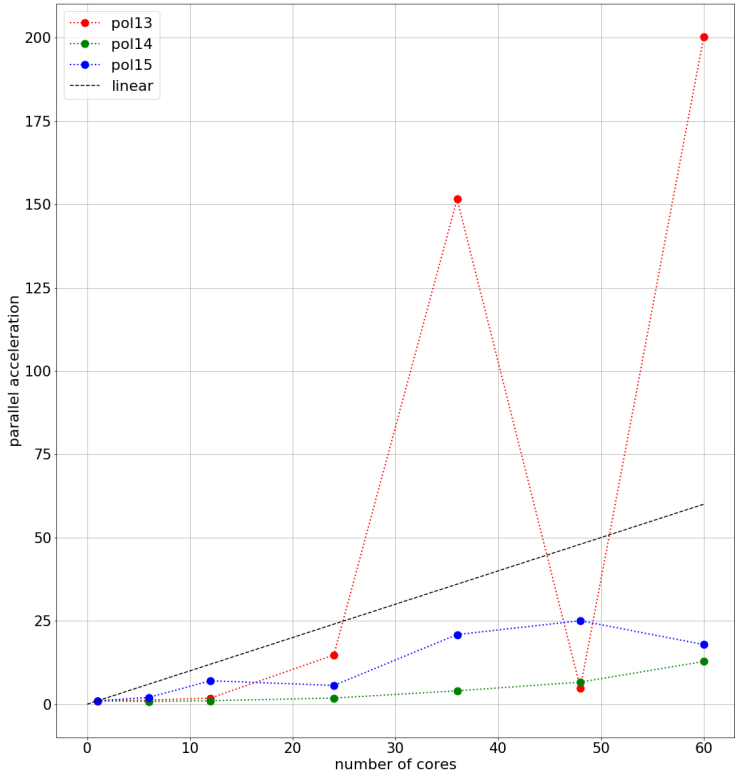
Obrázek 5: Na levém je obrázek výpočetního času na počtu jader pro dva počítače v klusteru. Na právém je stejný graf pro 3 počítače v klusteru.

Dále jsem z naměřených dat udělal grafy pro paralelní zrychlení a efektivitu, kterou jsem udělal pouze pro 3 vypočetní uzly. Na grafu níže 6 je graf paralelního zrychlení pro dva uzly. Při porovnání s grafem 4 je tento graf o poznání horší. Toto zpomalení je způsobeno komunikací mezi uzly a nemožností práce nad jednou sdílenou pamětí. Později bude pak dobré porovnat tento graf pro tři výpočetní uzly, který je uveden níže, kde očekávám lepší výsledek.



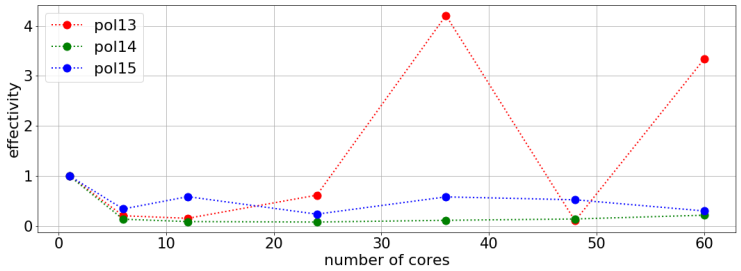
Obrázek 6: Na obrázku je graf paralelního zrychlení

Na grafu 7 je vidět paralelní zrychlení pro klastr se třemi výpočetnímy uzly a maximálním počtem 60 jader. Je vidět, že zrychlení pro instance 14 a 15 není takové, protože zde kvůli zakázaným políčkům není tak jednoduché najít optimální řešení nebo prořezávat, naopak pro prázdnou instanci 13 se prořezaní někdy podařilo což vedlo k super lineárnímu zrychlení.



Obrázek 7: Na obrázku je graf paralelního zrychlení

Na posledním obrázku 8 je vidět efektivita v závislsti na výpočetních jádrech. Graf koresponduje s paralelním zrychlení a jsou zde vidět dva peaky efektivity výšší než 1 přeně, kde došlo k superlineánímu paralelnímu zrychlení.



Obrázek 8: Na obrázku je graf paralelního zrychlení

6.5 Analýza a hodnocení

Paralelní algoritmus pro jeden procesor s datovým paralelismem funguje velice dobře. Z výsledků MPI programu je vidět, že by implementace mohla být určitě vylepšena ve více ohledech. První úzké místo je broadcastování zlepšujícího řešení, kde kontrola i četnost tohoto volání jde řídit. Dále zlepšit posílání jednotlivých úloh, kde by se nemusel posílat celý grid a neprovádět pokáždé serializaci, ale zefektivnit komunikaci. Dále také implementace na vláknech mastera, kterou jsem sice provedl, ale jednotlivá vlákna trávila příliž času v kritických sekcí, což bych v dalších pokusech omezil.

Škálovatelnost algoritmu je možná v mezích, kdy ještě BFS pro generování tasků 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 tasků na začátku a z měření se ukázalo, že pro počet uzlů není tak důležité vytvořit více tásků. Dále škálovatelnost také může výrazně ovlivnit komunikace přes master uzel, kdy by již nestačil odbavovat komunikaci. Toto by šlo vyřešit celkovou decentralizací algoritmu.

Celý problém byl s velkým větvením stromu a také velice datové závislý, kde rychlost velmi ovlivnilo jen počáteční generovaní rozložení. Dobrá trefa v tomto generování mohla umožnit témeř okamžité prořezání a superlineární zrychlení, protože se již zbytek stromu nemusel procházet, zatímco při řešení na jednom vlákně bylo řešení v procházení stromu deterministický a prořezání proběhlo vždy stejně.

7 Závěr

Semestrální práce byla velice zajímavá a práce na ní přínosná především z hlediska ošahání 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 programu. Dále také vyzkoušení si programovaní pro vypočetní 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.