Semestrální projekt MI-PDP 2019/2020:

Paralelní algoritmus pro řešení úlohy minimálního hranového řezu

Bc. Tomáš Chvosta

magisterské studium, FIT ČVUT, Thákurova 9, 160 00 Praha 6

Duben, 2020

1 Definice problému a popis sekvenčního algoritmu

V této kapitole je definován a analyzován zadaný problém a také je zde popsána a vysvětlena samotná implementace sekvenčního řešení.

1.1 Definice problému

Úkolem bylo vytvořit program, který je schopen nalézt minimální hranový řez zadaného grafu rozděleného na části specifikovaných velikostí. Ve své podstatě představuje minimální hranový řez rozdělení množiny uzlů zadaného grafu na dvě disjunktní podmnožiny X, Y takové, že součet ohodnocení všech hran vedoucích z X do Y (tedy velikost hranového řezu mezi X a Y) je nejmenší možný. Navíc je podmínkou, že množina X musí obsahovat uživatelem specifikovaný počet uzlů.

Vstupní data obsahují:

- n - přirozené číslo představující počet uzlů grafu G, pro které platí, že $20 \leq n \leq 160$
- k - přirozené číslo řádu jednotek představující průměrný stupeň uzlu grafu G, pro které platí, že $3 \leq k \leq n$
- $G(V, E)$ - jednoduchý souvislý neorientovaný hranově ohodnocený graf o n uzlech a průměrném stupni k, kde ohodnocení hran grafu jsou desetinná čísla z intervalu 0 až 1
- a - přirozené číslo představující počet vrcholů v množině X, pro které platí, že $n/2 \leq a \leq n/4$

Společně se zadáním této úlohy byly poskytnuty předgenerovaná testovací data obsahující různě velké instance řešeného problému. Testovací data jsou v následujícím formátu:

- první řádek obsahuje trojici n k a
- následuje $n * k/2$ trojic u_1 u_2 v, kde každá uvedená trojice reprezentuje neorientovanou hranu mezi dvěma uzly u_1 a u_2 s ohodnocením v

Výstupem programu je výpis podmnožin uzlů X a Y společně se součtem ohodnocení hran, které tyto množiny spojují. Dále je výstupem doba trvání nalezení minimálního hranového řezu.

1.2 Implementace sekvenčního řešení

Tato sekce obsahuje popis implementovaného řešení. Program nejprve načte vstupní data, která jsou popsána v předchozí sekci. Zadaný graf je uložen jako globální proměnná typu `vector`, která pro každý vrchol uchovává

množinu hran, které z něj vedou do vrcholů s nižším indexem. Po načtení vstupních dat je vytvořen `vector` o velikosti počtu vrcholů grafu, který bude pro každý vrchol ukládat informaci o tom, zda patří do množiny X nebo Y (reprezentováno pomocí hodnot $true$ a $false$). Před zahájením samotného výpočtu jsou vytvořeny globální proměnné pro cenu minimáního hranového řezu a pro samotný minimální řez.

Následně dochází k zahájení samotného výpočtu. Ten je realizován rekurzivní procedurou solve, která prochází jednotlivé stavy stavového prostoru. Stavy představují jednotlivé rozdělení vrcholů grafu do dvou množin X a Y a přechody představují přidání aktuálního vrcholu do jedné z množin. Jako počáteční je zvolen takový stav, ve kterém nebylo rozhodnuto pro žádný z vrcholů grafu, zda bude přidán do množiny X nebo Y.

Algoritmus pro prohledávání stavového prostoru je typu DFS metodou větví a hranic. Cílem prohledávání stavového prostoru je nalezení přípustného koncového stavu (stav, který vyhovuje podmínkám řešení) s nejmenší cenou. Počáteční odhad ceny je nastaven na součet ohodnocení všech hran v grafu, kdy je jasné, že žádný hranový řez nemůže mít vyšší cenu, než je tento počáteční odhad. Každé volání rekurze se rozdělí na dvě nové větve, kdy jedna větev se pokusí přidat aktuální vrchol do množiny X a druhá do množiny Y. Pro každé nově vzniklé rozdělení se zavolá tato rekurze pro následující vrchol. Rekurzivní procedura se snaží dostat do všech přípustných koncových stavů a v těchto stavech vždy zkontroluje, zda není cena nalezeného hranového řezu menší, než aktuální minimální cena uložená v globální proměnné. Pokud je nalezená cena menší, uloží ji spolu s řezem do globálních proměnných.

Během prohledávání stavového prostoru však dochází k jeho ořezávání a tak všechny koncové stavy nemusejí být dostupné. Je však vyloučeno, aby se v těchto stavech nacházelo lepší řešení, než je doposud nalezené řešení. K ořezání prostoru (ukončení výpočtu v aktuální větvi rekurze) dochází v případě, že je splněna jedna z následujících podmínek:

1. Po umístění všech zbylých vrcholů do množiny X bude velikost množiny X menší než cílový počet a. (Je jasné, že v této části stavového prostoru se nenachází žádný přípustný koncový stav.)
2. Aktuální velikost množiny X je větší než cílový počet a. (Opět je jasné, že v této části stavového prostoru se nenachází žádný přípustný koncový stav.)
3. Aktuální cena řezu je větší než doposud nalezená minimální cena. (V této části stavového prostoru se sice mohou vyskytovat přípustné koncové stavy, ale je jasné, že tyto stavy nemohou být řešením problému.)

Výpočet končí ve chvíli, kdy jsou prohledány všechny dostupné stavy stavového prostoru.

1.3 Tabulka naměřených hodnot

V této sekci je zobrazena tabulka s naměřenými časy u jednotivých zadaných instancí problému. Veškeré měření proběhlo na počítači s operačním systémem Linux Mint 18.3 se čtyřjádrovým procesorem Intel® CoreTM i7-6500U CPU o frekvenci 2,5 GHz, s RAM pamětí 8 GB.

Soubor	n	k	a	Čas
mhr_20_10_5.txt	20	10	5	0,00150s
mhr_30_10_10.txt	30	10	10	0,37193s
mhr_30_10_15.txt	30	10	15	1,25797s
mhr_34_10_15.txt	34	10	15	3,30883s
mhr_34_10_17.txt	34	10	17	5,51642s
mhr_37_15_17.txt	37	15	17	155,732s
mhr_37_15_18.txt	37	15	18	327,254s
mhr_38_15_18.txt	38	15	18	358,299s
mhr_38_20_15.txt	38	20	15	274,138s
mhr_39_20_20.txt	39	20	20	1344,71s

1.4 Spuštění programu

Kód programu byl napsán v jazyce C++ a je možné ho zkompilovat pomocí následujícího příkazu:

g++ -Wall -pedantic -Ofast -std=c++11 -o mhr mhr.cpp

Zkompilovaný program poté lze spustit následovně:

./mhr `m` `c` `t` < file

Parametr `m` je referenční cena minimálního hranového řezu, `c` je složitost referenčního řešení, `t` je referenční čas, `file` je název souboru se vstupními daty.

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

Tato kapitola popisuje první úpravu sekvenčního algoritmu, který je popsán v předchozí kapitole. Cílem této úpravy bylo paralelizovat výpočet pomocí „task paralelismu“ v OpenMP.

2.1 Popis paralelního algoritmu

Paralelní algorimus s použitím „task paralelismu“ v knihovně OpenMP ve své podstatě pouze lehce upravuje sekvenční řešení pro vícejádrový systém se sdílenou pamětí. Pomocí této úpravy lze jednotlivé větve rekurzivní procedury spouštět paralelně pomocí tasků. Je však potřeba optimalizovat algoritmus tak, abychom snížili režii paralelního výpočtu a nespouštěli zbytečně velké množství tasků. Velké množství úkolů by mohlo způsobit příliš velkou granularitu a režii týkající se neustáleho spouštění nových vláken a přiřazovaní práce těmto vláknům. Cílem tohoto algoritmu tedy není pouze paralelizovat sekvenční řešení, ale také snížit režii paralelního výpočtu.

2.2 Implementace paralelního algoritmu

Jak již bylo zmíněno, implementace „task paralelismu“ je z velké části založena na sekvenčním algoritmu z předchozí kapitoly a jedná se pouze o drobnou úpravu.

První úprava se týkala částí, které jsou zpracovány paralelně. Jednotlivé části je potřeba přesně vymezit a s knihovnou OpenMP vyznačit paralelní úkoly direktivou „#pragma omp task“. V případě našeho programu představují úkoly samotné rekuzivní volání, tedy přechod do dalšího stavu stavového prostoru (přesun k dalšímu vrcholu grafu). Je však potřeba zajistit, aby program zbytečně nevytvářel velký počet tasků a došlo ke snížení režie paralelního výpočtu. Proto je do programu přidána konstanta $TRESHOLD$ a podmínka, která dovolí vytvořit task pouze v případě, že rekurzivní proceduře zbývá rozdělit do množin X, Y alespoň tolik vrcholů, kolik stanovuje konstanta $TRESHOLD$. Přímo v této implementaci se task vytvoří pouze v případě, pokud zbývá rozdělit 25 vrcholů a více. Pokud zbývá méně vrcholů, dokončí program výpočet sekvenčně. Hodnota konstanty $TRESHOLD$ byla určena experimentálně. Pro hodnotu $TRESHOLD = 25$ vrací program nejlepší výsledky.

Druhou úpravou bylo potřeba vyřešit kritickou sekci. Jelikož aktuální nejlepší výsledek je uložen v globální proměnné, ke které může v jeden okamžik přistupovat více vláken najednou, je potřeba zajistit, aby do této proměnné nezapisovala dvě vlákna najednou. Je tedy potřeba přidat kritickou sekci. V knihovně OpenMP lze kritickou sekci vyřešit snadno pomocí direktivy „# pragma omp critical“. V bloku této direktivy se tedy nachází kontrola dosud nejlepšího výsledku a jeho aktualizace.

Další úprava se týkala předání parametrů pro volání rekurzivní procedury. Jedním z parametrů procedury je totiž reference na proměnnou typu `vector`, ve kterém je aktuální stav rozdělení vrcholů grafu do množin X a Y. Jelikož nelze garantovat, v jakém pořadí budou provedeny jednotlivé části rekurzivní procedury, je nutné, aby pro každý task byla zkopírována instance této proměnné. V případě, že se nevytváří nový task, není kopírování `vectoru` nutné.

Jelikož jsou jednotlivé tasky na sobě nezávislé a nemusejí na sebe čekat, není potřeba využívat direktivu „# pragma omp taskwait“, která k tomuto účelu slouží. Co je však potřeba přidat do programu, je direktiva „#pragma omp parallel“, která spustí paralelní výpočet. Ta je v této implementaci přidána k prvnímu volání rekurzivní procedury. Musíme zajistit, aby výpočet byl proveden paralelně, ale zároveň první volání této procedury by mělo být zahájeno pouze jedním vláknem. K tomu slouží direktiva „#pragma omp single“.

2.3 Parametry běhu algoritmu

Běh implementovaného paralelního algoritmu lze ovlivnit dvěma parametry. Jeden z nich je zmíněn v předchozí sekci a tím je konstanta $TRESHOLD$, která určuje, kdy jsou vytvořeny nové tasky a kdy je výpočet dokončen sekvenčně. Druhým parametrem je počet vláken, které může program využít. Defaultně je hodnota tohoto parametru nastavena na 8.

2.4 Spuštění program

Kód programu byl napsán v jazyce C++ a je možné ho zkompilovat pomocí následujícího příkazu:

```
g++ -Wall -pedantic -fopenmp -Ofast -std=c++11 -o mhr mhr.cpp
```

Bez přepínače „-fopenmp“ je možné program přeložit a spustit sekvenčně. Zkompilovaný program poté lze spustit následovně:

```
./mhr m c t < file
```

Parametr `m` je referenční cena minimálního hranového řezu, `c` je složitost referenčního řešení, `t` je referenční čas, `file` je název souboru se vstupními daty.

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

Tato kapitola popisuje druhou úpravu sekvenčního algoritmu, který je popsán v kapitole 1. Cílem této úpravy bylo paralelizovat výpočet pomocí „datového paralelismu“ v OpenMP.

3.1 Popis paralelního algoritmu

U paralelního algoritmu s použitím „datového paralelismu“ v knihovně OpenMP je potřeba program změnit trochu více než v případě „task paralelismu“. Ve své podstatě je potřeba vygenerovat určitý počet stavů pomocí procházení stavového prostoru algoritmem BFS (tedy procházením do šířky). Tyto stavy jsou uloženy do fronty, ze které jsou paralelně odebírány a je z nich spouštěn výpočet.

3.2 Implementace paralelního algoritmu

Jako u předchozího paralelního algoritmu vychází implementace „datového paralelismu“ ze sekvenčního řešení z kapitoly 1. V programu však bylo potřeba udělat několik změn.

Jak už bylo řečeno, tento algoritmus si nejprve vygeneruje určitý počet stavů a z těchto stavů se poté paralelně spustí výpočet. Je tedy potřeba vymyslet způsob, jak ukládat stavy. Konkrétně v této implementaci byla vytvořena třída `BFSState`, která si drží všechny parametry pro zavolání rekurzivní procedury a zároveň umožňuje vygenerovat další přípustné stavy pomocí procházení stavového prostoru do šířky.

Následně bylo potřeba změnit celý proces výpočtu. Program si nyní nejprve vytvoří frontu (pomocí kolekce `std::deque` z STL) a do ní vloží počáteční stav, který je stejný jako v případě sekvenčního algoritmu, akorát je uložen pomocí třídy `BFSState`. Následně je pomocí `while` cyklu a BFS procházení stavového prostoru vygenerován určitý počet stavů, které jsou vloženy do fronty. Ve chvíli, kdy se ve frontě nachází cílový počet stavů, je nad kolekcí spuštěn paralelní výpočet. K tomu je použita direktiva „#pragma omp parallel for“, která zajistí, že je `for` cyklus zpracován paralelně. V této direktivě je použito „schedule (dynamic)“. Díky této možnosti jsou stavy dynamicky přidělovány jednotlivým vláknům. Pro každý stav je následně zavolána rekurzivní procedura popsaná v kapitole týkající se sekvenčního řešení.

Poslední věc, kterou je potřeba přidat, je kritická sekce. V případě nalezení nového a lepšího řešení je kritická sekce ohlídána stejným způsobem jako v případě „task paralelismu“.

3.3 Parametry běhu algoritmu

Běh implementovaného paralelního algoritmu lze ovlivnit dvěma parametry. Jeden z těchto parametrů je konstanta $ENOUGH\_STATES$, která určuje počet stavů, které jsou vygenerovány do fronty před spuštěním paralelního výpočtu. Tento parametr byl zvolen experimentálně a je nastaven na hodnotu 800. Druhým parametrem je počet vláken, které může program využít. Defaultně je hodnota tohoto parametru nastavena na $MAX\_THREADS = 8$.

3.4 Spuštění programu

Kód programu byl napsán v jazyce C++ a je možné ho zkompilovat pomocí následujícího příkazu:

```
g++ -Wall -pedantic -fopenmp -Ofast -std=c++11 -o mhr mhr.cpp
```

Bez přepínače „-fopenmp“ je možné program přeložit a spustit sekvenčně. Zkompilovaný program poté lze spustit následovně:

./mhr `m` `c` `t` < file

Parametr `m` je referenční cena minimálního hranového řezu, `c` je složitost referenčního řešení, `t` je referenční čas, `file` je název souboru se vstupními daty.

4 Popis paralelního algoritmu a jeho implementace v MPI

Doposud jsme se zabývali paralelními algoritmy pro sdílenou paměť. V této kapitole je popsána poslední úprava algoritmu, která paralelizuje program pro distribuovanou paměť za pomoci knihovny MPI.

4.1 Popis paralelního algoritmu

Jak už bylo řečeno, tento typ algoritmu je určen pro systémy s distribuovanou pamětí. Je to tedy přesný opak předchozích implementací, které byly určeny pro systémy se sdílenou pamětí. Tato implementace předpokládá, že je vytvořeno více procesů a jednotlivé instance problémů (stavů) jsou posílány mezi všemi procesy. Samotnou synchronizaci procesů a komunikaci mezi nimi zajišťuje MPI knihovna.

Aby byl algoritmus paralelní, je potřeba vytvořit alespoň 2 procesy tak, aby si tyto procesy byly schopné vyměňovat navzájem informace, které potřebují k výpočtu. Procesy se dělí na typy Master a Slave. Master je hlavní proces, který vytváří jednotlivé instance problému a také pak zasílá Slave procesům, které je následně řeší a vracejí zpět výsledky. Master proces přijímá a vyhodnocuje všechny obdržené výsledky a tím sestaví řešení celého problému.

4.2 Implementace paralelního algoritmu

Na začátku programu je potřeba zavolat příkaz „MPI_Init“, který inicializuje MPI. Jelikož program využívá distribuovanou paměť a ne sdílenou, musí si každý proces načíst potřebná vstupní data (graf) zvlášť. Proto je potřeba načíst vstupní data ze souboru a nikoliv ze standardního vstupu, jelikož standardní vstup může číst pouze jeden z procesů.

Po načtení vstupních dat je třeba určit který proces bude Master a které procesy budou Slave. Konkrétně v této implementaci je využit příkaz „MPI_Comm_rank“, pomocí kterého získáme id procesu. Pokud má proces id 0, potom se jedná o Master proces. Všechny ostatní procesy jsou typu Slave.

Jelikož jednotlivé procesy mohou v rámci komunikace posílat základní datové typy je potřeba pomocí „MPI_Type_create_struct“ vytvořit strukturu v MPI pro jednotlivé stavy, které bude Master posílat Slave procesům k vyřešení, ale také musíme vytvořit strukturu pro výsledek, který následně Slave procesy pošlou zpět Master procesu.

Nyní už nic nebrání tomu, aby započal celý výpočet. Master proces nejprve vygeneruje určitý počet stavů pomocí BFS algoritmu (způsob je stejný jako v předchozí kapitole). Poté se každému Slave procesu pošle úkol z fronty, kterou disponuje Master. Po celou dobu má Master přehled o počtu aktivních Slave procesů a čeká na jejich odpovědi. Pokud obdrží odpověď, zkontroluje výsledek, a pokud je lepší než aktuální nejlepší výsledek, potom tento výsledek aktualizuje. Následně odešle další práci nebo v případě, že je fronta prázdná, pošle zprávu o ukončení. Výpočet skončí ve chvíli, kdy všechny Slave procesy ukončí svou činnost. Na konci Master vypíše výsledek a ukončí běh celého programu.

Slave proces řeší jednotlivé instance problému (stavy) pomocí paralelního algoritmu, který je popsát v předchozí kapitole. Používá tedy datový paralelismus s knihovnou OpenMP. Poté, co Slave proces přijme práci od Master procesu, provede výpočet a zašle výsledek zpět Master procesu a následně čeká na další práci případně na ukončení celého procesu.

4.3 Parametry běhu algoritmu

Běh implementovaného paralelního algoritmu lze ovlivnit třemi parametry. Jeden z těchto parametrů je konstanta $ENOUGH\_STATES\_MASTER$, která určuje počet stavů, které si musí Master proces vygenerovat do své fronty. Tento parametr byl zvolen experimentálně a je nastaven na hodnotu 80. Druhým parametrem je konstanta $ENOUGH\_STATES\_SLAVE$, která určuje počet stavů, které si musí Slave proces vygenerovat před spuštěním paralelního výpočtu pomocí datového paralelismu s knihovnou OpenMP. Tento parametr byl nastaven na hodnotu 800. Posledním parametrem je počet vláken, které může každý proces využít k paralelnímu výpočtu. Defaultně je hodnota tohoto parametru nastavena na $MAX\_THREADS = 8$.

4.4 Spuštění programu

Kód programu byl napsán v jazyce C++ a je možné ho zkompilovat pomocí následujícího příkazu:

mpic++ -Wall -pedantic -fopenmp -Ofast -std=c++11 -o mhr mhr.cpp

Zkompilovaný program poté lze spustit následovně:

mpiexec -n `p` ./mhr `m` `c` `t` < file

Parametr `p` udává počet spuštěných procesů, `m` je referenční cena minimálního hranového řezu, `c` je složitost referenčního řešení, `t` je referenční čas, `file` je název souboru se vstupními daty.

5 Naměřené výsledky a vyhodnocení

V této kapitole jsou výsledky měření a testování jednotlivých algoritmů. Všechna měření probíhala na fakultním klastru star.

5.1 Specifikace měření

Všechny zdrojové kódy byly napsány v jazyce C++ ve standardu C++11. Pro kompilaci byly použity kompilátory g++ a mpic++ verze 4.8.5-36 a byly využity maximální optimalizace kódu (přepínač `-Ofast`).

K testování byly vybrány 3 instance problému, všechny s časovou náročností od 2 do 6 minut (na klastru star). Jedná se o soubory s názvy „mhr_37_15_17.txt“, „mhr_38_15_18.txt“, „mhr_38_20_15.txt“. V následující tabulce jsou uvedeny základní informace o jednotlivých instancích a jsou také uvedeny časy pro referenční sekvenční implementaci (údaje ze zadání úlohy):

Soubor	n	k	a	**Referenční čas**
mhr_37_15_17.txt	37	15	17	178s
mhr_38_15_18.txt	38	15	18	410s
mhr_38_20_15.txt	38	20	15	276s

Pro všechna paralelní řešení bylo vypočteno zrychlení a také efektivita. Zrychlení je definováno pomocí následujícího vztahu:

$$S(n,p) = T_s(n)/T_P(n,p)$$

$T_S(n)$ je čas sekvenčního výpočtu a $T_P(n,p)$ je čas paralelního výpočtu při použití p vláken při instanci n. Dále efektivita je definována následovně:

$$E(n,p) = S(n,p)/p$$

Mělo by platit, že $E(n,p) \leq 1$. Pokud je efektivita větší než 1, pak se jedná o superlineární urychlení.

MPI řešení bylo spuštěno se třemi procesy, tedy s jedním Master a dvěma Slave procesy, jelikož fakultní klastr star více výpočetních prostředků neposkytuje.

5.2 Sekvenční řešení

Následující tabulka obsahuje naměřené časy pro sekvenční řešení:

Soubor	Naměřený čas
mhr_37_15_17.txt	148,91s
mhr_38_15_18.txt	334,858s
mhr_38_20_15.txt	228,099s

5.3 OpenMP řešení - task paralelismus

Následující tabulka obsahuje naměřené hodnoty pro soubor „mhr_37_15_17.txt“:

Počet vláken	Naměřený čas	Zrychlení	Efektivita
1	160,427s	0,928	0,928
2	77,7022s	1,916	0,958
4	37,2259s	4,000	1,000
8	25,783s	5,776	0,722
16	12,4603s	11,951	0,747
32	11,0938s	13,423	0,419

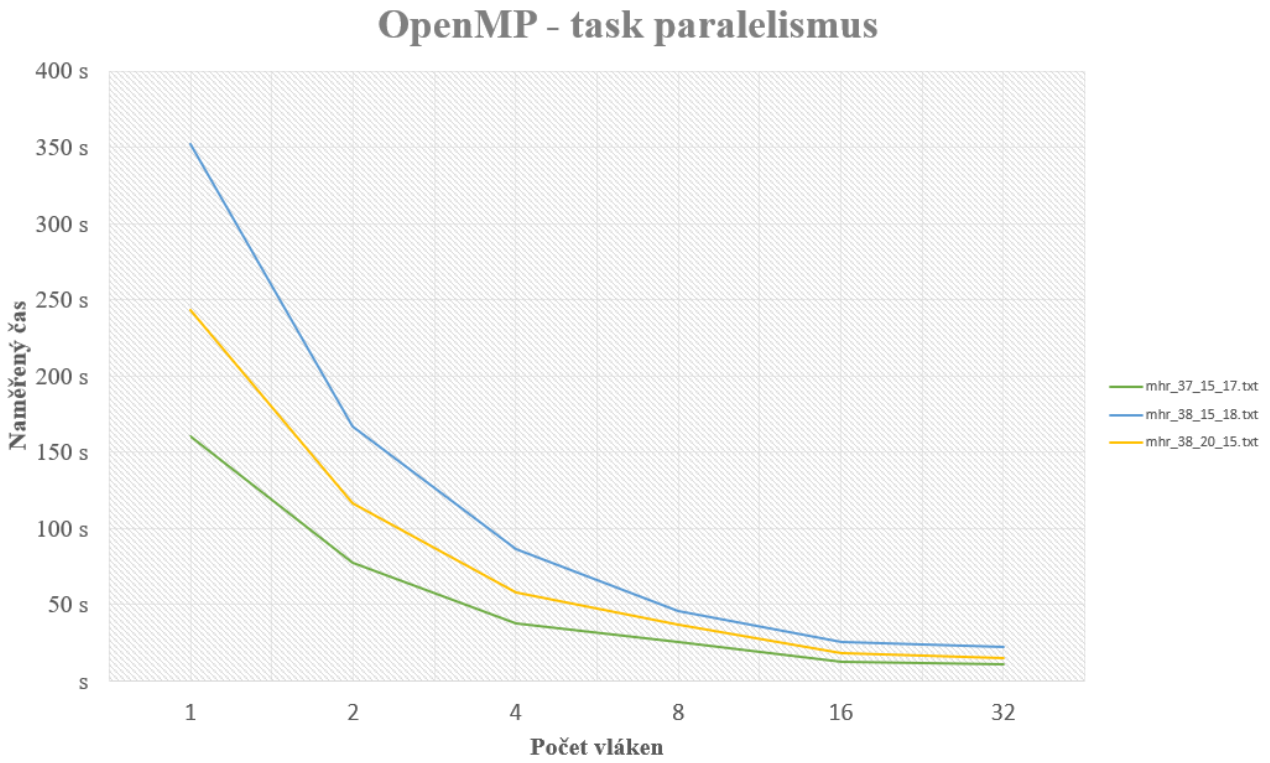
Následující tabulka obsahuje naměřené hodnoty pro soubor „mhr_38_15_18.txt“:

Počet vláken	Naměřený čas	Zrychlení	Efektivita
1	352,287s	0,951	0,951
2	167,17s	2,003	1,002
4	86,0805s	3,890	0,973
8	46,0313s	7,275	0,909
16	25,514s	13,124	0,820
32	22,1651s	15,107	0,472

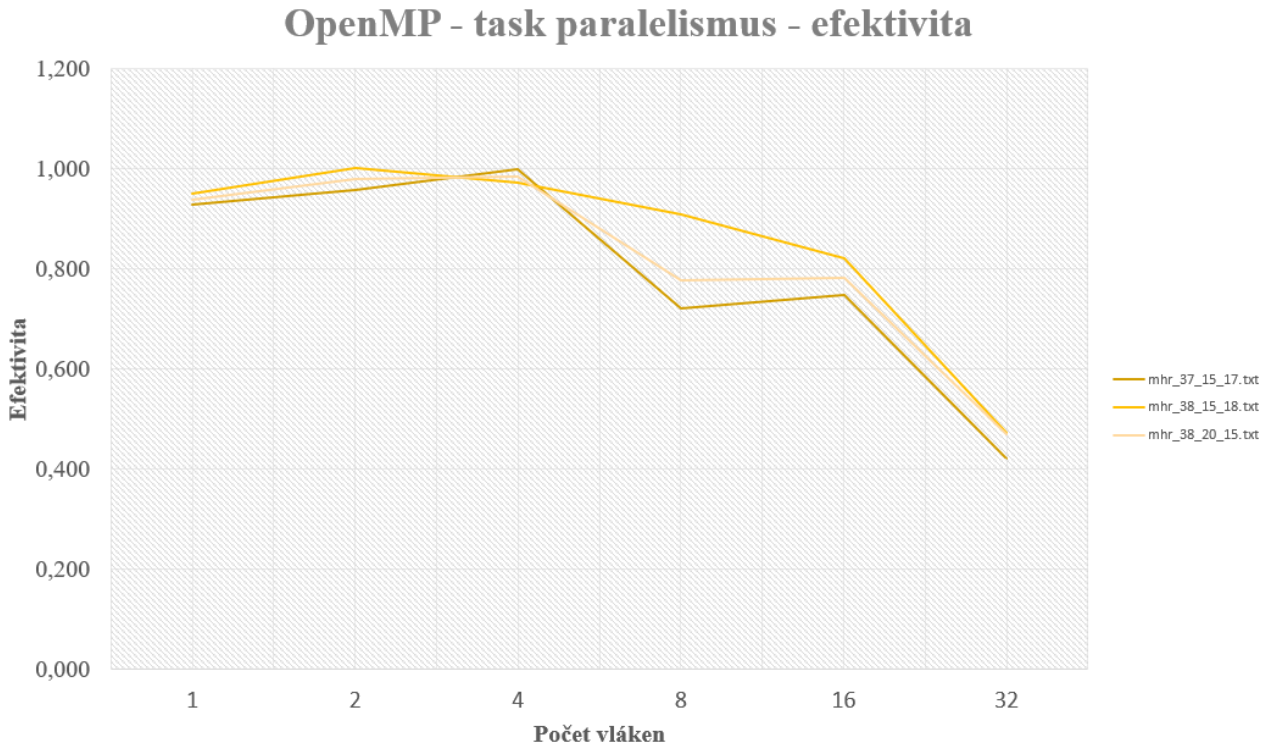
Následující tabulka obsahuje naměřené hodnoty pro soubor „mhr_38_20_15.txt“:

Počet vláken	Naměřený čas	Zrychlení	Efektivita
1	243,171s	0,938	0,938
2	116,546s	1,957	0,979
4	57,8624s	3,942	0,986
8	36,7412s	6,208	0,776
16	18,2058s	12,529	0,783
32	15,2233s	14,984	0,468

Graf níže znázorňuje zrychlení výpočtu jednotlivých instancí problému pro jednotlivé počty výpočetních vláken.



Následující graf znázorňuje efektivitu paralelního výpočtu u jednotlivých instancí problému pro jednotlivé počty výpočetních vláken.



5.4 OpenMP řešení - datový paralelismus

Následující tabulka obsahuje naměřené časy pro soubor „mhr_37_15_17.txt“:

Počet vláken	Naměřený čas	Zrychlení	Efektivita
1	154,4s	0,964	0,964
2	77,2827s	1,927	0,963
4	39,0135s	3,817	0,954
8	21,6281s	6,885	0,861
16	12,5573s	11,858	0,741
32	10,1305s	14,699	0,459

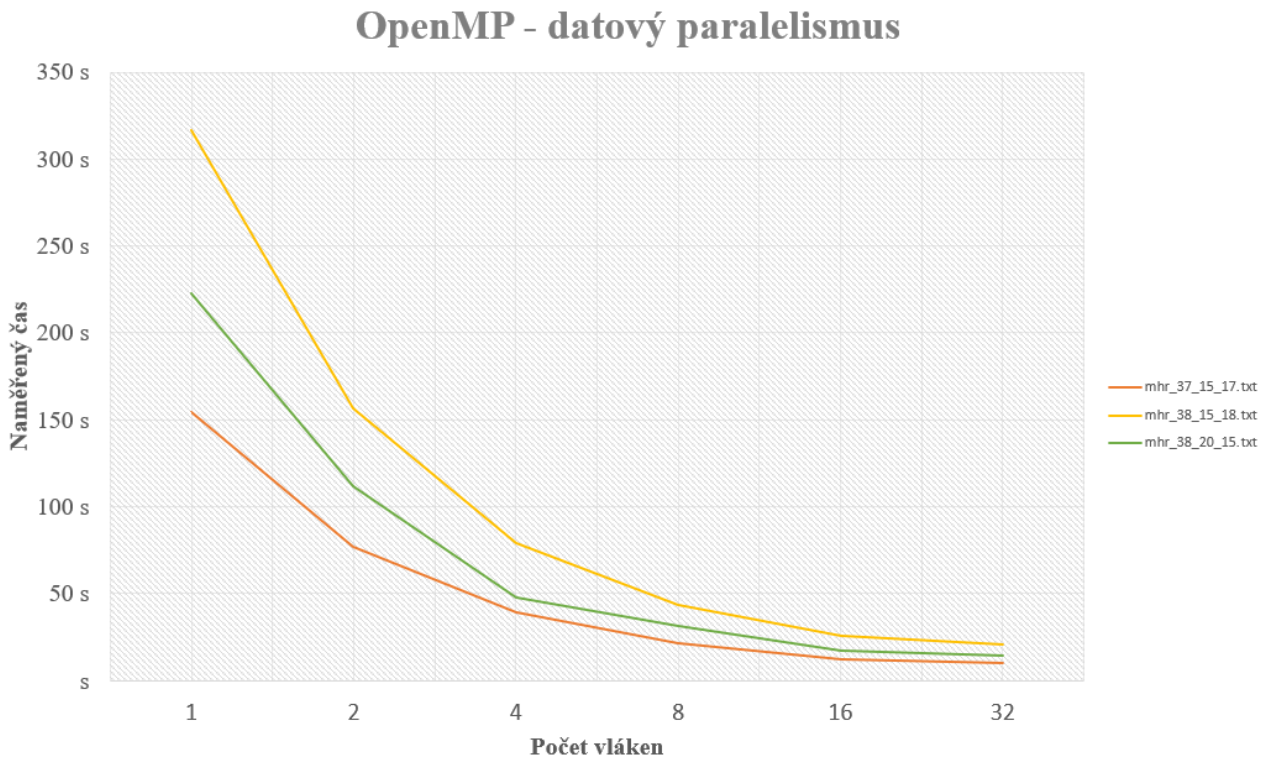
Následující tabulka obsahuje naměřené časy pro soubor „mhr_38_15_18.txt“:

Počet vláken	Naměřený čas	Zrychlení	Efektivita
1	316,323s	1,059	1,059
2	156,825s	2,135	1,068
4	79,3405s	4,221	1,055
8	43,883s	7,631	0,954
16	25,566s	13,098	0,819
32	20,5225s	16,317	0,510

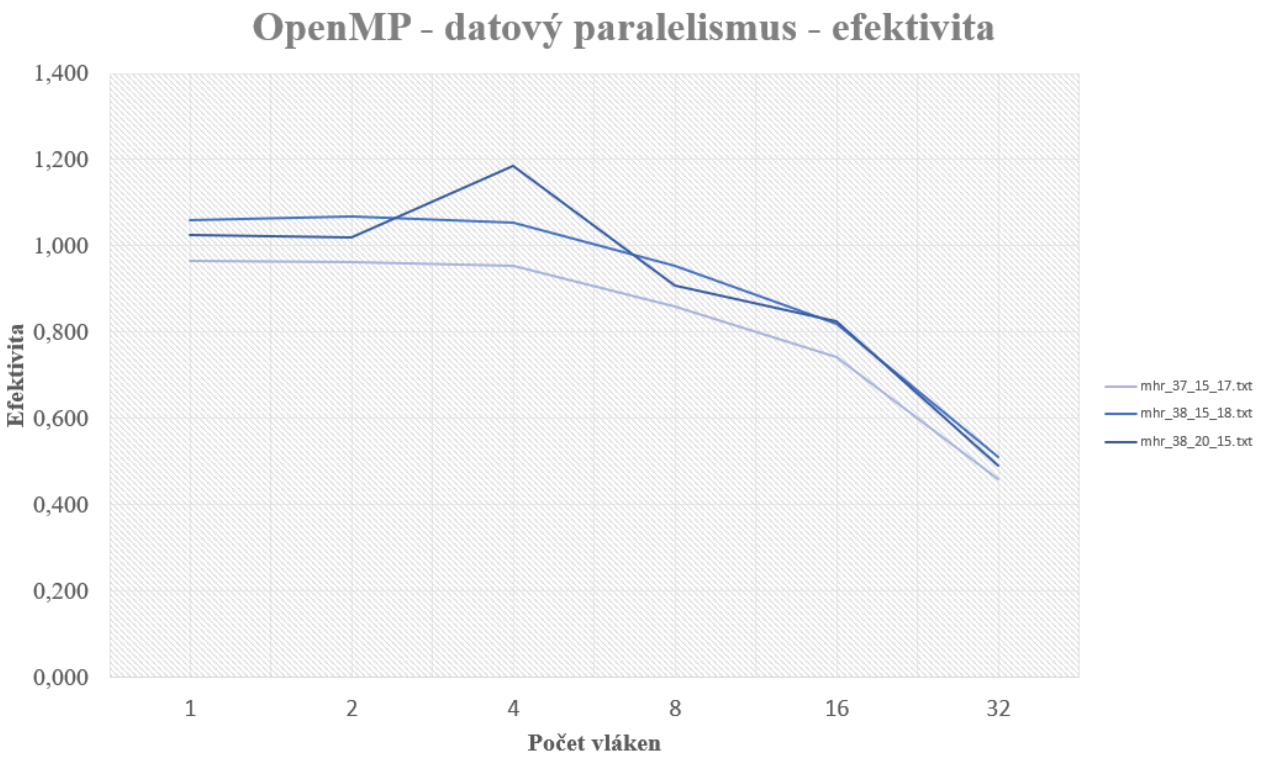
Následující tabulka obsahuje naměřené časy pro soubor „mhr_38_20_15.txt“:

Počet vláken	Naměřený čas	Zrychlení	Efektivita
1	222,581s	1,025	1,025
2	111,741s	2,041	1,021
4	48,0836s	4,744	1,186
8	31,3636s	7,273	0,909
16	17,2607s	13,215	0,826
32	14,5187s	15,711	0,491

Graf níže znázorňuje zrychlení výpočtu jednotlivých instancí problému pro jednotlivé počty výpočetních vláken.



Následující graf znázorňuje efektivitu paralelního výpočtu u jednotlivých instancí problému pro jednotlivé počty výpočetních vláken.



5.5 MPI řešení

Následující tabulka obsahuje naměřené časy pro soubor „mhr_37_15_17.txt“:

Počet vláken	Naměřený čas	Zrychlení	Efektivita
1	99,4959s	1,497	1,497
2	50,5023s	2,949	1,474
4	28,0085s	5,317	1,329
8	16,1332s	9,230	1,154
16	12,9762s	11,476	0,717
32	13,0327s	11,426	0,357

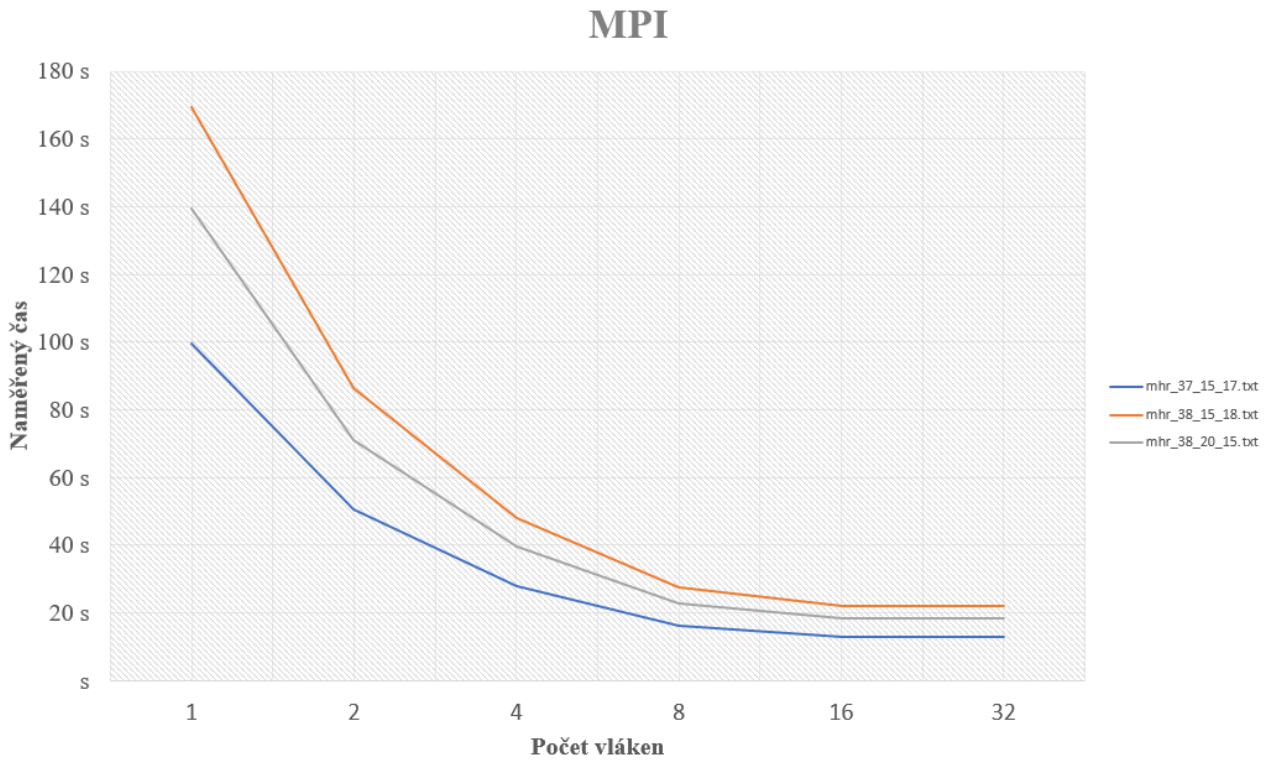
Následující tabulka obsahuje naměřené časy pro soubor „mhr_38_15_18.txt“:

Počet vláken	Naměřený čas	Zrychlení	Efektivita
1	169,494s	1,976	1,976
2	86,4336s	3,874	1,937
4	47,9184s	6,988	1,747
8	27,5716s	12,145	1,518
16	22,1723s	15,103	0,944
32	22,2093s	15,077	0,471

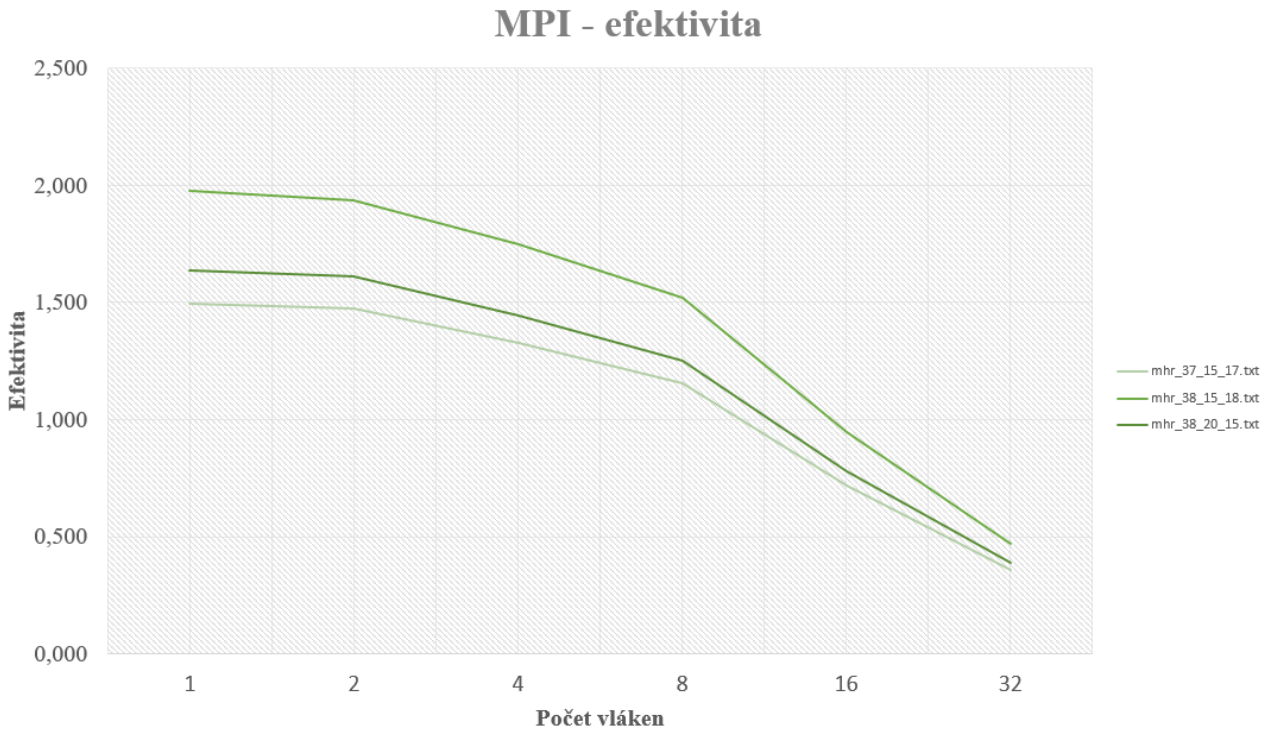
Následující tabulka obsahuje naměřené časy pro soubor „mhr_38_20_15.txt“:

Počet vláken	Naměřený čas	Zrychlení	Efektivita
1	139,421s	1,636	1,636
2	70,8683s	3,219	1,609
4	39,4595s	5,781	1,445
8	22,8227s	9,994	1,249
16	18,3458s	12,433	0,777
32	18,4017s	12,396	0,387

Graf níže znázorňuje zrychlení výpočtu jednotlivých instancí problému pro jednotlivé počty výpočetních vláken.

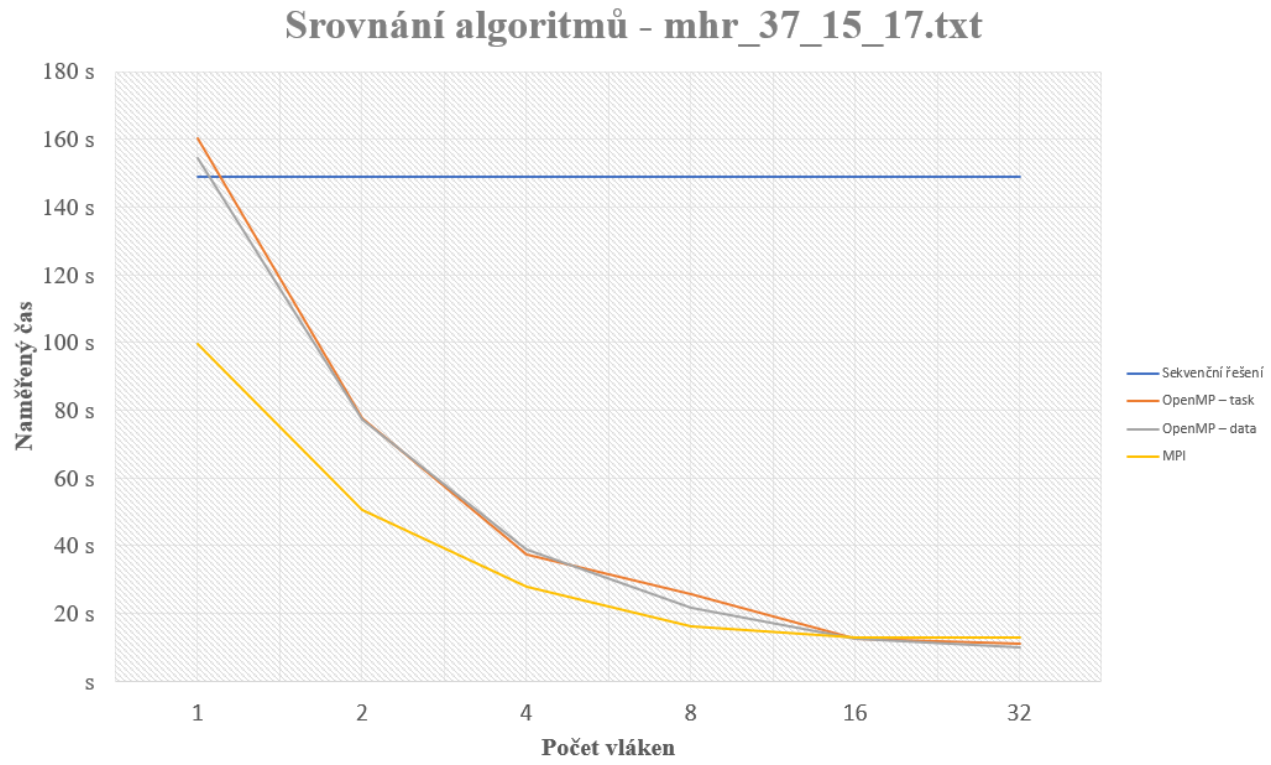


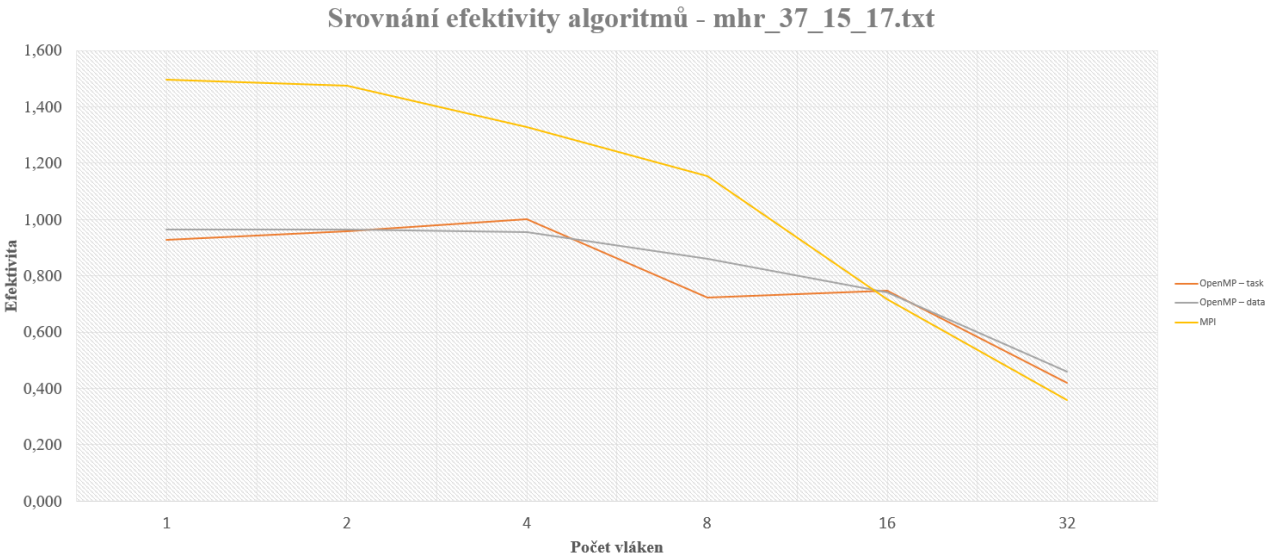
Následující graf znázorňuje efektivitu paralelního výpočtu u jednotlivých instancí problému pro jednotlivé počty výpočetních vláken.

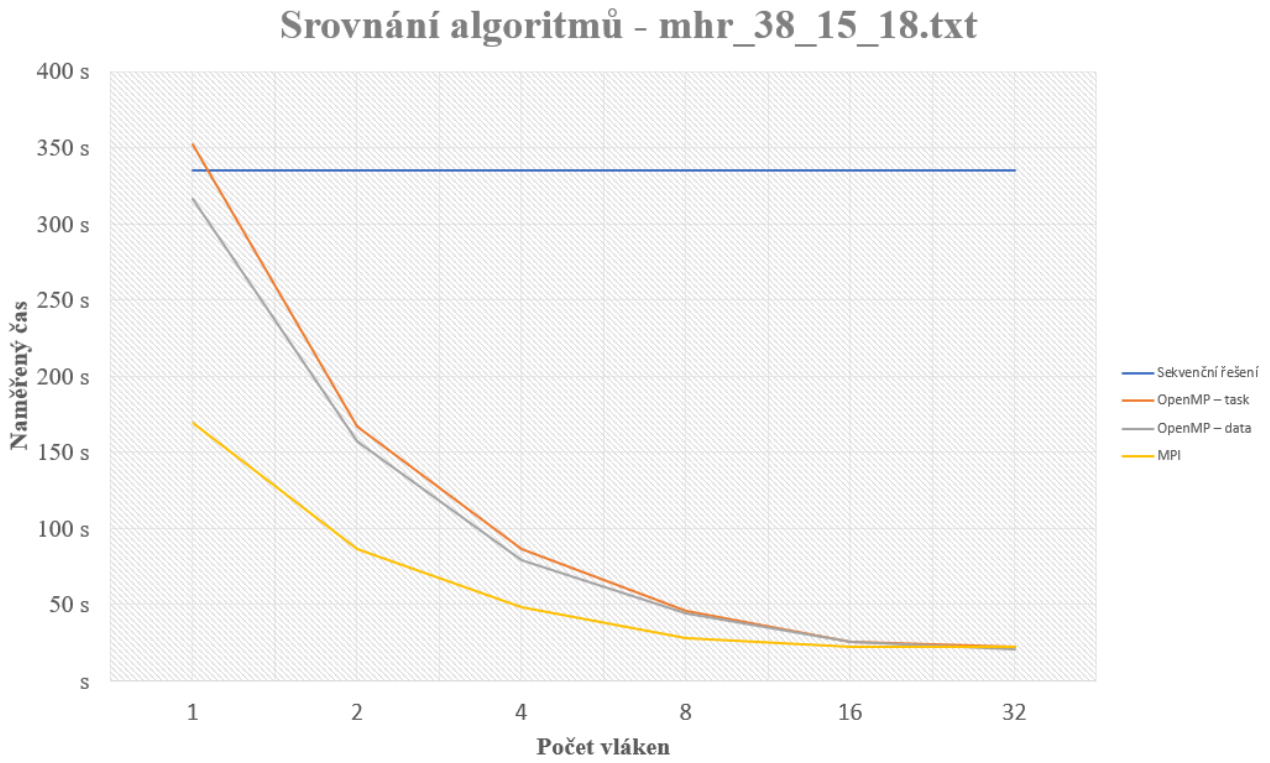


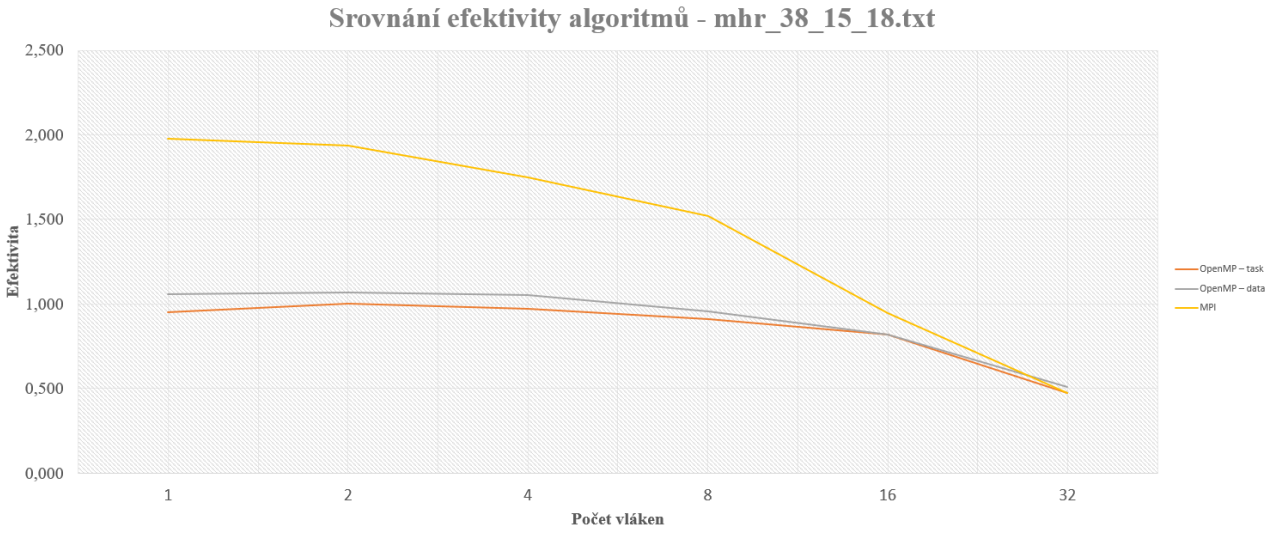
5.6 Grafické porovnání algoritmů

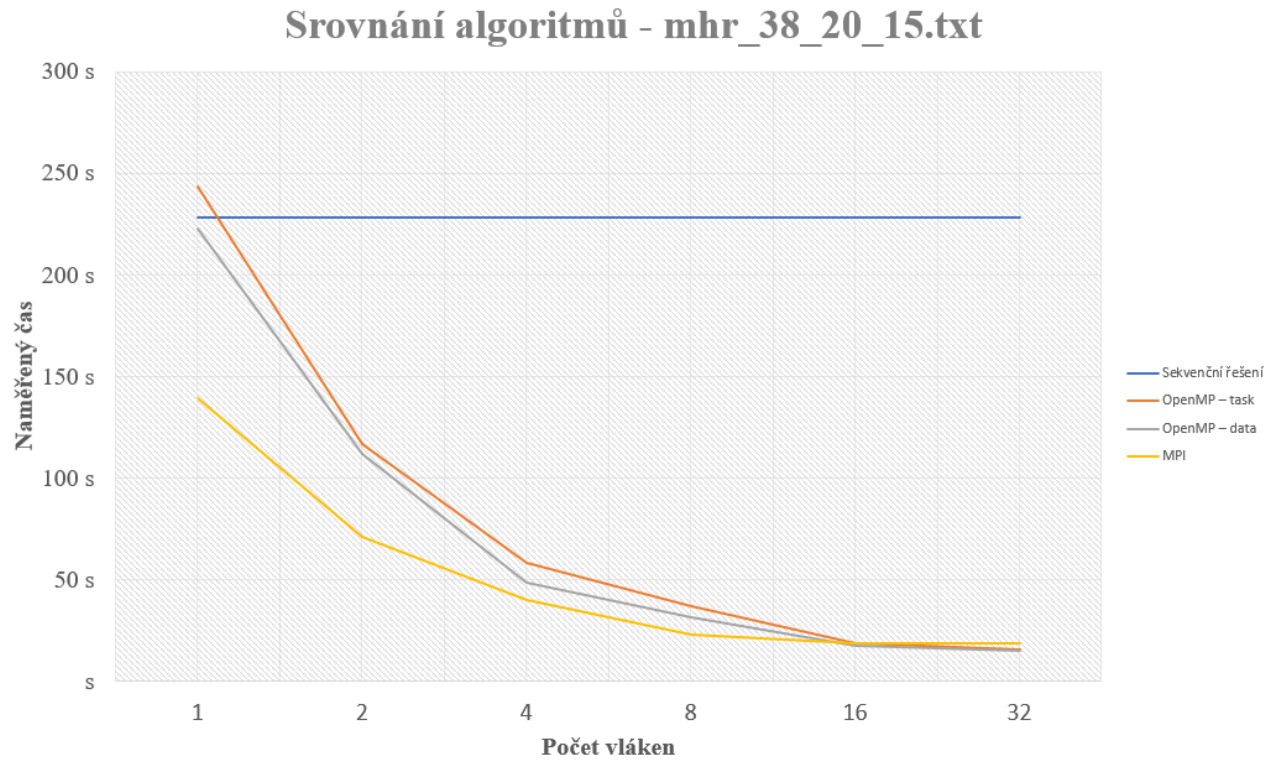
V této sekci jsou pomocí grafů znázorněna porovnání zrychlení a efektivity jednotlivých algoritmů.

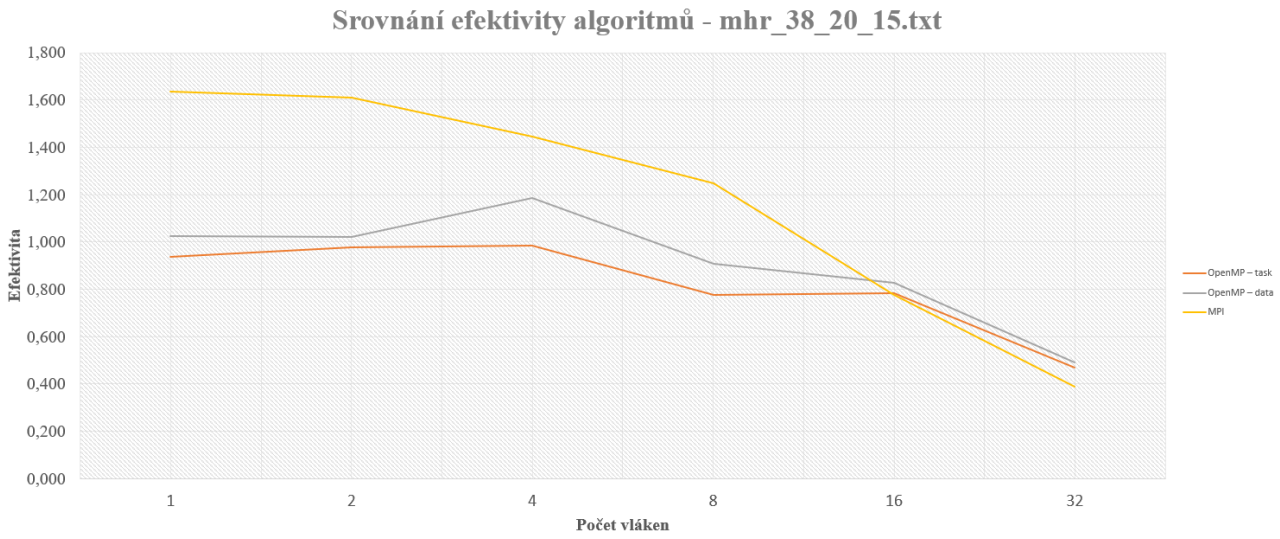

Srovnání algoritmů - mhr_37_15_17.txt
180 s
160 s
140 s
120 s
100 s
80 s
60 s
40 s
20 s
s
Naměřený čas
1
2
4
8
16
32
Počet vláken
Sekvenční řešení
OpenMP – task
OpenMP – data
MPI


Srovnání efektivity algoritmů - mhr_37_15_17.txt
1,600
1,400
1,200
1,000
0,800
0,600
0,400
0,200
0,000
Efektivita
1
2
4
8
16
32
Počet vláken
OpenMP – task
OpenMP – data
MPI


Srovnání algoritmů - mhr_38_15_18.txt
400 s
350 s
300 s
250 s
200 s
150 s
100 s
50 s
s
Naměřený čas
1
2
4
8
16
32
Počet vláken
Sekvenční řešení
OpenMP – task
OpenMP – data
MPI


Srovnání efektivity algoritmů - mhr_38_15_18.txt
2,500
2,000
1,500
1,000
0,500
0,000
Efektivita
1
2
4
8
16
32
Počet vláken
OpenMP – task
OpenMP – data
MPI


Srovnání algoritmů - mhr_38_20_15.txt
300 s
250 s
200 s
150 s
100 s
50 s
s
Naměřený čas
1
2
4
8
16
32
Počet vláken
Sekvenční řešení
OpenMP – task
OpenMP – data
MPI


Srovnání efektivity algoritmů - mhr_38_20_15.txt
1,800
1,600
1,400
1,200
1,000
0,800
0,600
0,400
0,200
0,000
Efektivita
1
2
4
8
16
32
Počet vláken
OpenMP – task
OpenMP – data
MPI

5.7 Analýza výsledků

Z naměřených hodnot můžeme vypozorovat, že v některých případech jsme dokázali zrychlit sekvenční řešení až zhruba 16 krát. Dále si můžeme všimnout, že zrychlení výpočtu při přechodu z 16 na 32 vláken je ve většině případů zanedbatelné. Důvodem je skutečnost, že na fakultním klastru star, na kterém probíhala všechna měření, bylo k dispozici pro výpočet pouze 20 vláken. Dá se tedy očekávat, že jakýkoliv vyšší počet se na zrychlení běhu algoritmu nijak neprojeví. Pokud bychom však měli k dispozici více vláken (případně více procesů), mohli bychom dosáhnout ještě lepších výsledků.

V případě OpenMP řešení jsme nejlepší hodnoty efektivity získali pro výpočty s 2 až 4 vlákny. V mnoha případech jsme dokonce dosáhli superlineáního urychlení. Pro MPI a obecně pro všechna měření platí, že efektivita s rostoucím počtem vláken ve většině případů klesá. Abychom zlepšili efektivitu, museli bychom zefektivnit paralelní výpočet, což už ale není moc možné.

Z grafů porovnání jednotlivých algoritmů je patrné, že MPI řešení dosahuje lepších hodnot zrychlení a efektivity než OpenMP řešení v případě, že má dostatek vláken pro výpočet. V opačném případě jsou výsledky srovnatelné, možná dokonce mírně lepší u OpenMP řešení.

6 Závěr

Díky této práci a tomuto předmětu jsem si vyzkoušel návrh a implementaci paralelních algoritmů týkající se prohledávání stavového prostoru. Knihovny OpenMP a MPI jsou uživatelsky velmi přívětivé a nabízí obrovské možnosti paralelizace jednotlivých algoritmů. Pomocí kombinace těchto knihoven lze získat nejlepší výsledky na libovolně velkých výpočetních klastrech, aniž bychom k tomu potřebovali sdílenou paměť.

Celkově byl pro mě obrovský přínos vyzkoušet si tyto knihovny pro paralelizaci algoritmů v praxi a zjistit, jakých výsledků tyto algoritmy při využití knihoven OpenMP a MPI dosahují. Jednotlivé výsledky měření jsou zobrazeny a popsány v předchozí kapitole.

7 Literatura

[1] Stránka předmětu MI-PDP.16 na Courses: MI-PDP Paralelní a distribuované programování [online]. duben 2020, [cit.2020-04-30]. Dostupné z: `https://courses.fit.cvut.cz/MI-PDP/index.html`

[2] Lísal, M.: Paralelní počítání. *Ústí nad labem: Univerzita Jana Evangelisty Purkyně, Přírodovědecká fakulta*, 2006. ISBN 80-7044-784-2.